



Laravel 5.5

Documentación oficial

Prólogo

¡Colabora!

En Laraveles estamos trabajando duro traduciendo la documentación completa de Laravel. No es una tarea sencilla y es un camino largo. Buscamos colaboradores comprometidos para finalizar el proceso y ayudar a mantenerlo.

[¡Sí, quiero colaborar!](#)

Traducción del archivo: 100%

Instalación

Instalación

- # Requisitos del servidor
- # Instalar Laravel
- # Configuración

Configuración del servidor Web

- # URL amigables

Instalación

¿Prefieres aprender con vídeos? Laracasts ofrece una introducción gratuita y completa a Laravel para los recién llegados al *framework*. Es un buen lugar para comenzar tu viaje.

Requisitos del servidor

El *framework* Laravel tiene unos pocos requerimientos de sistema. Por supuesto estos requisitos son satisfechos por la máquina virtual [Laravel Homestead](#), por lo que es muy recomendable utilizar *Homestead* como su entorno de desarrollo local para Laravel.

No obstante si no está utilizando Homestead, necesitará asegurarse que el servidor cumple con los siguientes requerimientos:

- PHP >= 7.0.0

- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Instalar Laravel

Laravel utiliza [Composer](#) para gestionar sus dependencias. Por lo tanto, antes de utilizar Laravel, asegúrese de tener instalado Composer en su máquina.

A través del Instalador de Laravel

En primer lugar, descargar al instalador de Laravel usando composer:

```
composer global require "laravel/installer"
```

Asegúrese de colocar el directorio del proveedor en su `$PATH` para hacerlo accesible a todo el sistema de ficheros para que el ejecutable Laravel pueda ser localizado. Este directorio existe en diferentes sitios dependiendo de su sistema operativo; no obstante, algunas localizaciones comunes incluyen:

- MacOS: `$HOME/.composer/vendor/bin`
- Distribuciones GNU / Linux: `$HOME/.config/composer/vendor/bin`

Una vez instalado, el comando `laravel new` creará una instalación nueva de Laravel en el directorio que se especifique. Por ejemplo, `laravel new blog` creará un directorio llamado `blog` conteniendo éste una instalación limpia de Laravel con todas las dependencias instaladas:

```
laravel new blog
```

Vía `composer create-project`

También se puede instalar Laravel ejecutando el comando de Composer `create-project` en la terminal:

```
composer create-project --prefer-dist laravel/laravel blog
```

Servidor de desarrollo local

Si se tiene PHP instalado localmente y desea utilizar el servidor de desarrollo *built-in* (incluido) en PHP para su aplicación, puede utilizar el comando Artisan `serve`. Este comando arrancará un servidor de desarrollo accesible en `http://localhost:8000` :

```
php artisan serve
```

Por supuesto, las opciones más robustas para desarrollo local son [Homestead](#) y [Valet](#).

Configuración

Directorio `public`

Después de instalar Laravel, debería configurar la raíz de su servidor web para que apunte al directorio `public`. El archivo `index.php` en este directorio sirve como *front controller* para todas las peticiones HTTP que entren en su aplicación.

Ficheros de configuración

Todos los archivos de configuración de Laravel Framework se encuentran en el directorio `config`. Cada opción está documentada, por lo que es más que recomendable navegar entre los diferentes archivos y conocer las diferentes opciones.

Permisos de directorios

Después de instalar Laravel, puede ser necesario configurar algunos permisos. Los directorios dentro de `storage` y de `bootstrap/cache` deberían tener permisos de escritura para el usuario del servidor web o Laravel no funcionará. Si se utiliza la máquina virtual [Homestead](#), estos permisos ya deben estar configurados.

Clave de la aplicación

Lo siguiente que se debe hacer una vez instalado Laravel es establecer la clave de aplicación a una cadena aleatoria. Si se instala Laravel utilizando Composer o el instalador Laravel, esta clave se habrá generado automáticamente a través del comando `php artisan key:generate` .

Normalmente, esta cadena debe contener 32 caracteres de longitud. Esta clave se debe establecer en el archivo de entorno `.env`. Si no se ha renombrado el archivo `.env.example` a `.env`, debe hacerse ahora. **¡Si no se establece la clave de aplicación, las sesiones de usuario y otros datos codificados no serán seguros!**

Configuración adicional

Laravel no necesita casi ninguna otra configuración para comenzar. ¡Ya puede empezar a programar! Sin embargo, puede querer revisar el archivo `config/app.php` y su documentación. Contiene varias opciones como `timezone` y `locale` que podrías desear cambiar en función de su aplicación.

También se pueden configurar algunos componentes adicionales de Laravel, tales como:

- [Cache](#)
- [Database](#)
- [Session](#)

Configuración del servidor Web

URL amigables

Apache

Laravel incluye un fichero `public/.htaccess` que es utilizado para proveer URLs sin el *front controller* `index.php` en la ruta. Antes de servir Laravel con Apache, asegúrese de activar el módulo `mod_rewrite` para que el servidor respete el fichero `.htaccess`.

Si el fichero `.htaccess` que provee Laravel no funciona con su instalación de Apache, intente esta alternativa:

```
Options +FollowSymLinks  
RewriteEngine On  
  
RewriteCond %{REQUEST_FILENAME} !-d  
RewriteCond %{REQUEST_FILENAME} !-f  
RewriteRule ^ index.php [L]
```

Nginx

Si está utilizando Nginx, la siguiente directiva en la configuración de su sitio redireccionará todas las peticiones al controlador frontal `index.php` :

```
location / {  
    try_files $uri $uri/ /index.php?$query_string;  
}
```

Por supuesto, cuando se utiliza [Homestead](#) o [Valet](#), las URLs serán automáticamente configuradas.

Notas de publicación

- # **Esquema de versiones**
 - # **Política de Soporte**
 - # **Laravel 5.5**
-

Esquema de versiones

Las versiones de Laravel siguen la siguiente convención: `paradigma.mayor.menor`. Los lanzamientos mayores se producen cada 6 meses (febrero y agosto), mientras que los menores pueden ocurrir varias veces a la semana. Los lanzamientos menores **nunca** contendrán cambios que "rompan" el código.

Al referenciar Laravel framework o sus componentes desde una aplicación o paquete, se debe indicar alguna restricción de versión como `5.5.*`, puesto que los lanzamientos mayores de Laravel incluyen cambios que podrían "romper" el código. Sin embargo, nos esforzamos por que el proceso de actualización entre versiones mayores se pueda realizar en un día o menos.

Los cambios de paradigma se separan por años de diferencia y representan cambios fundamentales en la arquitectura y convenciones del framework. Actualmente, no hay ningún nuevo paradigma en desarrollo.

¿Por qué Laravel no Utiliza un Versionado Semántico?

Por un lado, todos los componentes opcionales de Laravel (Cashier, Dusk, Valet, Socialite, etc.) **si** usan el versionado semántico. Sin embargo, Laravel en sí mismo no lo hace. La razón es que este sistema de versionado semántico es un modo "reduccionista" de determinar si dos piezas de código son compatibles. Incluso cuando se utiliza el versionado semántico, todavía se tiene que instalar el paquete actualizado y ejecutar una batería de tests para estar seguro de que *realmente* nada es incompatible con el código base.

Por el contrario, Laravel utiliza un sistema de versiones más comunicativo con el ámbito actual del lanzamiento. Además, puesto que los lanzamientos menores **nunca** contienen cambios con roturas intencionales, no se debería recibir una actualización con rotura si la restricción de versión sigue la convención `paradigma.mayor.*`.

Política de Soporte

Las las versiones LTS –*Long Term Support* (soporte a largo plazo)– como Laravel 5.5, se garantizan 2 años de solución de problemas generales y 3 años de soluciones relativas a seguridad. Estos lanzamientos son los qué más soporte y mantenimiento tienen. Para lanzamientos generales, se solucionarán problemas durante 6 meses y fallos de seguridad durante un año.

Laravel 5.5 (LTS)

Laravel 5.5 continua con las mejoras realizadas en Laravel 5.4 añadiendo auto-detección de paquetes, recursos API / transformaciones, auto-registro de comandos de consola, encadenado de *queued jobs*, límites a los *queued jobs*, intentos de ejecución de trabajos limitados por tiempo, e-mails renderizables, excepciones reportables y renderizables, mejor gestión de excepciones, mejoras en las pruebas de bases de datos, reglas de validación personalizadas, recursos front-end para React, los métodos `Route::view` y `Route::redirect` , "bloqueos" para los drivers de caché Memcached y Redis, notificaciones bajo demanda, soporte para Chrome headless en Dusk, atajos para Blade, mejor soporte para proxy y más.

Además, Laravel 5.5 coincide con el lanzamiento de [Laravel Horizon](#), un nuevo panel de control y sistema de configuración para sus colas basadas en Redis.

Esta documentación resume las mejoras más notables del framework; sin embargo, siempre están disponibles registros de cambios más completos en GitHub.

Laravel Horizon

Horizon proporciona un hermoso dashboard y una configuración controlada por código para sus colas Redis de Laravel. Horizon le permite monitorear fácilmente las métricas clave de su sistema de cola, tales como el rendimiento del trabajo, el tiempo de ejecución y las fallas del trabajo.

Toda la configuración se almacena en un único y sencillo archivo de configuración, permitiendo que su configuración permanezca en el lugar del control del código donde todo su equipo puede colaborar.

Para obtener más información sobre Horizon, consulte la sección [full Horizon documentation](#)

Package Discovery

Hay un tutorial gratuito en video para esta característica disponible en Laracasts.

En versiones anteriores de Laravel, la instalación de un paquete requería normalmente varios pasos adicionales, como añadir el proveedor de servicios a su archivo de configuración `app` y registrar las *facades* relevantes. Sin embargo, a partir de Laravel 5.5, Laravel puede detectar y registrar automáticamente proveedores de servicios y *facades* para usted.

Por ejemplo, puede experimentar esto instalando el popular paquete `barryvdh/laravel-debugbar` en su aplicación Laravel. Una vez instalado el paquete a través de Composer, la barra de depuración estará disponible para su aplicación sin configuración adicional:

```
composer require barryvdh/laravel-debugbar
```

Los desarrolladores de paquetes sólo necesitan añadir sus proveedores de servicios y *facades* al archivo `composer.json` del paquete:

```
"extra": {  
    "laravel": {  
        "providers": [  
            "Laravel\\Tinker\\TinkerServiceProvider"  
        ]  
    }  
},
```

Para obtener más información sobre la actualización de sus paquetes para utilizar el proveedor de servicios y el descubrimiento de fachadas, consulte la documentación completa en [desarrollo de paquetes](#).

API Resources

Al crear una API, es posible que necesite una capa de transformación entre sus modelos Eloquent y las respuestas JSON que se devuelven a los usuarios de la aplicación. Las clases de recursos de Laravel le permiten transformar sus modelos y colecciones de modelos en JSON de forma fácil y expresiva.

Una clase de recursos representa un modelo único que necesita ser transformado en una estructura JSON. Por ejemplo, aquí hay una clase de recurso simple `User` :

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class User extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Por supuesto, este es sólo el ejemplo más básico de un recurso API. Laravel también proporciona una variedad de métodos para ayudarle a construir sus recursos y colecciones de recursos. Para obtener más información, consulte la [documentación completa](#) sobre recursos API.

Comando de consola Auto-Registro

Hay un tutorial de vídeo gratuito disponible en Laracasts para esta característica.

Cuando cree nuevos comandos de consola, ya no es necesario que los liste manualmente en la lista de la propiedad `$commands` del kernel de Consola. En su lugar, un nuevo método `load` es llamado desde el método `commands` de su kernel, que escaneará el directorio dado en busca de comandos de consola y los registrará automáticamente:

```
/**  
 * Register the commands for the application.  
 *  
 * @return void  
 */  
protected function commands()  
{  
    $this->load(__DIR__ . '/Commands');  
  
    // ...  
}
```

Nuevos preajustes de Frontend

Hay un tutorial gratuito en video para esta característica disponible en Laracasts.

Mientras que el scaffolding de Vue básico todavía está incluido en Laravel 5.5, ahora están disponibles varias nuevas opciones de preajuste del módulo frontal. En una nueva aplicación Laravel, puede intercambiar el scaffolding Vue por un scaffolding React usando el comando `preset` :

```
php artisan preset react
```

O bien, puede eliminarlos completamente utilizando la configuración preestablecida `none` . Esta configuración prefijada dejará su aplicación con un archivo Sass basico y algunas utilidades JavaScript simples:

```
php artisan preset none
```

Estos comandos sólo están destinados a ser ejecutados en instalaciones de Laravel nuevas. No deben utilizarse en aplicaciones existentes.

Cadena de trabajo en cola

La cadena de trabajos permite especificar una lista de los trabajos en cola que deben ejecutarse en secuencia. Si falla un trabajo de la secuencia, el resto de los trabajos no se ejecutarán. Para ejecutar una cadena de trabajos en cola, puede utilizar el método `withChain` en cualquiera de sus trabajos despachables:

```
ProvisionServer::withChain([
    new InstallNginx,
    new InstallPhp
])->dispatch();
```

Límite de tasa de trabajos en cola

Si su aplicación interactúa con Redis, ahora puede acelerar sus trabajos en cola por tiempo o concurrencia. Esta característica puede ser de ayuda cuando sus trabajos en cola interactúan con APIs que también tienen una tasa limitada. Por ejemplo, puede acelerar un determinado tipo de trabajo para que sólo se ejecute 10 veces cada 60 segundos:

```
Redis::throttle('key')->allow(10)->every(60)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

    return $this->release(10);
});
```

En el ejemplo anterior, la `key` puede ser cualquier cadena que identifique de forma única el tipo de trabajo que desea clasificar. Por ejemplo, es posible que desee construir la clave basándose en el nombre de la clase del job y los IDs de los modelos Eloquent en los que opera.

Alternativamente, puede especificar el número máximo de `workers` que pueden procesar simultáneamente un trabajo determinado. Esto puede ser útil cuando un trabajo en cola está modificando un recurso que sólo debe ser modificado por un trabajo a la vez. Por ejemplo, podemos limitar los trabajos de un determinado tipo para que sólo sean procesados por un `worker` a la vez:

```
Redis::funnel('key')->limit(1)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

    return $this->release(10);
});
```

Intentos de trabajo basados en el tiempo

Como alternativa a definir cuántas veces se puede intentar un trabajo antes de que falle, ahora puede definir un momento en el que el trabajo debería tener tiempo muerto. Esto permite que un trabajo se intente cualquier número de veces dentro de un plazo determinado. Para definir el tiempo en el que un trabajo debe finalizar, añada un método `retryUntil` a la clase de su trabajo:

```
/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addSeconds(5);
}
```

También puede definir un método `retryUntil` en sus *queued event listeners*.

Objetos de regla de validación

Hay un tutorial gratuito en video para esta característica disponible en Laracasts.

Los objetos de reglas de validación proporcionan una forma nueva y compacta de añadir reglas de validación personalizadas a su aplicación. En versiones anteriores de Laravel, se utilizó el método `Validator::extend` para añadir reglas de validación personalizadas a través de *Closures*. Sin embargo, esto puede resultar engorroso. En Laravel 5.5, un nuevo comando Artisan `make:rule` generará una nueva regla de validación en el directorio `app/Rules`:

```
php artisan make:rule ValidName
```

Un objeto de regla sólo tiene dos métodos: `passes` y `message`. El método `passes` recibe el valor del atributo y el nombre, y retorna `true` o `false` dependiendo si el valor del atributo es válido o no. El método `message` retorna el mensaje de error de la validación que debe ser usado cuando la misma falla:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class ValidName implements Rule
{
    /**
     * Determine if the validation rule passes.
     *
     * @param  string  $attribute
     * @param  mixed   $value
     * @return bool
     */
    public function passes($attribute, $value)
    {
        return strlen($value) === 6;
    }

    /**
     * Get the validation error message.
     *
     * @return string
     */
    public function message()
    {
        return 'The name must be six characters long.';
    }
}
```

Una vez definida la regla, puede utilizarla simplemente pasando una instancia del objeto de la regla con sus otras reglas de validación:

```
use App\Rules\ValidName;

$request->validate([
    'name' => ['required', new ValidName],
]);

```

Trusted Proxy Integration

When running applications behind a load balancer that terminates TLS / SSL certificates, you may notice your application sometimes does not generate HTTPS links. Typically this is because your application is being forwarded traffic from your load balancer on port 80 and does not know it should generate secure links.

To solve this, many Laravel users install the [Trusted Proxies](#) package by Chris Fidao. Since this is such a common use case, Chris' package now ships with Laravel 5.5 by default.

A new `App\Http\Middleware\TrustProxies` middleware is included in the default Laravel 5.5 application. This middleware allows you to quickly customize the proxies that should be trusted by your application:

```

<?php

namespace App\Http\Middleware;

use Illuminate\Http\Request;
use Fideloper\Proxy\TrustProxies as Middleware;

class TrustProxies extends Middleware
{
    /**
     * The trusted proxies for this application.
     *
     * @var array
     */
    protected $proxies;

    /**
     * The current proxy header mappings.
     *
     * @var array
     */
    protected $headers = [
        Request::HEADER_FORWARDED => 'FORWARDED',
        Request::HEADER_X_FORWARDED_FOR => 'X_FORWARDED_FOR',
        Request::HEADER_X_FORWARDED_HOST => 'X_FORWARDED_HOST',
        Request::HEADER_X_FORWARDED_PORT => 'X_FORWARDED_PORT',
        Request::HEADER_X_FORWARDED_PROTO => 'X_FORWARDED_PROTO',
    ];
}

```

On-Demand Notifications

A veces, es posible que necesite enviar una notificación a alguien que no esté almacenado como "usuario" de su aplicación. Utilizando el nuevo método `Notificación::route`, puede especificar información de enrutamiento de notificación ad-hoc antes de enviar la notificación:

```

Notification::route('mail', 'taylor@laravel.com')
    ->route('nexmo', '5555555555')
    ->send(new InvoicePaid($invoice));

```

Renderable Mailables

Hay un tutorial gratuito en video para esta característica disponible en Laracasts.

Los Mailables (correos enviados por su aplicación) pueden ahora ser devueltos directamente desde las rutas, permitiéndole una vista previa rápida de los diseños del mailable en el navegador:

```
Route::get('/mailable', function () {
    $invoice = App\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

Renderable & Reportable Exceptions

Hay un tutorial gratuito en vídeo disponible en Laracasts para esta característica.

En versiones anteriores de Laravel, es posible que haya tenido que recurrir a la "comprobación de tipo" en su gestor de excepciones para generar una respuesta personalizada para una excepción dada. Por ejemplo, es posible que haya escrito un código como éste en el método `render` de su manejador de excepciones:

```
/**
 * Render an exception into an HTTP response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Exception $exception
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $exception)
{
    if ($exception instanceof SpecialException) {
        return response(...);
    }

    return parent::render($request, $exception);
}
```

En Laravel 5.5, ahora puede definir un método `render` directamente en sus excepciones. Esto le permite colocar la lógica de renderizado de la respuesta personalizada directamente en la excepción, lo que ayuda a evitar la acumulación de lógica condicional en su manejador de excepciones. Si también desea personalizar la lógica de informes para la excepción, puede definir un método `report` en la clase:

```
<?php

namespace App\Exceptions;

use Exception;

class SpecialException extends Exception
{
    /**
     * Report the exception.
     *
     * @return void
     */
    public function report()
    {
        //
    }

    /**
     * Report the exception.
     *
     * @param \Illuminate\Http\Request
     * @return void
     */
    public function render($request)
    {
        return response(...);
    }
}
```

Request Validation

Hay un tutorial gratuito en video para esta característica disponible en Laracasts.

El objeto `Illuminate\Http\Request` ahora proporciona un método `validate`, lo que le permite validar rápidamente una solicitud entrante desde una ruta *Closure* o controlador:

```
use Illuminate\Http\Request;

Route::get('/comment', function (Request $request) {
    $request->validate([
        'title' => 'required|string',
        'body' => 'required|string',
    ]);

    // ...
});
```

Manejo consistente de excepciones

La gestión de excepciones de validación es ahora coherente en todo el framework.

Anteriormente, había varias ubicaciones en el framework que requerían personalización para cambiar el formato predeterminado de las respuestas de error de validación JSON. Por otro lado, el formato predeterminado para las respuestas de validación JSON en Laravel 5.5 ahora se adhiere a la siguiente convención:

```
{
    "message": "The given data was invalid.",
    "errors": {
        "field-1": [
            "Error 1",
            "Error 2"
        ],
        "field-2": [
            "Error 1",
            "Error 2"
        ],
    }
}
```

Todos los formatos de error de validación JSON pueden controlarse definiendo un único método en su clase `App\Exceptions\Handler`. Por ejemplo, la personalización siguiente formateara las respuestas de validación JSON utilizando la convención de Laravel 5.4.

```

use Illuminate\Validation\ValidationException;

/**
 * Convert a validation exception into a JSON response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Illuminate\Validation\ValidationException $exception
 * @return \Illuminate\Http\JsonResponse
 */
protected function invalidJson($request, ValidationException $exception)
{
    return response()->json($exception->errors(), $exception->status);
}

```

Bloqueos en Cache

Los controladores de caché Redis y Memcached ahora tienen soporte para obtener y liberar "cerraduras" atómicas. Esto proporciona un método simple de obtener cerraduras arbitrarias sin preocuparse por las condiciones de la prueba. Por ejemplo, antes de realizar una tarea, es posible que desee obtener un bloqueo para que ningún otro proceso intente la misma tarea que ya está en curso:

```

if (Cache::lock('lock-name', 60)->get()) {
    // Lock obtained for 60 seconds, continue processing...

    Cache::lock('lock-name')->release();
} else {
    // Lock was not able to be obtained...
}

```

O bien, puede pasar un *Closure* al método `get`. El *Closure* sólo se ejecutará si se puede obtener el bloqueo y éste se liberará automáticamente después de ejecutarse el *Closure*:

```

Cache::lock('lock-name', 60)->get(function () {
    // Lock obtained for 60 seconds...
});

```

Además, puede "bloquear" hasta que la cerradura esté disponible:

```
if (Cache::lock('lock-name', 60)->block(10)) {  
    // Wait for a maximum of 10 seconds for the lock to become available...  
}
```

Mejoras en Blade

Hay un tutorial gratuito en vídeo disponible en Laracasts para esta característica.

La programación de una directiva personalizada es a veces más compleja de lo necesario cuando se definen expresiones condicionales simples y personalizadas. Por esta razón, Blade provee un método `Blade::if` que permite definir rápidamente una directiva condicional propia utilizando *Closures*. Por ejemplo, definamos una condición personalizada que compruebe el entorno de aplicación actual. Podemos hacer esto en el método `boot` de nuestro `AppServiceProvider` :

```
use Illuminate\Support\Facades\Blade;  
  
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    Blade::if('env', function ($environment) {  
        return app()->environment($environment);  
    });  
}
```

Una vez que la estructura condicional se ha definido, es muy fácil utilizarla en nuestras plantillas:

```
@env('local')  
    // The application is in the local environment...  
@else  
    // The application is not in the local environment...  
@enderror
```

Además de la capacidad de definir fácilmente directivas condicionales de Blade personalizadas, se han añadido nuevos accesos directos para comprobar rápidamente el estado de autenticación del usuario actual:

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

New Routing Methods

Hay un tutorial gratuito en vídeo disponible en Laracasts para esta característica.

Si está definiendo una ruta que se redirige a otra URI, ahora puede utilizar el método `Route::redirect`. Este método evita el tener que definir una ruta completa o un controlador para gestionar una simple redirección:

```
Route::redirect('/here', '/there', 301);
```

Si únicamente se necesita devolver una vista desde una ruta, se puede utilizar el método `Route::view`. Al igual que el método `redirect`, este método es como un acceso directo para no tener que definir la ruta completa o un controlador. El método `view` acepta una URI como primer parámetro y un nombre de vista como segundo. Además, se le puede pasar un array de datos a la vista como tercer parámetro opcional:

```
Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

Conexiones de Base de Datos "Sticky"

La opción `sticky`

Al configurar leer/ escribir las conexiones de la base de datos, una nueva opción de `sticky` configuración está disponible:

```
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
```

La opción `sticky` es un valor *optional* que puede ser usado para permitir la lectura inmediata de grabaciones que han sido escritas en la base de datos durante el ciclo de solicitud actual. Si la opción `sticky` está habilitada y se ha realizado una operación de "escritura" contra la base de datos durante el ciclo de solicitud actual, cualquier operación adicional de "lectura" utilizará la conexión de "escritura". Esto garantiza que cualquier dato escrito durante el ciclo de solicitud pueda leerse inmediatamente desde la base de datos durante la misma solicitud. Depende de usted decidir si este es el comportamiento deseado para su aplicación.

Guía de actualización

Actualizar a 5.5.0 desde 5.4

Actualizar a 5.5.0 desde 5.4

Tiempo estimado de actualización: 1 hora

Intentamos documentar cada posible cambio de ruptura. Puesto que algunos de estos cambios de ruptura están en partes oscuras del framework, sólo una parte de estos cambios pueden afectar a su aplicación.

PHP

Laravel 5.5 requiere PHP 7.0.0 o superior.

Actualización de dependencias

Actualice su dependencia `laravel/framework` a `5.5.*` en el archivo `composer.json`. Además, debe actualizar su dependencia `phpunit/phpunit` a `~6.0`. Luego, agregue el paquete `filp/whoops` con la versión `~2.0` a la sección `require-dev` de su archivo `composer.json`. Por último, en la sección `scripts` de su archivo `composer.json`, añada el comando artisan `package:discover` al evento `post-autoload-dump`:

```
"scripts": {  
    ...  
    "post-autoload-dump": [  
        "Illuminate\\Foundation\\ComposerScripts::postAutoloadDump",  
        "@php artisan package:discover"  
    ],  
}
```

Por supuesto, no olvide examinar cualquier paquete de terceros consumido por su aplicación y verificar que está usando la versión correcta con soporte para Laravel 5.5.

Laravel Installer

Si usted usa comúnmente el instalador de Laravel vía `laravel new`, debería actualizar su paquete de instalación de Laravel usando el comando `composer global update`.

Laravel Dusk

Laravel Dusk [2.0.0](#) ha sido liberado para proporcionar compatibilidad con Laravel 5.5 y pruebas de Chrome.

Pusher

El *Pusher event broadcasting driver* ahora requiere la versión [~3.0](#).

Swift Mailer

Laravel 5.5 requiere la versión [~6.0](#) de *Swift Mailer*.

Artisan

Comandos de Auto-Loading

En Laravel 5.5, el Artisan puede detectar automáticamente los comandos para que no tenga que registrarlos manualmente en su kernel. Para aprovechar esta nueva característica, debe añadir la siguiente línea al método `commands` de su clase `App\Console\Kernel`:

```
$this->load(__DIR__.'/Commands');
```

El método `fire`

Cualquier método `fire` presente en sus comandos de Artisan debe ser renombrado para `handle`.

El comando `optimize`

Con las mejoras recientes en cache de PHP op-code, el comando `optimize` Artisan ya no es necesario. Debe eliminar cualquier referencia a este comando de sus scripts de deploy, ya que se eliminará en una futura versión de Laravel.

Autorización

El método `authorizeResource` de los controladores

Al pasar un nombre de modelo de varias palabras al método `authorizeResource`, el segmento de ruta resultante será "snake case", coincidiendo con el comportamiento de los resource controllers.

El método `before` de las políticas (Policies)

El método `before` de una política no se llamará si la clase no contiene un método cuyo nombre coincide con el nombre de la habilidad (`ability`) que se está comprobando.

Cache

Driver de base de datos

Si está usando el driver de caché de base de datos, debe ejecutar `php artisan cache: clear` al desplegar su aplicación Laravel 5.5 actualizada por primera vez.

Eloquent

El método `belongsToMany`

Si está derogando el método `belongsToMany` en su modelo Eloquent, debe actualizar su escritura del método para reflejar la adición de nuevos argumentos:

```

/**
 * Define a many-to-many relationship.
 *
 * @param string $related
 * @param string $table
 * @param string $foreignPivotKey
 * @param string $relatedPivotKey
 * @param string $parentKey
 * @param string $relatedKey
 * @param string $relation
 * @return \Illuminate\Database\Eloquent\Relations\BelongsToMany
 */
public function belongsToMany($related, $table = null, $foreignPivotKey = null,
                             $relatedPivotKey = null, $parentKey = null,
                             $relatedKey = null, $relation = null)
{
    //
}

```

BelongsToMany `getQualifiedRelatedKeyName`

El método `getQualifiedRelifiedRelifiedKeyName` ha sido renombrado a `getQualifiedRelatedKeyName`.

BelongsToMany `getQualifiedForeignKeyName`

El método `getQualifiedForeignKeyName` ha sido renombrado a `getQualifiedForeignPivotKeyName`.

Método `is` en el modelo

Si está sobreescribiendo el método `is` de su modelo Eloquent, se debe eliminar el type-hint `Model` del método. Si está usando el método `is` en su modelo Eloquent, debe eliminar el argumento `$model` del método. Esto permite que el método `<0>is` reciba nulo como argumento:

```
/**  
 * Determine if two models have the same ID and belong to the same table.  
 *  
 * @param \Illuminate\Database\Eloquent\Model|null $model  
 * @return bool  
 */  
  
public function is($model)  
{  
    //  
}
```

Propiedad `$events` en el modelo

La propiedad `$events` en tus modelos deben ser renombradas por `$dispatchesEvents`. Este cambio fue hecho porque un numero alto de usuarios necesitan definir una relación `events`, que causó conflicto con el antiguo nombre de la propiedad.

Propiedad `$parent` en Pivot

La propiedad protegida `$parent` en la clase

`\Illuminate\Base\Database\Eloquent\Relations\Pivot` ha sido renombrada a `$pivotParent`.

Métodos `create` en las relaciones

Los métodos `BelongsToMany`, `HasOneOrMany`, y `MorphOrMany` `create` de las clases han sido modificados para proporcionar un valor predeterminado para el argumento `$attributes`. Si está usando estos métodos, debería actualizar sus nombres para que coincidan con la nueva definición:

```
public function create(array $attributes = [])  
{  
    //  
}
```

Modelos borrados con "Soft Deleted"

Cuando se utiliza el "soft deleted" con un modelo, la propiedad `exists` en el modelo permanecerá `true`.

Formato de columna `withCount`

Cuando se utiliza un alias, el método `withCount` ya no agregará automáticamente `_count` al nombre de la columna resultante. Por ejemplo, en Laravel 5.4, la siguiente consulta daría lugar a que se agregue una columna `bar_count` a la consulta:

