CODE BRIGHT

DESARROLLO DE APLICACIONES WEB CON LA VERSION 4 DEL FRAMEWORK DE LARAVEL PARA PRINCIPIANTES

POR DAYLE REES
& ANTONIO LAGUNA

Laravel: Code Bright (ES)

Desarrollo de aplicaciones web con la versión 4 del framework Laravel para principiantes

Dayle Rees y Antonio Laguna

Este libro está a la venta en http://leanpub.com/codebright-es

Esta versión se publicó en 2013-08-28

¡Twitea sobre el libro!

Por favor ayuda a Dayle Rees y Antonio Laguna hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:

¡Acabo de comprar la edición en castellano de Laravel: Code Bright por @daylerees y @belelros ! http://leanpub.com/codebright-es

El hashtag sugerido para este libro es #codebright-es.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

https://twitter.com/search/#codebright-es

También por estos autores

Libros por Dayle Rees

Laravel: Code Happy

Laravel: Code Happy (ES)

Laravel: Code Happy (JP)

Laravel: Code Bright

Code Happy (ITA)

Laravel: Code Bright (SR)

Laravel: Code Bright (HU)

Laravel: Code Bright (IT)

Laravel: Code Bright (PT-BR)

Libros por Antonio Laguna

Laravel: Code Happy (ES)

Descubriendo Node.js y Express

Índice general

Reconocimiento

Antes que nada, me gustaría agradecer a mi novia Emma, no solo por animarme con todas mis aventuras, ¡si no también por hacer esas increíbles fotos a los pandas rojos para ambos libros! ¡Te amo Emma!

Taylor Otwell, el último año ha sido increíble, gracias por darme la oportunidad de ser parte del equipo, y por tu amistad. Gracias por hacer un framework que es un placer usar, hace que nuestro código se lea como la poesía y por poner tanto tiempo y pasión en su desarrollo. Realmente he disfrutado trabajar contigo en esta nueva versión de Laravel, ¡y espero trabajar contigo en futuros proyectos!

Eric Barnes, Phill Sparks, Shawn McCool, Jason Lewis, Ian Landsman, gracias por todo el apoyo con el framework y por ser buenos colegas.

Gracias a mis padres, ¡que han estado apoyando mis esfuerzos frikis durante casi 28 años! ¡También gracias por comprar un billón de copias del primer libro para la familia!

Gracias a todo el que haya comprado el primer libro, Code Happy, y a toda la comunidad de Laravel. Sin vuestro soporte, un segundo título nunca hubiera ocurrido.

Errata

Este puede ser mi segundo libro y mi escritura puede haber mejorado desde la última vez, pero te aseguro que habrá muchos, muchos errores. Puedes ayudarme a apoyar el libro enviándome un correo con cualquier error que encuentres a sombragriselros@gmail.com[1] junto con el título de la sección.

Los errores serán corregidos conforme vayan siendo descubiertos. Las correcciones serán lanzadas con ediciones futuras del libro.

[1] mailto:sombragriselros@gmail.com

Feedback

De la misma forma, puedes enviarme cualquier feedback que tengas sobre el contenido del libro o lo que quieras, enviando un correo a sombragriselros@gmail.com[2] o un tweet a @belelros. Me esforzaré en responder a todo correo que reciba.

[2] mailto:sombragriselros@gmail.com

Traducciones

Si quieres traducir Code Bright a tu idioma, por favor envíame un correo a me@daylerees.com[3] con tus intenciones. Ofreceré un 50/50 de los beneficios de la copia traducida, que tendrán el mismo precio que la copia en Inglés.

El libro está escrito en formato markdown.

[3] mailto:me@daylerees.com

Introducción

Bueno, está claro que hace mucho tiempo que escribí un capítulo de un libro. Code Happy hace casi 12 meses que fue lanzado y amasó un total de casi tres mil ventas. Veamos si aun me acuerdo de cómo se hace esto de "escribir".

Si has leído el título anterior ya sabrás que primero soy un programador, y segundo un escritor. Por este motivo no encontrarás largas palabras en este libro. Nada que impresione a Shakespeare (¿aparte de las faltas ortográficas?). Lo que **verás** es una charla directa y sencilla para aprender sobre el framework de Laravel. ¡También verás pasión! No del tipo de las sábanas mojadas, si no entusiasmo por el framework de Laravel con el cual no se puede rivalizar. Me gusta escribir mis libros como si estuviera frente a ti teniendo una conversación. De hecho, si realmente quieres una conversación ¡ven a verme al canal IRC de Laravel!

Ahora es el momento de uno de esos pequeños párrafos de 'sobre el autor'. Realmente nadie quiere leerlo, pero nunca hace daño subir un poco el ego, ¿verdad?

Mi nombre es Dayle Rees (¡eso dice la portada!) y soy un programador web y un entusiasta del diseño. Vengo de un pequeño pueblo de la costa de Gales llamado Aberystwyth. En el momento de escribir mi último libro 'Code Happy', trabajaba para la Biblioteca Nacional de Gales en Aberystwyth, que es una de las tres oficiales del Reino Unido.

Desde entonces, me he mudado a Cardiff, que es la capital de Gales y comencé a trabajar en BoxUK. BoxUK es una consultoría de internet y organización de desarrollo en la cual trabajo con un equipo de programadores que están apasionados sobre el mundo del desarrollo web.

El desarrollo web no es solo mi trabajo, también es mi hobby. Disfruto descubriendo interesantes piezas de código o diseños bonitos. Creo que nuestras habilidades nos permiten hacer cosas realmente maravillosas, y me encanta ver como las ideas cobran vida.

Hace poco más de un año, comencé a ayudar a la comunidad de Laravel con fragmentos de código, diseños web, y ayudando de cualquier forma que pude. Desde entonces mi implicación ha aumentado. Laravel es ahora mi principal proyecto de código libre y ahora soy parte del equipo de desarrollo central del framework.

Con Laravel 4 (nombre en clave Illuminate) mi implicación ha alcanzado un nuevo hito. He estado trabajando junto a Taylor Otwell para hacer que esta versión sea el mejor framework que jamás hayas usado. ¡No me tomes la palabra! Comienza a usarlo y agradécenoslo luego cuando no puedas dejar de sonreir mientras programas.

Laravel es un ejemplo de cómo incluso una herramienta de desarrollo puede ser creativa. La belleza en la sintaxis de Laravel que sale de la mente del genio Taylor Otwell, no puede ser rivalizada. Nos permite escribir código que se lee como poesía friki, y nos permite disfrutar de nuestras tareas de programación.

> Entonces… Dayle, ¿qué ha cambiado desde la última versiónd del framework?

La simple, aunque confusa respuesta ¡es que todo y nada!

Laravel 4 ha sido reescrito de la nada, permitiendo más flexibilidad y haciéndolo fácil de probar junto a un millón (no es totalmente prceiso - no las cuentes) de nuevas características. Donde Laravel 3 te daba algo de libertado sobre cómo estructurar tu código, Laravel 4 permitirá a los hackers volverse locos y cambiar el frameworks para ajustarse a sus necesidades.

Cuando escucho que algo ha mejorado siempre intento buscar la parte negativa, pero con Laravel 4 no hay ninguna. Sigue manteniendo la belleza y la expresividad en la sintaxis que amas; ¡puede que descubras incluso que te gusta más!

> Entonces, ¿por qué has escrito un nuevo libro?

Code Happy cubrió entre Laravel 3.0 y 3.2.x y con casi tres mil copias vendidas, algo debí haber hecho bien. Seguro que podría haber hecho el libro por completo para que funcionara con Laravel 4. No obstante, esta versión del framework es reinvención. SI fuera a actualizar el libro perderías toda la información sobre la versión 3 la cual creo que es aun un gran framework. Mucha gente tendrá grandes proyectos basados en Laravel 3 y creo que deberían tener acceso a la información de Code Happy si la necesitan.

Además, están mis propias experiencias. He aprendido un montón sobre escribir libros desde que acabé Code Happy. He aprendido sobre mis errores comunes, los cuales puedo evitar. Puedo mejorar lo que ya he hecho, y espero hacerlo.

> ¡No leí Code Happy! ¿Debería?

Podrías si quisieras, puse algunas bromas divertidas en él. No obstante, este libro es para principiantes y por tanto empezaremos desde la base. Si ya has estado usando Laravel sigue y léete las partes interesantes para ver qué ha cambiado. Si eres nuevo al framework, te sugeriría que te quedaras conmigo y leyeras de principio a fin. ¡No te preocupes! Intentaré mentenerlo interesante. Pronto estarás creando aplicaciones PHP increíblemente expresivas con Laravel.

> ¿Cuándo estará el libro completo?

Como con mi anterior título, este libro es un libro en progreso. Significa que obtendrás cada capítulo mientras lo escribo. En su estado actual, el libro puede no estar completo, pero conforme añada nuevos capítulos, recibirás un nuevo correo y podrás descargar las actualizaciones gratuitamente.

Creo que este método de escritura ofrece una gran flexibilidad. Puedo relajarme con la escritura sabiendo que puedo cambiar algo fácilmente si lo tengo mal. No teniendo que preocuparme de una fecha límite, puedo escribir un libro que sienta que será de mejor cualidad. Puedo actualizar el título para versiones futuras, o destacar información adicional. Podrás acceder al contenido más rápidamente. También me permite lanzar el título junto a las nuevas versiones del framework.

Me he quedado sin preguntas

¡Bien! Ahora deberías estar deseando empezar a aprender. Únete a mi y comienza a disfrutar de Laravel. ¡No dudes en mandarme un tweet o un mensaje en IRC si quieres hablar!

Manual básico

> Ey, ¿¡este capítulo no estaba en Code Happy!?

Como un verdadero fan de Code Happy, ¡no se te escapa nada! ¡Bien hecho, leal lector!

Como ves, Laravel 4 usa nuevas tecnologías regularmente. Estas tecnologías pueden ser tomadas fácilmente por separado y en paralelo framework. Con esto en mente, creo que sería mejor empezar con un capítulo sobre nuevas tecnologías para ayudarte en tu experiencia de aprendizaje.

Los programadores web experimentados puede que ya hayan visto estas tecnologías, o que las estén usando. Eres libre de saltarte cualquiera de estas secciones seccione si quieres No me enfadaré. No de verdad… vamos. Ya no me necesitas…

Si aun estás leyendo, asumiré que eres mi amigo. ¡No como esos traidores que saltaron directamente al capítulo de las rutas!

Saltemos a nuestra primera lección para hablar sobre los espacios de nombres.

Espacios de nombres

En la versión 5.3 de PHP se añadió una nueva característica al lenguaje conocida como espacios de nombres. Muchos lenguajes modernos ya tenían esto desde hace tiempo, pero PHP llegaba un poco tarde a la fiesta. Sin embargo, cada nueva característica tiene un propósito. Veamos porqué los espacios de nombre pueden beneficiar nuestra aplicación.

En PHP no puedes tener dos clases que compartan el mismo nombre. Tienen que ser únicos. El problema con esta restricción es que si estás usando una librería de terceros que tiene una clase llamada `Usuario`, significa que no puedes crear tu propia clase que se llame `Usuario`. Esto es una verdadera vergüenza, porque es un nombre muy conveniente para una clase ¿verdad?

Los espacios de nombre de PHP nos permiten sortear este problema, de hecho podemos tener tantas clases `Usuario` como queramos. No solo eso, podemos usar espacios de nombre para contener código similar en pequeños y bonitos paquetes, o incluso para mostrar autoría.

Echemos un vistazo a una clase normal. Sí… Sé que las has usado antes. ¿Confía en mi en esto vale?

Let's jump right in to our first primer to talk about PHP namespaces.

Espacio de nombre global

He aquí una sencillísima clase.

```php
<?php

// app/models/Eddard.php

class Eddard
{

}
```

No hay nada especial en ella, si queremos usarla podemos hacer esto.

```php
<?php

// app/routes.php

$eddard = new Eddard();
```

> Dayle, sé algo de PHP.

Vale, vale lo siento. Básicamente, podemos pensar en estas clases como pertenecientes al espacio de nombre global. No sé si este es el nombre correcto para ello, pero me suena bastante lógico. Es tan solo una clase normal.

Espacio de nombre simple

Creemos otra clase junto a la original y global `Eddard`.

```php
<?php

namespace Stark;

// app/models/another.php

class Eddard
{

}
```

Aquí tenemos otra clase Eddard, con un pequeño cambio. Hemos añadido la directiva `namespace`. Esta línea `namespace Stark;` informa a PHP que todo lo que hacemos pertenece al espacio de nombre `Stark`. También significa que cualquier clase creada dentro de este archivo vivirá dentro del espacio de nombre `Stark`.

Ahora, cuando intentamos usar la clase `Eddard` una vez más.

```php
<?php

// app/routes.php

$eddard = new Eddard();
```

Una vez más, obtenemos una instancia de la primera clase que creamos en la primera sección. No la que hemos creado en el espacio de nombre `Stark`. Creemos una instancia de la clase `Eddard` del espacio de nombre `Stark`.

```php
<?php

// app/routes.php

$eddard = new Stark\Eddard();
```

Podemos instanciar una clase dentro de un espacio de nombre, añadiéndole como prefijo el nombre del espacio de nombre, y separándola con una barra invertida (\). Ahora tenemos una clase `Eddard` de dentro del espacio de nombre `Stark`. ¡¿No somos mágicos?!

Deberías saber que los espacios de nombre pueden tener tantos nombres de jerarquía como necesites. Por ejemplo:

```
Esta\Combinacion\De\Espacio\De\Nombre\Es\Estupida\Pero\Funciona
```

La teoría de la relatividad

Recuerda como te dije que PHP siempre reacciona de manera **relativa** al espacio de nombre actual. Bueno, veamos eso en acción.

```php
<?php

namespace Stark;

// app/routes.php

$eddard = new Eddard();
```

Añadiendo la directiva `namespace` al ejemplo hemos movido la ejecución del script PHP al espacio de nombre `Stark`. Ahora, porque estamos dentro del mismo espacio de nombre en el cual pusimos a `Eddard`, esta vez hemos recibido la clase `Eddard` de dentro. ¿Ves como todo es relativo?

Ahora que hemos cambiado el espacio de nombre, hemos creado un pequeño problema. ¿Puedes imaginarte cuál es? ¿Cómo podemos ahora instanciar la clase `Eddard` original? Aquella que no estaba en el espacio de nombre.

Por suerte, PHP tiene un truco para referirse a clases que están ubicadas en el espacio de nombre global. Simplemente añádeles una barra invertida (\) al inicio.

```php
<?php

// app/routes.php

$eddard = new \Eddard();
```

Con la barra invertida (\) inicial, PHP sabe que nos estamos refiriendo al `Eddard` del espacio de nombre global y e instanciará esa.

Usa tu imaginación un poco, como Barney te enseñó. Imagina que tenemos otra clase en un espacio de nombre llamada `Tully\Edmure`. Ahora queremos usar esta clase desde el framework `Stark`. ¿Cómo lo hacemos?

```php
<?php

namespace Stark;

// app/routes.php

$edmure = new \Tully\Edmure();
```

De nuevo, tenemos que prefijar con la barra invertida para volver al espacio de nombre global, antes de instanciar una clase del espacio de nombre `Tully`.

Puede ser cansado, referirse a clases con otros espacios de nombres por su jerarquía completa cada vez. Por suerte, hay un pequeño atajo que podemos usar. Veámoslo en acción.

```php
<?php

namespace Stark;

use Tully\Edmure;

// app/routes.php

$edmure = new Edmure();
```

Usando la sentencia `use`, podemos traer una clase de otro espacio de nombre. Permitiéndonos instanciarla solo por el nombre. Ahora no me preguntes porqué no necesita la barra invertida, porque no lo sé. Esta es la única excepción que conozco. ¡Lo siento! Aunque se la puedes añadir si quieres, solo que no necesitas hacerlo.

Para compensar la horrible inconsistencia, permíteme enseñarte otro bonito truco. Podemos darles a nuestras clases importadas pequeños apodos, como hacíamos en la guardería de PHP. Permíteme mostrártelo.

```php
<?php

namespace Stark;

use Tully\Brynden as PezNegro;

// app/routes.php

$edmure = new PezNegro();
```

Usando la palabra clave `as`, le hemoms dado a nuestra clase 'Tully/Brynden' el apodo de `PezNegro` para identificarla en el espacio de nombre actual. Bonito truco, ¿verdad? También es útil si necesitas usar dos clases que se llamen de forma similar en el mismo espacio de nombre, por ejemplo:

```php
<?php

namespace Targaryen;

use Dothraki\Daenerys as Khaleesi;

// app/routes.php

class Daenerys
{

}

// Targaryen\Daenerys
$daenerys = new Daenerys();

// Dothraki\Daenerys
$khaleesi = new Khaleesi();
```

Dándalo a `Daenerys` del espacio de nombre `Dothraki` el apodo de `Khaleesi`, podemos usar dos clases `Daenerys` únicamente por el nombre. Útil, ¿no? El juego trata sobre evitar conflictos y agrupar cosas por propósito o facción.

Puedes usar `use` tantas clases como necesites.

```
<?php

namespace Targaryen;

use Dothraki\Daenerys;
use Stark\Eddard;
use Lannister\Tyrion;
use Snow\Jon as Bastardo;
```

Estructura

Los espacios de nombre no son solo para evitar conflictos, también podemos usarlos para organizarlos y para autoría. Permíteme explicártelo con otro ejemplo.

Digamos que quiero crear una librería de código libre. Me encantaría que otros usaran mi código, ¡sería genial! El problema es que no quiero provocar ningún conflicto de nombre de clase problemático para la persona que use mi código. Eso sería terriblemente inconveniente. He aquí cómo puedo evitar causar problemas.

```
Dayle\Blog\Content\Post
Dayle\Blog\Content\Page
Dayle\Blog\Tag
```

Aquí hemos usado mi nombre, para mostrarte que he creado el código origina, y para separar mi código del de la otra persona usando mi librería. Dentro del espacio de nombre base, he creado un número de sub-espacios de nombre para organizar mi aplicación por su estructura interna.

En la sección de Composer aprenderás sobre cómo usar espacios de nombre para simplificar el acto de cargar definiciones de clases. Te recomiendo enormemente que eches un vistazo a este útil mecanismo.

Limitaciones

En realidad, me siento un poco culpable por llamar a este apartado “Limitaciones”. De lo que voy a hablarte no es realmente un fallo.

Como verás, en otros lenguajes, los espacios de nombre están implementados de manera similar, y otros lenguajes ofrecen algo adicional al interactuar con los espacios de nombres.

En Java, por ejemplo, puedes importar un número de clases al espacio de nombre actual usando la sentencia `import` con un asterisco. En Java, `import` es el equivalente de `use` y usa puntos para separar los espacios de nombre anidados (o paquetes). He aquí un ejemplo.

```
import dayle.blog.*;
```

Esto importaría todas las clases que estén ubicadas dentro del paquete `dayle.blog`.

En PHP no puedes hacer esto. Tienes que importar cada clase de manera individual. Lo siento. De hecho, ¿por qué digo lo siento? Ve a quejarte al equipo de PHP, pero, hazlo con delicadeza. Nos han dado muchas cosas chulas últimamente.

No obstante, he aquí un bonito truco que puedes usar. Imagina que tienes este espacio de nombre y estructura de clase, como en el anterior ejemplo.

```
Dayle\Blog\Content\Post
Dayle\Blog\Content\Page
Dayle\Blog\Tag
```

Podemos darle a un sub-espacio de nombre un apodo, para usar sus clases hijas. He aquí un ejemplo:

```
<?php

namespace Baratheon;

use Dayle\Blog as Cms;

// app/routes.php

$post = new Cms\Content\Post;
$page = new Cms\Content\Page;
$tag  = new Cms\Tag;
```

Esto debería ser útil si necesitas usar muchas clases dentro del mismo espacio de nombre. ¡Disfruta!

Ahora aprenderemos sobre Yeison. No, no sobre Jason Lewis el australiano, pero sobre las cadenas JSON. ¡Pasa de página y verás lo que quiero decir!

JSON

¿Qué es JSON?

JSON significa JavaScript Object Notation o *Notación de Objetos de JavaScript*. Fue nombrado de esta forma porque JavaScript fue el primer lenguaje en sacar provecho de este formato.

Esencialmente, JSON es un método legible para humanos de almacenar matrices y objetos con valores y cadenas. Se usa principalmente para transferir datos, y es mucho menos verboso que otras opciones como XML.

Comúnmente, es usado cuando la parte front-end de tu aplicación requiere algo de datos del back-end sin recargar la página. Normalmente esto se hace usando JavaScript con una petición AJAX.

Muchas APIs de software también sirven contenido usando este formato. Twitter posee un buen ejemplo de este tipo de APIs.

Desde la versión 5.2.0, PHP puede serializar objetos y matrices en JSON. Esto es algo de lo que personalmente he abusado un billón o así de veces y fue un gran añadido al lenguaje.

Si has trabajado con PHP desde hace tiempo, puede que hayas usado ya el método `serialize()` para transformar una cadena en una nueva instancia que contenga el valor original.

Básicamente es para eso para lo que usaremos JSON. No obstante, a ventaja es que JSON puede ser parseado por varios lenguajes, mientras que las cadenas serializadas solo pueden ser parseadas por PHP. La ventaja adicional es que nosotros (como humanos, y pandas) podemos leer cadenas JSON, pero las cadenas serializadas de PHP se ven fatal.

Ya basta de historia, metámonos en faena y echemos un ojo a algo de JSON.

Enough back story, let's dive right in and have a look at some JSON.

JSON Syntax

```
1 {"nombre":"Lushui","especie":"Panda","dieta":"Cosas verdes","edad":7,"colores":["\
2 red","marron","blanco"]}
```

¡Bien JSON! Vale, cuando dije que era legible para humanos puede que olvidara mencionar algo. Por defecto, JSON se guarda sin espacios entre sus valores lo cual lo puede hacer un poco más difícil de leer.

Esto se hace normalmente para ahorrar ancho de banda al transferir los datos, sin los espacios en blanco adicionales, la cadena JSON será mucho más corta y por tanto habrá menos bytes que transferir.

Las buenas noticias son que JSON no se inmuta con los espacios en blancos o saltos de línea entre sus claves y valores. ¡Vuélvete loco! Pon todos los espacios que quieras para hacerlo más legible.

Podríamos hacerlo a mano (no lo hagamos), pero hay muchas herramientas en internet para embellecer JSON. No voy a escoger una por ti. ¡Ve de caza! Puede que incluso encuentres una extensión para tu navegador que te permita leer respuestas JSON desde servidores web de forma fácil. ¡Te recomiendo que encuentres una de ellas!

Vamos a añadir algunos espacios en banco al JSON para hacerlo más fácil de leer. (Con sinceridad, esto lo hice a mano. No intentéis esto en casa amigos.)

```
{
    "nombre":       "Lushui",
    "especies":     "Panda",
    "dieta":        "Cosas verdes",
    "edad":         7,
    "colores":      ["rojo", "marrón", "blanco"]
}
```

¡Aha! Ahí vamos. Ahora tenemos una cadena JSON que representa al panda rojo que vive en la portada de mi libro. Lushui protege el conocimiento de Laravel de los fisgones.

Como puedes ver por e ejemplo, tenemos un número de parejas clave-valor. Siendo una de las parejas clave-valor una matriz. Con sinceridad, si alguna vez has usado JavaScript antes puede que te preguntes, ¿qué ha cambiado aquí? De hecho, echemos un vistazo a cómo se representaría en JavaScript.

```
var lushui = {
    nombre:      'Lushui',
    especies:    'Panda',
    dieta:       'Cosas verdes',
    edad:        7,
    colores:     ['rojo', 'marrón', 'blanco']
};
```

Con suerte, te habrás fijado en las similaridades entre los fragmentos de JavaScript y de JSON.

El fragmento de JavaScript está asignando un objeto literal a una variable, tal que así:

```
var lushui = { .. };
```

Bueno, JSON es un formato de transferencia de dato y no un lenguaje. No tiene el concepto de variables. Es esto por lo que no necesitas la asignación dentro del fragmento de JSON.

EL método de representar un objeto literal es muy similar. ¡No es ninguna coincidencia! Como dije antes, JSON se inventó originalmente para usarlo con JavaScript.

Tanto en el objeto JSON como en el de JavaScript, los objetos están contenidos entre { dos llaves } y consisten en parejas clave-valor de datos. En la variante de JavaScript no necesitan las comillas porque representan variables, pero acabamos de leer que JSON no tiene variables. Esto es correcto porque podemos usar cadenas como claves y eso es exactamente lo que hace JSON para solucionar este problema

Puede que te hayas dado cuenta de que usé comillas simples alrededor de los valores de JavaScript. ¡Es una trampa! Este comportamiento es perfectamente aceptable en JavaScript, pero las cadenas JSON **deben** estar contenidas entre comillas dobles. ¡Recordarlo tu debes joven padawan!

Tanto en JavaScript como en JSON, las parejas clave y valor deben estar separadas con dos puntos (`:`), y las parejas de clave-valor deben estar separadas con una coma (`,`).

JSON soporta cadenas y valores numéricos. Puedes ver el valor de la edad de Lushui es el de un entero con el valor de siete.

```
edad: 7,
```

JSON permite estos tipos de valores:

- Doble
- Flotante
- Cadena
- Booleano
- Matriz
- Objeto
- Null

Los valores numéricos se representan sin comillas dobles. Ten cuidado al escoger si entrecomillas un valor o no. Por ejemplo, los códigos postales de Estados Unidos consisten en cinco números. No obstante, si omitieras las comillas de un código postal, `07702` sería tomado como un entero y sería truncado a `7702`. Este es un error que se ha cobrado la vida de muchos aventureros de la web.

Los booleanos son representados por las palabras `true` y `false`, ambos sin comillas al igual que en PHP. Como dije antes, los valores de cadenas quedan dentro de comillas dobles **y no comillas simples**.

El valor null funciona de forma similar a la que lo hace en PHP, y es representado por la palabra `null` sin comillas. ¡Debería ser fácil de recordar!

Hemos visto objetos. Al igual que el objeto JSON principal en sí, están envueltos por llaves y pueden contener toda variedad de tipos de valores.

Las matrices, lucen de manera muy similar a su contrapartida en JavaScript

```
// JavaScript
['rojo', 'marron', 'blanco']


["rojo", "marron", "blanco"]
```

*__Nota__ Te habrás dado cuenta de que no añadí un comentario en línea para el fragmento JSON de arriba. Esto es porque JSON no soporta comentarios ya que es usado para transferir datos. ¡Tenlo en cuenta!

Como puedes ver, las marices para ambos están encapsuladas por `[` corchetes `]`, y con tienen una lista de valores separadas por coma (`,`) sin ningún índice. Una vez más, la única diferencia es que las dobles comillas han de ser usadas para las cadenas en JSON. ¿Ya te estás cansando de que diga eso?

Como mencioné, los valores que pueden ser contenidos en JSON incluyen tanto objetos como matrices. Los más listos de mis lectores (es decir, todos vosotros) pueden haberse dado cuenta de que JSON puede soportar objetos y matrices anidados. ¡Echémosle un vistazo a eso en acción!

```
{
    "un_objeto": {
        "una_matriz_de_objetos": [
            { "El": "secreto" },
            { "es": "que" },
            { "me": "siguen" },
            { "gustando": "los zapatos!" }
        ]
    }
}
```

Vale. Respira hondo. Aquí tenemos un objeto JSON que contiene una matriz de objetos. Esto está perfectamente bien, y permitirá a JSON representar algunas colecciones de datos complejas

JSON y PHP

Como mencioné antes, desde la versión 5.2.0, PHP ofrece soporte para serializar y deserializar datos de y hacia el formato JSON. Vamos a ver cómo funciona eso en acción.

Serializando una matriz PHP a JSON

Para serializar un valor PHP solo necesitamos el método `json_encode()`. Tal que así:

```
<?php

$verdad = array('panda' => '¡Increible!');
echo json_encode($verdad);
```

El resultado de este fragmento de código será una cadena JSON que contenga el siguiente valor.

```
{"panda":"¡Increible!"}
```

¡Bien! Vamos a asegurarnos de que podemos convertir esto de vuelta al formato que pueda ser entendido por PHP.

Deserializando una matriz PHP desde una cadena JSON

Para ello usaremos el método `json_decode`. Apuesto a que no lo viste venir.

```php
<?php

$verdad = json_decode('{"panda":"¡Increible!"}');
echo $verdad['panda'];
```

¡Genial! Allá vam... espera, ¿qué?

```
Fatal error: Cannot use object of type stdClass as array in ...
```

Como ves, `json_decode` no devuelve nuestro JSON como matriz PHP; usa un objeto del tipo `stdClass` para representar nuestros datos. En vez de ello vamos a acceder a la clave del objeto como un atributo:

```php
<?php

$verdad = json_decode('{"panda":"¡Increible!"}');
echo $verdad->panda;

// ¡Increible!
```

¡Genial! Eso es lo que queríamos. Si queríamos una matriz, PHP nos ofrece algunas formas de convertir este objeto en una, pero por suerte, `json_decode` aun tiene un truco bajo la manga.

Si usas `true` como segundo parámetro a la función, recibiremos la matriz PHP como esperábamos. ¡Gracias `json_decode`!

```php
<?php

$verdad = json_decode('{"panda":"¡Increible!”}', true);
echo $verdad['panda'];

// ¡Increible!
```

¡Hurrah!

Puede que te estés preguntando porqué he escrito un capítulo enorme sobre JSON en un libro de Laravel. Es más, puede que te estés preguntando porqué intento responder a esta pregunta al final del capítulo.

Es solo porque así es más divertido.

En la siguiente sección veremos con detalle el nuevo gestor de paquetes para PHP. Cuando comencemos a ver Composer comprenderás porque es tan importante conocimiento sobre JSON.

Composer

Composer es algo especial en el mundo de PHP. Ha cambiado la forma en la que gestionamos las dependencias de la aplicación y ha sofocado las lágrimas de muchos programadores PHP.

Como sabrás, en los días antiguos, cuando querías crear una aplicación que dependía de dependencias de terceros, tenías que instalarlas con PEAR o PECL. Estos dos gestores de dependencias tienen un conjunto de dependencias muy limitado y desactualizado y han sido una espina en la espinilla de los programadores PHP durante mucho tiempo.

Cuando un paquete está finalmente disponible puedes bajarte una versión específica y será instalada en tu sistema. No obstante, esta dependencia está enlazada con PHP en vez de con tu aplicación. Esto significa que si tienes dos aplicaciones que necesitan versiones distintas de las mismas dependencias… bueno, vas a pasarlo mal.

Entra Composer, el rey de los gestores de dependencia. Primero pensemos sobre los paquetes, ¿qué son?

Antes que nada, olvidémonos del concepto de 'aplicaciones' y 'proyectos' por ahora. La herramienta que estás construyendo se llama paquete. Imagina una pequeña caja que contiene todo lo necesario para ejecutar tu aplicación, y descríbela.

Esta caja solo requiere un pequeño trozo de papel (archivo) dentro para que sea registrada como paquete.

Configuración

Has aprendido sobre JSON en el último capítulo, ¿verdad? ¡Ahora estás preparado para esto! ¿Recuerdas que te dije que JSON se usaba para transferir datos entre aplicaciones web? Bueno, te mentí. No es que sea malo, solo hice que fuera fácil enseñarte sobre el tema dándote un pequeño trozo de su habilidad. Lo hago mucho, ¡espera muchas mentiras!

¿Recuerdas cómo JSON representa datos complejos? Bueno, entonces, ¿por qué no podemos usarlo en archivos para ofrecer configuración? Eso es exactamente lo que pensaron los chicos de Composer. ¿Quiénes somos nosotros para discutir con ellos?

Los archivos JSON usan la extensión `.json`. Composer espera que su configuración esté en la raíz de tu paquete con el nombre `composer.json`. ¡Recuérdalo! Laravel usa este archivo muy a menudo.

Abrámoslo y comencemos a meter algo de información sobre nuestro paquete.

```
{
    "name":             "marvel/xmen",
    "description":      "Mutantes que salvan al mundo de gente que lo odia.",
    "keywords":         ["mutante", "super heroe", "calvo", "tio"],
    "homepage":         "http://marvel.com/xmen",
    "time":             "1963-09-01",
    "license":          "MIT",
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
            "homepage":     "http://marvel.com",
            "role":         "Genio"
        }
    ]
}
```

Bueno, ahora tenemos un archivo `composer.json` en la raíz de nuestro paquete para los X-Men. ¿Por qué los X-men? Son increíbles, ese es el porqué.

A decir verdad, todas as **opciones** (claves) de este archivo son opcionales. Normalmente pondrás toda la información de arriba si pretendes redistribuir el paquete o lanzarlo al público.

Para ser totalmente sincero contigo, normalmente relleno toda esta información. No hace daño a nadie. La configuración de arriba se usa para identificar el paquete. He omitido unas cuantas claves que creo que están reservadas para circunstancias especiales. Si tienes curiosidad sobre cualquier configuración adicional, te recomiendo que le eches un ojo a la página de Composer[4] que contiene información y documentación.

También encontré esta útil hoja de trucos[5] en línea, que puede ser útil para los recién llegados a Composer al crear paquetes. Pasa el ratón sobre cada línea para descubrir más sobre los elementos de configuración.

Bueno, echemos un vistazo de cerca al archivo de configuración que he creado para el paquete de los X-Men.

```
"name": "marvel/xmen",
```

Este es el nombre del paquete. Si has usado Github[6] el formato del nombre te resultará familiar pero voy a explicarlo de cualquier forma.

[4] http://getcomposer.org/
[5] http://composer.json.jolicode.com/
[6] http://github.com

El nombre del paquete consiste en dos palabras separadas por una barra (/). La parte primera representa el dueño del paquete. En la mayoría de los casos los programadores tienden a usar su nombre de Github como propietario, y estoy totalmente de acuerdo con ello. Sin embargo, puedes usar cualquier nombre que quieras. Asegúrate de ser consistente en todos los paquetes que te pertenezcan.

La segunda parte del nombre es el nombre del paquete. Mantenlo simple y descriptivo. Una vez más, muchos programadores eligen usar el nombre del repositorio para el paquete al alojarlos en Github y, una vez más, estoy totalmente de acuerdo con este sistema.

```
"description": "Mutantes que salvan al mundo de gente que lo odia.",
```

Facilita una breve descripción de la funcionalidad del paquete. Recuerda, mantenlo sencillo. Si piensas usar el paquete para código libre, puedes meterte en detalles en el archivo `README` de tu repositorio. Si quieres poner algo de documentación personal este no es el lugar. Quizá te lo puedas tatuar en tu espalda y tener un espejo a mano. Esto es lo que más sentido tiene. Aunque las notas adhesivas también funcionarán bien.

```
"keywords": ["mutante", "super heroe", "calvo", "tio"],
```

Estas palabras claves son una matriz de cadenas usadas para representar tu paquete. Son similares a etiquetas en una plataforma de blogs y, esencialmente, sirven al mismo propósito. Las etiquetas te ofrecen metadatos de búsqueda para cuando tu paquete sea listado en un repositorio.

```
"homepage": "http://marvel.com/xmen",
```

La configuración de la página es útil para paquetes que van a ser de código libre. Puedes usar esta página para el proyecto o quizá para la URL del repositorio. Lo que creas que es más informativo.

Una vez más, te recuerdo que todas estas opciones son opcionales. Eres libre de omitirlas si no tienen sentido para tu paquete.

```
"time": "1963-09-01",
```

Esta es una de esas opciones que no veo muy a menudo. De acuerdo con la hoja de trucos, representa la fecha de lanzamiento de tu aplicación o librería . Imagino que no es obligatorio en la mayoría de las circunstancias por el echo de que la mayoría de los paquetes están en Github o algún sitio de control de versiones. Estos sitios suelen fechar cada cambio, etiqueta y otros eventos útiles.

Los formatos aceptados son `YYYY-MM-DD` y `YYYY-MM-DD HH:MM:SS`. ¡Ánimo y pon los datos si crees que debes!

```
"license": "MIT",
```

Si tu paquete está pensado para ser redistribuido, querrás ofrecer una licencia con él. Sin una licencia muchos programadores no podrán usar el paquete por restricciones legales. Escoge una licencia que se ajuste a tus requisitos, pero que no sea muy restrictiva para aquellos que esperan usar tu código. El proyecto de Laravel usa la licencia MIT que ofrece gran libertad.

Muchas licencias requieren que tengas una copia de la misma en el repositorio pero si también ofreces esta entrada en la configuración del archivo `composer.json`, el repositorio del paquete podrá listar el paquete por su licencia.

La sección de autores de la configuración ofrece información sobre los autores del paquete, y puede ser útil para aquellos usuarios que quieran contactar con el autor o autores.

Ten en cuenta que la sección de autores permite una matriz de autores para paquetes colaborativos. Veamos las opciones que tenemos.

```
"authors": [
    {
        "name":         "Stan Lee",
        "email":        "stan@marvel.com",
        "homepage":     "http://marvel.com",
        "role":         "Genio"
    }
]
```

Usa un objeto para representar cada autor individual. Nuestro ejemplo solo tiene un autor. Echemos un ojo a Stan Lee. No solo tiene un cameo en cada película de Marvel si no que también aparece en mi libro. ¡Valiente vejestorio!

```
"name": "Stan Lee",
```

No sé como simplificar esta línea. Si tienes problemas entendiéndola deberías considerar cerrar el libro y perseguir una carrera como marionetista de calcetines.

```
"email": "stan@marvel.com",
```

Asegúrate de ofrecer una dirección de correo válida para que puedas ser contactado si el paquete se rompe.

```
"homepage": "http://marvel.com",
```

En esta ocasión, se puede ofrecer una página personal, ¡ánimo y hazte con algunas visitas!

```
"role": "Genio"
```

La opción del rol define el rol del autor en el proyecto. Por ejemplo, programador, diseñador o incluso artista marionetista de calcetines. Si no se te ocurre nada ajustado, escribe algo gracioso.

Esto es todo lo que necesitas para describir tu paquete, ahora echemos un vistazo a algo más interesante. ¡Gestión de dependencias!

Gestión de dependencias

Ahora tienes una caja que contiene a los X-Men. Solo que no hay un montón de mutantes en tu caja aun. Para construir tu gran equipo de superhéroes (aplicación) tendrás que listar el apoyo de otros mutantes (dependencias de terceros). Echemos un vistazo a cómo Composer nos ayuda a solucionar esto.

```
{
    "name":             "marvel/xmen",
    "description":      "Mutantes que salvan al mundo de gente que lo odia.",
    "keywords":         ["mutante", "super heroe", "calvo", "tio"],
    "homepage":         "http://marvel.com/xmen",
    "time":             "1963-09-01",
    "license":          "MIT",
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
            "homepage":     "http://marvel.com",
            "role":         "Genio"
        }
    ],
    "require": {

    }
}
```

Ahora tenemos una nueva sección en nuestro archivo `composer.json` llamado require. Usaremos esto para listar nuestras dependenc... mutantes. Desde ahora omitiré el resto de la configuración y solo mostraré el bloque `require` para acortar los ejemplos. ¡Asegúrate de que sabes dónde va!

Sabemos que los X-Men dependen de::

- Lobezno

- Cíclope
- Tormenta
- Gambito

Hay muchos otros pero estos son bastante guays. Nos quedaremos con ellos por ahora. Como verás, podríamos copiar los archivos fuente para estos tíos en nuestra aplicación directamente, pero entonces tendríamos que actualizarlos nosotros con los cambios. Eso podría ser aburrido. Añadámoslo a la sección `require` para que Composer los gestione por nosotros.

```
"require": {
    "xmen/lobezno":   "1.0.0",
    "xmen/ciclope":   "1.0.1",
    "xmen/tormenta":  "1.2.0",
    "xmen/gambito":   "1.0.0"
}
```

Aquí hemos listado nuestras dependencias de mutantes y las versiones que querríamos usar. En este ejemplo todos pertenecen al mismo propietario que el paquete X-Men, pero podrían fácilmente pertenecer a otra persona.

La mayoría de los paquetes que se redistribuyen se almacenan en sitios de control de versiones como Github[7] o Bitbucket[8]. Los repositorios tienen a menudo un sistema de etiquetas en los que podemos definir versiones estables de nuestra aplicación. Por ejemplo con git podemos usar este comando:

```
git tag -a 1.0.0 -m 'Primera version.'
```

Con esto hemos creado la versión `1.0.0` de nuestra aplicación. Esta es una versión estable de la que la gente podría depender.

Echemos un ojo a la dependencia de Gambito.

```
"xmen/gambito": "1.0.0"
```

Deberías saber ahora que los nombres de los paquetes de Composer consisten en un propietario y un apodo del paquete separados por una barra (`/`). Con esta información sabemos que el paquete `gambito` fue escrito por e usuario `xmen`.

En la sección `require`, la clave para cada elemento es el nombre del paquete y el valor representa la versión requerida.

[7] http://github.com
[8] http://bitbucket.org

En el caso de Gambito, el número de la versión coincide con la etiqueta disponible en Github donde se almacena el código. ¿Ves ahora cómo la versión de la dependencia es específica a nuestra aplicación, y no al sistema completo?

Puedes añadir tantas dependencias como quieras a tu proyecto. Ánimo, ¡añade un billón! Prueba que me equivoco.

Escucha, ¿quieres saber un secreto? ¿Me prometes no contarlo? Vaya, oh oh. Acércate, déjame susurrártelo en el oído. Di las palabras que quieres oir…

> Tus dependencias pueden tener sus propias dependencias.

¡Es correcto! Tus dependencias son también paquetes de Composer. Tienen sus propios archivos `composer.json`. Esto significa que tienen su propia sección `require` con una lista de dependencias y esas dependencias puede que a su vez tengan las suyas propias.

Lo mejor es que Composer se encargará de gestionar e instalar estas dependencias anidadas por ti. ¿No es fantástico? Lobezno puede que necesite `herramientas/garras`, `herramientas/mascara-amarilla` y `poder/regeneracion` pero no tienes que preocuparte por ello. Basta con que coloques el paquete `xmen/lobezno` en la sección `require` y Composer se encargará del resto.

Con respecto a las versiones de las dependencias, pueden asumir varias formas. Por ejemplo puede que no te importen las actualizaciones menores de un componente. En ese caso, podrías usar un asterisco con la versión, tal que así:

```
1  "xmen/gambito": "1.0.*"
```

Ahora Composer instalará la última versión que empiece por `1.0`. Por ejemplo si Gambito tenía versiones `1.0.0` y `1.0.1`, entonces se instalará la `1.0.1`.

Tu paquete puede que también tenga un límite mínimo o máximo de versiones de paquetes. Esto puede ser definido usando los operadores `mayor-que` y `menor-que`.

```
1  "xmen/gambito": ">1.0.0"
```

El ejemplo de arriba podría ser satisfecho con cualquier versión del paquete `xmen/gambito` que sea mayor a la versión `1.0.0`.

```
1  "xmen/gambito": "<1.0.0"
```

De manera similar, el operador `menor-que` es satisfactible con paquetes que sean menores a la versión `1.0.0`. Permitiendo al paquete especificar una versión de dependencia máxima.

```
"xmen/gambito": "=>1.0.0"
"xmen/gambito": "=<1.0.0"
```

Incluyendo el signo igual = junto al operador de comparación, resultará en que la versión comparativa sea añadida a la lista de versiones que satisfagan la limitación de versión.

Ocasionalmente, puede que quieras introducir más de una versión o proveer un rango de valores para la versión de un paquete. Se puede añadir más de una limitacion separándolas por una coma (,). Por ejemplo:

```
"xmen/gambito": ">1.0.0,<1.0.2"
```

El ejemplo de arriba quedará satisfecho con la versión `1.0.1`.

Si no quieres instalar versiones estables, por ejemplo, podrías ser del tipo de gente que les gusta hacer puenting o saltar de un avión, puede que quieras usar versiones a la última. Composer es capaz de seleccionar ramas de un repositorio usando la siguiente sintaxis.

```
"xmen/gambito": "dev-nombrerama"
```

Por ejemplo, si quieres usar el código actual de la rama de desarrollo del proyecto Gambito en Github, tendrías que usar la versión `dev-desarrollo`.

```
"xmen/gambito": "dev-desarrollo"
```

Estas limitaciones de versión no funcionarán a menos que tengas un ajuste mínimo de estabilidad para tu paquete. Por defecto, Composer usa la configuración `stable` o cual restringe a que las versiones de las dependencias sean estables y etiquetadas.

Si quieres sobrescribir esta opción, cambia la configuración `minimum-stability` en tu archivo `composer.json`.

```
"require": {
    "xmen/gambito": "dev-master"
},
"minimum-stability": "dev"
```

Hay otros valores disponibles para el ajuste mínimo de estabilidad, pero explicarlos sería meternos en las profundidades de las etiquetas de la estabilidad. No quiero complicar en demasía este capítulo echándole un ojo a esto. Puede que vuelva más tarde al capítulo y explique el tema, pero por ahora te recomiendo que le eches un ojo a la documentación de Composer para versión de paquetes[9] para encontrar información adicional sobre el tema.

[9] http://getcomposer.org/doc/01-basic-usage.md#package-versions

A veces puede que te veas en la necesidad de usar dependencias que solo estén relacionadas con el desarrollo de tu aplicación. Estas dependencias pueden no ser necesarias para el uso del día a día de tu aplicación en un entorno de producción.

Composer te tiene cubierto en estas situaciones gracias a su sección `require-dev`. Imaginemos por un momento que nuestra aplicación necesita el Framework de pruebas Codeception[10] para ofrecer pruebas de aceptación. Estas pruebas no serán usadas en nuestro entorno de desarrollo por lo que añadámosla a la sección `require-dev` de nuestro `composer.json`.

```
"require": {
    "xmen/gambito": "dev-master"
},
"require-dev": {
    "codeception/codeception": "1.6.0.3"
}
```

El paquete `codeception/codeception` será solamente instalado si usamos `--dev` en Composer. Hablaremos más sobre este tema en la sección de instalación y uso.

Como puedes ver arriba, la sección `require-dev` usa exactamente el mismo formato que la sección `require`. De hecho, hay otras secciones que usan el mismo formato. Echemos un ojo a lo que hay disponible.

```
"conflict": {
    "marvel/spiderman": "1.0.0"
}
```

La sección `conflict` contiene una lista de paquetes que no viven en armonía con nuestro paquete. Composer no te dejará instalar esos paquetes a la vez.

```
"replace": {
    "xmen/gambito": "1.0.0"
}
```

La sección `replace` te informa de que este paquete puede ser usado como reemplazo de otro paquete. Es útil para paquetes que derivan de otros, pero ofrecen la misma funcionalidad.

[10] http://codeception.com/

```
"provide": {
    "xmen/gambito": "1.0.0"
}
```

Esta sección indica paquetes que son facilitados por el código de tu paquete. Si el código de Gambito está incluido en tu paquete principal, entonces sería absurdo instalarlo de nuevo. Usa esta sección para indicar a Composer qué paquetes están dentro de tu paquete principal. Recuerda, no necesitas listar tus dependencias aquí. Cualquier cosa que encuentres en `require` no cuenta.

```
"suggest": {
    "xmen/gambito": "1.0.0"
}
```

Tu paquete puede tener un número de paquetes adicionales que mejoren su funcionalidad pero que no sean estrictamente necesarios. ¿Por qué no añadirlos a la seccion `suggest`? Composer mencionará cualquier paquete en esta sección como sugerencias para instalar a la hora de ejecutar el comando de instalación de Composer.

Bueno, esto es todo lo que tengo sobre dependencias. Echemos un ojo a la siguiente pieza mágica de Composer. ¡Auto carga!

Auto carga

Ahora tenemos el conocimiento sobre cómo permitir a Composer obtener nuestras dependencias por nosotros, pero ¿cómo hacemos para usarlas? Podríamos usar `require()` sobre los archivos fuentes en PHP, pero eso requiere que sepamos exactamente donde se encuentran.

Nadie tiene tiempo para eso. Composer se encargará de ello. Si le decimos a Composer dónde están nuestras clases, y qué métodos puede usar para cargarlas, generará su propia carga automática, que podremos usar en nuestra aplicación para cargar definiciones de clases.

Las acciones hablan más fuerte que las palabras, así que vamos a mostrar un ejemplo.

```
"autoload": {

}
```

Esta es la sección en la que estará toda la configuración de auto carga. Simple, ¿verdad? ¡Bien! No te hago ser marionetista de calcetines.

Echemos un ojo al más simple mecanismo de carga, el método `files`.

```
"autoload": {
    "files": [
        "ruta/a/mi/primerarchivo.php",
        "ruta/a/mi/segundoarchivo.php"
    ]
}
```

El método de carga `files` ofrece una matriz de archivos que serán cargados cuando el autocargador de Composer se cargue en tu aplicación. Las rutas de los archivos se consideran relativas a la carpeta raíz de tu proyecto. Este método de carga es efectivo, pero no muy conveniente. No quieres añadir cada archivo manualmente en un proyecto grande. Echemos un vistazo a algunos métodos mejores para cargar grandes cantidades de archivos.

```
"autoload": {
    "classmap": [
        "src/Models",
        "src/Controllers"
    ]
}
```

El método `classmap` es otro método de carga que acepta una matriz. En esta ocasión, la matriz consiste en un número de directorios que son relativos a la raíz de tu proyecto.

Al generar el código de auto carga, Composer iterará sobre los directorios buscando archivos que contengan clases PHP. Estos archivos serán añadido a una colección una ruta de archivo a un nombre de clase. Cuando una aplicación está usando la auto carga de Composer e intenta instanciar una clase que no existe, Composer entrará y cargará la definición de la clase usando la información almacenada en su mapa.

No obstante, hay una parte negativa de usar este método. Necesitarás usar el comando `composer dump-autoload` para reconstruir el mapa de clases cada vez que añadas un archivo nuevo. Por suerte, hay un último mecanismo de carga y es tan inteligente que no necesita un mapa. Vamos a aprender primero sobre la carga de clases de PSR-0.

La carga de clases PSR-0 fue descrita por vez primera en el estándar PSR-0 de PHP y ofrece una forma sencilla de mapear clases en espacios de nombre a archivos que estuvieran contenidos en ellos.

Sabemos que si añades una declaración `namespace` a un archivo que contenga tu clase, tal que así:

```php
<?php

namespace Xmen;

class Lobezno
{
    // ...
}
```

La clase se convierte en `Xmen\Lobezno` y en lo que concierne a PHP es un animal distinto a la clase `Lobezno`.

Usando la auto carga `PSR-0`, la clase `Xmen\Lobezno` estaría ubicada en el archivo `Xmen/Lobezno.php`.

¿Ves cómo el espacio de nombre coincide con el directorio en el que está la clase? La clase `Lobezno` del espacio de nombre `Xmen` está ubicada dentro del directorio `Xmen`.

También deberías observar que los nombres de archivo coinciden con los nombres de clase, incluyendo los caracteres en mayúsculas. Hacer que el nombre del archivo coincida con el nombre de la clase es esencial para que la auto carga de `PSR-0` funcione correctamente.

Los espacios de nombres pueden tener varios niveles, por ejemplo, considera la siguiente clase.

```php
<?php

namespace Super\Feliz\Divertida;

class Hora
{
    // ...
}
```

La clase `Hora` está ubicada en el espacio de nombre `Super\Feliz\Divertida`. Por tanto PHP reconocerá `Super\Feliz\Divertida\Hora` y no `Hora`.

Esta clase estaría ubicada en la siguiente ruta de archivo.

```
Super/Feliz/Divertida/Hora.php
```

Una vez más, ¿ves cómo coinciden los espacios de nombres? Además, descubrirás que el nombre del archivo es el mismo que el de la clase.

Esto es todo sobre la auto carga `PSR-0`. ¡Es sencillo en realidad! Echemos un ojo a cómo podemos usarla con Composer para simplificar nuestra carga de clases.

```
"autoload": {
    "psr-0": {
        "Super\\Feliz\\Divertida\\Hora": "src/"
    }
}
```

Esta vez, nuestro bloque de auto carga `psr-0`, es un objeto en vez de una matriz. Esto es porque necesita una clave y un valor.

La clave para cada valor en el objeto representa un espacio de nombre. No te preocupes por las dobles barras invertidas. Se usan porque una única barra representaría un carácter de escape en JSON. ¡Recuérdalo a la hora de asignar espacios de nombre en archivos JSON!

El valor es el directorio al que se mapea el espacio de nombre. Me he fijado en que no necesitas la barra, pero muchos ejemplos la usan para denotar que es un directorio.

Lo siguiente es muy importante y es algo serio para mucha gente. Léelo con cuidado.

El segundo parámetro no es el directorio en el que están las clases para ese espacio de nombre. En vez de eso, es el directorio que **comienza** el espacio de nombre para el mapeo de directorio. Echemos un vistazo al ejemplo anterior para ilustrar esto.

¿Recuerdas la case super feliz divertida hora? Vamos a echarle otro vistazo.

```
<?php

namespace Super\Feliz\Divertida;

class Hora
{
    // ...
}
```

Bueno, ahora sabemos que la clase estará en el archivo `Super/Feliz/Divertida/Hora.php`. Con esto en mente, considera el siguiente fragmento de auto carga.

```
"autoload": {
    "psr-0": {
        "Super\\Feliz\\Divertida\\Hora": "src/"
    }
}
```

Podrías esperar que Composer buscara en `src/Hora.php` la clase. Esto sería **incorrecto**, y la clase no sería encontrada.

En vez de ello, la estructura de directorios debería estar en el siguiente formato.

```
src/Super/Feliz/Divertida/Hora.php
```

Esto es algo que pilla a mucha gente desprevenida cuando usan Composer por primera vez. No puedo decir suficientes veces lo importante que es recordar esto.

Si ejecutáramos una instalación de Composer ahora, y luego añadiéramos una clase `Vida.php` al mismo espacio de nombre, no tendríamos que regenerar el auto cargador. Compser sabría exactamente dónde están as clases con ese espacio de nombre y cómo cargarlas. ¡Genial!

Puede que te preguntes porque pongo mis archivos de espacio de nombre en una carpeta `src`. Es una convención común al escribir librerías basadas en Composer. De hecho, he aquí una estructura común de directorio/archivo para un paquete Composer.

```
src/            (Clases.)
tests/          (Pruebas Unitarias/Aceptación.)
docs/           (Documentación.)
composer.json
```

Eres libre de mantener este estándar o hacer lo que te haga más feliz. Laravel facilita su propia ubicación para las clases la cual describiremos en un capítulo más tardío.

Ahora que has aprendido sobre cómo definir tus mecanismos de auto carga, es hora de que miremos a cómo instalar y usar Composer, para que puedas empezar a disfrutar de las ventajas de su auto cargador.

Instalación

Ahora puede que te estés preguntando porqué elijo cubrir la instalación y uso al final del capítulo. Creo que tener buen conocimiento sobre la configuración te ayudará a entender lo que hace Composer detrás del telón mientras lo usamos.

El siguiente método de instalación es específico para sistemas basados en Unix como Linux o Mac OSX. Espero que Taylor pueda editar este capítulo y dar información sobre cómo instalar Composer en un entorno Windows ya que personalmente evito este sistema operativo en particular como a la plaga.

Composer es una aplicación basada en PHP, por ello tienes que tener el cliente de la interfaz de línea de comandos de PHP instalado antes de intentar usarlo. Asegúrate de ello ejecutando este comando.

```
php -v
```

Si PHP ha sido instalado correctamente, deberías ver algo similar a…

```
$ php -v
PHP 5.4.4 (cli) (built: Jul  4 2012 17:28:56)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.4.0, Copyright (c) 1998-2012 Zend Technologies
    with XCache v2.0.0, Copyright (c) 2005-2012, by mOo
```

Si al ejecutarlo aparece que estás usando una versión inferior a PHP 5.3.2, no vas a poder usar Composer hasta que actualices tu versión de PHP. De hecho, si estás usando algo menor que PHP 5.3.7 no podrás usar Laravel para nada.

Puedes usar `curl` para bajarte Composer. Mac OSX viene con esto por defecto. Muchas distribuciones de Linux tienen `curl` en sus repositorios de software, si no ha sido instalado ya como estándar. Usemos `curl` para descargar el ejecutable de Composer.

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

Un usuario experimentado de Linux puede estar preocupado porque `curl` está mandando el script de instalación a PHP. Es una queja justa, pero la instalación de Composer ha sido usada por miles de programadores y está probada ser segura. ¡No dejemos que eso te eche atrás de usar esta maravillosa pieza de software!

Asumiendo que el proceso de instalación se completara correctamente (te lo dirá), ahora tendrás un archivo `composer.phar` en el directorio de tu aplicación. Este es el ejecutable que puede ser usado para lanzar Composer, por ejemplo...

```
php composer.phar
```

... te mostrará una lista de comandos que tienes disponibles.

Ahora, podrías comenzar a usar Composer así, pero te sugeriría que lo instalaras de manera global. De esta manera, podrá ser usado por todos tus proyectos de Composer, y tendrás un comando más corto para ejecutarlo.

Para instalar Composer de manera global, muévelo a una ubicación en tu variable de entorno `PATH`. Puedes ver estas ubicaciones usando el siguiente comando.

```
echo $PATH
```

No obstante, `/usr/local/bin` es una ubicación aceptable para la mayoría de los sistemas. Cuando movamos el archivo, lo renombraremos a `composer` para hacerlo más sencillo de ejecutar. He aquí el comando que necesitamos:

```
sudo mv composer.phar /usr/local/bin/composer
```

Habiendo instalado Composer de forma gloal, ahora podemos usar la sintaxis corta para ver la misma lista de comandos. Este comando lo puedes ejecutar ahora desde cualquier ubicación en el sistema.

```
composer
```

Ey, esto es más limpio ¿no? Comencemos a usar esta cosa.

Uso

Asumamos que hemos creado el siguiente archivo `composer.json` en el directorio de nuestro paquete.

```
{
    "name":              "marvel/xmen",
    "description":       "Mutantes que salvan al mundo de gente que lo odia."
    "keywords":          ["mutante", "super heroe", "calvo", "tio"],
    "homepage":          "http://marvel.com/xmen",
    "time":              "1963-09-01",
    "license":           "MIT",
    "authors": [
        {
            "name":          "Stan Lee",
            "email":         "stan@marvel.com",
            "homepage":      "http://marvel.com",
            "role":          "Genio"
        }
    ],
    "require": {
        "xmen/lobezno":    "1.0.0",
        "xmen/ciclope":      "1.0.1",
        "xmen/tormenta":        "1.2.0",
        "xmen/gambito":       "1.0.0"
    },
    "autoload": {
        "classmap": [
            "src/Xmen"
        ]
    }
}
```

Vamos a usar el comando `install` para instalar todas las dependencias de nuestros paquetes y establecer nuestra auto carga.

```
composer install
```

La salida de Composer será algo similar a:

```
Loading composer repositories with package information
Installing dependencies

  - Installing herramientas/garras (1.1.0)
    Cloning bc0e1f0cc285127a38c232132132121a2fd53e94

  - Installing herramientas/mascara-amarilla (1.1.0)
    Cloning bc0e1f0cc285127a38c6c12312325dba2fd53e95

  - Installing poder/regeneracion (1.0.0)
    Cloning bc0e1f0cc2851213313128ea88bc5dba2fd53e94

  - Installing xmen/lobezno (1.0.0)
    Cloning bc0e1f0cc285127a38c6c8ea88bc523523523535

  - Installing xmen/ciclope (1.0.1)
    Cloning bc0e1f0cc2851272343248ea88bc5dba2fd54353

  - Installing xmen/tormenta (1.2.0)
    Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd53343

  - Installing xmen/gambito (1.0.0)
    Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd56642

Writing lock file
Generating autoload files
```

Recuerda que estos son paquetes falsos usados como ejemplo. ¡Descargarlos no funcionará! No obstante, ¡es divertido porque son X-men!

¿Entonces porqué hay siete paquetes instalados cuando yo solo he listado cuatro? Bueno, te olvidas de que Composer gestiona las dependencias de las dependencias automáticamente. Esos tres paquetes adicionales son las dependencias del paquete `xmen/lobezno`.

Imagino que te estarás preguntando dónde han sido instalados estos paquetes. Composer crea un directorio `vendor` en la raíz de tu proyecto para contener los archivos fuentes de tu paquete.

El paquete `xmen/lobezno` puede ser encontrado en `vendor/xmen/lobezno` donde encontrarás sus archivos fuente junto a su propio `composer.json`.

Composer también guarda algunos archivos propios relacionados con el sistema de auto carga en el directorio `vendor/composer`. No te preocupes por ello, no tendrás que editarlos nunca directamente.

Entonces, ¿cómo usamos las ventajas de las increíbles habilidades de auto carga? Bueno, la respuesta a eso es más simple que establecer la auto carga en sí. Simplemente `require()` o `include()` el archivo `vendor/autoload.php` en tu aplicación. Por ejemplo:

```
<?php

require 'vendor/autoload.php';

// Tu increible codigo de inicializacion aqui
```

¡Genial! Ahora puedes instanciar una clase que pertenezca a una de tus dependencias. Por ejemplo...

```
<?php

$gambito = new \Xmen\Gambito;
```

Composer hará toda la magia y autocargará la definición de la clase por ti. ¿No es eso fantástico? Ya no tienes que guarrear tu código fuente con miles de sentencias `include()`.

Si has añadido un archivo a un directorio relacionado con una clase, tendrás que ejecutar un comando antes de que Composer sea capaz de cargarlo.

```
composer dump-autoload
```

El comando de arriba reconstruirá todas los relaciones y creará un nuevo `autoload.php` para ti.