```
$users = User::withCount('foo as bar')->get();
```

Sin embargo, en Laravel 5.5, el alias se usará exactamente como se da. Si desea añadir `_count` a la columna resultante, debe especificar ese sufijo al definir el alias:

```
$users = User::withCount('foo as bar_count')->get();
```

Métodos & nombres de atributos del modelo

Para evitar el acceso a las propiedades privadas de un modelo cuando se utiliza el acceso al array, ya no es posible tener un método en el modelo con el mismo nombre que un atributo o propiedad. Si lo hace, se lanzarán excepciones al acceder a los atributos del modelo mediante el acceso al array (`$user['name']`) o el helper `data_get`.

Formato de excepción

En Laravel 5.5, todas las excepciones, incluidas las excepciones de validación, se convierten en respuestas HTTP mediante el gestor de excepciones. Además, el formato por defecto para los errores de validación JSON ha cambiado. El nuevo formato se ajusta a la siguiente convención:

```
{
    "message": "The given data was invalid.",
    "errors": {
        "field-1": [
            "Error 1",
            "Error 2"
        ],
        "field-2": [
            "Error 1",
            "Error 2"
        ],
    }
}
```

Sin embargo, si desea mantener el formato de error Laravel 5.4 JSON, puede agregar el siguiente método a su clase `App\Exceptions\Handler` :

```
use Illuminate\Validation\ValidationException;

/**
 * Convert a validation exception into a JSON response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Illuminate\Validation\ValidationException $exception
 * @return \Illuminate\Http\JsonResponse
 */
protected function invalidJson($request, ValidationException $exception)
{
    return response()->json($exception->errors(), $exception->status);
}
```

Intentos de autenticación JSON

Este cambio también afecta al formato del error de validación para los intentos de autenticación realizados sobre JSON. En Laravel 5.5, los fallos de autenticación JSON devolverán los mensajes de error siguiendo la nueva convención de formato descrita anteriormente.

Una nota sobre *Form Requests*

Si estaba personalizando el formato de respuesta de una solicitud de formulario individual, ahora debería reemplazar el método `failedValidation` y lanzar una instancia de `HttpResponseException` que contenga su respuesta personalizada:

```
use Illuminate\Http\Exceptions\HttpResponseException;

/**
 * Handle a failed validation attempt.
 *
 * @param \Illuminate\Contracts\Validation\Validator $validator
 * @return void
 *
 * @throws \Illuminate\Validation\ValidationException
 */
protected function failedValidation(Validator $validator)
{
    throw new HttpResponseException(response()->json(..., 422));
}
```

Sistema de archivos

El método `files`

El método `files` de la clase `Illuminate\Filesystem\Filesystem` ha cambiado para añadir el argumento `$hidden` y ahora devuelve un array de objetos `SplFileInfo`, similar al método `allFiles`. Anteriormente, el método `files` devolvía un array de nombres de ruta. La nueva forma es como sigue:

```
public function files($directory, $hidden = false)
```

Correo

Parámetros en desuso

Los argumentos `$data` y `$callback` fueron eliminados de los métodos `queue` y `later` del contrato `Illuminate\Contracts\Mail\MailQueue`:

```

/**
 * Queue a new e-mail message for sending.
 *
 * @param string|array|MailableContract $view
 * @param string $queue
 * @return mixed
 */
public function queue($view, $queue = null);

/**
 * Queue a new e-mail message for sending after (n) seconds.
 *
 * @param \DateTimeInterface\DateInterval|int $delay
 * @param string|array|MailableContract $view
 * @param string $queue
 * @return mixed
 */
public function later($delay, $view, $queue = null);

```

Colas

El helper `dispatch`

Si desea enviar un trabajo que se ejecuta inmediatamente y devuelve un valor del método `handle`, debe utilizar el método `dispatch_now` o `Bus::dispatch`:

```

use Illuminate\Support\Facades\Bus;

$value = dispatch_now(new Job);

$value = Bus::dispatchNow(new Job);

```

Peticiones

El método `all`

Si está sobreescribiendo el método `all` de la clase `Illuminate\Http\Request`, debería actualizarlo para reflejar el nuevo argumento `$keys`:

```
/**  
 * Get all of the input and files for the request.  
 *  
 * @param array|mixed $keys  
 * @return array  
 */  
  
public function all($keys = null)  
{  
    //  
}
```

El método `has`

El método `$request->has` devuelve `true` incluso si el valor de entrada es una cadena vacía o `null`. Se ha añadido un nuevo método `$request->filled` que proporciona el comportamiento anterior del método `has`.

El método `intersect`

Se ha eliminado el método `intersect`. Usted puede replicar este comportamiento usando `array_filter` en una llamada a `$request->only`:

```
return array_filter($request->only('foo'));
```

El método `only`

El método `only` devuelve ahora sólo los atributos que están presentes en la carga útil de petición. Si desea conservar el antiguo comportamiento del método `only`, puede utilizar en su lugar el método `all`.

```
return $request->all('foo');
```

El helper `request()`

El helper `request` ya no recuperará las claves anidadas. Si es necesario, puede utilizar el método `input` de la petición para lograr este comportamiento:

```
return request()->input('filters.date');
```

Testing

Verificación de Autenticación

Algunas verificaciones de autenticación fueron renombradas para una mejor consistencia con el resto de verificación del framework:

- `seeIsAuthenticated` ha sido renombrado como `assertAuthenticated` .
- `dontSeeIsAuthenticated` ha sido renombrado como `assertGuest` .
- `seeIsAuthenticatedAs` ha sido renombrado como `assertAuthenticatedAs` .
- `seeCredentials` ha sido renombrado como `assertCredentials` .
- `dontSeeCredentials` ha sido renombrado como `assertInvalidCredentials` .

Fake Mail

Si está usando el `fake Mail` para determinar si un *mailable* fue enviado a la cola (`queued`) durante una solicitud, ahora debería usar `Mail::assertQueued` en lugar de `Mail::assertSent` . Esta distinción le permite afirmar específicamente que el correo se ha puesto en cola para el envío en *background* y no se ha enviado durante la propia solicitud.

Tinker

Laravel Tinker ahora soporta omitir namespaces al referirse a las clases de su aplicación. Esta característica requiere un mapa optimizado de la clase Composer, por lo que debe añadir la directiva `optimize-autoloader` a la sección `config` de su archivo `composer.json` :

```
"config": {  
    ...  
    "optimize-autoloader": true  
}
```

Traducción

La `LoaderInterface`

La interfaz `Illuminate\Translation\LoaderInterface` ha sido movida a `Illuminate\Contracts\Translation\Loader` .

Validación

Metodos de validación

Todos los métodos de validación del validador son `public` en lugar de `protected`.

Vistas

Nombre de variables dinámicas con "With"

Al permitir que el método dinámico `__call` comparta variables con una vista, estas variables usarán automáticamente el "camel" case. Por ejemplo, dado lo siguiente:

```
return view('pool')->withMaximumVotes(100);
```

La variable `maximumVotes` puede ser accedida de esta manera en la plantilla:

```
{{ $maximumVotes }}
```

Directiva Blade `@php`

La directiva blade `@php` ya no acepta etiquetas en línea. En su lugar, utilice la forma completa de la directiva:

```
@php  
$teamMember = true;  
@endphp
```

Varios

También le animamos a que vea los cambios en el [Repositorio GitHub laravel/laravel](#).

Aunque muchos de estos cambios no son necesarios, es posible que desee mantener estos archivos sincronizados con su aplicación. Algunos de estos cambios se cubrirán en esta guía de actualización, pero otros, como los cambios en los archivos de configuración o los comentarios, no lo serán. Puede ver fácilmente los cambios con la [herramienta de comparación de GitHub](#) y elegir qué actualizaciones le son importantes.

Guía de contribución

- # [Reporte de errores](#)
 - # [Discusión del desarrollo del core](#)
 - # [¿Qué rama? – Branch](#)
 - # [Vulnerabilidades de seguridad](#)
 - # [Estilo de código](#)
 - # [PHPDoc](#)
 - # [StyleCI](#)
-

Reporte de errores

Para estimular la colaboración activa, Laravel recomienda *pull requests* encarecidamente, no simplemente informar de errores. Los "reportes de errores" se pueden enviar además en forma de *pull request* conteniendo un test fallido.

Sin embargo, si se presenta un reporte de error, el caso debe contener un título y una descripción clara del problema. También se debe incluir tanta información relevante como sea posible y ejemplos de código que demuestren el problema. El objetivo de un reporte es hacer fácil para ti - y otros - replicar el error y desarrollar una solución.

Recordar, los reportes se crean con la esperanza de que otros con el mismo problema sean capaces de colaborar contigo en su resolución. No se debe esperar que un reporte obtenga actividad automáticamente o que otros vengan a solucionarlo. Reportar un error sirve para ayudarnos a nosotros mismos y a otros a comenzar el camino para su solución.

El código fuente de Laravel se mantiene en GitHub, y hay repositorios para cada uno de los proyectos de Laravel:

- [Laravel Application](#)
- [Laravel Art](#)
- [Laravel Documentation](#)
- [Laravel Cashier](#)
- [Laravel Cashier para Braintree](#)

- [Laravel Envoy](#)
- [Laravel Framework](#)
- [Laravel Homestead](#)
- [Laravel Homestead Build Scripts](#)
- [Laravel Horizon](#)
- [Laravel Passport](#)
- [Laravel Scout](#)
- [Laravel Socialite](#)
- [Laravel Website](#)

Discusión del desarrollo del *core*

Puede proponer nuevas características o mejoras al comportamiento de Laravel en el [tablero de issues](#) de *Laravel Internals*. Si propone una nueva característica, implemente, al menos, parte del código necesario para completarla.

El debate sobre errores, nuevas características e implementaciones de características existentes tienen lugar en el canal `#internals` del equipo de Slack [LaraChat](#). Taylor Otwell, el encargado de Laravel, está normalmente presente en el canal de lunes a viernes de 8:00-17:00 (UTC-06:00 o America/Chicago) y esporádicamente en otros momentos.

¿Qué rama? – *Branch*

Todas las correcciones de errores deben enviarse al último *branch* (rama) estable o al *branch* LTS (5.5). Las correcciones **nunca** se deben enviar a la rama `master` a menos que se refieran a una característica que únicamente exista en la próxima versión.

Características **menores** que son **totalmente compatibles** desde versiones anteriores a la versión actual de Laravel se deben enviar a la rama de la última versión estable.

Características **mayores** se deben enviar siempre a la rama `master`, el cual contiene la próxima versión de Laravel.

Si se duda sobre si una característica se califica como menor o mayor, por favor preguntar a Taylor Otwell en el canal `#internals` de Slack [LaraChat](#).

Vulnerabilidades de seguridad

Si se descubre una vulnerabilidad de seguridad en Laravel, por favor enviar un e-mail a Taylor Otwell a taylor@laravel.com. Se abordarán sin demora todas las vulnerabilidades de seguridad.

Estilo de código

Laravel sigue el estándar de código [PSR-2](#) y el estándar de carga automática [PSR-4](#).

PHPDoc

A continuación se muestra un ejemplo válido de un bloque de documentación de Laravel. Se debe tener en cuenta que al atributo `@param` le siguen dos espacios, el tipo de argumento, dos espacios más y finaliza con el nombre de la variable:

```
/**
 * Register a binding with the container.
 *
 * @param string|array $abstract
 * @param \Closure|string|null $concrete
 * @param bool $shared
 * @return void
 */
public function bind($abstract, $concrete = null, $shared = false)
{
```

StyleCI

¡No se preocupe si el estilo de su código no es perfecto! [StyleCI](#) automáticamente fusionará cualquier corrección de estilo en el repositorio de Laravel después de fusionar las *pull requests*. Esto nos permite centrarnos en el contenido de la contribución y no en el estilo del código.

Primeros pasos

Instalación

Instalación

Requisitos del servidor

Instalar Laravel

Configuración

Configuración del servidor Web

URL amigables

Instalación

¿Prefieres aprender con vídeos? Laracasts ofrece una introducción gratuita y completa a Laravel para los recién llegados al *framework*. Es un buen lugar para comenzar tu viaje.

Requisitos del servidor

El *framework* Laravel tiene unos pocos requerimientos de sistema. Por supuesto estos requisitos son satisfechos por la máquina virtual [*Laravel Homestead*](#), por lo que es muy recomendable utilizar *Homestead* como su entorno de desarrollo local para Laravel.

No obstante si no está utilizando Homestead, necesitará asegurarse que el servidor cumple con los siguientes requerimientos:

- PHP >= 7.0.0
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Instalar Laravel

Laravel utiliza [Composer](#) para gestionar sus dependencias. Por lo tanto, antes de utilizar Laravel, asegúrese de tener instalado Composer en su máquina.

A través del Instalador de Laravel

En primer lugar, descargar al instalador de Laravel usando composer:

```
composer global require "laravel/installer"
```

Asegúrese de colocar el directorio del proveedor en su `$PATH` para hacerlo accesible a todo el sistema de ficheros para que el ejecutable Laravel pueda ser localizado. Este directorio existe en diferentes sitios dependiendo de su sistema operativo; no obstante, algunas localizaciones comunes incluyen:

- MacOS: `$HOME/.composer/vendor/bin`
- Distribuciones GNU / Linux: `$HOME/.config/composer/vendor/bin`

Una vez instalado, el comando `laravel new` creará una instalación nueva de Laravel en el directorio que se especifique. Por ejemplo, `laravel new blog` creará un directorio llamado `blog` contenido éste una instalación limpia de Laravel con todas las dependencias instaladas:

```
laravel new blog
```

Vía `composer create-project`

También se puede instalar Laravel ejecutando el comando de Composer `create-project` en la terminal:

```
composer create-project --prefer-dist laravel/laravel blog
```

Servidor de desarrollo local

Si se tiene PHP instalado localmente y desea utilizar el servidor de desarrollo *built-in* (incluido) en PHP para su aplicación, puede utilizar el comando Artisan `serve`. Este comando arrancará un servidor de desarrollo accesible en `http://localhost:8000`:

```
php artisan serve
```

Por supuesto, las opciones más robustas para desarrollo local son [Homestead](#) y [Valet](#).

Configuración

Directorio `public`

Después de instalar Laravel, debería configurar la raíz de su servidor web para que apunte al directorio `public`. El archivo `index.php` en este directorio sirve como *front controller* para todas las peticiones HTTP que entran en su aplicación.

Ficheros de configuración

Todos los archivos de configuración de Laravel Framework se encuentran en el directorio `config`. Cada opción está documentada, por lo que es más que recomendable navegar entre los diferentes archivos y conocer las diferentes opciones.

Permisos de directorios

Después de instalar Laravel, puede ser necesario configurar algunos permisos. Los directorios dentro de `storage` y de `bootstrap/cache` deberían tener permisos de escritura para el usuario del servidor web o Laravel no funcionará. Si se utiliza la máquina virtual [Homestead](#), estos permisos ya deben estar configurados.

Clave de la aplicación

Lo siguiente que se debe hacer una vez instalado Laravel es establecer la clave de aplicación a una cadena aleatoria. Si se instala Laravel utilizando Composer o el instalador Laravel, esta clave se habrá generado automáticamente a través del comando `php artisan key:generate`.

Normalmente, esta cadena debe contener 32 caracteres de longitud. Esta clave se debe establecer en el archivo de entorno `.env`. Si no se ha renombrado el archivo `.env.example` a `.env`, debe hacerse ahora. **Si no se establece la clave de aplicación, las sesiones de usuario y otros datos codificados no serán seguros!**

Configuración adicional

Laravel no necesita casi ninguna otra configuración para comenzar. ¡Ya puede empezar a programar! Sin embargo, puede querer revisar el archivo `config/app.php` y su documentación. Contiene varias opciones como `timezone` y `locale` que podrías desear cambiar en función de su aplicación.

También se pueden configurar algunos componentes adicionales de Laravel, tales como:

- [Cache](#)
- [Database](#)
- [Session](#)

Configuración del servidor Web

URL amigables

Apache

Laravel incluye un fichero `public/.htaccess` que es utilizado para proveer URLs sin el *front controller* `index.php` en la ruta. Antes de servir Laravel con Apache, asegúrese de activar el módulo `mod_rewrite` para que el servidor respete el fichero `.htaccess`.

Si el fichero `.htaccess` que provee Laravel no funciona con su instalación de Apache, intente esta alternativa:

```
Options +FollowSymLinks  
RewriteEngine On  
  
RewriteCond %{REQUEST_FILENAME} !-d  
RewriteCond %{REQUEST_FILENAME} !-f  
RewriteRule ^ index.php [L]
```

Nginx

Si está utilizando Nginx, la siguiente directiva en la configuración de su sitio redireccionará todas las peticiones al controlador frontal `index.php`:

```
location / {  
    try_files $uri $uri/ /index.php?$query_string;  
}
```

Por supuesto, cuando se utiliza [Homestead](#) o [Valet](#), las URLs serán automáticamente configuradas.

Configuración

Introducción

Configuración del entorno

Obtener la configuración del entorno

Determinar el entorno actual

Acceso a los valores de configuración

Configuración de almacenamiento caché

Modo mantenimiento

Introducción

Todos los archivos de configuración del *framework* Laravel están almacenados en el directorio `config`. Cada opción está documentada, así que no dude en consultar los archivos y familiarizarse con las diferentes opciones disponibles.

Configuración del entorno

A menudo es útil tener diferentes valores de configuración basados en el entorno donde se ejecute la aplicación. Por ejemplo, se puede utilizar un controlador de *cache* local diferente que el que se utiliza en el servidor de producción.

Para hacer esto más fácil, Laravel utiliza la librería de PHP [DotEnv](#) de Vance Lucas. En una instalación nueva de Laravel, el directorio raíz de la aplicación contendrá un archivo `.env.example`. Si usted instala Laravel través de *Composer*, este archivo automáticamente será renombrado como `.env`. De lo contrario, usted debe renombrarlo manualmente.

Su fichero `.env` no debería ser incorporado al control de versiones de su aplicación, pues cada desarrollador / servidor que utiliza la aplicación podría requerir una configuración de entorno diferente. Además, esto supondría un riesgo de seguridad en caso de que un intruso acceda a su repositorio de control de versiones, ya que cualquier credencial sensible quedaría expuesta.

Si usted está desarrollando con un equipo, puede continuar incluyendo un archivo `.env.example` con su aplicación. Estableciendo valores de ejemplo en el archivo de configuración de ejemplo, otros desarrolladores en su equipo pueden ver claramente que variables de entorno son necesarias para ejecutar su aplicación. Usted también puede crear un archivo `.env.testing`. Este archivo reemplazará el `.env` al ejecutar los tests de PHPUnit o ejecutar los comandos de Artisan con la opción `--env=testing`.

Cualquier variable en su archivo `.env` puede ser reemplazada por variables de entorno externas a nivel de servidor o de sistema.

Obtener la configuración del entorno

Todas las variables listadas en este archivo se cargarán en la variable super-global de PHP `$_ENV` cada vez que la aplicación reciba una petición. Sin embargo, usted puede utilizar el helper `env` para recuperar valores de esas variables en sus archivos de configuración. De hecho, si revisa los archivos de configuración de Laravel, puede observar que varias de las opciones ya usan este helper:

```
'debug' => env('APP_DEBUG', false),
```

El segundo valor pasado a la función `env` es el "*valor por defecto*". Este valor será usado si no existe ninguna variable de entorno para la clave dada.

Determinando el entorno actual

El entorno actual de la aplicación es determinado a través de la variable `APP_ENV` desde el archivo `.env`. Usted puede acceder a este valor mediante el método `environment` en la [facade App](#):

```
$environment = App::environment();
```

También puede pasar argumentos al método `environment` para comprobar si hay coincidencia en el entorno para un valor dado. El método retornará `true` si el entorno tiene coincidencia con alguno de los valores dados:

```
if (App::environment('local')) {  
    // The environment is local  
}  
  
if (App::environment(['local', 'staging'])) {  
    // The environment is either local OR staging...  
}
```

La detección del entorno de la aplicación actual puede ser anulada por una variable de entorno `APP_ENV` a nivel de servidor. Esto puede ser útil cuando necesite compartir la misma aplicación para diferentes entornos de configuración, de modo que se puede configurar un host para que coincida con un entorno dado en las configuraciones en su servidor.

Acceso a valores de configuración

Usted puede acceder fácilmente a los valores de configuración utilizando el *helper* global `config` desde cualquier lugar de la aplicación. Los valores de configuración pueden ser accedidos mediante una sintaxis de "punto", que incluye el nombre del archivo y la opción a la que se desea acceder. Se puede especificar un valor por defecto y será devuelto si la opción de configuración no existe:

```
$value = config('app.timezone');
```

Para establecer valores de configuración en tiempo de ejecución, pasa una matriz al *helper* `config`:

```
config(['app.timezone' => 'America/Chicago']);
```

Configuración de caché

Para mejorar el rendimiento de su aplicación, debería almacenar en la caché todos los archivos de configuración en un solo archivo utilizando el comando de *Artisan* `config:cache`. Esto combinará todas las opciones de configuración para su aplicación en un solo archivo que será cargado rápidamente por el *framework*.

Usted normalmente debe ejecutar el comando `php artisan config:cache` como parte de la rutina de implementación en producción. El comando no se debe ejecutarse durante el desarrollo local ya que las opciones de configuración necesitarán ser cambiadas frecuentemente durante el desarrollo de la aplicación.

Si usted ejecuta el comando `config:cache` durante su proceso de despliegue, debe asegurarse que sólo está llamando a la función `env` desde los archivos de configuración.

Modo mantenimiento

Cuando su aplicación está en modo de mantenimiento, una vista personalizada será mostrada para todas las peticiones en su aplicación. Esto hace muy sencillo "desactivar" la aplicación mientras se está actualizando o cuando se está realizando el mantenimiento. La comprobación del modo de mantenimiento está en la pila, por defecto, de *middleware* de su aplicación. Si la aplicación está en modo de mantenimiento, una excepción `MaintenanceModeException` será lanzada con un código de estado 503.

Para activar el modo de mantenimiento, simplemente ejecutar el comando de Artisan `down` :

```
php artisan down
```

También se pueden proporcionar las opciones `message` y `retry` para el comando `down`. El valor `message` puede ser utilizado para mostrar o registrar un mensaje personalizado, mientras que el valor `retry` será establecido como valor `Retry-After` en la cabecera HTTP:

```
php artisan down --message="Upgrading Database" --retry=60
```

Para desactivar el modo de mantenimiento, utilice el comando `up` :

```
php artisan up
```

Puede personalizar la plantilla del modo de mantenimiento predeterminado definiendo su propia plantilla en `resources/views/errors/503.blade.php` .

Mientras que la aplicación se encuentra en modo de mantenimiento, no se atenderán [trabajos encolados](#). Los trabajos continuarán siendo manejados normalmente una vez que la aplicación salga del modo de mantenimiento.

Alternativas al modo de mantenimiento

Puesto que el modo de mantenimiento requiere que su aplicación tenga varios segundos de tiempo de inactividad, considere alternativas como [Envoyer](#) para lograr un despliegue con tiempo de inactividad cero con Laravel.

Estructura de directorios

Introducción

El directorio raíz

```
# El directorio app  
# El directorio bootstrap  
# El directorio config  
# El directorio database  
# El directorio public  
# El directorio resources  
# El directorio routes  
# El directorio storage  
# El directorio tests  
# El directorio vendor
```

El directorio app

```
# El directorio Console  
# El directorio Events  
# El directorio Exceptions  
# El directorio Http  
# El directorio Jobs  
# El directorio Listeners  
# El directorio Mail  
# El directorio Notifications  
# El directorio Policies  
# El directorio Providers  
# El directorio Rules
```

Introducción

La estructura predeterminada de Laravel pretende proporcionar un punto de partida ideal tanto para grandes como para pequeñas aplicaciones. Por supuesto, puede organizar la aplicación a su gusto. Laravel apenas impone restricciones sobre donde almacenar una clase - siempre y cuando *Composer* pueda cargarla.

¿Dónde está el directorio para modelos?

Cuando se empieza con Laravel, muchos programadores se confunden al no encontrar una carpeta específica para `modelos`. Sin embargo, la falta de este directorio es intencional. La palabra *models* es ambigua puesto que puede significar cosas diferentes para gente diferente. Algunos programadores se refieren a "modelo" como la totalidad de la lógica de negocio de su aplicación, mientras que otros solo a aquella que interactúa con una base de datos relacional.

Es por esta razón que se establecen los modelos de Eloquent en el directorio `app` por defecto, permitiendo al programador guardarlos en cualquier otro sitio que elija.

El directorio *raíz*

El directorio `app`

La carpeta `app`, tal y como se puede esperar, contiene el núcleo del código de la aplicación. Se explorará este directorio en detalle a continuación; casi todas las clases de una aplicación estarán dentro de esta carpeta.

El directorio `bootstrap`

El directorio `bootstrap` contiene el archivo `app.php` el cual envuelve el framework. Esta carpeta contiene además el directorio `cache`, la cual contiene los archivos generados por el framework para mejorar su rendimiento como los archivos de caché de rutas y servicios.

El directorio `config`

El directorio `config`, como su nombre indica, contiene todos los archivos de configuración de la aplicación. Es una buena idea revisar el contenido de todos estos archivos para familiarizarse con las opciones disponibles.

El directorio `database`

La carpeta `database` contiene las migraciones de la base de datos y semillas. Si se desea, se puede utilizar esta carpeta para mantener una base de datos SQLite.

El directorio `public`

El directorio `public` contiene el archivo `index.php`, el cual es el punto de acceso para todas las peticiones que accedan a la aplicación y configura el *autoload*. Este directorio contendrá también los recursos JavaScript, CSS, imágenes y otros.

El directorio `resources`

El directorio `resources` contiene todas las vistas, así como todos los recursos LESS, SASS, JavaScript y otros archivos sin compilar. Este directorio también incluye los archivos de idioma.

El directorio `routes`

El directorio `routes` contiene todas las definiciones de rutas de la aplicación. Por defecto, varios archivos de rutas se incluyen con Laravel: `web.php` `api.php`, `console.php` y `channels.php`.

El archivo `web.php` contiene rutas que el `RouteServiceProvider` coloca en el grupo de middleware `web`, el cual provee el estado de la sesión, protección CSRF y cifrado de cookies. Si la aplicación no ofrecera una *API RESTful* que sea *Stateless*, lo mas probable es que todas las rutas sean definidas en el archivo `web.php`.

El archivo `api.php` contiene rutas que el `RouteServiceProvider` coloca en el grupo de middleware `api`, que proporciona límites de velocidad. Estas rutas están destinadas a ser *stateless*, por lo cual los *requests* que acceden a la aplicación mediante dichas rutas están destinados a ser autenticados mediante *tokens* y no tendrán acceso al estado de sesión.

El archivo `console.php` es donde se definen todos los comandos de consola basados en *Closures*. Cada *Closure* está ligada a una instancia de comando permitiendo un enfoque simple para interactuar con cada uno de los métodos *I/O* de los comandos. Aunque este archivo no define rutas HTTP, define puntos de entrada(rutas) basados en la consola en la aplicación.

El archivo `channels.php` es donde se deben registrar todos los canales que emiten eventos que sean compatibles con la aplicación.

El directorio `storage`

El directorio `storage` contiene todas plantillas Blade compiladas, sesiones y caches basados en archivos y otros archivos generados por el framework. Esta carpeta se encuentra separada en las carpetas `app`, `framework` y `logs`. La carpeta `app` debe ser usada para almacenar

cualquier archivo generado por la aplicación. El directorio `framework` se utiliza para almacenar archivos generados por el framework y caches. Finalmente, el directorio de `log` contiene los registros de logging de la aplicación.

El directorio `storage/app/public` puede ser usado para almacenar archivos generados por el usuario, tales como fotos de perfil, que deberían ser accesibles publicamente. Se debería crear un enlace simbólico en `public/storage` que apunte a esta carpeta. Se puede crear el enlace utilizando el comando `php artisan storage:link`.

El directorio `tests`

El directorio `tests` contiene las pruebas automatizadas. Se proporciona el ejemplo [PHPUnit](#) de fabrica. Cada clase test debería poseer el sufijo `Test`. Se pueden ejecutar las pruebas utilizando los comandos `phpunit` o `php vendor/bin/phpunit`.

El directorio `vendor`

El directorio `vendor` contiene las dependencias de [Composer](#).

El directorio `app`

La mayor parte de la aplicación reside en el directorio `app`. Por defecto, este directorio contiene el namespace `App` y se carga con Composer utilizando el [estándar de carga automática PSR-4](#).

El directorio `app` contiene una variedad de directorios adicionales tales como `Console`, `Http` y `Providers`. Los directorios `Console` y `Http` proveen un API al núcleo de la aplicación. Tanto el protocolo HTTP como CLI son mecanismos para interactuar con la aplicación, pero no contienen lógica. En otras palabras, son simplemente dos formas de emitir órdenes a la aplicación. El directorio `Console` contiene todos los comandos de Artisan, mientras que `Http` contiene los controladores, filtros y *requests*.

Al utilizar los comandos `make` de Artisan para crear clases se crearan distintas carpetas dentro del directorio `app`. Por ejemplo la carpeta `app/Jobs` no existirá hasta que se ejecute el comando `make:job` para crear un job.

Muchas de las clases en el directorio `app` pueden ser crear mediante comandos Artisan. Para listar los comandos disponibles, ejecutar el siguiente comando en la terminal `php artisan list make`.

El directorio `Console`

El directorio `Console` contiene todos los comandos personalizados de su aplicación. Estos comandos pueden generarse usando el comando `make:command`. Este directorio además almacena el *kernel* de la consola, que es dónde se registran los comandos Artisan personalizados y se definen las [tareas programadas](#).

El directorio `Events`

Por defecto este directorio no existe, pero será creado por el comando `event:generate` y `make:event`. El directorio `Events`, como era de esperar, almacena [clases de eventos](#). Los eventos se utilizan para alertar a otras partes de la aplicación de que una acción concreta ha ocurrido, proporcionando flexibilidad y desacoplamiento.

El directorio `Exceptions`

El directorio `Exceptions` contiene el gestor de excepciones y es un buen lugar para definir cualquier excepción que pueda producir la aplicación. Si se desea personalizar el modo en que se almacenan o se muestran las excepciones, se debe modificar la clase `Handler` en este directorio.

El directorio `Http`

El directorio `Http` contiene los *controladores*, *middleware* y *form requests*. Casi toda la lógica para gestionar peticiones se encuentra en este directorio.

El directorio `Jobs`

Este directorio no existe por defecto, pero será creado al ejecutar el comando Artisan `make:job`. El directorio `jobs` alberga los [queueable Jobs](#). Laravel puede poner en cola los *Jobs* o ejecutarlos de forma síncrona dentro del ciclo de vida de la petición actual. Los *Jobs* que se ejecutan de manera síncrona durante la petición actual son conocidos como "comandos" ya que son la implementación del *command pattern*.

El directorio `Listeners`

Este directorio no existe por defecto, pero será creado al ejecutar uno de los comandos de Artisan `event:generate` o `make:listener`. El directorio `Listeners` contiene las clases que gestionan los [eventos](#). Los *event listeners* reciben una instancia de un evento y responden en base a la lógica del evento en ejecución. Por ejemplo, un evento `UserRegistered` puede gestionarse por un listener `SendWelcomeEmail`.

El directorio `Mail`

Este directorio no existe por defecto, pero sera creado al ejecutar el comando de Artisan `make:mail` . El directorio `Mail` contiene todas las clases que representan correos electrónicos enviados por Laravel. Los objetos `Mail` permiten encapsular toda la lógica de construir un correo electrónico en una simple clase, este correo electrónico se puede enviar utilizando la función `Mail::send` .

El directorio `Notifications`

Este directorio no existe por defecto, pero sera creado al ejecutar el comando de Artisan `make:notification` . El directorio `Notifications` contiene todas las notificaciones que son enviadas por Laravel, como notificaciones simples acerca de eventos que se ejecutan en Laravel. Las notificaciones de Laravel ofrecen una variedad de drivers para ser enviadas, tales como email, Slack, SMS, o ser guardadas en una base de datos.

El directorio `Policies`

Este directorio no existe por defecto, pero sera creado si ejecuta el comando de Artisan `make:policy` . El directorio `Policies` contiene las clases de políticas de autorización para su aplicación. Las políticas se utilizan para determinar si un usuario puede realizar una acción determinada contra un recurso. Para obtener más información, consulte la [documentación de la autorización](#).

El directorio `Providers`

El directorio `Providers` contiene todos los [proveedores de servicios](#) para su aplicación. Los *service providers* inician su aplicación vinculando servicios en el *service container*, registrando eventos o realizando cualquier otra tarea para preparar su aplicación para las solicitudes entrantes.

En una aplicación nueva de Laravel, este directorio ya contendrá varios proveedores. Se es libre de agregar sus propios proveedores a este directorio según sea necesario.

El directorio `Rules`

Este directorio no existe por defecto, pero sera creado al ejecutar el comando de Artisan `make:rule` . El directorio `Rules` contiene los objetos de reglas de validación personalizados para su aplicación. Las reglas se utilizan para encapsular una lógica de validación complicada en un objeto simple. Para obtener más información, consulte la [documentación de validación](#).

Laravel Homestead

Introducción

Instalación & configuración

- # Primeros pasos
- # Configurar Homestead
- # Levantar la box de Vagrant
- # Instalación por proyecto
- # Instalando MariaDB
- # Instalando *ElasticSearch*
- # Alias

Uso diario

- # Accediendo globalmente a Homestead
- # Conectando a través de SSH
- # Conectar a bases de datos
- # Añadir nuevos sitios
- # Variables de entorno
- # Configurar programaciones cron
- # Configurar Mailhog
- # Puertos
- # Compartir el entorno
- # Múltiples versiones de PHP

Intefaces de red

Actualizar Homestead

Versiones antiguas

Configuración específica del proveedor

- # VirtualBox

Introducción

Laravel se esfuerza por hacer toda la experiencia de desarrollo de PHP agradable, incluyendo el entorno de desarrollo local. [Vagrant](#) provee una simple y elegante manera de gestionar y suministrar máquinas virtuales.

Laravel Homestead es una *Vagrant box* oficial que le provee de un maravilloso entorno de desarrollo sin requerirle instalar PHP, un servidor web, y cualquier otras aplicaciones de servidor en su máquina local. No más preocupaciones acerca de estropear su sistema operativo! Las "boxes" de Vagrant son completamente desecharables. Si algo sale mal, ¡se puede destruir la box y crearla nuevamente en cuestión de minutos!

Homestead se puede ejecutar sobre cualquier sistema Windows, Mac o Linux e incluye el servidor web Nginx, PHP 7.1, PHP 7.0, PHP 5.6, MySQL, PostgreSQL, Redis, Memcached, Node, y todas las características necesarias para desarrollar aplicaciones con Laravel.

Si está utilizando Windows, podría necesitar activar la virtualización por *hardware* (VT-x). Normalmente, ésta puede activarse a través de su BIOS. Si está utilizando Hyper-V en una sistema UEFI, además puede necesitar desactivar Hyper-V para acceder a VT-x.

Software Incluído

- Ubuntu 16.04
- Git
- PHP 7.1
- PHP 7.0
- PHP 5.6
- Nginx
- MySQL
- MariaDB
- Sqlite3
- PostgreSQL
- Composer
- Node (With Yarn, Bower, Grunt, and Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog

- ngrok

Instalación & Configuración

Primeros Pasos

Antes de arrancar tu entorno de *Homestead*, deberá instalar [VirtualBox 5.1 VMWare](#), or [Parallels](#) así como [Vagrant](#). Todos estos programas poseen instaladores visuales de fácil uso para los todos los sistemas operativos más populares.

Par usar como proveedor VMWare, necesitará adquirir VMWare Fusion o Workstation y el complemento [VMware Vagrant](#). Aunque no es gratuito, VMWare puede proporcionar un mejor rendimiento en carpetas compartidas, sin configuración extra.

Para utilizar el proveedor Parallels, necesitará instalar el complelto [Parallels Vagrant](#). Este es gratis.

Instalando Homestead con Vagrant

Una vez que VirtualBox / VMware y Vagrant han sido instalados, debe agregar la *box* de [laravel/homestead](#) a la instalación de Vagrant, utilizando el siguiente comando desde su consola. Tomará algunos minutos para que la *box* se descargue, esto dependerá de la velocidad de la conexión a internet:

```
vagrant box add laravel/homestead
```

Si este comando falla, asegúrese de que su instalación de Vagrant está actualizada.

Instalando Homestead

Puede instalar Homestead simplemente clonando el repositorio. Considere clonar el repositorio dentro de una carpeta llamada [Homestead](#) en su directorio *home*, así la Homestead *box* servirá como host para todos sus projectos Laravel:

```
cd ~  
  
git clone https://github.com/laravel/homestead.git Homestead
```

Debe consultar una versión etiquetada de Homestead ya que la rama `master` puede no ser siempre estable. Puede encontrar la última versión estable en la página [GitHub Release](#):

```
cd Homestead

// Clone the desired release...
git checkout v6.5.0
`
```

Una vez haya clonado el repositorio Homestead, ejecute el comando `bash init.sh` desde el directorio Homestead para crear el archivo de configuración `Homestead.yaml`. El fichero `Homestead.yaml` será colocado en el directorio Homestead:

```
// Mac / Linux...
bash init.sh

// Windows...
init.bat
```

Configurar Homestead

Establecer proveedor

La clave `proveedor` en su fichero `Homestead.yaml` indica que proveedor Vagrant debería ser usado: `virtualbox`, `vmware_fusion`, `vmware_workstation`, o `parallels`. Puede configurar el proveedor que prefiera:

```
provider: virtualbox
```

Configurar directorios compartidos

La propiedad `folders` del archivo `Homestead.yaml` enumera todas las carpetas que dese compartir con el entorno Homestead. Cuando los ficheros en esas carpetas cambien, se mantendrán sincronizados entre su máquina local y el entorno Homestead. Puede configurar tantas carpetas como sean necesarias:

```
folders:
```

- map: ~/code
- to: /home/vagrant/code

Si sólo está creando algunos sitios, este mapeo genérico funcionará bien. No obstante, cuando el número de sitios continúe creciendo, puede empezar a experimentar problemas de velocidad. Este problema puede ser muy exasperante en máquinas de bajo rendimiento o proyectos que contengan una gran cantidad de ficheros. Si experimenta este problema, pruebe a mapear cada proyecto sobre su propia carpeta Vagrant:

```
folders:
```

- map: ~/code/project1
- to: /home/vagrant/code/project1

- map: ~/code/project2
- to: /home/vagrant/code/project2

Para activar [NFS](#), sólo necesita añadir un simple indicador a su configuración de carpetas sincronizadas:

```
folders:
```

- map: ~/code
- to: /home/vagrant/code
- type: "nfs"

Cuando utilice NFS, debería considerar instalar el complemento `vagrant-bindfs`. Este complemento mantendrá los permisos de usuario y grupo correctos para los ficheros y directorios dentro de *Homestead box*.

Puede pasar cualquiera de las opciones soportadas por las [carpetas sincronizadas](#) de Vagrant enumerándolas en la sección de [opciones](#) :

```
folders:
```

```
- map: ~/code  
  to: /home/vagrant/code  
  type: "rsync"  
  
  options:  
    rsync__args: ["--verbose", "--archive", "--delete", "-zz"]  
    rsync__exclude: ["node_modules"]
```

Configurar sitios Nginx

¿No está familiarizado con Nginx? No hay problema. La propiedad `sites` le permite de una manera sencilla mapear un "dominio" a una carpeta en su entorno Homestead. Un ejemplo de configuración de una sitio web está incluído en el archivo `Homestead.yaml`. De nuevo, puede añadir tantos sitios a su entorno Homestead como sean necesarios. Homestead puede servir como un entorno virtualizado para cada proyecto de Laravel en el que esté trabajando:

```
sites:
```

```
- map: homestead.test  
  to: /home/vagrant/code/Laravel/public
```

Si cambia la propiedad `sites` después de provisionar *Homestead box*, debería volver a ejecutar `vagrant reload --provision` para actualizar la configuración de Nginx en la máquina virtual.

El archivo Hosts

Debe agregar los "dominios" para sus sitios web en Nginx en el fichero `hosts` en su máquina. El fichero `hosts` redireccionará las peticiones para sus sitios Homestead hacia su máquina Homestead. En Mac y Linux, este archivo está ubicado en `/etc/hosts`. En Windows el archivo se encuentra ubicado en `C:\Windows\System32\drivers\etc\hosts`. Las líneas que se deben añadir al archivo deben parecerse a las siguientes:

```
192.168.10.10 homestead.test
```

Asegúrese que la dirección IP listada es la misma que está establecida en su fichero `Homestead.yaml`. Una vez haya agregado el domino a su fichero `hosts` y lanzado *Vagrant box* será capaz de acceder al sitio web a través de su navegador web:

```
http://homestead.test
```

Levantando Vagrant box

Una vez que ha editado `Homestead.yaml` a su gusto, ejecute el comando `vagrant up` desde el directorio de Homestead. Vagrant iniciará la máquina virtual y automáticamente configurará los directorios compartidos y sitios web en Nginx.

Para apagar la máquina, puede utilizar el comando `vagrant destroy --force`.

Instalación por proyecto

En lugar de instalar Homestead de manera global y compartir el mismo *Homestead box* entre todos sus proyectos, puede configurar una instancia de Homestead para cada proyecto que gestione. Instalando Homestead para cada proyecto podrá beneficiarse si desea entregar el fichero `Vagrantfile` con su proyecto, permitiendo a otros trabajar en el proyecto usando simplemente `vagrant up`.

Para instalar Homestead directamente en su proyecto, se requiere la utilización de Composer:

```
composer require laravel/homestead --dev
```

Una vez que Homestead ha sido instalado, utilizar el comando `make` para generar el fichero `VagrantFile` y `Homestead.yaml` en la raíz del proyecto. El comando `make` configurará automáticamente las directivas de los `sitios` y `carpetas` en el archivo `Homestead.yaml`.

Mac / Linux:

```
php vendor/bin/homestead make
```

Windows:

```
vendor\bin\homestead make
```

A continuación, ejecute el comando `vagrant up` en su terminal y desde su navegador, acceda a su proyecto en `http://homestead.test`. Recordar, todavía es necesario añadir una fila al archivo `/etc/hosts` para el dominio `homestead.test` o el dominio elegido.

Instalando MariaDB

Si prefiere utilizar MariaDB en lugar de MySQL, puede agregar la opción `mariadb` a su fichero `Homestead.yaml`. Esta opción eliminará MySQL e instalará MariaDB. MariaDB sirve como reemplazo de MySQL así que deberías seguir utilizando el manejador de base de datos `mysql` en la configuración de la base de datos de su aplicación:

```
box: laravel/homestead
ip: "192.168.10.10"
memory: 2048
cpus: 4
provider: virtualbox
mariadb: true
```

Instalación de *Elasticsearch*

Para instalar *ElasticSearch*, agrege la opción `elasticsearch` a su fichero `Homestead.yaml`. La instalación por defecto creará un cluster llamado 'homestead' y reservará para él 2Gb de memoria. No debería proporcionarle a *Elasticsearch* más de la mitad de la memoria destinada a su sistema operativo, asegúrese que su máquina *Homestead* tiene al menos 4Gb de memoria:

```
box: laravel/homestead
ip: "192.168.10.10"
memory: 4096
cpus: 4
provider: virtualbox
elasticsearch: true
```

Alias

Podrá agregar alias *Bash* a su máquina *Homestead* modificando el fichero `aliases` dentro de su directorio Homestead:

```
alias c='clear'
alias ..='cd ..'
```

Después de haber actualizado el fichero `aliases`, debería recargar la máquina *Homestead* usando el comando `vagrant reload --provision`. Esto asegurará que sus nuevos alias están disponibles en la máquina.

Uso diario

Accediendo globalmente a *Homestead*

A veces puede necesitar arrancar su máquina *Homestead* `vagrant up` desde cualquier parte de su sistema. Para conseguir esto en sistemas Mac / Linux agregando una función *Bash* a su perfil *Bash*. En Windows, puede realizar esto agregando un fichero "batch" a su `PATH`. Estos *scripts* le permitirán ejecutar cualquier comando Vagrant desde cualquier sitio en su sistema y automáticamente apuntará a su instalación *Homestead*:

Mac / Linux

```
function homestead() {  
    cd ~/Homestead && vagrant $*  
}
```

Asegúrese de cambiar el camino `~/Homestead` en la función por la situación de su instalación actual de *Homestead*. Una vez la función esté instalada, podrá ejecutar comandos como `homestead up` o `homestead ssh` desde cualquier parte de su sistema.

Windows

Cree un fichero batch `homestead.bat`, en cualquier parte de su máquina, con el siguiente contenido:

```
@echo off

set cwd=%cd%
set homesteadVagrant=C:\Homestead

cd /d %homesteadVagrant% && vagrant %*
cd /d %cwd%

set cwd=
set homesteadVagrant=
```

Asegúrese de cambiar el camino de ejemplo `C:\Homestead` en el *script* a la situación actual de su instalación *Homestead*. Después de crear el fichero, agrege la situación a su `PATH`. Puede ahora ejecutar comandos como `homestead up` o `homestead ssh` desde cualquier parte de su sistema.

Conectando a través de SSH

Puede, usando SSH, entrar a su máquina virtual introduciendo en su terminal el comando `vagrant ssh` desde el directorio *Homestead*.

Pero probablemente necesitará frecuentemente entrar a su máquina utilizando SSH, considere agregar la "función" descrita a con anterioridad a su máquina anfitrión para usarla rápidamente.

Conectando a las bases de datos

La base de datos `homestead` está configurada para los sistemas MySQL y PostgreSQL desde el inicio. Para su conveniencia, el fichero `.env` configura el *framework* para utilizar esta base de datos desde el inicio.

Para conectar con su base de datos MySQL o PostgreSQL desde su cliente de base de datos en su sistema anfitrión, debería conectar a `127.0.0.1` y el puerto `33060` (MySQL) o `54320` (PostgreSQL). El nombre de usuario y contraseña para ambas bases de datos es `homestead` / `secret`.

Debería solo usar estos puertos no estándar cuando conecte con bases de datos desde su máquina anfitrión. Utilizará los puertos por defecto 3306 y 5432 en su configuración Laravel de la base de datos dado que Laravel está ejecutándose dentro de una máquina virtual.

Añadiendo Sitios adicionales

Una vez su entorno *Homestead* está aprovisionado y corriendo, puede querer agregar *sitios Nginx* adicionales para sus aplicaciones Laravel. Puede ejecutar cuantas instalaciones Laravel desee en un entorno único de *Homestead*. Para agregar *sitios* adicionales, simplemente agréguelos a su fichero [Homestead](#) :

```
sites:  
  
- map: homestead.test  
  to: /home/vagrant/code/Laravel/public  
- map: another.test  
  to: /home/vagrant/code/another/public
```

Si *Vagrant* no está gestionando su fichero "hosts" automáticamente, además deberá añadir el nuevo *sitio* a ese fichero:

```
192.168.10.10  homestead.test  
192.168.10.10  another.test
```

Una vez que el *sitio* ha sido agregado, ejecute el comando `vagrant reload --provision` desde su directorio *Homestead.

Tipos de *sitios*

Homestead soporta varios tipos de *sitios* los cuales le permiten fácilmente ejecutar proyectos que no estén basados en Laravel. Por ejemplo, podemos, fácilmente, agregar una aplicación *Symfony* a *Homestead* usando el tipo de *sitio* [symfony2](#) :

```
sites:

- map: symfony2.test
  to: /home/vagrant/code/Symfony/web
  type: symfony2
```

Los tipos de *sitios* disponibles son: `apache` , `laravel` (el defecto), `proxy` , `silverstripe` , `statamic` , `symfony2` , y `symfony4` .

Parámetros del sitio

Puede agregar a *Nginx* valores `fastcgi_param` adicionales a su *sitio* a través de la directiva de *sitio* `params` . Por ejemplo, nosotros agregamos un parámetro `F00` con el valor de `BAR` :

```
sites:

- map: homestead.test
  to: /home/vagrant/code/Laravel/public
  params:
    - key: F00
      value: BAR
```

Variables de entorno

Se pueden establecer variables de entorno globales añadiéndolas al archivo `Homestead.yaml` :

```
variables:

- key: APP_ENV
  value: local
- key: F00
  value: bar
```

Tras actualizar `Homestead.yaml` , asegúrese de re-provisionar la máquina ejecutando `vagrant reload --provision` . Esto actualizará la configuración de PHP-FPM para todas las versiones de PHP instaladas y actualizará el entorno para el usuario de `vagrant` .

Configurando programaciones Cron

Laravel ofrece una manera sencilla de [agendar tareas al Cron](#), agregando el comando de Artisan `schedule:run` especificando que se ejecute cada minuto. El comando `schedule:run` examinará la programación de trabajo definida en su clase `App\Console\Kernel` para determinar cuales trabajos deberían ser ejecutados.

Si quieres que el comando `schedule:run` corra para un sitio de Homestead en particular, deberás asignar la opción `schedule` a `true` cuando definas el sitio:

```
sites:  
  
    - map: homestead.test  
      to: /home/vagrant/code/Laravel/public  
      schedule: true
```

La tarea Cron para el sitio será definida en directorio `/etc/cron.d` de tu máquina virtual.

Configurando Mailhog

Mailhog le permite fácilmente capturar sus correos electrónicos salientes y examinarlos sin realmente enviarlos a sus destinatarios. Para empezar, actualice su fichero `.env` para usar la siguiente configuración de correo:

```
MAIL_DRIVER=smtp  
MAIL_HOST=localhost  
MAIL_PORT=1025  
MAIL_USERNAME=null  
MAIL_PASSWORD=null  
MAIL_ENCRYPTION=null
```

Puertos

Por defecto, los siguientes puertos se redirigen al entorno Homestead:

- **SSH:** 2222 → redirigido a 22
- **ngrok UI:** 4040 → redirigido a 4040
- **HTTP:** 8000 → redirigido a 80
- **HTTPS:** 44300 → redirigido a 443
- **MySQL:** 33060 → redirigido a 3306
- **PostgreSQL:** 54320 → redirigido a 5432

- **Mailhog:** 8025 → redirigido a 8025

Redirigir Otros Puertos

Si desea, se pueden redirigir puertos adicionales a la box de Vagrant, así como especificar su protocolo:

```
ports:  
  
  - send: 50000  
    to: 5000  
  - send: 7777  
    to: 777  
  protocol: udp
```

Compartiendo tu entorno

A veces puede desear compartir lo que está trabajando con otros compañeros o con un cliente. Vagrant integra un sistema que a través de `vagrant share` soporta esto, no obstante, esto no funcionará si tiene multiples *sitios* configurados en su fichero `Homestead.yaml`.

Para resolver este problema, *Homestead* incluye su propio comando `share`. Para empezar, conecte a través de SSH con su máquina *Homestead* con el comando `vagrant ssh` y ejecute `share homestead.test`. Esto compartirá el *sitio* `homestead.test` desde su fichero de configuración `Homestead.yaml`. Por supuesto, puede sustituir `homestead.test` por cualquiera de los otros sitios configurados:

```
share homestead.test
```

Después de ejecutar el comando, verá aparecer una pantalla *Ngrok* conteniendo el registro de actividad y las URLs públicamente accesibles para el *sitio* compartido. Si quisiera especificar una region específica, subdominio o cualquier otra opción *Ngrok* en tiempo de ejecución, puede agregarlas a su comando `share`:

```
share homestead.test -region=eu -subdomain=laravel
```

Recuerde que *Vagrant* es inherentemente inseguro y está exponiendo su máquina virtual a Internet cuando ejecuta el comando `share`.

Múltiples versiones de PHP

nota esta característica es compatible sólo con *Nginx*.

Homestead 6 soporta múltiples versiones de *PHP* en la misma máquina virtual. Puede especificar qué versión de *PHP* se usa para un determinado *sitio* en el fichero `Homestead.yaml`. Las versiones de *PHP* disponibles son: "5.6", "7.0", "7.1" y "7.2":

```
sites:  
  
  - map: homestead.test  
    to: /home/vagrant/code/Laravel/public  
    php: "5.6"
```

Además, puede utilizar cualquiera de las versiones soportadas usando la línea de comandos (CLI):

```
php5.6 artisan list  
php7.0 artisan list  
php7.1 artisan list  
php7.2 artisan list
```

Interfaces de red

La propiedad `networks` de `Homestead.yaml` configura los interfaces de red para su entorno *Homestead*. Puede configurar tantas interfaces como sean necesarias:

```
networks:  
  
  - type: "private_network"  
    ip: "192.168.10.20"
```

Para activar una interfaz enlazada, configure una configuración de `bridge` y cambie el tipo red a `public_network`:

```
networks:  
  
  - type: "public_network"  
    ip: "192.168.10.20"  
    bridge: "en1: Wi-Fi (AirPort)"
```

Para activar [DHCP](#), sólo elimine la opción `ip` de su configuración:

```
networks:  
  
  - type: "public_network"  
    bridge: "en1: Wi-Fi (AirPort)"
```

Actualizando Homestead

Puede actualizar *Homestead* en dos simples pasos. Primero, debería de actualizar la caja *Vagrant* utilizando el comando `vagrant box update`:

```
vagrant box update
```

A continuación, necesita actualizar el código fuente de *Homestead*. Si clonó el repositorio, puede simplemente ejecutar `git pull origin master` en el mismo sitio donde clonó originalmente el repositorio.

Si tiene instalado *Homestead* a través de su fichero `composer.json` del proyecto, debe asegurarse que éste contiene `"laravel/homestead": "^6"`, y actualizar sus dependencias:

```
composer update
```

Versiones antiguas

Si necesita una versión antigua de PHP compruebe la documentación de multiples versiones PHP antes de intentar utilizar una versión antigua de *Homestead*.

Puede sobreescribir fácilmente la versión del *box* que utiliza *Homestead* agregando la siguiente línea al fichero `Homestead.yaml`:

```
version: 0.6.0
```

Un ejemplo:

```
box: laravel/homestead
version: 0.6.0
ip: "192.168.10.10"
memory: 2048
cpus: 4
provider: virtualbox
```

Cuando utiliza una versión antigua de la caja de *Homestead* necesita combinarla con una versión compatible del código fuente de *Homestead*. A continuación se muestra una tabla que muestra las versiones compatibles de la caja, qué versión del código fuente de *Homestead* usar y la versión de PHP proporcionada:

	Versiones Homestead	Versiones Box
PHP 7.0	3.1.0	0.6.0
PHP 7.1	4.0.0	1.0.0
PHP 7.1	5.0.0	2.0.0
PHP 7.1	6.0.0	3.0.0
PHP 7.2 RC3	6.4.0	4.0.0

Configuraciones específicas de proveedor

VirtualBox

Por defecto, *Homestead* establece la configuración `natdnshostresolver` a `on`. Esto permite a *Homestead* utilizar las configuración del DNS de su sistema operativo anfitrión. Si quisiera sobreescribir este comportamiento, agregue las siguientes líneas a su fichero

`Homestead.yaml` :

```
provider: virtualbox  
natdnshostresolver: off
```

Laravel Valet

Introducción

Valet o Homestead

Instalación

Actualizar

Servir sitios

El comando "park"

El comando "link"

Asegurar sitios con TLS

Compartir sitios

Drivers de Valet personalizados

Drivers locales

Otros comandos de Valet

Introducción

Valet es un entorno de desarrollo para minimalistas de Mac. No Vagrant, sin archivo `/etc/hosts`. Incluso se pueden compartir los sitios de forma pública utilizando tunes locales. *Sí, también nos gusta.*

Laravel Valet configura su Mac para ejecutar [Nginx](#) en segundo plano cuando la máquina arranca. Entonces, utilizando [DnsMasq](#), Valet crea un proxy de todas las peticiones sobre el dominio `*.dev` para apuntar a los sitios instalados en su máquina.

En otras palabras, un entorno de desarrollo tremadamente potente que utiliza únicamente 7 MB de RAM. Valet no es un reemplazo para Vagrant o Homestead, sino una alternativa si se prefieren servicios básicos, alta velocidad o se está trabajando en una máquina con RAM limitada.

Por defecto, Valet soporta, pero no limitado a:

- [Laravel](#)
- [Lumen](#)

- [Bedrock](#)
- [CakePHP 3](#)
- [Concrete5](#)
- [Contao](#)
- [Craft](#)
- [Drupal](#)
- [Jigsaw](#)
- [Joomla](#)
- [Katana](#)
- [Kirby](#)
- [Magento](#)
- [OctoberCMS](#)
- [Sculpin](#)
- [Slim](#)
- [Statamic](#)
- HTML estático
- [Symfony](#)
- [WordPress](#)
- [Zend](#)

Sin embargo, se puede extender Valet con [drivers propios](#).

Valet o Homestead

Como ya puede que sepa, Laravel ofrece [Homestead](#), otro entorno de desarrollo local. Homestead y Valet difieren en la audiencia y en el enfoque. Homestead incluye una máquina virtual Ubuntu completa con configuración Nginx automática. Homestead es una buena elección si se quiere tener una máquina Linux virtualizada como entorno de desarrollo o se está en Windows o Linux.

Valet únicamente soporta Mac y requiere instalar PHP y un servidor de base de datos directamente en la máquina local. Esto es muy sencillo utilizando [Homebrew](#) con comandos como `brew install php71` y `brew install mysql`. Valet provee de un rapidísimo entorno de

desarrollo con un consumo de recursos mínimo, por lo que es bueno para desarrolladores quienes únicamente necesitan PHP / MySQL y no necesitan un entorno virtualizado completo.

Ambos son buenas opciones para configurar el entorno de desarrollo local. Cual elegir ya es decisión personal de cada individuo o equipo dependiendo de sus necesidades.

Instalación

Valet requiere macOS y [Homebrew](#). Antes de instalar, hay que asegurarse de no tener otros programas como Apache o Nginx enlazados al puerto 80.

- Instalar o actualizar [Homebrew](#) a la última versión con `brew update` .
- Instalar PHP 7.1 utilizando Homebrew `brew install homebrew/php/php71` .
- Instalar Valet con Composer vía `composer global require laravel/valet` . Asegurarse que el directorio `~/.composer/vendor/bin` está en el "PATH" del sistema.
- Ejecutar el comando `valet install` . Esto configurará e instalará Valet y DnsMasq y registrará el *daemon* para que Valet se ejecute cuando el sistema arranca.

Una vez que Valet se ha instalado, pruebe a hacer *ping* a cualquier dominio `*.dev` desde la terminal `ping foobar.dev` . Si Valet está correctamente instalado se debería ver que este dominio responde `127.0.0.1` .

Valet arrancará su *daemon* cada vez que la máquina inicie. No es necesario ejecutar `valet start` o `valet install` de nuevo una vez que la instalación de Valet esté completa.

Utilizar Otro Dominio

Por defecto, Valet sirve los proyectos utilizando el TLD `.dev` . Si se desea utilizar otro dominio, se puede hacer utilizando el comando `valet domain nombre-tld` .

Por ejemplo, para utilizar el dominio `.app` en lugar de `.dev` , ejecutar `valet domain app` y Valet comenzará a servir los proyectos bajo `*.app` de forma automática.

Base de datos

Si se requiere de una base de datos, se puede instalar MySQL a través de la línea de comandos `brew install mysql` . Una vez que MySQL esté instalado, se puede iniciar con el comando `brew services start mysql` . Se puede entonces conectar a la base de datos `127.0.0.1` con usuario `root` y sin contraseña.

Actualizar

Se puede actualizar la instalación de Valet utilizando el comando `composer global update`. Después de actualizar, es una buena práctica ejecutar el comando `valet install` para que Valet pueda ejecutar actualizaciones adicionales a los archivos de configuración si fuera necesario.

Actualizar a Valet 2.0

Valet 2.0 reemplaza el servidor web subyacente de Caddy a Nginx. Antes de actualizar a esta versión se deben ejecutar los siguientes comandos para detener y desinstalar el <0>daemon Caddy existente:

```
valet stop  
valet uninstall
```

A continuación, se debe actualizar a la última versión de Valet. Dependiendo de como se haya instalado Valet, se realizará esta acción a través de Git o Composer. Si se instaló a través de Composer, se debe utilizar el siguiente comando para actualizar a la última versión:

```
composer global require laravel/valet
```

Una vez que se ha descargado Valet, se debe ejecutar el comando `install` :