¿Qué pasa si queremos añadir otra dependencia a nuestro proyecto? Añadamos `xmen/bestia` a nuestro archivo `composer.json`.

```
{
    "name":             "marvel/xmen",
    "description":      "Mutantes que salvan al mundo de gente que lo odia."
    "keywords":         ["mutante", "super heroe", "calvo", "tio"],
    "homepage":         "http://marvel.com/xmen",
    "time":             "1963-09-01",
    "license":          "MIT",
    "authors": [
        {
            "name":         "Stan Lee",
            "email":        "stan@marvel.com",
```

```
            "homepage":     "http://marvel.com",
            "role":         "Genio"
        }
    ],
    "require": {
        "xmen/lobezno":      "1.0.0",
        "xmen/ciclope":      "1.0.1",
        "xmen/tormenta":     "1.2.0",
        "xmen/gambito":      "1.0.0",
                "xmen/bestia":       "1.1.0",
    },
    "autoload": {
        "classmap": [
            "src/Xmen"
        ]
    }
}
```

Ahora tenemos que ejecutar `composer install` de nuevo para que Composer puesta instalar nuestro paquete recién añadido.

```
Loading composer repositories with package information
Installing dependencies

  - Installing xmen/bestia (1.1.0)
    Cloning bc0e1f0c34343347a38c232132132121a2fd53e94

Writing lock file
Generating autoload files
```

Ahora hemos instalado `xmen/bestia` y podemos usarla sin más. ¡Brutal!

Puede que hayas observado la siguiente línea en la salida del comando `composer install`.

```
Writing lock file
```

También puede que te hayas dado cuenta de que Composer ha creado un archivo llamado `composer.lock` en la raíz de tu aplicación. ¿Qué es eso por lo que te oigo llorar?

El archivo `composer.lock` contiene información sobre tu paquete en el momento en el que el último `composer install` o `composer update` fue ejecutado. También contiene una lista de la versión **exacta** de cada dependencia que ha sido instalada.

¿Por qué es esto? Es simple. Cada vez que usas `composer install` cuando `composer.lock` está presente en el directorio, usará la última versión contenida en el archivo en vez de traerse versiones frescas para cada dependencia.

Esto significa que si mantienes tu archivo `composer.lock` en el código de tu aplicación (lo cual recomiendo), al ponerla en el entorno de producción, usará las mismas versiones de dependencias que has probado en tu entorno de desarrollo loca. Esto significa que puedes estar seguro de que Composer no instalará ninguna versión de dependencia que pueda romper tu aplicación.

Ten en cuenta que nunca deberías editar el archivo `composer.lock` manualmente.

Mientras que estamos tratando el tema de la dependencia de versiones, ¿por qué no miramos cómo actualizarlas? Por ejemplo, si teníamos el siguiente requisito en nuestro archivo `composer.json`.

```
"xmen/gambito": "1.0.*"
```

Composer podría instalar la versión `1.0.0` para nosotros. No obstante, ¿qué pasa si el paquete es actualizado a la versión `1.0.1` pocos días después?

Bueno, podemos usar el comando `composer update` para actualizar todas nuestras dependencias a sus últimas versiones. Echemos un vistazo a la salida.

```
$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)

      - Installing xmen/gambito (1.0.1)
        Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd56642

Generating autoload files
```

¡Genial! El paquete `xmen/gambito` ha sido actualizado a su última versión y nuestro fichero `composer.lock` ha sido actualizado

Si solo queríamos actualizar una dependencia en vez de todas, podríamos especificar el nombre del paquete a la hora de usar el comando `update`. Por ejemplo:

```
composer update xmen/gambito
```

Espera, ¿qué significa eso de `(including require-dev)`? Bueno, tienes que recordar la sección `require-dev` en el archio `composer.json` donde listábamos las dependencias de desarrollo. Bueno, Composer espera que el comando de actualización sea ejecutado únicamente en un entorno seguro o de pruebas. Por este motivo, asume que quieres que quieres actualizar también tus dependencias de desarrollo, y las descarga por ti.

Si no quieres esto, puedes usar el siguiente comando.

```
composer update --no-dev
```

Además, si quieres instalar dependencias al usar el comando `install`, usa este simple comando.

```
composer install --dev
```

Lo último que deberías saber, es que puedes usar el comando `composer self-update` para actualizar Composer en sí mismo. Asegúrate de usar `sudo` si lo has instalado de forma global.

```
sudo composer self-update
```

Bueno, esto cubre todo el conocimiento de Composer que necesitaremos al trabajar con Laravel. Ha sido un montón de información que digerir y un capítulo largo para estos cansados dedos, pero espero que hayas conseguido algo de todo esto.

Si crees que un tema en particular necesita ser expandido, ¡házmelo saber!

Arquitectura

Quiero hacer que Code Bright sea una experiencia de aprendizaje más completa y, por ese motivo, he decidido incluir este capítulo. Si estás deseando empezar con la programación, siéntete libre de saltártelo. No obstante, creo que será útil aprender cómo está construido Laravel.

En Laravel 3, el contenedor IoC era un componente que confundía a mucha gente. En Laravel 4, es la parte del framework que mantiene todo unido. No puedo comenzar a expresar su importancia. Bueno, de hecho, necesito que tenga sentido para este capítulo. Vamos a intentarlo.

El contenedor

En Laravel 4, el contenedor es creado al inicio durante el proceso de inicialización. Es un objeto llamado `$app`. Entonces, cuando hablamos de un contenedor ¿qué viene a la mente?

Echemos un ojo a la definición del diccionario. ¿Por qué? No estoy seguro… Aunque he visto a otros autores hacerlo. No quiero que me miren mal. Vamos a ello.

> **Nombre** - Contenedor. Un objeto usado para contener o es capaz de ello, especialmente para transporte o almacenamiento, como envase, cajas, etc.

Creo que esta descripción encaja con el contenedor de Laravel bastante bien. Ciertamente es usado para contener cosas. Seamos sinceros… si no contuviera cosas, 'Contenedor' sería un nombre terrible. Con respecto al transporte, ciertamente facilita el 'mover' otros componentes por el framework cuando necesitamos acceso a ellos.

No es realmente un envase, o una caja. Es más un sistema de almacenamiento con clave/valor.

Una vez que el objeto `$app` ha sido creado, puedes acceder a él usando el método `app()`. Con toda honestidad, probablemente no te hará falta. Pero está ligado a algo que explicaremos más tarde en este capítulo.

Vamos a ver el objeto `app` y lo que contiene. Podemos hacer esto en nuestro archivo `app/routes.php`.

```php
<?php

// app/routes.php

var_dump($app);

die();
```

Observa que no necesitamos usar el método `app() en el archivo` app/routes.php‘. Está disponible en el ámbito global, por lo que podemos acceder a ella directamente.

¡Wow, es un objeto enorme! Bueno, supongo que debería serlo... es el núcleo del framework. Descubrirás que es una instancia de `Illuminate\Foundation\Application`. Todos los componentes de Laravel 4 viven en el espacio de nombre `Illuminate`. Era su nombre en clave durante las primeras fases del diseño del framework, y simplemente no tenemos corazón para quitarlo. ¡Piensa en ello como si Laravel estuviera “iluminando” el mundo de PHP!

Si echamos un ojo al código de la clase `Application`, veremos que extiende `Container`. Ánimo, puedes encontrar el código en:

https://github.com/laravel/framework/blob/master/src/Illuminate/Foundation/Application.php[11]

La clase `Container` es donde ocurre toda la magia. Implemente la interfaz `ArrayAcces`. Esta es una interfaz especial que permite a los atributos de un objeto el ser accedido como una matriz, de manera similar a como se hace en los objetos literales de JavaScript. Por ejemplo, los siguientes dos métodos para acceder a atributos de un objeto serían posibles:

```php
<?php

$object->attribute = 'foo';
$object['attribute'] = 'foo';
```

Durante la inicialización del framework, se ejecutan varias clases que proveen servicios. Estas clases tienen el propósito de registrar componentes individuales del framework en el contenedor.

¿A qué me refiero con un componente? Bueno, un framework está hecho de diferentes partes. Por ejemplo, tenemos una capa de enrutado, un sistema de validación, una capa de autenticación y muchos más paquetes de código que se encargan de funciones específicas. Los llamamos componentes del framework, y ambos encajan en el contenedor para crear el framework completo.

Si es la primera vez que usas Laravel, este ejemplo no tendrá mucho sentido para ti. ¡No te preocupes! Es para que los usuarios de Laravel 3 piensen sobre ello. Echa un ojo a esto.

[11] https://github.com/laravel/framework/blob/master/src/Illuminate/Foundation/Application.php

```php
<?php

// app/routes.php

$app['router']->get('/', function()
{
    return '¡Gracias por comprar Code Bright!';
});
```

Aquí estamos accediendo a la capa de enrutado en el contenedor para crear una ruta que responda a una petición HTTP de tipo `GET`. Esto les será familiar a los usuarios de Laravel 3, aunque la sintaxis parecerá un poco extraña.

Como puedes ver, estamos accediendo al componente de enrutado usando una sintaxis de matriz en nuestro contenedor. Esta es la magia que nos da la interfaz `ArrayAccess`. Si queremos, podríamos acceder también a la capa de enrutado así:

```php
<?php

// app/routes.php

$app->router->get('/', function()
{
    return 'En serio, ¡gracias por comprar Code Bright!';
});
```

Acceder al componente como un atributo de nuestro contenedor funciona exactamente igual que el método que hemos usado anteriormente.

Aunque no es muy bonito, ¿no? Quiero decir, si eres nuevo al framework, probablemente hayas oído sobre la belleza de la sintaxis de Laravel. Si has usado Laravel 3 anteriormente, ya la habrás estado disfrutando.

¿Estás seguro de que no vamos a arruinar esa reputación? ¡Por supuesto que no! Echemos un ojo a algo más de magia en acción.

Fachadas

Mi colega Kenny Meyers hizo una reseña de 'Code Happy' en The Nerdary[12]. Me encantaría devolverle el favor. Nos dijo en la última conferencia de Laravel que tenía dificultades con la pronunciación de ciertas palabras, y apuesto a que esta es difícil. Así que, para Kenny, y aquellos que no habláis Inglés como lengua materna, he aquí cómo se pronuncia Facade.

[12] http://www.thenerdary.net/

> **N del T**: En inglés, Facade significa Fachada y dado que no es traducido por algunos, surge a veces la confusión y de ahí la broma del autor. Facade/Fachada es un patrón de diseño. Echa un ojo a la [Wikipedia](http://es.wikipedia.org/wiki/Facade_-(patr%C3%B3n_de_dise%C3%B1o) para más información.

"fah-sahd"

En Laravel, accedíamos a la mayoría de los componentes usando métodos estáticos. Por ejemplo, el ejemplo de enrutado del último capítulo sería algo así:

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      return 'Gracias Kenny, ¡te adoramos!';
8  });
```

Es hermoso... tenemos un nombre descriptivo para el componente, y a menudo un verbo que describe la acción realizada sobre ese componente.

Por desgracia, los programadores experimentados se encogerán al ver un método estático. Los métodos estáticos pueden ser difíciles de probar. No quiero complicar mucho las cosas al principio del libro porqué, así que tendrás que confiar en mi.

Esto creó un gran problema durante el diseño de Laravel 4. Nos gustaba nuestra bonita y sencilla sintaxis, pero también queríamos apoyar las mejores prácticas de desarrollo, y eso incluye escribir muchas, muchas pruebas, para asegurarnos de que el código es robusto.

Por suerte, Taylor apareció con la maravillosa idea de clases Fachada, nombrado por el patrón de diseño 'Fachada'. Usando Fachadas podemos tener lo mejor de ambos mundos. Bonitos métodos estáticos y componentes instanciados con métodos públicos. Esto significa que nuestro código es bonito y se puede probar fácilmente. Veamos cómo funciona.

En Laravel 4, hay un número de clases que tienen un alias en el espacio de nombre raíz. Esas son nuestras clases Fachada, y todas extienden la clase `Illuminate\Support\Facades\Facade`. Esa clase es muy lista. Voy a explicar su inteligencia con algunos ejemplos de código. Como verás, podemos usar métodos de rutado estáticos, al igual que en Laravel 3, tal que así:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return 'iGracias por las Fachadas, Taylor!';
});
```

La Fachada en este ejemplo es la clase `Route`. Cada Fachada está enlazada a una instancia de un componente en el contenedor. Esos métodos estáticos de la Fachada son atajos, y al ser llamados, llaman al método público apropiado del objeto que representan en el contenedor. Esto significa que el método `Route::get()` llama al siguiente método:

```php
<?php

// app/routes.php

$app['router']->get('/', function()
{
    return 'iGracias por las Fachadas, Taylor!';
});
```

¿Por qué esto es importante? Bueno, como verás, ofrece mucha flexibilidad.

Flexibilidad

¿Qué significa flexibilidad? No voy a sacar el diccionario esta vez. Sé que significa habilidad de cambiar. Esa es una buena explicación del beneficio del contenedor. Como verás, podemos cambiar, o incluso, reemplazar objetos y componentes que han sido registrados en el contenedor. Esto nos ofrece mucho poder.

Imaginemos por un momento que no nos gusta cómo funciona la capa de enrutado (no te preocupes, te gustará, solo imagina que no te gusta).

Como somos programadores épicos, escribiremos nuestra propia capa de enrutado, siendo la clase principal `SuperRouter`. Debido a la flexibilidad del contenedor, podemos reemplazar la capa de enrutado con la nuestra propia. Es tan fácil como podrías imaginar. Simplemente asignamos nuestra ruta al índice de `router`. Veámoslo en acción. Por cierto, no recomiendno hacer esto ahora mismo, especialmente si eres un novato. Solo queda bien para un ejemplo.

```php
<?php

// app/routes.php

$app['router'] = new SuperRouter();
```

Ahora, no solo podemos usar nuestro enrutador fuera del contenedor si no que nuestra Fachada `Route` seguirá accediendo a este componente.

De hecho, incluso es más impresionante. Cuando reemplazas componentes de Laravel, el framework intentará usar tus componentes para su tareas, como si fuera el suyo propio.

Permitirnos acceso a componentes del núcleo del framework de esta forma, nos ofrece gran potencial. ¡Seguro que estás impresionado!

Robustez

Como mencioné un par deveces antes, Laravel está hecho con varios componentes individuales. Cada componente es responsable de su propia funcionalidad. Esto significa que son muy respetuosos con el principio de responsabilidad única.

De hecho, muchos de los componentes de Laravel pueden funcionar por sí mismo, fuera del framework. Por este motivo, se pueden encontrar copias de los componentes en la organización Illuminate en Github[13]. Este conjunto de componentes está también disponible en Packagist[14] para que puedas usarlos en tus propios proyectos con la ayuda de Composer.

Entonces, ¿por qué se llama este capítulo Robustez? Bueno, como verás cada componente usado por el framework de Laravel está bien probado. El Framework contiene un conjunto de más de 900 pruebas y 1700 afirmaciones.

Gracias a ellos, podemos aceptar contribucioens y hacer cambios que ofrecerán un fructífero futuro para el framework, sin preocuparnos si un cambio ha roto algo, o elimina funcionalidad existente.

Bueno, es suficiente sobre la arquitectura. Apuesto a que estás preparado para meterte a desarrollar, ¿verdad? Vamos a comenzar.

[13] https://github.com/illuminate

[14] https://packagist.org/

Empezando

Laravel es un framework para el lenguaje de programación PHP. Aunque PHP es conocido por tener una sintaxis poco deseable, es fácil de usar, fácil de desplegar y se le puede encontrar en muchos de los sitios web modernos que usas día a día. Laravel no solo ofrece atajos útiles, herramientas y componentes para ayudarte a conseguir el éxito en tus proyectos basados en web, si no que también intenta arreglar alguna de las flaquezas de PHP.

Laravel tiene una sintaxis bonita, semántica y creativa, que le permite destacar entre la gran cantidad de frameworks disponibles para el lenguaje. Hace que PHP sea un placer, sin sacrificar potencia y eficiencia. Laravel es una gran elección para proyectos amateur así como para soluciones empresariales y, ya seas un auténtico profesional del lenguaje o un novato, Code Bright va a ayudarte a convertir las ideas que tienes en aplicaciones web funcionales.

Echemos un vistazo rápido a los requisitos para el framework y el libro.

Requisitos

- ***Un equipo*** Leer es genial, pero aprenderás más jugando con los ejemplos que encuentres en el libro.
- ***Un servidor web*** Laravel necesita un servidor web. No importa cuál sea pero he descubierto que la mayoría de la comunidad usa Apache o Nginx y hacer lo mismo te pondrá las cosas más fáciles a la hora de buscar ayuda si la necesitas.
- ***PHP: Versión 5.3 o superior*** Laravel es un framework de PHP, requiere el lenguaje de programación PHP. Confía en mi, lo vas a usar. Teniendo en cuenta que Laravel usa algunas características modernas del lenguaje, también necesitarás la versión 5.3.7 o superior. Puedes descubrir la versión que tienes la mayoría de los servidores web escribiendo `php -v` en la consola o usando el método `phpinfo()`.
- ***Un servidor de base de datos*** Aunque no es un requisito del framework, muchos de los ejemplos del libro interactúan con una base de datos. Por este motivo te recomendaría que configuraras un servidor de base de datos soportado por el conector PDO. Aunque te recomendaría que uses el flexible y gratuito `MySQL` de Su… Oracle, también puedes usar otras elecciones populares como SQL Server, Postgres y SQLite.
- ***Un editor de texto*** Lo necesitarás para jugar con los ejemplos que encuentres en este libro. Te recomiendo que uses Sublime Text 2 que, aunque no es gratis, es extremadamente sexy. No obstante hay millones de editores e IDEs disponibles, encuentra uno que se ajuste a las necesidades de la forma en que trabajas.

Ahora antes de que puedas empezar a trabajar con tus proyectos de Laravel, tenemos que descargarnos primero una copia del framework.

Instalación

No estoy seguro de que instalación sea el título más adecuado para esta sección. No obstante, no se me ocurría nada mejor.

Como verás, Laravel tiene un repositorio Github que actúa como 'plantilla' para tu nueva aplicación Laravel. Vamos a copiarlo en nuestra máquina local.

Usaremos git para 'clonar' el repositorio a una carpeta de nuestro servidor de desarrollo web. Este es el comando.

```
git clone git@github.com:laravel/laravel.git mi_proyecto
```

Ahora tendrás una plantilla de aplicación de Laravel en la carpeta `mi_proyecto`. Puede que descubras a otros refiriéndose a esta plantilla como 'paquete de la aplicación'. Este paquete contiene tu aplicación completa y el directorio seguramente esté controlado por un sistema de control de versiones.

Algunos usuarios experimentados de Laravel puede que recuerden un directorio llamando `laravel` que era usado para almacenar los archivos que contenían el framework. A veces nos podríamos referir a ellos como los archivos del 'núcleo' del framework. Bueno, ya no puedes encontrar ese directorio. Con el uso del poder de Composer, los archivos del núcleo del framework ahora existen como un paquete separado que es una dependencia de nuestra plantilla.

Echemos un vistazo al archivo `composer.json` dentro del directorio `mi_proyecto` para ver qué hay de nuevo.

```
{
    "require": {
        "laravel/framework": "4.0.*"
    },
    "autoload": {
        "classmap": [
            "app/commands",
            "app/controllers",
            "app/models",
            "app/database/migrations",
            "app/database/seeds",
            "app/tests/TestCase.php"
        ]
    },
    "scripts": {
        "post-update-cmd": "php artisan optimize"
    },
```

```
18      "minimum-stability": "dev"
19  }
```

Ten en cuenta que la versión de la dependencia puede haber cambiado desde que esta sección se actualizó por última vez. El resultado, no obstante, será el mismo.

Nuestro paquete de aplicación depende únicamente del paquete `laravel/framework` que contiene todos los componentes necesarios para hacer que tu aplicación funcione. Este archivo `composer.json` es nuestro. Es para nuestra aplicación y podemos editarlo como nos apetezca. No obstante, se han colocado algunos valores por defecto sensibles para ti. Tampoco te recomendaría eliminar el paquete `laravel/framework`. Pasarán cosas muy malas.

Ahora solo tenemos una plantilla. Ejecutemos `composer install` para instalar el núcleo del framework.

```
1  Loading composer repositories with package information
2  Installing dependencies
3      - Installing doctrine/lexer (dev-master bc0e1f0)
4        Cloning bc0e1f0cc285127a38c6c8ea88bc5dba2fd53e94
5
6      - Installing doctrine/annotations (v1.1)
7        Loading from cache
8
9      ... Many more packages here. ...
10
11     - Installing ircmaxell/password-compat (1.0.x-dev v1.0.0)
12       Cloning v1.0.0
13
14     - Installing swiftmailer/swiftmailer (dev-master e77eb35)
15       Cloning e77eb358a1aa7157afb922f33e2afc22f6a7bef2
16
17     - Installing laravel/framework (dev-master 227f5b8)
18       Cloning 227f5b85cc2201b6330a8f7ea75f0093a311fe3b
19
20 predis/predis suggests installing ext-phpiredis (Allows faster serialization and \
21 deserialization of the Redis protocol)
22 symfony/translation suggests installing symfony/config (2.2.*)
23 symfony/translation suggests installing symfony/yaml (2.2.*)
24 symfony/routing suggests installing symfony/config (2.2.*)
25 symfony/routing suggests installing symfony/yaml (2.2.*)
26 symfony/event-dispatcher suggests installing symfony/dependency-injection (2.2.*)
27 symfony/http-kernel suggests installing symfony/class-loader (2.2.*)
28 symfony/http-kernel suggests installing symfony/config (2.2.*)
```

```
symfony/http-kernel suggests installing symfony/dependency-injection (2.2.*)
symfony/debug suggests installing symfony/class-loader (~2.1)
monolog/monolog suggests installing mlehner/gelf-php (Allow sending log messages \
to a GrayLog2 server)
monolog/monolog suggests installing ext-amqp (Allow sending log messages to an AM\
QP server (1.0+ required))
monolog/monolog suggests installing ext-mongo (Allow sending log messages to a Mo\
ngoDB server)
monolog/monolog suggests installing doctrine/couchdb (Allow sending log messages \
to a CouchDB server)
monolog/monolog suggests installing raven/raven (Allow sending log messages to a \
Sentry server)
Writing lock file
Generating autoload files
```

Una vez más, las versiones de los paquetes pueden haber cambiado, pero el resultado será el mismo.

¡Vaya esa es una lista larga de paquetes! ¿Para qué son todos? Bueno, como verás, Laravel se aprovecha del potencial del código libre y la riqueza de paquetes que existen en Composer. Esos paquetes son dependencias del framework en sí.

Deberías, sin duda, echar un vistazo a los paquetes listados en la web de Packagist[15]. No hay nada bueno en reinventar la rueda cuando se trata de código de iniciación.

Teniendo en cuenta que has leído el capítulo de Composer, ahora sabrás que los archivos del núcleo de Laravel han sido instalados en la carpeta vendor. Como no querrás añadir esa carpeta a un sistema de control de versiones, Laravel te da un archivo `.gitignore` de ejemplo para ignorar la carpeta vendor, junto a otras cosas por defecto.

Ya casi hemos terminado de configurar nuestro entorno de desarrollo de Laravel. Solo nos queda una cosa. Cómo configurar nuestro servidor web.

Configuración del servidor web

Esta sección siempre fue una difícil de escribir. Como verás, todo el mundo tiene una configuración ligeramente diferente, y hay un gran número de servidores web diferentes disponibles.

Así que esto es lo que voy a hacer. Cubriré lo básico sobre hacia dónde tenemos que apuntar el servidor web. También ofreceré alguna configuración de ejemplo para servidores web comunes. No obstante, serán genéricas y requerirán muchos ajustes para adaptarse a cada situación. Al menos, ¡no podrás decir que no lo intenté!

Echemos un ojo al objetivo. Laravel tiene un directorio llamado `public` que contiene su código de inicialización. Este código es usado para arrancar el framework y gestionar todas las peticiones de tu aplicación.

[15] http://packagist.org

La carpeta `public` también contiene todos tus archivos públicos como JavaScript, CSS e imágenes. Esencialmente, cualquier cosa que pueda ser accedida a través de un enlace debería existir en el directorio `public`, y no en la raíz de nuestro proyecto.

La siguiente tarea será hacer saber al servidor web cómo gestionar las URLs bonitas.

> He escogido un nombre de dominio guay, ¿no tengo ya una URL bonita?

Por desgracia, no es así como funciona. La vida no es un camino de rosas, ¿sabes? El código de iniciación de Laravel está en un archivo llamado `index.php` en la carpeta `public`. Todas las peticiones al framework van a través de este archivo. Esto significa que, por defecto, la URL de nuestra página de libro de invitados será así:

```
1 http://eslavidauncaminoderosas.com/index.php/libroinvitados
```

Los usuarios de nuestro sitio no necesitan saber realmente que todo pasa a través de `index.php`. Tampoco es algo bueno para la optimización para motores de búsqueda. Por ese motivo, nuestro servidor web debería ser configurado para eliminar `index.php` de la URL, dejando únicamente el segmento 'bonito'. Normalmente, esto se consigue con una porción de configuración creada por un mago de las expresiones regulares.

Echemos un vistazo a algunas configuraciones de servidores web que nos permitirán conseguir el objetivo aquí mencionado. Una vez más, debo recordarte que estas configuraciones se deben usar como guía en líneas generales. Para información más detallada sobre configuración del servidor te recomendaría que echaras un vistazo a la documentación del servidor web que has elegido.

Empecemos con nginx.

Nginx

Nginx, pronunciado 'engine-X', es un maravilloso servidor web que he comenzado a usar recientemente. Para mi la elección fue simple. Era mucho más rápido que Apache no necesitaba configuración de estilo XML. Todo tenía sentido para mi.

En una distribución basada en Debian, como Ubuntu, puedes instalar nginx y PHP ejecutando el siguiente comando.

```
1 sudo apt-get install nginx php5-fpm
```

El segundo paquete es PHP-FPM, un módulo de FastCGI que permite a nginx ejecutar código PHP.

En Mac, esos paquetes están disponibles en Macports[16]. Los paquetes necesarios pueden ser instalados con el siguiente comando.

[16] http://www.macports.org/

```
sudo port install php54-fpm nginx
```

Tus archivos de configuración Ngninx estarán normalmente ubicados en `/etc/nginx/sites-enabled`. He aquí una plantilla que puedes usar para configurar tu nuevo sitio.

```
server {

    # Port that the web server will listen on.
    listen          80

    # Host that will serve this project.
    server_name     app.dev

    # Useful logs for debug.
    access_log      /path/to/access.log;
    error_log       /path/to/error.log;
    rewrite_log     on;

    # The location of our projects public directory.
    root            /path/to/our/public;

    # Point index to the Laravel front controller.
    index           index.php;

    location / {

        # URLs to attempt, including pretty ones.
        try_files   $uri $uri/ /index.php?$query_string;

    }

    # Remove trailing slash to please routing system.
    if (!-d $request_filename) {
        rewrite     ^/(.+)/$ /$1 permanent;
    }

    # PHP FPM configuration.
    location ~* \.php$ {
            fastcgi_pass                    unix:/var/run/php5-fpm.sock;
            fastcgi_index                   index.php;
            fastcgi_split_path_info         ^(.+\.php)(.*)$;
            include                         /etc/nginx/fastcgi_params;
```

```
38              fastcgi_param                   SCRIPT_FILENAME $document_root$fastcg\
39  i_script_name;
40      }
41
42      # We don't need .ht files with nginx.
43      location ~ /\.ht {
44              deny all;
45      }
46
47  }
```

Ahora, no todos los servidores web van a ser lo mismo. Esto significa que ofrecerte un archivo de configuración genérico que se ajuste a todas las situaciones sería una tarea imposible. Aun así, es suficiente para que puedas empezar.

He compartido la configuración estándar usada en este capítulo en Github para que todo el mundo pueda contribuir a ella. Puedes encontrarla en el repositorio daylerees/laravel-website-configs[17].

Aunque nginx es una gran elección de un servidor web, el servidor web Apache es una elección ampliamente usada. Echemos un vistazo a cómo configurarlo.

Apache

El servidor web Apache puede ser instalado en sistemas basados en Debian usando el siguiente comando.

```
1  sudo apt-get install apache2 php5
```

He aquí una configuración de un VirtualHost de Apache que cubrirá la mayoría de las situaciones. Eres libre de contribuir al repositorio en Github[18] si se necesita algún ajuste.

```
1  <VirtualHost *:80>
2
3      # Host that will serve this project.
4      ServerName      app.dev
5
6      # The location of our projects public directory.
7      DocumentRoot    /path/to/our/public
8
9      # Useful logs for debug.
```

[17] http://github.com/laravel-website-configs
[18] http://github.com/laravel-website-configs

```
    CustomLog       /path/to/access.log common
    ErrorLog        /path/to/error.log

    # Rewrites for pretty URLs, better not to rely on .htaccess.
    <Directory /path/to/our/public>
        <IfModule mod_rewrite.c>
            Options -MultiViews
            RewriteEngine On
            RewriteCond %{REQUEST_FILENAME} !-f
            RewriteRule ^ index.php [L]
        </IfModule>
    </Directory>

</VirtualHost>
```

Estructura del proyecto

Tenía un capítulo en el libro anterior que cubría la estructura de directorios del paquete y dónde está todo ubicado. Creo que había mucho valor en ese capítulo por lo que me gustaría reiterarlo junto a los cambios en la estructura de directorios que ha ocurrido desde Laravel 3.

Esta sección de iniciación asumirá que ya has ejecutado `composer install` sobre un proyecto fresco de Laravel 4.

Directorio raíz

Comencemos echando un vistazo a la estructura del directorio raíz.

- app/
- bootstrap/
- vendor/
- public/
- .gitattributes
- .gitignore
- artisan
- composer.json
- composer.lock
- phpunit.xml
- server.php

¿Por qué no vamos sobre cada uno de los elementos? ¡Me parece una gran idea!

app

Lo primero que tenemos es el directorio `app`. App es usado para ofrecer un hogar por defecto a todo el código personal de tu proyecto. Eso incluye clases que puedan ofrecer funcionalidad a la aplicación, archivos de configuración y más. La carpeta `app` es bastante importante por lo que en vez de hacer un pobre resumen en un único párrafo, la cubriremos en detalle al final de esta sección. Por ahora, únicamente es donde están los archivos de nuestro proyecto.

bootstrap

- autoload.php
- paths.php
- start.php

El directorio bootstrap contiene unos pocos archivos que están relacionados con los procedimientos de inicialización del framework. El archivo `autoload.php` contiene la mayoría de esos procedimientos y solo debería ser editado por usuarios experimentados de Laravel.

El archivo `paths.php` contiene una matriz de las rutas comunes del sistema que son usadas por el framework. Si por algún motivo decides alterar la estructura de directorios del framework, tendrás que alterar los contenidos de este archivo para reflejar tus cambios.

El archivo `start.php` contiene más procedimientos de inicialización para el framework. No quiero ahondar en esos detalles ahora mismo ya que puede causar confusión innecesaria. En vez de eso, probablemente deberías apuntar que se pueden establecer los *entornos* (`environments`) del framework aquí. Si no sabes para lo que se usan los entornos, no te preocupes. ¡Lo cubriremos más adelante!

Dicho sencillamente, los contenidos del directorio `bootstrap` solo deberían ser editados por usuarios experimentados de Laravel que necesiten alterar la forma del sistema de archivos del framework. Si eres nuevo con Laravel ignóralo por ahora, ¡pero no lo borres! Laravel necesita este directorio para funcionar.

vendor

El directorio `vendor` contiene todos los paquetes de Composer que son utilizados por tu aplicación. Por supuesto, esto incluye el paquete del framework de Laravel. Para más información sobre este directorio, por favor echa un ojo al capítulo sobre Composer.

public

- packages/
- .htaccess
- favicon.ico
- index.php

- robots.txt

El directorio `public` debería ser la única entrada web de una aplicación Laravel. Normalmente es donde se encuentran tus archivos CSS, JavaScript e imágenes. Echemos un vistazo más atentamente a sus contenidos.

El archivo `packages` será usado para contener cualquier fichero que necesite ser instalado por paquetes de terceras partes. Se mantienen en un directorio separado para que no creen conflictos con los ficheros de nuestra propia aplicación.

Laravel `4` viene con un archivo `.htaccess` para servidores Apache. Contiene algunas directivas de configuración estándar que tendrán sentido para la mayoría de los usuarios del framework. Si usas un servidor web alternativo puedes ignorar o borrar este archivo.

Por defecto, los navegadores web intentarán buscar en la raíz de un sitio para encontrar un archivo `favicon.ico`. Este es el archivo que controla la pequeña imagen de 16 x 16px que se muestra en las pestañas de tu navegador. El problema es que cuando el archivo no existe, al servidor web le gusta quejarse sobre ello. Esto causa entradas en el log innecesarias. Para contrarrestar este problema, Laravel da un archivo `favicon.icon` en blanco, que puede ser reemplazado más tarde si se desea.

El archivo `index.php` es el controlador frontal del framework de Laravel. Es el primer archivo que el servidor web ejecuta cuando llega una petición del navegador. Es el archivo que lanzará la inicialización del framework y comenzará a pasar la pelota. No borres este archivo, ¡no sería buena idea!

Laravel ha incluido el archivo `robots.txt` que permite todos los hosts por defecto. Esto es por tu bien, puedes modificar este archivo como creas necesario.

.gitattributes

Laravel ofrece algunos ajustes por defecto para control de versiones con Git. Git es el sistema de control de versiones más popular. Si no pretendes usar tu proyecto con Git, puedes borrar este archivo.

.gitignore

El archivo `.gitignore` contiene algunos ajustes para informar a Git de qué carpetas no debería controlar. Ten en cuenta que éste incluye tanto el directorio `vendor` como el directorio `storage`. El directorio `vendor` ha sido incluído para evitar que se controlen paquetes de terceras partes.

El archivo `composer.lock` también está incluido aunque puede que quieras borrar esta entrada para instalar las mismas versiones de dependencias de tu entorno de desarrollo. Descubrirás más sobre esto en el capítulo sobre Composer.

artisan

El archivo `artisan` es un ejecutable que es usado para ejecutar la interfaz de línea de comandos Artisan para Laravel. Artisan contiene un buen número de comandos útiles para ofrecer atajos o funcionalidad adicional al framework. Cubriremos estos comandos en detalles en un capítulo posterior.

composer.json y composer.lock

Tanto `composer.json` como `composer.lock`, contienen información sobre los paquetes de Composer usados por este proyecto. Una vez más puedes encontrar más información sobre estos archivos en el capítulo de Composer.

phpunit.xml

El archivo `phpunit.xml` ofrece configuración por defecto para las pruebas unitarias de PHPUnit. Gestionará la carga de dependencias de Composer y ejecutará cualquier prueba ubicada en la carpeta `app/tests`. Revelaremos información sobre las pruebas en Laravel en un capítulo posterior, ¡permanece atento!

server.php

@todo: Esto necesita más investigación

El directorio de la aplicación

Aquí es donde tu aplicación toma forma. Es el directorio en el cual pasarás la mayor parte de tu tiempo. Por este motivo, ¿por qué no os conocéis mejor?

- commands/
- config/
- controllers/
- database/
- lang/
- models/
- start/
- storage/
- tests/
- views/
- filters.php
- routes.php

commands

Este directorio contiene cualquier comando personalizado de Artisan que pueda necesitar tu aplicación. Como verás, Artisan no solo ofrece funcionalidad por defecto para ayudarte a construir tu proyecto, si no que también te ofrece la oportunidad de crear tus propios comandos para hacer cosas.

config

La configuración tanto para el framework como para tu aplicación se mantiene en este directorio. La configuración de Laravel existe como un conjunto de archivos PHP que contienen matrices

clave-valor. Este directorio también contiene sub-directorios que permiten distintas configuraciones cargadas en diferentes entornos.

controllers

Como el nombre sugiere, este directorio contendrá tus controladores. Los controladores pueden ser usados para facilitar lógica a la aplicación, y hacer de pegamento entre las partes separadas de tu aplicación. Este directorio ha sido añadido al archivo `composer.json` por defecto para la auto carga de clases.

database

Si escoges una base de datos como método para guardar cosas a largo plazo, este directorio sera usado para contener los archivos que crearán el esquema de tu base de datos, y los métodos para completarla con datos de ejemplo. La base de datos por defecto en SQLite también está ubicada en este directorio.

lang

El directorio `lang` contiene archivos PHP con matrices de cadenas que pueden ser usados para dar soporte de traducción a tu aplicación. Se pueden crear sub carpetas por región para que tengas distintos ficheros para múltiples idiomas.

models

Este directorio contendrá tus modelos. ¿Sorprendido? Los modelos son usados para representar el modelo de negocio o facilitar interacción con el almacenamiento. ¿Confundido? No te preocupes, cubriremos los modelos con más detalle en un capítulo posterior. Debes saber que hay un modelo `User` para ofrecerte autenticación en la aplicación por defecto. Al igual que el directorio `controllers`, este ha sido añadido a la sección de carga automática del archivo `composer.json` por defecto.

start

Mientras que el directorio `bootstrap` contiene los procedimientos de arranque que pertenecen al framework, el directorio `start` contiene procedimientos de arranque que pertenecen a tu aplicación. Como siempre, se ofrecen algunos por defecto.

storage

Cuando Laravel necesita escribir algo en el disco, lo hace en el directorio `storage`. Por este motivo tu servidor web debe poder escribir en esta ubicación.

tests

El directorio `tests` contiene todas las pruebas unitarias y de aceptación para tu aplicación. La configuración por defecto de PHPUnit que ha sido incluida por Laravel buscará pruebas en este directorio por defecto.

views

El directorio `views` es usado para contener las plantillas visuales de tu aplicación. Se facilita una vista `hello` por defecto para tu conveniencia.

filters.php

El archivo `filters.php` es usado para contener cualquier filtro de rutas de tu aplicación. Descubrirás más sobre los filtros en un capítulo futuro.

routes.php

El archivo `routes` contiene todas las rutas de tu aplicación. ¿No sabes lo que son las rutas? Bueno, no malgastemos más tiempo. ¡Hacia el próximo capítulo!

Enrutado básico

Estás en un largo y recto camino lleno de barro en medio de lo salvaje en… ¿Qué pais tiene salvajismo? ¡Australia! Eso es.

¡No te asustes! Esta sección requiere algo de imaginación, así que trabaja conmigo en ello. Supongo que nuestro protagonista debería tener un nombre porque los libros parecen hacerlo. Uhm… vamos a llamarlo Navegador Mc'Petición.

Navegador Mc'Petición está caminando por un camino lleno de barro, en algún lugar del país canguro. Navegador no es un nativo de la zona y se encuentra perdido sin saber a donde ir. No obstante, sabe dónde quiere estar. Un amigo le habló sobre "Jason Lewis' Refugio de Marsupiales con Bigote". Sabe que Noviembre se acerca rápidamente y los estilistas de canguros necesitan toda la ayuda posible.

De repente, Navegador se encuentra con lo que parece ser una encruzijada en la carretera allá en la distancia. Nervioso al ver algo distinto a su camino de barro, corre hacia la encruzijada esperando poder encontrar algo de señalación.

Al llegar, busca arriba y abajo cualquier indicación de qué camino debería tomar para llegar al refugio, pero no encuentra nada.

Por suerte para Navegador, en ese mismo momento, un programador web salvaje apareció. El fuerte y apuesto programador web clava una señal en la fuerte, seca y polvorienta tierra con facilidad y se aleja en menos de lo que dura un parpadeo.

Navegador ha quedado perplejo tanto por la velocidad como por lo apuesto del programador web. ¿He mencionado que era guapo? El programador web… no Navegador. Navegador se parece mucho a ESE TIO de ESE LIBRO que está diciendo todo el rato 'mi tesoro'. Al menos así es como lo veo.

Tras recuperar el aliento, Navegador comienza a estudiar la señal.

A la izquierda encontrarás "Establos de monta de Bob Bobcat", y a la derecha… ¡ah! Allá vamos. A la derecha encontrarás "Jason Lewis' Refugio de Marsupiales con Bigote".

Navegador ha encontrado su camino al Refugio de Marsupiales y ha vivido una larga y feliz vida aprendiendo cómo cepillar los distintos tipos de bigotes que los canguros pueden desarrollar.

Como ves, la moraleja de la historia es que sin un programador (GUAPO) que le muestre el camino, Navegador Mc'Petición no habría encontrado el camino a nuestra aplicac… el Refugio de Marsupiales.

Al igual que en la historia, una petición de un navegador web no llegaría a la lógica de tu aplicación sin algo de enrutado.

Enrutado básico

Echemos un ojo a la petición que se hace al framework de Laravel.

```
http://dominio.com/mi/pagina
```

En este ejemplo, estamos usando el protocolo `http` (usado por la mayoría de los navegadores web) para acceder a tu aplicación Laravel alojada en `dominio.com`. La porción `mi/pagina` de la URL es lo que usaremos para enrutar las peticiones web a la lógica apropiada.

Iré más allá y te mostraré el camino. Las rutas son definidas en el archivo `app/routes.php`, así que vamos allá y creemos una ruta que escuche la petición que hemos mencionado arriba.

```
<?php

// app/routes.php

Route::get('mi/pagina', function() {
    return '¡Hola mundo!';
});
```

Ahora entre en `http://dominio.com/mi/pagina` con tu navegador web, cambiando `dominio.com` por la dirección de tu aplicación Laravel. Probablemente sea `localhost`.

Si todo ha sido configurado correctamente, ¡verás ahora las palabras `¡Hola mundo!` con Times New Roman! Porqué no lo echamos un ojo más atentamente a la declaración de la ruta para ver cómo funciona.

Las rutas están siempre declaradas usando la clase `Route`. Eso es lo que tenemos al principio, antes de `::`. La parte `get` es el método que usamos para 'capturar' las peticiones que son realizadas usando el verbo 'GET' de HTTP hacia una URL concreta.

Como verás, todas las peticiones realizadas por un navegador web contienen un verbo. La mayoría de las veces, el verbo será `GET`, que es usado para solicitar una página web. Se envía una petición `GET` cada vez que escribes una nueva dirección web en tu navegador.

Aunque no es la única petición. También está `POST`, que es usada para hacer una petición y ofrecer algunos datos. Normalmente se usa para enviar un formulario en la que se necesita enviar los datos sin mostrarlo en la URL.

Hay otros verbos HTTP disponibles. He aquí algunos de los métodos que la clase de enrutado tiene disponible para ti:

```php
<?php

// app/routes.php

Route::get();
Route::post();
Route::put();
Route::delete();
Route::any();
```

Todos esos métodos aceptan los mismo parámetros, por lo que puedes usar cualquier método HTTP que sea apropiado para la situación. Esto es conocido como `enrutado REST`. Hablaremos sobre esto con más detalle luego. Por ahora, todo lo que tienes que saber es que se usa `GET` para hacer peticiones, y `POST` cuando tienes que mandar datos adicionales con la petición.

El método `Route::any()` es usado para hacerlo coincidir con cualquier verbo HTTP. No obstante, te recomendaría que usaras el verbo correcto para la situación en la que estás para que la aplicación sea más transparente.

Volvamos al ejemplo. He aquí para refrescar tu memoria:

```php
<?php

// app/routes.php

Route::get('mi/pagina', function() {
    return '¡Hola mundo!';
});
```

La siguiente porción del código es el primer parámetro del método `get()` (o cualquier otro verbo). Este parámetro define la URI con la que quieres hacer coincidir la URL. En este caso estamos haciendo coincidir `mi/pagina`.

El parámetro final es usado para ofrecer lógica para gestionar la petición. Aquí estamos usando una `Closur`, que también es conocida como una `función anónima`. Las closures son simplemente funciones sin nombre que pueden ser asignadas a variables, como lo haríamos con cualquier valor.

Por ejemplo, el fragmento de arriba podría también ser escrito así:

```php
<?php

// app/routes.php

$logic = function() {
    return '¡Hola mundo!';
}

Route::get('mi/pagina', $logic);
```

Aquí estamos guardando la `Closure` en la variable `$logic` y luego pasándosela al método `Route::get()`.

En esta ocasión, Laravel ejecutará la Closure solo cuando la petición actual esté usando el verbo `GET` de HTTP y la URI coincida con `mi/pagina`. Bajo esas condiciones, la sentencia `return` será procesada y se le pasará la cadena "¡Hola mundo!" al navegador.

Puedes definir tantas rutas como quieras. Por ejemplo:

```php
<?php

// app/routes.php

Route::get('primera/pagina', function() {
    return '¡Primera!';
});

Route::get('segunda/pagina', function() {
    return '¡Segunda!';
});

Route::get('tercera/pagina', function() {
    return '¡Patata!';
});
```

Intenta navegar a las siguientes URLs para ver cómo se comporta nuestra aplicación.

```
http://dominio.com/primera/pagina
http://dominio.com/segunda/pagina
http://dominio.com/tercera/pagina
```

Seguramente quieras asociar la raíz de tu aplicación web. Por ejemplo...

```
http://dominio.com
```

Normalmente, esta será usada para la página de inicio de tu aplicación. Creemos una ruta que coincida con esto.

```
<?php

// app/routes.php

Route::get('/', function() {
    return 'En la Rusia Soviética, la función te define a ti.';
});
```

> Ey,¡espera un segundo! ¿¡No tenemos que poner una barra al final de la URI?!

¡CÁLMATE LECTOR! Te estás volviéndo frenético.

Como ves, una ruta que contiene únicamente una barra invertida, coincidirá con la URL del sitio web, tenga o no tenga una barra al final. La ruta de arriba responderá a cualquiera de estas URLs.

```
http://dominio.com
http://dominio.com/
```

Las URLs pueden tener tantos segmentos (partes entre las barras) como quieras. Puedes usarlo para construir una jerarquía en el sitio.

Considera la siguiente estructura:

```
/
/libros
    /ficcion
    /ciencia
    /romance
/revistas
    /celebridades
    /tecnologia
```

Vale, es un sitio bastante sencillo pero es un gran ejemplo de estructura que a menudo encontrarás en la web. Vamos a recrearlo con rutas de Laravel.

Por claridad, he eliminado el contenido de cada Closure.

```php
<?php

// app/routes.php

// home page
Route::get('/', function() {});


// Rutas para la sección de libros
Route::get('/libros', function() {});
Route::get('/libros/ficcion', function() {});
Route::get('/libros/ciencia', function() {});
Route::get('/libros/romance', function() {});

// Rutas para la sección de revistas
Route::get('/revistas', function() {});
Route::get('/revistas/celebridades', function() {});
Route::get('/revistas/tecnologia', function() {});
```

Con esta colección de rutas, hemos creado fácilmente una jerarquía del sitio. Puede que que te hayas dado cuenta de que hay cierta repetición. Vamos a buscar una forma de minimizar esta repetición y así, dejar de repetirnos.

Parámetros de las rutas

Los parámetros de las rutas pueden ser utilziados para introducir valores de relleno en tus definiciones de ruta. Esto creará un patrón sobre el cual podamos recoger segmentos de la URI y pasarlos al gestor de la lógica de la aplicación.

Esto puede sonar un poco confuso, pero cuando lo veas en acción todo tendrá sentido. Allá vamos.

```php
<?php

// app/routes.php

// routes for the books section
Route::get('/libros', function()
{
    return 'Índice de libros.';
});

Route::get('/libros/{genero}', function($genero)
```

```
{
    return "Libros en la categoría {$genero}.";
});
```

En este ejemplo, hemos eliminado la necesidad de tener todas las rutas por género, incluyendo una variable en la ruta. La variable `{genero}` sacará todo lo que esté detrás de la URI `/libros/`. Esto pasará su valor al parámetro `$genero` de la Closure, que nos permitirá usar la información en nuestra parte de lógica.

Por ejemplo, si quisieras visitar la siguiente URL:

```
http://dominio.com/libros/crimen
```

Serías recibido con esta respuesta de texto:

```
Libros en la categoría crimen.
```

Podríamos eliminar también el requisito de ese parámetro usando uno opcional. Un parámetro puede ser convertido en opcional añadiendo un signo de interrogación (?) al final de su nombre. Por ejemplo:

```
<?php

// app/routes.php

// routes for the books section
Route::get('/libros/{genero?}', function($genero = null)
{
    if ($genero == null) return 'Índice de libros.';
    return "Libros en la categoría {$genero}.";
});
```

Si no se facilita un género en la URL, el valor de `$genero` será igual a `null` y se mostrará el mensaje 'Índice de libros.'.

Si no queremos que el valor del parámetro de una ruta sea `null` por defecto, podemos especificar una alternativa usando una asignación. Por ejemplo:

```php
<?php

// app/routes.php

// routes for the books section
Route::get('/libros/{genero?}', function($genero = 'Crimen')
{
    return "Libros en la categoría {$genero}.";
});
```

Ahora, si visitamos la siguiente URL:

```
http://dominio.com/libros
```

Recibiremos esta respuesta:

```
Libros en la categoría Crimen.
```

Espero que estés empezando a ver cómo se usan las rutas para dirigir tus peticiones en tu sitio y que son un 'pegamento' usado para mantener tu aplicación unida.

Hay mucho más sobre rutas. Antes de que volvamos sobre ellas, vamos a cubrir más sobre lo básico. En el próximo capítulo, echaremos un ojo a los tipos de respuesta que Laravel tiene que ofrecer.

Respuestas

Cuando alguien te hace una pregunta, a menos que no estes de humor o que la pregunta no tenga sentido, lo más probable es que les des una respuesta. Supongo que otra excepción son esas muletillas-preguntas del tipo, '¿Verdad?'. Incluso entonces, deberías darles aunque sea un asentimiento de cabeza... algún tipo de respuesta.

Las peticiones hechas a un servidor web no son distintas del tío que dice '¿Vale?'. Esperarán algo en respuesta. En el capítulo previo, nuestras respuestas tomaban la forma de cadenas devueltas desde las closures, algo así:

```
<?php

// app/routes.php

ruta::get('/', function()
{
    return 'Sí, vale.';
});
```

Aquí devolvemos la cadena "Sí, vale." al navegador. Esta cadena es nuestra respuesta y siempre es devuelta por una closure de una ruta, o una acción de un controlador, lo cual cubriremos más tarde.

Sería justo asumir que queremos enviar algo de `HTML` como nuestra respuesta. Este es el caso a menudo a la hora de desarrollar aplicaciones web.

¿Supongo que podríamos meter HTML en la cadena de respuesta?

```
<?php

// app/routes.php

ruta::get('/', function()
{
    return '<!doctype html>
            <html lang="es">
                <head>
                    <meta charset="UTF-8">
                    <title>¡Genial!</title>
                </head>
```

```
                <body>
                    <p>¡Esta es la respuesta perfecta!</p>
                </body>
            </html>';
});
```

¡Genial! Ahora ves el poder y la gracia de Laravel... solo bromeaba. No queremos servir HTML de esta forma. Hacerlo así sería bastante molesto y, lo que es más importante, ¡estaría mal!

Por suerte para nosotros, Laravel tiene un número de objetos de `Response` (respuesta) que hace que devolver una respuesta sea mucho más fácil. Revisemos la respuesta `View` ¡porque es la más excitante!

Vistas

Las vistas son la parte visual de tu aplicación. Dentro del patrón `Modelo Vista Controlador`, hacen la porción `Vista`. Es por ello por lo que se las llama vistas. No es física de partículas. Hablaremos de la física de partículas en un capítulo posterior.

Una vista es simplemente una plantilla de texto plano que puede ser devuelta al navegador, aunque es muy probable que tus vistas contengan HTML. Las vistas usan la extensión `.php` y normalmente están ubicadas dentro del directorio `app/views`. Esto significa que puedes parsear código `PHP` también en las vistas. Vamos a crear una sencilla vista por ahora.

```
<!-- app/views/simple.php -->

<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>¡Vistas!</title>
</head>
<body>
    <p>¡Oh sí!  ¡VISTAS!</p>
</body>
</html>
```

¡Genial! Un pequeño trozo de HTML guardado en `app/views/simple.php`. Ahora hagamos una vista.

> ¿No acabamos de hacer eso?

¡Jaja! Sí, hiciste una pequeña, pero no me quería referir a 'Hagamos otro archivo de vista.'. En vez de eso, hagamos un nuevo objeto de respuesta de vista con `make()`, basándonos en el archivo que acabamos de crear. Los archivos representan vistas que pueden ser llamadas plantillas. Esto te puede ayudar a distinguir entre el objeto `View` y las plantillas HTML.

```php
<?php

// app/routes.php

ruta::get('/', function()
{
    return View::make('simple');
});
```

> Oh, ¡ya veo! Este es el método `make()`.

Usando el método `View::make()` podemos crear una nueva instancia de un objeto de respuesta `View`. El primer parámetro que pasamos al método `make()` es la plantilla de la vista que queremos usar. Habrás observado que no usamos la ruta completa `app/views/simple.php`. Esto es porque Laravel es inteligente. Por defecto asumirá que tus vistas están ubicadas en `app/views` y buscará un archivo con una extensión apropiada para una vista.

Si miras con detenimiento la `Closure` verás que el objeto `View` que hemos creado está siendo devuelto. Esto es muy importante ya que Laravel servirá el contenido de la closure al navegadore web.

Ve e intenta entrar en `/` en tu aplicación. Genial, ¡esa es la plantilla que escribimos!

Más tarde en el libro aprenderemos cómo hacer diferentes tipos de plantillas que funcionen con la respuesta `View` para hacer tu vida más fácil. Por ahora nos quedaremos con lo básico y aprenderemos los conceptos más fundamentales de Laravel.

Datos de la vista

Poder mostrar plantillas es increíble. Realmente lo es. ¿Qué pasa si queremos usar algunos datos de nuestra Closure? En un capítulo anterior hemos aprendido cómo podemos usar parámetros en las rutas. Quizá queramos poder referirnos a esos parámetros en la Vista. Echemos un vistazo a cómo podríamos hacer eso.

```php
<?php

// app/routes.php

ruta::get('/{ardilla}', function($ardilla)
{
    $data['ardilla'] = $ardilla;

    return View::make('simple', $data);
});
```

En la ruta de arriba, tomamos el parámetro `$ardilla` y lo añadimos a nuestra matriz de datos. Como ves, el segundo parámetro del método `make()`, acepta una matriz que se pasa a la plantilla de la ruta. Normalmente llamo a mi matris `$data` para indicar que son los **datos** de mi plantilla, ¡pero puedes usar el nombre que quieras!

Antes de que empecemos a usar estos datos, hablemos un poco más sobre la matriz de datos. Cuando se pasa una matriz a la vista, las claves de la matriz son extraídas a variables que tienen como nombre la clave de la matriz, y el valor que tenían en esta. Es un poco confuso de explicar sin un ejemplo por lo que permíteme simplificarlo por ti.

Aquí tenemos una matriz para pasarle a `View::make()` como datos de la vista.

```
<?php
array('nombre' => 'Taylor Otwell', 'estado' => 'Gurú del Código')
```

En la plantilla de la vista podremos ahora acceder a esos valores tal que así:

```
<?php echo $nombre;           // devuelve 'Taylor Otwell' ?>

<?php echo $estado;         // devuelve 'Gurú del Código' ?>
```

Por lo que la clave `nombre` de nuestra matriz se convierte en la variable `$nombre` en las plantilla. Puedes almacenar matrices multi-dimensionales tan profundas como quieras en tu matriz de datos para vistas. ¡Experimenta!

Usemos la variable `$ardilla` en la plantilla simple que creamos antes.

```
<!-- app/views/simple.php -->

<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ardillas</title>
</head>
<body>
    <p>¡Me gustaría ser una ardilla <?php echo $ardilla; ?>!</p>
</body>
</html>
```

Ahora si visitamos la URI `/gris`, recibiremos una página que dice '¡Me gustaría ser una ardilla gris!'.

Bueno, fue simple ¿verdad? ¡Usando vistas ya no tienes que devolver cadenas desde tus Closures!

Antes mencioné que hay diferentes tipos de objetos de respuesta. En algunas circunstancias, puede que quieras redirigir el flujo de tu aplicación a otra ruta o porción de lógica. En esas circunstancias, el objeto de respuesta `Redirect` se vuelve útil. Ves, ¡Laravel te tiene cubierto!

Redirecciones

Una redirección es un tipo de objeto de respuesta especial que redirige el flujo de la aplicación a otra ruta. Creemos un par de rutas de ejemplo para que pueda explicarla con más detalle.

```
<?php

// app/routes.php

ruta::get('primera', function()
{
    return 'Primera ruta.';
});

ruta::get('segunda', function()
{
    return 'Segunda ruta.';
});
```

Habiendo añadido las rutas de arriba, recibirás 'Primera ruta.' al visitar `/primera` y 'Segunda ruta.' al visitar `/segunda`.

Excelente, eso es exactamente lo que esperábamos que pasara. Ahora cambiemos el flujo de la aplicación por completo añadiendo una redirección en la closure de la primera ruta.

```
<?php

// app/routes.php

ruta::get('primera', function()
{
    return Redirect::to('segunda');
});

ruta::get('segunda', function()
{
    return 'Segunda ruta.';
});
```

En la primera ruta, estamos ahora devolviendo el resultado del método `Redirect::to()` y pasando la URI de la ubicación del objetivo. En este caso estamos pasando la URI para la segunda ruta, `segunda` como la ubicación.

Si ahora visitas la URI `/primera` verás que eres recibido con el texto `Segunda ruta.`. Esto es porque al recibir el objeto `Redirect` devuelto, Laravel ha cambiado el flujo de nuestra aplicación para ir hacia la ubicación de destino. En este caso el flujo ha sido cambiado hacia la closure de la segunda ruta.

Esto puede ser muy útil cuando alguna condición de algún tipo ha fallado y tienes que redirigir al usuario a una ubicación más útil. He aquí un ejemplo usando el sistema de autenticación (que cubriremos más tarde) que te dará otro caso de uso.

```
<?php

// app/routes.php

ruta.::get('libros', function()
{
    if (Auth::guest()) return Redirect::to('login');

    // Solo mostramos los libros a los usuarios logados
});
```

En este ejemplo, si un usuario que no ha iniciado sesión visita la URI `/libros`, son considerados invitados y serán redirigidos a la página de inicio de sesión.

Más tarde descubrirás una forma mejor de limitar el acceso cuando discutamos los filtros de rutas, así que no leas mucho el ejemplo de arriba. En vez de ello, considera que podemos redirigir al usuario a destinos más útil si no se cumplen unas condiciones.

Respuestas personalizadas

Tanto `View` como `Redirect` heredan del objeto `Response` de Laravel. El objeto response es una instancia de una clase que puede ser devuelto a Laravel como resultado de una ruta en una closure o en una acción de un controlador para permitir al framework servir la respuesta adecuada al navegador.

Los objetos de respuesta normalmente contienen un cuerpo, un código de estado, cabeceras HTTP y otra información útil. Por ejemplo, el segmento del cuerpo de la vista sería su contenido HTML. El código de estado para un objeto `Redirect` será `301`. Laravel usa esta información para construir un resultado que pueda ser devuelto al navegador.

No estamos limitados a usar `View` y `Redirect`. También podemos crear nuestros propios objetos de respuesta que se ajusten a nuestras necesidades. Echemos un vistazo a cómo se puede hacer esto.

```php
<?php

// app/routes.php

ruta::get('respuesta/personalizada', function()
{
    return Response::make('¡Hola mundo!', 200);
});
```

En el ejemplo de arriba, usamos el método `Response::make()` para crear un nuevo objeto de respuesta. El primer parámetro es el contenido o cuerpo de la respuesta y el segundo es el código de estado HTTP que será servido con la respuesta.

Si no has visto códigos de estado HTTP antes, piensa en ellos como en notificaciones de estado para el navegador web que recibe tu página. Por ejemplo, si todo va bien, una respuesta estándar incluirá un estado `200`, que es 'TODO-BIEN' en lenguaje del navegador. Un código de estado `302` indica que ha ocurrido una redirección.

De hecho, apuesto que ya te has encontrado con la infame página `404` de no encontrado. La parte `404` es el código de estado recibido por el navegador cuando el recurso solicitado no pudo ser encontrado.

Dicho simplemente, la respuesta de arriba servirá el contenido '¡Hola mundo!' con un código HTTP de estado `200` para permitir al navegador saber que la petición tuvo éxito.

Las cabeceras HTTP son una colección de parejas clave-valor de datos que representan información útil para el navegador web que está recibiendo la respuesta. Normalmente, son usadas para indicar el formato del resultado o cuánto tiempo debería ser almacenado en caché el contenido. No obstante, podemos definir cabeceras personalizadas como queramos. Echemos un vistazo a cómo podemos establecer algunos valores de cabeceras.

```php
<?php

// app/routes.php

ruta::get('respuesta/personalizada', function()
{
    $response = Response::make('¡Hola mundo!', 200);
    $response->headers->set('nuestra clave', 'nuestro valor');
    return $response;
});
```

Hemos creado un objeto de respuesta de muestra como hicimos en el ejemplo anterior. No obstante, en esta ocasión, hemos añadido una cabecera personalizada.

Se puede acceder a la colección de cabeceras HTTP en la propiedad `headers` del objeto de respuesta.

```php
<?php

var_dump($response->headers);
```

Usando el método `set()` sobre esta colección nos permite añadir nuestras propias cabeceras a la colección usando la clave como primer parámetro y el valor asociado como el segundo.