```
valet install  
valet restart
```

Tras actualizar, puede que sea necesario re-vincular sus sitios.

Servir Sitios

Una vez que Valet se ha instalado, ya está listo para servir aplicaciones. Valet provee dos comandos para este propósito: `park` y `link`.

El Comando `park`

- Cree un nuevo directorio en su Mac como por ejemplo `mkdir ~/Sites`. A continuación, `cd ~/Sites` y ejecute `valet park`. Este comando incluirá el directorio actual en la lista que Valet debe buscar por sitios.

- A continuación, se puede crear una aplicación Laravel en este directorio `laravel new blog` .
- Acceder a `http://blog.dev` en el navegador.

Eso es todo. Cualquier proyecto que se cree en el directorio "parked" se servirá automáticamente utilizando la convención `http://folder-name.dev` .

El Comando `link`

El comando `link` se puede utilizar para servir sitios Laravel. Este comando resulta útil si se pretende servir un sitio en un directorio concreto pero no el directorio completo.

- Para utilizar el comando, hay que navegar hasta el directorio y ejecutar `valet link nombre` en la terminal. Valet creará un enlace simbólico en `~/.valet/Sites` que apuntará al directorio de la aplicación.
- Tras ejecutar el comando `link` , se puede acceder al sitio desde el navegador `http://nombre.dev` .

Para ver la lista de todos los directorios enlazados, se puede ejecutar el comando `valet links` . Se puede además utilizar el comando `valet unlink nombre` para eliminar un enlace simbólico.

Es posible utilizar `valet link` para servir el mismo proyecto desde varios (sub)dominios. Para añadir un subdominio u otro dominio al proyecto, simplemente ejecutar `valet link subdominio.nombre` desde el directorio del proyecto.

Asegurando Sitios con TLS

Por defecto, Valet sirve los sitios a través de HTTP. Sin embargo, si se pretende servir un sitio con encriptación TLS utilizando HTTP/2, utilizar el comando `secure` . Por ejemplo, si el sitio se sirve bajo el dominio `laraveles.dev` :

```
valet secure laravel
```

Para "insegurizar" un sitio y volver a servir el tráfico sobre HTTP, utilizar el comando `unsecure` . Así como el comando `secure` , acepta el nombre de `host` al que se aplica:

```
valet unsecure laravel
```

Compartir Sitios

Valet incluye además un comando para compartir los sitios locales con el mundo. No precisa de software adicional una vez que Valet está instalado.

Para compartir un sitio, simplemente hay que navegar al directorio en la terminal y ejecutar el comando `valet share`. Se generará una URL de acceso público y quedará automáticamente copiada al portapapeles lista para ser pegada en el navegador. Eso es todo.

Para dejar de compartir un sitio, `Control + C` cancelará el proceso.

`valet share` no soporta el compartir sitios que han sido asegurados utilizando el comando `valet secure`.

Drivers de Valet Personalizados

Se pueden desarrollar "drivers" para Valet propios para servir aplicaciones PHP que ejecutan otros frameworks o CMS que no se soporten de forma nativa. Al instalar Valet, se crea un directorio `~/.valet/Drivers` que contiene un archivo `SampleValetDriver.php`. Este archivo contiene un ejemplo de una implementación para demostrar como sería un driver personalizado. Un driver personalizado requiere tres métodos: `serves`, `isStaticFile` y `frontControllerPath`.

Todos estos métodos reciben `$sitePath`, `$siteName` y `$uri` como argumentos. `$sitePath` es el directorio completo del sitio que se está sirviendo en la máquina, por ejemplo `/Users/Laravel/Sites/my-project`. `$siteName` es el la porción del dominio "host" / "nombre del sitio" (`my-project`). `$uri` es la URI entrante (`/foo/bar`).

Una vez que se ha programado el driver, se almacenará en `~/.valet/Drivers` utilizando la convención de nombres `FrameworkValetDriver.php`. Por ejemplo, si se está desarrollando un driver para WordPress, el nombre del archivo debería ser `WordPressValetDriver.php`.

A continuación se muestra un ejemplo de cada método que el driver personalizado debe implementar.

El Método `serves`

El método `serves` retornará `true` si el driver debe ser responsable de la petición entrante. De otro modo, retornará `false`. Por lo tanto, en este método se debe intentar determinar si `$sitePath` contiene un proyecto del tipo que se está intentando servir.

Por ejemplo, imaginemos que se está programando `WordPressValetDriver`. El método debería ser algo así:

```
/**  
 * Determine if the driver serves the request.  
 *  
 * @param string $sitePath  
 * @param string $siteName  
 * @param string $uri  
 * @return bool  
 */  
  
public function serves($sitePath, $siteName, $uri)  
{  
    return is_dir($sitePath . '/wp-admin');  
}
```

El Método `isStaticFile`

El método `isStaticFile` determinará si la petición entrante busca un archivo estático como una imagen u hoja de estilos. Si el archivo es estático, se retornará la ruta completa del archivo en el disco. Si no es un archivo estático, este método debe retornar `false`:

```

/**
 * Determine if the incoming request is for a static file.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string|false
 */
public function isStaticFile($sitePath, $siteName, $uri)
{
    if (file_exists($staticFilePath = $sitePath . '/public/' . $uri)) {
        return $staticFilePath;
    }

    return false;
}

```

El método `isStaticFile` se llamará únicamente si el método `serves` retorna `true` para la petición entrante y si la URI no es `/`.

El Método `frontControllerPath`

El método `frontControllerPath` retornará la ruta completa del controlador principal de la aplicación, normalmente un archivo "index.php" o equivalente:

```

/**
 * Get the fully resolved path to the application's front controller.
 *
 * @param string $sitePath
 * @param string $siteName
 * @param string $uri
 * @return string
 */
public function frontControllerPath($sitePath, $siteName, $uri)
{
    return $sitePath . '/public/index.php';
}

```

Drivers Locales

Para definir un driver de Valet personalizado para una aplicación concreta, hay que crear un [LocalValetDriver.php](#) en el directorio raíz de la aplicación. El driver puede heredar de la clase [ValetDriver](#) o cualquier otro específico como [LaravelValetDriver](#):

```
class LocalValetDriver extends LaravelValetDriver
{
    /**
     * Determine if the driver serves the request.
     *
     * @param string $sitePath
     * @param string $siteName
     * @param string $uri
     * @return bool
     */
    public function serves($sitePath, $siteName, $uri)
    {
        return true;
    }

    /**
     * Get the fully resolved path to the application's front controller.
     *
     * @param string $sitePath
     * @param string $siteName
     * @param string $uri
     * @return string
     */
    public function frontControllerPath($sitePath, $siteName, $uri)
    {
        return $sitePath.'/public_html/index.php';
    }
}
```

Otros Comandos de Valet

Comando	Descripción
---------	-------------

Comando	Descripción
<code>valet forget</code>	Ejecutar este comando desde un directorio "parked" (aparcado) para eliminarlo de la lista de directorios.
<code>valet paths</code>	Ver todos los directorios aparcados.
<code>valet restart</code>	Reiniciar el <i>daemon</i> de Valet.
<code>valet start</code>	Iniciar el <i>daemon</i> de Valet.
<code>valet stop</code>	Detener el <i>daemon</i> de Valet.
<code>valet uninstall</code>	Desinstalar completamente el <i>daemon</i> de Valet.

Despliegue (Deployment)

Introducción

Configuración del Servidor

Nginx

Optimización

Optimizacion de Autoloader

Optimización de la Configuración

Optimización de las Rutas

Desplegar con Forge

Introducción

Cuando se tiene una aplicación de Laravel lista para el despliegue en producción, hay algunas cosas importantes que se pueden hacer para asegurarse de que la misma se ejecute tan eficientemente como sea posible. En este documento, se cubrirán algunos puntos de partida para asegurar que la aplicación se despliegue correctamente.

Configuración del Servidor

Nginx

Si se despliega la aplicación dentro de un servidor que esta corriendo Nginx, se puede usar el siguiente archivo de configuración como punto de partida para el servidor web. Lo más probable es que este archivo necesite ser personalizado dependiendo de la configuración del servidor. Si se necesita asistencia para el manejo de servidores, se puede considerar usar un servicio como [Laravel Forge](#):

```
server {

    listen 80;
    server_name example.com;
    root /example.com/public;

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-XSS-Protection "1; mode=block";
    add_header X-Content-Type-Options "nosniff";

    index index.html index.htm index.php;

    charset utf-8;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt { access_log off; log_not_found off; }

    error_page 404 /index.php;

    location ~ \.php$ {
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
        fastcgi_index index.php;
        include fastcgi_params;
    }

    location ~ /\.(?!well-known).* {
        deny all;
    }
}
```

Optimización

Optimizacion de Autoloader

Cuando se despliega a producción, hay que asegurarse de optimizar el *autoloader map* de Composer para que pueda encontrar rápidamente el archivo adecuado para cargar para una clase dada:

```
composer install --optimize-autoloader
```

Ademas de optimizar el *autoloader*, siempre se debe incluir el archivo `composer.lock` dentro del repositorio del proyecto. Las dependencias del proyecto se pueden instalar mucho mas rápido cuando el archivo `composer.lock` esta presente.

Optimización de la configuración

Cuando se despliega una aplicación en producción, hay que recordar ejecutar el comando de Artisan `config:cache` durante el proceso de despliegue:

```
php artisan config:cache
```

Este comando permite combinar todos los archivos de configuración de Laravel en uno solo archivo cacheado, lo que reduce en gran medida el número de peticiones que el framework debe realizar al sistema de archivos al cargar los valores de configuración.

Optimización de las Rutas

Cuando se despliega una aplicación en producción, hay que recordar ejecutar el comando de artisan `route:cache` durante el proceso de despliegue:

```
php artisan route:cache
```

Este comando reduce todos los registros de las rutas en una sola llamada dentro de un archivo en caché, mejorando el rendimiento del registro de rutas al momento de registrar cientos de las mismas.

Esta característica usa la serialización de PHP, y sólo puede almacenar en caché las rutas para aplicaciones que utilizan exclusivamente rutas basadas en controladores. PHP no puede serializar Clousures.

Desplegar con Forge

Si no se desea gestionar la configuración de un servidor o no se siente cómodo configurando todos los servicios necesarios para ejecutar una aplicación robusta de Laravel, se puede utilizar como una buena alternativa el servicio de [Laravel Forge](#).

Laravel Forge puede crear servidores en varios proveedores de infraestructura como lo son DigitalOcean, Linode, AWS y otros. Ademas, Forge instala y gestiona todas las herramientas necesarias para construir una aplicación de Laravel robusta, tal como Nginx, MySQL, Redis, Memcached, Beanstalk, y más.

Conceptos de arquitectura

Ciclo de Vida de la Petición

- # **Introducción**
 - # **Visión General del Ciclo de Vida**
 - # **Objetivo Service Providers**
-

Introducción

Cuando se usa cualquier herramienta en el "mundo real", te sientes más seguro si entiendes cómo funciona. El desarrollo de aplicaciones no es diferente. Cuando entiendes cómo funcionan las herramientas de desarrollo, te sientes más cómodo y seguro usándolas.

El objetivo de este documento es dar una buena visión general, y de alto nivel, de cómo funciona el framework Laravel. Al conocer mejor el framework en general, todo parece menos "mágico" y estarás más seguro construyendo aplicaciones. Si no entiendes a la primera todos los términos, ¡no te desanimes! Sólo trata de obtener una idea básica de qué está sucediendo, y el conocimiento crecerá a medida que se vayan explorando otras secciones de la documentación.

Visión General del Ciclo de Vida

Lo Primero

El punto de entrada de todas las peticiones a una aplicación Laravel es el archivo `public/index.php`. Todas las peticiones son dirigidas a este fichero por la configuración del servidor web (Apache / Nginx). El archivo `index.php` no contiene mucho código. Simplemente es el inicio de la carga del resto del framework.

El fichero `index.php` lee las definiciones de autoloader generadas por Composer, y entonces recupera una instancia de la aplicación Laravel desde el script `bootstrap/app.php`. La primera acción llevada a cabo por el propio Laravel es crear una instancia de la aplicación / [service container](#).

Núcleos HTTP / Console

Seguidamente, la petición entrante es enviada al núcleo HTTP o al núcleo de consola, dependiendo el tipo de petición que esté entrando a la aplicación. Estos dos núcleos sirven como ubicación central de todas las peticiones que pasen a través. Por ahora, centrémonos en el núcleo HTTP, localizado en el fichero `app/Http/Kernel.php`.

El núcleo HTTP hereda de la clase `Illuminate\Foundation\Http\Kernel`, la cual define un array de `bootstrappers` que se procesarán antes de que la petición sea ejecutada. La misión de estos "bootstrappers" es configurar el manejo de errores, el logging, [detectar el entorno de la aplicación](#) y realizar otras tareas que son necesarias antes de que se procese la petición actual.

El núcleo HTTP también define una lista de HTTP [middleware](#) que todas las peticiones deben pasar antes de que sean procesadas por la aplicación. Estos "middleware" manejan la lectura/escritura de la [sesión HTTP](#), detectan si la aplicación está en modo mantenimiento, [verifica el token CSRF](#), y más.

La firma del método `handle` del núcleo HTTP es bastante simple: recibe `Request` y devuelve `Response`. Pensar en el Núcleo como una gran caja negra que representa la totalidad de la aplicación. Aliméntalo con "HTTP requests" y devolverá "HTTP responses".

Proveedores de Servicio (Service Providers)

Una de las acciones más importantes de la inicialización del núcleo es la carga de los [service providers](#) de la aplicación. Todos los "service providers" de la aplicación están configurados dentro del archivo de configuración `config/app.php` en el array `providers`. Primero, para todos los providers se llamará al método `register`, una vez todos los providers hayan sido registrados se llamará al método `boot`.

Los "service providers" son responsables de la inicialización de los diversos componentes del framework, como la base de datos, colas, validaciones y los componentes de routing. Puesto que inicializan y configuran cada característica ofrecida por el framework, los "service providers" son el aspecto más importante del proceso de inicialización de Laravel.

Lanzar la Petición

Una vez la aplicación se ha inicializado y todos los "service providers" han sido registrados la [Petición](#) (Request) será traspasada al router para su lanzamiento. El router lanzará la request a una ruta o a un controlador, así como ejecutará cualquier middleware asociado a la ruta.

Objetivo Service Providers

Los Service Providers son verdaderamente la clave para inicializar una aplicación Laravel. La instancia de la aplicación está creada, los service providers están registrados y la petición es enviada a la aplicación inicializada. ¡Es así de simple!

Tener una firme convicción de como una aplicación Laravel se construye e inicializa a través de los service providers es muy valioso. Por supuesto, los service providers por defecto de la aplicación están almacenados en la carpeta `app/Providers` .

Por defecto, el `AppServiceProvider` tiene poco contenido. Este provider es un buen lugar donde añadir tus propias inicializaciones y los enlaces a service containers. Por supuesto que para grandes aplicaciones puede querer crear varios service providers, cada uno con una pequeña parte de la inicialización.

Service Container

Introducción

Binding (Enlazado)

- # Lo básico de un *Binding*
- # Enlazar interfaces a implementaciones
- # *Binding* contextual
- # Etiquetado

Resolución

- # El método *make*
- # Inyección automática

Eventos del Container

PSR-11

Introducción

El service container de Laravel es una potente herramienta para gestionar las dependencias de clases y llevar a cabo la inyección de estas. La inyección de dependencias es una frase moderna que básicamente significa: las dependencias de la clase se "inyectan" en la clase a través del constructor o, en algunos casos, métodos "setter".

Un simple ejemplo:

```
<?php

namespace App\Http\Controllers;

use App\User;
use App\Repositories\UserRepository;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function show($id)
    {
        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

En este ejemplo, `UserController` necesita obtener usuarios desde una fuente de datos. Por lo tanto, se **inyecta** un servicio que es capaz de ello. En este contexto, el `UserRepository` probablemente utilizará `Eloquent` para obtener la información del usuario de la base de datos. Sin embargo, puesto que el repositorio se inyecta, es muy sencillo intercambiarlo por otra implementación. Se podría además crear un *mock*, o una implementación ficticia de `UserRepository` para las pruebas de la aplicación.

Una comprensión profunda del service container de Laravel es básica para crear una aplicación grande y potente, así como para contribuir al core de Laravel.

Binding (Enlazado)

Lo básico de un *Binding*

Casi todos los *bindings* del service container se registrarán en `service providers`, por lo que la mayoría de estos ejemplos mostrarán el uso del *container* en ese contexto.

No hay necesidad de enlazar clases en el *container* si no dependen de una interfaz. El *container* no necesita conocer cómo crear esos objetos, puesto que puede resolverlos automáticamente utilizando el *reflection* nativo de PHP.

Bindings simples

En un *service provider* siempre se tendrá acceso al *container* a través de la propiedad `$this->app`. Se puede registrar un *binding* utilizando el método `bind`, pasando el nombre de la clase o interfaz que se desea registrar acompañado de un `Closure` que retornará una instancia de la clase:

```
$this->app->bind('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});
```

Tener en cuenta que se recibe el *container* en sí mismo como argumento. Se puede utilizar para resolver sub-dependencias del objeto que se está creando.

Enlazar un *singleton*

El método `singleton` enlaza una clase o interfaz al *container* y que únicamente debe ser resuelta una vez. Una vez que se ha resuelto un *binding singleton*, se devolverá la misma instancia del objeto a las subsecuentes llamadas en el *container*:

```
$this->app->singleton('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});
```

Enlazar instancias

Se puede enlazar la instancia de un objeto existente al container utilizando el método `instance`. La instancia proporcionada se retornará en las futuras llamadas:

```
$api = new HelpSpot\API(new HttpClient);

$this->app->instance('HelpSpot\API', $api);
```

Bindings primitivos

Se puede dar el caso de una clase que recibe otras clases inyectadas pero que además necesita de algún valor primitivo como un entero. Se puede utilizar un enlazado contextual para inyectar el valor que la clase necesite:

```
$this->app->when('App\Http\Controllers\UserController')
    ->needs('$variableName')
    ->give($value);
```

Enlazar interfaces a implementaciones

Una característica muy potente del service container es la capacidad de enlazar una interfaz a una implementación concreta. Por ejemplo, una interfaz `EventPusher` y una implementación `ReidsEventPusher`. Una vez que se ha programado la implementación `RedisEventPusher` de la interfaz, se puede registrar en el service container así:

```
$this->app->bind(
    'App\Contracts\EventPusher',
    'App\Services\RedisEventPusher'
);
```

Esto indica al *container* que debe injectar la clase `RedisEventPusher` cuando una clase necesite una implementación de `EventPusher`. Ahora se puede *type-hint* la interfaz `EventPusher` en un constructor, o cualquier otra localización donde las dependencias se injeten por el service container:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 *
 * @param EventPusher $pusher
 * @return void
 */
public function __construct(EventPusher $pusher)
{
    $this->pusher = $pusher;
}
```

Binding contextual

A veces se pueden tener dos clases que utilicen la misma interfaz, pero injectar diferentes implementaciones en cada clase. Por ejemplo, dos controladores pueden depender de diferentes implementaciones del [contrato Illuminate\Contracts\Filesystem\Filesystem](#). Laravel provee de una simple y fluida interfaz para definir este comportamiento:

```

use Illuminate\Support\Facades\Storage;
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;

$this->app->when(PhotoController::class)
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('local');
    });

$this->app->when(VideoController::class)
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('s3');
    });

```

Etiquetado

En ocasiones, puede ser necesario resolver toda una "categoría" concreta de bindings. Por ejemplo, quizás se está desarrollando un agregador de informes que recibe un array de diferentes implementaciones de la interfaz `Report`. Tras registrar las implementaciones `Report`, se les puede asignar una etiqueta utilizando el método `tag`:

```

$this->app->bind('SpeedReport', function () {
    //
});

$this->app->bind('MemoryReport', function () {
    //
});

$this->app->tag(['SpeedReport', 'MemoryReport'], 'reports');

```

Una vez que los servicios se han etiquetado, se pueden resolver utilizando el método `tagget`:

```

$this->app->bind('ReportAggregator', function ($app) {
    return new ReportAggregator($app->tagged('reports'));
});

```

Resolución

El método `make`

Se puede utilizar el método `make` para resolver la instancia de una clase desde el *container*. Este método aceptará el nombre de la clase o interfaz a resolver:

```
$api = $this->app->make('HelpSpot\API');
```

Se puede utilizar la función global `resolve` para acceder a la variable `$app` si no se tiene acceso a ella en alguna parte del código:

```
$api = resolve('HelpSpot\API');
```

Si alguna/s dependencias de la clase no se pueden resolver a través del *container*, se podrían injectar pasándolas como un *array* asociativo al método `makeWith`:

```
$api = $this->app->makeWith('HelpSpot\API', ['id' => 1]);
```

Inyección automática

Alternativamente y muy importante, se puede "type-hint" una dependencia en el constructor de cualquier clase que se resuelve en el *container*, incluyendo [controladores](#), [event listeners](#), [queue jobs](#), [middleware](#), y más. En la práctica, así es como la mayoría de objetos se resuelven por el *container*.

Por ejemplo, se puede type-hint un repositorio de la aplicación en el constructor de un controlador. El repositorio se resolverá automáticamente y se injectará en la clase:

```
<?php

namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the user with the given ID.
     *
     * @param int $id
     * @return Response
     */
    public function show($id)
    {
        //
    }
}
```

Eventos del Container

El *service container* dispara un evento cada vez que resuelve un objeto. Se puede capturar utilizando el método `resolving`:

```
$this->app->resolving(function ($object, $app) {  
    // Called when container resolves object of any type...  
});  
  
$this->app->resolving(HelpSpot\API::class, function ($api, $app) {  
    // Called when container resolves objects of type "HelpSpot\API"...  
});
```

Como se puede apreciar, el objeto a ser resuelto se pasará al *callback*, permitiendo establecer cualquier propiedad adicional al objeto antes de que se entregue.

PSR-11

El *service container* de Laravel implementa la interfaz [PSR-11](#). Por lo tanto, se puede *type-hint* (sugerencia de tipo) la interfaz PSR-11 del *container* para obtener una instancia del *container* de Laravel:

```
use Psr\Container\ContainerInterface;  
  
Route::get('/', function (ContainerInterface $container) {  
    $service = $container->get('Service');  
  
    //  
});
```

El método `get` lanzará una excepción si el identificador no se ha enlazado en el *container* explícitamente.

Service Providers

Introducción

Escribiendo Service Providers

- # El método *Register*

- # El método *Boot*

Registrando Providers

Providers diferidos

Introducción

Los *service providers* son la base de la inicialización de toda aplicación Laravel. Cualquier aplicación, así como los propios servicios de Laravel, se arrancan a través de *service providers*.

Pero, ¿qué quiere decir "inicializar/arrancar"? En general, se refiere a **registrar** cosas, incluyendo el registro de *bindings* (o enlaces) en el *service container*, *event listeners*, *middleware* e incluso rutas. Los *service providers* son el lugar central donde configurar la aplicación.

Si se abre el archivo `config/app.php` incluido con Laravel, se podrá observar el *array providers*. Son todas las clases de *service providers* que serán cargadas por la aplicación. Por supuesto, muchos de estos de estos *providers* son "diferidos", por lo que no se cargarán en cada petición, sino cuando los servicios que proporcionan sean realmente necesarios.

En esta visión general se aprenderá cómo escribir *service providers* propios y registrarlos en una aplicación Laravel.

Escribiendo Service Providers

Todos los *service providers* heredan de la clase `Illuminate\Support\ServiceProvider`. La mayoría de ellos contienen un método `register` y otro `boot`. En el método `register`, únicamente se deben **añadir *bindings* (enlazar cosas)** al *service container*. Nunca debe

intentar registrar ningún *event listeners*, rutas, o cualquier otra funcionalidad dentro del método `register`.

Se puede generar un nuevo *provider* a través del *Artisan CLI* con el comando `make:provider`:

```
php artisan make:provider RiakServiceProvider
```

El método *Register*

Como se ha comentado anteriormente, dentro del método `register`, únicamente se deben enlazar cosas dentro del service container. Nunca debe intentar registrar ningún escuchador de eventos, rutas, o cualquier otra funcionalidad dentro del método `register`. Por otra parte, en ocasiones se puede usar un servicio proporcionado por un *service provider* que no haya sido cargado todavía.

A continuación se muestra un ejemplo de un *service provider* básico. Dentro de cualquiera de los métodos de un *service provider*, siempre está disponible la propiedad `$app` que proporciona acceso al *service container*:

```
<?php

namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

Este *service provider* únicamente define el método `register` y lo utiliza para definir una implementación de `Riak\Connection` en el *service container*. Si no se entiende cómo funciona el *service container*, se puede consultar su [documentación](#).

El método *Boot*

Entonces, ¿qué ocurre si hay que registrar un *view composer* en un *service provider*? Suele hacerse dentro del método `boot`. **A este método se le llama después de que todos los demás *service providers* han sido registrados**, por tanto se tiene acceso a todos los demás servicios que hayan sido registrados por el framework:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        view()->composer('view', function () {
            //
        });
    }
}
```

El Método *Boot*, inyección de dependencias

Se pueden añadir dependencias para el método `boot` del *service provider*. El [service container](#) inyectará automáticamente cualquier dependencia que se necesite:

```
use Illuminate\Contracts\Routing\ResponseFactory;

public function boot(ResponseFactory $response)
{
    $response->macro('caps', function ($value) {
        //
    });
}
```

Registrando Providers

Todos los service providers están registrados en el archivo de configuración `config/app.php`. Este archivo contiene un *array* `providers` donde se pueden enumerar los nombres de los *service providers*. Por defecto se incluye un conjunto de *service providers* del núcleo de Laravel. Estos *providers* inicializan los componentes base de Laravel como el *mailer*, *queue*, *cache* y otros.

Para registrar un providers, simplemente hay que añadirlo al array:

```
'providers' => [
    // Other Service Providers

    App\Providers\ComposerServiceProvider::class,
],
```

Providers diferidos

Si el proveedor **sólo** registra enlaces en el service container, se puede diferir este registro hasta que uno de estos enlaces se necesite realmente. Aplazando la carga de estos providers se incrementará el rendimiento de la aplicación, puesto que no se cargará desde el sistema de archivo en cada petición.

Laravel comienza y almacena una lista de todos los services proporcionados por los service providers diferidos junto con el nombre de su clase. Entonces, sólo cuando se intente resolver uno de estos servicios Laravel cargará el service provider.

Para aplazar la carga de un provider, establecer la propiedad `defer` a `true` y definir el método `provides`. El método `provides` debe retornar los *bindings* del *service container* registrados:

```
<?php

namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Indicates if loading of the provider is deferred.
     *
     * @var bool
     */
    protected $defer = true;

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * Get the services provided by the provider.
     *
     * @return array
     */
    public function provides()
    {
        return [Connection::class];
    }
}
```


Facades

- # **Introducción**
 - # **Cuando utilizar facades**
 - # Facades vs inyección de dependencias
 - # Facades vs helpers
 - # **Cómo funcionan las facades**
 - # **Facades en tiempo real**
 - # **Referencia de facades**
-

Introducción

Las *facades* proveen de una interfaz "estática" a las clases disponibles en el [service container](#). Laravel incluye varias *facades* que proveen acceso a casi todas las características del framework. Las *facades* de Laravel sirven como "proxies estáticos" a las clases subyacentes del *service container*, proporcionando una sintaxis expresiva mientras que se proporciona más *testabilidad* y flexibilidad que con métodos estáticos tradicionales.

Todas las *facades* de Laravel se definen en el namespace `Illuminate\Support\Facades` . Se puede acceder a una *facade* así:

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

A través de la documentación de Laravel, muchos de los ejemplos utilizarán *facades* para demostrar varias características del framework.

Cuando utilizar facades

Las *facades* tienen muchas ventajas. Proporcionan una sintaxis fácil de recordar que permite utilizar las características de Laravel sin tener que recordar nombres de clases largos para injectar o configurados manualmente. Además, por su único uso de métodos de PHP dinámicos, son fáciles de testear.

Sin embargo, se debe tener cuidado al usar facades. El principal peligro de las *facades* es el deslizamiento del alcance de clase. Como las *facades* son tan fáciles de usar y no requieren inyección, puede ser fácil dejar que las clases sigan creciendo y usar muchas fachadas en una sola clase. Al usar la inyección de dependencia, este potencial se ve mitigado por los comentarios visuales que un gran constructor le indica que su clase está creciendo demasiado. Por lo tanto, al usar facades, se debe prestar especial atención al tamaño de la clase para que su alcance de su responsabilidad se mantenga estrecho.

Al crear un paquete de terceros que interactúa con Laravel, es mejor injectar Laravel contracts en lugar de utilizar de *facades*. Dado que los paquetes se construyen fuera de Laravel, no se tendrá acceso a los helpers de pruebas de los *facades* de Laravel.

Facades vs inyección de dependencias

Uno de los principales beneficios de la inyección de dependencias es la capacidad de intercambiar implementaciones de la clase inyectada. Esto es útil durante las pruebas ya que puede injectar un *mock* o un *stub* y verificar que se han llamado varios métodos en el *stub*.

Normalmente, no sería posible hacer un *mock* o *stub* de un método de clase verdaderamente estático. Sin embargo, dado que las *facades* usan métodos dinámicos para realizar llamadas de métodos proxy a objetos resueltos desde el *service container*, en realidad podemos probar *facades* de la misma manera que probaríamos una instancia de clase inyectada. Por ejemplo, dada la siguiente ruta:

```
use Illuminate\Support\Facades\Cache;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

Podemos escribir la siguiente prueba para verificar que el método `Cache::get` ha sido llamado con el argumento esperado:

```
use Illuminate\Support\Facades\Cache;

/**
 * A basic functional test example.
 *
 * @return void
 */
public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $this->visit('/cache')
        ->see('value');
}
```

Facades vs. Funciones Helper

Además de las *facades*, Laravel incluye gran variedad de funciones de ayuda o "helper" que pueden realizar tareas comunes tales como generar vistas, disparar eventos, despachar trabajos o enviar respuestas HTTP. Muchas de esas funciones de ayuda realizan una función equivalente a la de una *facade* correspondiente. Por ejemplo, esta llamada *facade* y esta función helper son equivalentes:

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

return view('profile');
```

No hay absolutamente ninguna diferencia práctica entre *facades* y funciones *helper*. Cuando se usan funciones *helper*, puedes hacer pruebas exactamente igual que si usaras la *facade* correspondiente. Por ejemplo, dada la siguiente ruta:

```
Route::get('/cache', function () {
    return cache('key');
});
```

En el fondo, el helper `cache` llama al método `get` en la clase subyacente a la `Facade Cache`. Entonces, aunque estamos usando la función auxiliar, podemos escribir la siguiente prueba para verificar que el método fue llamado con el argumento que esperábamos:

```
use Illuminate\Support\Facades\Cache;

/**
 * A basic functional test example.
 *
 * @return void
 */

public function testBasicExample()
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $this->visit('/cache')
        ->see('value');
}
```

Cómo funcionan las *facades*

En el contexto de una aplicación Laravel, una *Facade* es una clase que provee acceso a un objeto del *container*. La maquinaria que realiza este trabajo es la clase `Facade`. Tanto las *facades* de Laravel como cualquier facade personalizado, heredarán de la clase base `Illuminate\Support\Facades\Facade`.

La clase base `Facade` hace uso del método mágico `__callStatic()` para canalizar las llamadas de la *Facade* al objeto resuelto. En el siguiente ejemplo, se realiza una llamada al sistema de caché de Laravel. Al echar un vistazo a este código, se podría suponer que se llama al método estático `get` en la clase `Cache`:

```

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = Cache::get('user:'.$id);

        return view('profile', ['user' => $user]);
    }
}

```

Tener en cuenta que al comienzo del archivo se está "importando" la *Facade Cache*. Esta *Facade* sirve como un proxy para acceder a la implementación subyacente de la interfaz `Illuminate\Contracts\Cache\Factory`. Cualquier llamada realizada a través de la *Facade* se pasará a la instancia subyacente del servicio de caché de Laravel.

Si se observa la clase `Illuminate\Support\Facades\Cache`, no existe el método `get`:

```

class Cache extends Facade
{
    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'cache'; }
}

```

En su lugar, la *facade* `Cache` hereda de la clase base `Facade` y define el método `getFacadeAccessor()`. Recordar, la responsabilidad de este método es retornar el nombre de un *binding* en el *service container*. Cuando se hace uso de cualquier método estático de la *facade* `Cache`, Laravel resolverá el *binding* `cache` del *service container* y ejecutará el método solicitado (en este caso, `get`) en el objeto resuelto.

Facades en tiempo real

Usando *facades* en tiempo real, se puede tratar cualquier clase en su aplicación como si fuera una *facade*. Para ilustrar cómo se puede usar esto, examinemos una alternativa. Por ejemplo, supongamos que nuestro modelo `Podcast` tiene un método `publish`. Sin embargo, para publicar el podcast, necesitamos injectar una instancia de `Publisher`:

```
<?php

namespace App;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     *
     * @param Publisher $publisher
     * @return void
     */
    public function publish(Publisher $publisher)
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
    }
}
```

Inyectar una implementación de `publisher` en el método nos permite probar fácilmente el método de forma aislada ya que podemos simular el editor injectado. Sin embargo, nos exige pasar siempre una instancia de editor cada vez que llamemos al método `publish`. Al utilizar

facades en tiempo real, podemos mantener la misma capacidad de prueba sin tener que pasar explícitamente una instancia de `Publisher`. Para generar una fachada en tiempo real, simplemente se coloca `Facades` como prefijo en el namespace de la clase importada:

```
<?php

namespace App;

use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Publish the podcast.
     *
     * @return void
     */
    public function publish()
    {
        $this->update(['publishing' => now()]);

        Publisher::publish($this);
    }
}
```

Cuando se usa un *facade* en tiempo real, la implementación de *publisher* se resolverá fuera del *service container* utilizando la parte de la interfaz o nombre de clase que aparece después del prefijo `Facades`. Al testear, podemos utilizar los helpers de prueba de *facades* incorporados de Laravel para simular esta llamada a método:

```

<?php

namespace Tests\Feature;

use App\Podcast;
use Tests\TestCase;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;

class PodcastTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A test example.
     *
     * @return void
     */
    public function test_podcast_can_be_published()
    {
        $podcast = factory(Podcast::class)->create();

        Publisher::shouldReceive('publish')->once()->with($podcast);

        $podcast->publish();
    }
}

```

Referencia de Clases Facade

A continuación se detalla una lista de *facades* y sus clases subyacentes. Resultará útil para acceder de forma rápida a la documentación API para la raíz de una *Facade* concreta. La clave [service container binding](#) se incluye cuando aplica.

Facade	Clase	Service Container Binding
App	<u>Illuminate\Foundation\Application</u>	<u>app</u>

Facade	Clase	Service Container Binding
Artisan	Illuminate\Contracts\Console\Kernel	artisan
Auth	Illuminate\Auth\AuthManager	auth
Auth (Instance)	Illuminate\Contracts\Auth\Guard	auth.driver
Blade	Illuminate\View\Compilers\BladeCompiler	blade.compiler
Broadcast	Illuminate\Contracts\Broadcasting\Factory	
Broadcast (Instance)	Illuminate\Contracts\Broadcasting\Broadcaster	
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\CacheManager	cache
Cache (Instance)	Illuminate\Cache\Repository	cache.store
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie
Crypt	Illuminate\Encryption\Encrypter	encrypter
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	db.connection
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Gate	Illuminate\Contracts\Auth\Access\Gate	

Facade	Clase	Service Container Binding
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\Writer	log
Mail	Illuminate\Mail\Mailer	mailer
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords\PasswordBrokerManager	auth.password
Password (Instance)	Illuminate\Auth\Passwords\PasswordBroker	auth.password.broker
Queue	Illuminate\Queue\QueueManager	queue
Queue (Instance)	Illuminate\Contracts\Queue\Queue	queue.connection
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	redirect
Redis	Illuminate\Redis\RedisManager	redis
Redis (Instance)	Illuminate\Redis\Connections\Connection	redis.connection
Request	Illuminate\Http\Request	request
Response	Illuminate\Contracts\Routing\ResponseFactory	
Response (Instance)	Illuminate\Http\Response	

Facade	Clase	Service Container Binding
Route	Illuminate\Routing\Router	router
Schema	Illuminate\Database\Schema\Builder	
Session	Illuminate\Session\SessionManager	session
Session (Instance)	Illuminate\Session\Store	session.store
Storage	Illuminate\Filesystem\FilesystemManager	filesystem
Storage (Instance)	Illuminate\Contracts\Filesystem\Filesystem	filesystem.disk
URL	Illuminate\Routing\UrlGenerator	url
Validator	Illuminate\Validation\Factory	validator
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	view
View (Instance)	Illuminate\View\View	

Contratos

Introducción

Contratos vs *Facades*

Cuándo usar contratos

Acoplamiento débil

Simplicidad

Cómo utilizar contratos

Referencia de contratos

Introducción

Los contratos de Laravel son un conjunto de interfaces que definen los servicios principales del framework. Por ejemplo, el contrato `Illuminate\Contracts\Queue\Queue` define los métodos necesarios para la encolar trabajos, mientras que `Illuminate\Contracts\Mail\Mailer` define los métodos para enviar e-mails.

Cada contrato tiene una implementación correspondiente en el framework. Por ejemplo, Laravel provee una implementación de colas con una variedad de drivers, y una implementación de envío de correos que se apoya en [SwiftMailer](#).

Todos los contratos de Laravel se encuentran en [su propio repositorio de GitHub](#). Esto proporciona un punto de referencia sobre todos los contratos disponibles, así como un paquete desacoplado que puede ser utilizado por desarrolladores de paquetes.

Contratos vs *Facades*

Las [facades](#) de Laravel y los *helper* son una forma sencilla de utilizar los servicios de Laravel sin tener que hacer *type-hint* y resolver contratos fuera del *service container*. En la mayoría de los casos, cada *Facade* tiene un contrato equivalente.

A diferencia de las *facades*, que no exigen hacer un *require* en el constructor de la clase, los contratos permiten definir dependencias explícitas para las clases. Algunos desarrolladores prefieren definir explícitamente sus dependencias y por lo tanto prefieren el uso de los contratos, mientras otros desarrolladores prefieren la conveniencia de las *facades*.

La mayoría de las aplicaciones estarán bien tanto si se prefieren *facades* como contratos. Sin embargo, si se está desarrollando un paquete, se debería considerar con fuerza el uso de contratos ya que serán más fáciles de "testear" en un contexto de paquete.

Cuándo usar contratos

Como se discutió anteriormente, gran parte de la decisión de usar contratos o *facades* se reduce al gusto personal y a las preferencias del equipo de desarrollo. Tanto los contratos como las *facades* pueden usarse para crear aplicaciones de Laravel robustas y bien probadas. Mientras se mantengan en foco las responsabilidades de la clase, se notarán muy pocas diferencias entre el uso de *facades* y contratos.

Sin embargo, podrían existir muchas preguntas en relación a los contratos. Por ejemplo, ¿por qué usar interfaces? ¿No lo hacen todo más complicado? Simplificaremos las razones de uso de interfaces a los siguientes títulos: acoplamiento débil y simplicidad.

Acoplamiento débil

Primero, hay que revisar un código que está fuertemente acoplado a una implementación de caché. Tengamos en cuenta lo siguiente:

```

<?php

namespace App\Orders;

class Repository
{
    /**
     * The cache instance.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param \SomePackage\Cache\Memcached $cache
     * @return void
     */
    public function __construct(\SomePackage\Cache\Memcached $cache)
    {
        $this->cache = $cache;
    }

    /**
     * Retrieve an Order by ID.
     *
     * @param int $id
     * @return Order
     */
    public function find($id)
    {
        if ($this->cache->has($id)) {
            //
        }
    }
}

```

En esta clase, el código está fuertemente acoplado a la implementación de caché. Lo está porque depende directamente de una clase *Cache* concreta de un paquete. Si el API del paquete cambia, el código deberá cambiar también.

Además, si se quiere reemplazar la tecnología de caché subyacente (Memcached) con otra tecnología (Redis), habrá de nuevo que modificar el repositorio. El repositorio no debería conocer mucho sobre quien provee los datos o sobre cómo se proveen.

En lugar de este enfoque, se puede mejorar nuestro código dependiendo de una simple e independiente interfaz:

```
<?php

namespace App\Orders;

use Illuminate\Contracts\Cache\Repository as Cache;

class Repository
{
    /**
     * The cache instance.
     */
    protected $cache;

    /**
     * Create a new repository instance.
     *
     * @param Cache $cache
     * @return void
     */
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }
}
```

Ahora el código no está acoplado a ningún proveedor concreto, o incluso Laravel. Puesto que el paquete de contratos no contiene implementaciones ni dependencias, fácilmente se pueden escribir implementaciones alternativas a cualquier contrato, permitiendo reemplazar la implementación de caché sin modificar ningún código que ya se esté consumiendo.

Simplicidad

Cuando todos los servicios de Laravel están claramente definidos por interfaces, es muy fácil determinar la funcionalidad que ofrece un servicio. **Los contratos sirven como documentación de las características del framework.**

Además, cuando se depende de interfaces simples, el código suele ser más fácil de comprender y mantener. En lugar de rastrear los métodos disponibles en una clase grande y complicada, nos podemos referir a una simple y limpia interfaz.

Cómo utilizar contratos

Así que, ¿cómo se obtiene una implementación de un contrato? En realidad es bastante simple.

Muchos tipos de clases en Laravel se resuelven a través del [service container](#), incluyendo controladores, capturadores de eventos, *middleware*, colas de trabajo e incluso *Closures* de ruta. Por lo que, para obtener la implementación de un contrato, simplemente con hacer "type-hint" de la interfaz en el constructor de la clase a resolver.

Por ejemplo, veamos este capturador de eventos:

```
<?php

namespace App\Listeners;

use App\User;
use App\Events\OrderWasPlaced;
use Illuminate\Contracts\Redis\Database;

class CacheOrderInformation
{
    /**
     * The Redis database implementation.
     */
    protected $redis;

    /**
     * Create a new event handler instance.
     *
     * @param Database $redis
     * @return void
     */
    public function __construct(Database $redis)
    {
        $this->redis = $redis;
    }

    /**
     * Handle the event.
     *
     * @param OrderWasPlaced $event
     * @return void
     */
    public function handle(OrderWasPlaced $event)
    {
        //
    }
}
```

Cuando se resuelve el capturador, el *service container* leerá la firma del constructor de la clase e inyectará los valores apropiados. Para saber más acerca de registrar cosas en el *service container*, revise [su documentación](#).

Referencia de contratos

Esta tabla provee una referencia rápida a todos los contratos de Laravel y sus *facades* equivalentes:

Contrato	Facade referencia
Illuminate\Contracts\Auth\Access\Authorizable	
Illuminate\Contracts\Auth\Access\Gate	Gate
Illuminate\Contracts\Auth\Authenticatable	
Illuminate\Contracts\Auth\CanResetPassword	
Illuminate\Contracts\Auth\Factory	Auth
Illuminate\Contracts\Auth\Guard	Auth::guard()
Illuminate\Contracts\Auth>PasswordBroker	Password::broker()
Illuminate\Contracts\Auth>PasswordBrokerFactory	Password
Illuminate\Contracts\Auth\StatefulGuard	
Illuminate\Contracts\Auth\SupportsBasicAuth	
Illuminate\Contracts\Auth\UserProvider	
Illuminate\Contracts\Bus\Dispatcher	Bus
Illuminate\Contracts\Bus\QueueingDispatcher	Bus::dispatchToQueue()
Illuminate\Contracts\Broadcasting\Factory	Broadcast
Illuminate\Contracts\Broadcasting\Broadcaster	Broadcast::connection()
Illuminate\Contracts\Broadcasting\ShouldBroadcast	
Illuminate\Contracts\Broadcasting\ShouldBroadcastNow	
Illuminate\Contracts\Cache\Factory	Cache

Contrato	Facade referencia
Illuminate\Contracts\Cache\Lock	
Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	<code>Cache::driver()</code>
Illuminate\Contracts\Cache\Store	
Illuminate\Contracts\Config\Repository	<code>Config</code>
Illuminate\Contracts\Console\Application	
Illuminate\Contracts\Console\Kernel	<code>Artisan</code>
Illuminate\Contracts\Container\Container	<code>App</code>
Illuminate\Contracts\Cookie\Factory	<code>Cookie</code>
Illuminate\Contracts\Cookie\QueueingFactory	<code>Cookie::queue()</code>
Illuminate\Contracts\Database\ModelIdentifier	
Illuminate\Contracts\Debug\ExceptionHandler	
Illuminate\Contracts\Encryption\Encrypter	<code>Crypt</code>
Illuminate\Contracts\Events\Dispatcher	<code>Event</code>
Illuminate\Contracts\Filesystem\Cloud	<code>Storage::cloud()</code>
Illuminate\Contracts\Filesystem\Factory	<code>Storage</code>
Illuminate\Contracts\Filesystem\Filesystem	<code>Storage::disk()</code>
Illuminate\Contracts\Foundation\Application	<code>App</code>
Illuminate\Contracts\Hashing\Hasher	<code>Hash</code>
Illuminate\Contracts\Http\Kernel	

Contrato	Facade referencia
Illuminate\Contracts\Logging\Log	Log
Illuminate\Contracts\Mail\MailQueue	Mail::queue()
Illuminate\Contracts\Mail\Mailable	
Illuminate\Contracts\Mail\Mailer	Mail
Illuminate\Contracts\Notifications\Dispatcher	Notification
Illuminate\Contracts\Notifications\Factory	Notification
Illuminate\Contracts\Pagination\LengthAwarePaginator	
Illuminate\Contracts\Pagination\Paginator	
Illuminate\Contracts\Pipeline\Hub	
Illuminate\Contracts\Pipeline\Pipeline	
Illuminate\Contracts\Queue\EntityResolver	
Illuminate\Contracts\Queue\Factory	Queue
Illuminate\Contracts\Queue\Job	
Illuminate\Contracts\Queue\Monitor	Queue
Illuminate\Contracts\Queue\Queue	Queue::connection()
Illuminate\Contracts\Queue\QueueableCollection	
Illuminate\Contracts\Queue\QueueableEntity	
Illuminate\Contracts\Queue\ShouldQueue	
Illuminate\Contracts\Redis\Factory	Redis
Illuminate\Contracts\Routing\BindingRegistrar	Route

Contrato	Facade referencia
Illuminate\Contracts\Routing\Registrar	Route
Illuminate\Contracts\Routing\ResponseFactory	Response
Illuminate\Contracts\Routing\UrlGenerator	URL
Illuminate\Contracts\Routing\UrlRoutable	
Illuminate\Contracts\Session\Session	Session::driver()
Illuminate\Contracts\Support\Arrayable	
Illuminate\Contracts\Support\Htmlable	
Illuminate\Contracts\Support\Jsonable	
Illuminate\Contracts\Support\MessageBag	
Illuminate\Contracts\Support\MessageProvider	
Illuminate\Contracts\Support\Renderable	
Illuminate\Contracts\Support\Responsable	
Illuminate\Contracts\Translation\Loader	
Illuminate\Contracts\Translation\Translator	Lang
Illuminate\Contracts\Validation\Factory	Validator
Illuminate\Contracts\Validation\ImplicitRule	
Illuminate\Contracts\Validation\Rule	
Illuminate\Contracts\Validation\ValidatesWhenResolved	
Illuminate\Contracts\Validation\Validator	Validator::make()
Illuminate\Contracts\View\Engine	

Contrato	<i>Facade referencia</i>
<u>Illuminate\Contracts\View\Factory</u>	<code>View</code>
<u>Illuminate\Contracts\View\View</u>	<code>View::make()</code>

Lo básico

Rutas

Rutas básicas

- # Redirección de rutas
- # Rutas con vistas

Rutas con parámetros

- # Parámetros requeridos
- # Parámetros opcionales
- # Expresiones regulares

Nombres de rutas

Grupos de rutas

- # Middleware
- # Namespaces
- # Rutas de sub-dominios
- # Prefijos de rutas

Enlazar modelos a rutas

- # Enlazado implícito
- # Enlazado explícito

Falsear el método del formulario

Acceder a la ruta actual

Rutas Básicas

La ruta más básica en Laravel únicamente acepta la URI y un [Closure](#), ofreciendo una forma muy sencilla y expresiva de definir rutas:

```
Route::get('foo', function () {  
    return 'Hello World';  
});
```

El archivo de rutas por defecto

Todas las rutas de Laravel se definen en los archivos que se encuentran en la carpeta `routes` . El framework carga estos archivos de forma automática. El archivo `routes/web.php` define las rutas para la interfaz web. A estas rutas se les asigna el grupo de *middleware* `web` , el cual proporciona algunas características como el estado de la sesión y la protección CSRF. Las rutas en `routes/api.php` no tienen estado y se les asigna el grupo de *middleware* `api` .

Para la mayoría de aplicaciones, se comenzará definiendo las rutas en el archivo `routes/web.php` . Se puede acceder a las rutas definidas en `routes/web.php` simplemente escribiendo la URI definida en el navegador. Por ejemplo, se puede acceder a la siguiente ruta simplemente accediendo a `http://tu-app.dev/user` desde el navegador:

```
Route::get('/user', 'UserController@index');
```

El `RouteServiceProvider` anidará las rutas definidas en `routes/api.php` bajo un grupo de rutas. Este grupo añadirá el URI `/api` como prefijo de forma automática, por lo que no es necesario definirlo en cada una de las rutas del archivo. Se puede modificar el prefijo y otras opciones del grupo modificando la clase `RouteServiceProvider` .

Métodos de rutas disponibles

El `router` permite registrar rutas que respondan a cualquier verbo HTTP:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

En ocasiones puede ser necesario una ruta que responda a varios verbos HTTP. Puede hacerse utilizando el método `match` . O incluso se puede registrar una ruta que responda a todos los verbos HTTP utilizando el método `any` :

```
Route::match(['get', 'post'], '/', function () {
    //
});

Route::any('foo', function () {
    //
});
```

Protección CSRF

Cualquier formulario HTML que apunte a una ruta `POST`, `PUT` o `DELETE` definida en el archivo de rutas `web` debe incluir un campo con un token CSRF. De lo contrario, la petición se rechazará. Se puede averiguar más sobre protección CSRF en la [documentación CSRF](#):

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

Redireccionar rutas

Para definir una ruta que redirecciona a otra URI, se puede utilizar el método `Route::redirect`. Este método evita tener que definir una ruta completa o un controlador para gestionar una simple redirección:

```
Route::redirect('/here', '/there', 301);
```

Rutas con vistas

Si únicamente se necesita devolver una vista desde una ruta, se puede utilizar el método `Route::view`. Al igual que el método `redirect`, este método es como un acceso directo para no tener que definir la ruta completa o un controlador. El método `view` acepta una URI como primer parámetro y un nombre de vista como segundo. Además, se le puede pasar un array de datos a la vista como tercer parámetro opcional:

```
Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

Rutas con parámetros

Parámetros requeridos

En muchas ocasiones es necesario capturar segmentos de la URI dentro de una ruta. Por ejemplo, obtener el id de usuario en la URL. Esto se puede hacer mediante la definición de parámetros de ruta:

```
Route::get('user/{id}', function ($id) {
    return 'User ' . $id;
});
```

Es posible definir tantos parámetros como se requieran en la ruta:

```
Route::get('posts/{post}/comments/{comment}', function ($postId, $commentId) {
    //
});
```

Los parámetros de ruta se definen siempre dentro de llaves {} y deben contener únicamente caracteres alfabéticos y nunca contener - (guiones). Instead of using the - character, use an underscore (_). Los parámetros de rutas se injetan directamente en los callback/controladores en orden – los nombres de los argumentos no afectan.

Parámetros opcionales

A veces es necesario especificar un parámetro, pero la presencia de este parámetro es opcional. Se puede definir simplemente añadiendo el símbolo ? detrás del nombre del parámetro. Asegúrese de dar a esa variable un valor por defecto en su callback/controlador:

```
Route::get('user/{name?}', function ($name = null) {
    return $name;
});

Route::get('user/{name?}', function ($name = 'John') {
    return $name;
});
```

Expresiones regulares

Es posible limitar el formato de los elementos dentro de los parámetros de una ruta usando el método `where` en una instancia de `Route`. El método `where` acepta el nombre del parámetro y la expresión regular que define como se debe limitar el parámetro:

```

Route::get('user/{name}', function ($name) {
    //
})->where('name', '[A-Za-z]+');

Route::get('user/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);

```

Restricciones globales

Si se desea que un parámetro de ruta siempre este limitado por una expresión regular definida, se puede hacer utilizando el método `pattern`. Estos patrones se deben definir en el método `boot` dentro del `RouteServiceProvider`:

```

/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');

    parent::boot();
}

```

Una vez que el patrón se haya sido definido, se aplicará automáticamente a todas las rutas que utilicen ese nombre como parámetro:

```

Route::get('user/{id}', function ($id) {
    // Only executed if {id} is numeric...
});

```

Nombres de rutas

Las rutas con nombre permiten la generación de URL o redirecciones para rutas específicas. Se puede especificar el nombre de una ruta encadenando el método `name` a la definición de la ruta:

```
Route::get('user/profile', function () {
    //
})->name('profile');
```

También se pueden especificar nombres de rutas para acciones de controladores:

```
Route::get('user/profile', 'UserController@showProfile')->name('profile');
```

Generar URLs con nombres de ruta

Una vez que se ha asignado un nombre a una ruta concreta, se puede utilizar ese nombre para generar URLs o redirecciones a través del helper global `route`:

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');
```

Si la ruta define parámetros, se pueden pasar como segundo argumento a la función `route`. Los parámetros se insertarán automáticamente en la posición correcta de la URL:

```
Route::get('user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1]);
```

Inspeccionar la ruta actual

Para determinar si la petición actual coincide con el nombre de alguna ruta, se puede utilizar el método `named` de una instancia de `Route`. Por ejemplo, se puede comprobar el nombre de la ruta actual desde un *middleware* de ruta:

```

/**
 * Handle an incoming request.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Closure $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {
        //
    }

    return $next($request);
}

```

Grupos de rutas

Los Grupos de Rutas permiten compartir atributos de ruta, por ejemplo *middleware* o *namespaces*, a un grupo de rutas sin necesidad de definir los atributos a cada una de manera individual. Los atributos compartidos se pasan en un como array al primer parámetro del método `Route::group` .

Middleware

Para asignar un *middleware* a todas las rutas de un grupo, se puede utilizar el método `middleware` antes de definir el grupo. Los *middleware* se ejecutarán en el mismo orden que se pasen en el array:

```

Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second Middleware
    });

    Route::get('user/profile', function () {
        // Uses first & second Middleware
    });
});

```

Namespaces

Otro caso de uso común para los grupos de rutas es el asignar un mismo *namespace* PHP a un grupo de controladores utilizando el método `namespace` :

```
Route::namespace('Admin')->group(function () {
    // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

Recordar, el `RouteServiceProvider` incluye ya por defecto los archivos de ruta en un grupo con *namespace*, permitiendo registrar controladores sin tener que especificar el *namespace* completo `App\Http\Controllers`. Así, únicamente hay que especificar la porción del *namespace* que vendrá tras el *namespace* base `App\Http\Controllers`.

Rutas de sub-dominios

Los grupos de rutas se pueden utilizar para gestionar el enrutamiento de sub-dominios. A los sub-dominios se les pueden asignar parámetros al igual que a cualquier otra ruta, permitiendo capturar una porción del sub-domino para usarla en nuestra ruta o controlador. El sub-domino se especifica llamando al método `domain` antes de definir el grupo:

```
Route::domain('{account}.myapp.com')->group(function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

Prefijos de Rutas

El método `prefix` se puede utilizar para prefijar cada ruta en un grupo con una URI concreta. Por ejemplo, para añadir un prefijo a todas las URLs en el grupo `admin` :

```
Route::prefix('admin')->group(function () {
    Route::get('users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Enlazar modelos a rutas

Al inyectar el ID de un modelo a una ruta o acción de controlador, normalmente se ejecutará una consulta contra la base de datos para obtener el modelo que corresponde con dicho ID. El *route model binding* ayuda a inyectar instancias de modelo de forma automática en las rutas. Por ejemplo, en lugar de inyectar el ID de usuario, se puede inyectar la instancia de modelo completa `User` que concuerda con el ID dado.

Enlazado implícito – *Implicit binding*

Laravel resuelve de forma automática los modelos Eloquent que se definen en rutas o acciones de controladores cuyo parámetro coincide con el nombre de un segmento de la ruta. Por ejemplo:

```
Route::get('api/users/{user}', function (App\User $user) {
    return $user->email;
});
```

Puesto que la variable `$user` se requiere como tipo de `App\User` y el nombre de la variable coincide con el segmento URI `{user}`, Laravel inyectará automáticamente la instancia de modelo que corresponda con el valor de la petición URI. Si no se encuentra el modelo en la base de datos, se generará una respuesta HTTP 404 automáticamente.

Personalizar el nombre de la clave

Para utilizar una columna alternativa a `id` al obtener el modelo, se puede reemplazar el método `getRouteKeyName` en el modelo Eloquent:

```
/**
 * Get the route key for the model.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'slug';
}
```

Enlazado explícito

Para registrar un modelo de forma explícita, utilizar el método `model` para especificar la clase de un parámetro concreto. Se deben definir en el método `boot` del `RouteServiceProvider`:

```
public function boot()
{
    parent::boot();

    Route::model('user', App\User::class);
}
```

A continuación, definir una ruta que contenga un parámetro `{user}`:

```
Route::get('profile/{user}', function (App\User $user) {
    //
});
```

Puesto que se han enlazado todos los parámetros `{user}` al modelo `App\User`, se injectará una instancia de `User` en la ruta. Por ejemplo, una petición a `profile/1` injectará la instancia de `User` desde la base de datos que tenga un ID de `1`.

Si no se encuentra ninguna coincidencia en la base de datos, se retornará una respuesta HTTP 404 de forma automática.

Personalizar la lógica de resolución

Para utilizar una lógica propia de resolución, se puede utilizar el método `Route::bind`. La `Closure` que se pasa al método `bind` recibirá el valor del segmento URI y debe retornar la instancia de la clase a injectar en la ruta:

```
public function boot()
{
    parent::boot();

    Route::bind('user', function ($value) {
        return App\User::where('name', $value)->first() ?? abort(404);
    });
}
```

Falsear el método del formulario

Los formularios HTML no soportan las acciones `PUT`, `PATCH` o `DELETE`. Por lo que al definir rutas `PUT`, `PATCH` o `DELETE` que provengan de un formulario HTML, será necesario añadir un campo oculto (hidden) `_method` en el formulario. El valor del campo `_method` se utilizará como verbo HTTP:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

Se puede utilizar el helper `method_field` para generar el campo `_method`:

```
{{ method_field('PUT') }}
```

Acceder a la ruta actual

Se pueden utilizar los métodos `current`, `currentRouteName` y `currentRouteAction` de la `Facade Route` para acceder a información sobre la ruta que gestiona la petición actual:

```
$route = Route::current();

$name = Route::currentRouteName();

$action = Route::currentRouteAction();
```

Ir a la documentación del API para conocer todos los métodos disponibles de [la clase subyacente de la facade Route](#) y [la instancia Route](#).

Middleware

- # **Introducción**
 - # **Definir Middleware**
 - # **Registrar Middleware**
 - # Middleware Global
 - # Asignar Middleware a Rutas
 - # Grupos de Middleware
 - # **Middleware con Parámetros**
 - # **Middleware Terminable**
-

Introducción

Los middleware proporcionan una herramienta para filtrar las peticiones HTTP que entran a la aplicación. Por ejemplo, Laravel incluye un middleware que verifica si el actual usuario de la aplicación está autenticado. Si el usuario no está autenticado, el middleware redirigirá al usuario a la vista de login. Sin embargo, si el usuario está autenticado, el middleware permitirá que la petición continúe y se ejecute en la aplicación.

Por supuesto, se pueden crear middleware adicionales para realizar otro tipo tareas además de la autenticación. Un CORS middleware podría usarse para añadir los headers adecuados a todas las respuestas de la aplicación. Un logging middleware podría registrar en el log todas las peticiones hechas a la aplicación.

Hay una gran cantidad de middleware incluidos en Laravel framework, incluyendo middleware para autenticación y protección CSRF. Todos estos middleware están ubicados en el directorio `app/Http/Middleware` .

Definir Middleware

La forma más sencilla de crear un nuevo middleware es utilizar el comando Artisan `make:middleware` :

```
php artisan make:middleware CheckAge
```

Este comando creará un nuevo middleware llamado `CheckAge` en el directorio `app/Http/Middleware`. Este middleware solo permitirá acceso a la ruta si la `age` suministrada es mayor a 200. De otra forma, el middleware redirigirá a los usuarios de vuelta a la URI `home`.

```
<?php

namespace App\Http\Middleware;

use Closure;

class CheckAge
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->age <= 200) {
            return redirect('home');
        }

        return $next($request);
    }
}
```

Como puede verse, si la `age` es menor o igual a `200`, el middleware retornará un redirect HTTP al cliente; de otra manera la request será ejecutada por la aplicación. Para pasar la petición hacia abajo en la aplicación (permitir al middleware "pasar"), simplemente hay que llamar al callback `$next` con `$request`.

Lo mejor es imaginar los middleware como una serie de "capas" por las que las requests HTTP deben pasar antes de que lleguen a la aplicación. Cada capa puede examinar la request e incluso rechazarla por completo.

Middleware Antes & Después

Que el middleware se ejecute antes o después de que la petición entre en la aplicación depende del uso del middleware en si mismo. Por ejemplo, el siguiente middleware ejecuta algunas tareas antes de que la petición sea gestionada por la aplicación:

```
<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // Perform action

        return $next($request);
    }
}
```

Sin embargo, este middleware ejecuta las tareas después de que la petición haya sido gestionada por la aplicación:

```
<?php

namespace App\Http\Middleware;

use Closure;

class AfterMiddleware
{
    public function handle($request, Closure $next)
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}
```

Registrar Middleware

Middleware Global

Si se desea que un middleware se ejecute en todas las peticiones HTTP de la aplicación, simplemente debe listar el middleware en la propiedad `$middleware` de la clase `app/Http/Kernel.php`.