Una vez que la cabecera ha sido añadida, simplemente devolvemos el objeto de respuesta como hemos hecho anteriormente. El navegador recibirá las cabeceras junto a la respuesta y puede usar esta información como desee.

Pensemos en un ejemplo más útil. Hmmm... en vez de eso vamos a imaginar que queremos que nuestra aplicación sirva respuestas de markdown en vez de HTML. Sabemos que un navegador web no podría decodificar la respuesta markdown, pero puede que tengamos otra aplicación de escritorio que pueda.

Para indicar que el contenido es Markdown y no HTML, modificaremos la cabecera `Content-Type`. La cabecera `Content-Type` es una cabecera común usada por los navegadores para distinguir los distintos tipos de formatos de contenido que les son enviados. ¡No te confundas! Vamos a ver un ejemplo.

```php
<?php

// app/routes.php

ruta::get('respuesta/markdown', function()
{
    $response = Response::make('***algo de texto en negrita***', 200);
    $response->headers->set('Content-Type', 'text/x-markdown');
    return $response;
});
```

Habiendo establecido el cuerpo del objeto de la respuesta con algo de markdown de ejemplo, en este caso ***algo de texto en negrita***, y el la cabecera `Content-Type` al tipo mime del formato de texto plano Markdown, hemos servido una respuesta que puede ser identificada como Markdown.

Nuestra aplicación de escritorio puede hacer una petición a la URI `/markdown/respuesta`, examinar la cabecera `Content-Type`, y al recibir `text/x-markdown` sabrá que tiene que usar un transformador de Markdown para gestionar el cuerpo.

Ahora, ya que somos buenos amigos, voy a compartir un secreto contigo. Acércate. Ven aquí. Vamos a tener una charla. El objeto respuesta no pertenece a Laravel.

> ¿TRAICIÓN? ¿QUÉ LOCURA ES ESTA?

No te pongas nervioso. Como verás, para evitar reinventar la rueda, Laravel ha usado algunos componentes más robustos, que pertenecen al proyecto de Symfony 2. El objeto de respuesta de Laravel hereda la mayoría de su contenido del objeto `Response` que pertenece al componente `Symfony HTTPFoundation`.

Esto significa que si echamos un ojo al API del objeto de respuesta de Symfony, ¡nos daremos cuenta de que tenemos acceso a un montón de métodos adicionales que no están cubiertos en la documentación de Laravel! Ahora que he compartido este secreto, no hay nada que te detenga para convertirte en un maestro de Laravel.

La documentación del objeto de respuesta de Symfony puede ser encontrada aquí[19]. Si miras la página, te darás cuenta de que la clase tiene un atributo llamado `$headers`. ¡Eso es! Esa es la colección de cabeceras que hemos estado usando hace solo un minuto.

Teniendo en cuenta que el objeto de respuesta de Laravel hereda de este, eres libre de usar cualquier otro método que veas en la documentación del API de Symfony. Por ejemplo, echemos un ojo al método `setTtl()`. ¿Qué dice el API?

---

public Response setTtl(**integer $seconds**)

Sets the response's time-to-live for shared caches. This method adjusts the Cache-Control/s-maxage directive.

Parameters:

integer $seconds Number of seconds

Return Value:

Response

Bien, este método parece que establece el tiempo de vida de valores en la caché. No soy ningún experto en este tipo de cosas, pero un tiempo de vida sugiere cuánto tiempo es considerada útil antes de que sea descartada. En esta ocasión, TTL está relacionada con la caché del contenido.

Vamos a darle un valor por probar. Habiendo mirado al método, vemos que acepta un entero representando el tiempo de vida del valor en segundos. Démosle a esta respuesta 60 segundos de vida. Como algún tipo de cruel villano de James Bond.

[19] http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html"ElobjetoderespuestadeSymfony"

```php
<?php

// app/routes.php

ruta.::get('nuestra/respuesta', function()
{
    $response = Response::make('Bond, James Bond.', 200);
    $response->setTtl(60);
    return $response;
});
```

Ahora, cuando nuestra respuesta sea servida, tendrá un tiempo de vida de 60 segundos. Como verás, interrogando al componente ganamos conocimiento sobre opciones avanzadas que podemos usar para modificar la respuesta de nuestra aplicación para ajustarla a nuestras necesidades.

No te sientas abrumado por la cantidad de complejidad que contiene el objeto de respuesta básico. La mayoría de las veces, te valdrá con usar las clases `View` y `Response` de Laravel para servir respuestas HTTP simples. El ejemplo superior simplemente sirve como un buen punto de inicio para usuarios avanzados que busquen ajustar sus aplicaciones para escenarios concretos.

Atajos de respuestas

Laravel es tu amigo. Es verdad... bueno, no es verdad. Laravel es más que un amigo. Te ama. Realmente lo hace. Por ello, te ofrece algunas respuestas ya formadas para hacer tu vida más fácil. Echemos un vistazo a lo que tenemos disponible.

Respuestas JSON

A menudo en nuestra aplicación tendremos algunos datos que querremos servir como JSON. Puede que sea un simple objeto o una matriz de valores.

Larave ofrece un método `Response::json()` que configurará el objeto de respuesta con algunos detalles que son específicos para resultados JSON. Por ejemplo, una cabecera Content-Type apropiada.

Podríamos hacerlo manualmente pero, ¿por qué preocuparse cuando Laravel lo hace por nosotros? Echemos un vistazo al método en acción.

```php
<?php

// app/routes.php

ruta::get('respuesta/markdown', function()
{
    $data = array('iron', 'man', 'mola');
    return Response::json($data);
});
```

Dándole una matriz al método `Response::json()` conseguiremos convertirla en una cadena JSON y establecerla como cuerpo de nuestro nuevo objeto `Response`. Se usan las cabeceras HTTP apropiadas para explicar que lo que estamos devolviendo es JSON. El navegador web recibirá el siguiente cuerpo:

```
["iron","man","mola"]
```

Que es tanto compacto como cierto. ¡Disfruta usando este atajo para crear APIs limpias y funcionales con JSON!

Respuestas de descarga

Servir archivos directamente requiere que se envíen ciertas cabeceras. Por suerte, Laravel se ocupa de ello usando el atajo `Response::download()`. Veámoslo en acción.

```php
<?php

// app/routes.php

ruta::get('archivo/descarga', function()
{
    $file = 'ruta_a_mi_archivo.pdf';
    return Response::download($file);
});
```

Ahora, si navegamos hacia la URI `/archivo/descarga` el navegador iniciará una descarga en vez de mostrar una respuesta. El método `Response::download()` reciba una ruta a un archivo como parámetro y éste será servido cuando se devuelva la respuesta.

También puedes usar un segundo y tercer parámetro opcional para configurar un estado HTTP personalizado y una matriz de cabeceras HTTP. Por ejemplo:

```php
<?php

// app/routes.php

ruta.::get('archivo/descarga', function()
{
    $file = 'ruta_a_mi_archivo.pdf';
    return Response::download($file, 418, array('iron', 'man'));
});
```

Aquí serviremos nuestro archivo con un código de estado HTTP 418 (Soy una tetera) y una cabecera con valor `iron=man`.

Bueno, este capítulo fue mucho más largo de lo que esperé inicialmente, pero estoy seguro de que verás que devolver objetos de respuesta apropiados puede tener más valor que devolver sencillas cadenas.

En el próximo capítulo, veremos los filtros de ruta, que nos permiten proteger nuestra ruta o realizar acciones antes/después de que sean ejecutadas.

Filtros

Recuerdo hace un par de años atrás, cuando Jesse O'brien estaba planeando un evento privado en el que él y sus colegas verían al equipo local de hockey jugar su último partido contra los Pandas de Laravel.

Ahora todos sabemos que los poderosos Pandas de Laravel no podrían ser derrotados por los London Knights, pero Jesse no quería escuchar. Insistía en que podría ser el comienzo del camino al estrellato para los Knights.

El evento estaba planeado para tener lugar en Hoser Hut en el centro de Londres. Un lugar amistoso para cualquier que sea de muy-al-norte de América. (La tierra del sirope de Arce.)

Por desgracia, Hoser hut tenía la mala reputación de no ser muy amistoso para aquellos visitantes fuera de las fronteras. Era un hecho conocido que los Americanos eran lanzados por las ventanas del Hoser Hut normalmente. Por ese motivo, Jesse decidió que necesitaba algún tipo de filtro en la puerta para mantener fuera a los americanos. Por supuesto, el bueno del británico Dayle Rees siempre fue bienvenido al Hoser Hut. Era bienvenido en cualquier lugar.

Jesse empleó a un guardia de seguridad para que se quedara en la puerta del Hoser Hut y pidiera la identificación para ver si el visitante era Canadiense o no.

Como verás, lo que hizo Jesse fue implementar un filtro. Aquellos que pasarn el filtro tendrían acceso al cálido y cómodo Hoser Hut para ver cómo los Pandas de Laravel destruían a los London Knights. Sin embargo, aquellos Americanos que intentaran entrar al bar no cumplirían con los requisitos del filtro y les mostrarían el lado brillante de una bota.

Dejemos a Jessy con su juego y veamos cómo podemos usar los filtros para proteger las rutas de nuestras aplicaciones.

Filtros básicos

Los filtros son ciertos conjuntos de reglas que pueden ser aplicados a una ruta. Pueden ser aplicados antes o después de que se ejecute la lógica de una ruta. No obstante, descubrirás que los filtros que se ejecutan antes son más útiles. Estos filtros pueden alterar el flujo de la aplicación si no se cumplen una serie de reglas o criterios. Es una forma maravillosa de proteger nuestras rutas.

Como siempre, un ejemplo habla por sí solo. Echemos un ojo a un filtro, pero primero, necesitamos algo más. Veamos:

```
<!-- app/views/cumpleanos.php -->

<h1>¡Feliz cumpleaños!</h1>
<p>¡Feliz cumpleaños a Dayle, hurrah!</p>
```

¡Genial! Ahora que tenemos una vista de feliz cumpleaños, podemos crear nuestro primer filtro. Allá vamos:

```
<?php

// app/filters.php

Route::filter('cumpleanos', function()
{
    if (date('d/m') == '12/12') {
        return View::make('cumpleanos');
    }
});
```

Aquí tenemos nuestro primer filtro. Laravel nos da un archivo en `app/filters.php` como ubicación genérica para nuestros filtros, pero podemos ponerlos donde queramos.

Usamos el método `Route::filter()` para crear un nuevo filtro. El primer parámetro es un nombre amistoso que usaremos más tarde para aisgnar el filtro a una ruta. En este ejemplo he nombrado al filtro `cumpleanos`. El segundo parámetro es un callback, que en este ejemplo es una Closure.

El callback es una función que es llamada cuando se ejecuta el filtro. Si devuelve un objeto de tipo respuesta, como aquellos que usamos en la lógica de nuesutra ruta, la respuesta será devuelta y servida en vez del resultado de la lógica de nuestra ruta. Si no hay respuesta desde el filtro, la lógica de la ruta continuará normalmente.

Esto nos da un gran poder, así que ve y practica tu risa maligna. En serio, es algo importante.

> ¡Muahahahah!

Bueno, supongo que había que hacerlo. Como verás, podemos alterar el flujo de la aplicación, o realizar alguna acción y permitir que la lógica de la ruta continúe su ejecución. Por ejemplo, puede que solo queramos mostrar un cierto tipo de contenido en nuestra web a ciertos usuarios. Esto implicaría devolver una respuesta de redirección a otra página. De manera alternativa, podríamos escribir una entrada en un registro cada vez que el filtro fuera ejecutado, para ver qué páginas han sido visitadas. Quizás me esté desviando, echemos un ojo a nuestro filtro de ejemplo.

```php
<?php

// app/filters.php

Route::filter('cumpleanos', function()
{
    if (date('d/m') == '12/12') {
        return View::make('cumpleanos');
    }
});
```

Mirando con detenimiento nuestra closure, podemos ver que tenemos una condición y una respuesta. En nuestro filtro, si la fecha actual es igual a '12/12/84', que es por supuesto la fecha en la que nació la persona más improtante del universo, la closure devolverá la respuesta. Si se devuelve una respuesta desde la Closure, será redirigido a la vista de feliz cumpleaños. De otra forma, la lógica de nuestra ruta seguirá normal.

Por supuesto, antes de que el filtro sea útil, tenemos que añadirlo a una ruta. No obstante, antes de que podamos hacerlo tenemos que cambiar un poco la estructura de la ruta. ¿Recuerdas que te dije que los métodos de las rutas aceptan una closure como segundo parámetro? Bueno, te conté una pequeña mentira otra vez. Lo siento.

Como veras, los métodos de la ruta pueden aceptar una matriz como segundo parámetro. Podemos usar esta matriz para asignar parámetros adicionales a la ruta. Echemos un ojo a cómo se ve una matriz con una matriz como segundo parámetro.

```php
<?php

// app/routes.php

Route::get('/', array(function()
{
        return View::make('hola');
}));
```

Como ves, es bastante similar. Lo que hemos hecho aquí es meter la Closure en la matriz. Funciona como lo hacía antes. De hecho, mientras matenemos la closure en la matriz, podemos incluir otros valores. Es así como vamos a adjuntar nuestro filtro. Comencemos echando un vistazo a la opción de filtro `before` (antes):

```php
<?php

// app/routes.php

Route::get('/', array(
    'before' => 'cumpleanos',
    function()
    {
            return View::make('hola');
    }
));
```

Como puedes ver, hemos creado otra opción en nuestra matriz. El índice `before` le dice al framework que queremos ejecutar nuestro filtro `cumpleanos` antes de que la lógica de la ruta sea ejecutada. El valor `cumpleanos` es igual al apodo que le dimos a nuestro filtro.

Vamos a probar a ejecutar nuestra ruta `/`. Ahora, asumiendo que hoy no es el 12 de Diciembre, verás la página de Laravel. Esto es porque la condición del filtro ha fallado y no se devuelve ninguna respuesta.

Bueno, ahora esperemos hasta el 12 de Diciembre para que podamos ver qué ocurre cuando la condición del filtro pasa y la respuesta sea devuelta.

Solo bromeaba. Cambiemos el filtro para forzar que pase. Podemos cambiar la condición al valor booleano `true`.

```php
<?php

// app/filters.php

Route::filter('cumpleanos', function()
{
    if (true) {
        return View::make('cumpleanos');
    }
});
```

Allá vamos, visita ahora `/` para ver si ha cambiado algo. ¡Hurrah, es mi cumpleaños! Cantemos todos cumpleaños feliz. De hecho, mejor esperamos hasta diciembre. Para que podamos ver que la lógica del filtro ha tenido éxito y se devuelva la vista del cumpleaños feliz.

Podemos asociar un filtro a la opción `after` (después), de una ruta. De esta forma, el filtro será ejecutado después de la lógica de tu ruta. He aquí ejemplo:

```php
<?php

// app/routes.php

Route::get('/', array(
    'after' => 'cumpleanos',
    function()
    {
       return View::make('hola');
    }
));
```

No obstante, tienes que recordar que el filtro de *despues* no puede cambiar la respuesta. Por tanto, nuestro filtro de cumpleaños no tiene mucho sentido *después*. No obstante, puedes dejar algo registrado, o hacer alguna operación de limpieza. ¡Recuerda que está ahí para ti si lo necesitas!

Filtros múltiples

Otra cosa que deberías saber es que puedes aplicar tantos filtros como quieras a una ruta. Echemos un vistazo a algunos ejemplos en acción. Primero, añadamos varios filtros `before`:

```php
<?php

// app/routes.php

Route::get('/', array(
    'before' => 'cumpleanos|navidad',
    function()
    {
       return View::make('hola');
    }
));
```

Aquí hemos añadido los filtros tanto `cumpleanos` como `navidad` en el apartado `before` a la ruta. Dejaré que tu imaginación decida lo que hace el filtro `navidad`, pero asegúrate de que sea algo mágico.

El caracter `|` (tubería) puede ser usado para separar una lista de filtros. Serán ejecutados de izquierda a derecha, y el primero que devuelva una respuesta terminará la petición y esa respuesta será devuelta como resultado.

Si quieres, puedes usar una matriz para ofrecer varios filtros. Puede que eso te parezca más del estilo *php*.

```php
<?php

// app/routes.php

Route::get('/', array(
    'before' => array('cumpleanos', 'navidad'),
    function()
    {
       return View::make('hola');
    }
));
```

Usa lo que sea que se ajusta a tu código, personalmente me gustan las matrices. Si quieres, puedes asignar también un filtro `before` y otro `after` a la vez, tal que así:

```php
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'cumpleanos',
    'after'     => 'navidad',
    function()
    {
       return View::make('hola');
    }
));
```

Naturalmente, el filtro `before` se ejecutará primero, luego la lógica de la ruta y, finalmente, el filtro `after`.

¿Crees que has acabado ya con los filtros? ¡No te emociones!

Parámetros de los filtros

Al igual que las funciones de PHP, los filtros pueden aceptar parámetros. Esta es una gran forma de evitar repeticiones y que permite mayor flexibilidad. Vamos allá con un ejemplo, como siempre.

```php
<?php

// app/filters.php

// before

Route::filter('test', function($ruta, $peticion)
{

});

// after

Route::filter('test', function($ruta, $peticion, $respuesta)
{

});
```

Espera, ¿por qué hay dos filtros?

¡Bien observado! Bueno, realmente son el mismo filtro, pero aun así, tu pregunta sigue siendo válida. Como verás, Laravel ofrece diferentes conjuntos de parámetros para los parámetros de `before` y `after`. Te habrás dado cuenta de que ambos reciben las variables `$ruta` y `$peticion`. Puedes llamarlas como quieras, pero las llamé así por un motivo.

Si usaras `var_dump()` con el primer parámetro, verías que es una instancia de `Illuminate\Routing\Route`. Recordarás que `Illuminate` es el nombre en clave usado para los componentes de Laravel 4. La clase `Route` representa una ruta usada por la capa de enrutamiento. Esta instancia representa la ruta actual que está siendo ejecutada. Inteligente, ¿verdad? La instancia de `Route` es enogme, anímate y hazlo si no crees a este astuto galés. Podrías interrogarlo por la información que contiene, o incluso alterar algunos de sus valores para manipular el framework. No obstante, este es un tema avanzado, y está fuera del alcance de este capítulo por lo que vamos a centrarnos en el siguiente parámetro.

Como puede que hayas adivinado, el siguiente parámetro es una instancia del objeto de la petición actual. La instancia de `Illuminate\Http\Request` representa el estado de la petición que ha sido enviada a tu servidor web. Contiene información sobre la URL, datos pasados con la petición y algo de información adicional.

El filtro `after` recibe un parámetro adicional, una instancia del objeto de respuesta que ha sido devuelto desde la ruta sobre la que el filtro actúa. Esta instancia es la que es servida como respuesta de la petición actual.

Bien, esos parámetros que nos dio Laravel pueden ser útiles para usuarios avanzados del framework pero, ¿no sería genial si pudieramos pasar nuestros propios parámetros a los filtros de nuestras rutas? Echemos un vistazo a cómo podemos hacerlo.

Primero, tenemos que añadir una variable de relleno a la Closure de nuestro filtro, debería venir después de los que Laravel nos da, tal que así:

```
<?php

// app/filters.php

Route::filter('cumpleanos', function($routa, $peticion, $fecha)
{
    if (date('d/m') == $fecha) {
        return View::make('cumpleanos');
    }
});
```

Nuestro filtro `cumpleanos` ha sido alterado para aceptar un parámetro `$fecha`. Si la fecha actual coincide con la fecha facilitada, el filtro del cumpleaños es ejecutado.

Ahora, todo lo que necesitamos saber es cómo pasar los parámetros a los filtros. Echemos un vistazo.

```
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'cumpleanos:12/12',
    function()
    {
       return View::make('hola');
    }
));
```

El parámetro que pasamos a nuestro filtro, viene tras el signo de dos puntos : cuando lo asignamos a la ruta. Ve y prúebalo, cambia la fecha a la del día de hoy y mira como se activa el filtro.

Si queremos facilitar más datos, tenemos que usar más variables de relleno en la closure. Será algo así.

```php
<?php

// app/filters.php

Route::filter('cumpleanos', function($routa, $peticion, $primera, $segunda, $terc\
era)
{
    return "{$primera} - {$segunda} - {$tercera}";
});
```

Podemos aceptar tantos parámtros como queramos. Para usar múltiples parámetros, primero tenemos que añadir dos puntos : entre el nombre del filtro y sus parámetros. Los parámetros en sí deben ser separados por coma ,. He aquí un ejemplo:

```php
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'cumpleanos:foo,bar,baz',
    function()
    {
       return View::make('hola');
    }
));
```

Los valores `foo`, `bar` y `baz` serán pasados a las variables y añadidos al filtro. Merece la pena destacar que al igual que las funciones, los parámetros de los filtros pueden tener valores por defectos, haciéndolos opcionales. He aquí un ejemplo:

```php
<?php

// app/filter.php

Route::filter('ejemplo', function($ruta, $peticion, $opcional = '¡Sip!')
{
    return $opcional;
});
```

Pasa el parámetro opcional o no lo hagas. Es cosa tuya, ¡es tu framework!

Eres libre de usar tantos parámetros como quieras para hacer que tus filtros sean más eficientes. Aprovecha esta gran característica.

Filtros de clases

Las Closures son geniales. Son muy convenientes, y funcionan bien en mis ejemplos. No obstante, están limitadas en las que son escritas. No podemos instanciarlas, esto las hace difícil de probar.

Por este motivo, cualquier característica de Laravel que necesite una Closure tendrá también una alternativa. Una clase PHP. Echemos un ojo a cómo podemos usar una clase para representar nuestros filtros.

Antes de que hagamos la clase, necesitamos algo en que guardarlo. Creemos una carpeta nueva en `/app` llamada `filtros`, y actualiza tu archivo `composer.json` para incluir la nueva carpeta.

```
"autoload": {
    "classmap": [
        "app/commands",
        "app/controllers",
        "app/models",
        "app/filters",
        "app/database/migrations",
        "app/database/seeds",
        "app/tests/TestCase.php"
    ]
}
```

Ahora creemos una nueva clase para nuestro filtro de cumpleaños. Allá vamos:

```
<?php

// app/filters/cumpleanos.php

class filtroCumpleanos
{
    public function filter($ruta, $peticion, $fecha)
    {
        if (date('d/m') == $fecha) {
            return View::make('cumpleanos');
        }
    }
}
```

He llamado a mi clase `filtroCumpleanos`, no tienes que añadirle el prefijo `filtro`, pero me gusta hacerlo. Lo que necesitas sin embargo, es el método `filter()`. Funciona como la Closure. De hecho,

dado que funciona como una Closure no tengo que explicarlo otra vez. En vez de ello, echemos un vistazo a cómo podemos enganchar un filtro a una ruta.

Primero, tenemos que crear un alias del filtro. Una vez más, usaremos el método `Route::filter()`. No obstante, esta vez pasaremos una cadena en vez de una closure como segundo parámetro. Tal que así:

```
<?php

// app/routes.php

Route::filter('cumpleanos', 'filtroCumpleanos');
```

El segundo parámetro es una cadena que identifica la clase de filtro a usar. Si la clase de filtro está ubicada en un espacio de nombre, añádelo también.

Ahora que hemos creado un alias de filtro, podemos añadirlo a la ruta como hicimos antes.

```
<?php

// app/routes.php

Route::get('/', array(
    'before'    => 'cumpleanos',
    function()
    {
       return View::make('hola');
    }
));
```

Recuerda que tendrás que ejecutar `composer dump-autoload` antes de que Composer, y Laravel, puedan encontrar nuestra clase de filtro.

Si pretendes probar tu código por completo, escribir filtros como clases es la mejor forma de hacerlo. Descubrirás más sobre las pruebas en un capítulo posterior.

Filtros globales

Si echas un vistazo dentro de `/app/filters.php` descubrirás dos filtros extraños. Estos son filtros globales que son ejecutados antes y después de qcada petición de tu aplicación.

```php
<?php

// app/filters.php

App::before(function($request)
{
        //
});


App::after(function($request, $response)
{
        //
});
```

Funcionan exactamente igual que los filtros normales, excepto que aplican a todas las rutas por defecto. esto significa que no hay necesidad de añadirlos a los índices `before` y `after` de las matrices de nuestras rutas.

Filtros por defecto

En `app/filters.php` hay algunos filtros que ya han sido creados para ti. Echemos un ojo a los tres primeros.

```php
<?php

// app/filters.php

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


Route::filter('auth.basic', function()
{
        return Auth::basic();
});

Route::filter('guest', function()
{
```

```
18      if (Auth::check()) return Redirect::to('/');
19  });
```

Todos estos filtros están relacionados con la capa de autenticación de Laravel. Pueden ser usados para restringir acceso a las rutas a usuarios que estén, o no, logados en la aplicación web.

En un capítulo posterior, echaremos un ojo más de cerca a la capa de autenticación y el contenido de esos filtros tendrá más sentido. Por ahora, ¡ya sabes que esos filtros están esperándote!

El cuarto filtro es un filtro para evitar la falsificación de petición en sitios cruzados y es tal que así:

```
1   <?php
2
3   // app/filters.php
4
5   Route::filter('csrf', function()
6   {
7           if (Session::token() != Input::get('_token'))
8           {
9                   throw new Illuminate\Session\TokenMismatchException;
10          }
11  });
```

Puedes asociarla a las rutas que quieras proteger de ser usadas desde otro origen que no sea tu propia aplicación. Esta es una medida de seguridad muy útil que se usa principalmente para proteger rutas que son el objetivo de formularios o envío de datos.

Tómate la libertad de usar los filtros que Laravel ofrece, están ahí para ahorrarte tiempo.

Patrones de filtros

¿No quieres asociar un filtro manualmente a todas tus rutas? No, no puedo quejarme. A veces los dedos se cansan, estoy escribiendo un libro, lo sé. Intentemos buscar una forma de ahorrar a tus falanges algo de esfuerzo. He aquí un patrón de filtro.

El patrón de filtros te permite hacer coincidir un filtro `before` a un número de rutas facilitando un patrón de enrutado con un asterisco. Veámoslo en acción.

```php
<?php

// app/routes.php

Route::when('perfil/*', 'cumpleanos');
```

El método `Route::when()` de arriba, ejecutará el filtro `cumpleanos` en todas las rutas cuya URI comiencen con `perfil/`. El asterisco del primer parámetro actúa como comodín. Es una gran forma de adjuntar un filtro `before` a varias rutas a la vez.

Controladores

Creando controladores

En el capítulo de enrutado básico, vimos cómo enlazar rutas a closures, pequeñas cajas de lógica, que conforman la estructura de nuestra aplicación. Las closures son una forma bonita y rápida de escribir una aplicación y, personalmente, creo que se ven genial con los ejemplos de código del libro. No obstante, la opción preferida para la lógica de la aplicación es el Controlador.

El Controlador es una clase usada para alojar lógica de enrutado. Normalmente, el Controlador contendrá un número de métodos públicos conocidos como acciones. Puedes pensar en las acciones como la alternativa directa a las closures que usamos en el capítulo anterior, son muy similares tanto en apariencia como en funcionalidad.

No me gusta explicar lo que es algo, sin mostrar primero un ejemplo. Así que vamos a ponernos manos a la obra y miremos un controlador. He aquí uno que acabo de hacer:

```
1  <?php
2
3  // app/controllers/ArticuloController.php
4
5  class ArticuloController extends BaseController
6  {
7      public function mostrarIndex()
8      {
9          return View::make('index');
10     }
11
12     public function mostrarUnico($idArticulo)
13     {
14         return View::make('unico');
15     }
16 }
```

¡Aquí está nuestro controlador! Bonito y sencillo. Este ejemplo queda bien ocn un blog o algún otro CMS. Idealmente, un blog tendrá una página para ver una lista de todos los artículos, y otra página para mostrar un artículo único en detalle. Ambas actividades están relacionadas con el concepto de un Artículo, lo cual significa que tendría sentido agrupar esta lógica. Es esto por lo que la lógica está almacenada en `ArticuloController`.

Puedes llamar al Controlador como quieras. Mientras este extienda a `BaseController` o `Controller`, Laravel sabrá lo que intentas hacer. No obstante, añadirle el sufijo `Controller` es algo estándar que los programadores web usan. Si piensas trabajar con otros, los estándares pueden ser realmente útiles.

Nuestro controlador ha sido creado en el directorio `app/controllers` que Laravel ha creado por nosotros. Este directorio está siendo cargado desde composer.json por defecto. Si `app/controllers` no se ajusta a tu flujo de trabajo, puedes poner el archivo donde quieras. No obstante, asegúrate de que la clase puede ser cargada automáticamente por Composer. Para más información sobre este tema, por favor vuelve a lo básico sobre Composer.

Los métodos de la clase de nuestro Controlador son los que realmente contienen nuestra lógica. Una vez más, puedes nombrarlos como quieras, pero personalmente me gusta usar la palabra `mostrar` como prefijo si el resultado es que muestran una página web. Estos métodos ***deben*** ser públicos para que Laravel pueda enrutarlos. Puedes añadir métodos privados adicionales para añadir abstracción, pero no podrás lanzar rutas sobre ellos. De hecho, hay un lugar mejor para ese tipo de código sobre el cual aprenderemos en el capítulo de los modelos.

Echemos un vistiazo de cerca a nuestra priemra acción que, en este ejemplo, será usada para mostrar una lista de artículos de un blog.

```
1  public function mostrarIndex()
2  {
3      return View::make('index');
4  }
```

Uhhmm, ¿no te resulta extrañamente familiar? Vamos a compararlo con una closure de una ruta que podría ser usada para lograr el mismo efecto.

```
1  Route::get('index', function()
2  {
3      return View::make('index');
4  });
```

Como puedes ver, la función interna es casi idéntica. La única diferencia es que la acción del controlador tiene un nombre, y la closure es anónima. De hecho, la acción del controlador puede contener cualquier código que tenga la closure. Esto implica que todo lo que hemos aprendido es aun aplicable. Vaya... ¡Si fueran diferentes podría haber vendido otro libro sobre controladores!

Hay otra diferencia entre los dos fragmentos de código. En el capítulo de enrutado básico, vimos cómo rutar una URI a una pieza de lógica contenida en una Closure. En el ejemplo anterior la URI `/index` será enrutada a la lógica de nuestra aplicación. No obstante, la acción de nuestro Controlador no menciona una URI para nada. ¿Cómo sabe Laravel que tiene que redirigir sus rutas a nuestro controlador? Echemos un ojo a cómo funciona el enrutado con controladores y esperemos encontrar una respuesta a nuestra pregunta.

Enrutado de Controlador

Los Controladores son bonitos y limpios y ofrecen una forma sencilla de agrupar lógica común. No obstante, no son útiles a menos que nuestros usuarios puedan alcanzar esa lógica. por suerte, el método para enlazar una URI a un Controlador es similar al método que usamos para las Closures. Echemos un ojo.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('index', 'ArticuloController@mostrarIndex');
```

Para enlazar una URI a un Controlador, tenemos que definir una nueva ruta en el archivo `/app/routes.php`. Estamos usando el mismo método `Route::get()` que utilizamos al usar Closures. No obstante, el segundo parámetro es totalmente diferente. En esta ocasión tenemos una cadena.

La cadena consiste en dos secciones que están separadas por una arroba (`@`). Echemos un ojo otra vez al Controlador que creamos en la última sección.

```
1  <?php
2
3  // app/controllers/ArticuloController.php
4
5  class ArticuloController extends BaseController
6  {
7      public function mostrarIndex()
8      {
9          return View::make('index');
10     }
11
12     public function mostrarUnico($idArticulo)
13     {
14         return View::make('unico');
15     }
16 }
```

Así que podemos ver por el ejemplo que el nombre de la clase es `ArticuloController`, y la acción a la que queremos enrutar es llamada `mostrarIndex`. Pongámosla juntas, con una arroba (`@`) en el medio.

```
ArticuloController@mostrarIndex
```

Realmente es así de simple. Ahora podemos usar cualquier método que hemos descubierto en el capítulo básico sobre rutas, y apuntarlas a los controladores. Por ejemplo, he aquí una acción de un controlador que responderá a una petición HTTP de tipo `POST`.

```
<?php

// app/routes.php

Route::post('articulo/nuevo', 'ArticuloController@nuevoArticulo');
```

Estoy seguro de que sois inteligentes, ¿habéis comprado el libro no? Ahora puedes ver que la ruta de arriba responderá a peticiones `POST` a la URI `/articulo/nuevo` y que será gestionada por la acción `nuevoArticulo()` en `ArticuloController`.

He aquí algo bonito a saber. Puedes añadir espacios de nombre a tu controlador y Laravel no se quejará. Solamente tienes que asegurarte de que incluyes el espacio de nombre en la declaración de tu ruta, ¡y todo funcionará bien! Veámoslo en acción.

```
<?php

// app/controllers/Articulo.php

namespace Blog\Controller;

use View;
use BaseController;

class Articulo extends BaseController
{
    public function mostrarIndex()
    {
        return View::make('index');
    }
}
```

Aquí tenemos un Controlador similar al que usamos en el otro ejemplo. En esta ocasión, está contenido en el espacio de nombre `Blog\Controller`. Teniendo en cuenta que está ubicado en la sección `Controller` del espacio de nombre, he omitido el sufijo `Controller` del nombre de la clase. Esta es mi experiencia personal, te dejaré decidir si quieres mantenerla o no.

Veamos cómo puede ser enrutado un controlador en un espacio de nombre. ¡Probablemente ya lo hayas averiguado!

```php
<?php

// app/routes.php

Route::post('index', 'Blog\Controller\Articulo@mostrarIndex');
```

Es igual que antes, solo que en esta ocasión el nombre del controlador ha sido prefijado con su espacio de nombre. Como ves, ¡los espacios de nombre no tienen que complicar las cosas! Incluso puedes almacenar tu controlador con espacio de nombre en un directorio anidado, o en cualquier sitio con un esquema de carga PSR-0. A Laravel no le importa, mientras que Composer sepa dónde encontrar tu clase, Laravel podrá usarla.

Controladores RESTful

Laravel ofrece soluciones, ya lo sabemos. También opciones, los Controladores RESTful son el ejemplo principal de esto.

Sabemos que podemos definir verbos de peticiones HTTP que queramos asociar usando los métodos de rutas. Esto es realmente conveniente a la hora de enrutar a Closures. No obstante, cuando enrutamos a Controladores puede que quieras mantener la definición del verbo de la petición junto a la lógica de tu aplicación. Bueno, las buenas noticias es que Laravel te ofrece esta configuración alternativa.

Alteremos nuestro Controlador un poco, ¿vale?

```php
<?php

// app/controllers/Articulo.php

namespace Blog\Controller;

use View;
use BaseController;

class Articulo extends BaseController
{
    public function getCrear()
    {
        return View::make('crear');
    }

    public function postCrear()
```

```
18      {
19          // Gestionar el formulario de creacion
20      }
21  }
```

Aquí tenemos nuestro Controlador de `Articulo` una vez más. La intención de las acciones son facilitar una forma de crear y gestionar la creación de un nuevo artículo del blog. Descubrirás que los nombres de las acciones tienen como prefijo `get` y `post`. Esos son nuestros verbos HTTP.

Por tanto, en nuestro ejemplo podrías pensar que hemos representado puntos para las siguientes URLs:

```
1  GET  /crear
2  POST /crear
```

También podrías preguntarte cómo enrutamos hacia nuestro controlador RESTful. Como verás, usar métodos de verbos en la clase `Route` no tendría mucho sentido. Bueno, di adios a enrutar acciones individuales. Echemos un vistazo a otro método de enrutado.

```
1  <?php
2
3  // app/routes.php
4
5  Route::controller('Articulo', 'Blog\Controller\Articulo');
```

Este único método enrutará todas las acciones representada en nuestro controlador RESTful. Miremos con más detenimiento al método.

El primer parámetro es la URL base. Normalmente, el enrutado RESTful es usado para representar un objeto por lo que, en la mayoría de las ocasiones, la URL base será el nombre de ese objeto. Puedes pensar en ello como un prefijo de las acciones que hemos creado en nuestro controlador RESTful.

El segundo parámetro ya te será familiar. Es el controlador al que intentamos enrutar. Una vez más, Laravel aceptará felizmente un controlador con un espacio de nombre como objetivo de la ruta por lo que eres libre de organizar tus Controladores como quieras.

Como puedes ver, usar este método de enrutado a tus controladores, te ofrece una ventaja distinta sobre el método de enrutado original. Con este método, solo tendrás que facilitar una única entrada de enrutado para el controlador, en vez de enrutar cualquier acción de forma independiente.

Blade

En este capítulo, aprenderemos cómo dominar Blade. Lo necesitarás. Para reclamar el lugar que te corresponde como artesano de PHP, tendrás que desafiar y derrotar al Alto Señor Otwell en combate armado.

Es un rito de iniciación. Solo entonces podrás tomar el lugar que te merece en el concilio de Laravel y ganar un hueco en la mesa de bebidas de Phil Sturgeon.

Una vez al mes, nosotros los miembros del Concilio, nos dirigimos al campo de batalla de PHP, montamos sobre nuestros temidos Pandas de montar de Laravel, para batallar con los programadores de otros frameworks. Si quieres cabalgar hacia la batalla con nosotros y pelear por el honor de Laravel **tienes** que aprender a dominar Blade.

- **N. del T**: Blade en inglés significa hoja de un cuchillo o espada. El autor intenta hacer una broma con ello. *

Bueno, tener es una palabra fuerte supongo. Quiero decir, podrías también dominar las plantillas Blade. No es tan extravagante y no tiene que ver con cabalgar fieros pandas. No obstante es bastante útil y quizá más apropiado para los programadores que las batallas sangrientas.

Está bien, cambiaré de tema. Hablemos sobre las plantillas de Blade.

Puede que te estés preguntando por el nombre 'Blade'. Cuando escribí el capítulo, decidí preguntarle a Taylor. Parece ser que la plataforma de desarrollo web de .NET tiene una herramienta de plantillas llamada 'Razor', de la cual se deriva mucha de la sintaxis de Blade. Razor… blade… razor blade. Eso es. Nada divertido aquí, lo siento. :(

- **N. del T**: Razorblade significa hoja de afeitar en inglés. *

De hecho olvida lo que te acabo de decir, reinventemos la historia. Solo entre nosotros. El lenguaje de plantillas fue nombrado por el alter ego de Taylor, 'Blade', durante sus días de caza de Vampiros. Esa es la historia real.

Bien, hagamos una plantilla.

Creando plantillas

Lo sé, lo sé. Ya te he enseñado a crear vistas, ¿verdad? Son muy útiles para separar el aspecto visual de tu aplicación de su lógica pero, no significa que no se puedan mejorar.

El problema con las plantillas PHP estándar es que tenemos que insertar esas feas etiquetas de PHP en ellas para usar los datos que nuestras porciones de lógica han facilitado. Quedan fuera de lugar en nuestras bonitas plantillas HTML. ¡Las destrozan! Me enfurece tanto... Podría... Podría. No, déjame estar un momento.

> Erm...

Está bien, mi furia se ha calmado. Creemos una plantilla de Blade para quitar a PHP del medio. Para comenzar, tendremos que hacer un nuevo archivo. Las plantillas de Blade están en la misma ubicación que nuestros archivos de vistas estándar. La única diferencia es que usan la extensión `.blade.php` en vez de únicamente `.php`.

Creemos una plantilla sencilla.

```
<!-- app/views/ejemplo.blade.php -->

<h1>Querido señor Otwell</h1>
<p>Por el presente le reto a un duelo por el honor de Laravel.</p>

<?php echo $ardilla; ?>
```

Aquí tenemos nuestra plantilla de blade, parece muy similar a lo que hemos visto hasta ahora, ¿verdad? Eso es porque Blade se encarga de ejecutar el archivo como PHP. ¿Ves nuestra `$ardilla`? Cada archivo tiene que tener una ardilla. Vale, no es cierto, pero muestra que PHP funciona como antes.

Podemos mostrar esto usando la misma sintaxis que usaríamos normalmente para una vista normal. Puede que hayas asumido que necesitarías pasarle `ejemplo.blade` al método `View::make()`, pero eso sería incorrecto.

```
<?php

// app/routes.php

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

¿Ves? Laravel sabe lo que es una plantilla Blade, y cómo buscar una. Por este motivo, la sentencia `View::make()` no ha cambiado para nada. ¡Qué conveniente!

No obstante, Blade tiene algunos trucos propios. Echemos un ojo al primero.

Salida PHP

Muchas de las plantillas harán uso de `echo` de datos facilitados por la parte lógica de tu aplicación. Normalmente sería algo así:

```
<!-- app/views/ejemplo.blade.php -->

<p><?php echo $taylorElCazadorDeVampiros; ?></p>
```

No es que sea muy complicado, pero podría ser mejorado. Veamos cómo podemos mostrar lo mismo con plantillas Blade.

```
<!-- app/views/ejemplo.blade.php -->

<p>{{ $taylorElCazadorDeVampiros }}</p>
```

Todo lo que esté rodeado por `{{` dobles llaves `}}` es transformado en un `echo` por Blade cuando se procesa la plantilla. Esta sintaxis es mucho más limpia y más fácil de aprender.

Teniendo en cuenta que las directivas de Blade son traducidas directamente a PHP, podemos usar cualquier código PHP dentro de las llaves, incluyendo métodos.

```
<!-- app/views/ejemplo.blade.php -->

<p>{{ date('d/m/y') }}</p>
```

Ni siquiera tienes que usar el punto y coma de cierre. Laravel lo hace por ti.

A veces querrás protegerte de escapar un valor que pongas en la plantilla. Puede que te sean familiares métodos como `strip_tags()` y `htmlentities()` para esto. ¿Por qué? Bueno, considera la salida de esta plantilla.

```
<!-- app/views/ejemplo.blade.php -->

<p>{{ '<script>alert("¡BACON APESTOSO!");</script>' }}</p>
```

¡vaya pieza de código tan desagradable! Eso provocaría que se inyectara JavaScript en la página y aparecería una ventana emergente con el texto '¡BACON APESTOSO!'. Malo ¿Por qué? Esos programadores Ruby siempre intentan romper nuestros sitios web.

Tenemos el poder de protegernos de esto. Si usáramos `{{{` tres llaves `}}}` en vez de `{{` dos `}}` nuestra salida será escapada, convirtiendo el texto en entidades HTML por lo que el código JavaScript se mostraría como texto. ¡Inofensivo!

Veámoslo en acción.

```
<!-- app/views/ejemplo.blade.php -->

<p>{{{ '<script>alert("¡BACON APESTOSO!");</script>' }}}</p>
```

Todo lo que hemos cambiado ha sido el número de llaves alrededor de nuestro código JavaScript. Veamos el código fuente de la página para ver cómo se ve.

```
<!-- app/views/ejemplo.blade.php -->

<p>&lt;script&gt;alert(&quot;¡BACON APESTOSO!&quot;);&lt;/script&gt;</p>
```

Como puedes ver, las etiquetas HTML y algunos otros caracteres han sido reemplazados por las entidades HTML equivalentes. ¡Nuestro sitio web está salvado!

¡Sigamos!

Estructuras de control

PHP tiene ciertas estructuras de control. Sentencias `if`, `while`, bucles `foreach` y `for`. ¡Si no las has visto antes significa que este libro no es para ti!

En las plantillas lo más probable es que uses la sintaxis alternativa de las estructuras de control usando los dos puntos `:`. Si tus sentencias `if` se ven así:

```
<!-- app/views/ejemplo.blade.php -->

<? if ($algo) : ?>
    <p>¡Algo es verdadero!</p>
<? else : ?>
    <p>¡Algo es falso!</p>
<? endif; ?>
```

De nuevo, sacan del paso, pero no son muy divertidas de escribir. Te ralentizarán, pero por suerte para ti, ¡Blade viene al rescate!

He aquí cómo luce el fragmento de arriba en una plantilla Blade.

```
<!-- app/views/ejemplo.blade.php -->

@if ($algo)
    <p>¡Algo es verdadero!</p>
@else
    <p>¡Algo es falso!</p>
@endif
```

Eso es mucho más limpio, ¿verdad? Echemos un ojo a lo que hemos quitado. Para empezar, las etiquetas `<?` y `?>` de apertura y cierre. Probablemente las más complicadas de escribir.

También quitamos los dos puntos `:` y el punto y coma `;`. ¡No necesitamos esos gasta espacio en nuestras plantillas!

Por último, hemos hecho una adición a la sintaxis habitual. Hemos añadido como prefijo a nuestras líneas de sentencia de control un símbolo de arroba `@`. De hecho, todas las estructuras de control de Blade y métodos de ayuda tienen este símbolo para que el compilador de plantillas sepa cómo gestionarlos.

Añadamos un `elseif` a la mezcla para aumentar el ejemplo.

```
<!-- app/views/ejemplo.blade.php -->

@if ($algo == 'Panda Rojo')
    <p>¡Algo es rojo, blanco y marrón!</p>
@elseif ($algo == 'Panda Gigante')
    <p>¡Ago es blanco y negro!</p>
@else
    <p>Algo podría ser una ardilla.</p>
@endif
```

Hemos añadido otra sentencia a la mezcla, siguiendo la misma regla de eliminar etiquetas PHP, dos puntos y pnuto y coma, y añadiendo el símbolo `@`. Todo funciona perfectamente.

He aquí un desafío. Intenta imaginarte un bucle `foreach` de PHP representado con sintaxis Blade. Cierra tus ojos, imagínatelo. Concéntrate... ¡concéntrate!

¿Se parecía a esto?

```
<!-- app/views/ejemplo.blade.php -->

      c~~p ,---------.
 ,---'oo  )           \
( O O                  )/
 `=^='                 /
       \    ,     .   /
       \\  |-----'|  /
       ||__|    |_|__|
```

¿No? Bien, porque es un hipopótamo. No obstante, si se parecía al siguiente fragmento, te daré una galleta.

```
<!-- app/views/ejemplo.blade.php -->

@foreach ($muchasCosas as $cosa)
    <p>{{ $cosa }}</p>
@endforeach
```

¡Disfruta de tu galleta! Como puedes ver, usamos la sintaxis `{{ echo }}` de Blade para mostrar el valor del bucle. Un bucle `for` es exactamente como te imaginas. He aquí un ejemplo de referencia.

```
<!-- app/views/ejemplo.blade.php -->

@for ($i = 0; $i < 999; $i++)
    <p>¡Ni siquiera {{ $i }} pandas rojos, son suficientes!</p>
@endfor
```

Simple y exactamente lo que esperabas. El bucle `while` sigue las mismas reglas, pero voy a mostrar un rápido ejemplo para que sea una buena referencia para un capítulo posterior.

```
<!-- app/views/ejemplo.blade.php -->

@while (esBonita($kieraKnightly))
    <p>Este buce probablemente no acabe nunca.</p>
@endwhile
```

Bien, así que eres un maestro condicional ahora. Nada puede contigo, ¿verdad colega? Ni siquiera la condición `unless`.

> Err Dayle, Ruby tiene eso. No sé si PHP ti...

OK, me has pillado. PHP no tiene condición `unless`. No obstante, Blade tiene una función que lo permite. Veamos un ejemplo.

```
<!-- app/views/ejemplo.blade.php -->

@unless (elMundoAcabe())
    <p>Sigue sonriendo.</p>
@endunless
```

`unless` es justo lo contrario a una sentencia `if`. Una sentencia `if` evalúa si una condición es verdadera, y luego ejecuta algo de lógica. No obstante, la sentencia `unless` se ejecutará únicamente si la sentencia evalúa en `false`. Puedes pensar en ella como en una estructura de control para pesimistas.

Plantillas

Blade incluye algunos otros métodos de ayuda para hacer que tus plantillas sean más sencillas de construir y gestionar. Aun así, no escribe las vistas por ti, quizá podríamos añadirla a la lista de tareas de Laravel 5…

Hasta que eso ocurra, tendremos que escribir nuestras propias plantillas. Eso no significa que tengamos que ponerlo todo en un único archivo.

Con PHP, puedes usar `include()` para incluir un archivo en el actual, ejecutando sus contenidos. Podrías hacer lo mismo con las vistas para separarlas por archivos para mejorar su organización. Laravel nos ayuda con ello, usando el método `@include()` de Blade para importar una vista en otra, convirtiendo su contenido como plantilla de Blade si es necesario. Echemos un vistazo a un ejemplo de esto en acción.

He aquí el archivo `cabecera.blade.php` que contiene la cabecera de nuestra página, y posíblemente otras páginas.

```
<!-- app/views/cabecera.blade.php -->

<h1>¿Cuándo hace bacon el Narval?</h1>
```

He aquí la plantilla del pie de página.

```
<!-- app/views/pie.blade.php -->

<small>Información facilitada basándonos en la investigación del 3 de Mayo de 201\
3.</small>
```

Ahora, aquí tenemos nuestra plantilla principal. La que se muestra por una Closure de una ruta o una acción de un controlador.

```
<!-- app/views/ejemplo.blade.php -->

<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Narvales</title>
</head>
<body>
    @include('header')
    <p>El Narval seguramente haga el Bacon a media noche, ¡mi buen señor!</p>
    @include('footer')
</body>
</html>
```

Como puedes ver, los métodos de la plantilla `ejemplo.blade.php`, están trayendo el contenido de nuestra cabecera y pie de página. El método `include` toma el nombre de la vista como parámetro, en el mismo formato corto que `View::make()` que usamos anteriormente. Echemos un ojo al documento resultante.

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Narvales</title>
</head>
<body>
    <h1>¿Cuándo hace bacon el Narval?</h1>
    <p>El Narval seguramente haga el Bacon a media noche, ¡mi buen señor!</p>
    <small>Información facilitada basándonos en la investigación del 3 de Mayo de\
 2013.</small>
</body>
</html>
```

Nuestras plantillas incluídas han sido... bueno, incluídas en la página. Esto hace nuestra cabecera y pie de páginas más reusables y evita que nos repitamos. Podemos incluirlas en otras páginas para evitar tener que repetir contenido y haciendo que solo tengamos que editar contenido en un único archivo. Hay una mejor forma de hacer esto no obstante así que ¡sigue leyendo!

Herencia de plantillas

Blade ofrece una forma de construir plantillas que puede ayudar a la herencia. Mucha gente lo encuentra algo confuso aunque es una característica bastante bonita. Voy a intentar simplificarlo lo

mejor que pueda y espero que descubras que el arte de crear plantillas es una experiencia placentera.

Antes que nada, pensemos sobre plantillas. Hay algunas partes de una página web que no cambian mucho en las páginas. Esas son las etiquetas que tienen que estar presentes para visualizar cualquier página. Podemos llamarla nuestro código de relleno si quieres. He aquí un ejemplo:

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
</body>
</html>
```

Vamos a usar esta plantilla para todas nuestras páginas. ¿Por qué no se lo decimos a Laravel? Digamos que es una plantilla de Blade. Para hacerlo, solo tenemos que definir algunas áreas en las que podamos insertar contenido, en Laravel podemos llamar a esas áreas 'secciones'. He aquí cómo las definimos

```
<!-- app/views/layouts/base.blade.php -->

<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title></title>
    @section('cabecera')
        <link rel="stylesheet" href="estilo.css" />
    @show
</head>
<body>
    @yield('body')
</body>
</html>
```

Primero, hemos creados una plantilla con dos secciones. Primero miremos a la que tiene el cuerpo, esa es fácil. Se ve así:

```
@yield('cuerpo')
```

Esta sentencia la dice a Blade que cree una sección aquí, para poder rellenarla de contenido más tarde. Le damos el apodo `cuerpo` para que podamos referirnos a ella más tarde.

La otra sección se ve así:

```
@section('cabecera')
    <link rel="stylesheet" href="style.css" />
@show
```

Esta es muy similar a la sección `yield` excepto que puedes dar algo de contenido por defecto. En el ejemplo de arriba, el contenido entre las etiquetas `@section` y `@show` será mostrado a menos que una plantilla hija decida sobreescribirla.

Entonces, ¿a qué me refiero como plantilla hija? Bueno, como siempre veamos un ejemplo.

```
<!-- app/views/inicio.blade.php -->

@extends('layouts.base')

@section('cuerpo')
    <h1>¡Hurrah!</h1>
    <p>¡Tenemos una plantilla!</p>
@stop
```

Bien, vayamos poco a poco. Primero tenemos la función `extends`:

```
@extends('layouts.base')
```

Esto le dice a Blade qué plantilla vamos a usar para mostrar nuestro contenido. El nombre pasamos a la función debería parecerse a esos que les pasamos a `View::make()`, así que, en esta situación nos estamos refiriendo al archivo `base.blade.php` en el directorio `layouts` dentro de `app/views`. Recuerda que un punto (`.`) representa un separador de directorios al usar vistas.

En Laravel 3 esta función era llamada `@layout()`, pero ha sido renombrada para estar más en línea con otros motores de plantillas como Twig, de Symfony. ¡Cuidado programadores de Laravel!

Ahora que sabemos qué plantilla estamos usando, es hora de rellenar los huecos. Podemos usar la función `@section` de Blade para inyectar contenido en secciones de la plantilla padre. Se ve tal que así:

```
@section('cuerpo')
    <h1>¡Hurrah!</h1>
    <p>¡Tenemso una plantilla!</p>
@stop
```

A la función `@section` le pasamos el apodo que le dimos a nuestra sección dentro de nuestra plantilla padre. ¿Recuerdas? La llamamos `cuerpo`. Todo lo que esté contenido dentro de `@section` y `@stop` será inyectado en la plantilla padre, en la que está `@yield('cuerpo')`.

Creemos una ruta para ver esto en acción. Para mostrar una plantilla, solo necesitamos añadir una respuesta `View::make()` para mostrar la plantilla hija, tal que así:

```php
<?php

// app/routes.php

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

Ahora, si visitamos / y miramos el código fuente, veremos que la página es así:

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="estilo.css" />
</head>
<body>
    <h1>¡Hurrah!</h1>
    <p>¡Tenemos una plantilla!</p>
</body>
</html>
```

Vale, el formato puede ser un poco diferente, pero el contenido debería ser el mismo. Nuestra sección ha sido inyectada en nuestra plantilla padre. Teniendo en cuenta que no hemos sobreescrito el contenido de la sección `cabecera`, el valor por defecto ha sido insertado.

Como ves, podríamos tener tantas plantillas hijas como queramos, heredando de esta plantilla padre. Esto nos ahorra el esfuerzo de tener que repetir el código de relleno.

Cambiemos la plantilla hija un poco para darnos algo de contenido para la 'cabecera'. Tal que así:

```
<!-- app/views/home.blade.php -->

@extends('layouts.base')

@section('cabecera')
    <link rel="stylesheet" href="otro.css" />
@stop

@section('cuerpo')
    <h1>¡Hurrah!</h1>
```

```
    <p>¡Tenemos una plantilla!</p>
@stop
```

Como puedes imaginar, la sección de la cabecera ha inyectado nuestro archivo CSS adicional, y el código de nuestra página es ahora tal que así:

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="otro.css" />
</head>
<body>
    <h1>¡Hurrah!</h1>
    <p>We have a template!</p>
</body>
</html>
```

¿Recuerdas que la sección de cabecera tenía algo de contenido por defecto entre `@section` y `@show`? Bueno, podríamos querido añadir algo al contenido en vez de reemplazarlo. Para ello podemos usar la etiqueta `@parent`. Modifiquemos nuestra plantilla hija para usarla, tal que así:

```
<!-- app/views/inicio.blade.php -->

@extends('layouts.base')

@section('cabecera')
    @parent
    <link rel="stylesheet" href="otro.css" />
@stop

@section('cuerpo')
    <h1>¡Hurrah!</h1>
    <p>¡Tenemos una plantilla!</p>
@stop
```

La etiqueta `@parent` le indica a Blade que reemplace esto, con el contenido por defecto dentro de la sección padre. Esta sentencia puede parecer un poco confusa pero es realmente simple. Echemos un vistazo a cómo se ve el código fuente ahora.

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title></title>
    <link rel="stylesheet" href="estilo.css" />
    <link rel="stylesheet" href="otro.css" />
</head>
<body>
    <h1>¡Hurrah!</h1>
    <p>¡Tenemos una plantilla!</p>
</body>
</html>
```

¿Ves? Nuestra etiqueta `@parent` fue reemplazada con el contenido por defecto de la sección del padre. Puedes usar este método añadir nuevas entradas a un menú o archivos de recursos adicionales.

Puedes tener tantas cadenas de herencia en una plantilla Blade como quieras, el siguiente ejemplo es totalmente correcto.

```
<!-- app/views/primero.blade.php -->

<p>Primero</p>
@yield('mensaje')
@yield('final')

<!-- app/views/segundo.blade.php -->
@extends('primero')

@section('mensaje')
    <p>Segundo</p>
    @yield('mensaje')
@stop

<!-- app/views/tercero.blade.php -->
@extends('segundo')

@section('mensaje')
    @parent
    <p>Tercero</p>
    @yield('mensaje')
@stop

```

```
<!-- app/views/cuarto.blade.php -->
@extends('tercero')

@section('mensaje')
    @parent
    <p>Cuarto</p>
@stop

@section('final')
    <p>Quinto</p>
@stop
```

¡Vaya locura! Intenta seguir la cadena de herencia para ver cómo se construye la salida. Sería mejor empezar las plantillas hijas y seguir a cada padre. Si mostráramos la vista `cuarto`, este sería el resultado.

```
<p>Primero</p>
<p>Segundo</p>
<p>Tercero</p>
<p>Cuarto</p>
<p>Quinto</p>
```

Para simplificarlo:

Cuarto extiende Tercero, que extiende Segundo, que extiende Primero, que es la plantilla básica.

Quizá te hayas dado cuenta de que la sección `final` de la plantilla básica tiene contenido que es ofrecido por la plantilla `cuarto`. Esto significa que puedes proveer contenido para una sección desde cualquier 'capa'. Como puedes ver, Blade es **muy** flexible.

Comentarios

Como probablemente sepas, HTML tiene sus propios métodos para incluir comentarios. Son así:

```
<!-- Este es un encantador comentario HTML. -->
```

Estás en lo cierto comentario, eres encantador, pero por desgracia también te muestras junto al resto del código fuente. No queremos realmente que la gente lea la información que está pensada para ser para programadores.

Al contrario que los comentarios de HTML, los de PHP no se muestran cuando se pre-procesa la página. Esto significa que no serán mostrados al ver el código. Podríamos incluir comentarios PHP en nuestros archivos así:

```
<?php // Este es un comentario secreto de PHP. ?>
```

Seguro, ahora nuestro contenido está oculto. Aunque es un poco feo, ¿verdad? No hay cabida para la fealdad en nuestras utópicas plantillas Blade. Usemos comentarios de Blade, se compilan en comentarios PHP directamente.

```
{{-- Este es un comentario bonito y secreto de Blade. --}}
```

Usa comentarios de Blade cuando quieras poner tus notas en tus vistas para que solo las vean los programadores.

Rutas avanzadas

Oh ya veo, has vuelto a por más. ¿Las rutas básicas no eran lo suficientemente buenas para ti? ¿No eres un poco avaricioso? No temas aventurero de Laravel, tengo algo de postre para ti.

En el capítulo de los filtros has aprendido sobre cómo podemos pasar una matriz como segundo parámetro a los métodos de enrutado para poder incluir más información en la definición de nuestra ruta. Tal que así:

```
<?php

// app/routes.php

Route::get('/', array(
    'before' => 'filtrosexy',
    function() {
        return View::make('hola');
    }
));
```

En este ejemplo, estamos usando la sintaxis de matriz para incluir información sobre qué filtro queremos aplicar a la ruta. Aunque no acaba aquí ya que puedes hacer mucho más con esta matriz. Veamos lo que tenemos disponible.

Rutas nombradas

Las URIs son bonitas y chachis. Ciertamente ayudan cuando se trata de darle estructura al sitio, pero cuando tienes un sitio más complejo pueden volverse un poco más largas. Estoy seguro de que no quiere tener que recordar cada URI de tu sitio ya que se convertirá en algo aburrido rápidamente.

Por fortuna, Laravel ofrece la habilidad de nombrar las rutas para aliviar parte del aburrimiento. Como verás, podemos darle a nuestras rutas un apodo, que se parece a esto:

```php
<?php

// app/routes.php

Route::get('/mi/larga/ruta/calendario', array(
    'as' => 'calendario',
    function() {
        return View::make('calendario');
    }
));
```

Usando el índice `as` en la matriz de nuestra ruta, podemos asignarle un apodo a una ruta. Intenta mantenerlo corto aunque descriptivo. Como verás, Laravel tiene un varios métodos que te ayudan a generar enlaces al recurso servido por tu aplicación, y muchos de ellos tienen la habilidad de usar rutas nombradas. No voy a cubrirlos todos aquí, hay un capítulo que los cubre todos en detalle, no obstante he aquí un sencillo ejemplo.

```php
// app/views/ejemplo.blade.php

{{ route('calendario') }}
```

Este sencillo método mostrará la URL de la ruta nombrada cuyo apodo pases. En este caso, devolverá `http://localhost/mi/larga/ruta/calendario`. Las llaves son únicamente para mostrar el contenido dentro de una plantilla de Blade. Aun te acuerdas de Blade, ¿verdad? ¡Eso espero!

Así que, ¿cómo de útil es esto? Bueno, como dije antes, no tienes que recordar largas URLs nunca más. No obstante, puede que tengas un súper-cerebro. Recordar URLs puede ser algo trivial para ti. Aquí hay otra ventaja que me gustaría compartir.

Imaginemos por un segundo que tienes un número de vistas con enlaces a una cierta ruta. Si los enlaces de las rutas fueran introducidos manualmente, y cambiaras la URL de la ruta, tendrías que cambiar también todas las URLs. En una aplicación larga, esto podría resultar en una enorme pérdida de tu tiempo y, asumámoslo, ahora eres un programador Laravel. Tu tiempo vale mucho dinero.

Si usamos el método `route()` y luego decidimos cambiar nuestra URL, ya no tendremos que modificar todos los enlaces. Todos serán resueltos por su apodo. Siempre intento nombrar a mis rutas si puedo, me ahorra mucho tiempo si necesito reestructurarlas.

¿Recuerdas el objeto de respuesta `Redirect`? Pues bien, puedes usar el método `route` sobre él para redirigir a una ruta con nombre. Por ejemplo:

```php
<?php

return new Redirect::route('calendario');
```

Además, si quieres obtener el apodo de la ruta actual, puedes usar el conveniente método `currentRouteName()` en la clase `Route`. Tal que así:

```php
<?php

// app/routes.php

$actual = Route::currentRouteName();
```

Asegúrate de recordar que todas estas características avanzadas están disponibles para los Controladores también así como para las Closures enrutadas. Para enrutar un controlador, simplemente añade el parámetro `uses` a la matriz de enrutado junto a la pareja de acción del controlador.

```php
<?php

// app/routes.php

Route::get('/mi/larga/ruta/calendario', array(
    'as' => 'calendario',
    'uses' => 'CalendarioController@mostrarCalendario'
));
```

Fácil, ¿verdad? Ahora echemos un ojo a cómo podemos hacer que nuestras rutas sean más seguras.

Rutas seguras

Puede que quieras que tus rutas respondan a URLs HTTP seguras que puedan gestionar datos confidenciales. Las URLs de HTTPS están sobre el protocolo SSL o TLS para aumentar la seguridad cuando lo necesites. He aquí cómo puedes permitir que tus rutas funcionen con este protocolo.

```php
<?php

// app/routes.php

Route::get('contenido/secreto', array(
    'https',
    function () {
        return '¡Ardilla secreta!';
    }
));
```

Añadiendo HTTPS al índice de nuestra matriz de enrutado, nuestra ruta ahora responderá a peticiones realizadas usando el protocolo HTTPS.

Limitaciones de parámetros

En el capítulo de enrutado básico, descubrimos cómo podríamos usar parámetros desde nuestra estructura de URLs en la lógica de nuestra aplicación. Para una Closure enrutada, se ve así:

```php
<?php

// app/routes.php

Route::get('salvar/{princesa}', function($princesa)
{
    return "Lo siento, {$princesa} está en otro castillo. :(";
});
```

Bueno, estoy seguro de no haber escuchado nunca a una princesa que se llame '!1337f15h'. Me suena más a un jugador de Counter Strike. No queremos que nuestras rutas respondan a princesas de mentira, así que porqué no intentamos validar nuestros parámetros para asegurarnos de que solo consistan en letras.

Vamos allá con un ejemplo de esto en acción.

```php
<?php

// app/routes.php

Route::get('salvar/{princesa}', function($princesa)
{
    return "Lo siento, la {$princesa} está en otro castillo. :(";
})->where('princesa', '[A-Za-z]+');
```

En el ejemplo de arriba, encadenamos el método `where()` al final de la definición de nuestra ruta. El método `where` acepta el nombre del parámetro como primer parámetro, y una expresión regular como segundo parámetro.

No voy a intentar cubrir las expresiones regulares en detalle. El tema es largo, no en serio, es increíblemente largo. Podría tener un libro en sí mismo. Dicho con sencillez, la expresión regular de arriba se asegura de que el nombre de la princesa esté formado por letras en mayúsculas o minúsculas y de que al menos tenga una letra.

Si el parámetro no satisface nuestra expresión regular, la ruta no coincidirá. El enrutador continuará buscando con qué hacerlo coincidir.

Puedes asociar tantas condiciones como quieras a tu ruta. Mira este ejemplo:

```php
<?php

// app/routes.php

Route::get('salvar/{princesa}/{unicornio}', function($princesa, $unicornio)
{
    return "{$princesa} le encanta {$unicornio}";
})->where('princesa', '[A-Za-z]+')
  ->where('unicornio', '[0-9]+');
```

El parámetro `unicornio` ha sido validado contra uno o más números, porque como sabemos, los unicornios siempre tienen nombres numéricos. Como mi buen amigo 3240012.

Grupos de rutas

¿Recuerdas cómo pudimos poner condiciones a nuestras rutas en el capítulo de los Filtros? Era útil, ¿verdad? Sería una vergüenza tener que asociar el mismo filtro a muchas definiciones de rutas.

¿No sería genial si pudiéramos encapsular nuestras rutas y aplicarle un filtro al contenedor? Bueno, puede que ya te hayas dado cuenta pero he aquí un ejemplo que hace exactamente eso.

```php
<?php

// app/routes.php

Route::group(array('before' => 'solobrogramadores'), function()
{

    // Primera ruta
    Route::get('/primera', function() {
        return '¡Tío!';
    });

    // Segunda ruta
    Route::get('/segunda', function() {
        return '¡Tíoooooo!';
    });

    // Tercera ruta
    Route::get('/tercera', function() {
        return 'Ven a mi.';
    });

});
```

En el ejemplo de arriba, estamos usando el método `group()` sobre el objeto `Route`. El primer parámetro es una matriz. Funciona igual que las que hemos estado usando en nuestros métodos de enrutado. Puede aceptar filtros, índices seguros y muchos de los otros filtros de enrutado. El segundo parámetro debería ser una Closure.

Cuando defines rutas adicionales en esta Closure, las rutas heredan las propiedades del grupo. Las tres rutas dentro del grupo están protegidas por el grupo `before` llamado `solobrogramadores`.

Ahora, podemos usar los filtros de matriz que hemos descubierto antes en el capítulo, sobre los grupos. Pero también podemos usar algunas nuevas características que son específicas a grupos de rutas. Echemos un vistazo a esas nuevas características.

Prefijo de rutas

Si muchas de tus rutas comparten una estructura común de URLs, podrías usar un prefijo para evitar repeticiones.

Mira este ejemplo:

```php
<?php

// app/routes.php

Route::group(array('prefix' => 'libros'), function()
{

    // Primera ruta
    Route::get('/primero', function() {
        return 'El color de la magia';
    });

    // Segunda ruta
    Route::get('/segundo', function() {
        return 'El segador';
    });

    // Tercera ruta
    Route::get('/tercero function() {
        return 'Lores y damas';
    });

});
```

Usando la opción del prefijo en el grupo de rutas, podemos especificar un prefijo común a todas las URIs definidas dentro de éste. Por ejemplo, las tres rutas de arriba ahora están accesibles en las siguientes URLs.

```
/libros/primero

/libros/segundo

/libros/tercero
```

Usa prefijos de rutas para evitar repeticiones en tus rutas, y para agruparlas con propósito organizativo o estructural.

Enrutado de dominio

Las URIs no son la única forma de diferenciar una ruta. El dominio también puede cambiar. Por ejemplo, las siguientes URLs pueden diferenciar recursos distintos.

```
http://miaplicacion.dev/mi/ruta

http://otra.miaplicacion.dev/mi/ruta

http://tercera.miaplicacion.dev/mi/ruta
```

En los ejemplos de arriba, puedes ver que el subdominio es diferente. Descubramos cómo podemos usar el enrutado de dominios para servir contenido diferente desde diferentes dominios.

He aquí un ejemplo de un enrutado basado en dominios:

```
<?php

// app/routes.php

Route::group(array('domain' => 'miaplicacion.dev'), function()
{
    Route::get('mi/ruta', function() {
        return 'Hola desde myapp.dev!';
    });
});

Route::group(array('domain' => 'otra.miaplicacion.dev'), function()
{
    Route::get('mi/ruta', function() {
        return 'Hola desde otra.miaplicacion.dev!';
    });
});

Route::group(array('domain' => 'tercera.miaplicacion.dev'), function()
{
    Route::get('mi/ruta', function() {
        return 'Hola desde tercera.miaplicacion.dev!';
    });
});
```

Añadiendo el índice `domain` a la matriz de enrutado en grupo, podemos facilitar el nombre de un host, que *debe* coincidir con el actual dominio para que cualquiera de las rutas sea ejecutada.

El nombre del host puede ser un subdominio o un subdominio completamente diferente. Mientras el servidor web esté configurado para servir peticiones desde cada host, Laravel podrá hacerse cargo.

Eso no es todo sobre la rutas basadas en dominios. También podemos capturar porciones del nombre del host para usarlos como parámetros, al igual que hicimos con el enrutado basado en URIs. He aquí un ejemplo de esto en acción.

```php
<?php

// app/routes.php

Route::group(array('domain' => '{usuario}.miaplicacion.dev'), function()
{
    Route::get('perfil/{pagina}', function($usuario, $pagina) {
        // ...
    });
});
```

Puedes usar una variable para el parámetro de un dominio dentro del índice `domain` usando { llaves }, al igual que con los parámetros de las URI. El valor del parámetro será pasado antes que ningún parámetro de la ruta sea mostrado al grupo.

Por ejemplo, si visitáramos la siguiente URL:

```
http://taylor.miaplicacion.dev/perfil/avatar
```

El primer valor `$usuario` que sería pasado a la Closure interna, sería `taylor`, y el segundo valor `$pagina` sería `avatar`.

Usando una combinación de subdominios, variables y parámetros de enrutado podrías añadir el nombre de usuario de los usuarios de tu aplicación.

Generación de URLs

Tu aplicación web toca rutas y URLs. Después de todo, son las que dirigen a tus usuarios a tus páginas. Al final del día, servir páginas es lo que cualquier aplicación web debe hacer.

Tus usuarios puede que no estén interesados mucho tiempo en tu aplicación si solo sirves una página, y si pretendes moverlos por tu web o por tu aplicación web, tendrás que usar una característica crítica de la web. ¿Qué característica te oigo preguntar? Bueno, ¡hiper-enlaces!

Para poder construir hiper-enlaces, necesitamos crear URLs a nuestra aplicación. Podríamos hacerlo a mano, pero Laravel nos puede ahorrar esfuerzos ofreciéndonos algunos métodos de ayuda para asistirnos en la creación de URLs. Echemos un vistazo a esto.

La URL actual

Obtener la URL actual en Laravel es fácil. Simplemente usa el método `URL::current()`. Vamos a crear una sencilla ruta para probarlo.

```
<?php

// app/routes.php

Route::get('/url/actual', function()
{
    return URL::current();
});
```

Ahora, si visitamos la url `/url/actual`, recibiremos el siguiente mensaje de respuesta.

```
http://miaplicacion.dev/url/actual
```

Bueno, fue sencillo, ¿no? Echemos un vistazo a `URL::full()` ahora, verás que devuelve la URL actual.

> Ehem... ¿no acabamos de hacer eso?

Bueno, es un poco diferente, vamos a probar la última ruta una vez más, pero esta vez vamos a incluir algunos datos como parámetros GET.

```
http://miaplicacion.dev/url/actual?foo=bar
```

Como verás, el resultado de `URL::current()` elimina los datos de la petición adicionales, tal que así:

```
http://miaplicacion.dev/url/actual
```

El método `URL::full()` es un poco diferente. Vamos a modificar nuestra ruta existente para usarlo. Tal que así:

```
<?php

// app/routes.php

Route::get('/url/actual', function()
{
    return URL::full();
});
```

Ahora probemos la URL `/url/actual?foo=bar` de nuevo. Esta vez obtenemos el siguiente resultado:

```
http://miaplicacion.dev/url/actual?foo=bar
```

Como ves, el método `URL::full()` también incluye los datos adicionales de la petición.

Esta siguiente no es realmente una forma de obtener la URL actual, pero creo que, definitivamente, tiene su lugar en este sub apartado. Como verás, es un método de obtener la URL anterior que se encuentra en la cabecera `referer` de la petición.

He usado una trampa con una respuesta de redirección para mostrar la salida. Echa un vistazo al siguiente ejemplo.

```
<?php

// app/routes.php

Route::get('primera', function()
{
    // Redirige a la segunda ruta
    return Redirect::to('segunda');
});

Route::get('segunda', function()
{
    return URL::previous();
});
```

Nuestra primera ruta, redirige a la segunda. La segunda ruta mostrará la URL de la anterior petición usando método `URL::previous()`.

Visitemos la URI `/primera` para ver lo que ocurre.

Puede que hayas visto la notificación de redirección durante apenas un segundo, pero con suerte habrás recibido el siguiente mensaje de respuesta:

```
http://miaplicacion.dev/primera
```

Como verás, tras la redirección, el método `URL::previous` nos da la URL de la petición anterior, que en este caso es la URL a la primera ruta. ¡Es tan simple como eso!

Generando URLs del framework

Esta sección trata sobre generar URLs que nos ayuden a navegar por las diferentes rutas o páginas de nuestro sitio o aplicación.

Comencemos generando URLs específicas a una URI. Podemos hacerlo usando el método `URL::to()`. Así:

```
<?php

// app/routes.php

Route::get('ejemplo', function()
{
    return URL::to('otra/ruta');
});
```

La respuesta que recibimos al visitar `/ejemplo` es así:

```
http://miaplicacion.dev/otra/ruta
```

Como puedes ver, Laravel ha creado una URL a la ruta que le hemos pedido. Fíjate que `otra/ruta` no existe, pero podemos enlazarla si queremos. Asegúrate de que te acuerdas de esto al generar enlaces a URIs.

Puedes especificar parámetros adicionales al método `URL::to()` en forma de matriz. Estos parámetros será añadidos al final de la ruta. He aquí un ejemplo:

```php
<?php

// app/routes.php

Route::get('ejemplo', function()
{
    return URL::to('otra/ruta', array('foo', 'bar'));
});
```

El resultado tendrá la siguiente forma.

```
http://miaplicacion.dev/otra/ruta/foo/bar
```

Si quieres que tus URLs generadas usen el protocolo HTTPS, tienes dos opciones. La primera opción es pasar `true` como tercer parámetro del método `URL::to()`, así:

```php
URL::to('otra/ruta', array('foo', 'bar'), true);
```

No obstante, si no quieres usar parámetros, tendrás que pasar una matriz vacía, o `null` como segundo parámetro. En vez de ello, es más efectivo usar el método `URL::secure()` que es mucho más descriptivo. Tal que así:

```php
URL::secure('otra/ruta');
```

Una vez más, puedes pasar una matriz de parámetros de ruta como segundo parámetro al método `URL::secure()`, así:

```php
URL::secure('otra/ruta', array('foo', 'bar'));
```

Echemos un vistazo al siguiente método de generación. ¿Recuerdas que descubrimos cómo darle apodos a nuestras rutas en el capítulo de enrutado avanzado? Las rutas con nombre son así:

```php
<?php

// app/routes.php

Route::get('el/mejor/vengador', array('as' => 'ironman', function()
{
    return 'Tony Stark';
}));

Route::get('ejemplo', function()
{
    return URL::route('ironman');
});
```

Si visitamos la ruta `/ejemplo`, recibimos la siguiente respuesta.

```
http://miaplicacion.dev/el/mejor/vengador
```

Laravel ha cogido nuestro apodo de ruta, y ha encontrado la URI asociada. Si cambiáramos la URI, la salida cambiaría también. Esto es muy útil ya que evitas tener que cambiar una única URI para muchas vistas.

Al igual que el método `URL::to()`, el método `URL::route()` puede aceptar una matriz de parámetros como segundo parámetro del método. No solo eso, si no que lo insertará en el orden correcto de la URI. Echemos un vistazo a esto en acción.

```php
<?php

// app/routes.php

Route::get('el/{primero}/vengador/{segundo}', array(
    'as' => 'ironman',
    function($primero, $segundo) {
        return "Tony Stark, el {$primero} vengador {$segundo}.";
    }
));

Route::get('ejemplo', function()
{
    return URL::route('ironman', array('mejor', 'conocido'));
});
```

Si visitamos la siguiente URL...

```
http://miaplicacion.dev/ejemplo
```

... Laravel rellenará los espacios blancos en el orden correcto, con los parámetros facilitados. La siguiente URL será mostrada como respuesta.

```
http://miaplicacion.dev/el/mejor/vengador/conocido
```

Hay un método final de enrutado de este tipo que necesitas saber, y es cómo enrutar a acciones de controladores. De hecho, este debería ser realmente simple ya que sigue el mismo patrón que el método `URL::route()`. Echemos un ojo a un ejemplo.

```
<?php

// app/routes.php

// Nuestro controlador
class Stark extends BaseController
{
    public function tony()
    {
        return 'Puedes contar conmigo, para hacerme feliz.';
    }
}

// Ruta al controlador Stark
Route::get('yo/soy/iron/man', 'Stark@tony');

Route::get('ejemplo', function()
{
    return URL::action('Stark@tony');
});
```

En este ejemplo, creamos un nuevo controlador llamado `Stark` con una acción `tony/()`. Creamos una nueva ruta para la acción del controlador. Luego creamos una ruta de ejemplo que devuelve el valor del método `URL::action()`. El primer parámetro de este método es la combinación de clase y acción para la que queremos obtener una URL. El formato de este parámetro es idéntico a lo que usamos para enrutar controladores.

Si visitamos la URI `/ejemplo`, recibiremos la siguiente respuesta.

```
http://miaplicacion.dev/yo/soy/iron/man
```

Laravel ha identificado la URL para la pareja controlador-acción que hemos pedido y la ha entregado como respuesta. Al igual que con los otros métodos, podemos pasar una matriz de parámetros como segundo parámetro al método `URL::action()`. Veámoslo en acción.

```
<?php

// app/routes.php

// Nuestro Controlador
class Stark extends BaseController
{
    public function tony($queEsTony)
    {
        // ...
    }
}

// Ruta al controlador Stark
Route::get('tony/el/genio/{primero}', 'Stark@tony');

Route::get('example', function()
{
    return URL::action('Stark@tony', array('narcisista'));
});
```

Al igual que en el último ejemplo, ofrecemos una matriz con un único parámetro al método `URL::action()`, y Laravel construye la URL al controlador, con el parámetro en la ubicación correcta.

La URL que recibimos es así.

```
http://miaplicacion.dev/tony/el/genio/narcisista
```

Bueno, eso es todo sobre generación de URLs. Siento haber sido un poco repetitivo, pero espero que sea un buen capítulo de referencia.

URLs de recursos

Las URLs a recursos como imágenes, archivos CSS y JavaScript necesitan ser gestionadas de forma diferente. La mayoría de vosotros estaréis usando URLs bonitas con Laravel. Este es el acto de

reescribir una URL para eliminar el controlador `index.php` del frente, y nuestras URLs estarán más optimizadas en cuanto a SEO se refiere.

No obstante, en algunas situaciones puede que no quieras usar URLs bonitas. Si intentaras enlazar a un recurso usando los métodos mencionados en los anteriores sub-capítulos, la porción `index.php` de la URL sería añadida y los enlaces no funcionarían.

Incluso con URLs bonitas, no queremos enlazar a nuestros recursos usando URLs relativas porque nuestros segmentos de rutas estarán confundidos con la estructura de carpetas.

Como siempre, Laravel, y Taylor, van un paso por delante nuestra. Hay métodos para generar URLs absolutas a nuestros recursos. Echemos un vistazo a algunos de estos métodos.

Primero, tenemos `URL::asset()`, echemos un vistazo a esto en acción. El primer parámetro al método es la ruta relativa al recurso.

```
<?php

// app/routes.php

Route::get('ejemplo', function()
{
    return URL::asset('img/logo.png');
});
```

Ahora, si visitamos la URI `/ejemplo`, recibiremos la siguiente respuesta.

```
http://miaplicacion.dev/img/logo.png
```

Laravel ha creado una ruta absoluta al recurso por nosotros. Si queremos usar el protocolo seguro HTTPS para hacer referencia a nuestros recursos, podemos pasar `true` como segundo parámetro al método `URL::asset()`, así:

```
<?php

// app/routes.php

Route::get('ejemplo', function()
{
    return URL::asset('img/logo.png', true);
});
```

Ahora, recibimos la siguiente respuesta de nuestra URI `/ejemplo`.

```
https://miaplicacion.dev/img/logo.png
```

¡Genial! Laravel también ofrece un método mucho más descriptivo de generar URLs a recursos seguros. Simplemente usa el método `URL::secureAsset()` y pasa la ruta relativa a tu recurso.

```php
<?php

// app/routes.php

Route::get('example', function()
{
    return URL::secureAsset('img/logo.png');
});
```

La respuesta de esta ruta es la misma que la del método anterior.

```
https://miaplicacion.dev/img/logo.png
```

Atajos de generación

Los métodos que hemos mencionado están disponibles para que los uses en tus vistas. Ve y disfruta de todas las características que ofrecen.

No obstante, es una buena práctica que las vistas sean cortas y bonitas. También te quita algo de estrés de los dedos. Es esto por lo que Laravel ofrece algunos atajos de algunos de los métodos que tiene la clase `URL`. Echemos un vistazo a lo que hay disponible.

Primero tenemos la función `url()` acepta los mismos parámetros que el método `URL::to()`, así que no lo cubriré de nuevo. He aquí un ejemplo del método en acción.

```
<!-- app/views/ejemplo.blade.php -->

<a href="{{ url('mi/ruta', array('foo', 'bar'), true) }}">Mi ruta</a>
```

Ahora, si echamos un ojo al enlace que ha generado la ruta, veremos lo siguiente.

```
<a href="https://miaplicacion.dev/my/route/foo/bar">Mi ruta</a>
```

La URL ha sido creada de la misma forma que con el método `URL::to()`. Como antes, hay un método corto que podemos usar para generar una URL segura. Es tal que así:

```
<!-- app/views/ejemplo.blade.php -->

<a href="{{ secure_url('mi/ruta', array('foo', 'bar')) }}">Mi ruta</a>
```

La función `secure_url()` los mismos parámetros que el método `URL::secure()`. El primero es la ruta, y el segundo es una matriz de parámetros de la ruta que puedan ser añadidos.

La función `route()` es un atajo del método `URL::route()`, y puede ser usada para generar URLs de rutas con nombre. Se ve así:

```
<!-- app/views/ejemplo.blade.php -->

<a href="{{ route('miruta') }}">Mi ruta</a>
```

Como puedes haber imaginado, hay un atajo para el tercer método de generación de URLs de rutas. La función `action()` puede ser usada como atajo del método `URL::action()`, y puede ser usado para generar enlaces a acciones de controladores.

```
<!-- app/views/ejemplo.blade.php -->

<a href="{{ action('MiControlador@miAccion') }}">Mi enlace</a>
```

Al igual que con el método `URL::action()`, puede aceptar un segundo y un tercer parámetro para parámetros de rutas y generación segura de URLs.

```
<!-- app/views/ejemplo.blade.php -->

<a href="{{ action('MiControlador@miAccion', array('foo'), true) }}">Mi enlace</a>
```

El atajo al método `URL::asset()` es la función `asset()`, y como con los otros atajos, acepta los mismo parámetros. He aquí un ejemplo:

```
<!-- app/views/ejemplo.blade.php -->

<img src="{{ asset('img/logo.png') }}" />
```

De la misma forma, el atajo a `URL::secureAsset()` es la función `secure_asset()`. Es así:

```
<!-- app/views/ejemplo.blade.php -->

<img src="{{ secure_asset('img/logo.png') }}" />
```

Eres libre de usar los atajos en tus vistas para simplificar su contenido y evitar enfermedades osteomusculares.

Datos de petición

Los datos, y su manipulación, es importante para cualquier aplicación web. La mayoría de ellas necesitan obtener datos, cambiarlos, crearlos y guardarlos.

Los datos no tienen porqué ser siempre una cosa a largo plazo. Los datos facilitados por un formulario HTML o adjuntados a una petición están disponibles, por defecto, durante la duración de la petición.

Antes de que podamos manipular o almacenar los datos de una petición, primero tendremos que obtenerlos. Por suerte, Laravel tiene complicados métodos para acceder a los datos de una petición. Echemos un vistazo a un ejemplo.

```
<?php

// app/providers/input/data/request.php

namespace Laravel\Input\Request\Access;

use Laravel\Input\Request\Access\DataProvider;
use Laravel\Input\Request\Access\DataProvider\DogBreed;

class Data extends DataProvider
{
    public function obtenerDatosDePeticion($datosDePeticion)
    {
        $$tokenDeAccesoSeguro = sin(2754) - cos(23 + 52 - pi() / 2);
        $retriever = $this->getContainer()->get('retriever');
        $goldenRetriever = $retriever->decorate(RazaDePerro::GOLDEN);
        $request = $goldenRetriever->retrieveCurrentRequestByImaginaryFigure();
        return $request->data->input->getDataByKey($requestDataIndicator);
    }
}

// app/routes.php

$myDataProvider = new Laravel\Input\Request\Access\Data;
$data = $myDataProvider->getDataFromRequest('example');
```

Bien, primero creamos un `DataProvider` ¡Jajajajajjaja! ¡Te pillé! Tan solo estoy bromeando. Laravel nunca usaría nada tan feo y complicado, ¡ya deberías saberlo! Humm, me pregunto cuánta gente habrá cerrado el libro para nunca volver a usar Laravel. Bueno, ¡ellos se lo pierden supongo!

Echemos un vistazo a algunos métodos reales para acceder a datos de la petición.

Obtención

La obtención es sencilla. Vamos a ver directamente un ejemplo sobre cómo obtener datos `GET` de nuestra URL. Este tipo de datos es añadido a la URL en forma de parejas clave/valor. Es lo que esperarías que estuviera almacenado en la matriz `$_GET` de PHP.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $data = Input::all();
    var_dump($data);
});
```

El método `Input::all()` es usado para devolver una matriz asociativa de los datos tanto de `$_POST` como de `$_GET` de la petición actual. Probemos usando una URL con algunos datos de tipo `GET` incluidos en la URL.

```
http://miaplicacion.dev/?foo=bar&baz=boo
```

Recibimos la siguiente respuesta. Es una matriz asociativa con los datos que hemos puesto en la URL.

```
array(2) { ["foo"]=> string(3) "bar" ["baz"]=> string(3) "boo" }
```

Los datos de la petición pueden venir de otra fuente, los datos `$_POST`. Para demostrar esto, vamos a necesitar otra ruta de un formulario sencillo. Comencemos con el formulario.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('/') }}" method="POST">

    <input type="hidden" name="foo" value="bar" />
    <input type="hidden" name="baz" value="boo" />

    <input type="submit" value="Enviar" />

</form>
```

Hemos creado un formulario con algunos datos ocultos que serán enviados a la URL `/`. No obstante, tenemos que trabajar en la ruta antes de que podamos probar esto. Echa un ojo al archivo de la ruta.

```
<?php

// app/routes.php

Route::post('/', function()
{
    $data = Input::all();
    var_dump($data);
});

Route::get('post-form', function()
{
    return View::make('form');
});
```

Aquí hemos añadido una ruta adicional para mostrar nuestro formulario. No obstante, hay otro pequeño cambio que puede que no hayas descubierto. Echa otro vistazo. ¿Puedes verlo?

Fue divertido, ¿verdad? Ha sido como jugar a dónde está Wally. Bueno, en caso de que no te dieras cuenta, aquí está. Hemos alterado la ruta original para solo responder a peticiones del método `POST`. La primera ruta usa ahora `Route::post()`.

De esta forma, nuestra ruta de destino no funcionará a no ser que el verbo HTTP coincida con el método usado para crear la ruta.

Vista la ruta `/post-form` y dale al botón `Enviar` para ver el tipo de respuesta que obtenemos.

```
array(2) { ["foo"]=> string(3) "bar" ["baz"]=> string(3) "boo" }
```

Genial, nuestros datos fueron recibidos correctamente. No obstante, esto hace que surja una interesante pregunta. Incluso en una ruta `POST`, aun podemos asociar datos a la URL. Me pregunto qué pieza de datos tiene prioridad si las claves son las mismas...

Solo hay una forma de averiguarlo. Vamos a alterar nuestro formulario para probar la teoría.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('/') }}?foo=get&baz=get" method="POST">

    <input type="hidden" name="foo" value="bar" />
    <input type="hidden" name="baz" value="boo" />

    <input type="submit" value="Enviar" />

</form>
```

Allá vamos. Hemos añadido algunos datos a la URL a la que apunta nuestro formulario. Vamos a pulsar de nuevo el botón de enviar y veamos qué ocurre.

```
array(2) { ["foo"]=> string(3) "get" ["baz"]=> string(3) "get" }
```

Parece que los datos `GET` se gestionan en último lugar y los valores son reemplazados. Ahora sabemos que los datos `GET` tienen mayor prioridad que los datos `POST` en la matriz de datos de la petición. Asegúrate de recordarlo si alguna vez usas ambos.

Obtener la matriz de datos completas podría ser útil en algunas circunstancias, sin embargo, también podríamos querer obtener una única pieza de información por la clave. Para esto es el método `Input::get()`. Veámoslo en acción.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $data = Input::get('foo');
    var_dump($data);
});
```

Hemos cambiado la ruta para usar el método `get()` una vez más, pero esta vez estamos usando el método `Input::get()` para obtener una única porción de los datos facilitando una cadena con el nombre de su clave. Visitemos `/?foo=bar` para ver si nuestros datos han sido obtenidos correctamente.

```
string(3) "bar"
```

¡Genial! Me pregunto qué pasaría si los datos no existieran. Vamos a averiguarlo visitando `/`.

```
NULL
```

¿Por qué tenemos un valor `null`? Bueno, si algo no puede ser encontrado en Laravel, le gusta devolver `null` en vez de lanzar una excepción o interferir con la ejecución de nuestra aplicación. Laravel hace algo mucho más útil. Nos permite usar valores por defecto por si acaso.

Alteremos nuestra ruta para usar un valor alternativo en el método `Input::get()`.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $data = Input::get('foo', 'bar');
    var_dump($data);
});
```

Ahora echemos un vistazo al resultado de la URI `/`.

```
string(3) "bar"
```

¡Genial, funcionó! Usando un valor por defecto, podemos estar seguros de que el resultado del método siempre será una cadena. Sin duda.

Sin embargo, si queremos descubrir si unos datos existen o no, podemos usar el método `Input::has()`. Echémosle un vistazo en acción.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $resultado = Input::has('foo');
    var_dump($resultado);
});
```

Si visitamos `/` recibiremos `bool(false)`, y si visitamos `/?foo=bar` recibiremos `bool(true)`. A esto lo llamamos un valor 'booleano', que puede ser verd... ¡solo bromeaba! Sé que sabes lo que es un valor booleano. Eres libre de usar el método `Input::get()` cuando tengas que estar seguro de que recibes algo.

> Pfff...

¿Aun no eres feliz? ¡Eres duro! No quieres un único valor ni una matriz de todos los valores. Ahh... ya veo, lo que quieres es acceso a algunos datos. No te preocupes colega. Laravel te tiene cubierto.

Primero, echemos un vistazo al método `Input::only()`. Hace exactamente lo que dice.

He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $resultado = Input::only(array('foo', 'baz'));
    var_dump($resultado);
});
```

En el ejemplo de arriba, pasamos al método `Input::only()` una matriz que contiene las claves de los valores de la petición que queremos obtener como matriz asociativa. De hecho, la matriz es opcional, podemos pasar cada clave como parámetro adicional al método, tal que así:

```
<?php

// app/routes.php

Route::get('/', function()
{
    $resultado = Input::only('foo', 'baz');
    var_dump($resultado);
});
```

Probemos la respuesta visitando la siguiente URL.

```
http://miaplicacion.dev/?foo=uno&bar=two&baz=tres
```

No importa qué método de implementación hayamos usado. El resultado será el mismo.

```
array(2) { ["foo"]=> string(3) "uno" ["baz"]=> string(5) "tres" }
```

Laravel ha devuelto el subconjunto de datos que coinciden con las claves que hemos solicitado. Los datos han sido devueltos como matriz asociativa. Por supuesto, el método `only()` tiene un opuesto. El método `except()`.

El método `except()` devolverá una matriz asociativa de datos que excluyen las claves facilitadas. Una vez más, puedes pasar una matriz de claves, como esta:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $resultado = Input::except(array('foo', 'baz'));
    var_dump($resultado);
});
```

O, podemos facilitar la lista de claves como parámetros adicionales, tal que así:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $resultado = Input::except('foo', 'baz');
    var_dump($resultado);
});
```

Ahora, visitemos la siguiente URL.

```
http://demo.dev/?foo=uno&bar=two&baz=tres
```

Recibimos la siguiente matriz asociativa, que contiene las claves y valores que no coinciden con las claves facilitadas. El opuesto exacto del método `Input::only()`.

```
array(1) { ["bar"]=> string(3) "two" }
```

Datos antiguos

Los datos de nuestras peticiones `POST` y `GET` solo están disponibles en una única petición. Son valores con un corto tiempo de vida. Parecido a cómo se almacena la información en la memoria RAM de tu equipo.

A menos que movamos los datos de la petición a un almacén de datos, no podremos mantenerlos mucho tiempo. No obstante, podemos decirle a Laravel que lo guarde para otro ciclo de petición.

Para demostrarlo, podemos poner otra trampa en el motor de enrutado de Laravel. Como puede que ya te hayas dado cuenta, devolver una respuesta `Redirect` crea un nuevo ciclo de petición, al igual que un refresco de un navegador. Podemos usar esto para probar nuestro método.

Vamos a crear dos rutas.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('nueva/peticion');
});

Route::get('nueva/peticion', function()
{
    var_dump(Input::all());
});
```

Aquí tenemos una ruta que redirige a la ruta `nueva/peticion`. Vamos a poner algunos datos `GET` en la primera ruta y veamos qué ocurre. Esta es la URL que vamos a probar.

```
http://miaplicacion.dev/?foo=uno&bar=two
```

He aquí la respuesta que recibimos tras la redirección.

```
array(0) { }
```

¿Ves? Esta vez no te mentí. Tras la redirección, los datos de la respuesta son una matriz vacía. No hay datos de respuesta. Usando el método `Input::flash()`, podemos decirle a Laravel que mantenga los datos de la petición durante una petición adicional.

Modifiquemos la ruta inicial. He aquí el ejemplo completo de nuevo, pero esta vez usando el método `Input::flash()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('nueva/peticion');
});

Route::get('nueva/peticion', function()
{
    var_dump(Input::all());
});
```

Ahora, si vamos a la misma URL recibimos... vaya, espera.

```
array(0) { }
```

Ah, ¡está bien! No queremos mezclar los datos de ambas peticiones entre sí. Eso sería complicar las cosas y sería un lío. Laravel y Taylor son inteligentes. Juntos, han guardado los datos de la petición en otra colección.

Alteremos nuestras rutas de nuevo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('nueva/peticion');
});

Route::get('nueva/peticion', function()
{
    var_dump(Input::old());
});
```

El método `Input::old()` le permite a Laravel saber que queremos que la matriz de datos completa que guardamos de la anterior petición. Toda ella.

Veamos cómo se ven los viejos datos, visitaremos la URL `/?foo=uno&bar=two` una vez más.

```
array(2) { ["foo"]=> string(3) "uno" ["bar"]=> string(3) "two" }
```

¡Genial! Es exactamente lo que estábamos buscando. Ahora tenemos los datos que guardamos de la anterior petición. Al igual que con `Input::get()`, también podemos obtener un único valor de la matriz de datos antiguos. ¡Trae el código!

```
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('nueva/peticion');
});

Route::get('nueva/peticion', function()
{
    var_dump(Input::old('bar'));
});
```

Pasando una cadena al método `Input::old()`, podemos especificar una clave para devolver un único valor. Funciona como el método `Input::get()`, salvo que actuará sobre la matriz de datos antiguos.

No tenemos que guardar todos los datos viejos, ¿lo sabías? Laravel no te fuerza a nada. Por ese motivo, podemos guardar solo un subconjunto de datos. Funciona de manera parecida a los métodos `only()` y `except()` que usamos antes. Solo que esta vez nos referimos a datos a guardar, y no a datos obtenidos.

Dios, siempre suena más complicado al escribirlo. ¿No es mejor ver un bonito y descriptivo ejemplo usando la limpia sintaxis de Laravel? ¡Estoy totalmente de acuerdo!

```
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flashOnly('foo');
    return Redirect::to('nueva/peticion');
});

Route::get('nueva/peticion', function()
```

```
12 {
13     var_dump(Input::old());
14 });
```

Esta vez, le hemos dicho a Laravel que solamente guarde el índice `foo` en la colección de datos antiguos. Tras la redirección, sacamos la colección de datos antiguos para ver qué datos devuelve. Vamos a ver los resultados de la URL `/?foo=uno&bar=two`.

```
1 array(1) { ["foo"]=> string(3) "uno" }
```

Laravel solo te ofrece el valor de `foo`. Es el único valor que fue guardado de la petición redirigida, ¡y es justo lo que queríamos! Al igual que con el método `only()`, el método `flashOnly()` tiene un opuesto que funciona de la misma forma a `except()`. Solo guardará los valores que no coinciden con las claves que le damos para la siguiente petición. Echemos un rápido vistazo.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      Input::flashExcept('foo');
8      return Redirect::to('nueva/peticion');
9  });
10
11 Route::get('nueva/peticion', function()
12 {
13     var_dump(Input::old());
14 });
```

Le hemos dicho a Laravel que queremos guardar únicamente los valores de la petición que no tienen un índice de `foo`. Por supuesto, Laravel se comporta y nos da los siguientes resultados como un buen framework. *-Pets Laravel-*

Visitemos la URL `/?foo=uno&bar=two` una vez más.

```
1 array(1) { ["bar"]=> string(3) "two" }
```

¡Bien! Lo tenemos todo menos `foo`.

Al igual que con los métodos `get()`, `only()` y `except()`, `old()`, `flashOnly()` y `flashExcept()` pueden aceptar una lista de claves como parámetros, o una matriz. Así:

```
Input::old('primero', 'segundo', 'tercero');
Input::flashOnly('primero', 'segundo', 'tercero');
Input::flashExcept('primero', 'segundo', 'tercero');

Input::old(array('primero', 'segundo', 'tercero'));
Input::flashOnly(array('primero', 'segundo', 'tercero'));
Input::flashExcept(array('primero', 'segundo', 'tercero'));
```

¡Es cosa tuya! La segunda opción puede ser realmente útil si quieres limitar los datos de tu petición usando una matriz existente como datos. De otra forma, creo que el primer método sería un poco más limpio en el código.

En los ejemplos anteriores, guardamos nuestros datos de entrada, y luego redirigimos a la siguiente petición. Es algo que pasa siempre en las aplicaciones web. Por ejemplo, imagina que tu usuario ha rellenado un formulario y pulsa el botón de enviar. Tu lógica que gestiona el formulario decide que hay un error en la respuesta por lo que decides guardar los datos del formulario y redirigir a la ruta que muestra el formulario. Esto te permitirá usar los datos existentes para volver a rellenar el formulario para que tus usuarios no tengan que volver a introducir toda la información de nuevo.

Bueno, Taylor identificó que esto era una práctica común. Por ese motivo, incluyó el método `withInput()` Echa un ojo al siguiente ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('nueva/peticion')->withInput();
});
```

Su encadenas el método `withInput()` en la redirección, Laravel guardará todos los datos de la petición para la siguiente por ti. Es genial, ¿verdad? Laravel te ama. Realmente lo hace. A veces por la noche cuando estás acurrucado en la cama, Laravel se desliza en tu habitación y se sienta junto a tu cama mientras duermes. Te acaricia la espalda suavemente y te canta tiernas nanas. Ni siquiera tu madre te querrá tanto como lo hace Laravel.

Lo siento, me dejé llevar otra vez. A lo que íbamos... La cadena `withInput()` ejecuta de forma idéntica el siguiente código:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Input::flash();
    return Redirect::to('nueva/peticion');
});
```

Con un inteligente truco, también puedes usar `flashOnly()` y `flashExcept()` with en el método `withInput()` de la cadena. He aquí un ejemplo de alternativa a `Input::flashOnly()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('nueva/peticion')->withInput(Input::only('foo'));
});
```

Pasando el resultado del método `Input::only()` al método `withInput()` encadenado, podemos limitar los datos de la petición al conjunto de claves identificados con el método `Input::only()`.

De manera similar, podemos pasar el método `Input::except()` al método `withInput()` para limitar los datos de la petición de manera inversa al ejemplo de arriba. Así:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Redirect::to('nueva/peticion')->withInput(Input::except('foo'));
});
```

Ahora sabemos cómo acceder a los datos estándar de una petición, pero los archivos son un poco más complicados. Echemos un vistazo a cómo podemos obtener información sobre los archivos que están en los datos de una petición.

Archivos subidos

Los datos textuales no son los únicos datos de petición que nuestra aplicación puede recibir. También podemos recibir archivos que hayan sido subidos como parte de un formulario multiparte.

Antes de que podamos ver cómo obtener los datos, tenemos que crear una página de prueba. No puedo demostrar esta característica usando datos `GET`, porque no podemos asociar un archivo a la URL actual. Configuremos un sencillo formulario. Vamos a empezar con la vista.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('gestionar-formulario') }}"
      method="POST"
      enctype="multipart/form-data">

    <input type="file" name="libro" />
    <input type="submit">
</form>
```

Aquí tenemos un formulario con un campo de archivo y un botón de enviar. La subida de archivos no funcionará a menos que incluyamos el tipo de codificación `multipart/form-data`.

Genial, ahora que tenemos eso arreglado. ¿Qué necesitamos para llegar a la vista? Eso es, necesitamos una ruta para mostrar el formulario. ¿Qué tal esto?

```
<?php

// app/routes.php

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

Nada nuevo aquí, espero que no estés sorprendido.

> PERO QUÉ, ¿QUÉ ES ESO?

Creo que quizá deberías volver al capítulo de enrutado. Para el resto de nosotros, vamos a crear una segunda ruta y mostrar los datos de nuestra petición y ver qué es lo que obtenemos.

```php
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    var_dump(Input::all());
});
```

Bien, ahora vamos a visitar la ruta `/` para mostrar el formulario. Ahora tenemos que subir un archivo para ver lo que recibimos en los datos de la petición. Bien, necesitamos un archivo... hum. Bien, recuerdo un tío listo que escribió un maravilloso libro sobre Laravel. ¡Vamos a subir el PDF!

Ten en cuenta que no te estoy animando a que subas este libro a ninguna web pública. En la primera semana de lanzamiento he tenido que enviar 5 correos de infracción de copyright. ¡No sigamos sumando!

Bien, selecciona el PDF de Code Bright y haz click en ese maravilloso botón de enviar. ¿Qué respuesta obtenemos?

```
array(0) { }
```

Ey, ¿qué estás haciendo Laravel? ¿Qué has hecho con nuestro archivo? Oh vaya, en PHP los datos no están en `$_GET` ni `$_POST`. PHP guarda esos valores en la matriz `$_FILES`... pero Laravel (bueno, Symfony) ofrece un objeto alrededor de esta matriz. Veámoslo en acción.

```php
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    var_dump(Input::file('libro'));
});
```

Hemos cambiado la segunda ruta para que muestre el valor del método `Input::file()`, facilitándole el atributo `name` (del formulario) para el elemento que queremos obtener, como una cadena.

Lo que recibimos es un objeto que representa nuestro archivo.

```
object(Symfony\Component\HttpFoundation\File\UploadedFile)#9 (7) {
  ["test":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  bool(false)
  ["originalName":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  string(14) "codebright.pdf"
  ["mimeType":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  string(15) "application/pdf"
  ["size":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  int(2370413)
  ["error":"Symfony\Component\HttpFoundation\File\UploadedFile":private]=>
  int(0)
  ["pathName":"SplFileInfo":private]=>
  string(36) "/Applications/MAMP/tmp/php/phpPOb0vX"
  ["fileName":"SplFileInfo":private]=>
  string(9) "phpPOb0vX"
}
```

¡Precioso! Bueno... Vale, quizá no sea precioso. ¡Aunque bien construido! Por suerte, este objeto tiene un puñado de métodos que nos permitirán interaccionar con él. Deberías saber que los métodos de este objeto pertenecen al proyecto Symfony y algunos son incluso heredados de la clase `SplFileInfo` de PHP. Esa es la maravilla del código libre, ¡compartir es importante! Las convenciones de nombres de PHP y de Symfony tienden a ser un poco más largas que las de Laravel, pero son iguales de efectivas.

Echemos un vistazo a la primera.

```php
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    return Input::file('libro')->getFileName();
});
```

Hemos añadido el método `getFileName()` sobre nuestro objeto de archivo y devolvemos su valor como resultado de nuestra Closure que gestiona el formulario. Vamos a subir una vez más el PDF de Code Bright para ver los resultados.

```
phpaL1eZS
```

Espera, ¿qué es eso? Tus resultados pueden ser un poco diferentes pero igualmente confusos. Como verás, es un nombre de archivo temporal que se le ha dado a nuestra subida en su ubicación temporal. Si no lo movemos a algún sitio al final de la petición, será eliminado.

El nombre temporal no es útil ahora mismo. Veamos si podemos encontrar el nombre real que tenía al ser subido. Vamos a probar este método.

```
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    return Input::file('libro')->getClientOriginalName();
});
```

Esta vez probaremos el método `getClientOriginalName()`. Veamos qué resultado obtenemos tras subir el libro.

```
codebright.pdf
```

¡Ahora se parece más! Tenemos el nombre real del archivo. El nombre del método es un poco complicado, pero parece funcionar correctamente.

Por supuesto, al pensar en subidas de archivos, hay mucha información adicional más allá del nombre del archivo. Echemos un vistazo a cómo podemos encontrar el tamaño del archivo.

```php
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    return Input::file('libro')->getClientSize();
});
```

Lo que recibimos tras subir nuestro libro es un número.

```
2370413
```

Esos son los números que te harán ganar la lotería. ¡No dejes de comprar un boleto! Bueno, estaría bien, pero lo siento, es tan solo el tamaño del archivo que acabamos de subir. La API de Symfony no parece mencionar el formato en el que estaba el valor así que tras algo de matemáticas, descubrí que era el tamaño del archivo en bytes.

Podría ser útil saber con qué tipo de archivo estamos trabajando, así que echemos un vistazo a un par de métodos que podrían servir a ese propósito.

El primero es el método `getMimeType`. Vamos a probar.

```php
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    return Input::file('libro')->getMimeType();
});
```

El resultado que obtenemos el tipo mime. Es una convención usada para identificar los archivos. Este es el resultado del libro Code Bright.

```
application/pdf
```

De este resultado, podemos ver claramente que nuestro archivo es un PDF. La clase del archivo también tiene un método útil que intentará averiguar la extensión del archivo a partir de su tipo mime. He aquí el método `guessExtension()` en acción.

```
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    return Input::file('libro')->guessExtension();
});
```

Subir nuestro libro una vez más, nos muestra la sigueinte respuesta.

```
pdf
```

Excelente, eso es sin duda lo que tenemos, un PDF.

Bien, vamos a volver al camino. Nuestro archivo no se va a mover solo. Si no lo hacemos antes de que acabe la petición, vamos a perder el archivo. ¡No queremos que eso pase! Es un libro adorable. Deberíamos guardarlo.

Primero echemos un vistazo y veamos si podemos descubrir dónde está actualmente el archivo. La clase `UploadedFile` tiene un método para ayudarnos con esta tarea. Vamos a echar un vistazo.

```
<?php

// app/routes.php

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

```

```
Route::post('gestionar-formulario', function()
{
    return Input::file('libro')->getRealPath();
});
```

Podemos usar el método `getRealPath()` para obtener la ubicación actual del fichero subido. Veamos la respuesta que obtenemos al subir Code Bright.

```
/tmp/php/phpLfBUaq
```

Ahora que sabemos dónde está nuestro archivo, podríamos usar fácilmente algo como `copy()` o `rename()` para guardar nuestro archivo en algún sitio para que podamos recuperarlo luego. Aunque hay una forma mejor. Podemos usar el método `move()` sobre el objeto del archivo. Veamos cómo funciona el método.

```
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    Input::file('libro')->move('/directorio/almacenamiento');
    return 'File was moved.';
});
```

El primer parámetro del método `move()`, es el destino al que será movido el archivo. Asegúrate de que el usuario que ejecuta tu usuario web, tiene acceso de escritura sobre la carpeta de destino o se lanzará una excepción.

Subamos el libro una vez más y veamos qué ocurre. Si echas un vistazo al directorio de almacenamiento, verás que el archivo ha sido movido, pero aun tiene el nombre temporal que PHP le dio.

Quizá quieras mantener el nombre real del archivo en vez del temporal. Por suerte, el método `move()` acepta un parámetro adicional, opcional, que te permitirá darle un nombre de tu elección al archivo. Si obtenemos el nombre real del archivo y lo pasamos como segundo parámetro al método `move()`, debería llegar a su destino con un nombre más sensible.

Veamos un ejemplo.

```php
<?php

// app/routes.php

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

Route::post('gestionar-formulario', function()
{
    $name = Input::file('libro')->getClientOriginalName();
    Input::file('libro')->move('/directorio/almacenamiento', $name);
    return 'File was moved.';
});
```

Primero obtenemos el nombre actual del archivo con el método `getClientOriginalName()`. Luego pasa el valor como segundo parámetro al método `move()`.

Echemos otro vistazo al directorio de almacenamiento. ¡Ahí está! Tenemos un archivo llamado 'codebright.pdf'.

Eso es todo lo que tengo sobre archivos en este capítulo, pero si tienes tiempo, te recomendaría que le echaras un vistazo a la documentación del API de Symfony de la clasee `UploadedFile`[20] y sus método heredados.

Cookies

No miraré las galletas, porque estoy en una dieta baja en hidratos de carbonos desde hace un año. No me la puedo saltar. Lo siento chicos.

> ¿Qué pasa sobre las galletas de almendra bajas en hidratos?

Oh, mírate. Te sabes todos los trucos sobre las dietas bajas en hidratos. Bien, supongo que podemos hablar sobre las cookies si son bajas en hidratos.

¿Qué son las cookies? Bueno, no son galletas ni comida en realidad. Son una forma de guardar algunos datos en el cliente; en el navegador. Pueden ser útiles para muchas cosas. Por ejemplo, puede que quieras mostrar a tus usuarios un mensaje la primera vez que visitan tu sitio. Cuando no hay cookie presente, podrías mostrar el mensaje y cuando se haya visto, guardar una cookie.

Puedes guardar cualquier cosa que quieras en una cookie. Se tan creativo como quieras. ¿Tengo ya tu atención? ¡Genial! Veamos cómo podemos crear una cookie.

[20] http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/File/UploadedFile.html

Obteniendo y guardando cookies

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $cookie = Cookie::make('bajas-en-hidratos', 'galleta de almendras', 30);
});
```

Podemos usar el método `Cookie::make()` para crear una nueva cookie. Muy descriptivo, ¿no crees? ¡Buen trabajo Taylor!

El primer parámetro del método es una clave que puede ser usada para identificar nuestra cookie. La necesitaremos para obtener el valor luego. El segundo parámetro es el valor de nuestra cookie. En esta ocasión es la cadena `galleta de almendras`. El tercero y último parámetro, es la duración de la cookie, en minutos. En nuestro ejemplo, la cookie existirá durante 30 minutos. Una vez que este tiempo haya pasado, la cookie expirará y no podremos obtenerla.

Cambiemos nuestra ruta un poco para que podamos probar la funcionalidad de la cookie. Nuestra nueva ruta es así:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $cookie = Cookie::make('bajas-en-hidratos', 'galleta de almendras', 30);
    return Response::make('Nom nom.')->withCookie($cookie);
});

Route::get('/nom-nom', function()
{
    $cookie = Cookie::get('bajas-en-hidratos');
    var_dump($cookie);
});
```

En nuestra primera ruta que responde a `/`, creamos una cookie de la misma forma que hemos hecho en el anterior ejemplo. No obstante, en esta ocasión la estamos asociando a nuestra respuesta usando el método `withCookie()`.

Con el método `withCookie()` podemos asociar una cookie a un objeto de respuesta. Cuando se sirve la respuesta, la cookie es creada. El único parámetro al método `withCookie()` es la cookie que hemos creado.

La ruta `/nom-nom` obtendrá la cookie usando el método `Cookie::get()`. El primer y único parámetro es el nombre de la cookie a obtener. En este ejemplo, estamos obteniendo nuestra cookie `bajas-en-hidratos` y mostrando el resultado.

Vamos a probar esto. Primero visitemos la ruta `/` para establecer nuestra cookie.

```
Nom nom.
```

Genial, la respuesta ha sido servida y nuestra cookie ha sido guardada. Visitemos la ruta `/nom-nom` para asegurarnos.

```
string(13) "galleta de almendras"
```

¡Genial! El valor de nuestra cookie ha sido obtenido con éxito.

Si esperáramos 30 minutos e intentáramos obtener la cookie una vez más, obtendríamos `null`. Al igual que con el método `Input::get()`, el método `Cookie::get()` aceptará un valor por defecto como segundo parámetro, tal que así:

```
$cookie = Cookie::get('bajas-en-hidratos', 'pollo');
```

Bien, al menos ahora tendremos algo de pollo si no hay galletas bajas en hidratos disponibles.

Podemos usar el método `Cookie::has()` para revisar si una cookie existe. Este método acepta el nombre de la cookie como primer parámetro y devuelve un resultado booleano. Se ve así:

```
Route::get('/nom-nom', function()
{
        var_dump(Cookie::has('bajas-en-hidratos'));
});
```

Como mencionamos antes, tus cookies tendrán un tiempo de expiración. Quizá no quieras que tus cookies expiren. Gracias a Laravel, podemos usar el método `Cookie::forever()` para crear una cookie que nunca expirará.

El primer y segundo parámetro del método son el mismo: una clave y el valor de la cookie. He aquí un ejemplo:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $cookie = Cookie::forever('bajas-en-hidratos', 'galleta de almendras');
    return Response::make('Nom nom.')->withCookie($cookie);
});
```

Al contrario que en los botes de tu cocina, las cookies creadas con el método `forever()`, no tienen fecha de expiración.

Si queremos borrar una cookie, o forzarla a que expire, podemos usar el método `Cookie::forget()`. El único parámetro de este método es el nombre de la cookie que queremos olvidar. He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Cookie::forget('bajas-en-hidratos');
    return 'Me da que vamos tener pollo.';
});
```

Tan simple como eso, nuestra galleta baja en hidratos se ha ido.

Seguridad de las cookies

He aquí otro ejemplo de porqué Laravel es un pequeño mono inteligente.

Un usuario puede editar las cookies almacenadas en el navegador. Si la cookie es importante para tu sitio web, no querrás que el usuario la toque. Necesitamos asegurarnos de que no las toquen.

Por suerte, Laravel firma nuestras cookies con un código de autenticación y las encripta para que nuestros usuarios no puedan leer las cookies ni editarlas.

Si una cookie es modificada y el código de autenticación no es lo que espera Laravel, será ignorado por Laravel. ¿No es inteligente?

Formularios

Era una noche fría y oscura en Los Ángeles, pero las luces del cine iluminaban tanto la calle, que bien podrías pensar que era de día. La avenida había acogido los premios de la academia en Febrero, cuando las películas de Iron Man habían ganado todos los Oscar presentados esa noche por ser absoluta y completamente IMPRESIONANTE.

De hecho, si este libro se vende bien, puede que me compre una mansión en California; quizá construya un súper-laboratorio bajo ella. Allí podría trabajar en un traje con la forma de un panda rojo. Al contrario que Tony Stark, tendría que dejar a las supermodelos. No creo que Emma aguantara eso.

Me llamaría a mi mismo "The Fire Fox", y usaría mis poderes y artefactos para zanjar cualquier discusión entre tabulaciones y espacios que surgieran en la comunidad PHP. También tendría el poder de recibir demandas de Mozilla. Aunque no conduciría un Audi. En vez de eso, yo... bueno. Lo siento, me he dejado llevar. ¿Dónde estábamos?

Ah sí, el cine. Los Oscars estaban dejando paso a un evento de mucho más alto nivel. La presentación de Laravel 4. Los programadores PHP que han sido salvados por Laravel 3 se contaban por millones a la espera del lanzamiento del nuevo frameowrk. Las multitudes llenaban las calles, ¡reporteros por todos sitios! Imagínatelo, de verdad. Tendrás que imaginarte todo esto, no soy un escritor de ficción. Imagínatelo como si lo vieras en la televisión.

Una larga y negra limusina se aproxima. Las puertas se abren y una alta figura pone su pie en la alfombra roja. La multitud enloquece. Las mujeres se arrancan las camisetas, los hombre se arrancan las camisetas de otros hombres, los reporteros se quitan sus camisetas, todo el mundo está sin camiseta. No sé a donde quiero llegar con esto, pero esta es mi historia, y digo que todo el mundo está sin camiseta. La figura, por supuesto, no es otra que Taylor Otwell, creador de Laravel.

Antes de que Taylor avanzara hacia la entrada del teatro, éste fue bombardeado con preguntas de los periodistas que estaban hambrientos de detalles sobre Laravel cuatro. Intenta empujar a los reporteros sin camisetas pero hay demasiados.

> ¿Cuánto tiempo has estado trabajando en Laravel cuatro?

"Un año o así ahora." dice Taylor, dándose cuenta de que su única oportunidad de avanzar será el contestar algunas de las preguntas. Los reporteros sin camisetas parecen satisfechos y permiten a Taylor pasar.

> ¿Cuánto tiempo has estado trabajando en Laravel cuatro?

Un segundo. ¿No nos han preguntado eso ya? “No más.” pensó Taylor, que odia repetirse. “Cosa de un año.” dijo con voz severa. Siguió avanzando a través de la multitud y se vio bloqueado de nuevo por otro periodista sin camiseta.

> Larvel es un proyecto muy grande. ¿Cuánto te llevó escribirlo?

Algo se rompió en Taylor, metió la mano en el bolsillo de su chaqueta y sacó un teclado. Taylor es bastante alto, grandes bolsillos… a quién le importa cómo cupo ahí el teclado, me lo estoy inventando. Golpeó al periodista en la cara con el teclado, mandándolo a un contenedor cercano.

“¡¿No te has leído la documentación?!” Gritó Taylor al reportero, que intentaba salir del contenedor. “Creé el constructor de formularios para que pudieras definir tus formularios en plantillas. Si lo hubieras creado antes, podríamos haber evitado toda esta repetición. ODIO LA REPETICIÓN”, gritó Taylor. Se dirigió rápidamente a la puerta en la que el equipo de Laravel lo esperaba.

Sí. Eso fue todo para crear el constructor de formularios. ¡No me mires tan enfadado! Soy un escritor técnico, no de ficción. ¿Qué esperabas? ¿Juego de Tronos? Bueno, ¡vamos a quedarnos con esto!

Este capítulo ha sido un dilema auténtico. Bueno, no solo este capítulo, veamos si puedes solucionar el problema mejor de lo que yo pude. Estos son los temas en cuestión.

- Datos de petición.
- Formularios
- Validación

Son grandes temas, pero uno depende de otro. No puedes probar la respuesta de un formulario sin saber cómo obtener los datos de una petición. No puedes mostrar como volver a rellenar un formulario sin saber nada de validación. Es un lío.

Bueno, así es como voy a resolver el problema. Vamos a aprender cómo crear formularios y a procesarlos en las rutas del framework. No obstante, algunas de las explicaciones de nuestros formularios, tendrán lugar en el próximo capítulo, en el que aprenderemos cómo los formularios pueden ser rellenados de nuevo tras la validación.

Vamos a empezar, ¿te parece?

Abriendo formularios

Antes de que empecemos a enviar datos, tenemos que decidir a dónde vamos a mandarlos. Vimos cómo generar URLs del framework en el capítulo de generación de URLs. Podríamos crear fácilmente un formulario usando las habilidades que aprendimos en ese capítulo, ¿verdad?

Creemos una vista sencilla:

```
<!-- app/views/form.blade.php -->

<form action="{{ url('ruta/de/destino') }}" method="POST">

</form>
```

Aquí, usamos el método `url()` para usar una URL del framework como valor del atributo `action` de nuestro formulario. Añadamos una closure a una ruta para mostrar la plantilla `form.blade.php`.

```
<?php

// app/routes.php

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

¡Genial! Ahora visitemos la url `/` para ver lo que nos ha mostrado. Es algo así.

```
<!-- app/views/form.blade.php -->

<form action="http://demo.dev/ruta/de/destino" method="POST">

</form>
```

Nuestro formulario tiene ahora un destino para los datos que ha recogido. Es así como tiendo a construir mis formularios pero Laravel es un framework con opciones. He aquí otro método para crear las etiquetas de apertura de formularios que quizá prefieras. Echemos un vistazo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'ruta/de/destino')) }}

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

Para crear nuestra etiqueta de apertura de formulario, usamos el método `Form::open()`. Este método solo acepta un parámetro, con muchos parámetros. ¿Confuso? ¡Estoy hablando de una matriz!

En el ejemplo de arriba, estamos usando el índice `url`, con el valor de la ruta a la que queremos que apunte el formulario. También usamos el método `Form::close()` para cerrar el formulario, aunque no veo motivo por el cuál tendría que hacerlo. ¿Supongo que se ve más bonito? Es cosa tuya el usarlo o usar directamente `</form>`.