Asignar Middleware a Rutas

Si se desea que el middleware se ejecute en rutas específicas, primero debe asignarse al middleware un identificador en el archivo `app/Http/Kernel.php`. La propiedad `$routeMiddleware` de esta clase contiene registros de middleware incluidos por Laravel por defecto. Para agregar middleware personalizados, simplemente debeadirse a la lista el nuevo middleware y asignarle el identificador de acceso rápido que se deseé. Por ejemplo:

```
// Within App\Http\Kernel Class...

protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];
```

Una vez que el middleware ha sido registrado en el kernel HTTP, se puede utilizar el identificador de `middleware` para asignarlo a una ruta:

```
Route::get('admin/profile', function () {
    //
})->middleware('auth');
```

Se puede además asignar varios middleware a una ruta:

```
Route::get('/', function () {
    //
})->middleware('first', 'second');
```

Al asignar un middleware, se puede también pasar el nombre completo de la clase:

```
use App\Http\Middleware\CheckAge;

Route::get('admin/profile', function () {
    //
})->middleware(CheckAge::class);
```

Grupos de Middleware

En ocasiones es útil agrupar varios middleware sobre un mismo identificador haciendo la asignación a rutas mucho más simple. Esto se puede hacer utilizando la propiedad `$middlewareGroups` del kernel HTTP.

Por defecto, Laravel incluye los grupos de middleware `web` y `api` que contienen el middleware común que se suele aplicar a las rutas de web UI y API:

```
/**  
 * The application's route middleware groups.  
 *  
 * @var array  
 */  
  
protected $middlewareGroups = [  
    'web' => [  
        \App\Http\Middleware\EncryptCookies::class,  
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,  
        \Illuminate\Session\Middleware\StartSession::class,  
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,  
        \App\Http\Middleware\VerifyCsrfToken::class,  
        \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    ],  
  
    'api' => [  
        'throttle:60,1',  
        'auth:api',  
    ],  
];
```

Los grupos de middleware se pueden asignar a rutas y acciones de controladores utilizando la misma sintaxis que un middleware individual. De nuevo, los grupos únicamente permiten añadir varios middleware de una vez:

```
Route::get('/', function () {  
    //  
})->middleware('web');  
  
Route::group(['middleware' => ['web']], function () {  
    //  
});
```

El grupo `web` se aplica directamente al archivo `routes/web.php` a través del `RouteServiceProvider`.

Middleware con Parámetros

Los middleware pueden recibir parámetros adicionales. Por ejemplo, si la aplicación necesita verificar que el usuario autenticado tiene asignado cierto "rol" antes de ejecutar una acción, puede crearse un middleware `CheckRole` que reciba el nombre del rol como parámetro adicional.

Los parámetros adicionales del middleware deben ser pasados después de argumento `$next` :

```
<?php

namespace App\Http\Middleware;

use Closure;

class CheckRole
{
    /**
     * Handle the incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }
}
```

Los parámetros del middleware pueden ser especificados se define la ruta, separando el nombre del middleware y los parámetros con `:`. Multiples parámetros deben ser separados por comas:

```
Route::put('post/{id}', function ($id) {
    //
})->middleware('role:editor');
```

Middleware Terminable

En ocasiones, un middleware necesita realizar algunas acciones después de que la respuesta HTTP ha sido enviada al navegador. Por ejemplo, el middleware "session" incluido por defecto en Laravel, registra los datos de sesión *después* de que la respuesta haya sido enviada al navegador. Si se define un método `terminate` en el middleware, se ejecutará automáticamente después de que la respuesta se haya enviado al navegador.

```
<?php

namespace Illuminate\Session\Middleware;

use Closure;

class StartSession
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // Store the session data...
    }
}
```

El método `terminate` debe recibir tanto la petición como la respuesta. Una vez que se ha definido un middleware terminable, este debeadirse a la lista de rutas o middleware globales en `app/Http/Kernel.php`.

Cuando se llama al método `terminate` en el middleware, Laravel resolverá una nueva instancia del middleware desde el [service container](#). Si se desea usar la misma instancia cuando los métodos `handle` y `terminate` son llamados, debe registrarse en el container el middleware usando el método `singleton` del container.

Protección CSRF

- # Introducción
 - # Exluyendo URIs
 - # X-CSRF-Token
 - # X-XSRF-Token
-

Introducción

Laravel hace fácil la protección de su aplicación de ataques [cross-site request forgery](#) (CSRF). Solicitudes de falsificación a través de sitios *Cross-site request forgeries* son un tipo de ataques maliciosos a través de los cuales comandos no autorizados son ejecutados en nombre de un usuario autenticado.

Laravel genera automáticamente una prueba *token* CSRF para cada sesión de usuario activo administrado por la aplicación. Esta prueba se utiliza para verificar que el usuario autenticado es el que realmente hace las peticiones a la aplicación.

Cada vez que defina un formulario HTML en su aplicación usted debería incluir un campo prueba CSRF oculto en el formulario para que la protección *middleware* pueda validar la petición. Usted puede utilizar la función de ayuda `csrf_field` para generar el campo con la prueba:

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

El *middleware* `VerifyCsrfToken` el cual está incluído en el grupo de *middlewares* web, automáticamente verificará que la prueba en la petición de entrada concuerda con la almacenada en la sesión.

Pruebas CSRF y JavaScript

Cuando construye aplicaciones conducidas por JavaScript, es conveniente que su librería JavaScript de HTTP adjunte automáticamente la prueba CSRF a cada petición de salida. By default, the `resources/assets/js/bootstrap.js` file registers the value of the `csrf-token` meta tag with the Axios HTTP library. Si usted no está utilizando esta librería, necesitará configurar manualmente este comportamiento para su aplicación.

Excluir URIs de la protección CSRF

A veces puede desear excluir un conjunto de URIs de la protección CSRF. Por ejemplo, si usted está usando [Stripe](#) para procesar pagos y está utilizando su sistema de enganche web, necesitará excluir la ruta del manejador del enganche web de la protección CSRF ya que Stripe no conocerá la prueba CRSF que debe enviar a sus rutas.

Normalmente, usted debería situar este tipo de cosas en rutas fuera de su grupo *middleware* web que el `RouteServiceProvider` aplica a todas las rutas en el fichero `routes/web.php`. No obstante, usted puede también excluir las rutas agregándolas a la propiedad `$except` del *middleware* `VerifyCsrfToken`:

```
<?php

namespace App\Http\Middleware;

use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as Middleware;

class VerifyCsrfToken extends Middleware
{
    /**
     * The URIs that should be excluded from CSRF verification.
     *
     * @var array
     */
    protected $except = [
        'stripe/*',
    ];
}
```

X-CSRF-TOKEN

Además para comprobar la prueba CSRF como parametro POST el *middleware* `VerifyCsrfToken` comprobará también el `X-CSRF-TOKEN` para las cabeceras de las peticiones. Podría, por ejemplo, almacenar la prueba en una `meta` etiqueta HTML:

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

Entonces, una vez haya creado la `meta` etiqueta, usted puede indicar a una librería como Jquery como agregar automáticamente el token a todas las cabeceras de las peticiones. Esto proporciona una protección CSRF simple y conveniente para sus aplicaciones basadas en AJAX:

```
$.ajaxSetup({
  headers: {
    'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
  }
});
```

Por defecto, el fichero `resources/assets/js/bootstrap.js` registra el valor de la meta etiqueta `csrf-token` con la librería HTTP Axios. Si no está utilizando esta librería, usted necesitará configurar manualmente este comportamiento para su aplicación.

X-XSRF-TOKEN

Laravel almacena la prueba CSRF en la *cookie* `XSRF-TOKEN` que es incluída con cada respuesta generada por el framework. Puede utilizar la *cookie* para establecer la cabecera de la petición `X-XSRF-TOKEN`.

Esta *cookie* es primordialmente enviada por conveniencia para algunos frameworks de JavaScript u librerías como Angular y Axios, sitúa automáticamente su valor en la cabecera `X-XSRF-TOKEN`.

Controladores

Introducción

Controladores básicos

- # Definir controladores
- # Controladores & *namespaces*
- # Controladores de acción única

Middleware y controladores

Controladores de recursos – *resource controllers*

- # Rutas de recursos parciales
- # Nombrar rutas de recursos
- # Nombrar parámetros en rutas de recursos
- # Traducir las URIs de los recursos
- # Complementar a los controladores de recursos

Inyección de dependencias & controladores

Caché de rutas

Introducción

En lugar de definir toda la lógica para la gestión de una petición dentro de *Closures* o funciones anónimas en los archivos de rutas, se puede organizar este comportamiento en unas clases llamadas Controladores (*controllers*). Los controladores pueden agrupar la lógica de gestión de peticiones relacionadas en una única clase. Estos controladores se encuentran normalmente en el directorio [app/Http/Controllers](#) .

Controladores básicos

Definir controladores

A continuación se muestra un ejemplo de una clase de controlador básico. Tenga en cuenta que el controlador hereda de la clase de controlador base incluida con Laravel. La clase base provee de una serie de métodos útiles como el método [middleware](#) , que se puede usar para

adjuntar un *middleware* a las acciones del controlador:

```
<?php

namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

Se puede apuntar una ruta a la acción de este controlador así:

```
Route::get('user/{id}', 'UserController@show');
```

Ahora, cuando una petición concuerda con la URI de la ruta, se ejecutará el método `show` de la clase `UserController`. Por supuesto, los parámetros de la ruta se pasarán también a este método.

Los controladores no **requieren** heredar la clase base. Sin embargo, no se tendrá acceso a las características como los métodos `middleware`, `validate`, y `dispatch`.

Controladores & namespaces

Es muy importante tener en cuenta que no es necesario especificar el *namespace* completo del controlador cuando se define la ruta del controlador. Como el `RouteServiceProvider` carga los archivos de ruta dentro de un grupo de rutas que contiene el *namespace*, únicamente

especificamos la porción del nombre de clase que viene después del *namespace* `App\Http\Controllers`.

Si se prefiere anidar u organizar los controladores más profundos que el directorio `App\Http\Controllers`, simplemente se debe utilizar el nombre de la clase relativo a `App\Http\Controllers` como *namespace* raíz. Por lo que, si el la clase del controlador es `App\Http\Controllers\Photos\AdminController`, se debe registrar la siguiente ruta:

```
Route::get('foo', 'Photos\AdminController@method');
```

Controladores de acción única

Si se necesita, se puede definir un controlador que únicamente gestione una única acción, simplemente es necesario colocar el método `__invoke` dentro del controlador:

```
<?php

namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class ShowProfile extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function __invoke($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

Cuando se registra una ruta de un controlador de acción única, no se necesita especificar ningún método en la ruta:

```
Route::get('user/{id}', 'ShowProfile');
```

Middleware y controladores

A las rutas de los controladores se les puede asignar [middleware](#) del siguiente modo:

```
Route::get('profile', 'UserController@show')->middleware('auth');
```

Sin embargo, es más conveniente especificar el *middleware* en el constructor del controlador. Utilizando el método [middleware](#) desde el constructor del controlador, se puede asignar un *middleware* a las acciones del controlador. Incluso se puede restringir el *middleware* a únicamente ciertos métodos:

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('log')->only('index');

        $this->middleware('subscribed')->except('store');
    }
}
```

Los controladores también permiten registrar *middleware* usando un *Closure* o función anónima. Esto proporciona una forma conveniente de definir un *middleware* para un solo controlador sin definir una clase *middleware* completa:

```
$this->middleware(function ($request, $next) {  
    // ...  
  
    return $next($request);  
});
```

Se puede asignar un *middleware* a un subconjunto de acciones del controlador; sin embargo, esto puede indicar que el controlador está creciendo demasiado. En su lugar, se recomienda dividir el controlador en controladores más pequeños.

Controladores de recursos – *resource controllers*

El *routing* de recursos de Laravel asigna las rutas "CRUD" típicas a un controlador con una sola línea de código. Por ejemplo, la creación de un controlador que gestiona todas las peticiones HTTP sobre "photos" (fotos) almacenadas por nuestra aplicación. Utilizando el comando de Artisan `make:controller`, se puede crear un controlador rápidamente:

```
php artisan make:controller PhotoController --resource
```

El comando generará un controlador en el archivo `app/Http/Controllers/PhotoController.php`. El controlador incluirá un método para cada una de las operaciones disponibles para el recurso.

A continuación, se puede registrar una ruta de recursos para el controlador:

```
Route::resource('photos', 'PhotoController');
```

Esta única declaración crea varias rutas que gestionan los diferentes métodos sobre un recurso. El controlador generado incluirá los métodos ya declarados para cada una de estas acciones, incluyendo notas que informan sobre que URLs y verbos gestionan.

Se pueden registrar varios controladores de recursos a la vez pasando una *array* al método `resources`:

```
Route::resources([
    'photos' => 'PhotoController',
    'posts' => 'PostController'
]);
```

Acciones gestionadas por controladores de recursos

Verbo	URI	Acción	Nombre de Ruta
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Especificar el modelo del recurso

Si se está utilizando el *route model binding* y se desea que los métodos del controlador de recursos incluyan un *type-hint* de una instancia del modelo, se puede usar la opción `--model` al generar el controlador:

```
php artisan make:controller PhotoController --resource --model=Photo
```

Suplantación de métodos en formularios

Los formularios HTML no pueden realizar peticiones `PUT`, `PATCH`, o `DELETE`, para hacerlo se necesita agregar un campo oculto `_method` para suplantar estos verbos HTTP. El helper `method_field` permite crear este campo de forma rápida:

```
{{ method_field('PUT') }}
```

Rutas de recursos parciales

Cuando se declara una ruta de recursos, se puede especificar un subconjunto de acciones que el controlador debe manejar en lugar del conjunto completo de acciones predeterminadas:

```
Route::resource('photo', 'PhotoController', ['only' => [
    'index', 'show'
]]);

Route::resource('photo', 'PhotoController', ['except' => [
    'create', 'store', 'update', 'destroy'
]]);
```

Rutas de recursos para API

Al declarar rutas de recursos que consumirá un API, normalmente se querrá excluir rutas que presenten plantillas HTML, como `create` y `edit`. Para su comodidad, se puede usar el método `apiResource` para excluir automáticamente estas dos rutas:

```
Route::apiResource('photo', 'PhotoController');
```

Se pueden registrar varios controladores de recursos de API a la vez pasando una *array* al método `apiResources`:

```
Route::apiResources([
    'photos' => 'PhotoController',
    'posts' => 'PostController'
]);
```

Nombrar rutas de recursos

Por defecto, todas las acciones de los controladores de recursos tienen un nombre de ruta; sin embargo, se puede sobrescribir este nombre pasando un *array* `names` con sus opciones:

```
Route::resource('photo', 'PhotoController', ['names' => [
    'create' => 'photo.build'
]]);
```

Nombrar parámetros en rutas de recursos

Por defecto, `Route::resource` crea los parámetros de ruta para las rutas de recursos utilizando la versión "singular" del nombre del recurso. Se puede sobrescribir esto fácilmente por recurso pasando `parameters` en el *array* de opciones. El *array* de `parameters` debe ser un *array* asociativo de los nombres de los recursos y el nombre de su parámetro:

```
Route::resource('user', 'AdminController', ['parameters' => [
    'user' => 'admin_user'
]]);
```

El ejemplo anterior genera las siguientes URLs para la ruta `show` del recurso:

```
/user/{admin_user}
```

Traducir las URLs de los recursos

Por defecto, `Route::resource` crea las URLs de los recursos usando verbos en inglés. Si se necesita traducir los verbos de las acciones `create` y `edit`, se puede usar el método `Route::resourceVerbs`. Se puede hacer esto en el método `boot` de nuestro `AppServiceProvider`:

```
use Illuminate\Support\Facades\Route;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);
}
```

Una vez que se han personalizado los verbos, una ruta de recursos como `Route::resource('fotos', 'PhotoController')` producirá las siguientes URLs:

```
/fotos/crear
```

```
/fotos/{foto}/editar
```

Complementar a los controladores de recursos

Si es necesario agregar rutas adicionales a un controlador de recursos más allá de las predeterminadas, se deben definir antes de llamar a `Route::resource`; de otro modo, las rutas definidas por el método `resource` pueden prevalecer sobre las rutas suplementarias:

```
Route::get('photos/popular', 'PhotoController@method');
```

```
Route::resource('photos', 'PhotoController');
```

Recuerde mantener sus controladores enfocados. Si se encuentra rutinariamente necesitando métodos fuera del conjunto típico de las acciones de recursos, considere dividir su controlador en dos controladores más pequeños.

Inyección de dependencias & controladores

Inyección en constructores

El [service container](#) de Laravel se utiliza para resolver todos los controladores. Como resultado, se puede *type-hint* (firma del método) cualquier dependencia que el controlador pueda tener en su constructor. Las dependencias declaradas se resuelven automáticamente y se inyectan en la instancia del controlador:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

Y por supuesto, se puede incluir cualquier [Contracto de Laravel](#). Si el contenedor puede resolverlo, se puede utilizar en la firma del constructor. Dependiendo de la aplicación, inyectar las dependencias dentro del controlador puede ofrecer mejor control sobre el *testing*.

Inyección en métodos

Además de inyectar en constructores, se puede hacer *type-hint* de dependencias en los métodos del controlador. Un caso de uso muy común para la inyección es la de la instancia de `Illuminate\Http\Request` dentro de los métodos del controlador:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->name;

        //
    }
}
```

Si el método del controlador también espera datos de entrada de un parámetro en la ruta se deben listar los parámetros de ruta después de las otras dependencias. Por ejemplo, si la ruta está definida así:

```
Route::put('user/{id}', 'UserController@update');
```

Se podría hacer *type-hint* de `Illuminate\Http\Request` y acceder al parámetro de ruta `id` definiendo el método del controlador de la siguiente forma:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the given user.
     *
     * @param Request $request
     * @param string $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Caché de rutas

Las rutas basadas en *Closures* o funciones anónimas no se pueden almacenar en caché. Para utilizar el almacenamiento en caché de rutas, debe convertir las rutas de *Closure* en controladores.

Si la aplicación usa exclusivamente rutas basadas en controladores, se puede aprovechar la caché de rutas de Laravel. Utilizando la caché de rutas se reducirá drásticamente el tiempo que toma la aplicación en registrar todas las rutas. En algunos casos, ¡el registro de rutas puede ser hasta 100x más rápido. Para generar una caché de rutas, simplemente hay que ejecutar el comando de Artisan `route:cache` :

```
php artisan route:cache
```

Tras ejecutar el comando, el archivo de rutas en caché se cargará en cada solicitud. Recordar, si se añaden nuevas rutas se debe generar de nuevo una nueva caché. Es por esto, se recomienda utilizar el comando `route:cache` únicamente en el entorno de producción.

Se puede utilizar el comando `route:clear` para limpiar las rutas en caché:

```
php artisan route:clear
```

Peticiones HTTP

Acceder a la petición

- # Método & ruta de la petición
- # Peticiones PSR-7

Trimming de datos & normalización

Obtener datos de entrada

- # Datos de entrada antiguos
- # Cookies

Archivos

- # Recuperación de archivos subidos
- # Almacenamiento de archivos subidos

Configurar proxies de confianza

Acceder a la petición

Para obtener una instancia de la petición HTTP actual vía inyección de dependencias, se debe hacer *type-hint* de la clase `Illuminate\Http\Request` en el método del controlador. La instancia de la petición entrante será automáticamente inyectada por el [service container](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

Inyección de dependencias & parámetros de rutas

Si el método del controlador también espera datos de entrada de un parámetro en la ruta se deben listar los parámetros de ruta después de las otras dependencias. Por ejemplo, si la ruta está definida así:

```
Route::put('user/{id}', 'UserController@update');
```

Se podría hacer *type-hint* de `Illuminate\Http\Request` y acceder al parámetro de ruta `id` definiendo el método del controlador de la siguiente forma:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     *
     * @param Request $request
     * @param string $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Accediendo a las peticiones usando *Closures* de ruta

Se puede usar el *type-hinting* de la clase `Illuminate\Http\Request` en un *Closure* de ruta. El *service container* injectará automáticamente la petición entrante dentro del *Closure* al ejecutarse:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    //
});
```

Método & ruta de la petición

La instancia de `Illuminate\Http\Request` provee una variedad de métodos para examinar una petición HTTP dentro de la aplicación, extiende de la clase `Symfony\Component\HttpFoundation\Request`. A continuación se muestran los métodos más importantes.

Obtener la ruta de la petición

El método `path` retorna la información de la ruta de la petición. Así que, si la petición se realizara sobre `http://domain.com/foo/bar`, el método `path` retornaría `foo/bar`:

```
$uri = $request->path();
```

El método `is` permite verificar si la ruta coincide con un patrón determinado. Se puede utilizar el carácter `*` como comodín al utilizar este método:

```
if ($request->is('admin/*')) {  
    //  
}
```

Obtener la URL de la petición

Para obtener la URL completa de una petición entrante se pueden usar los métodos `url` o `fullUrl`. El método `url` retorna la URL sin la cadena de consulta, mientras que `fullUrl` incluye todos los parámetros:

```
// Without Query String...  
$url = $request->url();  
  
// With Query String...  
$url = $request->fullUrl();
```

Obtener el método de la petición

El método `method` retornará el verbo HTTP de la petición. Además se puede utilizar el método `isMethod` para verificar que el verbo HTTP coincide con una cadena dada:

```
$method = $request->method();  
  
if ($request->isMethod('post')) {  
    //  
}
```

El [estándar PSR-7](#) especifica interfaces para los mensajes HTTP, incluyendo respuestas y peticiones. Si se desea obtener una instancia de una petición PSR-7 en lugar de una petición de Laravel, primero se deben instalar algunas librerías. Laravel utiliza el componente *Symfony HTTP Message Bridge* para convertir las típicas peticiones y respuestas de Laravel en implementaciones compatibles con PSR-7:

```
composer require symfony/psr-http-message-bridge
composer require zendframework/zend-diactoros
```

Una vez instaladas las librerías, se puede obtener una petición PSR-7 usando el *type-hinting* de la interfaz de la petición dentro del *route Closure* o del método de un controlador:

```
use Psr\Http\Message\ServerRequestInterface;

Route::get('/', function (ServerRequestInterface $request) {
    //
});
```

Si se retorna una instancia de una respuesta PSR-7 desde la ruta o controlador, automáticamente se convierte en una instancia de respuesta de Laravel que se mostrará por el framework.

Trimming de datos & normalización

Por defecto, Laravel incluye los *middleware* `TrimStrings` y `ConvertEmptyStringsToNull` de manera global dentro del *stack* de *middlewares*. Estos *middleware* se listan en la clase `App\Http\Kernel`. Los *middleware* aplicarán la función `trim` automáticamente todos los campos de cadena entrantes a petición, así como convertirán cualquier campo de cadena vacío en `null`. Esto le permite no tener que preocuparse por las cuestiones de normalización en sus rutas y controladores.

Si desea deshabilitar este comportamiento, puede eliminar los dos *middleware* de la pila de *middleware* de su aplicación eliminándolos de la propiedad `$middleware` de su clase `App\Http\Kernel`.

Obtener datos de entrada

Obtener todos los datos de entrada

También se pueden recuperar todos los datos de entrada como un `array` usando el método `all`:

```
$input = $request->all();
```

Recuperar un valor de entrada

Usando unos pocos métodos sencillos, se puede acceder a todos los datos ingresados por el usuario desde la instancia de `Illuminate\Http\Request` sin preocuparse por el método HTTP que se haya usado para la petición. Sin importar el verbo HTTP, el método `input` se puede usar para recuperar las entradas del usuario:

```
$name = $request->input('name');
```

Se puede pasar un valor por defecto como segundo argumento del método `input`. Este valor se retornará si el valor de la entrada solicitada no está presente en la petición:

```
$name = $request->input('name', 'Sally');
```

Cuando se trabaja con formularios que contienen *arrays*, se usa la "notación por puntos" o "*dot notation*" para acceder a los datos:

```
$name = $request->input('products.0.name');  
  
$names = $request->input('products.*.name');
```

Recuperar valores de la cadena de consulta

Mientras que el método `input` recupera valores de toda la carga útil de la solicitud (incluyendo la cadena de consulta), el método `query` sólo recupera valores de la cadena de consulta:

```
$name = $request->query('name');
```

Si los datos de valor de la cadena de consulta solicitada no están presentes, se devolverá el segundo argumento de este método:

```
$name = $request->query('name', 'Helen');
```

Puede llamar al método `query` sin ningún argumento para recuperar todos los valores de la cadena de consulta como *array* asociativo:

```
$query = $request->query();
```

Recuperar datos de entrada mediante propiedades dinámicas

También puede acceder a los datos de entrada de usuario utilizando las propiedades dinámicas en la instancia de `Illuminate\Http\Request`. Por ejemplo, si uno de los formularios de su solicitud contiene un campo `name`, puede acceder al valor del campo así:

```
$name = $request->name;
```

Cuando se utilizan propiedades dinámicas, Laravel primero buscará el valor del parámetro en la carga útil solicitada. Si no está presente, Laravel buscará el campo en los parámetros de ruta.

Recuperar valores de entrada JSON

Cuando envíe solicitudes JSON a su aplicación, puede acceder a los datos JSON mediante el método `input`, siempre que el encabezado `Content-Type` de la solicitud esté correctamente configurado en `application/json`. Incluso puede utilizar la "sintaxis de puntos" para buscar en los *array* JSON:

```
$name = $request->input('user.name');
```

Recuperar una parte de los datos de entrada

Si se quiere recuperar un subconjunto de los datos de entrada, se pueden usar los métodos `only` y `except`. Ambos métodos aceptan un `array` o una lista dinámica de argumentos:

```
$input = $request->only(['username', 'password']);  
  
$input = $request->only('username', 'password');  
  
$input = $request->except(['credit_card']);  
  
$input = $request->except('credit_card');
```

El método `only` devuelve todos los pares clave/valor que solicite; sin embargo, no devuelve los pares clave/valores que no están presentes en la solicitud.

Determinar si un valor de entrada está presente

Puede utilizar el método `has` para determinar si un valor está presente en la solicitud. El método `has` devuelve `true` si el valor está presente:

```
if ($request->has('name')) {  
    //  
}
```

Cuando se le pasa un *array*, el método `has` determinará si todos los valores especificados están presentes:

```
if ($request->has(['name', 'email'])) {  
    //  
}
```

Si desea determinar si un valor está presente en la solicitud y no está vacío, puede utilizar el método `filled`:

```
if ($request->filled('name')) {  
    //  
}
```

Datos de entrada antiguos – *Old input*

Laravel le permite mantener los datos de entrada de una solicitud durante la próxima solicitud. Esta característica es particularmente útil para llenar formularios después de detectar errores de validación. Sin embargo, si está usando la [validación](#) incluida en Laravel, es poco probable que tenga que utilizar manualmente estos métodos, ya que algunas de las funciones de validación los llamarán automáticamente.

Flashing Input To The Session

El método `flash` en la clase `Illuminate\Http\Request` mantendrá la entrada actual en la [sesión](#) para que esté disponible durante la próxima solicitud del usuario a la aplicación:

```
$request->flash();
```

También puede utilizar los métodos `flashOnly` y `flashExcept` para enviar un subconjunto de los datos de solicitud a la sesión. Estos métodos son útiles para mantener la información confidencial, como las contraseñas, fuera de la sesión:

```
$request->flashOnly(['username', 'email']);  
  
$request->flashExcept('password');
```

Flashing Input Then Redirecting

Dado que a menudo querrá hacer un *flash input* a la sesión y luego redirigir a la página anterior, puede encadenar fácilmente el *input flashing* a un redireccionamiento usando el método `withInput`:

```
return redirect('form')->withInput();  
  
return redirect('form')->withInput(  
    $request->except('password')  
);
```

Obtener datos de entrada antiguos

Para obtener la entrada de la petición anterior, utilice el método `old` de la instancia `Request`. El método `old` extraerá los datos de entrada anteriores de la [session](#):

```
$username = $request->old('username');
```

Laravel también proporciona un *helper* global `old`. Si está mostrando la entrada antigua dentro de una [plantilla Blade](#), es más conveniente utilizar el helper `old`. Si no existe ninguna entrada antigua para el campo dado, se devolverá `null`:

```
<input type="text" name="username" value="{{ old('username') }}>
```

Cookies

Obtener las cookies de la petición

Todas las *cookies* creadas por el framework de Laravel están cifradas y firmadas con un código de autenticación, lo que significa que se considerarán inválidas si han sido modificadas por el cliente. Para recuperar un valor de una *cookie* de la solicitud, utilice el método `cookie` en una instancia de `Illuminate\Http\Request`:

```
$value = $request->cookie('name');
```

Alternativamente, puede utilizar la *facade* `Cookie` para acceder a los valores de las *cookies*:

```
$value = Cookie::get('name');
```

Añadir cookies a las respuestas

Puede adjuntar una *cookie* a una respuesta con una instancia de `Illuminate\Http\Response` usando el método `cookie`. Debe proporcionar el nombre, valor y número de minutos que la *cookie* debe considerarse válida:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

El método `cookie` acepta además ciertos argumentos que se usan menos frecuentemente. Generalmente estos argumentos tienen el mismo propósito y significado que los argumentos del método nativo de PHP [setcookie](#):

```
return response('Hello World')->cookie(
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
);
```

Por otro lado, se puede utilizar la *facade* `Cookie` para crear una "cola" de *cookies* a añadir a la respuesta de la aplicación. El método `queue` acepta una instancia de `Cookie` o los argumentos necesarios para crear una instancia de `Cookie`. Estas *cookies* se adjuntarán a la respuesta antes de que se devuelva al navegador:

```
Cookie::queue(Cookie::make('name', 'value', $minutes));

Cookie::queue('name', 'value', $minutes);
```

Generación de cookies

Si desea generar una instancia de `Symfony\Component\HttpFoundation\Cookie` en una respuesta posterior, puede utilizar el *helper* global `cookie`. Esta *cookie* no será enviada de vuelta al cliente a menos que se adjunte a una instancia de una respuesta:

```
$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);
```

Archivos

Obtener archivos subidos

Puede acceder a los archivos subidos desde una instancia `Illuminate\Http\Request` usando el método `file` o usando propiedades dinámicas. El método `file` devuelve una instancia de la clase `Illuminate\Http\UploadedFile`, que hereda la clase PHP `SplFileInfo` y proporciona una variedad de métodos para interactuar con el archivo:

```
$file = $request->file('photo');

$file = $request->photo;
```

Puede determinar si un archivo está presente en la solicitud utilizando el método `hasFile` :

```
if ($request->hasFile('photo')) {  
    //  
}
```

Validación de subidas exitosas

Además de comprobar si el archivo está presente, puede verificar que no hubo problemas para cargar el archivo a través del método `isValid` :

```
if ($request->file('photo')->isValid()) {  
    //  
}
```

Extensiones & rutas de archivo

La clase `UploadedFile` también contiene métodos para acceder a la ruta totalmente calificada del archivo y su extensión. El método `extension` intentará adivinar la extensión del archivo en función de su contenido. Esta extensión puede ser diferente de la que fue suministrada por el cliente:

```
$path = $request->photo->path();  
  
$extension = $request->photo->extension();
```

Otros métodos

Existen otros métodos en la instancia de `UploadedFile`. Consulte la [documentación API de la clase](#) para obtener más información sobre estos métodos.

Almacenar archivos subidos

Para almacenar un archivo subido, normalmente utilizará uno de sus [filesystems](#) configurados. La clase `UploadedFile` tiene un método `store` que moverá un archivo subido a uno de sus discos, que puede ser una ubicación en su sistema de archivos local o incluso una ubicación de almacenamiento en la nube como Amazon S3.

El método `store` acepta la ruta donde se debe almacenar el archivo en relación al directorio raíz configurado del sistema de archivos. Esta ruta no debe contener un nombre de archivo, ya que se generará automáticamente un ID único para que sirva como nombre de archivo.

El método `store` también acepta un segundo argumento opcional para el nombre del disco que debe usarse para almacenar el archivo. El método devolverá la ruta del archivo relativa a la raíz del disco:

```
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

Si no desea que se genere automáticamente un nombre de archivo, puede utilizar el método `storeAs`, que acepta la ruta, el nombre de archivo y el nombre del disco como argumentos:

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

Configurar proxies de confianza

Al ejecutar su aplicación tras un balanceador de carga (*load balancer*) que finaliza los certificados TLS/SSL, es posible que en ocasiones su aplicación no genere los enlaces HTTPS. Normalmente ocurre porque la aplicación está recibiendo tráfico redirigido desde el *load balancer* en el puerto 80 y no sabe si debe generar enlaces seguros o no.