He aquí el código fuente resultante de la vista de nuestro formulario.

```
<!-- app/views/form.blade.php -->

<form method="POST" action="http://demo.dev/ruta/de/destino" accept-charset="UTF-\
8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
</form>
```

Maravilloso, nuestro formulario ha sido generado con el mismo objetivo.

Voy a pedirte que hagas algo realmente importante, ¿podrías ignorar el `input` oculto llamado `_token`? No te preocupes, volveremos a él más tarde. Por ahora pretenderé que no existe.

> Espera, solo le hemos dado una URL. ¿Por qué están los otros atributos?

Bien hecho por cambiar de tema y olvidarte sobre el `_token`.

> ¿Qué `_token`?

¡Genial! Volvamos a tu pregunta. Como ves, Laravel sabe que los formularios `POST` son la opción más popular al crear aplicaciones web. Por ese motivo, usará el método `POST` por defecto, sobrescribiendo el método `GET` que HTML asume.

El atributo `acccept-charset` es otro atributo útil que Laravel nos ofrece. Se asegura de que se usará `UTF-8` a la hora de codificar los caracteres al enviar el formulario. Está bien para la mayoría de las situaciones.

Quizá esos atributos no se ajusten a tu situación. ¿Qué pasa si queremos usar los nuestros propios?

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'ruta/de/destino',
    'method' => 'GET',
    'accept-charset' => 'ISO-8859-1'
)) }}

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

Aquí, hemos usado el índice `method` de nuestra matriz `Form::open()` para especificar que queremos usar `GET` como método para nuestro formulario. También hemos usado el índice `accept-charset` para usar otra codificación diferente.

Este es el nuevo código de nuestro HTML.

```
<!-- app/views/form.blade.php -->

<form method="GET" action="http://demo.dev/ruta/de/destino" accept-charset="ISO-8\
859-1">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
</form>
```

¡Genial! Aunque valoramos la opinión de Laravel con sus ajustes por defecto, tenemos toda la flexibilidad necesaria para reescribirlos. Teniendo en cuenta que tenemos tanto poder, ¿por qué no creamos un formulario que use el verbo HTTP `DELETE`?

Sabemos que podemos reescribir los ajustes usando el índice `method` de la matriz. Vamos a probar.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url' => 'ruta/de/destino',
    'method' => 'DELETE'
)) }}

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

Me parece perfecto, vamos a ver qué tal luce el código HTML.

```
<!-- app/views/form.blade.php -->

<form method="POST" action="http://demo.dev/ruta/de/destino" accept-charset="UTF-\
8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <input name="_method" type="hidden" value="DELETE">
</form>
```

Espera, ¿qué pasa? No queríamos un método `POST`. ¿Qué es ese `input` adicional? Oh si, ya recuerdo. Los formularios HTML son un poco lerdos. HTML4 solo soporta `POST` y `GET` en los métodos de sus formularios. HTML5 soporta métodos adicionales pero no queremos limitar nuestra aplicación a navegadores que solo usen HTML5.

No te preocupes, Laravel tiene otro truco bajo la manga. Usando un `input` oculto llamado `_method` podemos dar otro valor para representar verbos HTTP compatibles con HTML5. Las rutas de Laravel mirarán esta petición POST, verán el atributo `_method` y rutarán a la acción adecuada. Esto funcionará tanto en HTML4 como en HTML5. ¿No es genial?

Recuerda que hemos aprendido cómo subir archivos y obtener información sobre ellos en el último capítulo. Quizá recuerdes también que el formulario necesitaba un `enctype` especial con valor `multipart/form-data` para que los archivos fueran subidos correctamente.

Bueno, Laravel nos da una forma fácil de activar este atributo, veamos cómo funciona.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url'       => 'ruta/de/destino',
    'files'     => true
)) }}

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

Usando un nuevo índice, `files`, con un valor booleano de `true`, Laravel añadirá el atributo necesario para permitir la subida de ficheros. He aquí el código generado del anterior ejemplo.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/ruta/de/destino"
      accept-charset="UTF-8"
      enctype="multipart/form-data">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
</form>
```

He formateado la respuesta un poco para hacer que sea más legible, pero no he cambiado el contenido. Como puedes ver, Laravel usa el tipo de codificación correcta por nosotros. ¡Gracias otra vez Laravel!

En los capítulos de generación de URLs y de enrutado, puede que hayas aprendido que las URLs a rutas nombradas y a acciones de controladores, se pueden generar usando métodos especiales. Descubramos cómo podemos seleccionar esas rutas como el destino de nuestros formularios, usando índices especiales en el método `Form::open()`.

Primero, echemos un vistazo a cómo seleccionar una ruta nombrada. He aquí un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'route'       => 'mi_ruta'
)) }}

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

En vez de usar el índice `url`, usamos `route` y ponemos el nombre de la ruta como valor. ¡Es tan simple como eso! De hecho, es tan fácil como usar acciones de los controladores. Veamos otro ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'action'       => 'MiControlador@miAccion'
)) }}

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

Para que nuestro formulario apunte a la acción del controlador usamos el índice `action` en vez de `url` o `route`, con el valor de la pareja Acción-Controlador.

Bueno, ahora que sabemos cómo abrir formularios, creo que es el momento de que empecemos a añadir algunos campos. ¡Vamos allá!

Campos de formularios

Los formularios no son muy útiles a menos que tengan formas de recoger datos. Veamos algunos de los campos de formularios que podemos generaro usando la librería de Laravel.

Etiquetas de campos

Espera, tenemos algo de lo que ocuparnos antes de tratar con los campos de los formularios en sí. Necesitamos etiquetas para que la gente sepa qué valores introducir. Las etiquetas pueden ser generadas usando el método `Form::label()`. Echemos un vistazo a un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('nombre', 'Nombre') }}
{{ Form::close() }}
```

El primer parámetro del método `Form::label()` coincide con el atributo `name` del campo al que describe. El segundo valor, es el texto de la etiqueta. Echemos un vistazo al código generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="nombre">Nombre</label>
</form>
```

¡Genial! Deberíamos usar las etiquetas siempre que podamos para describir nuestros campos. Predicaré con el ejemplo y me encargaré de incluirlas en todos los ejemplos del capítulo.

Merece la pena destacar que podemos incluir otros atributos en nuestro elemento, facilitando un tercer parámetro al método `label()`. El tercer parámetro es una matriz de parejas de atributos y valores. He aquí un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('nombre', 'Nombre',
                   array('id' => 'nombre')) }}
{{ Form::close() }}
```

En el ejemplo de arriba hemos añadido una `id` con el valor de `nombre` al elemento `label`. He aquí el código generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="nombre" id="nombre">Nombre</label>
</form>
```

¡Brillante! Bueno, ahora sabemos cómo etiquetar nuestros campos, ya es hora de que creemos algunos.

Campos de texto

Los campos de texto pueden ser usados para recoger datos en forma de cadenas. Echemos un vistazo al método generador para este tipo de campo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('nombre', 'Nombre') }}
    {{ Form::text('nombre', 'Taylor Otwell') }}
{{ Form::close() }}
```

El primer parámetro del método `Form::text()` es el atributo `name` para el elemento. Esto es lo que usaremos para identificar el valor del campo en la colección de datos de la petición.

El segundo parámetro es opcional. Es un valor por defecto para el elemento. Veamos el código fuente que ha sido generado por el método.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="nombre">Nombre</label>
    <input name="nombre" type="text" value="Taylor Otwell" id="nombre">
</form>
```

Nuestro elemento ha sido creado, junto a su atributo `name` y a su valor por defecto. Al igual que el método `Form::label()`, nuestro método `Form::text()` puede aceptar un tercer parámetro opcional para facilitar una matriz de atributos HTMLs en forma de pareja de atributo-valor, así:

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('nombre', 'Nombre') }}
    {{ Form::text('nombre', 'Taylor Otwell',
                        array('id' => 'nombre')) }}
{{ Form::close() }}
```

La verdad es, que cada uno de estos generadores acepta una matriz opcional de atributos como parámetro opcional. Funcionan de la misma forma, por lo que no voy a explicar esta característica en cada sección. ¡Pero tenlo en cuenta!

Campos de área de texto

El campo de área de texto, o `textarea`, funciona de manera similar al de texto, excepto que acepta texto en formato multi-línea. Veamos cómo puede ser generado.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('descripcion', 'Descripción') }}
    {{ Form::textarea('descripcion', '¡El mejor campo!') }}
{{ Form::close() }}
```

El primer parámetor es el atributo `name` para el elemento y el segundo, una vez más, es el valor por defecto. He aquí el elemento del formulario tal y como aparece generado en el código HTML.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="descripcion">Descripción</label>
    <textarea name="descripcion"
              cols="50"
              rows="10"
              id="descripcion">¡El mejor campo!</textarea>
</form>
```

Como puedes ver del código generado, Laravel ha puesto algunos valores por defectos, como el número de columnas y filas para el campo. Podemos reescribir esos valores aádiendo nuevos valores para ellos en el tercer y último parámetro opcional. ¡Espero que no te hayas olvidado sobre la matriz de atributos!

Campos de contraseñas

Algunas cosas son secretas. De hecho puedes contarme tus secretos si quieres. NO soy el tipo de persona que los publicaría como bromas en un libro técnico para garantizar algunas risas.

No obstante, si queremos que nuestros usuarios puedan escribir sus secretos más ocultos en nuestra aplicación de forma segura, tenemos que asegurarnos de que las letras no se muestran en la pantalla. El campo de tipo `password` es perfecto para esto. He aquí cómo puedes generar uno.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('secreto', 'Súper Secreto') }}
    {{ Form::password('secreto') }}
{{ Form::close() }}
```

El primer parámetro para el método `Form::password()` es el atributo `name`. Como siempre, hay un parámetro final opcional para una matriz de atributos del elemento.

He aquí el código generado por el anterior ejemplo.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="secreto">Súper Secreto</label>
    <input name="secreto" type="password" value="" id="secreto">
</form>
```

Casillas de verificación

Las casillas de verificación, o `checkbox`, permiten que tus formularios capturen valores booleanos. Echemos un vistazo a cómo podemos generar una casilla de verificación.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('pandas_son_bonitos', '¿Son bonitos los pandas?') }}
    {{ Form::checkbox('pandas_son_bonitos', '1', true) }}
{{ Form::close() }}
```

El primer parámetro del método `Form::checkbox()` es el campo `name` y el segundo parámetro es el campo `value`. El tercer y opcional valor es si la casilla está o no marcada por defecto. Por defecto estará desmarcada.

He aquí el código generado por el anterior ejemplo.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="pandas_son_bonitos">¿Son bonitos los pandas?</label>
    <input checked="checked"
           name="pandas_son_bonitos"
           type="checkbox"
           value="1"
           id="pandas_son_bonitos">
</form>
```

Botones de opción

Los botones de opción, o `radio`, tienen opciones similares a las casillas de verificación. La diferencia es que puedes usar varios botones para ofrecer una forma de selección de un pequeño grupo de valores. He aquí un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('color_panda', 'Los pandas son') }}
    {{ Form::radio('color_panda', 'rojos', true) }} Rojos
    {{ Form::radio('color_panda', 'negros') }} Negros
    {{ Form::radio('color_panda', 'blancos') }} Blancos
{{ Form::close() }}
```

En el ejemplo de arriba, descubrirás que tenemos un número de botones de opción con el mismo atributo `name`. Solo uno de ellos puede estar seleccionado a la vez. Esto permitirá a los usuarios elegir el color de un panda.

Este es el código generado al generar la vista del formulario.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="color_panda">Los pandas son</label>
    <input checked="checked"
           name="color_panda"
           type="radio"
           value="rojos"
           id="color_panda"> Rojos
    <input name="color_panda"
           type="radio"
           value="negros"
           id="color_panda"> Negros
    <input name="color_panda"
           type="radio"
           value="blancos"
           id="color_panda"> Blancos
</form>
```

Desplegables de selección

Los desplegables de selección, o `select`, son otra forma de ofrecer una elección de valores al usuario de tu aplicación. Veamos cómo podemos crear uno.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('color_panda', 'Los pandas son') }}
    {{ Form::select('color_panda', array(
        'rojos'       => 'Rojos',
        'negros'      => 'Negros',
        'blancos'     => 'Blancos'
    ), 'rojos') }}
{{ Form::close() }}
```

El método `Form::select()` acepta un atributo `name` como parámetro, y una matriz de parejas clave-valor como segundo parámetro. La clave para la opción seleccionada, será devuelta con los datos de la petición. El tercer, y opcional, parámetro es la clave del valor que aparecerá marcado por defecto. En el ejemplo de arriba, la opción `Rojos` estará marcada cuando cargue la página.

Este es el código generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="color_panda">Los pandas son</label>
    <select id="color_panda" name="color_panda">
        <option value="rojos" selected="selected">Rojos</option>
        <option value="negros">Negros</option>
        <option value="blancos">Blancos</option>
    </select>
</form>
```

Si queremos, podemos organizar nuestras opciones por categorías. Para activar esta opción, todo lo que tenemos que hacer es facilitar una matriz multidimensional como segundo parámetro. El primer nivel será la categoría, y el segundo nivel será la lista de valores como antes.

Este es un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('bear', 'Bears are?') }}
    {{ Form::select('bear', array(
        'Panda' => array(
            'rojos'       => 'Rojos',
            'negros'     => 'Negros',
            'blancos'     => 'Blancos'
        ),
        'Personaje' => array(
            'pooh'      => 'Pooh',
            'baloo'     => 'Baloo'
        )
    ), 'negros') }}
{{ Form::close() }}
```

Hemos añadido una nueva dimensión a nuestra matriz de valores. El nivel superior está ahora dividido entre `Panda` y `Personaje`. Esos dos grupos serán representados como elementos `optgroup` en el desplegable.

Este es el código generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="bear">Bears are?</label>
    <select id="bear" name="bear">
        <optgroup label="Panda">
            <option value="rojos">Rojos</option>
            <option value="negros" selected="selected">Negros</option>
            <option value="blancos">Blancos</option>
        </optgroup>
        <optgroup label="Personaje">
            <option value="pooh">Pooh</option>
            <option value="baloo">Baloo</option>
        </optgroup>
    </select>
</form>
```

Campo de correo

El tipo de campo de correo tiene los mismos parámetros que el método `Form::text()`. La diferencia, es que el campo `Form::email()` crea un nuevo elemento de HTML5 que validará el valor facilitado en el lado del cliente para asegurarse de que hay una dirección de correo válida.

He aquí otro ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::label('email', 'Dirección e-mail') }}
    {{ Form::email('email', 'me@daylerees.com') }}
{{ Form::close() }}
```

El primer parámetro para el método `Form::email()` es el atributo `name` de nuestro campo. El segundo parámetro es un valor por defecto. Como siempre, y espero que no lo hayas olvidado, hay siempre un último y opcional parámetro que no es otra cosa que una matriz de atributos adicionales.

He aquí cómo queda el código del ejemplo anterior.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="email">Dirección e-mail</label>
    <input name="email"
           type="email"
           value="me@daylerees.com"
           id="email">
</form>
```

Campos de subida de archivos

En el capítulo anterior, aprendimos cómo gestionar la subida de archivos. Veamos cómo podemos generar un campo de subida de archivos.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array(
    'url'   => 'mi/ruta',
    'files' => true
)) }}
    {{ Form::label('avatar', 'Avatar') }}
    {{ Form::file('avatar') }}
{{ Form::close() }}
```

El primer parámetro del método `Form::file()` es el atributo `name` del campo de subida de archivos, no obstante, para que la subida funcione, tenemos que asegurarnos de que el formulario es abierto usando la opción `files` para incluir el tipo de codificación multi-parte.

Este es el código generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8"
      enctype="multipart/form-data">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <label for="avatar">Avatar</label>
    <input name="avatar" type="file" id="avatar">
</form>
```

Campos ocultos

Algunas veces, nuestros campos no están pensados para capturar entrada de datos. Podemos usar campos ocultos para ofrecer datos adicionales con nuestro formulario. Veamos cómo podemos generar un campo oculto.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::hidden('panda', 'luishi') }}
{{ Form::close() }}
```

El primer parámetro para el método `Form::hidden()` es el atributo `name`, apuesto a que no te lo esperabas. El segundo parámetro es, por supuesto, el valor.

He aquí cómo queda el código de la página.looks.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <input name="panda" type="hidden" value="luishi">
</form>
```

Botones de un formulario

Nuestros formularios no quedan bien si no podemos enviarlos. Echemos un vistazo de cerca a los botones que tenemos disponibles.

Botón de enviar

Primero, está el botón de enviar. No hay nada mejor que un clásico. He aquí cómo generarlo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::submit('Guardar') }}
{{ Form::close() }}
```

El primer parámetro de `Form::submit()` es el `value`, que en el caso de un botón es la etiqueta usada para identificar el botón. Como con todos los métodos de generación de campos, este método acepta un último parámetro opcional para facilitar atributos adicionales.

Este es el código HTML generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <input type="submit" value="Guardar">
</form>
```

¡Genial! Ahora podemos enviar nuestros formularios. Echemos un vistazo a una forma alternativa de botón.

Botones normales

Cuando necesitamos un botón que no sea usado para enviar nuestro formulario, podemos usar el método `Form::button()`. He aquí un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::button('Sonríe') }}
{{ Form::close() }}
```

Los parámetros para el método `Form::button()` son exactamente los mismos que para el método `Form::submit()`, mencionado anteriormente. Este es el código generado del ejemplo.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <button type="button">Sonríe</button>
</form>
```

Botones de imagen

En vez de un botón nativo, puedes usar el botón del tipo imagen de HTML5 para enviar tu formulario. He aquí un ejemplo.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::image(asset('mi/imagen.gif', 'submit')) }}
{{ Form::close() }}
```

El primer parámetro para el método `Form::image()` es la URL a la imagen a usar por el botón. He usado el método `asset()` para generar la URL. El segundo parámetro, es el valor del botón.

Este es el código generado.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <input src="https://demo.dev/mi/imagen.gif" type="image">
</form>
```

Botón de resetear

El botón de resetear puede ser usado para limpiar el contenido de tu formulario. Es usado por los usuarios de tu aplicación cuando han cometido algún error. He aquí un ejemplo de cómo generar uno.

```
<!-- app/views/form.blade.php -->

{{ Form::open(array('url' => 'mi/ruta')) }}
    {{ Form::reset('Limpiar') }}
{{ Form::close() }}
```

El primer parámetro del método `Form::reset()` es la etiqueta que quieres que aparezca en el botón. Este es el código generado del ejemplo anterior.

```
<!-- app/views/form.blade.php -->

<form method="POST"
      action="http://demo.dev/mi/ruta"
      accept-charset="UTF-8">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
    <input type="reset" value="Limpiar">
</form>
```

Macros de formularios

En la sección anterior, descubrimos varios métodos para generar campos en tus formularios. Pueden ser usados para ahorrarte mucho tiempo, pero quizá tengas un tipo de campo especial que es específico de tu aplicación.

Por suerte para nosotros, Taylor ya pensó en esto. Laravel viene equipado con un método que te permite definir tus propios generadores, veamos cómo funciona.

¿Dónde deberíamos poner nuestras macros? Ah, ya sé, hagamos un archivo llamado `app/macros.php`, luego podemos incluirlo en nuestro archivo de rutas. He aquí el contenido:

```
<?php

// app/macros.php

Form::macro('nombreCompleto', function()
{
    return '<p>Nombre completo: <input type="text" name="nombre_completo"></p>';
});
```

Para definir una macro, usamos el método `Form::macro()`. El primer parámetro es el nombre que será usado por el método que genera nuestro campo del formulario. Te sugeriría que usaras `camelCase`.

El segundo parámetro del método es una Closure, el valor devuelto por esta closure debe ser el código fuente necesario para construir tu campo.

Creemos una sencilla ruta para mostrar el resultado de nuestra nueva macro de formulario. Aquí está:

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Form::nombreCompleto();
});
```

Nuestra ruta está devolviendo el valor de `nombreCompleto()` en la clase `Form`. Este no es el método normal estático, es algo de magia que Laravel nos ofrece para llamar a neustras propias macros de formularios. El nombre de los métodos coinciden con el apodo que le hemos dado a la macro del formulario. Echemos un vistazo a lo que devuelve la ruta que acabamos de crear.

```
<p>Nombre completo: <input type="text" name="nombre_completo"></p>
```

Como puedes ver, el resultado de la closure ha sido devuelto.

¿Qué pasa si queremos facilitar parámetros a nuestras macros? ¡Por supuesto, sin problemas! Alteremos primero nuestra macro.

```php
<?php

// app/macros.php

Form::macro('nombreCompleto', function($name)
{
    return '<p>Nombre completo: <input type="text" name="'.$name.'"></p>';
});
```

Usando variables en la closure, podemos usar sus valores en el código devuelto. En el ejemplo anterior, hemos añadido al método una forma de recibir el atributo `name` para nuestro nuevo tipo de campo.

Echemos un vistazo a cómo podemos pasar este parámetro.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Form::nombreCompleto('mi_campo');
});
```

Bueno, fue fácil ¿verdad? Tan solo pasamos el valor de nuestro parámetro al método mágico que Laravel nos facilita. Echemos un vistazo nuevamente al resultado de nuestra ruta.

```
<p>Nombre completo: <input type="text" name="mi_campo"></p>
```

Genial, el nuevo valor del nombre ha sido insertado en nuestro campo del formulario. ¡Otra forma de evitar la repetición! ¡Disfruta!

Seguridad de formularios

Es genial cuando recibimos datos de nuestros formularios. Es para lo que son, ¿verdad? El problema es, que si nuestras rutas pueden ser el destino de nuestros propios formularios, ¿qué detiene a una fuente externa el enviar datos peligrosos a nuestro sitio?

Lo que necesitamos es una forma de asegurarnos de que los datos enviados pertenecen a nuestra propia aplicación web. No hay problemas. Laravel puede hacerse cargo por nosotros. Probablemente sepas que este tipo de ataque es conocido como Falsificación de petición de sitios cruzados o CSRF por sus siglas en inglés.

Ahora, ¿recuerdas el campo `_token` oculto sobre el que te pedí que no te preocuparas? Bien, ahora quiero que empieces a preocuparte.

> ARGH. ¡Vamos a morir todos!

Bien hecho. Déjame explicarte lo que es ese token. Es algo así como una frase secreta que Laravel conoce. Ha sido creada usando el valor `secret` de la configuración de nuestra aplicación. Si le decimos a Laravel que revise este valor, podemos asegurarnos de que los datos usados por un formulario, pertenecen a nuestra aplicación.

Podríamos revisar el token nosotros mismos comparando su valor con el resultado del método `Session::token()`, pero Laravel nos ofrece algo mejor. ¿Recuerdas nuestro capítulo de Filtros? Bueno, Laravel ha incluido un filtro `csrf`. Vamos a añadirlo a la ruta.

```
<?php

// app/routes.php

Route::post('/gestionar-formulario', array('before' => 'csrf', function()
{
    // Handle our posted form data.
}));
```

Adjuntando el filtro `csrf` como filtro `before` a la ruta que gestionará los datos de nuestro formulario, podemos asegurarnos de que el campo `_token` está presente y es correcto. Si nuestro token no está presente o ha sido modificado, se lanzará una excepción `Illuminate\Session\TokenMismatchException` y la ruta no será ejecutada.

En un capítulo posterior, aprenderemos cómo adjuntar gestores de errores a cierto tipo de excepciones de manera que podamos gestionar esta circunstancia de forma adecuada.

Si usaste `Form::open()` para crear la apertura del formulario, el token de seguridad estará incluido por defecto. No obstante, si quieres añadir el token de manera manual, simplemente añade el método generador del token. He aquí un ejemplo.

```
<!-- app/views/form.blade.php -->

<form action="{{ url('gestionar-formulario') }}" method="POST">
    {{ Form::token() }}
</form>
```

El método `Form::token()` insertará el campo oculto del token en nuestro formulario. He aquí el código que será generado.

```
<!-- app/views/form.blade.php -->

<form action="http://demo.dev/gestionar-formulario" method="POST">
    <input name="_token" type="hidden" value="83KCsmJF1Z2LMZfhb17ihvt9ks5NEcAwFoR\
FTq6u">
</form>
```

Para resumir, a menos que quieras que las aplicaciones externas y los usuarios puedan enviar datos a tus formularios, deberías usar el filtro `CSRF` para proteger las rutas que gestionan los datos de tus formularios.

Validación

Hace unos meses, me empecé a estresar mucho. El trabajo se acumulaba para el lanzamiento de Laravel 4, lo cual significaba que tenía que crear y construir el nuevo sitio, responder a las modificaciones de la documentación, junto al típico rol de soporte que cualquier programador de un framework tiene que asumir. Fueron tiempos difíciles. Además de todo eso, también tuve la presión de intentar crear Code Bright y hacerlo tan exitoso como Code Happy para poder traer nuevos programadores a la comunidad.

Lo que debería haber hecho es tomarme un descanso en uno de los Parques de Phill. Los Parques de Phill son unos parques de vacaciones ubicados alrededor del mundo que son los destinos de vacaciones de los programadores. Fueron abiertos por el CEO de los Parques de Phill, Phill Sparks, en los años 2000. Los parques son ahora el único lugar al que los programadores pueden ir a desmelenarse y bañarse tranquilamente.

Los parques ofrecen una amplia variedad de actividades para programadores que quieran relajarse tras un año de intenso trabajo. Esto incluye torneos de FizzBuzz, debates de tabulaciones y espacios, etc.

Si hubiera sabido sobre los parques, podría haberme relajado y disfrutar las semanas que precedían al lanzamiento. Bueno, al menos tú los conoces, ¿no? Como lector de Code Bright, tienes acceso a un cupón que te ofrece un 90% de descuento en la entrada de los parques. ¡Aprovecha esta oferta! Simplemente busca a Phill Sparks en IRC, Twitter, Correo o en persona y cítale el código del cupón: `CODE_BRIGHT_TE_TROLLEA`. Si no responde, sigue spameándolo. En algún momento te dará tus vacaciones.

> Entonces, ¿qué tiene esto que ver con la validación?

Bueno, cuando empecé este capítulo, tenía un maravilloso plan en mente. Por desgracia, me he distraído completamente por la diversión y los juegos ofrecidos en los Parques de Phill y ahora no tengo ni idea. Simplemente me inventaré algo. Por ahora ha ido bien, ¿verdad?

Los Parques de Phill son un lugar muy especial para los programadores, y solo los programadores. El problema con ellos es que, sin prestarles atención, puede ser difícil de distinguirlos entre otros rols de la industria de la programación web. Por ejemplo, los diseñadores.

No queremos que los diseñadores se cuelen en los parques como programadores. Comenzarían a juzgar la elección de la decoración y a quejarse por el uso de Comic Sans como fuente del menú de la cafetería. ¡Arruinaría el ambiente por completo! No, no los queremos en el parque. Lo que necesitamos es alguna forma de validar nuestros visitantes para asegurarnos de que son programadores. ¿Lo ves? ¡Te dije que encarrilaría esto!

Lo que validamos son un conjunto de atributos. Pensemos en algunos de los atributos de un visitante de un parque. Oh, oh, ya sé. ¡Podemos crear una lista! ¡Las listas son divertidas!

- Bebida favorita
- Navegador web elegido
- Tipo de vello facial
- Habilidad para hablar a las mujeres

Excelente, tenemos un número de atributos que pueden ser usados para describir a los visitantes al parque. Usando algunas reglas de validación, podemos asegurarnos de que nuestro visitante es un programador. Para el parque, simplemente nos aseguraremos de que los atributos de un visitante cumplen los siguientes requisitos.

- Bebida favorita: Cerveza.
- Navegador web elegido: Google Chrome.
- Tipo de vello facial: Barba.
- Habilidad para hablar a las mujeres: Ninguna.

Nota: Estos valores son solo por diversión. Por ejemplo, conozco muchas mujeres que son programadoras y que son fantásticas en lo que hacen y estoy seguro de que tienen habilidad de hablar con otras mujeres.

De cualquier forma, asegurándonos de que los atributos de un visitante del parque coinciden con los de un programadores, podemos dejarlos pasar en el parque. No obstante, si un escurridizo intenta colarse en recepción con los siguientes atributos…

- Bebida favorita: Frappuchino del Starbucks.
- Navegador web elegido: Mobile Safari.
- Tipo de vello facial: Ninguno.
- Habilidad para hablar a las mujeres: Habla con las mujeres, siempre.

entonces … oh no, tenemos un programador, o un programador Ruby, y no podemos dejarlos pasar por la puerta.

Una vez más, esto es solo por diversión. Por favor, no me envíes un correo de odio. Espera hasta que acabe el capítulo.

Lo que hemos hecho es implementar validación para que podamos asegurarnos de que la gente que entra al parque son programadores y no impostores.

Los Parques de Phill están ahora seguros, ¿pero qué pasa con nuestras aplicaciones? Echemos un vistazo a cómo podemos utilizar la validación en Laravel para asegurarnos de que obtenemos lo que esperamos.

Validación simple

No confíes en tus usuarios. Si hay algo que he aprendido trabajando en esta industria es que si tienes un punto de debilidad en tu aplicación, los usuarios lo encontrarán. Confía en mí cuando te digo que lo explotarán. No les des la oportunidad, usa validación para asegurarte de que recibes buenos datos.

¿A qué nos referimos con buenos datos? Echemos un vistazo a un ejemplo.

Digamos que tenemos un formulario HTML que usamos para conseguir los datos de registro de nuestra aplicación. De hecho, siempre es bueno tener une pequeña sesión de revisión. Vamos a crear un formulario con el sistema de plantillas de Blade y el constructor de formularios.

Esta es nuestra vista.

```
<!-- app/views/form.blade.php -->

<h1>Formulario de registro para los Parques de Phil</h1>

{{ Form::open(array('url' => 'registro')) }}

    {{-- Campo de usuario. ------------------------}}
    {{ Form::label('usuario', 'Usuario') }}
    {{ Form::text('usuario') }}

    {{-- Campo de dirección de correo. -------------------}}
    {{ Form::label('email', 'Dirección de correo') }}
    {{ Form::email('email') }}

    {{-- Campo de contraseña. ------------------------}}
    {{ Form::label('password', 'Contraseña') }}
    {{ Form::password('password') }}

    {{-- Campo de confirmación de contraseña -----------}}
    {{ Form::label('password_confirmation', 'Confirmación de contraseña') }}
    {{ Form::password('password_confirmation') }}

    {{-- Campo de confirmación de contraseña. --------------------}}
    {{ Form::submit('Registrar') }}

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

Wow, ¿no es bonita?

> Es una bonita vista

Excelente, me alegra ver que aun estás mentalmente condicionado tras leer Code Happy.

Puedes ver que nuestro formulario de registro apunta a la ruta `/registro` y por defecto usará el verbo `POST`.

Tenemos un campo de texto para un usuario, un campo de correo para la dirección de correo del usuario, y dos campos de contraseñas para obtener una contraseña y la confirmación. Al final del formulario, tenemos un botón de enviar que podemos usar para enviar el formulario como siempre.

Puede que te hayas fijado en los comentarios de Blade que he añadido al código. Es un hábito mío para ayudarme a navegar por los campos si tengo que volver al formulario. Puedes hacer lo mismo, pero si no quieres, ¡no te preocupes! ¡Programa a tu forma!

Bien, ahora vamos a necesitar un par de rutas para mostrar y gestionar este formulario. Vamos allá y añadámosla a nuestro archivo `routes.php` ahora.

```
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    $data = Input::all();

    // gestionar el formulario...
});
```

Tenemos una ruta `GET /` que será usada para mostrar el formulario, y una ruta `POST /registro` para gestionar su envío.

En la ruta que gestiona el formulario, hemos recogido todos los datos del formulario, pero no podemos usarlos. ¿Por qué? Bueno, mejor te lo enseño.

Ve y abre la ruta `/` y deberías ver el formulario. Bien, no lo rellenes. Tan solo pulsa el botón `Registrar`. La pantalla se volverá blanco y la segunda ruta será activada y nuestro formulario habrá enviado información en blanco. Si usáramos la información vacía, podríamos causar problemas en nuestra aplicación o incluso vulnerabilidades de seguridad. Son datos malos.

Podemos evitar ese desastre implementando una validación para asegurar que nuestros datos son buenos. Antes de que podamos hacer la validación, necesitamos tener una lista de restricciones de validación. Podemos hacerlo en forma de matriz. ¡Estás listo? Bien, echemos un vistazo.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    $data = Input::all();

    $reglas = array(
        'usuario' => 'alpha_num'
    );
});
```

Bien, comencemos poco a poco. Un conjunto de reglas de validación tiene la forma de una matriz asociativa. Las claves de la matriz representan al campo que se valida. El valor consiste en una o varias reglas que serán usadas para validar. Comenzaremos viendo una única validación sobre un único campo.

```
array(
    'usuario' => 'alpha_num'
)
```

En el ejemplo de arriba, queremos validar que el campo `usuario` cumple con la regla de validación `alpha_num`. Esta regla puede ser usada para asegurarte de que un valor contiene únicamente valores alfanuméricos.

Vamos a configurar un objeto de validación para asegurarnos de que nuestra regla funciona correctamente. Para llevar a cabo una validación con Laravel, primero tenemos que crear una instancia del objeto `Validation`. Allá vamos.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'alpha_num'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);
});
```

Podemos usar el método `Validator::make()` para crear una nueva instancia de nuestro validador. El primer parámetro para el método `make()` es una matriz de datos que será validada. En este ejemplo intentamos validar los datos de nuestra petición, pero podríamos validar cualquier otra matriz de datos fácilmente. El segundo parámetro al método, es un conjunto de reglas que serán usadas para validar los datos.

Ahora que nuestra instancia del validador ha sido creada, podemos usarla para mirar si los datos pasan las validaciones que hemos facilitado. Vamos a poner un ejemplo.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
```

```
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'alpha_num'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/');
});
```

Para probar el resultado de la validación, podemos usar el método `passes()` en la instancia del validador. Este método devolverá una respuesta booleana para mostrar si la validación ha pasado o no. Una respuesta `true` indica que los datos pasan todas las reglas de validación. Uno `false` indica que los datos no cumplen con los requisitos.

Usamos una sentencia `if` en el anterior ejemplo para decidir si salvamos los datos o si redirigimos al formulario. Vamos a probarlo visitando la URI `/`.

Primero, entra el valor `zoidberg` en el campo del nombre del usuario y pulsa el botón de enviar. Sabemos que el valor `zoidberg` es alfanumérico, por lo que la validación pasará. Somos recibidos con la siguiente frase.

```
Datos guardados.
```

Genial, eso es lo que esperábamos. Ahora invirtamos los resultados de la validación. Ve y visita nuevamente la URI `/`. Esta vez introduce el valor `!!!`. Sabemos que un signo de exclamación no es un carácter alfanumérico, por lo que a validación fallará. Pulsa el botón y deberíamos volver al formulario de registro.

Adoro el método `passes()`, es muy útil. El único problema es que es un poco optimista. El mundo no es un lugar perfecto, seamos sinceros con nosotros mismos. No somos todos Robert Downey Jr. Permitámosnos ser un poco más pesimistas, ¿no? Excelente, espera... satisfactorio. Vamos a probar el método `fails()`.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'alpha_num'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->fails()) {
        return Redirect::to('/');
    }

    // Normalmente haríamos algo con los datos.
    return 'Datos guardados.';
});
```

El método `fails()` devuelve el booleano opuesto al método `passes()`. ¡Qué maravillosamente pesimista! Eres libre de elegir el método que se ajuste a tu humor actual. Si quieres, también podrías escribir tus sentimientos usando comentarios de PHP.

Algunas reglas de validación son capaces de aceptar parámetros. Cambiemos la regla `alpha_num` por la regla `min`. He aquí el código.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'min:3'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/');
});
```

La regla de validación `min` se asegura de que un valor es mayor o igual que el parámetro facilitado. El parámetro se ofrece tras los dos puntos `:`. Esta regla de validación es un poco especial. Reacciona de manera diferente dependiendo de los datos proveídos.

Por ejemplo, en una cadena, nuestro parámetro `3`, se asegurará de que el valor tiene al menos `3` caracteres. En un valor numérico, se asegurará de que el valor es matemáticamente mayor o igual que nuestro parámetro. Finalmente, en archivos subidos, la validación `min` se asegurará de que el tamaño del archivo en bytes es mayor o igual que el parámetro proveído.

Probemos nuestra regla de validación. Primero visita la página `/` e introduce el valor 'Jo' en el campo de usuario. Si envías el formulario, serás redirigido de vuelta al formulario de registro. Esto pasa porque nuestro valor no es suficientemente largo.

> Eso es lo que ella d...

Oh no, no lo hagas. Esto es un libro serio, no vamos a estropearlo con bromas de 'ella dijo'.

Bueno, ahora hemos usado dos reglas de validación, pero ¿qué pasa si queremos usarlas juntas? No hay problema. Laravel nos permitirá usar cualquier número de reglas de validación en nuestros campos. Echemos un vistazo a cómo podemos hacer eso.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'alpha_num|min:3'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/');
});
```

Como puedes ver, podemos pasar varias validaciones separándolas con una pleca `|`. Este capítulo tendría que haber llegado mucho antes, no obstante, uso Linux en el trabajo y Mac en casa, y tardé en encontrar la tecla de la pleca.

De cualquier forma, hay una forma alternativa de usar varias reglas de validación. Si no eres un abuelo de los 60, puede que no aprecies las plecas. Si quieres, puedes usar una matriz multidimensional para especificar y añadir reglas de validación adicionales.

He aquí un ejemplo.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => array('alpha_num', 'min:3')
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/');
});
```

Laravel es un framework flexible y da a sus usuarios opciones. Usa cualquier método que quieras. Usaré la pleca, para parecer un distinguido caballero Británico.

Reglas de validación

Bien gente, escuchad. Hay un puñado de reglas de validación, así que si vamos a ir por todas, necesito vuestra atención completa. Si estás leyendo esto en un Kindle en la cama, deja el libro ahora y vete a dormir. Necesitas estar más despierto.

Estas son las reglas de validación disponibles en orden alfabético.

accepted

Esta regla puede ser usada para asegurar de que se ha facilitado una confirmación positiva. Pasará si el valor es uno de los siguientes valores: `yes`, `on` o el valor numérico `1`. Su propósito es para cuando necesitas asegurarte de que un usuario ha aceptado algo, por ejemplo, los términos y condiciones.

```
array(
    'campo' => 'accepted'
);
```

---

active_url

Esta regla se asegurará de que es una URL válida. Para hacerlo, usa el método de PHP `checkdnsrr()`, que no solo revisa la estructura de la URL si no que confirma que la URL está disponible en tus registros de DNS.

```
array(
    'campo' => 'active_url'
);
```

---

after

La regla de validación `after` acepta un único parámetro, una cadena que representa una fecha. La regla se asegurará de que el campo contiene una fecha que ocurre tras el parámetro facilitado. Laravel usará el método `startotime()` de PHP para convertir ambos valores y compararlos.

```
array(
    'campo' => 'after:12/12/13'
);
```

---

alpha

La regla de validación `alpha` se asegura de que el campo solo contiene caracteres alfabéticos.

```
array(
    'campo' => 'alpha'
);
```

---

alpha_dash

La regla `alpha_dash` se asegura de que el valor contiene valores alfabéticos y también guiones - y/o guiones bajos _. Esta regla de validación es muy útil para validar porciones de URLs como las urls amigables o 'slugs'.

```
array(
    'campo' => 'alpha_dash'
);
```

---

alpha_num

La regla `alpha_num` se asegura de que el valor consiste en caracteres alfanuméricos. Me gusta usar esta regla en campos de nombres de usuario.

```
array(
    'campo' => 'alpha_num'
);
```

---

before

La regla `before` acepta un único parámetro. El valor en cuestión debe ocurrir antes del parámetro cuando ambos valores hayan sido convertidos usando el método `strtotime()` de PHP. Es el opuesto exacto de `after`.

```
array(
    'campo' => 'before:12/12/13'
);
```

---

between

La regla `between` acepta dos parámetros. El valor que está siendo validado debe tener un tamaño que exista entre esos dos parámetros. El tipo de comparación depende del tipo de los datos comparados. por ejemplo, en campos numéricos, la comparación será matemática. En una cadena, la comparación será basada en la longitud de los caracteres de la cadena. En un archivo, en el tamaño en bytes.

```
array(
    'campo' => 'between:5,7'
);
```

---

confirmed

La regla de validación `confirmed` puede ser usada para asegurar que existe otro campo que coincide con el nombre del campo actual más el sufijo `_confirmation`. El valor que es validado debe coincidir con este otro campo. Un uso para esta campo es para campos de confirmación de contraseñas para asegurar que el usuario no ha introducido un error tipográfico en alguno de los campos. El siguiente ejemplo se asegurará de que el campo `campo`, coincide con el valor de `campo_confirmation`.

```
array(
    'campo' => 'confirm'
);
```

---

date

La regla de validación `date` se asegurará de que nuestro valor es una fecha válida. Será confirmada usando el método `strtotime()` de PHP.

```
array(
    'campo' => 'date'
);
```

---

date_format

La regla de validación `date_format` se asegura de que nuestro valor es una cadena de fecha que coincide con el formato facilitado como parámetro. Para aprender a cómo construir cadenas de formato de fechas, echa un vistazo a la documentación de PHP del método `date()`.

```
array(
    'campo' => 'date_format:d/m/y'
);
```

---

different

La regla de validación `different` se asegura de que el valor validado es diferente al valor contenido en el campo descrito por el parámetro de la regla.

```
array(
    'campo' => 'different:another_campo'
);
```

---

email

La regla de validación `email` se asegura de que el campo validado es una dirección de correo válida. Esta regla es muy útil a la hora de crear formularios de registro.

```
array(
    'campo' => 'email'
);
```

---

exists

La regla `exists` se asegura de que el valor está presente en la tabla de la base de datos identificada por el parámetro de la regla. La columna que será buscada será la del mismo nombre que la del campo que es validado. De manera alternativa, puedes usar un segundo parámetro para especificar el nombre de la columna.

Esta regla puede ser muy útil en campos de registro de formularios para revisar si un usuario ya existe en la base de datos.

```
array(
    'campo' => 'exists:users,usuario'
);
```

Cualquier pareja de parámetros adicionales será añadida a la consulta como cláusulas `where` adicionales. Así:

```
array(
    'campo' => 'exists:users,usuario,rol,admin'
);
```

El ejemplo anterior buscará si el valor existe en la columna `usuario` de la tabla `users`. La columna `rol` debe contener el valor de `admin`.

---

image

La regla de validación `image` se asegura de que el archivo que ha sido subido es una imagen válida. Por ejemplo, la extensión debe ser una de las siguientes: `.bmp`, `.gif`, `.jpeg` o `.png`.

```
array(
    'campo' => 'image'
);
```

---

in

La validación `in` se asegura de que el valor coincide con uno de los parámetros.

```
array(
    'campo' => 'in:rojo,marron,blanco'
);
```

---

integer

¡Esta es fácil! La validación `integer` se asegura de que el campo es un entero. ¡Eso es todo!

```
array(
    'campo' => 'integer'
);
```

---

ip

La regla de validación `ip` revisará que el valor es una dirección IP bien formada.

```
array(
    'campo' => 'ip'
);
```

---

max

La regla de validación `max` es el opuesto exacto de la regla de validación `min`. Se asegurará de que el tamaño del campo que es validado es menor o igual que el parámetro facilitado. Si el campo es una cadena, el parámetro se referirá a la longitud de la misma. Para valores numéricos, la comparación será matemática. Para subidas de archivos, la comparación será contra el tamaño del archivo en bytes.

```
array(
    'campo' => 'max:3'
);
```

---

mimes

La regla de validación `mimes` se asegura de que la cadena facilitada es el nombre de un Mimo Francés. Solo bromeaba. Esta regla se asegurará de que el tipo mime de un archivo subido coincide con uno de los parámetros ofrecidos.

```
array(
    'campo' => 'mimes:pdf,doc,docx'
);
```

---

min

La regla de validación `min` es el opuesto directo de la regla `max`. Puede ser usada para asegurar que el valor de un campo es mayor o igual que el parámetro facilitado. Si el campo es una cadena, el parámetro se referirá a la longitud de la misma. Para valores numéricos, la comparación será matemática. Para subidas de archivos, la comparación será contra el tamaño del archivo en bytes.

```
array(
    'campo' => 'min:5'
);
```

---

not_in

Como el nombre sugiere, esta regla de validación es el opuesto exacto de la regla `in`. Se asegurará de que el valor del campo no existe en la lista de parámetros facilitados.

```
array(
    'campo' => 'not_in:azul,verde,rosa'
);
```

---

numeric

La regla `numeric` s asegura de que el campo contiene un valor numérico.

```
array(
    'campo' => 'numeric'
);
```

---

regex

La regla de validación `regex` es la regla más flexible. Con esta regla puedes ofrecer una expresión regular personalizada como parámetro, que el campo en cuestión debe cumplir. Este libro no cubrirá las expresiones regulares en detalle, teniendo en cuenta que es un tema muy largo y podría valer para un libro completo.

Deberías ver que los caracteres `|` pueden ser usados en expresiones regulares por lo que deberías usar matrices anidadas en vez de plecas `|` para adjuntar varias reglas de validación al usar la regla `regex`.

```
array(
    'campo' => 'regex:[a-z]'
);
```

---

required

La regla de validación `required` puede ser usada para asegurarte de que el campo actual existe en la matriz de datos.

```
array(
    'campo' => 'required'
);
```

---

required_if

La regla de validación `required_if` se asegura de que el campo actual debe estar presente solo si un campo definido por el primer parámetro de la primera regla, coincide con el valor facilitado en el segundo parámetro.

```
array(
    'campo' => 'required_if:usuario,zoidberg'
);
```

---

required_with

La regla `required_with` es usada para asegurar de que el valor actual está presente solo si uno o más de los campos definidos por los parámetros de la regla, está también presente.

```
array(
    'campo' => 'required_with:edad,altura'
);
```

---

required_without

La regla `required_without` es el opuesto directo de la regla `required_with`. Puede ser usada para asegurar de que el campo actual está presente, solo cuando los campos definidos por los parámetros de la regla no lo están.

```
array(
    'campo' => 'required_without:edad,altura'
);
```

---

same

La regla de validación `same` es el opuesto directo de la regla `different`. Es usada para asegurarse de que el valor del campo definido en el parámetro de la regla.

```
array(
    'campo' => 'same:edad'
);
```

---

size

La regla `size` puede ser usada para asegurar de que el valor de un campo es de un tamaño concreto. Si el campo es una cadena, el parámetro se referirá a la longitud de la misma. Para valores numéricos, la comparación será matemática. Para subidas de archivos, la comparación será contra el tamaño del archivo en bytes.

```
array(
    'campo' => 'size:8'
);
```

---

unique

La regla `unique` se asegura de que el valor del campo actual no está presente en la tabla de la base de datos definida por el parámetro de la regla. Por defecto, la regla usará el nombre del campo como columna de la tabla en la que buscar el valor, no obstante, puedes usar una columna alternativa en el segundo parámetro de la regla. Esta regla es útil para revisar si un usuario provee un nombre de usuario único a la hora de gestionar formularios de registro.

```
array(
    'campo' => 'unique:users,usuario'
);
```

Puedes proveer parámetros extra opcionales para listar un número de IDs de columnas que serán ignoradas por la regla.

```
array(
    'campo' => 'unique:users,usuario,4,3,2,1'
);
```

---

url

La regla `url` puede ser usada para asegurar que el campo contiene una URL válida. Al contrario que `active_url`, la regla `url` solo revisa que el formato de la cadena es válido y no revisa los registros de las DNS.

```
array(
    'campo' => 'url'
);
```

Bueno, esas son todas. No fue tan mal, ¿verdad? Aun no hemos acabado. Echemos un vistazo a los mensajes de error.

Mensajes de error

En la primera parte, hemos aprendido cómo llevar a cabo validaciones y cómo redirigir de vuelta a un formulario cuando fallan. no obstante, ese método no ofrece al usuario ningún tipo de feedback constructivo.

Por suerte, Laravel recoge errores describiendo porqué fallan las validaciones. Echemos un vistazo a cómo podemos acceder a esta información.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'alpha_num'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    // Collect the validation error messages object.
    $errors = $validador->messages();

    return Redirect::to('/');
});
```

En el ejemplo superior, verás que podemos acceder a los mensajes de error de validación, usando

el método `messages()` de nuestra instancia del validador. Ahora, teniendo en cuenta que estamos redirigiendo a nuestra ruta del formulario, ¿cómo accedemos a los mensajes de error en nuestros formularios?

Bueno, creo que podríamos usar el método `withErrors()` para esto.

> No te creo, ¡me sigues mintiendo!

¿Oh si? Vamos a ver esto.

```
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario' => 'alpha_num'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/')->withErrors($validador);
});
```

Descubrirás que pasamos la instancia del validador al método `withErrors()` encadenado. Este método deja los mensajes de error en la sesión. Antes de que continuemos, vamos a añadir más reglas de validación en nuestro ejemplo.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario'    => 'required|alpha_num|min:3|max:32',
        'email'      => 'required|email',
        'password'   => 'required|confirm|min:3'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/')->withErrors($validador);
});
```

Ahora veamos cómo podemos acceder a nuestros errores desde la vista del formulario.

```
<!-- app/views/form.blade.php -->

<h1>Formulario de registro para los Parques de Phil</h1>

{{ Form::open(array('url' => 'registro')) }}

    <ul class="errors">
    @foreach($errors->all() as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>

    {{-- Campo de usuario. ------------------------}}
    {{ Form::label('usuario', 'Usuario') }}
    {{ Form::text('usuario') }}

    {{-- Campo de dirección de correo. -------------------}}
    {{ Form::label('email', 'Dirección de correo') }}
    {{ Form::email('email') }}

    {{-- Campo de contraseña. ------------------------}}
    {{ Form::label('password', 'Contraseña') }}
    {{ Form::password('password') }}

    {{-- Campo de confirmación de contraseña -----------}}
    {{ Form::label('password_confirmation', 'Confirmación de contraseña') }}
    {{ Form::password('password_confirmation') }}

    {{-- Campo de confirmación de contraseña. --------------------}}
    {{ Form::submit('Registrar') }}

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

Cuando nuestra vista está cargada, la variable `$errors` es añadida a los datos de la vista. Siempre está ahí, y siempre hay un contenedor para los errores, por lo que no tienes que preocuparte sobre revisar su existencia o contenidos. Si hemos usado `withErrors()` para dejar nuestros errores en la sesión desde una petición anterior, Laravel los añadirá al objeto de errores. ¡Qué conveniente!

Podemos acceder a una matriz de todos los mensajes de error usando el método `all()` sobre la instancia `$errors` de mensajes de error. En la vista de arriba, iteramos sobre la matriz completa de mensajes mostrando cada uno de ellos en un elemento de lista.

Bien, ya basta de parloteo. Vamos a probarlo. Ve y visita la URL `/`. Envía el formulario sin introducir ninguna información y veamos qué pasa.

Somos redirigidos de vuelta al formulario. No obstante, esta vez, aparece un conjunto de mensajes de error.

- The usuario field is required.
- The email field is required.
- The password field is required.

¡Genial! Ahora los usuarios de nuestra aplicación están al tanto de cualquier error de validación. No obstante, es más conveniente para nuestros usuarios si los mensajes de error están más cerca de los campos que describen. Alteremos la vista un poco.

```
<!-- app/views/form.blade.php -->

<h1>Formulario de registro para los Parques de Phil</h1>

{{ Form::open(array('url' => 'registro')) }}

    {{-- Campo usuario. ------------------------}}
    <ul class="errors">
    @foreach($errors->get('usuario') as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>
    {{ Form::label('usuario', 'Usuario') }}
    {{ Form::text('usuario') }}

    {{-- Campo de Dirección de correo. -------------------}}
    <ul class="errors">
    @foreach($errors->get('email') as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>
    {{ Form::label('email', 'Dirección de correo') }}
    {{ Form::email('email') }}

    {{-- Campo de Contraseña . ------------------------}}
    <ul class="errors">
    @foreach($errors->get('password') as $message)
        <li>{{ $message }}</li>
    @endforeach
    </ul>
    {{ Form::label('password', 'Contraseña') }}
```

```
32      {{ Form::password('password') }}
33
34      {{-- Campo de Confirmación de contraseña. -----------}}
35      {{ Form::label('password_confirmation', 'Confirmación de contraseña') }}
36      {{ Form::password('password_confirmation') }}
37
38      {{-- Botón de enviar. --------------------}}
39      {{ Form::submit('Registrar') }}
40
41  {{ Form::close() }}
```

Podemos usar el método `get()` sobre el objeto de errores de validación para obtener un único campo. Simplemente pasa el nombre del campo como primer parámetro del método.

Vamos a volver a enviar el formulario. Esta vez, vamos a colocar únicamente un signo de exclamación `!` en el campo del nombre de usuario. Echemos un vistazo a los errores que aparecen sobre el campo del nombre de usuario.

- The usuario may only contain letters and numbers.
- The usuario must be at least 3 characters.

Eso está mejor. Bueno... un poco mejor. La mayoría de los formularios muestran únicamente un error de validación por campo, para no agobiar al usuario.

Echemos un vistazo a cómo podemos hacer esto con Laravel y sus errores de validación.

```
1   <!-- app/views/form.blade.php -->
2
3   <h1>Formulario de registro para los Parques de Phil</h1>
4
5   {{ Form::open(array('url' => 'registro')) }}
6
7       {{-- Campo de usuario. ------------------------}}
8       <span class="error">{{ $errors->first('usuario') }}</span>
9       {{ Form::label('usuario', 'Usuario') }}
10      {{ Form::text('usuario') }}
11
12      {{-- Campo de Dirección de correo. -------------------}}
13      <span class="error">{{ $errors->first('email') }}</span>
14      {{ Form::label('email', 'Dirección de correo') }}
15      {{ Form::email('email') }}
16
17      {{-- Campo de Contraseña. ------------------------}}
```

```
    <span class="error">{{ $errors->first('password') }}</span>
    {{ Form::label('password', 'Contraseña') }}
    {{ Form::password('password') }}

    {{-- Campo de Confirmación de contraseña. -----------}}
    {{ Form::label('password_confirmation', 'Confirmación de contraseña') }}
    {{ Form::password('password_confirmation') }}

    {{-- Botón de enviar. --------------------}}
    {{ Form::submit('Registrar') }}

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

Usando el método `first()` sobre el objeto de errores de validación y pasando el nombre del campo como parámetro, podemos obtener el primer mensaje de error para ese campo.

Una vez más, envía el formulario con solo una señal de exclamación `!` en el campo del usuario. Esta vez, solo recibimos un error para el primer campo.

- The usuario may only contain letters and numbers.

Por defecto la instancia de mensajes de validación devuelven una matriz vacía o `null` si no hay mensaje. Esto significa que puedes usarla sin tener que saber si existe o no el mensaje. No obstante, si por algún motivo quieres revisar si hay o no un mensaje de error, puedes usar el método `has()`.

```
@if($errors->has('email'))
    <p>Ey, iun eror!</p>
@endif
```

En el ejemplo anterior para los métodos `all()` y `first()`, habrás visto que hemos puesto nuestros errores en elementos HTML. No obstante, si uno de nuestros métodos devuelve `null`, el HTML seguirá siendo mostrado.

Podemos evitar eso pasando el HTML contenido en una cadena formateada como segundo parámetro a los métodos `all()` y `first()`. Por ejemplo, he aquí el método `all()` con los elementos de lista como segundo parámetro.

```
<!-- app/views/form.blade.php -->

<h1>Formulario de registro para los Parques de Phil</h1>

{{ Form::open(array('url' => 'registro')) }}

    <ul class="errors">
    @foreach($errors->all('<li>:message</li>') as $message)
        {{ $message }}
    @endforeach
    </ul>

    {{-- Campo de usuario. ------------------------}}
    {{ Form::label('usuario', 'Usuario') }}
    {{ Form::text('usuario') }}

    {{-- Campo de dirección de correo. -------------------}}
    {{ Form::label('email', 'Dirección de correo') }}
    {{ Form::email('email') }}

    {{-- Campo de contraseña. ------------------------}}
    {{ Form::label('password', 'Contraseña') }}
    {{ Form::password('password') }}

    {{-- Campo de confirmación de contraseña -----------}}
    {{ Form::label('password_confirmation', 'Confirmación de contraseña') }}
    {{ Form::password('password_confirmation') }}

    {{-- Campo de confirmación de contraseña. --------------------}}
    {{ Form::submit('Registrar') }}

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

La porción `:message` del segundo parámetro del método `all()` será reemplazada con el mensaje de error cuando se construya la matriz.

El método `first()` tiene un parámetro opcional similar.

```
<!-- app/views/form.blade.php -->

<h1>Formulario de registro para los Parques de Phil</h1>

{{ Form::open(array('url' => 'registro')) }}

    {{-- Campo de usuario. ------------------------}}
    {{ $errors->first('usuario', '<span class="error">:message</span>') }}
    {{ Form::label('usuario', 'Usuario') }}
    {{ Form::text('usuario') }}

    {{-- Campo de dirección de correo. -------------------}}
    {{ $errors->first('email', '<span class="error">:message</span>') }}
    {{ Form::label('email', 'Dirección de correo') }}
    {{ Form::email('email') }}

    {{-- Campo de contraseña. ------------------------}}
    {{ $errors->first('password', '<span class="error">:message</span>') }}
    {{ Form::label('password', 'Contraseña') }}
    {{ Form::password('password') }}

    {{-- Campo de confirmación de contraseña -----------}}
    {{ Form::label('password_confirmation', 'Confirmación de contraseña') }}
    {{ Form::password('password_confirmation') }}

    {{-- Campo de confirmación de contraseña. --------------------}}
    {{ Form::submit('Registrar') }}

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

Reglas de validación personalizadas

Oh, ya veo. No eres feliz con todo lo que Laravel te da. Querrías tener tus propios métodos. Muy bien, es hora de sacar la artillería. Laravel es lo suficientemente flexible para permitirte especificar tus propias reglas.

Echemos un vistazo a cómo podemos hacer eso.

```php
<?php

// app/routes.php

Validator::extend('increible', function($campo, $value, $params)
{
    return $value == 'increible';
});

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario'   => 'increible',
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/')->withErrors($validador);
});
```

No hay ubicación por defecto para reglas de validación personalizadas por lo que las he añadidos en el archivo `routes.php` para simplificar el ejemplo. Podrías incluir un archivo `validators.php` y usar ahí las validaciones si quieres.

Hemos creado la regla de validación `increible` para nuestro nombre de usuario. Vamos a echar un vistazo a cómo crear la regla.

```php
<?php

// app/routes.php

Validator::extend('increible', function($campo, $value, $params)
{
    return $value == 'increible';
});
```

Para crear una regla de validación personalizada, usamos el método `Validator::extend()`. El primer parámetro del método es el nombre de usuario que será pasado a la regla de validación. Esto es lo que usaremos para adjuntarlo a un campo. El segundo parámetro es una closure. Si la función devuelve como resultado `true`, la validación habrá pasado. Si por el contrario devuelve `false`, la validación habrá fallado.

Los parámetros son pasados a la closure así. El primer parámetro es una cadena que contiene el nombre del campo que está siendo validado. En el ejemplo de arriba, el primer parámetro contendría la cadena `usuario`.

El segundo parámetro contiene el valor del campo.

El tercer parámetro contiene una matriz de cualquier parámetro que haya sido pasado a la regla de validación. Úsalo para personalizar las reglas como necesites.

Si prefieres definir tus reglas de validación en una clase, en vez de en una closure, no podrás. Deja de querer cosas.

Espera, estoy bromeando. Laravel puede hacerlo. Vamos a crear una clase que cumpla esto.

```php
<?php

// app/validators/ValidacionPersonalizada.php

class ValidacionPersonalizada
{
    public function increible($campo, $value, $params)
    {
        return $value == 'increible';
    }
}
```

Como puedes ver, nuestra clase de validación contiene cualquier número de métodos que tienen los mismos parámetros que nuestra closure de validación. Esto significa que una clase de validación puede tener tantas reglas como queramos.

Una vez más, no hay ubicación perfecta para estas clases, por lo que tendrás que definir tu propia estructura del proyecto. Yo suelo poner mi clase de validación en la carpeta `app/validators` y mapeo esa carpeta con Composer.

Bueno, eso es todo.

> Espera, la clase no tiene ningún apodo de validación.

Ah sí, casi me olvido. ¡Bien hecho lector observador! Como veras, para que la regla de validación sea descubierta, necesitamos volver a usar el método `Validator::extend()` de nuevo. Echemos un vistazo.

```
1  <?php
2
3  // app/routes.php
4
5  Validator::extend('increible', 'ValidacionPersonalizada@increible');
```

En esta ocasión, el método `Validator::extend()` recibe una cadena como segundo parámetro. Al igual que el enrutado a un controlador, la cadena consiste en el nombre de la clase y la acción separada por un símbolo `@`.

En un capítulo posterior, aprenderemos cómo extender la clase `Validation` como método alternativo y más avanzado de ofrecer reglas de validación.

Mensajes de validación personalizados

Laravel ofrece mensajes de validación para todas las reglas que trae o por defecto, pero qué ocurre cuando las que vienen no te gustan, quieres escribir las tuyas, o quieres simplemente traducirlas.

Bueno, ¡no te preocupes! Laravel te permite cambiar los mensajes de validación. Simplemente tenemos que crear una matriz adicional y pasarla al método `make()` de la instancia del validador.

> Hey, ya hemos visto el método `Vaidator::make()`

Es cierto, pero una vez más te he mentido.

> ¿Por qué sigues atormentándome?

No estoy del todo seguro. Supongo que lo tengo como hobby a estas alturas. De cualquier forma, echemos un vistazo a una matriz de ejemplo de errores de validación personalizados.

```php
<?php

// app/routes.php

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

Route::post('/registro', function()
{
    // Obtener todos los datos de la petición
    $data = Input::all();

    // Crear el conjunto de validaciones.
    $reglas = array(
        'usuario'   => 'min:3',
    );

    // Crea la matriz de mensajes.
    $mensajes = array(
        'min' => 'Ey tronco, este campo no es suficientemente largo.'
    );

    // Crear instancia del validador.
    $validador = Validator::make($data, $reglas, $mensajes);

    if ($validador->passes()) {
        // Normalmente haríamos algo con los datos.
        return 'Datos guardados.';
    }

    return Redirect::to('/')->withErrors($validador);
});
```

Es un gran ejemplo, vamos a pulsar el botón de enfocar.

> No puedo encontrar el botón enfocar en mi teclado.

Bueno, si tienes un Mac, es el que está arriba de la tecla de tabulación. Se parece a esto ±.

> ¿Estás seguro?

Bueno, ¿tienes otra idea de para qué sirve ese botón?

> Ah, ya te entiendo.

Bien, vamos a centrarnos en el ejemplo.

```
<?php

// Crea la matriz de mensajes.
$mensajes = array(
    'min' => 'Ey tronco, este campo no es suficientemente largo.'
);

// Crear instancia del validador.
$validador = Validator::make($data, $reglas, $mensajes);
```

La matriz de mensajes es un tercer parámetro opcional al método `Validator::make()`. Contiene cualquier mensaje de validación personalizado que quieras facilitar y también sobrescribe los mensajes de validación por defecto. La clave de la matriz representa el nombre de la regla de validación, y el valor es el mensaje a mostrar cuando la regla falla.

En el ejemplo de arriba, hemos cambiado los mensajes de validación para la regla de validación `min`.

También podemos usar este método para ofrecer errores de validación para nuestras reglas de validación personalizadas. Nuestras reglas personalizadas no tendrán mensajes de error por defecto por lo que normalmente es buena idea proveerlos.

```
<?php

// Crea la matriz de mensajes.
$mensajes = array(
    'increible' => 'Por favor, introduce un valor que sea suficientemente increib\
le.'
);

// Crear instancia del validador.
$validador = Validator::make($data, $reglas, $mensajes);
```

Si queremos proveer mensajes de error personalizados para un campo en concreto, podemos hacerlo usando el nombre del campo, un `.` punto, y el tipo de validación como clave de la matriz.

```php
<?php

// Crea la matriz de mensajes.
$mensajes = array(
    'usuario.min' => 'Humm, parece pequeño..'
);

// Crear instancia del validador.
$validador = Validator::make($data, $reglas, $mensajes);
```

El mensaje del ejemplo anterior solo será mostrado si la regla `min` falla para el campo `usuario`. Otros fallos de la regla `min` usarán el mensaje de error por defecto.

Eso es todo lo que tengo que ofrecer sobre validación ahora mismo. En un capítulo posterior aprenderemos cómo extender la clase `Validator` de Laravel y cómo ofrecer mensajes de error traducibles, pero por ahora, movámonos al siguiente capítulo para aprender sobre bases de datos.

Bases de datos

Ahora tengo que hacer una confesión. No soy un gran fan de las bases de datos. Una vez me mordió una cuando niño, y una vez mordido… Vale, vale, bromeo. Es porque una base de datos mató a mi hermano.

Otra vez más, solo bromeaba, no tengo hermanos. No sé siquiera porqué no las disfruto, supongo que me gustan las cosas visuales, las bonitas, las divertidas. Las bases de datos son simplemente grandes tablas de datos, lo anti-divertido. Por suerte, estos días tenemos adorables ORMs que nos permiten acceder a nuestras bases de datos como instancias de objetos. En capítulos posteriores descubrirás más sobre el ORM de Laravel llamado Eloquent. Eloquent es adorable y hace que trabajar con bases de datos sea una experiencia placentera, incluso para los que odian las bases de datos como yo.

Bueno, apartemos mi odio por ahora y hablamos sobre el concepto de una base de datos. ¿Por qué necesitamos una? Bueno, quizá no la necesitemos.

¿Tiene tu aplicación que guardar datos que estén disponibles en futuras peticiones?

> Solo quiero mostrar páginas estáticas.

Bueno, entonces no necesitas una base de datos, pero ¿qué ocurre cuando necesitas almacenar datos y mostrarlos en otras rutas de tu aplicación? Entonces, necesitas un método de almacenamiento y te alegrarás de haber leído los siguientes capítulos.

Abstracción

Entonces, ¿que bases de datos podemos usar con Laravel cuatro? Veamos si alguna de estas te gusta.

- MySQL Community / Standard / Enterprise Server[21]
- SQLite[22]
- PostgreSQL[23]
- SQL Server[24]

[21] http://www.mysql.com/products/
[22] http://www.sqlite.org/
[23] http://www.postgresql.org/
[24] http://www.microsoft.com/en-us/sqlserver/default.aspx

Como puedes ver, tienes varias opciones a la hora de elegir una plataforma de bases de datos. Para este libro, usaré MySQL Community Server Edition[25]. Es una buena plataforma gratis, y una de las más populares usadas en desarrollo.

No tienes que preocuparte sobre el uso de otras bases de datos. Laravel ofrece una capa de abstracción, separa los componentes de bases de datos del framework del código SQL, facilitándote consultas para distintos tipos de bases de datos. Dicho sencillamente, no tienes que preocuparte por la sintaxis SQL, dejemos que Laravel se encargue de ello.

Otra ventaja de usar la capa de abstracción de bases de datos de Laravel es la seguridad. En la mayoría de las situaciones, a menos que diga lo contrario, no tendrás que preocuparte sobre escapar los valores que envías a la base de datos desde Laravel. Laravel escapará los valores por ti, en un esfuerzo de prevenir varias formas de ataques de inyección.

Vamos a medir de la flexibilidad de la capa de abstracción de Laravel por un momento. ¿Entonces, puedes cambiar de servidor de base de datos cuando quieras, sin tener que cambiar nada del código que has escrito, y no tienes que preocuparte por la seguridad? Eso me suena a un gran trabajo que ha sido eliminado de tus proyectos. Escapar valores es por defecto, no tenemos que hacer nada. Dejemos que Laravel se encargue de eso.

Ahora que sabemos que queremos usar una base de datos, aprendamos cómo podemos configurar Laravel para que use una. No te preocupes, ¡es muy sencillo! Primero echa un vistazo a las opciones de configuración.

Configuración

Toda la configuración de base de datos de Laravel está en un archivo ubicado en `app/config/database.php`. Es fácil de recordar, ¿verdad? Demos un viaje por el archivo y veamos algunas de las opciones de configuración disponibles.

```
1  /*
2  |--------------------------------------------------------------------------
3  | PDO Fetch Style
4  |--------------------------------------------------------------------------
5  |
6  | By default, database results will be returned as instances of the PHP
7  | stdClass object; however, you may desire to retrieve records in an
8  | array format for simplicity. Here you can tweak the fetch style.
9  |
10 */
11
12 'fetch' => PDO::FETCH_CLASS,
```

[25] http://www.mysql.com/products/community/

Cando las filas son devueltas desde una consulta que uno de los componentes de bases de dato de Laravel ha ejecutado, por defecto volverán en forma de objetos `stdClass` de PHP. Esto significa que podrás acceder a los datos en sus columnas en un formato similar a este.

```
<?php

echo $libro->nombre;
echo $libro->autor;
```

No obstante, si quieres alterar el formato en que se devuelve cada fila, cambia la opción `fetch` de la configuración a algo que te sea más apropiado. Vamos a alterarlo para usar la opción `PDO::FETCH_ASSOC` que usará una matriz PHP asociativa para almacenar nuestras filas. Ahora podemos acceder a las filas de la siguiente forma.

```
<?php

echo $libro['nombre'];
echo $libro['autor'];
```

Para ver una lista de opciones completas echa un vistazo a la documentación de constantes PDO de PHP[26], echa un vistazo a las que empiezan por `FETCH_`.

Ahora echemos un vistazo a la matriz de conexiones. He aquí como aparece por defecto.

```
<?php

'connections' => array(

    'sqlite' => array(
        'driver'   => 'sqlite',
        'database' => __DIR__.'/../database/production.sqlite',
        'prefix'   => '',
    ),

    'mysql' => array(
        'driver'    => 'mysql',
        'host'      => 'localhost',
        'database'  => 'database',
        'username'  => 'root',
        'password'  => '',
```

[26] http://www.php.net/manual/es/pdo.constants.php

```
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
    ),

    'pgsql' => array(
        'driver'   => 'pgsql',
        'host'     => 'localhost',
        'database' => 'database',
        'username' => 'root',
        'password' => '',
        'charset'  => 'utf8',
        'prefix'   => '',
        'schema'   => 'public',
    ),

    'sqlsrv' => array(
        'driver'   => 'sqlsrv',
        'host'     => 'localhost',
        'database' => 'database',
        'username' => 'root',
        'password' => '',
        'prefix'   => '',
    ),

),
```

Wow, esa es una lista enorme de conexiones por defecto. Eso hace que sea más fácil empezar. Ahora, mirando a la matriz de arriba, puedes pensar que tenemos un índice diferente para cada tipo de base de datos. No obstante, si te fijas con detenimiento, descubrirás que cada una tiene un `driver` que puede ser usado para especificar el tipo de base de datos. Esto significa que podríamos tener fácilmente una matriz de conexiones MySQL diferentes, tal que así:

```
<?php

'connections' => array(

    'mysql' => array(
        'driver'    => 'mysql',
        'host'      => 'localhost',
        'database'  => 'database',
        'username'  => 'root',
```

```
        'password'  => '',
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
    ),

    'mysql_2' => array(
        'driver'    => 'mysql',
        'host'      => 'localhost',
        'database'  => 'database2',
        'username'  => 'root',
        'password'  => '',
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
    ),

    'mysql_3' => array(
        'driver'    => 'mysql',
        'host'      => 'localhost',
        'database'  => 'database3',
        'username'  => 'root',
        'password'  => '',
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
    ),

),
```

Teniendo diferentes conexiones, podemos cambiar las bases de datos a voluntad. De esa forma, nuestra aplicación no solo tiene una base de datos. Supongo que estarás de acuerdo en que es muy flexible.

El primer índice de la matriz de conexiones, es simplemente el apodo que se le da a la conexión, que podemos usar cuando tengamos que llevar a cabo una acción en una base de datos específica de nuestro código. Puedes llamar a tus bases de datos como quieras. Ahora vamos al lío. Vamos a echar un vistazo de cerca a una matriz de conexión individual. Este es nuestro ejemplo una vez más.

```
'my_connection' => array(
    'driver'    => 'mysql',
    'host'      => 'localhost',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
),
```

La opción del driver puede ser usada para especificar el tipo de base de datos a la que pretendemos conectar.

```
'driver'    => 'mysql',
```

Estos son los posibles valores:

- `mysql` - MySQL
- `sqlite` - SQLite
- `pgsql` - PostgreSQL
- `sqlsrv` - SQL Server

Ahora, tenemos el índice `host`, que puede ser usado para especificar la ubicación de red de la máquina que contiene el servidor de bases de datos.

```
'host'      => 'localhost',
```

Puedes o bien usar una dirección IP (`168.122.122.5`) o un nombre de host (`database.ejemplo.com`). En el entorno local de desarrollo usarás normalmente `127.0.0.1` o `localhost` para referirte a la máquina actual.

Bases de datos SQLite

Las bases de datos SQLite están guardadas en una ubicación del disco y por tanto no tienen una entrada como host. Por ese motivo, puedes omitir este índice de un bloque de conexión SQLite.

El siguiente índice de la matriz de conexión es la opción `database`.

```
'database'  => 'nombre_basedatos',
```

Es una cadena para identificar el nombre de la base de datos sobre la que la conexión debe actuar. En el caso de una base de datos SQLite, es usado para especificar el archivo que se usa para almacenar la base de datos, por ejemplo:

```
'database' => __DIR__.'/path/to/database.sqlite',
```

Los índices `username` y `password` pueden ser usados para ofrecer las credenciales de acceso de tu conexión de la base de datos.

```
'username'  => 'dayle',
'password'  => 'emma_w4tson_esta_buena',
```

Bases de dato SQLite

Una vez más, las bases de dato SQLite son un poco diferentes. No tienen credenciales, puedes omitir esos índices en un bloque de conexión SQLite.

El siguiente índice de configuración, es el `charset`, puede ser usado para especificar el conjunto de caracteres por defecto de una conexión a la base de datos.

```
'charset'   => 'utf8',
```

Bases de datos SQLite

¡Lo adivinaste! La base de datos SQLite no soporta esta opción. Deja este índice fuera de la matriz de conexión.

Puedes establecer la codificación usada de una base de datos usando el índice `collation`.

```
'collation' => 'utf8_unicode_ci',
```

Bases de datos SQLite

Una vez más, SQLite decide ser un copo de nieve único. No tienes que usar este índice.

Finalmente, tenemos el opción `prefix`, que puede ser usado para añadir un prefijo común a las tablas de la base de datos.

```
'prefix'    => '',
```

Preparando

Si quieres trabajar con los ejemplos de los próximos capítulos, vas a tener que crear una conexión con una base de datos que funcione. Ve y descarga una plataforma e instálala.

Luego crea una matriz de conexión, y rellena los parámetros necesarios. Ya casi estás. Simplemente tenemos que decirle a Laravel qué conexión debe usar por defecto. Echa otro vistazo al archivo `app/config/database.php`.

```
/*
|--------------------------------------------------------------------------
| Default Database Connection Name
|--------------------------------------------------------------------------
|
| Here you may specify which of the database connections below you wish
| to use as your default connection for all database work. Of course
| you may use many connections at once using the Database library.
|
*/

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

En el índice `default` de la matriz, deberíamos colocar el identificador para la nueva conexión que hemos creado. De esta forma, no tendremos que especificar una conexión cada vez que pretendamos usar la base de datos

Bueno, sé que te emocionas con las bases de datos. ¡Raro, que eres un raro! No gastemos más tiempo. Pasa la página y examinemos el constructor del esquema.

Constructor del Esquema

Bien, así que has decidido que quieres almacenar cosas en la base de datos. La base de datos no es exactamente un almacén clave-valor. En la base de datos, nuestros datos pueden tener estructura, tener distintos tipos, y tener relaciones. Sexis y maravillosas relaciones.

Para poder almacenar datos estructurados, primero tenemos que definir la estructura. Este no es un libro sobre SQL por lo que espero que entiendas el concepto de tabla de una base de datos y sus columnas. En este capítulo vamos a echar un vistazo a la clase `Schema`, que podemos usar para definir la estructura de nuestras tablas. No vamos a almacenar ningún dato en este capítulo, por lo que asegúrate de que tienes estructuras en mente, y no datos.

En el próximo capítulo aprenderás sobre una ubicación ideal para comenzar a crear la estructura de tu base de datos, pero me gusta describir cada cosa en aislamiento. Por ahora, escribiremos nuestro propio código para crear el esquema en closures enrutadas. Bueno, no perdamos más tiempo, y echemos un vistazo rápido al constructor de consultas.

Creando tablas

Para crear una tabla, tenemos que usar el método `create()` de la clase `Schema`. He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
        Schema::create('usuarios', function($tabla)
    {
        // Let's not get carried away.
    });
});
```

El método `Schema::create()` acepta dos parámetros. El primero es el nombre de la tabla que queremos crear. En este caso, estamos creando una tabla llamada `usuarios`. Si la tabla que estamos creando será usada para almacenar datos representando un tipo de objeto, deberíamos nombrar la tabla en minúsculas con el plural del objeto. Las columnas de la base de datos y tablas son nombradas normalmente usando nombres en minúsculas, en el que los espacios son reemplazados con guiones bajos (_) y todos los caracteres son en minúsculas.

El segundo parámetro al método es una closure, con un único parámetro. En el ejemplo de arriba hemos llamado al parámetro `$tabla`, pero puedes llamarlo como quieras. El parámetro `$tabla` puede ser usado para crear la estructura de la tabla.

Añadamos una clave primaria autoincremental a nuestra tabla, de esta forma, las filas de nuestra tabla pueden ser identificadas por un índice único.

```
<?php

// app/routes.php

Route::get('/', function()
{
        Schema::create('usuarios', function($tabla)
    {
        $tabla->increments('id');
    });
});
```

El método `increments()` está disponible en nuestra instancia de `$tabla` para crear una nueva columna autoincremental. Una columna autoincremental será rellenada automáticamente con un valor entero que incrementa con cada fila que se añada. Comenzará en uno. Esta columna será también la clave primaria de la table. El primer parámetro al método `increments` es el nombre de la columna que será creada. Sencillo, ¿verdad?

Vamos a añadir más columnas a esta tabla.

```
<?php

// app/routes.php

Route::get('/', function()
{
        Schema::create('usuarios', function($tabla)
    {
        $tabla->increments('id');
        $tabla->string('usuario', 32);
        $tabla->string('email', 320);
        $tabla->string('password', 60);
        $tabla->timestamps();
    });
});
```

¡Genial! Ahora tenemos una estructura para crear la estructura de nuestra tabla de usuarios. No te preocupes por cada columna ahora, las cubriremos con detalle en la siguiente sección. Primero creemos esta tabla visitando la URI `/` para activar nuestra closure enrutada.

Ahora echemos un vistazo a la estructura de la base de datos que ha sido creada para nosotros. No sé qué tipo de base de datos has decidido elegir, pero voy a usar MySQL en el libro, por lo que echaré un vistazo a la base de datos usando la interfaz de línea de comandos de MySQL. Eres libre de usar cualquier software que quieras.

```
mysql> use myapp;
Database changed

mysql> describe usuarios;
+------------+------------------+-----+----------------+
| Field      | Type             | Key | Extra          |
+------------+------------------+-----+----------------+
| id         | int(10) unsigned | PRI | auto_increment |
| usuario    | varchar(32)      |     |                |
| email      | varchar(320)     |     |                |
| password   | varchar(60)      |     |                |
| created_at | timestamp        |     |                |
| updated_at | timestamp        |     |                |
+------------+------------------+-----+----------------+
6 rows in set (0.00 sec)
```

Bien, he simplificado la tabla un poco para que quepa en el libro, pero espero que pilles la idea. Nuestra estructura ha sido creada usando las guías que hemos creado en nuestro objeto `$tabla`.

Debes estar preguntándote qué métodos y columnas están disponibles en el objeto `$tabla`. ¡Echemos un ojo!

Tipos de columnas

Vamos a examinar los métodos que están disponibles del objeto `$tabla`. Voy a dejar la closure enrutada fuera de los ejemplos para simplificar las cosa por lo que tendrás que usar tu imaginación. Comencemos.

---

increments

El método `increments` añade una clave primaria autoincremental a la tabla. Este es un método muy útil para crear la estructura de modelos ORM de Eloquent, lo cual haremos en un capítulo posterior.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->increments('id');
});
```

El primer y único parámetro para el método `increments()` es el nombre de la columna a crear. Esta es la estructura resultante de la tabla.

```
+-------+------------------+-----+----------------+
| Field | Type             | Key | Extra          |
+-------+------------------+-----+----------------+
| id    | int(10) unsigned | PRI | auto_increment |
+-------+------------------+-----+----------------+
```

---

bigIncrements

Oh, ¿no tenías suficiente con el método `increments`? Bueno, el método `bigIncrements` creará un entero grande, en vez de uno normal.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->bigIncrements('id');
});
```

Al igual que el método `increments`, el método `bigIncrements` aceptará un único parámetro que es una cadena con el nombre de la columna.

```
+-------+---------------------+-----+----------------+
| Field | Type                | Key | Extra          |
+-------+---------------------+-----+----------------+
| id    | bigint(20) unsigned | PRI | auto_increment |
+-------+---------------------+-----+----------------+
```

---

string

El método `string()` puede ser usado para crear columnas `varchar`, que son útiles para almacenar cadenas de valores cortas.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('apodo', 128);
});
```

El primer parámetro es el nombre de la columna a crear, no obstante, hay un segundo parámetro opcional para definir la longitud de la cadena en caracteres. EL valor por defecto es 255.

```
+----------+--------------+
| Field    | Type         |
+----------+--------------+
| apodo | varchar(128) |
+----------+--------------+
```

---

text

El método `text()` puede ser usado para almacenar grandes cantidades de texto que no caben una columna de tipo `varchar`. Por ejemplo, este tipo de columna puede ser sado para contener el cuerpo del texto de un artículo de un blog.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->text('cuerpo');
});
```

El método `text()` acepta un único parámetro. El nombre de la columna que será creada.

```
+--------+------+
| Field  | Type |
+--------+------+
| cuerpo | text |
+--------+------+
```

---

integer

El tipo de columna `integer` puede ser usado para almacenar valores enteros, ¿te sorprende? ¡No encuentro forma de hacerlo más interesante! Bueno, supongo que podría mencionar cómo de útiles son los valores enteros al referenciar un id autoincremental de otra tabla. Podemos usar este método para crear relaciones entre tablas.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->integer('tamano_zapatos');
});
```

El primer parámetro del método `integer()` es el nombre de la columna. El segundo parámetro es un valor booleano que puede ser usado para definir si la columna ha de ser o no autoincremental. El tercer parámetro es usado para definir si el entero tiene o no signo. Un entero con signo puede ser positivo o negativo, no obstante, si defines un entero sin signos, puedes almacenar únicamente valores positivos. Los enteros con signo pueden ir desde −2,147,483,648 hasta 2,147,483,647, mientras que un entero sin signo puede contener valores desde 0 hasta 4,294,967,295.

```
+----------------+---------+
| Field          | Type    |
+----------------+---------+
| tamano_zapatos | int(11) |
+----------------+---------+
```

---

bigInteger

Los valores de entero grande funcionan exactamente igual que los enteros normales, solo que son mucho más grandes. Uno con signo puede variar desde −9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807, mientras que aquellos sin signo tienen un rango desde 0 hasta 18,446,744,073,709,551,615. Un entero de este tamaño es usado normalmente par almacenar el tamaño de mi cintura en pulgadas.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->bigInteger('tamano_cintura');
});
```

Los parámetros del método son exactamente los mismos que los de método `integer()` por lo que no me repetiré. Si ya los has olvidado, quizá deberías volver a revisar la sección sobre `integer`.

```
+----------------+------------+
| Field          | Type       |
+----------------+------------+
| tamano_cintura | bigint(20) |
+----------------+------------+
```

---

mediumInteger

Esta columna es otro tipo de entero, veamos si podemos pasar por todas estas columnas más rápido, ¿vale? Voy a especificar los rangos desde ahora. Con signo el rango es desde −8388608 hasta 8388607. El rango sin signo es desde 0 hasta 16777215.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->mediumInteger('tamano');
});
```

Los parámetros son idénticos a los del método `integer()`.

```
+---------+--------------+
| Field   | Type         |
+---------+--------------+
| tamano  | mediumint(9) |
+---------+--------------+
```

---

tinyInteger

Este es otro tipo de columna de entero. El rango varía desde −128 hasta 127. El rango sin signo va desde 0 hasta 255.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->tinyInteger('tamano');
});
```

Los parámetros son idénticos a los del método `integer()`.

```
+---------+------------+
| Field   | Type       |
+---------+------------+
| tamano  | tinyint(1) |
+---------+------------+
```

---

smallInteger

Este es otro tipo de columna de entero. El rango varía desde −32768 hasta 32767. El rango sin signo va desde 0 hasta 65535.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->smallInteger('tamano');
});
```

Los parámetros son idénticos a los del método `integer()`.

```
+---------+-------------+
| Field   | Type        |
+---------+-------------+
| tamano  | smallint(6) |
+---------+-------------+
```

---

float

La columna `float` es usada para almacenar números de coma flotante. He aquí cómo puede ser definida.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->float('tamano');
});
```

El primer parámetro es el nombre usado para identificar la columna. El segundo y tercer parámetro, que son opcionales, pueden ser usados para especificar la longitud del valor, y el número de lugares decimales a usar para representar el valor. Por defecto, estos valores toman el valor de 8 y 2 respectivamente.

```
+---------+------------+
| Field   | Type       |
+---------+------------+
| tamano  | float(8,2) |
+---------+------------+
```

---

decimal

El método `decimal()` es usado para almacenar... espera... ¡valores decimales! Es muy similar al método `float()`.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->decimal('tamano');
});
```

El método acepta el nombre de la columna como primer parámetro, y dos parámetros opcionales para representar la longitud y el número de decimales que deberían ser usados para definir la columna. Los valores por defecto de estos parámetros son, de nuevo, 8 y 2.

```
+---------+--------------+
| Field   | Type         |
+---------+--------------+
| tamano  | decimal(8,2) |
+---------+--------------+
```

---

boolean

No todos los valores consisten en grandes rangos de dígitos y caracteres. Algunos solo tienen dos estados, `true` o `false`, `1` o `0`. Los tipos de columna booleanos pueden ser usados para representar esos valores.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->boolean('caliente');
});
```

El único parámetro para el método `boolean` es el nombre de la columna que crea.

```
+------------+------------+
| Field      | Type       |
+------------+------------+
| caliente   | tinyint(1) |
+------------+------------+
```

El `tinyint` del ejemplo de arriba no es un error. Los enteros pequeños son usados para representar valores booleanos como `1` o `0`. Me gustaría mencionar que casi quemo la cocina al distraerme mientras escribía esta sección. Pensé que podría ser interesante que conocieras la peligrosa vida de un escritor técnico. ¿No? Está bien, sigamos.

---

enum

El tipo de enumeración almacenará cadenas que están contenidas en una lista de valores permitidos. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $allow = array('Walt', 'Jesse', 'Saul');
    $tabla->enum('who', $allow);
});
```

El primer parámetro es el nombre de la columna que será creada. El segundo parámetro es una matriz de valores que están permitidos para este tipo.

```
+-------+-----------------------------+------+
| Field | Type                        | Null |
+-------+-----------------------------+------+
| who   | enum('Walt','Jesse','Saul') | NO   |
+-------+-----------------------------+------+
```

---

date

Como el nombre sugiere, el método `date()` puede ser usado para crear columnas que almacenan fechas.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->date('cuando');
});
```

El primer y único parámetro es usado para especificar el nombre de la columna que será creada.

```
+---------+------+
| Field   | Type |
+---------+------+
| cuando  | date |
+---------+------+
```

---

dateTime

El método `dateTime()` no solo te permitirá almacenar una fecha, sino también la hora. No bromeo, de verdad que es así. Lo sé, lo sé, muchos de estos métodos son similares, pero confía en mi, este será un buen capítulo de referencia.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->dateTime('cuando');
});
```

Una vez más, el nombre de la columna a ser creada es el único parámetro.

```
+---------+----------+
| Field   | Type     |
+---------+----------+
| cuando  | datetime |
+---------+----------+
```

---

time

¿No quieres que la fecha sea incluida con la hora? ¡Bien! Usa el método `time()`.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->time('cuando');
});
```

Una vez más, el primer y único parámetro del método `time()` es el nombre de la columna que es creada.

```
+---------+------+
| Field   | Type |
+---------+------+
| cuando  | time |
+---------+------+
```

---

timestamp

El método `timestamp()` puede ser usado para almacenar una fecha y su hora en formato `TIMESTAMP`. ¿Sorprendido? ¿No? Oh... vaya, echemos un vistazo a cómo funciona.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->timestamp('cuando');
});
```

El primer y único valor es el nombre de la columna de la base de datos que será creada.

```
+---------+-----------+---------------------+
| Field   | Type      | Default             |
+---------+-----------+---------------------+
| cuando  | timestamp | 0000-00-00 00:00:00 |
+---------+-----------+---------------------+
```

---

binary

El método `binary()` puede ser usado para crear columnas que almacenen datos binarios. Estos tipos de columnas pueden ser útiles para almacenar archivos binarios, como las imágenes.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->binary('imagen');
});
```

El único parámetro del método `binary` es el nombre de la columna que está siendo creada.

```
+--------+------+
| Field  | Type |
+--------+------+
| imagen | blob |
+--------+------+
```

Tipos de columnas especiales

Laravel incluye varios tipos de columnas especiales que tienen usos variados. Echémosle un vistazo. Primero, tenemos el método `timestamps()`.

Este método puede ser usado para añadir dos columnas de tipo `TIMESTAMP` a la tabla. Las columnas `created_at` y `updated_at` pueden ser usadas para indicar cuándo se insertó y actualizó una columna. En un capítulo posterior aprenderemos cómo el ORM de Laravel, Eloquent, puede actualizar estas columnas automáticamente cuando el ORM crea o actualiza una fila. Echemos un vistazo a cómo usar el método `timestamps()`.

```
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->timestamps();
});
```

El método `timestamps()` no acepta ningún parámetro. Esta es la estructura que es creada.

```
+------------+-----------+---------------------+
| Field      | Type      | Default             |
+------------+-----------+---------------------+
| created_at | timestamp | 0000-00-00 00:00:00 |
| updated_at | timestamp | 0000-00-00 00:00:00 |
+------------+-----------+---------------------+
```

Luego tenemos el método `softDeletes()`. A veces querrás marcar una fila como borrada, sin de hecho borrarla. Esto es útil cuando tienes datos que puede que quieras recuperar en el futuro. Con este método, puedes crear una columna indicador en la fila para mostrar que la fila ha sido borrada. La columna creada se llama `deleted_at` y será de tipo `TIMESTAMP`. Una vez más, Eloquent será capaz de actualizar esta columna, sin borrar la fila cuando uses el método borrar. He aquí cómo podemos añadir la columna `deleted_at` a nuestra tabla.

```
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->softDeletes();
});
```

El método `softDeletes()` no acepta ningún parámetro. He aquí la tabla resultante.

```
+------------+-----------+------+
| Field      | Type      | Null |
+------------+-----------+------+
| deleted_at | timestamp | YES  |
+------------+-----------+------+
```

Modificadores de columnas

Los modificadores de columnas pueden ser usados para añadir restricciones adicionales o propiedades a las columnas que hemos creado con el método `create()`. Por ejemplo, antes hemos usado el método `increments()` para crear un índice en una columna que era tanto autoincremental como clave primera. Es un atajo útil, pero echemos un vistazo a cómo podemos convertir otra columna en una clave primaria usando modificadores de columnas.

Primero, hacemos una nueva columna, y declaramos que debe contener valores únicos.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('usuario')->unique();
});
```

Encadenando el método `unique()` al método creador de la columna, le habremos dicho a la base de datos que no se permiten valores duplicados para esta columna. Nuestra clave primaria debería ser usada para identificar valores individuales, por lo que no queremos tener valores duplicados.

Hagamos que la columna `usuario` se la clave primaria.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('usuario')->unique();
    $tabla->primary('usuario');
});
```

Podemos marcar cualquier columna como clave primaria usando el método `primary()` el único parámetro de este método es una cadena que representa el nombre de la columna a marcar como la clave. Describamos la tabla que acabamos de crear.

```
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| usuario  | varchar(255) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

¡Genial! Tenemos una nueva clave primaria.

Este es un buen truco, tanto el método `primary()` como el `unique()` pueden actuar por su cuenta, o fluentemente encadenados a un valor existente. Esto significa que el ejemplo de arriba podría haber sido escrito así:

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('usuario')->unique()->primary();
});
```

El ejemplo de arriba muestra cómo el modificador de la columna puede ser encadenado a una definición de columna existente. De manera alternativa, los modificadores de columnas pueden ser usados en aislamiento para facilitar el nombre de una columna como parámetro.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('usuario');
    $tabla->unique('usuario');
    $tabla->primary('usuario');
});
```

Si no te satisface el tener un único índice primario para tu tabla, puedes usar claves compuestas facilitando una matriz de nombres de columnas al método `primary()` que hemos usado en el ejemplo anterior. Echemos un vistazo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->integer('id');
    $tabla->string('usuario');
    $tabla->string('email');
    $keys = array('id', 'usuario', 'email');
    $tabla->primary($keys);
});
```

Ahora, nuestras tres nuevas columnas actuarán como clave compuesta, en las que una combinación de los valores contenidos en esas columnas serán una referencia única a un rol en particular. Echemos un vistazo al resultado de `describe`.

```
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| id       | int(11)      | NO   | PRI | NULL    |       |
| usuario  | varchar(255) | NO   | PRI | NULL    |       |
| email    | varchar(255) | NO   | PRI | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

Podemos acelerar nuestras consultas marcando las columnas que usamos para buscar información como índices. Podemos usar el método `index()` para marcar una columna como un índice. Puede ser usada con fluidez, así:

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->integer('edad')->index();
});
```

O en aislamiento, así:

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->integer('edad');
    $tabla->index('edad');
});
```

De cualquier forma, el resultado será el mismo. La columna será marcada como índice.

```
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| edad   | int(11) | NO   | MUL | NULL    |       |
+--------+---------+------+-----+---------+-------+
```

También podemos pasar una matriz de nombres de columnas al método `index()` para marcar múltiples columnas como índices. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->integer('edad');
    $tabla->integer('peso');
    $tabla->index(array('edad', 'peso'));
});
```

Esta es la estructura resultante.

```
+---------+---------+------+-----+---------+-------+
| Field   | Type    | Null | Key | Default | Extra |
+---------+---------+------+-----+---------+-------+
| edad    | int(11) | NO   | MUL | NULL    |       |
| peso    | int(11) | NO   |     | NULL    |       |
+---------+---------+------+-----+---------+-------+
```

A veces queremos establecer una restricción en una columna para indicar si debe o no contener un valor `null`. Podemos establecer una columna como *nullable* usando el método `nullable()`. Puede ser usado como parte de un método encadenado, así:

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->nullable();
});
```

Esta es la estructura de la tabla resultante.

```
+---------+--------------+------+
| Field   | Type         | Null |
+---------+--------------+------+
| nombre  | varchar(255) | YES  |
+---------+--------------+------+
```

Como puedes ver, la columna puede contener ahora un valor `null`. Si **no** queremos que la columna permitar un valor `null`, podemos pasar el valor booleano `false` como primer parámetro del método encadenado `nullable()`, así:

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->nullable(false);
});
```

Ahora echemos otro vistazo a la estructura resultante de la tabla.

```
+---------+--------------+------+
| Field   | Type         | Null |
+---------+--------------+------+
| nombre  | varchar(255) | NO   |
+---------+--------------+------+
```

Como puede sver, la columna `nombre` ya no puede contener un valor `null`.

Si quisiéramos que nuestras columnas contuvieran un valor por defecto cuando se crea una nueva fila, podemos ofrecer un valor por defecto encadenando el método `default()` en la nueva definición de la columna. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->default('John Doe');
});
```

El primer y único parámetro para el métod `default()` es el valor por defecto para la columna. Echemos otro vistazo a la estructura resultante.

```
+---------+--------------+------+-----+----------+
| Field   | Type         | Null | Key | Default  |
+---------+--------------+------+-----+----------+
| nombre  | varchar(255) | NO   |     | John Doe |
+---------+--------------+------+-----+----------+
```

Si no facilitamos un valor para la columna `nombre` al crear una nueva fila, por defecto será `John Doe`.

Tenemos otro modificador final que ver. Uno que no es necesario, pero es un buen atajo. ¿Recuerdas crear columnas de enteros en la sección anterior? Usábamos un parámetro booleano para especificar si un entero tenía signo o no. Bueno, podemos usar el método encadenado `unsigned()` sobre una columna de tipo entero para indicar que no debe contener valores negativos. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->integer('edad')->unsigned();
});
```

Esta es la estructura resultante de la tabla, tras usar el método `unsigned()` encadenado.

```
+---------+------------------+
| Field   | Type             |
+---------+------------------+
| edad    | int(10) unsigned |
+---------+------------------+
```

Puedes elegir tanto el valor booleano como el método `unsigned()`, la elección es tuya al cien por cien.

Actualizando tablas

Una vez que una tabla ha sido creada, no hay forma de cambiarla.

> ¿Estás seguro? El encabezado dice...

Estoy seguro, no hay manera alguna.

> Humm, pero el encabezado dice actualizar tablas.

¿No lo vas a dejar pasar no? Bien, iba a echarme una siesta, pero me has convencido. Necesitas aprender cómo actualizar tablas, así que vamos a empezar.

Primero, podemos cambiar el nombre de una tabla que ya hemos creado usando el método `Schema::renombre()`. Vamos a ver un ejemplo.

```
<?php

// Crea la tabla de usuarios
Schema::create('usuarios', function($tabla)
{
    $tabla->increments('id');
});

// Renombra la tabla a idiotas.
Schema::renombre('usuarios', 'idiotas');
```

El primer parámetro del método `renombre()` es el nombre de la tabla que queremos cambiar. El segundo parámetro al método es el nombre nuevo de la tabla.

Si queremos alterar las columnas de una tabla existente, tenemos que usar el método `Schema::table()`. Echemos un vistazo.

```
<?php

Schema::table('ejemplo', function($tabla)
{
    // Modifica la $tabla...
});
```

El método `table()` es casi idéntico al método `create()` que hemos usado antes para crear una tabla. La única diferencia es que actúa sobre una tabla existente que especificamos como primer parámetro al método. Una vez más, el segundo parámetro contiene una Closure con un parámetro de una instancia del constructor de tablas.

Podemos usar cualquiera de los métodos de creación que descubrimos en la sección anterior para añadir nuevas columnas a la tabla existente, He aquí un ejemplo.

```
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->increments('id');
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->string('nombre');
});
```

En el ejemplo de arriba usamos el método `Schema::create()` para crear la tabla `ejemplo` con una clave primaria. Luego usamos el método `Schema::table()` para añadir una columna de cadena al a tabla existente.

Este es el resultado de `describe ejemplo;`:

```
+---------+------------------+-----+----------------+
| Field   | Type             | Key | Extra          |
+---------+------------------+-----+----------------+
| id      | int(10) unsigned | PRI | auto_increment |
| nombre  | varchar(255)     |     |                |
+---------+------------------+-----+----------------+
```

Ahora, puedes usar cualquiera de los métodos de creación de columnas que hemos aprendido en la sección anterior para añadir columnas adicionales a la tabla. No voy a cubrir cada método de creación de nuevo, sus parámetros no han cambiado. Si necesitas un refresco, echa un vistazo de nuevo a la sección anterior.

Si decidimos que no necesitamos una columna en nuestra tabla, podemos usar el método `dropColumn()` para eliminarla. Echemos un vistazo a esto en acción.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->increments('id');
    $tabla->string('nombre');
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropColumn('nombre');
});
```

En ele jemplo de arriba, creamos la tabla `ejemplo` con dos columnas. Luego usamos el método `dropColumn()` para eliminar la columna `nombre` de la tabla. El método `dropColumn` aceptará un parámetro de cadena, que es el nombre de la columna a eliminar.

Así es como se ve la tabla `ejemplo` tras haber ejecutado el código anterior.

```
+-------+------------------+------+-----+----------------+
| Field | Type             | Null | Key | Extra          |
+-------+------------------+------+-----+----------------+
| id    | int(10) unsigned | NO   | PRI | auto_increment |
+-------+------------------+------+-----+----------------+
```

Como puedes ver, la columna `nombre` ha sido eliminada con éxito.

Si quisiéramos eliminar más de una columna a la vez, podemos ofrecer una matriz de nombres de columnas como primer parámetro al método `dropColumn()`...

```php
<?php

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropColumn(array('nombre', 'edad'));
});
```

... o simplemente podemos ofrecer múltiples parámetros de nombres de columnas.

```php
<?php

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropColumn('nombre', 'edad');
});
```

Eres libre de usar cualquier método que se ajuste a tu estilo de programar.

Aunque no tenemos que borrar nuestras columnas. Si queremos, simplemente podemos renombrarlos. Echemos un vistazo a un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre');
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->renombreColumn('nombre', 'apodo');
});
```

El método `renombreColumn()` es usado para cambiar el nombre de una columna. El primer parámetro del método es el nombre de la columna que queremos renombrar, y el segundo parámetro es el nuevo nombre para la columna. Esta es la estructura de tabla resultante para el ejemplo anterior.

```
+----------+--------------+
| Field    | Type         |
+----------+--------------+
| apodo    | varchar(255) |
+----------+--------------+
```

Ahora, ¿recuerdas las claves primarias que hemos creado en la sección anterior? ¿Qué pasa si no queremos que esas columnas sean claves primarias? No hay problema, tan solo eliminamos la clave. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->primary();
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropPrimary('nombre');
});
```

Usando el método `dropPrimary()`, ofrecemos el nombre de la columna como parámetro. Esta columna tendrá el atributo de clave primaria eliminado. He aquí cómo la tabla se ve tras ejecutar el código.

```
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| nombre  | varchar(255) | NO   |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
```

Como puedes ver, el nombre de la columna ya no es una clave primaria. Para eliminar varias claves primarias (en una compuesta), puedes ofrecer una matriz de nombres de columnas como primer parámetro del método `dropPrimary()`. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre');
    $tabla->string('email');
    $tabla->primary(array('nombre', 'email'));
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropPrimary(array('nombre', 'email'));
});
```

Podemos eliminar el atributo `unique` de una columna, usando el método `dropUnique`. Este método acepta un único parámetro, que consiste en el nombre de la tabla, nombre de la columna, y `unique` separado por guiones bajos. He aquí un ejemplo de liminar el atributo `unique` de una columna.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->unique();
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropUnique('ejemplo_nombre_unique');
});
```

Una vez más, podemos pasar una matriz de nombres de columnas en el mismo formato al método `dropUnique()` si queremos. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->unique();
    $tabla->string('email')->unique();
});

Schema::table('ejemplo', function($tabla)
{
    $columns = array('ejemplo_nombre_unique', 'ejemplo_email_unique');
    $tabla->dropUnique($columns);
});
```

Finalmente, podemos eliminar un atributo `index` de una columna de una tabla usando... espera... vale, seguro que lo adivinaste. Podemos usar el método `dropIndex()`. Simplemente indica el nombre de la columna en el mismo formato que usamos para el método `dropUnique()`, eso es, el nombre de la tabla, el nombre de la columna e `index`. Por ejemplo:

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre')->index();
});

Schema::table('ejemplo', function($tabla)
{
    $tabla->dropIndex('ejemplo_nombre_index');
});
```

Por algún motivo, no pude usar una matriz de columnas en el método `dropIndex()`. Le preguntaré a Taylor sobre esto y actualizaré el capítulo con cualquier cambio. Por ahora, sigamos.

Borrando tablas

Para borrar una tabla, primero saca una goma.

Solo bromeaba, podemos borrar una tabla usando el método `Schema::drop()`, veamos el método en acción.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre');
});

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

Para borrar una tabla simplemente pasamos el nombre de la tabla como primer parámetro en el método `Schema::drop()`. Intentemos describir la tabla para ver si existe.

```
mysql> describe ejemplo;
ERROR 1146 (42S02): Table 'myapp.ejemplo' doesn't exist
```

Bueno, supongo que funcionó. Parece que se fue.

Si intentamos eliminar una tabla que no existe, obtendremos un error. Podemos evitarlo usando el método `dropIfExists()`. Como el nombre sugiere, solo borrará la tabla si existe. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->string('nombre');
});

Schema::dropIfExists('ejemplo');
```

Al igual que el método `drop()`, el método `dropIfExists()` acepta un único parámetro. El nombre de la tabla a borrar.

Trucos del esquema

¿Trucos? Quizá no. Sin embargo, esta sección es usada para métodos que no tienen lugar en las secciones anteriores. No malgastemos el tiempo y miremos el primer método.

Podemos usar el método `Schema::connection()` para realizar cambios en el esquema en una base de datos alternativa o conexión. Echemos un ojo a un ejemplo.

```php
<?php

Schema::connection('mysql')->create('ejemplo', function($tabla)
{
    $tabla->increments('id');
});

Schema::connection('mysql')->table('ejemplo', function($tabla)
{
    $tabla->string('nombre');
});
```

El método `connection()` puede ser ubicado antes de cualquier método de la clase `Schema` para formar una cadena. El primer parámetro para el método, es el nombre de la conexión a la base de datos sobre los que actuarán el resto de métodos.

El método `connection()` puede ser muy útil si necesitas escribir una aplicación que use varias bases de datos.

Ahora, tenemos un par de métodos que podemos usar para revisar la existencia de columnas y tablas. Vamos a ver un ejemplo.

```php
<?php

if (Schema::hasTable('autor')) {
    Schema::create('libros', function($tabla)
    {
        $tabla->increments('id');
    });
}
```

Podemos usar el método `hasTable()` para revisar la existencia de una tabla. El primer parámetro del método es el nombre de la tabla que queremos revisar. En el anterior ejemplo hemos creado la tabla `libros` solo si la tabla `autors` existe.

Como puede que hayas descubierto, tenemos unmétodo similar para revisar la existencia de una columna. Veamos otro ejemplo:

```php
<?php

if (Schema::hasColumn('ejemplo', 'id')) {
    Schema::table('ejemplo', function($tabla)
    {
        $tabla->string('nombre');
    });
}
```

Podemos usar el método `Schema::hasColumn()` para revisar si una tabla tiene una columna. El primer parámetro es la tabla y el segundo es el nombre de la columna que queremos buscar. En el ejemplo anterior, añadiremos una columna `nombre` a la tabla `ejemplo`.

Si resulta que eres un genio de las bases de datos, puede que quieras cambiar el motor de almacenamiento usado por la tabla. He aquí un ejemplo.

```php
<?php

Schema::create('ejemplo', function($tabla)
{
    $tabla->engine = 'InnoDB';
    $tabla->increments('id');
});
```

Simplemente cambia el valor del atributo `engine` al nombre que quieras usar. He aquí algunos de los motores de almacenamiento para las bases de dato MySQL:

- MyISAM
- InnoDB
- IBMDM2I
- MERGE
- MEMORY
- EXAMPLE
- FEDERATED
- ARCHIVE
- CSV
- BLACKHOLE

Para más información sobre ellos, por favor consulta la documentación de MySQL[27] para el tema..

En las bases de datos MySQL, puedes reordenar las columnas usando el método `after()`. He aquí un ejemplo.

```
1  <?php
2
3  Schema::create('ejemplo', function($tabla)
4  {
5      $tabla->string('nombre')->after('id');
6      $tabla->increments('id');
7  });
```

Simplemente encadena el método `after()` sobre la columna que quieras reposicionar. El único parámetro del método es el nombre de la columna detrás de la que quieres colocar el campo. Eres libre de usar este método, aunque te recomendaría que crearas tus tablas usando el orden que pretendes, será mucho más limpio.

Bueno, eso es todo lo que tengo sobre esquemas de bases de datos. ¿Por qué no aprendemos algún lugar más adecuado para colocar el código que construye nuestros esquemas? Vamos a pasar al capítulo de esquemas.

[27] http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html

Migraciones

Tenemos un sistema impresionante en la mansión de Dayle. Un sistema que permite que todas las tareas del día sean completadas sin problema por mi ejército de pandas rojos. Permíteme compartirlo contigo. He aquí una lista de tareas para mis mayordomos. ¿Les podráis echar una mano?

- **9:00 AM** - Lavar y vestir a Dayle.
- **10:00 AM** - Cocinar y hacer a la parrilla carnes raras y exóticas para el desayuno.
- **12:00 PM** - (Almuerzo) Los pandas subirán a un árbol y descansarán un rato.
- **02:00 PM** - Pulir la colección de cosas de Apple.
- **04:00 PM** - Preparar el trono de escritura para el próximo capítulo de Code Bright.
- **09:00 PM** - Llevar a Dayle del trono a la cama y arroparlo.

Esa es mi lista para los pandas rojos. Tienen un día ajetreado y no sé qué hacer con ellos. El problema es que la lista tiene un orden muy específico. No queremos que los pandas me lleven a la cama antes de que haya visitado el trono de escritura, si no, no tendrías nuevos capítulos. Además, no tiene sentido el hacer lo mismo dos veces. Los pandas necesitan asegurarse de que las tareas se realizan una sola vez y de manera secuencial.

Los pandas son tan listos, que se les ocurrió la solución al problema. Les di la lista en un bloc de notas con un bolígrafo y, bueno, se alegran mucho cuando les das regalos. Hubo mucho jugueteo. Bueno, no importa, decidieron escribir su propia lista.

Los pandas se dedicaron a escribir una segunda lista. Cuando completaban una tarea, que por supuesto estaba completada siguiendo el orden de la primera, escribían el tiempo y el nombre de la tarea en la segunda lista. De esta forma, no volvían a repetir la misma tarea.

No es mala idea, tengo que admitir. Por suerte, algunos tíos inteligentes inventaron una idea similar para las bases de datos. Echemos un vistazo a las migraciones.

Concepto básico

Al crear tu base de dato, no podrías crear su estructura a mano. Vale, puedes escribir algo de SQL para describir tus columnas, pero ¿qué pasa si por error borras la base de datos? ¿Qué pasa si trabajas en equipo? No tienes que estar pasando las exportaciones de la base de datos para mantener la base de datos sincronizada.

Aquí es cuando las migraciones prueban ser útiles. Son un número de scripts PHP que son usados para cambiar la estructura y/o contenido de tu base de datos. Las migraciones tienen una fecha y hora marcadas por lo que se ejecutan siempre en el orden correcto.

Laravel mantiene un registro de qué migraciones ha ejecutado . De esta forma, solo ejecutará las migraciones adicionales que hayan podido ser añadidas.

Usando migraciones, tanto tú como tu equipo tendréis siempre la misma estructura de la base de datos, de manera consistente y estable. ¿Sabes qué? La acción dice más que las palabras. Vamos a crear una nueva migración y comencemos el proceso de aprendizaje.

Creando migraciones

Para crear una migración tenemos que usar la interfaz de línea de comandos Artisan. Ve y abre una ventana del terminal y dirígete a la carpeta del proyecto. Aprendimos cómo crear esquemas en el capítulo anterior, y te dije que había un lugar mejor para poner el esquema. Bueno, por supuesto que estaba hablando sobre las migraciones. Vamos a recrear la construcción del esquema que hemos usado para crear la tabla de usuarios. Comenzaremos usando Artisan para crear una migración llamada `create_users`.

```
$ php artisan migrate:make create_users
Created Migration: 2013_06_30_124846_create_users
Generating optimized class loader
Compiling common classes
```

Llamamos al método de Artisan, `migrate:make` y lde damos un nombre a nuestra migración. Laravel habrá generado una plantilla de migración en el directorio `app/database/migrations`. La plantilla estará ubicada en un fichero nombrado con el parámetro que le has dado al parámetro `migrate:make`, con una fecha y hora añadidas. En este caso, nuestra plantilla está ubicada en el siguiente archivo.

```
app/database/migrations/2013_06_30_124846_create_users.php
```

Abramos el archivo en nuestro editor de texto y veamos qué tenemos.

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreateUsers extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
```

```
12          public function up()
13          {
14                  //
15          }
16
17          /**
18           * Reverse the migrations.
19           *
20           * @return void
21           */
22          public function down()
23          {
24                  //
25          }
26
27  }
```

Aquí tenemos nuestra clase de migración. Es importante que uses siempre comandos de Artisan para generar migraciones, no te arriesgues a romper las fechas y horas y por tanto la estructura de tu base de datos. Se un buen lector, usa la línea de comandos.

En la clase de migración, tenemos dos métodos públicos `up()` y `down()`. Ahora imagina una línea entre esos dos métodos, o escribe una con un comentario si no aprendiste a usar la imaginación.

A cada lado de la línea, lo opuesto debe ocurrir. Lo que sea que hagas en el método `up()` debes deshacerlo en el método `down()`. Como ves, las migraciones son bi-direccionales. Podemos ejecutar una migración para actualizar la estructura o contenido de nuestra base de datos, pero también podemos deshacer la migración para devolverla a su estado original.

Primero, vamos a rellenar el método `up()`.

```
1   <?php
2
3   /**
4    * Run the migrations.
5    *
6    * @return void
7    */
8   public function up()
9   {
10          Schema::create('users', function($table)
11          {
12                  $table->increments('id');
13                  $table->string('name', 128);
```

```
14                  $table->string('email');
15                  $table->string('password', 60);
16                  $table->timestamps();
17          });
18  }
```

Espero que no haya nada que te confunda en el fragmento de código: Si no lo entiendes, echa un capítulo al capítulo anterior.

Bien, ahora sabemos que lo que sube, tiene que bajar. Por ese motivo, vamos a escribir el método `down()` y creemos el cambio inverso del método `up()`.

Allá vamos...

```
1   <?php
2
3   /**
4    * Reverse the migrations.
5    *
6    * @return void
7    */
8   public function down()
9   {
10          Schema::drop('users');
11  }
```

Vale, vale, supongo que no es lo directamente opuesto. Supongo que querías que eliminara cada columna individualmente y luego la tabla. Bueno, como ves, ambas terminan igual. La tabla `users` es borrada así que, ¿por qué no lo hacemos en una línea?

Antes de que continuemos a la siguiente sección, echemos un vistazo a un par de trucos relativos a crear migraciones. Usando los parámetros `--create` y `--table` sobre el comando `migrate:make`, podemos crear automáticamente algo de código para la creación de una nueva tabla.

Simplemente ejecutamos...

```
1   php artisan migrate:make create_users --create --table=users
```

... y recibimos lo siguiente.

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsers extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::create('users', function(Blueprint $table)
                {
                        $table->increments('id');
                        $table->timestamps();
                });
        }

        /**
         * Reverse the migrations.
         *
         * @return void
         */
        public function down()
        {
                Schema::drop('users');
        }

}
```

¡Genial! Ese atajo nos ha ahorrado algo de tiempo. Habrás notado que además de añadir los métodos `Schema::create()` y `Schema::drop()` para nuestra nueva tabla, Laravle también ha añadido los métodos `increments()` y `timestamps()`. Esto hace sencillo el crear modelos compatibles con Eloquent rápidamente. No te preocupes por Eloquent por ahora, descubriremos más sobre eso pronto.

Un truco final para la creación de migraciones, es cómo almacenarlos en una ubicación diferente a la del directorio `app/database/migrations`. Podemos usar el parámetro `--path` para definir una nueva ubicación para nuestras clases de migración.

```
$ php artisan migrate:make create_users --path=app/migs
Created Migration: 2013_06_30_155341_create_users
Generating optimized class loader
Compiling common classes
```

Ahora la migración será creada en el directorio `app/migs` relativo a la ruta de nuestro proyecto. No obstante, a la hora de ejecutar tus migraciones, Artisan no mirará en esta nueva ubicación por defecto, así que asegúrate de que le indicas dónde encontrar tus migraciones. Descubriremos más sobre eso en la siguiente sección.

Ejecutando migraciones

Teniendo en cuenta que hemos hecho el esfuerzo de crear nuestra migración, sería una vergüenza no ejecutarlas, ¿no? Vamos a preparar la base de datos para usar migraciones. ¿Recuerdas que te dije que Laravel usa una tabla de la base de datos para almacenar el estado de sus migraciones? Bueno, primero necesitamos crear esa tabla.

Ahora, puedes llamar a la tabla como quieras. La configuración está ubicada en `app/config/database.php`.

```
/*
|--------------------------------------------------------------------------
| Migration Repository Table
|--------------------------------------------------------------------------
|
| This table keeps track of all the migrations that have already run for
| your application. Using this information, we can determine which of
| the migrations on disk have not actually be run in the databases.
|
*/

'migrations' => 'migrations',
```

Simplemente cambia el índice `migrations` al nombre de la tabla que quieres usar para registrar el estado de tus migraciones. Como ves, ya hay un valor por defecto.

Podemos instalar nuestra tabla de migraciones ejecutando otro comando de Artisan. Ejecutemos el comando `install` ahora.

```
$ php artisan migrate:install
Migration table created successfully.
```

Ahora, examinemos nuestra base de datos, y busca la tabla `migrations` para ver lo que ha sido creado.

```
mysql> describe migrations;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| migration | varchar(255) | NO   |     | NULL    |       |
| batch     | int(11)      | NO   |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
2 rows in set (0.01 sec)
```

Una nueva tabla con dos campos ha sido creada. No te preocupes sobre la implementación de la tabla de migraciones pero ten por seguro que ha sido creada, y el sistema de migraciones ha quedado instalado.

Ok, te he mentido otra vez. No sé cómo sigue esto pasando. Quizá debería visitar a un psiquiatra o algo. bueno, te dije que teníamos que instalar la tabla de migraciones, y mentí.

Como ves, Laravel creará la tabla automáticamente por nosotros si no existe cuando las migraciones son ejecutadas. Instalará el sistema de migraciones por ti. Al menos ya sabes el comando `migrate:install` ahora, ¿no? Es como si hubiera planeado todo este engaño...

Bien, comencemos y ejecutemos nuestra migración la primera vez. Podemos usar el comando `migrate` para hacerlo.

```
$ php artisan migrate
    Migrated: 2013_06_30_124846_create_users
```

La salida del comando es una lista de migraciones que han sido ejecutadas. Veamos en nuestra base de datos si nuestra tabla `users` ha sido creada.

```
mysql> describe users;
+------------+------------------+
| Field      | Type             |
+------------+------------------+
| id         | int(10) unsigned |
| name       | varchar(128)     |
| email      | varchar(255)     |
| password   | varchar(60)      |
| created_at | timestamp        |
| updated_at | timestamp        |
+------------+------------------+
6 rows in set (0.01 sec)
```

He acortado la tabla un poco para que quede bien con el formato del libro, pero puedes ver que nuestra tabla `users` ha sido creada correctamente. ¡Genial!

Ahora añadamos una columna `title` a la tabla de nuestros usuarios. Puede que estes tentado de abrir la migración que hemos hecho y actualziar el esquema para añadir la nueva columna. Por favor, no lo hagas.

Como verás, si alguno de tus compañeros ha estado trabajando en el proyecto y ya ha ejecutado nuestra primera migración, no recibirá nuestro cambio, y las bases de datos estarían en estados diferentes.

En vez de eso, creemos una nueva migración para alterar nuestra base de datos. Allá vamos.

```
$ php artisan migrate:make add_title_to_users
Created Migration: 2013_06_30_151627_add_title_to_users
Generating optimized class loader
Compiling common classes
```

Habrás notado que le he dado a la migración un nombre descriptivo, deberías seguir este patrón. Vamos a alterar el esquema de nuestra base de datos en el método `up()` para añadir la columna `title`.

```
<?php

use Illuminate\Database\Migrations\Migration;

class AddTitleToUsers extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::table('users', function($table)
                {
                        $table->string('title');
                });
        }

        /**
         * Reverse the migrations.
         *
         * @return void
         */
```

```
25          public function down()
26          {
27                  //
28          }
29
30  }
```

Genial, eso debería añadir la columna que necesitamos a la tabla `users`. Ahora dilo conmigo.

Lo que sube, ha de bajar.

Es correcto, necesitamos usar el método `down()` para esta clase. Vamos a alterar la tabla para eliminar la columna `title`.

```
1   <?php
2
3   use Illuminate\Database\Migrations\Migration;
4
5   class AddTitleToUsers extends Migration {
6
7           /**
8            * Run the migrations.
9            *
10           * @return void
11           */
12          public function up()
13          {
14                  Schema::table('users', function($table)
15                  {
16                          $table->string('title');
17                  });
18          }
19
20          /**
21           * Reverse the migrations.
22           *
23           * @return void
24           */
25          public function down()
26          {
27                  Schema::table('users', function($table)
28                  {
```

```
29                          $table->dropColumn('title');
30                  });
31          }
32
33  }
```

Perfecto, ahora Laravel puede ejecutar nuestra migración, y también deshacer los cambios si lo necesita. Ejecutemos nuestras migraciones de nuevo.

```
1  $ php artisan migrate
2          Migrated: 2013_06_30_151627_add_title_to_users
```

Laravel sabe que nuestra migración anterior ha sido ejecutado ya, por lo que solo ejecuta nuestra última migración. Examinemos la tabla una vez más.

```
1  mysql> describe users;
2  +------------+------------------+
3  | Field      | Type             |
4  +------------+------------------+
5  | id         | int(10) unsigned |
6  | name       | varchar(128)     |
7  | email      | varchar(255)     |
8  | password   | varchar(60)      |
9  | created_at | timestamp        |
10 | updated_at | timestamp        |
11 | title      | varchar(255)     |
12 +------------+------------------+
13 7 rows in set (0.00 sec)
```

Como puedes ver, nuestra nueva columna ha sido añadida a la tabla de usuarios. Si nuestra migración fuera compartida con el resto del equipo, podrían ejecutar `migrate` para alinear sus bases de datos con la nueva estructura.

Si por algún motivo tenemos que alterar alguno de los archivos de migración, podemos usar el comando `migrate:refresh` de Artisan para deshacer todas las migraciones y luego ejecutarlo una vez más. Vamos a intentarlo con nuestra tabla de usuarios.

```
$ php artisan migrate:refresh
Rolled back: 2013_06_30_151627_add_title_to_users
Rolled back: 2013_06_30_124846_create_users
Nothing to rollback.
Migrated: 2013_06_30_124846_create_users
Migrated: 2013_06_30_151627_add_title_to_users
```

Nuestras migraciones han sido deshechas usando los métodos `down()` y luego s ehan vuelto a ejecutar nuevamente en el orden correcto usando los métodos `up()`. Nuestra base de datos está, una vez más, en perfecto estado.