Para solucionar esto, se puede utilizar el *middleware* `App\Http\Middleware\TrustProxies` que se incluye con Laravel, el cual permitirá personalizar los *load balancers* o *proxies* en los que debe confiar la aplicación. Los "proxies de confianza" (*trusted proxies*) se deben listar como un *array* en la propiedad `$proxies` de este *middleware*. Además de configurar *trusted proxies*, se pueden configurar cabeceras a enviar por el proxy con información sobre la petición original:

```

<?php

namespace App\Http\Middleware;

use Illuminate\Http\Request;
use Fideloper\Proxy\TrustProxies as Middleware;

class TrustProxies extends Middleware
{
    /**
     * The trusted proxies for this application.
     *
     * @var array
     */
    protected $proxies = [
        '192.168.1.1',
        '192.168.1.2',
    ];

    /**
     * The current proxy header mappings.
     *
     * @var array
     */
    protected $headers = [
        Request::HEADER_FORWARDED => 'FORWARDED',
        Request::HEADER_X_FORWARDED_FOR => 'X_FORWARDED_FOR',
        Request::HEADER_X_FORWARDED_HOST => 'X_FORWARDED_HOST',
        Request::HEADER_X_FORWARDED_PORT => 'X_FORWARDED_PORT',
        Request::HEADER_X_FORWARDED_PROTO => 'X_FORWARDED_PROTO',
    ];
}

```

Confiar en todos los *proxies*

Si se está utilizando Amazon AWS o otro proveedor de *load balancer* "cloud", quizá no se conozca la IP de los *balancers* actuales. En este caso, se puede utilizar `**` para confiar en todos los proxies:

```
/**  
 * The trusted proxies for this application.  
 *  
 * @var array  
 */  
protected $proxies = '***';
```

Respuestas HTTP

Crear respuestas

- # Añadir cabeceras a las respuestas
- # Añadir *cookies* a las respuestas
- # *Cookies & encriptación*

Redirecciones

- # Redirigir a rutas con nombre
- # Redireccionar a acciones de controladores
- # Redireccionar con datos de sesión *flash*

Otros tipos de respuestas

- # Respuestas de vistas
- # Respuestas JSON
- # Descargas de archivos
- # Respuesta de archivos

Macros para respuestas

Crear respuestas

Strings & arrays

Todas las rutas y controladores deben retornar una respuesta para enviarla al navegador del usuario. Laravel provee diferentes formas para retornar estas respuestas. La respuesta más básica es simplemente devolver una cadena desde una ruta o controlador. El framework convertirá automáticamente la cadena en una respuesta HTTP completa:

```
Route::get('/', function () {  
    return 'Hello World';  
});
```

Además de devolver cadenas desde sus rutas y controladores, también puede devolver *arrays*. El framework convertirá automáticamente el *array* en una respuesta JSON:

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

¿Sabía que puede retornar Colecciones Eloquent desde rutas o controladores? Se convertirán a JSON automáticamente. ¡Pruébelas!

Objetos Response

Normalmente no solo se retornarán cadenas o *arrays* desde las acciones de las rutas. En su lugar, se retornarán instancias de `Illuminate\Http\Response` o [vistas](#).

Retornar una instancia `Response` completa permite personalizar el código de respuesta HTTP y sus cabeceras. Una instancia `Response` hereda de la clase `Symfony\Component\HttpFoundation\Response`, la cual provee una gran variedad de métodos para la creación de respuestas HTTP:

```
Route::get('home', function () {
    return response('Hello World', 200)
        ->header('Content-Type', 'text/plain');
});
```

Añadir cabeceras a las respuestas

Hay que tener en cuenta que la mayoría de los métodos de una respuesta son encadenables, permitiendo una construcción fluida. Por ejemplo, se puede utilizar el método `header` para añadir una serie de cabeceras a la respuesta antes de devolverla al usuario:

```
return response($content)
    ->header('Content-Type', $type)
    ->header('X-Header-One', 'Header Value')
    ->header('X-Header-Two', 'Header Value');
```

O se puede utilizar el método `withHeaders` para especificar un *array* de cabeceras:

```
return response($content)
    ->withHeaders([
        'Content-Type' => $type,
        'X-Header-One' => 'Header Value',
        'X-Header-Two' => 'Header Value',
    ]);
]
```

Añadir *cookies* a las respuestas

El método `cookie` en una instancia "Response" permite añadir *cookies* a la misma. Por ejemplo, se puede utilizar el método `cookie` para generar una *cookie* y añadirla a la respuesta de forma fluida:

```
return response($content)
    ->header('Content-Type', $type)
    ->cookie('name', 'value', $minutes);
```

El método `cookie` acepta además ciertos argumentos que se usan menos frecuentemente. Generalmente estos argumentos tienen el mismo propósito y significado que los argumentos del método nativo de PHP [setcookie](#):

```
->cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

Por otro lado, se puede utilizar la facade `Cookie` para crear una "cola" de *cookies* para añadir a la respuesta de la aplicación. El método `queue` acepta una instancia de `Cookie` o los argumentos necesarios para crear una instancia de `Cookie`. Éstas *cookies* se adjuntarán a la respuesta antes de que se devuelva al navegador:

```
Cookie::queue(Cookie::make('name', 'value', $minutes));

Cookie::queue('name', 'value', $minutes);
```

Cookies & encriptación

Por defecto, todas las *cookies* generadas por Laravel están encriptadas y firmadas por lo que no pueden ser modificadas o leídas por un cliente. Para desactivar la encriptación de un conjunto de *cookies* generadas por la aplicación, se puede utilizar la propiedad `$except` del middleware `App\Http\Middleware\EncryptCookies`, el cual se encuentra en el directorio `app/Http/Middleware` :

```
/**  
 * The names of the cookies that should not be encrypted.  
 *  
 * @var array  
 */  
  
protected $except = [  
    'cookie_name',  
];
```

Redirecciones

Las redirecciones son instancias de la clase `Illuminate\Http\RedirectResponse` y contienen las cabeceras apropiadas para redirigir al usuario a otra URL. Hay varias formas de generar una instancia de `RedirectResponse`. La más sencilla es utilizando el helper global `redirect`:

```
Route::get('dashboard', function () {  
    return redirect('home/dashboard');  
});
```

En ocasiones puede ser necesario redirigir al usuario a una ubicación anterior, como cuando el envío de un formulario es inválido. Se puede hacer utilizando la función global `back`.

Puesto que esta característica utiliza la [sesión](#), hay que asegurarse de que la ruta que llama al método `back` está usando el grupo de middleware `web` o tiene todos los middleware de sesión aplicados:

```
Route::post('user/profile', function () {  
    // Validate the request...  
  
    return back()->withInput();  
});
```

Redireccionar a rutas con nombre

Cuando llama al helper `redirect` sin parámetros, se devuelve una instancia de `Illuminate\Routing\Redirector`, lo que le permite llamar a cualquier método en la instancia `Redirector`. Por ejemplo, para generar una respuesta `RedirectResponse` a una ruta determinada, puede utilizar el método `route`:

```
return redirect()->route('login');
```

Si su ruta tiene parámetros, puede pasarlo como segundo argumento al método `route`:

```
// For a route with the following URI: profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

Rellenar parámetros mediante modelos Eloquent

Si está redirigiendo a una ruta con un parámetro "ID" que está siendo traída desde un modelo Eloquent, simplemente puede pasar el modelo mismo. El ID se extraerá automáticamente:

```
// For a route with the following URI: profile/{id}

return redirect()->route('profile', [$user]);
```

Si desea personalizar el valor que se coloca en el parámetro de ruta, debe sobreescibir el método `getRouteKey` en su modelo Eloquent:

```
/**
 * Get the value of the model's route key.
 *
 * @return mixed
 */
public function getRouteKey()
{
    return $this->slug;
}
```

Redireccionar a acciones de controladores

También puede generar redirecciones a las [acciones del controlador](#). Para ello, pase el nombre del controlador y de la acción al método `action`. Recuerde que no necesita especificar el *namespace* completo al controlador ya que el `RouteServiceProvider` de Laravel configurará automáticamente el *namespace* del controlador base:

```
return redirect()->action('HomeController@index');
```

Si la ruta del controlador requiere parámetros, puede pasarlos como segundo argumento al método `action`:

```
return redirect()->action(
    'UserController@profile', ['id' => 1]
);
```

Redireccionar con datos de sesión *flash*

La redirección a una nueva URL y [flash de los datos a la sesión](#) se hacen generalmente al mismo tiempo. Normalmente, esto se hace después de realizar con éxito una acción como cuando se muestra un mensaje de éxito en la sesión. Para mayor comodidad, puede crear una instancia `RedirectResponse` y los datos *flash* a la sesión en una sola cadena de métodos fluidos:

```
Route::post('user/profile', function () {
    // Update the user's profile...

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

Después de redirigir al usuario, puede mostrar el mensaje de [sesión](#). Por ejemplo usando [Sintaxis Blade](#):

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

Otros tipos de respuestas

El helper `response` se puede utilizar para generar otros tipos de instancias de respuesta. Cuando se llama al helper `response` sin argumentos, se devuelve una implementación del [contrato Illuminate\Contracts\Routing\ResponseFactory](#). Este contrato proporciona varios métodos útiles para generar respuestas.

Respuestas de vistas

Si necesita control sobre el estado y los encabezados de la respuesta pero también necesita devolver una [vista](#) como contenido de la respuesta, debe utilizar el método [view](#) :

```
return response()
    ->view('hello', $data, 200)
    ->header('Content-Type', $type);
```

Por supuesto, si no necesita pasar un código de estado HTTP personalizado o encabezados personalizados, debe utilizar el helper global [view](#).

Respuestas JSON

El método [json](#) fijará automáticamente el encabezado [Content-Type](#) a [aplicación/json](#), así como convertirá el *array* dado a JSON usando la función PHP [json_encode](#):

```
return response()->json([
    'name' => 'Abigail',
    'state' => 'CA'
]);
```

Si desea crear una respuesta JSONP, puede utilizar el método [json](#) en combinación con el método [withCallback](#):

```
return response()
    ->json(['name' => 'Abigail', 'state' => 'CA'])
    ->withCallback($request->input('callback'));
```

Descargas de archivos

El método [download](#) se puede utilizar para generar una respuesta que obligue al navegador del usuario a descargar el archivo en la ruta dada. El método [download](#) acepta un nombre de archivo como segundo argumento del método, que determinará el nombre del archivo que verá el usuario que lo descarga. Finalmente, puede pasar un *array* de cabeceras HTTP como tercer argumento al método:

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);

return response()->download($pathToFile)->deleteFileAfterSend(true);
```

Symfony HttpFoundation, que administra las descargas de archivos, requiere que el archivo descargado tenga un nombre de archivo ASCII.

Respuesta de archivos

El método `file` se puede utilizar para mostrar un archivo, como una imagen o PDF, directamente en el navegador del usuario en lugar de iniciar una descarga. Este método acepta la ruta de acceso al archivo como su primer argumento y *array* de cabeceras como su segundo parámetro:

```
return response()->file($pathToFile);

return response()->file($pathToFile, $headers);
```

Macros para respuestas

Si desea definir una respuesta personalizada que pueda reutilizar en una variedad de rutas y controladores, puede utilizar el método `macro` en la *facade* `Respuesta`. Por ejemplo, desde el método `boot` de un proveedor de servicios:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Response;

class ResponseMacroServiceProvider extends ServiceProvider
{
    /**
     * Register the application's response macros.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function ($value) {
            return Response::make(strtoupper($value));
        });
    }
}
```

La función `macro` acepta un nombre como primer argumento, y un *Closure* como segundo. El *Closure* del *macro* se ejecutará cuando se llame al nombre de la macro desde una implementación de `ResponseFactory` o el helper `response`:

```
return response()->caps('foo');
```

Vistas

- # **Crear vistas**
- # **Pasar datos a vistas**
 - # Compartir datos en todas las vistas
- # **Composers de Vistas (compositores)**

Crear vistas

Para saber más sobre como escribir plantillas en Blade, comprobar la documentación completa de Blade.

Las vistas contienen el HTML que se sirve por cualquier aplicación y separa la lógica del controlador/aplicación de la lógica de presentación. Las vistas se almacenan en la carpeta `resources/views` . Una vista simple podría ser algo como esto:

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Puesto que esta vista se almacena en `resources/views/greetings.blade.php` , se retornará utilizando la función global `view` de este modo:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

Como se puede observar, el primer argumento pasado al helper `view` corresponde al nombre del archivo ubicado en el directorio `resources/views`. El segundo parámetro es un array de datos que debe estar presente en la vista. En este caso, se le está pasando la variable `name`, la cual se muestra en la vista utilizando la [sintaxis Blade](#).

Por su puesto, las vistas pueden estar anidadas dentro de cualquier sub-directorio de la carpeta `resources/views`. La notación de "puntos" o "dot notation" se utiliza para hacer referencia a vistas anidadas. Por ejemplo, si una vista está almacenada en `resources/views/admin/profile.blade.php`, se puede hacer referencia a ella de la siguiente manera:

```
return view('admin.profile', $data);
```

Determinar si existe una vista

Para determinar si una vista existe, se puede utilizar la facade `View`. El método `exists` retornará `true` si la vista existe:

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    //
}
```

Crear la primera vista disponible

Utilizando el método `first` se puede crear la primera vista que se encuentre en un array de vistas. Es útil si una aplicación o paquete permite personalizar o sobrescribir las vistas:

```
return view()->first(['custom.admin', 'admin'], $data);
```

Por supuesto, también se puede llamar a este método a través de la [facade View](#):

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

Pasar datos a vistas

Como se ha visto en ejemplos anteriores, se puede pasar un array de datos a las vistas:

```
return view('greetings', ['name' => 'Victoria']);
```

Al pasar información de esta forma, los datos deben ser un array con pares de clave/valor.

Dentro de la vista, se pueden acceder a estos valores utilizando su clave correspondiente, tal como `<?php echo $key; ?>`. Como alternativa a pasar un array de datos al helper `view`, se pueden pasar piezas individuales de datos a la vista utilizando el método `with`:

```
return view('greeting')->with('name', 'Victoria');
```

Compartir datos en todas las vistas

Ocasionalmente, puede surgir la necesidad de compartir ciertos datos en todas las vistas de la aplicación. Esto se puede hacer utilizando el método de la facade `share`. Por lo general, las llamadas al método `share` se hacen dentro del método `boot` de un service provider. Estas llamadas se pueden añadir en `AppServiceProvider` o generar un service provider independiente para alojarlas:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        View::share('key', 'value');
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Composers de Vistas (compositores)

Los *view composers* son *callbacks* o métodos de una clase que son llamados cuando una vista es renderizada. Si hay datos susceptibles de ser vinculados a una vista cada vez que esta es renderizada, un *view composer* ayudará a organizar esta lógica en una única ubicación.

Para este ejemplo se registrarán los *view composer* en un [service provider](#). Se utilizará la *facade* `View` para acceder a la implementación subyacente del contrato `Illuminate\Contracts\View\Factory`. Laravel no incluye un directorio predeterminado para los

view composer. Así que se deja a libertad el programador de organizarlos como desee. Por ejemplo, se podría crear un directorio [app/Http/ViewComposers](#) :

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        View::composer(
            'profile', 'App\Http\ViewComposers\ProfileComposer'
        );

        // Using Closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Si se crea un nuevo *service provider* que almacene los registros de *view composers*, será necesario añadirlo al array de `providers` en el archivo de configuración `config/app.php`.

Luego de registrar el *composer*, el método `ProfileComposer@compose` se ejecutara cada vez que la vista `profile` esté renderizada. Por lo tanto, se define la clase del *composer*:

```
<?php

namespace App\Http\ViewComposers;

use Illuminate\View\View;
use App\Repositories\UserRepository;

class ProfileComposer
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new profile composer.
     *
     * @param UserRepository $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies automatically resolved by service container...
        $this->users = $users;
    }

    /**
     * Bind data to the view.
     *
     * @param View $view
     * @return void
     */
    public function compose(View $view)
    {
        $view->with('count', $this->users->count());
    }
}
```

Justo antes de que la vista sea renderizada, los *composers* del método `compose` se llaman desde la instancia `Illuminate\Contracts\View\View`. Se puede utilizar el método `with` para enlazar datos con la vista.

Todos los *view composers* se resuelven a través del service container, así que se pueden añadir tantas dependencias como se necesiten en el constructor del *composer*.

Asociar un *composer* a varias vistas

Se puede conectar un *view composer* a múltiples vistas al mismo tiempo pasando un array de las vistas como primer argumento del método `composer`:

```
View::composer(  
    ['profile', 'dashboard'],  
    'App\Http\ViewComposers\MyViewComposer'  
)
```

El método `composer` acepta el carácter `*` como comodín, permitiendo adjuntar un *composer* a todas las vistas:

```
View::composer('*', function ($view) {  
    //  
});
```

*View creators

Los View **creators** son muy similares a los *view composers*; sin embargo se ejecutan inmediatamente después de que una vista se instancie en lugar de esperar a que esté a punto de renderizar. Para registrar un *view creator*, simplemente se utiliza el método `creator`:

```
View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');
```

Generación de URL

Introducción

Conceptos básicos

Generar URLs básicas

Acceder a la URL actual

URL de rutas con nombre

URLs a acciones de controladores

Valores por defecto

Introducción

Laravel incorpora varias funciones de asistencia en la generación de URLs. Por supuesto, son principalmente útiles para la generación de enlaces en las vistas y respuestas API, o para generar redirecciones a otra parte de la aplicación.

Conceptos Básicos

Generar URLs básicas

La función `url` se puede utilizar para generar URLs de la aplicación. Estas URLs utilizarán directamente el esquema HTTP o HTTPS y host de la petición:

```
$post = App\Post::find(1);

echo url("/posts/{$post->id}");

// http://example.com/posts/1
```

Acceder a la URL actual

Si no se especifica un directorio a la función `url`, se retornará una instancia de `Illuminate\Routing\UrlGenerator`, permitiendo acceder a información sobre la URL actual:

```
// Get the current URL without the query string...
echo url()->current();

// Get the current URL including the query string...
echo url()->full();

// Get the full URL for the previous request...
echo url()->previous();
```

Cada uno de estos métodos se puede acceder también a través de `URL` como [facade](#):

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

URL de rutas con nombre

La función `route` genera URLs a rutas con nombre. Las rutas con nombre permiten generar URLs desacopladas de la URL definida en la ruta. Por lo tanto, si la URL de la ruta cambia, no es necesario modificar las llamadas a la función `route`. Por ejemplo, imaginar una aplicación que contiene la siguiente ruta:

```
Route::get('/post/{post}', function () {
    //
})->name('post.show');
```

Para generar una URL a esta ruta se usaría la función `route` de este modo:

```
echo route('post.show', ['post' => 1]);

// http://example.com/post/1
```

A menudo se generarán URLs utilizando la clave primaria de un [modelo Eloquent](#). Por esta razón, se pueden pasar modelos Eloquent como parámetro. La función `route` extraerá directamente la clave primaria del modelo:

```
echo route('post.show', ['post' => $post]);
```

URLs a acciones de controladores

La función `action` genera una dirección URL para una acción determinada de un controlador. No se necesita pasar el namespace completo del controlador. En vez de eso, hay que pasar el nombre de la clase controller relativo al namespace `App\Http\Controllers` :

```
$url = action('HomeController@index');
```

Si el método del controlador acepta parámetros de ruta, se pueden pasar como segundo argumento a la función:

```
$url = action('UserController@profile', ['id' => 1]);
```

Valores por defecto

Para algunas aplicaciones, se pueden especificar valores por defecto para ciertos parámetros URL. Por ejemplo, imaginar que muchas de las rutas definen un parámetro `{locale}` :

```
Route::get('/{locale}/posts', function () {
    //
})->name('post.index');
```

Es tedioso ir pasando `locale` cada vez que se llama a la función `route`. Se puede utilizar el método `URL::defaults` para definir un valor por defecto para este parámetro que será aplicado siempre durante la petición actual. Se puede llamar a este método desde un [route middleware](#) (middleware de ruta) por lo que se tendrá acceso a la petición actual:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\URL;

class SetDefaultLocaleForUrls
{
    public function handle($request, Closure $next)
    {
        URL::defaults(['locale' => $request->user()->locale]);

        return $next($request);
    }
}
```

Una vez que el valor por defecto para `locale` se ha establecido, no será necesario pasarlo más al generar URLs a través del helper `route`.

Sesiones HTTP

Introducción

- # Configuración
- # Pre-requisitos del driver

Usar la sesión

- # Obtener datos
- # Almacenar datos
- # Datos flash
- # Borrar datos
- # Regenerar el ID de sesión

Añadir drivers de sesión personalizados

- # Implementar el driver
- # Registrar el driver

Introducción

Puesto que las aplicaciones HTTP no poseen un estado, las sesiones proveen un modo de almacenar información sobre el usuario a través de las diferentes peticiones. Laravel incluye una gran variedad de sistemas de sesiones a los que se accede a través de un API unificada. Soporte para sistemas populares como [Memcached](#), [Redis](#) y bases de datos de serie.

Configuración

El archivo de configuración de sesión se encuentra en `config/session.php`. Asegúrese de revisar todas las opciones disponibles en este archivo. Por defecto, Laravel está configurado para utilizar el controlador de sesión `file`, el cual funciona bien para la mayoría de aplicaciones. En aplicaciones en producción, debería considerarse utilizar el controlador `memcached` o `redis` para conseguir un mejor rendimiento.

El la opción de configuración `driver` define el lugar donde se almacenará la información de la sesión para cada petición. Laravel incluye varios drivers:

- `file` - las sesiones se almacenan en `storage/framework/sessions/`.

- `cookie` - las sesiones se almacenan en cookies seguras y encriptadas.
- `database` - las sesiones se almacenan en una base de datos relacional.
- `memcached` / `redis` - las sesiones se almacenan en uno de estos veloces almacenamientos basados en cache.
- `array` - las sesiones se almacenan en un *array* PHP y no se persistirán.

El driver `array` se utiliza en testing y previene la persistencia de los datos de sesión.

Pre-requisitos del driver

Base de datos

Al utilizar el driver `database` (base de datos), será necesario crear una tabla para almacenar los elementos de la sesión. A continuación se muestra el `Schema` para dicha tabla:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->unsignedInteger('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

Se puede utilizar el comando de Artisan `session:table` para generar esta migración:

```
php artisan session:table

php artisan migrate
```

Redis

Antes de utilizar las sesiones de Laravel con Redis, hay que instalar el paquete `predis/predis` (~1.0) a través de Composer. Se puede configurar la conexión con Redis en el archivo de configuración `database`. En el archivo de configuración `session`, se utiliza la opción `connection` para especificar que conexión de Redis se utilizará para la sesión.

Usar la sesión

Obtener datos

Hay dos formas de trabajar con los datos de sesión en Laravel: el *helper* `session` y a través de una instancia `Request`. Primero, revisemos el acceso a la sesión a través de una instancia `Request`, la cual se puede incluir como *sugerencia de tipo* en el método del controlador. Recordar, las dependencias de los métodos de un controlador se inyectan automáticamente a través del service container:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function show(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

Cuando se obtiene un valor de la sesión, se puede pasar un valor por defecto como segundo argumento al método `get`. Este valor por defecto se retornará si la clave especificada no existe en la sesión. También se puede pasar un `Closure` como valor por defecto al método `get` y si la clave solicitada no existe, se ejecutará y retornará su resultado:

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

El helper global session

También se puede utilizar la función global de PHP `session` para obtener y almacenar datos en la sesión. Al llamar al helper `session` con una cadena como único argumento, retornará el valor de esa clave en la sesión. Si se le llama con una pareja de clave / valor, se almacenarán esos valores en la sesión:

```
Route::get('home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Specifying a default value...
    $value = session('key', 'default');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

Hay una pequeña diferencia práctica entre utilizar la sesión a través de una instancia `Request` HTTP y utilizar el helper `session`. Ambos métodos son testables a través del método `assertSessionHas`, el cual está disponible en todos los `test cases`.

Obtener todos los datos de la sesión

Para recuperar toda la información de la sesión, se puede utilizar el método `all`:

```
$data = $request->session()->all();
```

Determinar si un elemento existe en la sesión

Para determinar si un valor está presente en la sesión, se puede utilizar el método `has`. Este método retornará `true` si el valor está presente o `null` si no lo está:

```
if ($request->session()->has('users')) {  
    //  
}
```

Para determinar si un valor está presente en la sesión, incluso si el valor es `null`, se puede utilizar el método `exists`. El método `exists` retornará `true` si el valor está presente:

```
if ($request->session()->exists('users')) {  
    //  
}
```

Almacenar datos

Para guardar datos en la sesión, se utiliza normalmente el método `put` o el helper `session`:

```
// Via a request instance...  
$request->session()->put('key', 'value');  
  
// Via the global helper...  
session(['key' => 'value']);
```

Añadir datos a *arrays* en la sesión

El método `push` se utiliza para incluir un nuevo valor en un elemento de la sesión que sea un *array*. Por ejemplo, si la clave `user.teams` contiene un *array* de nombres, se puede añadir un nuevo valor al *array* así:

```
$request->session()->push('user.teams', 'developers');
```

Obtener & eliminar un elemento

El método `pull` obtendrá y eliminará un elemento de la sesión en un única declaración:

```
$value = $request->session()->pull('key', 'default');
```

Mostrar información

Muchas veces deseara almacenar elementos en la sesión solo para la siguiente petición. Puede hacer esto usando el método `flash`. La información almacenada en sesión utilizando este método solo estará disponible durante la petición HTTP subsecuente y luego sera eliminada. Mostrar informacion es principalmente util para mensajes de corta vida:

```
$request->session()->flash('status', 'Task was successful!');
```

Si se necesita mantener la información *flash* durante varias peticiones, se puede utilizar el método `reflash`, el cual mantendrá los datos *flash* para una petición más. Si únicamente se necesita mantener una información concreta, utilizar el método `keep`:

```
$request->session()->reflash();
```

```
$request->session()->keep(['username', 'email']);
```

Borrar datos

El método `forget` eliminará una porción de información de la sesión. Si desea borrar toda la información, se debe utilizar el método `flush`:

```
$request->session()->forget('key');
```

```
$request->session()->flush();
```

Regenerando el ID de sesion

Regenerar el ID de sesión es una tarea que se suele llevar a cabo para prevenir ataques de fijación de sesión por parte de usuarios maliciosos en la aplicación.

Laravel regenera el ID de sesión de forma automática durante la autenticación si se está utilizando el `LoginController` que lleva incorporado; sin embargo, se puede hacer de forma manual con el método `regenerate`.

```
$request->session()->regenerate();
```

Añadir drivers de sesión personalizados

Implementar el Driver

Un driver de sesión personalizado debe implementar `SessionHandlerInterface`. La interfaz contiene unos pocos métodos que se deben implementar. Un ejemplo de una implementación para MongoDB sería algo así:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionId) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}

}
```

Laravel no incluye ningún directorio para almacenar extensiones. Se pueden establecer donde convengan. En este ejemplo, se ha creado un directorio `Extensiones` para alojar el `MongoSessionHandler`.

Puesto que el propósito de estos métodos no es intuitivo, se van a revisar a continuación:

- El método `open` se puede utilizar en sistemas de almacenamiento de sesión basados en archivos. Puesto que Laravel incluye el controlador de sesión `file`, casi nunca será necesario poner nada en este método. Se puede dejar vacío. Es solo un hecho sobre el pobre diseño de interfaz (del cual discutiremos luego) que PHP requiere para implementar este método.
- El método `close`, como el `open`, puede no tenerse en cuenta. Para muchos de los controladores, este no es necesario.
- El método `read` debe retornar la versión de la cadena de la información de sesión asociada con el `$sessionId` dado. No es necesario hacer ninguna serialización u otra codificación cuando recupera o almacena información de sesión, ya que Laravel se encargará de esto automáticamente.
- El método `write` debe escribir la cadena `$data` asociada con el `$sessionId` en algún sistema de almacenamiento persistente, como MongoDB, Dynamo, etc. De nuevo, no

se debe realizar ninguna serialización – Laravel lo gestionará por nosotros.

- El método `destroy` debe eliminar la información asociada con el `$sessionId` del almacenamiento.
- El método `gc` debe destruir toda la información de sesión que sea anterior al `$lifetime` dado, el cual es un *timestamp* de UNIX. Para sistemas auto-expirables, como *Memcached* y *Redis*, este método debe dejarse vacío.

Registrar el driver

Una vez que se ha implementado el driver, ya se puede registrar en el framework. Para añadir drivers de sesión adicionales a Laravel, se puede utilizar el método `extend` de la [Session facade](#). Se debe llamar al método `extend` desde el método `boot` de un [service provider](#). Se podría hacer desde el `AppServiceProvider` existente o crear uno nuevo:

```

<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionHandler();
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

Una vez que el driver de sesión se ha registrado, se puede utilizar el driver `mongo` del archivo de configuración `config/session.php`.