¿Recuerdas cuando usamos el parámetro `--path` en el capítulo anterior para colocar nuestras migraciones en una nueva ubicación. Bueno, te prometí que te mostraría cómo ejecutarlas. Puede que mienta de vez en cuando, pero nunca dejo de cumplir mis promesas. Echemos un vistazo a cómo podemos ejecutar migraciones no estándar.

```
$ php artisan migrate --path=app/migs
Migrated: 2013_06_30_155341_create_users
```

¿Ves? Es fácil. Tan solo volvemos a usar el parámetro `--path` para especificar la ubicación en la que están las migraciones relativas a la raíz de la aplicación.

Te dije que las migraciones osn bi-direccionales, eso significa que deberíamos poder deshacerlas, ¿no? Sigamos.

Deshaciendo migraciones

Deshaciendo migraciones... Sabemos que podemos usar `migrate` para ejecutar nuestras migraciones, pero ¿cómo las deshacemos?

Bueno, asumamos que hemos usado el comando `migrate` para reestructurar nuestra base de datos con la última migración de un miembro de nuestro equipo. Por desgracia, el esquema de nuestros amigos ha roto algo del código, y la aplicación no funciona.

Tenemos que deshacer los cambios que nuestro colega ha hecho. Para hacerlo, podemos usar el comando `rollback`. Vamos a intentarlo.

```
$ php artisan migrate:rollback
Rolled back: 2013_06_30_151627_add_title_to_users
```

Al usar el comando `rollback`, Laravel deshace la última migración que hicimos con `migrate`. Es como si no hubiéramos ejecutado `migrate` esa última vez.

Si queremos deshacer **todas** las migraciones, podemos usar el comando `reset`.

```
$ php artisan migrate:reset
Rolled back: 2013_06_30_151627_add_title_to_users
Rolled back: 2013_06_30_124846_create_users
Nothing to rollback.
```

Debes saber que el comando `reset` no eliminará la tabla de migraciones.

Trucos de migraciones

Oh, ¿quieres más que hacer? Ya veo. No te preocupes. No seré yo el que te lo impida. Vamos a aprender algunas cosas más sobre el sistema de migraciones.

¿Recuerdas la matriz de conexiones que descubrimos en el archivo de configuración de bases de datos en `app/config/database.php`? Podemos realizar nuestras migraciones en otra conexión, usando el parámetro `--database` para seleccionar cualquiera de ellas. Así:

```
$ php artisan migrate --database=mysql
Migrated: 2013_06_30_124846_create_users
Migrated: 2013_06_30_151627_add_title_to_users
```

Ahora tu migración será realizada contra la conexión que hemos llamado `mysql` en el archivo de configuración.

Humm... Aun no estás impresionado. Está bien, tengo otro truco para ti.

Podemos ejecutar nuestras migraciones sin alterar la base de datos, y podemos ver las consultas SQL que serán el resultado de nuestra migración. De esta forma podemos ver lo que hará la siguiente migración, sin arriesgar nuestra base de datos. Es realmente útil para depurar.

Para ver el resultado SQL de un comando de migración, tan solo tienes que añadir el parámetro `--pretend`. He aquí un ejemplo.

```
$ php artisan migrate --pretend
CreateUsers: create table `users` (`id` int unsigned not null
auto_increment primary key, `name` varchar(128) not null,
`email` varchar(255) not null, `password` varchar(60) not null,
`created_at` timestamp default 0 not null, `updated_at` timestamp
default 0 not null) default character set utf8 collate utf8_unicode_ci
AddTitleToUsers: alter table `users` add `title` varchar(255) not null
```

Ahora podemos ver las consultas que hubieran sido ejecutadas contra nuestra base de datos si no hubiera estado el parámetro `--pretend`. Bonito truco, ¿verdad?

Sí, me has pillado…

¡Te lo dije!

En el próximo capítulo hablaremos sobre Eloquent. Eloquent es un método maravilloso de representar las filas de tu base de datos como objetos PHP.

El ORM Eloquent

Hemos aprendido cómo configurar nuestra base de datos y cómo podemos usar el constructor de esquemas para estructurar las tablas de nuestra base de datos, pero ahora es hora de ponernos manos a la obra y aprender cómo podemos almacenar información en la base de datos.

Ahora, algunos de vosotros que ya han lidiado con componentes de bases de datos de Laravel, o incluso aquellos de vosotros que hayan estado usando Laravel 3, puede que os estéis preguntando por qué elijo empezar con el ORM. ¿Por qué no comenzar con las sentencias SQL y luego con la construcción de consultas?

Bueno, echemos un vistazo y pensemos dónde estamos. Eres un programador, un programador PHP. Teniendo en cuenta que estás leyendo este libro, espero que seas un programador de PHP 5+, y que estés usando programación orientada a objetos.

Si estamos describiendo las entidades de nuestra aplicación como objetos, tiene sentido almacenarlas como tal, obtenerlas como tal, etc.

Imaginemos que estamos escribiendo una tienda de libros online.

El diseño de aplicaciones orientadas a objetos nos ha enseñado que tenemos que identificar los objetos de nuestra aplicación. Bueno, una tienda de libros no va a ser muy exitosa sin ningún libro, ¿verdad? Así que es lógico pensar que queremos que un objeto Libro represente a un libro individual usado por nuestra aplicación. Normalmente, nos referiríamos a ellos como Modelos, teniendo en cuenta que representan parte del modelo de negocio de la aplicación. He aquí un ejemplo.

```
1  <?php
2
3  class Libro
4  {
5      /**
6       * Nombre del libro
7       *
8       * @var string
9       */
10     public $nombre;
11
12     /**
13      * Descripción de libro
14      *
15      * @var string
16      */
```

```
17      public $descripcion;
18  }
19
20  $libro = new Libro;
21  $libro->nombre = 'El color de la magia';
22  $libro->descripcion = '¡Rincewind y Dosflores en problemas!';
```

¡Maravilloso!

Hemos creado un libro para representar el libro de Terry Pratchett, "El color de la magia", ¡uno de mis libros favoritos! Ahora almacenemos este libro en nuestra base de datos. Asumiermos que hemos usado el constructor del esquema y que ya tenemos una tabla llamada libros con todas las columnas necesarias.

Primero, tenemos que construir una consulta SQL. Sé que probablemente prepararías una consulta escapada por motivos de seguridad, pero quiero mantener el ejemplo sencillo. Esto debería bastar...

```
1   <?php
2
3   $query = "
4       INSERT INTO
5           libros
6       VALUES (
7           '{$libro->nombre}',
8           '{$libro->descripcion}'
9       );
10  ";
```

Creamos una consulta SQL para insertar nuestro objeto en la base de datos. Esta consulta puede luego ser ejecutada en cualquier adaptado que normalmente usarías.

Creo que es una verdadera vergüenza que tengamos que crear consultas SQL solo para almacenar datos en la base de datos. ¿Por qué molestarte en crear el objeto en primer lugar si vamos a transformarlo en una cadena para almacenarlo? Tendríamos que construir el objeto de nuevo al obtenerlo de la base de datos también. Es malgastar el tiempo...

Creo que deberíamos ser capaces de "lanzar" nuestros objetos directamente a la base de datos sin tener que construir esas consultas SQL horribles. ¿Quizá algo así?

```php
<?php

$libro = new Libro;
$libro->nombre = 'The Colour of Magic';
$libro->descripcion = 'Rincewind and Twoflower in trouble!';
$libro->save();
```

El método `save()` gestionaría el lado SQL de las cosas por nosotros. Persistiendo el objeto en la base de datos. ¡Eso sería genial! Alguien debería hacer esto.

Ya lo han hecho colega.

¿Qué? ¿De verdad? Vaya... Creí que había encontrado la idea que me traería fama y fortuna. Bueno, supongo que es algo bueno.

Ah sí, ahora me acuerdo. Esta funcionalidad es proveída por un mapeador de objetos relacional, o simplemente ORM por sus siglas en inglés. Los ORMs pueden ser usados para permitirnos asociar los objetos de nuestra aplicación con las tablas de la base de datos, y las instancias individuales de esos objetos como las filas.

El ORM se encargará de obtener el objeto y de la persistencia por nosotros, no tendremos que escribir ni una sola línea de SQL. Eso es bueno, ¡porque no soporto el SQL! Es feo y aburrido. Los objetos son mucho más divertidos, ¿verdad?

Muchos ORMs ofrecen también la habilidad de gestionar las relaciones entre múltiples tipos de objetos. Por ejemplo, libros y autores. Autores y editoriales, etc.

Laravel viene con su propio ORM llamado 'Eloquent'. Eloquent hace mucho honor a su nombre. Su sintaxis es bastante bonita y hace que interactuar con la capa de la base de datos de tu aplicación sea una experiencia placentera, en vez de un rollo.

Cuando pienso sobre la capa de almacenamiento, la palabra CRUD me viene ala mente. Son las acciones que se pueden realizar.

- **C** - Crear una nueva fila.
- **R** - Leer filas existentes.
- **U** - Actualizar filas existentes..
- **D** - Borrar filas existentes.

> **N del T**: C de Create, R de Read, U de Update y D de Delete.

Vamos a aprender más sobre Eloquent mirando cada acción en orden. Comenzaremos con la creación de instancias de modelos de Eloquent.

Creando nuevos modelos

Antes de que creemos nuestro primer modelo de Eloquent, necesitamos un ejemplo de datos... Acabo de crear un nuevo juego de PC, así que vamos allá con los videojuegos. Podemos crear algunos objetos para representar videojuegos, pero primero, tenemos que crear el esquema de la tabla.

Crearemos una nueva migración para crear el esquema, para nuestra tabla `juegos`.

```
<?php

// app/database/migrations/2013_07_10_213946_crear_juegos.php

use Illuminate\Database\Migrations\Migration;

class CrearJuegos extends Migration {

	/**
	 * Run the migrations.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('juegos', function($tabla)
		{
			$tabla->increments('id');
			$tabla->string('nombre', 128);
			$tabla->text('descripcion');
		});
	}

	/**
	 * Reverse the migrations.
	 *
	 * @return void
	 */
	public function down()
	{
		Schema::drop('juegos');
	}

}
```

Espero que este ejemplo de código no necesite presentación. Si has descubierto algo confuso en el ejemplo, quizá quieras echar otro vistazo al capítulo de construcción de esquema.

Te habrás dado cuenta que hemos llamado a nuestra tabla `juegos`. Eso es porque pretendemos llamar a nuestro modelo de Eloquent `Juego`. Eloquent es listo, y por defecto buscará la forma plural del nombre del modelo como la tabla a usar para guardar instancias de nuestros objetos. Este comportamiento puede ser sobreescrito, pero mantengámoslo simple por ahora.

Los modelos de Eloquent tienen unos requisitos básicos. Un modelo debe tener una columna autoincremental llamada `id`. Es una clave primaria única que puede ser usada para identificar un único archivo de la tabla. Puedes añadir esta columna a la tabla a la estructura fácilmente, usando el método `increments()`.

Vamos a ejecutar la migración para actualizar la base de datos.

```
$ php artisan migrate
Migrated: 2013_07_10_213946_crear_juegos
```

Ahora podemos empezar. Creemos un nuevo modelo de Eloquent para representar nuestros juegos.

```
<?php

// app/models/Juego.php

class Juego extends Eloquent
{

}
```

Aquí tenemos un modelo Eloquent que podemos usar para representar nuestros juegos. ¿Sorprendido? Supongo que es un poco escueto, pero eso es algo bueno. Muchos ORMs necesitan un mapa XML del esquema de la base de datos o anotaciones para cada una de las columnas de la tabla. No necesitamos nada de eso porque Eloquent asume ciertas cosas.

Vamos a crear un nuevo juego.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $game = new Juego;
    $game->nombre = 'Assassins Creed';
    $game->descripcion = 'Assassins VS templarios.';
    $game->save();
});
```

Ey, ¡eso me suena de algo! ¿No es lo que intentamos hacer en primer lugar? Es limpio y simple. Creamos una nueva instancia de nuestro modelo `Juego` y establecemos sus atributos públicos, los cuales se asocian a columnas de tabla con los valore sque necesitamos. Cuando hemos terminado, simplemente llamamos al método `save()` sobre el objeto para crear la nueva fila en la base de datos.

Visitemos la URI `/`. No esperamos recibir respuesta, teniendo en cuenta que la consulta se ejecutará y no devolverá nada desde la lógica. No obstante, recibimos algo bastante diferente.

Recibimos una pantalla de error. Una bonita pantalla de error. Una pantalla de error realmente bonita. El tío que creó el estilo debe tener habilidades, ¿verdad? Eh... De cualquier forma, ¿cuál es el error?

```
SQLSTATE[42S22]: Column not found: 1054 Unknown column 'updated_at' in 'field lis\
t' (SQL: insert into `juegos` (`nombre`, `descripcion`, `updated_at`, `created_at\
`) values (?, ?, ?, ?)) (Bindings: array ( 0 => 'Assassins Creed', 1 => 'Assassin\
s VS templarios.', 2 => '2013-07-14 16:30:55', 3 => '2013-07-14 16:30:55', ))
```

Cuando Eloquent crea nuestro nuevo modelo, intenta usar las columnas `updated_at` y `created_at` para rellenarlas con la fecha actual. Eso es porque espera que las hayamos añadido con el método `->timestamps()` al crear el esquema. Es algo por defecto ya que no hace daño a nadie el tener esa información. No obstante, si estás usando una tabla existente, o simplemente no quieres tener las columnas presentes, puedes desactivarlo.

Para desactivar esta característica automática en los modelos de Eloquent, añade un nuevo atributo público a tu modelo.

```php
<?php

// app/models/Juego.php

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

El atributo público `$timestamps` es heredado de la clase base Eloquent. Es un valor booleano que puede ser usado para activar o desactivar esta característica. En el ejemplo de arriba lo establecemos a `false`, el cual le hace saber a Eloquent que queremos desactivarlo.

Vamos a visitar la URI `/` una vez más. Esta vez la página muestra un resultado negro. No te preocupes, es porque no devolvemos nada. No recibimos ningún mensaje de error por lo que la consulta SQL debe haber sido ejecutada. Vamos a examinar la tabla `juegos` para ver el resultado.

```
mysql> use myapp;
Database changed
mysql> select * from juegos;
+----+-----------------+--------------------------+
| id | nombre          | descripcion              |
+----+-----------------+--------------------------+
|  1 | Assassins Creed | Assassins VS templarios. |
+----+-----------------+--------------------------+
1 row in set (0.00 sec)
```

Podemos ver que nuestra nueva fila ha sido insertada correctamente. ¡Bien! Hemos insertado un nuevo registro sin escribir una sola línea de SQL. Ese es el tipo de cosas que me gusta.

Quizá te hayas dado cuenta de que no hemos especificado un valor `id` par anuestro objeto. La columna `id` es incrementada automáticamente, por lo que la capa de la base de datos se encargará de eso por nosotros. Es normalmente una mala idea el modificar la columna `id` de un modelo de Eloquent. Intenta evitarlo a menos que sepas lo que estás haciendo.

Hemos usado el atributo `$timestamps` para desactivar las fechas automáticas. En vez de ello, vamos a ver lo que ocurre cuando las activamos. Primero necesitamos alterar el esquema de nuestra base de datos. Es una mala idea modificar manualmente el esquema o actualizar migraciones existentes. Esto es porque el estado de nuestra base de datos puede quedar desincronizado. En vez de ello, vamos a crear una nueva migración que nuestro equipo pueda ejecutar también para recibir nuestros cambios.

```
$ php artisan migrate:make anade_timestamps_a_juegos
Created Migration: 2013_07_14_165416_anade_timestamps_a_juegos
```

Nuestra migración ha sido creada. Habrás descubierto que le hemos dado a la migración un nombre descriptivo para representar nuestras intenciones con la migración. Esto podría ser información útil cuando se ejecute la migración. Usemos el constructor del esquema para añadir las fechas a la tabla.

```
<?php

// app/database/migrations/2013_07_14_165416_anade_timestamps_a_juegos.php

use Illuminate\Database\Migrations\Migration;

class AddTimestampsToJuegos extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::table('juegos', function($tabla)
                {
                        $tabla->timestamps();
                });
        }

        /**
         * Reverse the migrations.
         *
         * @return void
         */
        public function down()
        {
                //
        }

}
```

Hemos usado `Schema::table()` para alterar nuestra tabla `juegos` y añadir el método `timestamps()` para añadir automáticamente esas columnas. Ahora, damas y caballeros, decidlo conmigo.

¡Todo lo que sube ha de bajar!

¡Aprendes rápido! Buent rabajo. Eliminemos las columnas de fechas de la tabla en el método `down()`.

```
<?php

// app/database/migrations/2013_07_14_165416_anade_timestamps_a_juegos.php

use Illuminate\Database\Migrations\Migration;

class AddTimestampsToJuegos extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::table('juegos', function($tabla)
                {
                        $tabla->timestamps();
                });
        }

        /**
         * Reverse the migrations.
         *
         * @return void
         */
        public function down()
        {
                Schema::table('juegos', function($tabla)
                {
                        $tabla->dropColumn('updated_at', 'created_at');
                });
        }

}
```

He usado el método `dropColumn()` del constructor del esquema para eliminar las columnas `updated_at` y `created_at` en el método `down()`. Pensé que habría un adorable método `dropTimestamps()`

pero parece que no. No hay problema. Es un proyecto libre así que mandaré el código cuando tenga algo de tiempo... ¿Te animas?

Ejecutemos nuestra nueva migración para añadir las nuevas columnas a nuestra tabla `juegos`.

```
$ php artisan migrate
Migrated: 2013_07_14_165416_anade_timestamps_a_juegos
```

Ahora tenemos una elección, podemos o bien establecer el atributo `$timestamps` de nuestro modelo a `true` lo cual activaría la característica.

```
<?php

// app/models/Juego.php

class Juego extends Eloquent
{
    public $timestamps = true;
}
```

O... podríamos símplemente eliminarla. Esto es porque el valor por defecto del atributo `$timestamps` de Eloquent es `true`. Su valor será heredado por nuestro modelo.

```
<?php

// app/models/Juego.php

class Juego extends Eloquent
{

}
```

Genial, ahora podemos ejecutar nuevamente la URI `/` para insertar una nueva fila. Examinemos la tabla `juegos` en la base de datos para ver los resultados.

```
mysql> use myapp;
Database changed
mysql> select * from juegos;
+----+----------------+-------------------------+--------------------+--------\
-------------+
| id | nombre         | descripcion             | created_at         | updated\
_at          |
+----+----------------+-------------------------+--------------------+--------\
-------------+
|  1 | Assassins Creed | Assassins VS templarios. | 2013-07-14 17:14:13 | 2013-07\
-14 17:14:13 |
+----+----------------+-------------------------+--------------------+--------\
-------------+
1 row in set (0.00 sec)
```

Como puedes ver, las columnas `created_at` y `updated_at` han sido rellenadas con la fecha por nosotros. ¡Eso ahorra mucho tiempo! Quizá quieras recordar a alguno de tus miembros que ya llevan un año usando tu aplicación, y podrías comparar la fecha actual con la columna `created_at` para ese propósito.

Antes de que nos movamos a la siguiente sección, me gustaría compartir un pequeño truco contigo. Si no quieres que el nombre de la tabla de tu base de datos sea la forma plural del nombre de tu modelo, tendrás que hacérselo saber a Eloquent. Tan solo añade el atributo público `$tabla` a tu modelo y establece su valor al nombre de la tabla que quieras. Eloquent usará ese nombre para todas las consultas futuras relacionadas con ese modelo.

```php
<?php

// app/models/Juego.php

class Juego extends Eloquent
{
    public $tabla = 'gamezilla_roar';
}
```

Si eliges añadir un espacio de nombre a tus modelos, tendrás que usar el atributo `$tabla` para usar nombres de tablas simples. Esto es porque el modelo con un espacio de nombre y clase llamados `MyApp\Models\Juego` resultaría en un nombre de tabla `my_app_models_juegos` para evitar colisiones con paquetes de otros espacios de nombre que también usen la base de datos. También te habrás dado cuenta de que Eloquent es muy inteligente y expandirá los espacios de nombres que usan CamelCase a su variante snake_case.

Hemos aprendido cómo crear nuevas filas en nuestra base de datos usando Eloquent para tratarlos como objetos PHP. Ahora, echemos un vistazo a cómo podemos obtener esas filas.

Leyendo modelos existentes

Eloquent ofrece varios métodos para obtener instancias de modelos. Las veremos todas en un capítulo futuro, pero por ahora nos quedaremos con el método `find()` para obtener una única instancia usando la columna `id`. He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $game = Juego::find(1);
    return $game->nombre;
});
```

Hemos usado el método estático `find()` de nuestro modelo para obtener una única instancia de `Juego` que representa la fila de la base de datos que tiene una `id` con valor de `1`. Ahora podremos acceder a los atributos públicos de la instancia del modelo para obtener los valores de las columnas. Visitemos la URI `/` para ver el resultado.

```
Assassins Creed
```

Genial, nuestro valor existente ha sido obtenido. El método `find()` es heredado de la clase padre Eloquent, y no necesitamos crearlo en nuestro modelo. Como dije antes, hay muchos otros métodos para obtener filas, los cuales cubriremos en capítulos posteriores. Por ahora, echemos un vistazo a cómo podemos actualizar filas de la tabla.

Actualizando modelos existentes

Si has creado un modelo recientemente, lo más probable es que lo hayas asignado a una variable. En la sección anterior, hemos creado una nueva instancia de nuestro modelo `Juego`, asignándolo a la variable `$game`, actualizado sus columnas y usado el método `save()` para persistirlo en la base de datos.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $game = new Juego;
    $game->nombre = 'Assassins Creed';
    $game->descripcion = 'Assassins VS templarios.';
    $game->save();
});
```

Solo porque hayamos guardado nuestra instancia, no significa que no podamos modificarla. Podemos alterar su valor y llamar al método `save()` una vez más para actualizar la fila existente. La primera vez que usamos el método `save()` sobre un nuevo objeto, creará una nueva fila y le asignará el valor de la columna `id`. Las siguientes llamadas al método `save()` persistirán los cambios a las columnas para la fila existente en nuestra base de datos.

Echa un vistazo al siguiente ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $game = new Juego;
    $game->nombre = 'Assassins Creed';
    $game->descripcion = 'Muéstrales por qué, Altair.';
    $game->save();

    $game->nombre = 'Assassins Creed 2';
    $game->descripcion = 'Requiescat in pace, Ezio.';
    $game->save();

    $game->nombre = 'Assassins Creed 3';
    $game->descripcion = 'Rompe algunas caras, Connor.';
    $game->save();
});
```

Podrías imaginar que el ejemplo anterior crearía tres entradas en la tabla `juegos` pero te equivocarías. Quizá te hayas dado cuenta de que solo estamos creando una única nueva instancia de la clase `Juego`.

Todas las llamadas futura al método `save()` servirán para modificar esta fila de la base de datos. El último estado guardado del objeto estará presente en la base de datos.

Voy a truncar la tabla `juegos` de la base de datos para demostrar este ejemplo. Ve y pruébalo también si estás siguiéndome en casa.

```
mysql> truncate juegos;
Query OK, 0 rows affected (0.00 sec)
```

Ahora visitemos la URI `/` una vez más para ejecutar la lógica de la ruta. He aquí el contenido resultante de la tabla `juegos`.

```
mysql> select * from juegos;
+----+------------------+-----------------------------+---------------------+--\
------------------+
| id | nombre           | descripcion                 | created_at          | u\
pdated_at          |
+----+------------------+-----------------------------+---------------------+--\
------------------+
|  1 | Assassins Creed 3 | Rompe algunas caras, Connor. | 2013-07-14 17:38:50 | 2\
013-07-14 17:38:50 |
+----+------------------+-----------------------------+---------------------+--\
------------------+
1 row in set (0.00 sec)
```

Como puedes ver, `Assassins Creed 3` fue actualizado por nuestro último método `save()`.

El método de antes muy útil cuando ya tienes una referencia a una instancia existente de nuestro modelo, ¿pero qué pasa si no? ¿Qué pasa si creaste el modelo hace mucho tiempo? Podemos simplificar el método `find()` para obtener una instancia que represente una fila existente de la base de datos, y alterarla como queramos.

He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $game = Juego::find(1);
    $game->nombre = 'Assassins Creed 4';
    $game->descripcion = 'Rayos y truenos, Edward.';
    $game->save();
});
```

Al inspeccionar la tabla `juegos`, descubrimos que nuestra fila fue insertada con una `id` con valor `1`. Usando el método estático `find()` en nuestro modelo, y la `id` conocida, podemos obtener una instancia de `Juego` que represente la fila existente. Una vez que la nueva instancia ha sido devuelta, podemos modificar el valor de la columna y usar el método `save()` como hicimos antes.

Esta es la fila resultante.

```
mysql> select * from juegos;
+----+------------------+--------------------------+--------------------+------\
---------------+
| id | nombre           | descripcion              | created_at         | updat\
ed_at          |
+----+------------------+--------------------------+--------------------+------\
---------------+
|  1 | Assassins Creed 4 | Rayos y truenos, Edward. | 2013-07-14 17:38:50 | 2013-\
07-14 17:49:28 |
+----+------------------+--------------------------+--------------------+------\
---------------+
1 row in set (0.00 sec)
```

Como puedes ver, nuestra fila existente ha sido actualizada acordemente. Una vez más, no hemos escrito ni una sola línea de SQL. Solo bonito y elocuente PHP. También habrás visto que la columna `updated_at` ha sido actualizada acordemente. ¡Muy útil!

Eliminando modelos existentes

Borrar modelos de Eloquent es un proceso simple. Primero necesitamos hacernos con una instancia del modelo que queremos borrar. Para ello podríamos usar el método `find()` que hemos descubierto anteriormente.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $game = Juego::find(1);
});
```

Una vez que tenemos las pezuñas sobre una instancia de un modelo Eloquent, podemos usar el método `delete()` para eleminar la fila representada por el modelo de la base de datos.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $game = Juego::find(1);
    $game->delete();
});
```

También podemos borrar una única instancia o múltiples instancias de nuestros modelos, usando sus `ids`, y el método estático `destroy()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Juego::destroy(1);
});
```

Para destruir múltiples registros, puedes pasar varios `ids` como parámetros al método `destroy()`...

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Juego::destroy(1, 2, 3);
});
```

... o una matriz de `ids`, así:

```php
<?php

// app/routes.php

Route::get('/', function()
{
    Juego::destroy(array(1, 2, 3));
});
```

¡Es cosa tuya!

SQL ofrece un número de formas diferentes y complejas de consultar un conjunto de registros. No te preocupes, Eloquent también puede encargarse de esto. En el próximo capítulo tenemos que aprender los distintos métodos de consultas disponibles en Eloquent. Vamos, ¡pasa la página!

Consultas de Eloquent

En el capítulo anterior, hemos descubierto cómo expresar nuestras filas de la base de datos como columnas, nuestras tablas como clases. Esto elimina la necesidad de escribir sentencias usando SQL y resulta en que el código es mucho más legible. Estamos escribiendo PHP, ¿verdad? ¿Por qué complicarnos la vida añadiendo otro lenguaje?

Bueno, hay cosas buenas sobre SQL. Por ejemplo, la parte de Q. En inglés son *querys* que son las consultas. Con SQL podemos usar varias comparaciones complejas y usar aritmética pera obtener solo los campos que necesitamos. Replicar **toda** esta funcionalidad con Eloquent sería una tarea tremenda, pero por suerte, Eloquent tiene métodos alternativos para las consultas más útiles. Para las cosas que faltan, podemos usar consultas corrientes y molientes que devolverán instancias de modelos Eloquent. Echaremos un vistazo a esto un poco más tarde. Primero preparemos nuestra base de datos para este capítulo.

Preparación

Pronto aprenderás cómo rellenar la base de datos con datos de ejemplos usando una técnica conocida como *seeding* pero por ahora crearemos algunos registros de prueba, en la base de datos, usando Eloquent. No usaremos nada nuevo aquí, usaremos las habilidades que hemos aprendido en los capítulos recientes.

Primero, tenemos que crear una migración para crear el esquema de nuestra tabla de ejemplo. Vamos a usar álbumes de música como datos demostración. Vamos a crear una migración para construir la tabla `albums`.

```
1 $ php artisan migrate:make crear_albums
2 Created Migration: 2013_07_21_103250_crear_albums
```

Ahora, rellenemos los métodos con el código necesario para crear la nueva tabla `albums`.

```php
<?php

use Illuminate\Database\Migrations\Migration;

// app/database/migrations/2013_07_21_103250_crear_albums.php

class CrearAlbums extends Migration {

        /**
         * Run the migrations.
         *
         * @return void
         */
        public function up()
        {
                Schema::create('albums', function($table)
                {
                        $table->increments('id');
                        $table->string('titulo', 256);
                        $table->string('artista', 256);
                        $table->string('genero', 128);
                        $table->integer('ano');
                });
        }

        /**
         * Reverse the migrations.
         *
         * @return void
         */
        public function down()
        {
                Schema::drop('albums');
        }

}
```

En el método `up()` de nuestra migración, usamos `Schema` para crear una nueva tabla llamada `albums`. La tabla contendrá columnas `varchar` para el título del álbum, el artista que lo toca, y el género. También tenemos un campo `id` autoincremental requerido por Eloquent, y finalmente un campo entero para almacenar el año de lanzamiento del álbum.

En el método `down()` eliminamos la tabla, restaurando la base de datos a su forma original.

Vamos a ejecutar nuestra migración para estructurar la base de datos.

```
$ php artisan migrate
Migration table created successfully.
Migrated: 2013_07_21_103250_crear_albums
```

Nuestra base de datos ahora tiene una estructura para contener datos de nuestros álbumes. Ahora necesitamos crear una definición de modelo de Eloquent para que podamos interactuar con nuestras filas como si fueran objetos PHP.

```
<?php

// app/models/Album.php

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

Adorable, simple, limpio. Hemos desactivado timestamps en nuestra definición de modelo `Album` para simplificar los ejemplos de esta sección. Normalmente las añado a todos mis modelos. Aunque puede afectar un poco al rendimiento y necesita un poco más de espacio, suelen resultar muy útiles a la hora de auditar un modelo.

Ahora tenemos todo lo que necesitamos para rellenar nuestra tabla de la base de datos con datos de álbumes. Como mencioné antes, usaríamos otro método para esta tarea, pero por ahora simplemente crearemos una Closure enrutada. Como resultado, esto resultará un poco repetitivo, pero por ahora vamos a pasar de eso. Solo vamos a visitar la ruta una sola vez.

```
<?php

// app/routes.php

Route::get('/seed', function()
{
    $album = new Album;
    $album->titulo          = 'Some Mad Hope';
    $album->artist         = 'Matt Nathanson';
    $album->genero          = 'Acoustic Rock';
    $album->ano            = 2007;
    $album->save();

```

```
14      $album = new Album;
15      $album->titulo          = 'Please';
16      $album->artist         = 'Matt Nathanson';
17      $album->genero          = 'Acoustic Rock';
18      $album->ano            = 1993;
19      $album->save();
20
21      $album = new Album;
22      $album->titulo          = 'Leaving Through The Window';
23      $album->artist         = 'Something Corporate';
24      $album->genero          = 'Piano Rock';
25      $album->ano            = 2002;
26      $album->save();
27
28      $album = new Album;
29      $album->titulo          = 'North';
30      $album->artist         = 'Something Corporate';
31      $album->genero          = 'Piano Rock';
32      $album->ano            = 2002;
33      $album->save();
34
35      $album = new Album;
36      $album->titulo          = '...Anywhere But Here';
37      $album->artist         = 'The Ataris';
38      $album->genero          = 'Punk Rock';
39      $album->ano            = 1997;
40      $album->save();
41
42      $album = new Album;
43      $album->titulo          = '...Is A Real Boy';
44      $album->artist         = 'Say Anything';
45      $album->genero          = 'Indie Rock';
46      $album->ano            = 2006;
47      $album->save();
48  });
```

Esos son algunos de mis álbumes favoritos. Espero que los que no seáis fans del punk rock no os sintáis ofendidos por mis gustos musicales y continuéis con este capítulo.

Como puedes ver, cada una de las filas la hemos creado con una nueva instancia del modelo `Album`, rellenando todos los campos y guardándolos en la base de datos.

Ve y visita la URI `/seed` para rellenar la tabla `albums` con algunos de nuestros datos de ejemplo. Deberías recibir una página en blanco porque no devolvimos ninguna respuesta desde la ruta.

Ahora que ya tenemos los datos de ejemplos en la base de datos, puedes borrar la ruta `/seed`. No la necesitamos más. Nuestra preparación está completa, comencemos aprendiendo sobre las consultas de Eloquent.

Eloquent a cadena

Los objetos de PHP pueden contener un método opcional llamado `__toString()`. Puede que ya lo conozcas, fue añadido en PHP 5.2 junto a algunos otros métodos mágicos. Este método puede ser usado para controlar cómo debería ser representado el objeto como una cadena.

Gracias a este método, nuestros modelos Eloquent pueden ser expresados también como cadenas. Como ves, la clase de Eloquent que extendemos en nuestros modelos, contiene un método `__toString()`. Este método devuelve una cadena JSON que representa el valor de nuestro modelo Eloquent.

Puede sonar un poco confuso y hace mucho que no ponemos un ejemplo. Primero veamos la forma normal de exponer los valores contenidos en una instancia de un modelo de Eloquent.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $album = Album::find(1);
    return $album->titulo;
});
```

En el ejemplo de arriba, usamos el método `find()` heredado, de nuestro modelo `Album` y pasamos un valor entero de `1` para obtener una instancia que represente la fila que tenga una `id` con valor `1`. Luego devolvemos el atributo `titulo` de la instancia del modelo para ser representada como respuesta de la vista.

Si visitamos `/`, recibimos la siguiente respuesta.

Some Mad Hope

El título del primer álbum que insertamos en nuestra base de datos, como esperábamos. Ahora modifiquemos la ruta para devolver la instancia del modelo como respuesta de la closure.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::find(1);
});
```

Visitemos la URI `/` para examinar la respuesta

¡Eso me suena a JSON! Todas las cadenas JSON creadas por el Laravel eliminan los espacios en blanco y la indentación para ahorrar ancho de banda a la hora de transferir datos. Voy a poner bonitos todos los ejemplos JSON de este capítulo, así que no te sorprendas si las respuestas se ven más raras que las mostradas en este capítulo.

Vamos a ver.

```
{
    id: 1,
    titulo: "Some Mad Hope",
    artista: "Matt Nathanson",
    genero: "Acoustic Rock",
    ano: 2007
}
```

Laravel ha ejecutado el método `__toString()` de la instancia de nuestro modelo Eloquent para representar su valor como cadena JSON. Es realmente útil a la hora de crear APIs REST que sirven JSON. Es también una buena forma de expresar la salida de tus consultas para el resto del capítulo.

Algunos métodos de Eloquent devolverán varias instancias de un modelo como resultado, en vez de un único modelo. Veamos rápidamente el resultado del método `all()` que obtiene todas las filas como instancias de modelos Eloquent.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $albums = Album::all();
    foreach ($albums as $album) {
        echo $album->titulo;
    }
});
```

Usamos el método `all()` de nuestro modelo `Album` para obtener una matriz de instancias de modelos Eloquent que representan las filas de nuestra tabla `albums`. Luego iteramos sobre la matriz y mostramos el título de cada una de las instancias de `Album`.

Este es el resultamos que recibimos al visitar la URI `/`.

```
Some Mad HopePleaseLeaving Through The WindowNorth...Anywhere But Here...Is A Rea\
l Boy
```

Genial, esos son los títulos de los álbumes. Están todos juntos porque no hemos insertado ningún separador. No te preocupes, al menos los obtenemos todos.

Siento esto, pero, una vez más te he mentido. Si has usado Laravel 3 anteriormente, el concepto de método de obtención de modelos te será familiar. No obstante, Laravel 4 no devuelve una matriz de esos métodos, tal cual, devuelve una `Collection`.

> No te creo, si no devuelve una matriz ¿cómo hemos podido iterar por los resultados?

Eso es simple. El objeto `Collection` implementa una interfaz que permite que el objeto sea iterable. Puede ser objeto de un bucle usando las mismas funcionalidades que las matrices de PHP estándar.

> Humm, ya veo. No voy a quitarte la etiqueta de mentiroso tan fácilmente aun así.

Ah, ya veo, ¿necesitas más pruebas? Vamos a mostrar el valor de la variable `$albums` para ver con lo que estamos trabajando. Esto debería bastar.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $albums = Album::all();
    var_dump($albums);
});
```

Cuando visitemos la URI `/` recibimos la siguiente respuesta.

```
object(Illuminate\Database\Eloquent\Collection)[134]
  protected 'items' =>
    array (size=6)
      0 =>
        object(Album)[127]
          public 'timestamps' => boolean false
          protected 'connection' => null
          protected 'table' => null
          protected 'primaryKey' => stri

        ... loads more information ...
```

> Vaya, no estabas mintiendo esta vez

Como verás, el resultado de cualquier método que devuelva varias instancias de un modelo es representado por una instancia de `Illuminate\Database\Eloquent\Collection`. Puedes ver por la salida de `var_dump` que este objeto contiene una matriz interna de las instancias de nuestros modelos llamada `items`.

La ventaja de este objeto es que también contiene varios métodos útiles para transformar y obtener instancias de nuestros modelos. En un capítulo posterior los examinaremos en más detalle, pero vale la pena saber que el objeto `Collection` también tiene un método `__toString()`. Este método funciona de manera similar al de nuestras instancias de modelos, pero en vez de eso crea una cadena JSON que representa nuestras instancias de modelos como matrices multidimensionales.

Vamos a devolver el objeto resultado del método `all()` como respuesta de nuestra closure enrutada. Así:

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::all();
});
```

La respuesta que recibimos tras visitar la URI `/` es la siguiente.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    },
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
```

```
43      }
44  ]
```

Recibimos una cadena JSON que contiene una matriz de objetos que representan los valores de los álbumes de forma individual.

Entonces, ¿por qué estamos aprendiendo sobre la funcionalidad `__toString()` ahora? ¿No pretendemos construir un API JSON este capítulo no? No, no estamos preparados para eso aun.

Puedo usar la salida JSON para mostrar los resultados de las consultas que se estarán ejecutando a lo largo del capítulo. Sería más legible que un montón de bucles `foreach()` como resultados. Ahora sabes porqué exactamente se muestran como JSON. ¡Todo el mundo gana!

Ahora que tenemos la base de datos rellena con datos, y hemos identificado una forma de consultar resultados, echemos un vistazo a la estructura de las consultas de Eloquent.

Estructura de las consultas

Las consultas de Eloquent son usadas para obtener resultados basándonos en varias reglas o criterios. No siempre quieres obtener todas las filas de álbumes. A veces solo quieres obtener la discografía de un único artista. En esas circunstancias, usaríamos una consulta, para pedir únicamente las filas que tengan una columna `artista` con el artista que queremos.

Las consultas pueden ser separadas en tres partes.

- El modelo.
- Limitaciones de las consultas
- Métodos de obtención

El modelo es la instancia del modelo sobre la que queremos que la consulta actúe. Todos los ejemplos de esta sección formarán consultas basándose en el modelo `Album`.

Las limitaciones de las consultas son reglas que son usadas para obtener un subconjunto de filas de la tabla. De esta forma podemos devolver únicamente las filas en las que estamos interesados. La más familiar en SQL es la cláusula `WHERE`.

Finalmente, tenemos los métodos de obtención. Esos son los métodos que son usados para formar las consultas, y devolver los resultados.

Echemos un vistazo a la estructura de una consulta Eloquent en su forma más simple.

```php
<?php

Model::fetch();
```

Todas as consultas actúan sobre uno de nuestros modelos Eloquent. Los métodos de limitación son completamente opcionales, en el ejemplo anterior, no están presentes. Luego tenemos el método de obtención. Este método no existe, solamente lo estamos usando para mostrar la forma de una consulta. El primer método de una cadena de una consulta es siempre llamado de forma estática, con dos puntos dobles `::`.

Las consultas de Eloquent pueden no tener filtros, un único filtro o varios. Es totalmente cosa tuya. He aquí como se vería una consulta con un único filtro.

```php
<?php

Model::constraint()
    ->fetch();
```

Observa cómo el filtro es ahora el método estático, y el método de obtención ha sido encadenado al final del primer método. Podemos añadir tantas constantes a la consulta como queramos, por ejemplo:

```php
<?php

Model::constraint()
        ->constraint()
        ->constraint()
    ->fetch();
```

Los filtros son totalmente opcionales, pero todas las consultas deben comenzar con un modelo y finalizar con un método de obtención. He aquí un ejemplo usando nuestro modelo `Album`.

```php
<?php

Album::all();
```

`Album` es nuestro modelo, y `all()` es uno de nuestros métodos de obtención, porque es usado para obtener los resultados de nuestra consulta.

Los métodos de obtención pueden ser usados para devolver un único modelo o una instancia de una colección de modelos. No obstante, como hemos descubierto antes, ambos pueden ser expresados en formato JSON como respuesta de una closure enrutada o acción de controlador.

Sabemos que los filtros de las consultas son opcionales, así que empecemos mirando la variedad de métodos de obtención que están disponibles.

Métodos de obtención

Comencemos con algunos de los métodos de obtención que puedes haber visto en algún capítulo anterior. Primero tenemos el método `find()`.

Find

El método `find()` puede ser usado para obtener un único modelo Eloquent por el valor `id` de su fila. El primer parámetro es un entero y se devolverá una única instancia. He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::find(1);
});
```

Queremos obtener una fila de la base de datos con una `id` de `1`, por lo que una única instancia de un modelo es devuelta.

```
{
    id: 1,
    titulo: "Some Mad Hope",
    artista: "Matt Nathanson",
    genero: "Acoustic Rock",
    ano: 2007
}
```

Si usáramos una matriz de valores `id`, obtenemos una `Collection` de instancias de modelos.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::find(array(1, 3));
});
```

He aquí el resultado. Una colección que contenga instancias de modelos con `ids` con valor `1` y `3`.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    }
]
```

All

El método `all()` puede ser usado para obtener una colección de instancias de modelos que representan todas las filas contenidas en la tabla. He aquí un ejemplo del metodo `all()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::all();
});
```

Recibimos una colección conteniendo instancias de todos los álbumes contenidos en nuestra base de datos.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    },
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
```

```
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

First

En circunstancias en las que obtenemos una colección de modelos, el método `first()` puede ser usado para obtener el *primer* objeto. Es muy útil si solo quieres un objeto en vez de una colección de ellos. Sin un filtro, `first()` simplemente devolverá la primera fila de la tabla de la base de datos.

He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::first();
});
```

Recibimos una única instancia de un modelo, el primer álbum almacenado en la tabla de la base de datos.

```
{
    id: 1,
    titulo: "Some Mad Hope",
    artista: "Matt Nathanson",
    genero: "Acoustic Rock",
    ano: 2007
}
```

Update

No tenemos solo que devolver una instancia de un modelo, podemos actualizarla también. Usando el método `update()` podemos actualizar los valores de las filas de la tabla que son resultado de una consulta Eloquent. Simplemente pasa una matriz clave-valor como primer parámetro al método `update()` para cambiar los valores de la columna para cada fila. La matriz representa el nombre de la columna a cambiar y el valor representa el nuevo valor para la columna.

El método `update()` es especial, y no puede ser usado sin filtro, por tanto usaré una filtro `where` sencillo en el ejemplo. Si no lo entiendes, no te preocupes demasiado. Cubriremos los filtros con más detalle en la siguiente sección. He aquí un ejemplo que modificará la tabla `albums`. (No te preocupes, la restauraré para el siguiente ejemplo.)

```
<?php

// app/routes.php

Route::get('/', function()
{
        Album::where('artista', '=', 'Matt Nathanson')
            ->update(array('artista' => 'Dayle Rees'));

        return Album::all();
});
```

Actualizamos el campo `artista` de todas las filas cuyo artista sea `Matt Nathanson`, cambiando su valor a `Dayle Rees`. El método `update()` no obtiene instancias de un modelo por lo que devuelve en vez de eso una colección de todos los modelos usando `all()`.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Dayle Rees",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Dayle Rees",
        genero: "Acoustic Rock",
        ano: 1993
    },
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
```

```
    }
]
```

Como puedes ver, ahora soy una estrella del rock. ¡Genial!

Delete

Al igual que el método `update()`, el método `delete()` no devolverá ninguna instancia. En vez de ello, eliminará las filas que son el resultado de la consulta de la tabla.

```
<?php

// app/routes.php

Route::get('/', function()
{
        Album::where('artista', '=', 'Matt Nathanson')
            ->delete();

        return Album::all();
});
```

Consultamos todos los álbumes que tienen una columna `artista` con valor de `Matt Nathanson`. Luego usamos el método `delete()` para borrar las filas de la base de datos.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
```

```
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

Los álbumes de Matt Nathanson han sido eliminados de nuestra base de datos. Es una auténtica vergüenza porque hace muy buena música.

He aquí un truco rápido. Si quieres borrar todas las filas de un modelo en concreto, puede que encuentres el método `truncate()` más descriptivo.

He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
        Album::truncate();
        return Album::all();
});
```

Como puedes ver, todas las filas se han esfumado.

```
[ ]
```

Get

`get` es el método de obtención más importante. Es usado para obtener el resultado de la consulta. Por ejemplo, si usamos un filtro con `where()` para limitar un conjunto de resultados para obtener un único artista, no tendría sentido usar el método `all()`. En ve de ello, usamos el método `get()` para obtener una colección de instancias de modelos.

¿Confuso? He aquí el método `get()` en combinación con el filtro `where()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Something Corporate')
        ->get();
});
```

Recibimos una colección de instancias de modelos que tienen como valor de la columna `Artist`, el artista `Something Corporate`.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    }
]
```

El método `get()` tiene un parámetro opcional. Puedes pasar una matriz de nombres de columnas y los objetos resultantes solo contendrán valores para esas columnas. He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Something Corporate')
        ->get(array('id', 'titulo'));
});
```

Pasamos una matriz con los valores `id` y `titulo` al método `get()`, he aquí el conjunto resultante que obtenemos.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window"
    },
    {
        id: 4,
        titulo: "North"
    }
]
```

Como puedes ver, solo las columnas que hemos solicitado están presentes en los resultados.

Pluck

El método `pluck()` puede ser usado para obtener un valor de una única columna. He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::pluck('artista');
});
```

El primer y único parámetro, es el nombre de la columna de la que queremos obtener el valor. Si la consulta obtiene varios resultados, solo el valor del primer resultado será devuelto. He aquí el resultado que obtenemos del ejemplo anterior.

```
Matt Nathanson
```

Lists

Aunque el método `pluck()` puede obtener un único valor para una columna en concreto, el método `lists()` obtendrá una matriz de valores para la columna especificada en todas las instancias del resultado. Clarifiquémoslo con un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::lists('artista');
});
```

Una vez más, el método `lists()` acepta un único parámetro. El nombre de la columna de la que queremos obtener todos los valores. He aquí el resultado.

```
[
    "Matt Nathanson",
    "Matt Nathanson",
    "Something Corporate",
    "Something Corporate",
    "The Ataris",
    "Say Anything"
]
```

Como puedes ver, hemos obtenido los valores contenidos en la columna `artista` para todas las filas de nuestra tabla.

ToSql

Bien, este no es un método de obtención, ¡pero es muy útil! Puedes usar el método `toSql()` en cualquier lugar en el que normalmente fueras a usar un método de obtención, normalmente al final de una cadena de consulta, y devolverá el SQL que representa la consulta como una cadena.

Echemos un vistazo a un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Something Corporate')
        ->toSql();
});
```

Similar al ejemplo anterior, pero esta vez llamamos al métoo `toSql()` en vez de a `get()`. Este es el resultado que recibimos.

```
select * from `albums` where `artista` = ?
```

¡Muy útil para depurar!

¿Para qué es el símbolo de interrogación?

El constructor de consultas usa sentencias preparadas. Esto significa que la marca de interrogación será reemplazada por valores actuales. El beneficio de esto es que los valores reemplazados serán escapados antes de ser colocados en la cadena, para evitar intentos de inyección SQL.

Ahora que hemos descubierto los métodos de obtención, es hora de aprender sobre cómo añadir filtros a nuestras consultas.

Limitaciones de las consultas

Los métodos de obtención vistos anteriormente, son útiles para obtener colecciones de modelos e instancias de nuestra base de datos. No obstante, a veces necesitamos afinar el resultado para que las filas sean solo unas pocas. Es aquí cuando los filtros de las consultas entran en juego.

En matemáticas, la aritmética nos permite capturar un subconjunto de un conjunto mucho más grande de valores. Esto es esencialmente lo que intentamos conseguir aquí usando filtros de la consulta. No obstante, también he colado algunos métodos de transformación por aquí para cambiar la ordenación de los resultados.

Comencemos con un método que representa el filtro SQL más común, la clausula `WHERE`.

Where

El primer filtro que vamos a examinar es el método `where()`. Si has usado SQL en el pasado, seguramente ya hayas usado la cláusula `WHERE` para obtener filas de una tabla, filtrando por el valor de sus columnas.

Vamosa a ver un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Matt Nathanson')
        ->get();
});
```

Usamos el método `where()` para limitar los resultados de los álbumes a aquellos que tengan una columna `artista` con valor `Matt Nathanson` únicamente.

El método `where()` aceptará tres parámetros. El primero es el nombre de la columna sobre la que queremos filtrar. En este ejemplo la comparación es contra la columna `artista`. El segundo parámetro es el operador usado en la comparación En nuestro ejemplo queremos asegurarnos de que `artista` **sea igual** a un valor, por lo que usamos el símbolo `=`.

Podríamos haber usado cualquier otro símbolo común de comparación soportado por SQL, como `<`, `>`, `=>`, `=<`, etc. Experimenta con los tipos de operadores para obtener el resultado que necesitas.

El tercer parámetro es el valor con el que comparamos. En nuestro ejemplo, nos queremos asegurar de que la columna `artista` coincide con el nombre `Matt Nathanson`, por lo que, en este caso, `Matt Nathanson` es el valor.

Una vez más, el método `where()` es únicamente un filtro. Usaremos el método `get()` para obtener una colección de resultados. Echemos un vistazo a la respuesta desde la URI `/`.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    }
]
```

Excelente, los álbumes que tienen el valor `Matt Nathanson` en la columna `artista` nos han sido devueltos. Esto nos sería útil si pensáramos ofrecer secciones en una web de música para mostrar discografías de un artista en concreto.

Merece la pena recordar que los métodos `get()` y `first()` son intercambiables. Alteremos el ejemplo para obtener únicamente la primera instancia que coincida con la condición.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Matt Nathanson')
        ->first();
});
```

Ahora la consulta solo devolverá una única instancia de un modelo, representando la fila que cumple con los requisitos. Este es el resultado de visitar la URI `/`.

```
{
    id: 1,
    titulo: "Some Mad Hope",
    artista: "Matt Nathanson",
    genero: "Acoustic Rock",
    ano: 2007
}
```

Intentemos otro operador con el método `where()`, ¿Qué pasa con `LIKE`? El operador `LIKE` de SQL, puede ser usado para comparar partes de una cadena usando un símbolo de porcentaje `%` como comodín.

He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('titulo', 'LIKE', '...%')
        ->get();
});
```

En el ejemplo de arriba, queremos obtener todas las filas cuyo campo `titulo` comience con tres puntos `...`. El símbolo del porcentaje `%` hará saber a la base de datos que no nos importa lo que venga detrás de los tres puntos. Como nota, sé que tres puntos son puntos suspensivos, solo pensé que sería más fácil para los lectores.

Echemos un vistazo al resultado de la URI `/`.

```
[
    {
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

Recibimos una colección de resultados para los álbumes con títulos '...Anywhere But Here' y '...Is A Real Boy', ambos comienzan con tres puntos, y son álbumes fabulosos.

No estamos limitados a un solo método `where()` en una consulta. Podemos encadenar varios métodos `where()` juntos para obtener filas basadas en varios filtros. He aquí eun ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('titulo', 'LIKE', '...%')
        ->where('artista', '=', 'Say Anything')
        ->get();
});
```

En el ejemplo anterior, queremos encontrar filas que tengan una columna `titulo` que comience con tres puntos **y** una columna `artista` que sea igual a 'Say Anything'. El **y** es importante. Ambos filtros deben ser cumplidos para que la fila sea devuelta.

He aquí el resultado.

```
[
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

Como ves, obtenemos una colección que contiene una única instancia, un álbum con un título que comienza con tres puntos y cuyo artista es 'Say Anything'. Recibimos una colección que contiene un disco. ¡Uno de mis favoritos!

OrWhere

No siempre necesitamos que dos filtros se cumplan. A veces, que coincida una de las dos condiciones es suficiente para nosotros. En situaciones como esas, podemos usar el método `orWhere()`. De hecho, muchos de los filtros de este capítulo, tendrán una alternativa con el prefijo `or` que permitirán que varios filtros coincidas. Por ese motivo, no voy a usar secciones separadas pora las variaciones `or` en el futuro.

Como siempre, he aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('titulo', 'LIKE', '...%')
        ->orWhere('artista', '=', 'Something Corporate')
        ->get();
});
```

Usamos un filtro inicial, indicando que el título del álbum **tiene que** comenzar con tres puntos (sí, sé que son puntos suspensivos). Luego incluimos el método `orWhere()` que indica que nuestro resultado también puede consistir en aquellos cuya columna `artista` tenga un valor de `Something Corporate`.

Echa un vistazo al resultado.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
```

```
27          genero: "Indie Rock",
28          ano: 2006
29      }
30  ]
```

Recibimos una colección de instancias que tienen un `titulo` que empieza por tres puntos **o** cuya columna `artista` tenga el valor de `Something Corporate`.

Puedes encadenar tantos `where()` y `orWhere()` como necesites para filtrar las filas de tu tabla hasta obtener los resultados necesarios.

WhereRaw

El método `whereRaw()` puede ser usado para usar una cadena SQL para ejecutar una condición `WHERE`. He aquí un ejemplo.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      return Album::whereRaw('artist = ? and titulo LIKE ?', array(
8          'Say Anything', '...%'
9      ))
10     ->get();
11 });
```

El método `whereRaw()` acepta una cadena SQL como primer parámetro. Todos los signos de interrogación de la cadena son reemplazados la matriz de elementos del segundo parámetro en orden secuencial. Si ya has usado este tipo de métodos, puede ser que esto ya te resulte familiar. Los valores serán escapados para evitar ataques de inyección SQL.

Una vez que el constructor de consultas ha llevado a cabo las transformaciones necesarias, el SQL resultante será algo así:

```
1  artist = 'Say Anything' and titulo LIKE '...%'
```

El resultado de nuestra consulta es el siguiente:

```
[
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

Puedes usar el método `whereRaw()` en circunstancias en las que necesites SQL complejo además de tu limitación de tipo `where()`. Al igual que el método `where()`, el método `whereRaw()` puede ser encadenado varias veces con otros métodos para limitar más tu conjunto. Una vez más, hay un método `orWhereRaw()` para permitir condiciones alternativas.

WhereBetween

El método `whereBetween()` es usado para revisar que el valor de la columna está entre dos valores proveídos. Es mejor describirlo con un ejemplo, de hecho, creo que todo es mejor explicarlo con un ejemplo. Extraño, ¿verdad? Quizá es porque el código de Laravel tiende a hablar por sí mismo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereBetween('ano', array('2000', '2010'))
    ->get();
});
```

El primer parámetro es el nombre de la columna que queremos comparar. El segundo parámetro es una matriz de dos valores, el valor de inicio y el límite. En el ejemplo anterior, estamos buscando álbumes que tengan un año entre 2000 y 2010. Aquí están los resultados.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

El resultado es el esperado, álbumes del año 2000.

AL igual que con otros métodos del tipo `where()`, puedes encadenar como quieras y como siempre, tenemos una alternativa `orWhereBetween()`.

WhereNested

El método `whereNested()` es una forma limpia de aplicar varios filtros `where` en una consulta. Simplemente pasa una closure como primer parámetro al método, y dale a la Closure un parámetro de relleno nombrado como quieras. Me gusta llamarlo `$query`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereNested(function($query)
    {
        $query->where('ano', '>', 2000);
        $query->where('ano', '<', 2005);
    })
    ->get();
});
```

En la Closure, puedes aplicar tantos `where()` como `orWhere()` como quieras al objeto `$query`, lo cuál luego se convertirá en parte de su consulta principal. ¡Tan solo se ve mucho más limpio! Este es el resultado del ejemplo anterior.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    }
]
```

En este caso no hay alternativa `orWhereNested()` a este método, pero he aquí un secreto... también puedes pasar una closure al método `orWhere()`. He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereNested(function($query)
    {
        $query->where('ano', '>', 2000);
        $query->where('ano', '<', 2005);
    })
    ->orWhere(function($query)
    {
        $query->where('ano', '=', 1997);
    })
    ->get();
});
```

Queremos un álbum que haya sido lanzado entre 2000 y 2005, **o** cuyo añao de lanzamiento sea 1997. Este es el SQL generado del método anterior.

```
select * from `albums` where (`ano` > ? and `ano` < ?) or (`ano` = ?)
```

Estos son los resultados de la consulta anterior.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 5,
```

```
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    }
]
```

WhereIn

El método `whereIn()` puede ser usado para revisar que una columna existe en un conjunto de valores. Es realmente útil cuando ya tienes una matriz de posibles valores a mano. Echemos un vistazo a cómo podemos usarlo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $values = array('Something Corporate', 'The Ataris');
    return Album::whereIn('artista', $values)->get();
});
```

El primer parámetro es que el método `whereIn()` es la columna sobre la que comparar. El segundo valor es la matriz de valores a buscar.

El SQL resultante de la consulta anterior se ve así.

```
select * from `albums` where `artista` in (?, ?)
```

Esta es la colección de resultados que recibimos de la consulta.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
```

```
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    }
]
```

El método `whereIn()` también tiene el método alternativo `orWhereIn()` y puede ser encadenado varias veces.

WhereNotIn

El método `whereNotIn()` es el opuesto directo al método `whereIn()`. Esta vez tenemos una lista de valores, y el valor de la columna no debe existir en el conjunto.

Echemos un vistazo al ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $values = array('Something Corporate', 'The Ataris');
    return Album::whereNotIn('artista', $values)->get();
});
```

Una vez más, pasamos la columna a comparar como primer parámetro, y nuestra matriz de valores como segundo parámetro.

Esta es la consulta resultante.

```
select * from `albums` where `artista` not in (?, ?)
```

Finalmente, este es el resultado de nuestra consulta. Todos los álbumes que no han sido identificados por los artistas por los valores de nuestra matriz.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

Una vez más, hay una opción `orWhereNotIn()` alternativa.

WhereNull

El filtro `whereNull()` puede ser usado para obtener filas que tengan un valor de columna de `NULL`. Veamos un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereNull('artista')->get();
});
```

El único parámetro para el método `whereNull()` es el nombre de la columna que quieres que contenga un valor `NULL`. Echemos un vistazo a la consulta generada.

```
select * from `albums` where `artista` is null
```

Ahora echemos un vistazo al resultado de la consulta.

```
[ ]
```

Ohh, está bien, no tenemos ningún valor `NULL` en la base de datos. No quiero reescribir este capítulo otra vez, así que tendrás que usar tu imaginación. Si tuviéramos una columna `artista` con un valor de `NULL`, la fila aparecería en el conjunto.

Sí, ¡lo adivinaste! Hay un método `orWhereNull()` también disponible.

WhereNotNull

El método `whereNotNull()` es el opuesto del método `whereNull()`, así que esta vez podremos ver algunos resultados. Devolverá filas que tienen valor que no sean iguales a `NULL`. Echemos un vistazo más de cerca.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereNotNull('artista')->get();
});
```

El primer y único parámetro es el nombre de la coluna. Esta es la consulta SQL generada.

```
select * from `albums` where `artista` is not null
```

He aquí el conjunto resultante de la consulta.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    },
    ... 4 more ...
]
```

Todos los álbumes de la base de datos, ya que la columna `artista` no tiene ningún valor `NULL`.

Una vez más, tenemos disponible el método `orWhereNotNull()` está disponible para realizar filtros del tipo **or**.

OrderBy

El método `orderBy()` puede ser usado para ordenar los resultados devueltos por tu consulta por el valor de una columna específica. Vamos a ver un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Matt Nathanson')
        ->orderBy('ano')
        ->get();
});
```

El primer parámetro al método `orderBy()` es el nombre de la columna por la que queremos ordenar. Por defecto, la ordenación será ascendiente.

Este es el SQL generado.

```
select * from `albums` where `artista` = ? order by `ano` asc
```

He aquí el conjunto de resultados de la consulta.

```
[
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    },
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    }
]
```

Genial, nuestros álbumes han sido devuelto en orden ascendentes por año de lanzamiento. ¿Qué pasa si lo queremos en orden descendente? No te preocupes, ¡Laravel te tiene cubierto!

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Matt Nathanson')
        ->orderBy('ano', 'desc')
        ->get();
});
```

Añadimos un segundo parámetro al método `orderBy()` con valor `desc`. Esto le dice a Laravel que queremos obtener los resultados en orden descendente. Este es el SQL generado.d

```
select * from `albums` where `artista` = ? order by `ano` desc
```

Este es el conjunto de resultados devueltos.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    }
]
```

¡Genial! Nuestros resultados están ahora en orden descendiente.

Podemos usar la cláusula `orderBy()` con cualquier combinación de filtros en este capítulo. También puedes usar `orderBy()` adicionales para usar más ordenación, en el orden en que los métodos son facilitados.

Take

El método `take()` puede ser usado para limitar el conjunto de resultados. He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::take(2)
        ->get();
});
```

El primer parámetro del método `take()` es la cantidad de filas que quieres obtener. En el ejemplo anterior, solo queremos que la consulta devuelva dos objetos.

He aquí el SQL generado por la consulta.

```
select * from `albums` limit 2
```

Finalmente, he aquí el conjunto de resultado que recibimos.

```
[
    {
        id: 1,
        titulo: "Some Mad Hope",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 2007
    },
    {
        id: 2,
        titulo: "Please",
        artista: "Matt Nathanson",
        genero: "Acoustic Rock",
        ano: 1993
    }
]
```

El método puede ser usado en combinación con cualquier otro filtro.

Skip

Al usar el método `take()`, el método `skip()` puede ser usado para saltarnos algunos resultados. He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::take(2)
        ->skip(2)
        ->get();
});
```

El método `skip()` acepta un único parámetro para saltarnos algunos resultados. En el resultado de arriba, las dos primeras filas serán descartadas del conjunto de resultados. Esta es la consulta generada.

```
select * from `albums` limit 2 offset 2
```

Este es el resultado que recibimos.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    }
]
```

Como puedes ver, nos hemos saltado la primera y segunda fila, comenzando con la tercera fila de la base de datos.

Consultas where mágicas

Bien, ¡es hora de la magia! Ahora, ya debes estar más que familiarizado con el método `where()`. Es responsable de restringir los resultados filtrando sobre los valores de las columnas. He aquí un ejemplo para recordarte:

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('artista', '=', 'Something Corporate')
        ->get();
});
```

Pretendemos obtener un conjunto de resultados en los que la columna `artista` sea igual a `Something Corporate`. Es una forma bonita y limpia de restringir resultados. ¿Podría ser más limpio? Bueno, ¡podría! Podemos usar una sintaxis mágica.

Echa un vistazo al siguiente ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereArtist('Something Corporate')->get();
});
```

Espera, ¿cuál es el método `whereArtist()`? No lo hemos aprendido hasta ahora. Bueno, este método es un poco especial. Primero visitemos la URI `/` para ver el resultado obtenido.

```
[
    {
        id: 3,
        titulo: "Leaving Through The Window",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    },
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    }
]
```

Parece que funciona de manera similar al método `where()` con un operador =. Bien, es momento de explicar lo qué está pasando. El método `where()` con igual es uno de los más comunes y por ello, Taylor ofrece un atajo conveniente.

Simplemente puedes añadir el nombre de la columna al método `where()`. Primero, debes capitalizar la primera letra del campo con el que quieres comparar. En nuestro ejemplo, usamos la columna `artista` por loq ue el nombre del método será `whereArtist()`. Si el nombre del campo está separado

por guiones bajos, por ejemplo, `shoe_size`, debemos capitalizar la primera letra de cada palabra `whereShoeSize()`.

El único parámetro al método mágico `where()` es el valor esperado en la columna. Veamos otro ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::whereTitulo('North')->get();
});
```

Obtenemos todos los álbumes, con una columna `titulo` cuyo valor sea `North`. Este es el resultado de la consulta.

```
[
    {
        id: 4,
        titulo: "North",
        artista: "Something Corporate",
        genero: "Piano Rock",
        ano: 2002
    }
]
```

Maravilloso, ¡ahí está nuestro álbum! Asegúrate de recordar el filtro `where()` mágico si estás obteniendo valores por valores de columnas.

Alcance de la consulta

El alcance de la consulta puede ser muy útil si te descubres repitiendo las mismas consultas una y otra vez. Comencemos con un ejemplo. Recuerda aquellos dos álbumes que comenzaban con tres puntos '...Is A Real Boy' y '...Anywhere But Here'. Imaginemos que obtener álbumes que comienzan con tres puntos es una acción común de nuestra aplicación.

Veamos, podríamos solicitar los álbumes cada vez, así.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::where('titulo', 'LIKE', '...%')->get();
});
```

Eso sería horriblemente repetitivo, ¿no? No queremos repetirnos. ¿Por qué no usamos una predefinida? Comencemos. Primero revisitemos el modelo `Album`. Ahora se ve así.

```php
<?php

// app/models/Album.php

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

Vamos a añadir un nuevo método al modelo. Nuestro nuevo modelo es así.

```php
<?php

// app/models/Album.php

class Album extends Eloquent
{
    public $timestamps = false;

    public function scopeTresPuntos($query)
    {
        return $query->where('titulo', 'LIKE', '...%');
    }
}
```

Hemos añadido el método `scopeTresPuntos()` a nuestro modelo. Es un método especial con una función específica, nos ayudará a reusar consultas comunes. Todos los métodos empiezan con la palabra `scope` y luego un identificador. El método aceptará un único parámetro, un objeto `$query`. Este objeto puede ser usado para construir consultas como las que hemos descubierto en las secciones

anteriores. En nuestro ejemplo, usamos una sentencia `return` para devolver el valor de un método `where()`. El método `where()` toma la misma forma que en nuestros ejemplos anteriores.

Ahora volvamos a nuestro archivo de rutas. Alteremos nuestra consulta existente. Esta es la nueva Closure enrutada.

```
<?php

// app/routes.php

Route::get('/', function()
{
    return Album::tresPuntos()->get();
});
```

Cambiamos nuestra consulta `where()` para en vez de eso, llamar al método `tresPuntos()`. Luego simplemente usamos `get()` sobre el resultado para obtener los resultados. Nota que la parte `scope` del nombre del método, no está incluído. Asegúrate de dejarlo fuera de las llamadas de los métodos. Echemos un vistazo al resultado.

```
[
    {
        id: 5,
        titulo: "...Anywhere But Here",
        artista: "The Ataris",
        genero: "Punk Rock",
        ano: 1997
    },
    {
        id: 6,
        titulo: "...Is A Real Boy",
        artista: "Say Anything",
        genero: "Indie Rock",
        ano: 2006
    }
]
```

Genial, ese es el resultado que esperábamos. Úsalo tantas veces como quieras para evitar repeticiones.

Colecciones de Eloquent

Me encantan las colecciones. Tengo demasiadas. De niño me gustaba coleccionar juguetes de los transformers y tazas de huevos. Como adulto, colecciono videojuegos y comics manga. Las colecciones frikis son las mejores.

Laravel también tiene sus propias colecciones. Tiene una colección de maravillosos fans, ansiosos de ayudarse unos a los otros y desarrollar la comunidad. Tiene una colección de increíbles programadores que contribuyen a él. Tiene una colección de historias sobre de dónde viene el nombre, la mayoría de ellas falsas. También tiene colecciones de Eloquent.

La clase Collection

Las colecciones de Eloquent, son una extensión de la clase `Collection` de Laravel con algunos métodos útiles para gestionar resultados de las consultas. La clase `Collection` en sí, es nada más que un envoltorio para una matriz de objetos, pero tiene otors métodos interesantes para ayudarte a extraer elementos de la matriz.

En Laravel 3, una matriz de instancias de un modelo era lo que obtenías de cualquier método de consultas que fuera usado para obtener varios resultados. No te preocupes, aun puedes iterar sobre una colección de resultados con los bucles que ofrece PHP porque hereda algunas propiedades de una matriz, pero dado que la colección es una clase y no un tipo nativo, hay algunos métodos disponibles en el objeto.

Echemos un vistazo a los métodos disponibles en la clase `Collection`. Para nuestros ejemplos, usaremos la tabla `albums` del capítulo anterior, y asumamos que la colección es el resultado de una llamada a `Album::all()`, así.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      $coleccion = Album::all();
8  });
```

Métodos de colección

Echemos un vistazo a los métodos disponibles en la clase `Collection`. Algunos de los métodos están relacionados con insertar y obtener elementos por su clave. No obstante, en el caso de los resultados de Eloquent, las claves no coinciden con la clave primaria de las tablas que las instancias del modelo representan, y por tanto, esos métodos nos serán muy útiles. En vez de eso, cubriré los métodos que tienen algún uso.

All

El método `all()` puede ser usado para obtener la matriz interna usada por el objeto `Collection`. Esto significa que si quieres que tus resultados sean idénticos a los obtenidos por Laravel 3, usa el método `all()` y tendrás tu instancia de la matriz.

Vamos a usar `var_dump()` sobre el resultado.

He aquí nuestro código.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump($coleccion->all());
});
```

Y este es el resultado.

```
array (size=6)
  0 =>
    object(Album)[127]
      public 'timestamps' => boolean false
      protected 'connection' => null
      protected 'table' => null
      protected 'primaryKey' => string 'id' (length=2)
      protected 'perPage' => int 15
      public 'incrementing' => boolean true
      protected 'attributes' =>
        array (size=5)
          'id' => int 1
```

```
          'titulo' => string 'Some Mad Hope' (length=13)
          'artista' => string 'Matt Nathanson' (length=14)
          'genero' => string 'Acoustic Rock' (length=13)
          'ano' => int 2007
      protected 'original' =>
        array (size=5)
          'id' => int 1
          'titulo' => string 'Some Mad Hope' (length=13)
          'artista' => string 'Matt Nathanson' (length=14)
          'genero' => string 'Acoustic Rock' (length=13)
          'ano' => int 2007
      protected 'relations' =>
        array (size=0)
          empty
      protected 'hidden' =>
        array (size=0)
          empty
      protected 'visible' =>
        array (size=0)
          empty
      protected 'fillable' =>
        array (size=0)
          empty
      protected 'guarded' =>
        array (size=1)
          0 => string '*' (length=1)
      protected 'touches' =>
        array (size=0)
          empty
      protected 'with' =>
        array (size=0)
          empty
      public 'exists' => boolean true
      protected 'softDelete' => boolean false
  1 =>
    object(Album)[128]
    ... TONNES MORE INFO ...
```

Como puedes ver, tenemos una matriz para las instancias de nuestros modelos Eloquent.

First

El método `first()` de la colección puede ser usado para obtener el primer elemento del conjunto. Este será el primer elemento contenido en la matriz interna de la colección.

Vamos a probar.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump($coleccion->first());
});
```

Ahora, visitemos la URI `/` para ver el resultado. ¿Qué esperas que sea?

```
object(Album)[127]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 1
      'titulo' => string 'Some Mad Hope' (length=13)
      'artista' => string 'Matt Nathanson' (length=14)
      'genero' => string 'Acoustic Rock' (length=13)
      'ano' => int 2007
  protected 'original' =>
    array (size=5)
      'id' => int 1
      'titulo' => string 'Some Mad Hope' (length=13)
      'artista' => string 'Matt Nathanson' (length=14)
      'genero' => string 'Acoustic Rock' (length=13)
      'ano' => int 2007
  protected 'relations' =>
    array (size=0)
      empty
```

```
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
    array (size=0)
      empty
  public 'exists' => boolean true
  protected 'softDelete' => boolean false
```

¡Eso es! Es una única instancia de un modelo que representa unuo de nuestros álbumes. La primera fila que insertamos en la tabla. Observa que solo es la primera fila porque usamos el método `all()` como parte de la consulta. Si hubiéramos usado una consulta diferente, la matriz sería diferente posiblemente también en orden por lo que la llamada a `first()` daría un resultado distinto.

Last

Esta debería ser obvia. El método `first()` lo usamos para obtener el primer valor contenido en la matriz interna de la colección, por tanto el método `last()` nos dará el último elemento de la matriz.

Probemos la teoría.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump($coleccion->last());
});
```

Este es el resultado de visitar `/`.

```
object(Album)[138]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 6
      'titulo' => string '...Is A Real Boy' (length=16)
      'artista' => string 'Say Anything' (length=12)
      'genero' => string 'Indie Rock' (length=10)
      'ano' => int 2006
  protected 'original' =>
    array (size=5)
      'id' => int 6
      'titulo' => string '...Is A Real Boy' (length=16)
      'artista' => string 'Say Anything' (length=12)
      'genero' => string 'Indie Rock' (length=10)
      'ano' => int 2006
  protected 'relations' =>
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
    array (size=0)
      empty
```

```
43    public 'exists' => boolean true
44    protected 'softDelete' => boolean false
```

¡Genial! Ese es el último álbum de la matriz interna. También es el último álbum en la base de datos, pero eso depende de la consulta que usemos para obtener la colección.

Shift

El método `shift()` es similar al método `first()`. Obtendrá el primer valor de la matriz interna. No obstante, al contrario que el método `first()`, el método `shift()` también eliminará el valor de la matriz. Probémoslo con una pequeña prueba.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      $coleccion = Album::all();
8      var_dump(count($coleccion));
9      var_dump($coleccion->shift());
10     var_dump(count($coleccion));
11 });
```

Nuestra prueba mostrará el número de elementos dentro de la colección, usando el método `count()` de PHP antes y después de que hayamos usado `shift()`. Recuerda que las colecciones heredan muchas propiedades de una matriz, esto permite que el método `count()` funcione.

Echemos un vistazo al resultado de nuestra prueba.

```
1  int 6
2  object(Album)[127]
3    public 'timestamps' => boolean false
4    protected 'connection' => null
5    protected 'table' => null
6    protected 'primaryKey' => string 'id' (length=2)
7    protected 'perPage' => int 15
8    public 'incrementing' => boolean true
9    protected 'attributes' =>
10     array (size=5)
11       'id' => int 1
12       'titulo' => string 'Some Mad Hope' (length=13)
```

```
      'artista' => string 'Matt Nathanson' (length=14)
      'genero' => string 'Acoustic Rock' (length=13)
      'ano' => int 2007
  protected 'original' =>
    array (size=5)
      'id' => int 1
      'titulo' => string 'Some Mad Hope' (length=13)
      'artista' => string 'Matt Nathanson' (length=14)
      'genero' => string 'Acoustic Rock' (length=13)
      'ano' => int 2007
  protected 'relations' =>
    array (size=0)
      empty
  protected 'hidden' =>
    array (size=0)
      empty
  protected 'visible' =>
    array (size=0)
      empty
  protected 'fillable' =>
    array (size=0)
      empty
  protected 'guarded' =>
    array (size=1)
      0 => string '*' (length=1)
  protected 'touches' =>
    array (size=0)
      empty
  protected 'with' =>
    array (size=0)
      empty
  public 'exists' => boolean true
  protected 'softDelete' => boolean false
int 5
```

Como puedes ver, obtenemos nuestra instancia del modelo `Album`. Es la misma que recibimos al usar el método `first()`. No obstante, si echas un vistazo a los dos valores enteros, verás que el tamaño de la matriz se ha reducido. Esto es porque la instancia no solo ha sido devuelta desde el método, si no que también ha sido eliminada de la matriz.

Pop

Pop es un género de música que lleva varias décadas con nosotros, y es usado generalmente para animar el consumo de alcohol, o los sueños salvajes de los adolescentes.

Oh sí, también es un método de las colecciones de instancias de Modelos de Eloquent. Funciona de manera similar a `shift()` ya que devolverá el valor final de la matriz interna, y lo eliminará. Veamos el resultado en una prueba.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump(count($coleccion));
    var_dump($coleccion->pop());
    var_dump(count($coleccion));
});
```

Este es el resultado.

```
int 6
object(Album)[138]
  public 'timestamps' => boolean false
  protected 'connection' => null
  protected 'table' => null
  protected 'primaryKey' => string 'id' (length=2)
  protected 'perPage' => int 15
  public 'incrementing' => boolean true
  protected 'attributes' =>
    array (size=5)
      'id' => int 6
      'titulo' => string '...Is A Real Boy' (length=16)
      'artista' => string 'Say Anything' (length=12)
      'genero' => string 'Indie Rock' (length=10)
      'ano' => int 2006
  protected 'original' =>
    array (size=5)
      'id' => int 6
      'titulo' => string '...Is A Real Boy' (length=16)
      'artista' => string 'Say Anything' (length=12)
```

```
21        'genero' => string 'Indie Rock' (length=10)
22        'ano' => int 2006
23    protected 'relations' =>
24      array (size=0)
25        empty
26    protected 'hidden' =>
27      array (size=0)
28        empty
29    protected 'visible' =>
30      array (size=0)
31        empty
32    protected 'fillable' =>
33      array (size=0)
34        empty
35    protected 'guarded' =>
36      array (size=1)
37        0 => string '*' (length=1)
38    protected 'touches' =>
39      array (size=0)
40        empty
41    protected 'with' =>
42      array (size=0)
43        empty
44    public 'exists' => boolean true
45    protected 'softDelete' => boolean false
46  int 5
```

Recibimos el último elemento de la matriz, y el resultado de nuestros métodos `count()` sugieren que la longitud de la matriz ha sido reducida. Esto es porque el valor recibido fue eliminado de la matriz interna.

Each

Si has usado alguna vez la librería 'Underscore' para JavaScript o PHP, estos métodos que vienen ahora te serán familiares. En vez de crear un bucle `foreach()` para iterar por los resultados, podemos pasar una Closure al método `each()`.

Queda mejor descrito en un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    $coleccion->each(function($album)
    {
        var_dump($album->titulo);
    });
});
```

Nuestra Closure acepta un parámetro que será el parámetro para el objeto actual en la iteración. La closure se pasa al método `each()`. Para cada iteración, mostraremos el valor de la columna `titulo` de cada fila.

Veamos el resultado.

```
string 'Some Mad Hope' (length=13)
string 'Please' (length=6)
string 'Leaving Through The Window' (length=26)
string 'North' (length=5)
string '...Anywhere But Here' (length=20)
string '...Is A Real Boy' (length=16)
```

¡Brillante! Esos son los títulos de los álbumes como esperábamos.

Map

La función `map()` funciona de manera similar al método `each()`. No obstante, puede ser usada para iterar y funcionar con los elementos de nuestra colección, devolviendo una nueva colección como resultado.

Imaginemos que queremos prefijar todos los títulos de nuestros álbumes con `Oda a un panda divertido:`. Podemos hacerlo usando la función `map()`.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();

    $new = $coleccion->map(function($album)
    {
        return 'Oda a un panda divertido: '.$album->titulo;
    });

    var_dump($new);
});
```

Primero nos aseguramos que el valor del método `Collection::map()` es asignado a una variable. Luego iteramos por la colección de la misma forma que con el método `each()` pero esta vez devolvemos el valor que queremos que esté presente en la nueva colección.

Este es el resultado.

```
object(Illuminate\Database\Eloquent\Collection)[117]
  protected 'items' =>
    array (size=6)
      0 => string 'Oda a un panda divertido: Some Mad Hope' (length=37)
      1 => string 'Oda a un panda divertido: Please' (length=30)
      2 => string 'Oda a un panda divertido: Leaving Through The Window' (length=\
50)
      3 => string 'Oda a un panda divertido: North' (length=29)
      4 => string 'Oda a un panda divertido: ...Anywhere But Here' (length=44)
      5 => string 'Oda a un panda divertido: ...Is A Real Boy' (length=40)
```

Ahora tenemos una colección de cadenas que hemos construido usando nuestro método `map()`.

Filter

El método `filter()` puede ser usado para reducir el número de elementos contenidos en la colección resultante, usando unan Closure. Si el resultado de la Closure es el valor `true`, el elemento quedará en la colección resultante. Si por el contrario devuelve `false` o nada, el elemento no existirá en la nueva colección.

Es más fácil de entender con un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();

    $new = $coleccion->filter(function($album)
    {
        if ($album->artist == 'Something Corporate') {
            return true;
        }
    });

    var_dump($new);
});
```

Iteramos la colección con el método `filter()` y nuestra Closure. Para cada iteración, si el valor de la columna `artist` de nuestra instancia de modelo es igual a `Something Corporate`, devolveremos `true`. Esto indica que la instancia debería estar presente en la nueva colección.

He aquí el resultado.

```
object(Illuminate\Database\Eloquent\Collection)[117]
  protected 'items' =>
    array (size=2)
      2 =>
        object(Album)[135]
          public 'timestamps' => boolean false
          protected 'connection' => null
          protected 'table' => null
          protected 'primaryKey' => string 'id' (length=2)
          protected 'perPage' => int 15
          public 'incrementing' => boolean true
          protected 'attributes' =>
            array (size=5)
              'id' => int 3
              'titulo' => string 'Leaving Through The Window' (length=26)
              'artista' => string 'Something Corporate' (length=19)
              'genero' => string 'Piano Rock' (length=10)
              'ano' => int 2002
          protected 'original' =>
```

```
            array (size=5)
              'id' => int 3
              'titulo' => string 'Leaving Through The Window' (length=26)
              'artista' => string 'Something Corporate' (length=19)
              'genero' => string 'Piano Rock' (length=10)
              'ano' => int 2002
      3 =>
        object(Album)[136]
          public 'timestamps' => boolean false
          protected 'connection' => null
          protected 'table' => null
          protected 'primaryKey' => string 'id' (length=2)
          protected 'perPage' => int 15
          public 'incrementing' => boolean true
          protected 'attributes' =>
            array (size=5)
              'id' => int 4
              'titulo' => string 'North' (length=5)
              'artista' => string 'Something Corporate' (length=19)
              'genero' => string 'Piano Rock' (length=10)
              'ano' => int 2002
          protected 'original' =>
            array (size=5)
              'id' => int 4
              'titulo' => string 'North' (length=5)
              'artista' => string 'Something Corporate' (length=19)
              'genero' => string 'Piano Rock' (length=10)
              'ano' => int 2002
```

He acortado los resultados un poco para ahorrar espacio, pero puedes ver claramente que tenemos dos álbumes de “Something Corporate”.

Sort

El método `sort()` puede ser usado para ordenar la colección. Entrega nuestra closure al método `uasort()` de PHP que usa valores enteros para representar la comparación entre dos valores. Nuestra closure recibe dos parámetros, llamémoslos A y B. Luego aplicamos las reglas de ordenación para devolver un entero desde nuestra closure.

Si A > B devolvemos 1. Si A < B devolvemos -1. Si A = B devolvemos 0.

He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();

    $coleccion->sort(function($a, $b)
    {
        $a = $a->ano;
        $b = $b->ano;
        if ($a === $b) {
            return 0;
        }
        return ($a > $b) ? 1 : -1;
    });

    $coleccion->each(function($album)
    {
        var_dump($album->ano);
    });
});
```

Pasamos una closure con dos parámetros, que representan dos álbumes de la colección. Primero asignamos `$a` y `$b` a la columna `ano` para simplificar el código de comparazión. Si `$a` es igual a `$b`, devolvemos `0`. Si `$a` es mayor que `$b`, devolvemos `1`, en caso contrario devolvemos `-1`.

Este método es destructivo. Altera la colección original. Iteramos en la colección mostrando los años en el nuevo orden. Este es el resultado.

```
int 1993
int 1997
int 2002
int 2002
int 2006
int 2007
```

Como puedes ver, nuestros álbumes están ahora ordenados por `ano` en orden ascendente. He aquí algunos deberes para ti. Intenta modificar el ejemplo anterior cambiando un único caracter para que estén ordenados en orden descendente. ¡Prometo que se puede hacer!

Reverse

El método `reverse()` puede ser usado para dar la vuelta a los modelos contenidos en la matriz interna. ¿Realmente necesitas un ejemplo? Bueno, está bien... allá vamos.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();

    $coleccion->each(function($album)
    {
        var_dump($album->titulo);
    });

    $reverse = $coleccion->reverse();

    $reverse->each(function($album)
    {
        var_dump($album->titulo);
    });
});
```

Primero iteramos a través de todos los álbumes mostrando sus títulos. Luego le damos la vuelta a la colección, asignando la nueva colección a la variable `$reverse`. Luego iteramos sobre la colección `$reverse` para ver qué ha cambiado.

Este es el resultado.

```
string 'Some Mad Hope' (length=13)
string 'Please' (length=6)
string 'Leaving Through The Window' (length=26)
string 'North' (length=5)
string '...Anywhere But Here' (length=20)
string '...Is A Real Boy' (length=16)


string '...Is A Real Boy' (length=16)
string '...Anywhere But Here' (length=20)
string 'North' (length=5)
```

```
12 string 'Leaving Through The Window' (length=26)
13 string 'Please' (length=6)
14 string 'Some Mad Hope' (length=13)
```

¡Genial! Nuestra colección se ha dado la vuelta.

Merge

El método `merge()` puede ser usado para combinar dos colecciones. El único parámetro del método es la colección que debería ser combinada. He aquí un ejemplo.

```
1  <?php
2
3  // app/routes.php
4
5  Route::get('/', function()
6  {
7      $a = Album::where('artista', '=', 'Something Corporate')
8          ->get();
9      $b = Album::where('artista', '=', 'Matt Nathanson')
10         ->get();
11
12     $result = $a->merge($b);
13
14     $result->each(function($album)
15     {
16         echo $album->titulo.'<br />';
17     });
18 });
```

En el ejemplo de arriba, realizamos dos consultas. En el conjunto `$a` hay una colección de álbumes de “Something Corporate”. En el conjunto `$b` hay álbumes de “Matt Nathanson”. Usamos el método `merge()` para combinar las colecciones y asignar el resultado a una nueva colección llamada `$result`. Luego mostramos los títulos de los álbume sen un bucle `each()`.

He aquí los resultados.

```
1 Leaving Through The Window
2 North
3 Some Mad Hope
4 Please
```

Slice

El método `slice()` es el equivalente de la función del mismo nombre de PHP. Puede ser usada para crear un subconjunto de modelos usando un número de la colección. ¿Confundido? Echa un vistazo a esto.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();

    $sliced = $coleccion->slice(2, 4);

    $sliced->each(function($album)
    {
        echo $album->titulo.'<br />';
    });
});
```

Creamos una nueva colección usando el método `slice()`. El primer parámetro es la posición en la que comienza el nuevo subconjunto. Aquí le estamos diciendo que comience a cortar por el segundo elemento de la matriz. El segundo parámetro, que es opcional, es la longitud de la nueva colección . Le decimos a `slice()` que queremos cuatro elementos, tras el sgundo elemento de la matriz.

Echemos un vistazo a la matriz.

```
Leaving Through The Window
North
...Anywhere But Here
...Is A Real Boy
```

¿Sabías que podemos pasar un valor negativo al método `slice()`? Por ejemplo...

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();

    $sliced = $coleccion->slice(-2, 4);

    $sliced->each(function($album)
    {
        echo $album->titulo.'<br />';
    });
});
```

Pasando `-2` como primer parámetro, le estamos diciendo que queremos comenzar la colección dos elementos desde **el final** de la colección. Este es el resultado.

```
...Anywhere But Here
...Is A Real Boy
```

Espera, ¿no le dijimos que nos diera cuatro modelos?

Sí, sin embargo teniendo en cuenta que estamos posicionados al final de la matriz, solo hay dos elementos disponibles para obtener. El método `slice()` no da la vuelta a la colección.

IsEmpty

El método `isEmpty()` puede ser usado para revisar si el contenedor tiene o no elementos. ¡Apuesto a que no te lo esperabas! No acepta valores y devuelve un valor booleano. He aquí un ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    // This query will return items.
    $a = Album::all();

```

```
    // This query won't.
    $b = Album::where('titulo', '=', 'foo')->get();

    var_dump($a->isEmpty());
    var_dump($b->isEmpty());
});
```

Aha, ¡una trampa! Sabemos que la primera devolverá resultados y que la segunda no. Veamos el resultado del método `isEmpty()` de ambas, para ver lo que obtenemos.

```
boolean false
boolean true
```

El primer `isEmpty()` devuelve un valor `false` porque la matriz tiene elementos. La segunda coelcción, está vacía, y el método devuelve el valor `true`.

ToArray

El método `toArray()` puede ser usado para devolver la matriz interna de la colección. Además, cualquier elemento de la matriz que pueda ser transformado en una matriz, por ejemplo los objetos, serán transformados en el proceso.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump( $coleccion->toArray() );
});
```

Este es el resultado.

```
array (size=6)
  0 =>
    array (size=5)
      'id' => int 1
      'titulo' => string 'Some Mad Hope' (length=13)
      'artista' => string 'Matt Nathanson' (length=14)
      'genero' => string 'Acoustic Rock' (length=13)
      'ano' => int 2007
  1 =>
    array (size=5)
      'id' => int 2
      'titulo' => string 'Please' (length=6)
      'artista' => string 'Matt Nathanson' (length=14)
      'genero' => string 'Acoustic Rock' (length=13)
      'ano' => int 1993
  2 =>
    array (size=5)
      'id' => int 3
      'titulo' => string 'Leaving Through The Window' (length=26)
      'artista' => string 'Something Corporate' (length=19)
      'genero' => string 'Piano Rock' (length=10)
      'ano' => int 2002
      ... and more ...
```

Como pudes ver, no solo hay una matriz que representa nuestra colección, si no que las instancias han sido tambiént ransformadas en matrices.

ToJson

El método `toJson()` transformará la colección en una cadena JSON que puede ser usada para representar sus contenidos. En los capítulos anteriores descubrimos cómo devolver colecciones directamente desde una closure enrutada o acción del controlador, para servir una respuesta JSON. El método `toString()`, que permite que la colección sea transformada a JSON, hace una llamada al método `toJson()` internamente.

Echemos un vistazo al ejemplo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump( $coleccion->toJson() );
});
```

Este es el resultado del ejemplo anterior.

```
string '[{"id":1,"titulo":"Some Mad Hope","artista":"Matt Nathanson","genero":"Ac\
oustic Rock","ano":2007},{"id":2,"titulo":"Please","artista":"Matt Nathanson","ge\
nero":"Acoustic Rock","ano":1993},{"id":3,"titulo":"Leaving Through The Window","\
artista":"Something Corporate","genero":"Piano Rock","ano":2002},{"id":4,"titulo"\
:"North","artista":"Something Corporate","genero":"Piano Rock","ano":2002},{"id":\
5,"titulo":"...Anywhere But Here","artista":"The Ataris","genero":"Punk Rock","an\
o":1997},{"id":6,"titulo":"...Is A Real Boy","artista":"Say Anything","genero":"I\
ndie Rock","ano":2006}]' (length=570)
```

Es nuestra colección completa, representada por una cadena JSON.

Count

En un ejemplo anterior, usé el método `count()` de PHP para contar el número de instancias de un modelo que están dentro de la matriz interna de la colección. ¡Qué tonto fui! Me había olvidado del método `count()` de la colección. Aunque hacen lo mismo... Déjame mostrártelo.

```php
<?php

// app/routes.php

Route::get('/', function()
{
    $coleccion = Album::all();
    var_dump( $coleccion->count() );
});
```

Este es el resultado. ¡Apuesto a que no puedes esperar!

```
int 6
```

Así que el resultaod es el mismo, ¿cuál debería usar?

El que se ajuste a ti. Te dejaré esa decisión a ti. Ya llevas más de 300 páginas del libro y eres viejo y sabio.

Mejores prácticas

Algunos de los métodos disponibles en la colección son duplicados de aquellos disponibles en el constructor de consultas. Por ejemplo, podemos usar el método `first()` al crear una consulta de Eloquent para obtener un único modelo. También podemos usarlo sobre la colección para obtener el primer elemento de la colección. Echa un vistazo al siguiente ejemplo.

```
    Album::all()->first();
    Album::first();
```

Esas dos líneas llegan al mismo resultado. ¿Cuál deberíamos usar?

Bueno, como siempre, la respuesta es *depende*. Lo sé, a nadie le gusta escuchar eso, pero a veces es verdad. Echemos un ojo a diferentes escenarios.

En el primer escenario, queremos mostrar el título del primer álbum almacenado en la base de datos de nuestras plantillas. ¡No hay problemas! Podemos usar el método `first()` del constructor de consultas de Eloquent.

Album::first();

En el segundo escenario, queremos mostrar una lista de todos los títulos de los álbumes, pero también mostrar el título del primer álbum en su propia caja. Por ejemplo en la sección “Album del Año”. Bien, podríamos hacer algo así supongo.

```
$allAlbums = Album::all();
$albumDelAno = Album::first();
```

Estoy seguro de lo que tenemos qu ehacer, pero este método ejecutará dos consultas contra la base de datos. Aumentar el número de consultas es una forma de reducir el rendimiento de la aplicación rápidamente. Puede que no importe en nuestros ejemplos, pero en un programa empresarial, cada segundo es dinero perdido.

Podríamos ahorrar algo de carga, y reducir el número de consultas, delegando algo de carga a la colección.

```
$allAlbums = Album::all();
$albumDelAno = $allAlbums->first();
```

Obtenemos todos los álbumes, pero esta vez, usamos el método `first()` en la colección para obtener el álbum del año. El resultado es una sola consulta.

Elegir si realizar una consulta sobre la colección o la base de datos es importante. Consultar la base de datos permitirá búsquedas más complejas y rápidas mientras que en la colección consumiremos más CPU y memoria. Por toro lado, usando métodos en la colección podemos ahorrar consultas SQL.

Usa tu mejor juicio para decidir qué metodo usar. ¡Confío en ti!

Relaciones de Eloquent

Era una noche fresca y húmeda en ciudad Eloquent. Las gotas de lluvia bajaban por la ventana de la oficina de Zack, como lágrimas de aquellos que no tienen esperanza. Ciudad Eloquent fue una vez un lugar de paz y pureza pero ahora estaba corrupta. La paz se tornó en guerras de bandas, contrabando y otros crímenes. Las fuerzas de la ley locales fueron compradas hace largo tiempo y ahora se hacían los ciegos ante la violencia en las calles.

Zack fue una vez parte de la fuerza policial de ciudad Eloquent, pero como último hombre honesto en una ciudad corrupta, se le obligó a dejar su trabajo e intentar arreglar la ciudad a su manera. Se convirtió en investigador privado.

Zack se sentó en su antigua mesa de roble junto a la ventana de su oficina en la planta 14, con una botella de whiskey y un cigarro barato. La oficina estaba desgastada y, como Zack, no había sido limpiada en algún tiempo. No obstante, era barata, y el dinero no es que sobrara.

> Knock knock.

Una preciosa rubia alta puso un pie en la entrada. Zack se giró para encarar a la mujer angelical.

"¿Eres tú el investigador?" preguntó la rubia misteriosa.

"Mi nombre es Zack Kitzmiller muñeca, pero puedes llamarme como quieras" dijo Zack, con una sonrisa de suficiencia acentuada por su barba de varios días.

"Mr Kitzmiller, mi nombre es Pivoté Tableux, y busco al hombre que mató a mi marido. La culpa de este asesinato **pertenece a** Enrico Barnez."

"Enrico Barnez es el mayor traficante de la ciudad, está bien escondido y encontrarlo no será fácil." gruñó Zack. "Va a costarte, ¿estás seguro de que tienes dinero para cubrir esto?"

"Mister Kitzmiller, el dinero no será un problema. Mi familia **tiene muchos** fondos" dijo Pivoté.

"Eso es todo lo que necesitaba escuchar." dijo Zack con una sonrisa fría.

Zack se puso su fedora y su largo abrigo y se marchó.

Zack fue hacia la entrada del peurto. Partes de un barco roto y vehículos irreconocibles estaban por lo suelos.

"¿Eres Messy Stinkman?" gritó Zack a un trabajador del puerto.

“Quizá lo zea, quizá no lo zea… depende quié pregunte.”

“Busco a tu jefe. Enrico barnez. ¿Dónde puedo encontrarlo?” dijo Zack, con cautela.

“Mira, er jefe **tiene muchos** amigos peligrosos. ¿Cómo demonios voy a estar a salvo si te cuento onde está?” Ves este almacén, todo **pertenece a** Enrico. Todo lo que me pertence. Lo ziento amigo, simplemente no pueo ayudarte.”

Zack decidió que no iba a sacarle nada de información sin inspiración. Agarró el tubo de escape de un coche y le dio la tunda de su vida. ***Zack Kitzmiller es un auténtico malote***.

“Vale, te lo daréh. ¿Quieres saber dónde está Enrico? Date la vuelta.” lloriqueó Messy.

Zack se dio la vuelta lentamente, y se encontró con la cara de Enrico barnez. Enrico era un hombre poderoso. Muy poderoso pero bajito.

“Extraño, ¿por qué me estás buscando?”

Zack sabía cómo de peligroso era el hombre que estaba ante él. Decidió no responder, en vez de eso sacó una pistola del bolsillo de su abrigo y le apuntó a Enrico a la cabeza.

No fue lo suficientemente rápido, Enrico también sacó su pistola. Zack y Enrico se quedaron paralizados. Ambos tomaron aire y exhalaron un largo suspiro. Una pistola fue disparada y un cuerpo calló al suelo.

En la vida, no todo son palomas y perdices.

Introducción a las relaciones

En los capítulos anteriores hemos descubierto cómo representar las filas almacenadas en nuestras tablas de la base de datos como objetos. Las instancias de clases representan a una única fila. Esto significa que los hemos separado en su forma más sencilla. Libro, Fruta, BandaDeChicos, en lo que sean.

Dado que esos objetos ahora son simples, si fueramos a almacenar datos relacionados con ellos, tendríamos que crear nuevos objetos y formar relaciones entre ellos. ¿A qué nos referimos con *relaciones*? Bueno, no me refiero a abrazos y besos y sábanas mojadas por el sudor. Este tipo de relaciones se explica mejor con un ejemplo.

Vamos a tomar nuestro modelo Libro del anterior capítulo. Si pensamos en los libros por un momento, llegaremos a la conclusión de que han tenido que ser escritos por alguien. Sé que podrías pensar que yo no existo, pero la triste verdad es que sí. En algún lugar hay un Británico loco que es un fanático de Laravel. Hay otros autores por ahí también, no solo yo.

Así que ahora sabemos que un Libro **pertenece a** un autor. Esa es nuestra primera relación. El autor tiene un *enlace* al libro. En cierta manera, el autor identifica al libro. No es necesariamente una propiedad del libro, como su título por ejemplo. No, un autor tiene su propio conjunto de propiedades. Un nombre, un lugar de nacimiento, un sabor de pizza favorito. Merece ser su propio modelo. Una clase Autor.

¿Entonces cómo los enlazamos? En las bases de datos relacionales, podemos usar claves foráneas para identificar relaciones. Normalmente son columnas de Enteros.

Vamos a crear dos tablas de ejemplo.

libros

```
+---------+------------------------+
| id (PK) | nombre                 |
+---------+------------------------+
| 1       | Code Sexy              |
| 2       | Code Dutch             |
| 3       | Code Bright            |
+---------+------------------------+
```

Esta es nuestra tabla Libros con tres libros en ella. Descubrirás que cada tabla tiene una clave primaria tal y como requiere el ORM de Eloquent. Puede ser usado para identificar cada fila individual. Ahora echemos un vistazo a la segunda tabla.

autores

```
+---------+------------------------+
| id (PK) | nombre                 |
+---------+------------------------+
| 1       | Dayle Rees             |
| 2       | Matthew Machuga        |
| 3       | Shawn McCool           |
+---------+------------------------+
```

Aquí tenemos otra tabla que contiene el nombre de tres apuestos programadores. La llamaremos la tabla Autores. Observa como cada fila tiene, una vez más, una clave primaria que puede ser usada como identificador único.

Bien, formemos una relación entre ambas tablas. Es una relación entre Libro y Autor, ignoraremos co-autoría por ahora y asumiremos que un libro solo puede tener un autor. Vamos a añadir una nueva columna a la tabla Libros.

libros

```
+---------+------------------------+-----------------+
| id (PK) | nombre                   | autor_id (FK)  |
+---------+------------------------+-----------------+
| 1       | Code Sexy              | 2               |
| 2       | Code Dutch             | 3               |
| 3       | Code Bright            | 1               |
+-----------------------------------+-----------------+
```

Hemos añadido nuestra primera clave foranea. Esta es el campo `autor_id`. Una vez más, es un campo entero, pero no es usado para identificar filas en esta tabla, en vez de eso es usada para identificar filas relacionadas en otra tabla. Una *clave foranea* se usa normalmente para identificar una clave primaria en una tabla adyacente, no obstante, en algunas circunstancias puede ser usada para identificar otros campos.

¿Ves cómo la adición de una clave foranea ha creado relaciones entre el autor y un libro? Nuestra clave foranea `autor_id`, referencia la clave primaria de la tabla Libros. Echa un vistazo atentamente a la fila tres de la tabla Libros. Code Bright tiene un `autor_id` de `1`. Ahora echa un vistazo a la fila `1` de la tabla Autores y verás `Dayle Rees`. Eso significa que Code Bright fue escrito por Dayle Rees. Es tan simple como eso.

> ¿Por qué no colocamos la clave foranea en la tabla Autores?

Bueno, un autor podría tener varios libros, ¿verdad? Por ejemplo, echa un vistazo a esta relación.

autores

```
+---------+------------------------+
| id (PK) | nombre                 |
+---------+------------------------+
| 1       | Dayle Rees             |
| 2       | Matthew Machuga        |
| 3       | Shawn McCool           |
+---------------------------------+
```

libros

```
+---------+------------------------+-----------------+
| id (PK) | nombre                 | autor_id (FK)   |
+---------+------------------------+-----------------+
| 1       | Code Sexy              | 2               |
| 2       | Code Dutch             | 3               |
| 3       | Code Bright            | 1               |
| 4       | Code Happy             | 1               |
+---------+------------------------+-----------------+
```

Observa como Dayle Rees, quiero decir, yo, observa como yo... oh dios he arruinado esta frase. Observa cómo hay dos libros que me pertenecen. Tanto `Code Happy` como `Code Bright` tienen un valor `autor_id` de `1`. Esto significa que ambos fueron escritos por Dayle Rees.

Ahora intentemos expresar el ejemplo anterior con la clave foranea en la tabla Autor.

autores

```
+---------+------------------------+----------------+-----------+
| id (PK) | nombre                 | libro_uno (FK) | libro_dos |
+---------+------------------------+----------------+-----------+
| 1       | Dayle Rees             | 3              | 4         |
| 2       | Matthew Machuga        | 1              | null      |
| 3       | Shawn McCool           | 2              | null      |
+---------+------------------------+----------------+-----------+
```

libros

```
+---------+------------------------+-----------------+
| id (PK) | nombre                   | autor_id (FK)  |
+---------+------------------------+-----------------+
| 1       | Code Sexy              | 2               |
| 2       | Code Dutch             | 3               |
| 3       | Code Bright            | 1               |
| 4       | Code Happy             | 1               |
+---------+------------------------+-----------------+
```

Como puedes ver, tenemos que añadir variables claves foráneas a la tabla Autores. No solo eso, sino que dado que algunos autores no tienen dos libros, muchas de las columnas tendrán valor `null` en ellas. ¿Qué pasaría si quisiéramos que nuestros autores tuvieran tres o cuatro libros? No podemos seguir añadiendo columnas, ¡las tablas serían un desastre!

> Muy bien, mantendremos la clave foranea en la tabla Libros.

Entonces, ¿por qué no aprendemos los nombres de esos tipos de relacioens? Será útil cuando llegue la hora de implementar esos tipos de relaciones con Eloquent. Allá vamos.

Tenemos el campo `id` en el Libro, que identificar su único autor. Esto significa que **pertenece a** un Autor. Esa es nuestra primera relación.

> *N del T:* La primera aparición en negrita es una traducción literal de la relación para que sea más sencillo de comprender mientras que la segunda queda sin traducir ya que es lo que usaremos para definir las relaciones.

Libro **belongs_to** Autor

Las relaciones también tienen sus variaciones inversas. Si un Libro **pertenece a** un Autor, esto significa que un Autor **tiene muchos** Libros. Ahora hemos aprendido el nombre de otra relación.

Autor **has_many** Libro

Si el autor solo tuviera un único libro, pero la tabla `libros` contiene la clave primaria, usaríamos la relación **tiene un** / **has_one**.

Ahora, no te olvides de esos tipos, pero sigamos y aprendamos un cuarto. Ahora necesitamos otro ejemplo. Hum… ¿Qué tal un sistema de favoritos? En el que un Usuario pueda votar un Libro. Vamos a intentar expresarlo usando las relaciones que hemos descubierto.

Usuario **has_many** Libro

Libro **has_many** Usuario

La relación *has many* (tiene muchos) es la relación que creó el favorito. No necesitamos una entidad completa para expresar un favorito teniendo en cuenta que no tiene atributos. Cuando ambas relaciones y sus inversas son del tipo **has_many** tenemos que implementar un nuevo tipo de relación. Primero, en vez de decir **tiene muchos** diremos que **pertenece a muchos** (**belongs_to_many**). De esta forma no la confundiremos con la otra relación. Este nuevo tipo de relación forma una relación del tipo **muchos a muchos** y necesita una tabla adicional.

Vamos a ver la estructura de la tabla.

users

```
+---------+------------------------+
| id (PK) | nombre                 |
+---------+------------------------+
| 1       | Dayle Rees             |
| 2       | Matthew Machuga        |
| 3       | Shawn McCool           |
+---------------------------------+
```

libros

```
+---------+------------------------+
| id (PK) | nombre                 |
+---------+------------------------+
| 1       | Code Sexy              |
| 2       | Code Dutch             |
| 3       | Code Bright            |
| 4       | Code Happy             |
+---------+------------------------+
```

libro_user

```
+-------------------------+
| id  | user_id | libro_id |
+-----+---------+----------+
| 1   | 1       | 2        |
| 2   | 1       | 3        |
| 3   | 2       | 2        |
| 4   | 3       | 2        |
+-----+---------+----------+
```

Espera un minuto, ¿qué es esa tercera tabla?

¡Bien señalado! Esa será nuetra tabla de unión, o pivote, o de búsqueda, o intermedia, o… bueno, te haces una idea. Tiene muchos nombres. La documentación de Laravel suele referirse a ellas como tablas pivote, así que me quedaré con eso. Siempre que necesites una relación **many_to_many** descubrirás que necesitas una tabla pivote. Es la tabla de la base de datos que enlaza las dos entidades juntas usando dos claves foráneas para definir las filas de las otras tablas.

Mirando a las primeras dos filas de la tabla pivote, podemos ver que el usuario Dayle Rees ha añadido a favoritos tanto `Code Dutch` como `Code Bright`. También podemos ver que el usuario Matthew Machuga y Shawn McCool han añadido ambos a favoritos `Code Dutch`.

Hay un tipo adicional de relación conocida como relación **polimórfica**. Debido a su naturaleza compleja, y su largo nombre, la cubriremos en un capítulo posterior de tácticas avanzadas de Eloquent. Aun no la necesitamos.

Suficiente aprendizaje. Es hora de un aprendizaje diferente. El práctico. Ahora que hemos descubierto la variedad de relaciones disponibles, vamos a aprender cómo implementarlas en Eloquent.

Implementando relaciones

Bien, vamos a prepararnos. Primero vamos a tener que construir algunas tablas. Normalmente crearía una nueva migración para cada tabla, pero para simplificar los ejemplos, las incluiré todas en una.

Vamos a crear un sistema que use Artistas, Álbumes y Oyentes. Los oyentes son usuarios que les gusta escuchar una variedad de álbumes. Vamos a pensar en las relaciones.

- Un *Artista* **has_many** *Álbum*es.
- Un *Álbum* *belongs_to* un *Artista*.
- Un *Oyente* **belongs_to_many** *Álbum*es.
- Un *Álbum* **belongs_to_many** *Oyente*s.

Tenemos relaciones simples entre un Artista y muchos Álbumes, y una relación de muchos a muchos entre Oyentes y Álbumes. Así que pensemos sobre la estructura de nuestras tablas. Si un Álbum **pertenece a** un Artista, entonces la clave foránea identificativa de la relación, estará en la tabla Álbum. La relación muchos a muchos necesitará una tabla pivote.

Sabemos que Eloquent necesita una forma plurar del modelo para el nombre de la tabla (a menos que especifiquemos lo contrario), ¿pero cómo nombramos la tabla pivote? El formato por defecto para la tabla pivote es la forma singular de ambos modelos separados por un guión bajo _. El orden de las instancias debería ser alfabético. Eso significa que nuestra tabla pivote sería llamada `album_oyente`.

Todas las columnas de las claves foráneas siguen una convención de nombres similar. La forma del singular del modelo relacionado con `_id` de sufijo. Nuestra tabla pivote contendrá tanto `album_id` como `oyente_id`.

Echemos un vistazo a la migración construida, ahora examinaremos cualquier cosa nueva con más detalle.

```
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
4
5  // app/datebase/migrations/2013_08_26_130751_create_tables.php
6
7  class CreateTables extends Migration {
8
9          /**
10          * Run the migrations.
11          *
12          * @return void
13          */
14         public function up()
15         {
16                 Schema::create('artistas', function($table)
17                 {
18                         $table->increments('id');
```

```
19                          $table->string('nombre', 64);
20                          $table->timestamps();
21                  });
22
23                  Schema::create('albums', function($table)
24                  {
25                          $table->increments('id');
26                          $table->string('nombre', 64);
27                          $table->integer('artista_id');
28                          $table->timestamps();
29                  });
30
31                  Schema::create('oyentes', function($table)
32                  {
33                          $table->increments('id');
34                          $table->string('nombre', 64);
35                          $table->timestamps();
36                  });
37
38                  Schema::create('album_oyente', function($table)
39                  {
40                          $table->integer('album_id');
41                          $table->integer('oyente_id');
42                  });
43          }
44
45          /**
46           * Reverse the migrations.
47           *
48           * @return void
49           */
50          public function down()
51          {
52                  Schema::drop('artistas');
53                  Schema::drop('albums');
54                  Schema::drop('oyentes');
55                  Schema::drop('album_oyente');
56          }
57
58  }
```

Observa que en el momento de escribir este capítulo, no pude hacer que las claves foráneas funcionaran. Sigo recibiendo este error.

```
SQLSTATE[HY000]: General error: 1215 Cannot add foreign key constraint (SQL: alte\
r table `albums` add constraint albums_artista_id_foreign foreign key (`artista_i\
d`) references `artistas` (`id`)
```

Seguramente volveré al capítulo más tarde para corregirlo. Si tienes mejor idea de lo que está yendo mal, ¡no dejes de mandarme un correo!

Las entradas del constructor de esquemas para Álbum, Artista y Oyente son modelos Eloquent típicos, no obstante tenemos que crear aun nuestra tabla pivote. Simplemente creamos una tabla `album_oyente` y añade dos campos de entero para que actúen como claves foráneas. No necesitamos los campos de tipo timestamp ni clave primaria ya que esta tabla simplemente actúa como *unión* entre dos instancias de los modelos.

Es hora de crear nuestros modelos Eloquent. Comencemos con `Artista`.

```php
<?php

class Artista extends Eloquent
{
    // Artista __has_many__ Album
    public function albums()
    {
        return $this->hasMany('Album');
    }
}
```

¡Aquí hay algo nuevo! Dentro de la definición de nuestro modelo de Eloquent, tenemos un método relaciones. Examinemos esto con detenimiento.

```php
public function albums()
```

El nombre del método público no necesita un formato estricto. Servirá como apodoque podremos usar para refeerirnos a la relación. Este método se podría haber llamado tranquilamente `albumesRelacionados()`.

```php
return $this->hasMany('Album');
```

En el método de las relaciones devolvemos el resultado del método `$this->hasMany()`. Este es solo uno de los muchos métodos de relaciones que heredan de la clase base de Eloquent. El primer parámetro de l método de la relación es el nombre completo del modelo a relacionar. Si decidimos añadir un espacio de nombre al modelo posteriormente, tendremos que insertar la clase con espacio de nombre completa como parámetro.

Si nuestra clave foránea en la tabla `Album` tiene un nombre diferente al esquema por defecto de `artista_id`, podemos especificar un nombre alternativo para la columna como segundo parámetro opcional a este método. Por ejemplo:

```
return $this->hasMany('Album', 'the_related_artist');
```

¿Así que, qué es exactamente lo que estamos devolviendo? Bueno, no te preocupes por eso ahora. Finalicemos creando la definición del modelo primero. Luego tenemos el modelo `Album`.

```
<?php

class Album extends Eloquent
{
    // Album __belongs_to__ Artista
    public function artista()
    {
        return $this->belongsTo('Artista');
    }

    // Album __belongs_to_many__ Oyentes
    public function oyentes()
    {
        return $this->belongsToMany('Oyente');
    }
}
```

La tabla `Album` tiene dos métodos de relación. Una vez más, los nombres de los métodos públicos no son importantes. Echemos un vistazo al contenido del primero método.

```
return $this->belongsTo('Artista');
```

Teniendo en cuenta que la clave foránea existe en esta tabla, usamos el método `$this->belongsTo()` para indicar que el modelo `Album` está relacionada con el nombre del modelo, y una vez más podemos ofrecer un segundo parámetro opcional para usar un nombre de columna alternativo.

El segundo método forma uno de los lados de nuestra relación de muchos a muchos. Echemos un vistazo más de cerca.

```
return $this->belongsToMany('Oyente');
```

El método `$this->belongsToMany()` informa a Eloquent que debería buscar una tabla pivote para modelos relacionados. El primer parámetro es, una vez más, el nombre del modelo relacionado incluyendo un espacio de nombre si está presente. Esta vez, tenemos un conjunto de parámetros opcionales diferentes. Vamos a construir otro ejemplo.

```
return $this->belongsToMany('Oyente', 'my_pivot_table', 'first', 'second');
```

El segundo parámetro, que es opcional, es el nombre de la tabla pivote para que usen los objetos relacionados. El tercer y cuarto parámetro son usados para identificar esquemas de nombres alternativo para las dos claves foráneas que son usadas para relacionar nuestros objetos con la tabla pivote.

Nos falta un modelo. Echemos un vistazo más de cerca al Oyente.

```
<?php

class Oyente extends ELoquent
{
    // Oyente __belongs_to_many__ Album
    public function albums()
    {
        return $this->belongsToMany('Album');
    }
}
```

El oyente forma el lado opuesto... bueno, ¿realmente es el opuesto? El oyente forma el *otro* lado de la relación muchos a muchos. Una vez más podemos usar el método `$this->belongsToMany()` para construir la relación.

¡Genial! Nuestros modelos han sido creados. Vamos a crear algunas instancias.

Relacionando y consultando

Primero vamos a crear un Artista y un Álbum, y una asociación entre los dos. Usaré mi closure enrutada `/` ya que creo que es una forma bonita y simple de mostrar código.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $artista = new Artista;
    $artista->nombre = 'Eve 6';
    $artista->save();

    $album = new Album;
```

```
    $album->nombre = 'Horrorscope';
    $album->artista()->associate($artista);
    $album->save();

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

Espera, ¿qué es esa nueva línea?

```
$album->artista()->associate($artista);
```

Vamos a ver el método por partes. He aquí el primer método.

```
$album->artista();
```

¿Lo recuerdas? Ese es el método de relación que hemos creado en el objeto `Album`. Así que, ¿qué devuelve exactamente? El método devuelve una instancia del constructor de consultas de Eloquent, al igual que el que usamos en el capítulo anterior.

No obstante, este constructor de consultas tendrá algunas limitaciones para nosotros. El conjunto actual de resultados representado por el constructor de consultas será el `Artista` relacionado (en este caso, una colección de uno) a nuestro `Album`.

Es lo mismo que lo siguiente.

```
Artist::where('album_id', '=', __NUESTRO_ALBUM_ACTUAL__);
```

¿Es útil verdad? Podríamos encadenar en más consultas si quisiéramos. Por ejemplo:

```
$album->artista()->where('genero', '=', 'rock')->take(2);
```

Asegúrate de recordar que el constructor de consultas necesita un método activador al final de la sentencia para devolver una colección. Así:

```
$album->artista()->where('genero', '=', 'rock')->take(2)->get();
```

Esto también significa que podemos obtener una colección completa de artistas relacionados realizando lo siguiente.

```
$album->artista()->get();
```

O una única instancia usando el método `first()`.

```
$album->artista()->first();
```

¡Oh, la flexibilidad! Bien, echemos un vistazo a nuestro ejemplo anterior.

```
$album->artista()->associate($artista);
```

> No he visto una asociación antes.

No te preocupes. Porque sea nuevo no significa que sea malo. Solo las arañas son malas. Y Kevin Bacon.

El método asociado es un método disponible en las relaciones para crear una relación entre dos objetos. Actualizará la clave foránea para la relación acordemente.

Esto significa que pasando nuestra instancia de `$album` al método `associate()` habremos actualizado la columna `artista_id` de nuestra fila de la tabla `Album` con la ID del `Artista`. Si quisiéramos, podríamos gestionar la clave foránea directamente.

He aquí un ejemplo.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $artista = new Artista;
    $artista->nombre = 'Eve 6';
    $artista->save();

    $album = new Album;
    $album->nombre = 'Horrorscope';
    $album->artista_id = $artista->id;
    $album->save();

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

Ambos fragmentos de código tendrían el mismo resultado y se crearía una relación entre los dos objetos.

No te olvides de guardar el modelo con el método `save()` que pretendes relacionar, antes de pasarlo al método `associate()`. El motivo para ello es que la instancia del modelo necesitará una clave primaria para crear la relación, y el modelo solo recibirá el valor de una clave primaria al ser guardado.

Relacionar objetos que formen relaciones de muchos a muchos se hace de forma ligeramente diferente. Echemos un vistazo introduciendo el modelo `Oyente`.

```
<?php

// app/routes.php

Route::get('/', function()
{
    $artista = new Artista;
    $artista->nombre = 'Eve 6';
    $artista->save();

    $album = new Album;
    $album->nombre = 'Horrorscope';
    $album->artista()->associate($artista);
    $album->save();

    $oyente = new Oyente;
    $oyente->nombre = 'Naruto Uzumaki';
    $oyente->save();
    $oyente->albums()->save($album);

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

Vamos a centrarnos en este trozo:

```
$oyente = new Oyente;
$oyente->nombre = 'Naruto Uzumaki';
$oyente->save();
$oyente->albums()->save($album);
```

Tras haber rellenado la instancia del modelo `Oyente`, debemos guardarla. Esto es porque necesitamos tener una clave primaria antes de que podamos crear una entrada en nuestra tabla pivote.

Esta vez, en vez de usar el método `associate()` en nuestra relación, usamos el método `save()`, pasando el objeto a relacionar como primer parámetro. El efecto es el mismo, solo que esta vez la tabla pivote será actualizada para definir la relación.

Si quisieras asociar un modelo por su clave primaria directamente, puedes usar el método `attach()` que acepta la clave primaria como el primer parámetro. Por ejemplo.

```
$album->artista()->attach(2);
```

Para eliminar una relación entre dos objetos puedes usar el método `detach()` en la relación. Solo pasa la clave primaria o un objeto como primer parámetro.

Por ejemplo:

```
<?php

// app/routes.php

Route::get('/', function()
{
    $album = Album::find(5);
    $oyente = Oyente::find(2);
    $album->oyentes()->detach($oyente);

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

Podemos eliminar todas las asociaciones de un objeto que tenga con relaciones muchos a muchos llamando al método `detach()` sin parámetros en el método de la relación.

He aquí un ejemplo.

```
$album->oyentes()->detach();
```

Ahora el álbum no tendrá oyentes relacionados.

Eloquent es un tema bello y amplio, repleto de características apasionantes pero no quiero agobiarte con todas ellas ahora mismo. Solo hará que te olvides de las cosas básicas que acabamos de aprender.

Ahora mismo, tenemos todo el conocimiento que necesitamos para comenzar a crear una aplicación básica. Crearemos un sistema para gestionar la gran cantidad de juegos de Playstation 3 que tengo. ¡Odiaría olvidar los que tengo!

Si aun quieres más de Eloquent no te preocupes. Volveremos a este tema en capítulos posteriores.

Pronto...

> Ey, ¿dónde está el resto de mi libro?

Como dije claramente en la página de descripción del libro, este título está publicado en progreso. Si estás viendo esta página, significa que aun no está terminado.

Aunque no tienes que preocuparte, tengo grandes planes para este título. Seguiré escribiendo capítulos y añadiré correcciones y actualizaciones hasta que esté completo. Todos los capítulos futuros y actualizaciones están disponibles de forma gratuita para aquellos que hayan comprado el libro. Ve a Leanpub cuando recibas un email de una actualización. ¡Tan fácil como eso!

Me gustaría aprovechar ahora para agradecer a todos los que apoyáis mi escritura. Me ha encantado escribir ambos títulos y pretendo escribir más en el futuro. Si no comprárais los libros, me enviarais correos y demás, probablemente no habría encontrado este maravilloso hobby.

Si Code Happy y Code Bright te han ayudado, me alegraría si pudieras compartir la URL del libro con tus amigos. Es http://leanpub.com/codebright[28], en caso de que la hayas perdido. :)

Mi plan es el siguiente.

- Describir los conceptos del framework y sus componentes en detalle.
- Escribir un capítulo de crea una aplicación sencilla, sobre lo que hemos tratado.
- Ahondar en las características del frameowrk
- Varios capítulos de crea una aplicación con cosas más avanzadas
- Mejores práticas y trucos

También considero un capítulo FAQ basado en el feedback que obtengo del libro, por lo que si tienes una pregunta házmelo saber.

Si alguno de vosotros quiere hablar conmigo, podéis contactar conmigo en me@daylerees.com[29] o daylerees on Twitter[30]. A menudo me encontraréis en el IRC de Freenode en el canal `#laravel`, pero por favor no te ofendas si no respondo rápidamente, ¡tengo trabajo!

Gracias una vez más por ser parte de Code Bright.

Con amor,

Dayle, y su ferviente ejército de pandas rojos.

xxx

[28] http://leanpub.com/codebright
[29] mailto:me@daylerees.com
[30] http://twitter.com/daylerees