Validación

- # **Introducción**
- # **Comienzo rápido para la validación**
 - # Definir las rutas
 - # Crear el controlador
 - # Definir la lógica de validación
 - # Mostrar los errores de validación
 - # Nota sobre los campos opcionales
- # **Validación de la solicitud del formulario (Form Request)**
 - # Creación de la solicitud del formulario
 - # Autorización de las solicitudes de formulario
 - # Personalizar los mensajes de error
- # **Crear validadores manualmente**
 - # Redirección automática
 - # Nombrar a los contendores de errores
 - # Hook después de la validación
- # **Gestionar mensajes de error**
 - # Mensajes de error personalizados
- # **Reglas de validación disponibles**
- # **Agregar reglas condicionales**
- # **Validación de Arrays**
- # **Reglas de validación personalizadas**
 - # Utilizar objetos de reglas
 - # Utilizar extensiones

Introducción

Laravel incluye varias propuestas para validar la entrada de datos de su aplicación. Por defecto, la clase base del controlador de Laravel utiliza el *contrato* (*trait*) `ValidatesRequests` el cual provee un método para validar la petición HTTP entrante con una gran variedad de reglas de validación muy potentes.

Comienzo rápido con la validación

Para saber más sobre las características de las potentes reglas de validación, puede echar un vistazo a un ejemplo completo para validar un formulario y mostrar los mensajes de error al usuario.

Definir las rutas

Primero, asumiremos que tenemos definidas las siguientes rutas en el archivo

`routes/web.php` :

```
Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@store');
```

Por supuesto, la ruta `GET` mostrará un formulario al usuario para crear un nuevo post en un blog, mientras que la ruta `POST` almacenará ese artículo en la base de datos.

Crear el controlador

A continuación, veamos un controlador simple que gestione estas rutas. Dejaremos el método `store` vacío por el momento:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```

Escribiendo la lógica de validación

Ya estamos listos para incluir la lógica para validar el nuevo artículo en el método `store`. Para hacer esto, usaremos el método `validate` proporcionado por el objeto

`Illuminate\Http\Request`. Si se pasa la regla de validación, el código continuará ejecutándose normalmente; sin embargo, si la regla falla, se lanzará una excepción y la respuesta apropiada será enviada, automáticamente, de vuelta al usuario. En el caso de una petición HTTP tradicional, se generará una respuesta de redirección, mientras que para peticiones AJAX se enviará una respuesta en formato JSON.

Para entender mejor el método `validate`, veamos el interior del método `store`:

```
/**  
 * Store a new blog post.  
 *  
 * @param Request $request  
 * @return Response  
 */  
  
public function store(Request $request)  
{  
    $validatedData = $request->validate([  
        'title' => 'required|unique:posts|max:255',  
        'body' => 'required',  
    ]);  
  
    // The blog post is valid...  
}
```

Como se puede observar, simplemente se pasamos las reglas de validación deseadas al método `validate`. De nuevo, si la validación falla, será generada una apropiada respuesta de forma automática. Si la validación pasa, el controlador continuará ejecutándose con normalidad.

Detener la validación en el primer fallo

A veces puede que quiera deterner la validación en curso en un atributo después del primer error de validación. Para ello, asigne la regla `bail` al atributo:

```
$request->validate([  
    'title' => 'bail|required|unique:posts|max:255',  
    'body' => 'required',  
]);
```

En este ejemplo, si la regla `unique` en el atributo `title` falla, la regla de validación `max` no es comprobada. Las reglas son validadas en el orden que son asignadas.

Consideraciones sobre los atributos anidados

Si su petición HTTP contiene parámetros "anidados", puede especificarlos en las reglas de validación utilizando la notación de "puntos":

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

Mostrar los errores de validación

Ahora, ¿qué sucede si los parámetros de validación entrantes no pasan las reglas de validación? Como se mencionó anteriormente, Laravel redireccionará automáticamente al usuario a su ubicación anterior. Además, todos los errores de validación serán *flashed* automáticamente a la [sesión](#).

Cuando se habla de flash, es en relación a un modo de guardar los errores en la sesión, y éstos serán eliminados cuando sean accedidos o esa sesión sea renovada.

De nuevo, observe que no teníamos que vincular explícitamente los mensajes de error a la vista en nuestra ruta `GET`. Esto se debe a que Laravel comprobará si hay errores en los datos de la sesión y automáticamente los enlazará a la vista si los datos están disponibles. La variable `$errors` será una instancia de `Illuminate\Support\MessageBag`. Para más información sobre cómo trabajar con este objeto, [eche un vistazo a su documentación](#).

La variable `$errors` está ligada a la vista a través del *middleware* `Illuminate\View\Middleware\ShareErrorsFromSession` el cual es proporcionado por el grupo de *middlewares* `web`. **Cuando este middleware es aplicado, la variable `$errors` estará siempre disponible en sus vistas**, permitiendo asumir convenientemente que la variable `$errors` siempre está definida y se puede ser usada con seguridad.

Así, en el ejemplo, el usuario será redirigido al método `create` de nuestro controlador cuando falle la validación, lo que nos permite mostrar los mensajes de error en la vista:

```

<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->

```

Nota sobre los campos opcionales

Por defecto, Laravel incluye los *middleware* `TrimStrings` y `ConvertEmptyStringsToNull` en la pila global de *middlewares*. Estos *middlewares** son enumerados en la pila por la clase `App\Http\Kernel`. A causa de esto, a menudo, necesitará marcar la solicitud de campos "opcionales" como `nullable` si no quiere que el validador los considere los valores `nulos (null)` como inválidos.

A causa de esto, puede que necesitase colocar la regla `<0>nullable` a los campos "opcionales" si no desea que el validador considere valores `<0>null` como no válidos. Por ejemplo:

```

$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);

```

En este ejemplo, estamos especificando que el campo `publish_at` puede ser `null` o la representación de una fecha válida. Si el modificador `nullable` no es añadido a la definición de la regla, el validador se consideraría `null` como una fecha invalida.

Peticiones AJAX & Validación

En este ejemplo, utilizamos un formulario tradicional para enviar datos a la aplicación. Sin embargo, muchas aplicaciones utilizan peticiones AJAX. Cuando usamos el método `validate` durante una petición AJAX, Laravel no generará automáticamente una respuesta de redirección. En su lugar, genera una respuesta JSON conteniendo todos los errores de validación. Esta respuesta JSON se enviará con un código de estado HTTP 422.

Validación de peticiones (Form Request)

Creación de peticiones de formularios

Para escenarios de validación mas complejos, se pueden crear "peticiones de formularios" ("form request"). Las peticiones de formularios son clases que contienen la lógica de la validación. Para crear una clase del tipo *form request*, utilice el comando Artisan `make:request` desde su consola:

```
php artisan make:request StoreBlogPost
```

La clase generada será guardada en el directorio `app/Http/Requests`. Si el directorio no existe, será creado automáticamente cuando ejecute el comando `make:request`. Agretemos unas pocas reglas de validación al método `rules`:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

Entonces, ¿cómo son evaluadas las reglas de validación? Todo lo que usted necesita hacer es especificar el tipo de la petición en su método controlador. La solicitud entrante del formulario es validada antes de que se llame al método del controlador, lo que significa que no necesita abarrotar su controlador con la lógica de validación:

```

/**
 * Store the incoming blog post.
 *
 * @param StoreBlogPost $request
 * @return Response
 */
public function store(StoreBlogPost $request)
{
    // The incoming request is valid...
}

```

Si la validación falla, se generará automáticamente una respuesta de redirección que envia al usuario de vuelta a su anterior localización. Los errores también se mostrarán "temporalmente" en la sesión así que están para su visualización. Si la petición era una solicitud AJAX, se devolverá al usuario una respuesta HTTP con un código de estado 422 incluyendo una representación en formato JSON de los errores de validación.

Agregando *Hooks* después de las peticiones de formulario

Si se desea agregar un *hook* "posterior" a una petición de formulario (form request), puede usar el método `withValidator`. Este método recibe el validador totalmente construido, permitiéndole llamar a cualquiera de sus métodos antes que las reglas de validación sean realmente evaluadas:

```

/**
 * Configure the validator instance.
 *
 * @param \Illuminate\Validation\Validator $validator
 * @return void
 */
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with this field!');
        }
    });
}

```

Autorizacion de las peticiones de formulario

Las clases `form request` tambien contienen un metodo `authorize`. Dentro de este método, usted puede comprobar si el usuario autenticado tiene realmente el permiso para actualizar un recurso determinado. Por ejemplo, usted puede determinar si un usuario es, realmente, dueño de un comentario de un *blog* que esta intentando actualizar:

```
/**  
 * Determine if the user is authorized to make this request.  
 *  
 * @return bool  
 */  
  
public function authorize()  
{  
    $comment = Comment::find($this->route('comment'));  
  
    return $comment && $this->user()->can('update', $comment);  
}
```

Dado que todos los *Form Requests* extienden de la clase base `Request` de Laravel, se puede utilizar el método `user` para acceder al usuario autenticado. También observe la llamada al método `route` en el ejemplo anterior. Este método le garantiza el acceso a los parámetros de la URI definidos en la ruta que se está siendo llamada, como el parámetro `{comment}` en el siguiente ejemplo:

```
Route::post('comment/{comment}');
```

Si el método `authorize` retorna `false`, es devuelta automáticamente una respuesta HTTP con un estado 403 y el método de su controlador no es ejecutado.

Si su plan es tener una lógica de autorización en otra parte de la aplicación, simplemente retorne `true` en el método `authorize`:

```
/**  
 * Determine if the user is authorized to make this request.  
 *  
 * @return bool  
 */  
public function authorize()  
{  
    return true;  
}
```

Personalizar los mensajes de error

Usted puede personalizar los mensajes de error usados en la petición de formulario sobreescribiendo el método `messages`. Este método debería retornar una matriz de pares de atributos / reglas y sus correspondientes mensajes de error:

```
/**  
 * Get the error messages for the defined validation rules.  
 *  
 * @return array  
 */  
public function messages()  
{  
    return [  
        'title.required' => 'A title is required',  
        'body.required'  => 'A message is required',  
    ];  
}
```

Crear validadores manualmente

Si no desea utilizar el método `validate` dentro de una petición, usted podría crear manualmente, una instancia del validador utilizando el [facade Validator](#). El método `make` del *facade* genera una nueva instancia de validator:

```

<?php

namespace App\Http\Controllers;

use Validator;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                ->withErrors($validator)
                ->withInput();
        }

        // Store the blog post...
    }
}

```

El primer argumento pasado al método `make` son los datos a validar. El segundo argumento es la regla de validación que debería ser aplicada a los datos.

Después de comprobar si la solicitud de validación falló, usted puede usar el método `withErrors` para *flash* los mensajes de error en la sesión. Cuando se usa ese método, la variable `$errors` será compartida automáticamente con sus vistas después de la redirección, permitiéndole mostrarlos fácilmente de nuevo al usuario. El método `withErrors` acepta un objeto `validator`, un `MessageBag` o una `matriz` de PHP.

Redirección automática

Si desea crear una instancia del validador de manera manual y aun así aprovechar el redireccionamiento automático ofrecido por el método `validate` de las peticiones, puede llamar el método `validate` en la instancia de tipo `validator`. De tal manera que si la validación falla, el usuario automáticamente sera redirigido, o en caso de que sea una petición AJAX, se retornara una respuesta en JSON:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Nombrar a los *Error Bags*

Si se tiene múltiples formularios en una sola pagina, es posible querer darle nombre a los errores de `MessageBag`, lo que le permite recuperar los mensajes de error para un formulario específico. Esto se hace pasando un nombre como segundo argumento del método `withErrors` :

```
return redirect('register')
    ->withErrors($validator, 'login');
```

Se puede acceder al nombre de la instancia de `MessageBag` desde la variable `$errors` :

```
{{ $errors->login->first('email') }}
```

Hook después de la validación

El validador también permite adjuntar *callbacks* para que se ejecuten después de que la validación se haya completado. Esto le permite realizar fácilmente una validación adicional e incluso agregar más mensajes de error a la colección de mensajes. Para usarlo, se utiliza el método `after` en una instancia del validador:

```
$validator = Validator::make(...);

$validator->after(function ($validator) {

    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this field!');
    }
});

if ($validator->fails()) {
    //
}
```

Gestionar mensajes de error

Después de llamar al método `errors` en una instancia `Validator`, se recibirá una instancia de `Illuminate\Support\MessageBag`, que tiene una variedad de métodos convenientes para trabajar con los mensajes de error. La variable `$errors` se encuentra disponible en todas las vista como una instancia de la clase `MessageBag`.

Recuperar el primer mensaje de error para un campo

Para recuperar el primer mensaje de error de un campo se utiliza el método `first`:

```
$errors = $validator->errors();

echo $errors->first('email');
```

Recuperar todos los mensajes de error para un campo

Si se desea recuperar un *array* de todos los mensajes para un campo determinado, se utiliza el método `get`:

```
foreach ($errors->get('email') as $message) {
    //
}
```

Si está validando un campo de formulario del tipo *array*, se pueden recuperar todos los mensajes para cada uno de los elementos del mismo utilizando el carácter * :

```
foreach ($errors->get('attachments.*') as $message) {  
    //  
}
```

Recuperar los mensajes de error para todos los campos

Para recuperar todos los mensajes de error se utiliza el método `all` :

```
foreach ($errors->all() as $message) {  
    //  
}
```

Verificar si existen mensajes para un campo

El método `has` determinara si existe algún mensaje de error para el campo determinado:

```
if ($errors->has('email')) {  
    //  
}
```

Mensajes de error personalizados

Si es necesario, se pueden personalizar los mensajes de error de las validaciones en lugar de mostrar los que vienen por defecto. Hay varias maneras de especificar mensajes personalizados. La primera, es pasar los mensajes como tercer argumento del método `Validator::make` :

```
$messages = [  
    'required' => 'The :attribute field is required.',  
];  
  
$validator = Validator::make($input, $rules, $messages);
```

En el ejemplo, se utiliza el *place-holder* `:attribute` que es reemplazado por el nombre del campo que se esta validando. Se pueden utilizar otros tipos de *place-holders* en los mensajes de validación. Por ejemplo:

```
$messages = [
    'same'      => 'The :attribute and :other must match.',
    'size'      => 'The :attribute must be exactly :size.',

    'between'   => 'The :attribute value :input is not between :min - :max.',
    'in'        => 'The :attribute must be one of the following types: :values',
];
];
```

Especificar un mensaje de personalizado para un atributo

Cuando se necesita, es posible personalizar un mensaje de error para un campo específico. Se puede utilizar la notación "dot" o por "puntos". Se indica el nombre del atributo primero, seguido de la regla de validación:

```
$messages = [
    'email.required' => 'We need to know your e-mail address!',
];
];
```

Especificar un mensaje personalizado en los archivos de lenguaje

En la mayoría de los casos, probablemente quiera especificar el mensaje personalizado en el archivo de lenguaje en lugar de pasarlo directamente al [Validator](#). Para hacerlo, se deben agregar los mensajes dentro del array `custom` del archivo de lenguaje ubicado en `resources/lang/xx/validation.php`.

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
];
```

Especificar un atributo personalizado en los archivos de lenguaje

Si se desea cambiar la porción del mensaje de la validación correspondiente a `:attribute` para reemplazarlo con un nombre de atributo personalizado, se puede especificar el mismo en el array `attributes` del archivo de lenguaje `resources/lang/xx/validation.php`:

```
'attributes' => [
    'email' => 'email address',
],
```

Reglas de validación disponibles

A continuación se muestra una lista de todas las reglas de validación disponibles con su función:

Accepted	Digits Between	Nullable
Active URL	Dimensions (Image Files)	Not In
After (Date)	Distinct	Numeric
After Or Equal (Date)	E-Mail	Present
Alpha	Exists (Database)	Regular Expression
Alpha Dash	File	Required
Alpha Numeric	Filled	Required If
Array	Image (File)	Required Unless
Before (Date)	In	Required With
Before Or Equal (Date)	In Array	Required With All
Between	Integer	Required Without
Boolean	IP Address	Required Without All
Confirmed	JSON	Same
Date	Max	Size
Date Equals	MIME Types	String
Date Format	MIME Type By File	Timezone
Different	Extension	Unique (Database)
Digits	Min	URL

accepted

El campo a validar debe contener *yes*, *no*, *1* o *true*. Puede ser útil para validar la aceptación de "Términos de servicio".

active_url

El campo a validar debe contener un registro A o AAAA válido de acuerdo con la función de PHP [dns_get_record](#).

after:date

El campo a validar debe contener un valor posterior a una fecha concreta. Las fechas se pasarán a la función de PHP `strtotime` :

```
'start_date' => 'required|date|after:tomorrow'
```

En lugar de pasar una fecha para ser evaluada por `strtotime`, se puede especificar otro campo con el que comparar la fecha:

```
'finish_date' => 'required|date|after:start_date'
```

afteror_equal:_date

El campo bajo validación debe ser un valor posterior o igual a la fecha dada. Para obtener más información, consulte la regla [after](#).

alpha

El campo a validar debe contener únicamente caracteres alfabéticos.

alpha_dash

El campo a validar debe contener caracteres alfa-numéricos, así como guiones altos y bajos.

alpha_num

El campo a validar debe contener únicamente caracteres alfa-numéricos.

array

El campo a validar debe ser un `array` PHP.

before:date

El campo a validar debe contener una fecha anterior a la fecha dada. Las fechas serán pasadas a la función `strtotime` de PHP.

beforeor_equal:_date

El campo a validar debe contener una fecha anterior a la fecha dada. Las fechas serán pasadas a la función `strtotime` de PHP.

`between:min,max`

El campo a validar debe tener un tamaño entre *min* y *max*. Cadenas, caracteres numéricos, arrays y archivos se evalúan de la misma forma que la regla [size](#) .

`boolean`

El campo a validar debe poder ser transformado a un valor booleano. Los valores aceptados son `true` , `false` , `1` , ```` , `"1"` , `y "0``` .

`confirmed`

El campo a validar debe coincidir con el campo `foo_confirmation` . Por ejemplo, si el campo fuera `password` , se debería proporcionar además un campo `password_confirmation` .

`date`

El campo a validar debe contener una fecha válida de acuerdo con la función de PHP `strtotime` .

`date_equals:date`

El campo a validar debe contener una fecha igual a la fecha dada. Las fechas serán pasadas a la función `strtotime` de PHP.

`date_format:format`

El campo a validar debe cumplir el *format* dado. Se puede usar **either** `date` o `date_format` cuando se valida un campo, no ambos.

`different:field`

El campo a validar debe contener un valor diferente a *field*.

`digits:value`

El campo a validar debe ser numérico y una longitud exacta de *value*.

`digits_between:min,max`

El campo a validar debe tener un tamaño entre *min* y *max*.

`dimensions`

El archivo bajo validación debe ser una imagen que cumpla con las restricciones de dimensión especificadas dentro los parámetros de la regla:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Las restricciones disponibles son: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.

El *ratio* debe representarse como el ancho dividido por la altura. Esto puede especificarse mediante una declaración como `3/2` o un flotante como `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Dado que esta regla requiere varios argumentos, se puede utilizar el método `Rule::dimensions` para construir la regla con fluidez:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxLength(500)->ratio(3 / 2),
    ],
]);
```

distinct

Cuando se trabaja con *arrays*, el campo a validar no debe tener valores duplicados.

```
'foo.*.id' => 'distinct'
```

email

El campo a validar debe contener un valor formateado como una dirección de correo electrónico.

exists:table,column

El campo a validar debe existir en una tabla de la base de datos.

Uso básico de la regla exists

```
'state' => 'exists:states'
```

Especificar un nombre de columna

```
'state' => 'exists:states,abbreviation'
```

Ocasionalmente, se puede necesitar especificar la conexión a base de datos a usar para la query de `exists`. Esto se puede lograr colocando antes del nombre de la tabla el nombre de la conexión, utilizando la sintaxis "dot" o por "punto":

```
'email' => 'exists:connection.staff,email'
```

Si se desea personalizar la *consulta* que se ejecuta con la regla de validación, se puede usar la clase `Rule` para definir la regla. En el ejemplo, se especifican las reglas de validación como un *array* en lugar de utilizar el carácter `|` para delimitarlas:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            $query->where('account_id', 1);
        }),
    ],
]);
```

file

El campo a validar debe ser un archivo cargado con éxito.

filled

El campo a validar no debe estar vacío cuando se encuentra presente.

image

El archivo a validar debe ser una imagen (jpeg, png, bmp, gif o svg)

in:foo,bar,...

El campo a validar debe incluir alguno de los valores listados. Como esta regla a menudo requiere el `implode` de un `array`, `Rule::in` puede usarse para construir la regla con fluidez:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

in_array:anotherfield

En el campo a validar debe existir valores en *anotherfield*.

integer

El campo a validar debe ser un entero.

ip

El campo a validar debe contener una dirección IP.

ipv4

El campo a validar debe contener una dirección IPv4.

ipv6

El campo a validar debe contener una dirección IPv6.

json

El campo a validar debe contener una cadena JSON válida.

max:value

El campo a validar debe ser inferior o igual que un máximo *value*. Cadenas, caracteres numéricos, arrays y archivos se evalúan de la misma forma que la regla [size](#) .

mimetypes:text/plain,...

El campo a validar debe coincidir con uno de los tipos de MIME que se declaren:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

Para determinar el tipo de MIME del archivo cargado, se leerán los contenidos del archivo y el framework intentará adivinar el tipo MIME, que puede ser diferente del tipo MIME proporcionado por el cliente.

mimes:foo,bar,...

El archivo a validar debe contener un tipo MIME que corresponda a una de las extensiones listadas.

Uso básico de la regla MIME

```
'photo' => 'mimes:image/jpeg,image/bmp,image/png'
```

Aunque solo hay que especificar las extensiones, la regla realmente valida utilizando los tipos MIME del archivo leyendo el contenido del archivo y averiguando su tipo MIME.

La lista completa de tipos MIME y sus extensiones correspondientes se puede encontrar en el siguiente enlace: <http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

min:value

El campo a validar debe contener un valor mínimo de *value*. Cadenas de texto, números, arrays y archivos se evalúan del mismo modo que la regla [size](#).

nullable

El campo a validar permite valores `null`. Esto es particularmente útil cuando se validan primitivas, como cadenas y enteros que pueden contener valores `null`.

not_in:foo,bar,...

El campo a validar no debe estar incluido dentro del listado de valores. El método `Rule::notIn` se puede usar para escribir con fluidez la validación:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

numeric

El campo a validar debe ser numérico.

present

El campo bajo validación debe estar presente en los datos de entrada pero puede estar vacío.

regex:pattern

El campo a validar debe coincidir con la expresión regular pasada.

Nota: Cuando se utiliza el patrón `regex`, es posible que sea necesario especificar las reglas en un *array* en lugar de utilizar los delimitadores `|`, especialmente si la expresión regular contiene este carácter.

required

El campo a validar debe estar presente en los datos proporcionados y nunca vacío. El campo se considera "vacío" si se cumple alguna de las siguientes condiciones:

- El valor es `null`.
- El valor es una cadena vacía.
- El valor es un *array* vacío o un objeto `Countable` vacío.
- El valor es un archivo subido sin directorio.

required_if:anotherfield,value,...

El campo a validar debe estar presente y no estar vacío si el campo *anotherfield* es igual a algún *value*.

required_unless:anotherfield,value,...

El campo a validar debe estar presente y no estar vacío a no ser que el campo *anotherfield* sea igual a algún *value*.

required_with:foo,bar,...

El campo a validar debe estar presente y no vacío *únicamente si* alguno de los otros campos están presentes.

required_with_all:foo,bar,...

El campo a validar debe estar presente y no vacío *únicamente si* todos los otros campos están presentes.

required_without:foo,bar,...

El campo a validar debe estar presente y no vacío *únicamente cuando* alguno de los otros campos no están presentes.

required_without_all:foo,bar,...

El campo a validar debe estar presente y no vacío *únicamente cuando* todos campos no estén presentes.

same:field

El campo *field* debe coincidir con el campo a validar.

size:value

El campo a validar debe tener un tamaño *value*. Para cadenas, *value* corresponde al número de caracteres. Para datos numéricos, *value* corresponde con un valor entero. Para un *array*, *size* corresponde al `count` del *array*. Para archivos, *size* corresponde al tamaño en kilobytes.

string

El campo de la validación debe ser una cadena de texto. Si se desea que el campo permita valores `null`, se debe asignar la regla `nullable` al campo.

timezone

El campo a validar debe corresponderse con alguno de los identificadores de zona horaria según la función de PHP `timezone_identifiers_list`.

unique:table,column,except,idColumn

El campo a validar debe contener un valor único en una tabla de la base de datos. Si la opción `columna` no se especifica, se utilizará el nombre del campo.

Especificar un nombre de columna:

```
'email' => 'unique:users,email_address'
```

Seleccionar conexión de base de datos

En ocasiones, puede ser necesario establecer la conexión a utilizar para las consultas realizadas por el Validator en la base de datos. Como en el ejemplo anterior, establecer `unique:users` como una regla de validación utilizará la conexión por defecto para consultar la base de datos. Para reemplazar esto, especificar la conexión y el nombre de la tabla utilizando sintaxis de "puntos" o "dot notation":

```
'email' => 'unique:connection.users,email_address'
```

Forzar una regla `unique` para ignorar un ID:

A veces es necesario ignorar un ID concreto en una comprobación `unique`. Por ejemplo, considerar una pantalla de "actualización de perfil" que incluye el nombre de usuario, e-mail y ubicación. Por supuesto, hay que verificar que la dirección de e-mail es única. Sin embargo, si el usuario solo cambia el campo nombre y no el campo e-mail, no será correcto si se lanza un error de validación porque el usuario es el propietario de esa dirección de e-mail.

Para indicar al validador que ignore el ID de usuario, se utilizará la clase `Rule` para definir la regla de forma fluida. En este ejemplo, se especificarán además las reglas de validación como `array` en lugar de utilizar el carácter `|` para delimitar las reglas:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

Si la tabla usa una clave primaria distinta a `id`, se puede especificar el nombre de la columna llamando al método `ignore`:

```
'email' => Rule::unique('users')->ignore($user->id, 'user_id')
```

Agregar cláusulas `Where` adicionales:

Se pueden especificar además cláusulas adicionales personalizando la consulta utilizando el método `where`. Por ejemplo, añadir una cláusula que verifique que el `account_id` es `1`:

```
'email' => Rule::unique('users')->where(function ($query) {
    return $query->where('account_id', 1);
})
```

url

El campo a validar debe ser una URL valida.

Agregar reglas condicionales

Validar únicamente si está presente

En algunos casos, se pueden ejecutar validaciones a campos **únicamente** si ese campo está presente en los datos de entrada. Para ello, añadir la regla `sometimes` a la lista de reglas:

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

En el ejemplo anterior, el campo `email` será validado únicamente si está presente en el array `$data`.

Si se intenta validar un campo que debe estar siempre presente pero puede estar vacío, revisar esta nota para campos opcionales.

Validación condicional compleja

A veces es necesario agregar reglas de validación basadas en lógica condicional más compleja. Por ejemplo, se puede requerir un campo si otro posee un valor mayor a 100. O, tal vez que un campo posea un valor concreto únicamente cuando otro campo está presente. Añadir estas reglas de validación debería ser sencillo. Primero, crear la instancia `Validator` con las *reglas estáticas* que no cambiarán:

```
$v = Validator::make($data, [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Supongamos que nuestra aplicación es para coleccionistas de video juegos. Si el coleccionista se registra en la aplicación y posee más de 100 juegos, requeriremos que explique por qué tiene tantos juegos. Por ejemplo, tal vez sea una tienda de reventa de juegos, o quizás simplemente disfruten de la colección. Para añadir condicionalmente este requerimiento, se puede utilizar el método `sometimes` de la instancia `Validator`.

```
$v->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

El primer argumento pasado al método `sometimes` será el nombre del campo a validar de forma condicional. El segundo argumento serán las reglas de este campo. El tercer parámetro será un `Closure` que retornará `true` si las reglas deben tenerse en cuenta. Este método hace que crear validaciones condicionales complejas resulte muy sencillo. Se pueden añadir validaciones condicionales para varios campos a la vez:

```
$v->sometimes(['reason', 'cost'], 'required', function ($input) {
    return $input->games >= 100;
});
```

El parámetro `$input` pasado al `Closure` será una instancia de `Illuminate\Support\Fluent` y se puede utilizar para acceder a los datos de entrada y archivos.

Validación de Arrays

Validar un *array* basado en campos de un formulario no debería ser complicado. Se puede usar "dot notation" para validar atributos que sean *array*. Por ejemplo, si la petición HTTP contiene en campo `photos[profile]` , se puede validar de la siguiente forma:

```
$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

También se puede validar cada elemento de un *array*. Por ejemplo, para validar que cada correo electrónico en un campo determinado del *array* es único, se puede hacer lo siguiente:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Del mismo modo, se puede usar el carácter `*` para especificar los mensajes de validación en los archivos de idioma, por lo que es muy fácil utilizar un solo mensaje de validación para campos basados en *array*:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail address',
    ]
],
```

Reglas de validación personalizadas

Usando objetos *Rule*

Laravel incluye varias reglas de validación muy útiles; sin embargo, se pueden especificar reglas propias. Una de los métodos para registrar las reglas de validación personalizadas es usando los *rule objects*. Para generar un nuevo *rule object*, se puede usar el comando `make:rule` de Artisan. Se usara este comando para generar una regla que verifique que una cadena esté en mayúscula. Laravel ubicara la nueva regla en el directorio `app/Rules` :

```
php artisan make:rule Uppercase
```

Una vez que el objeto ha sido creado, se puede definir su comportamiento. Un *rule object* contiene dos métodos: `passes` y `message`. El método `passes` recibe el valor del atributo y el nombre, y retorna `true` o `false` dependiendo si el valor del atributo es válido o no. El método `message` retorna el mensaje de error de la validación que debe ser usado cuando la misma falla:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule
{
    /**
     * Determine if the validation rule passes.
     *
     * @param  string  $attribute
     * @param  mixed   $value
     * @return bool
     */
    public function passes($attribute, $value)
    {
        return strtoupper($value) === $value;
    }

    /**
     * Get the validation error message.
     *
     * @return string
     */
    public function message()
    {
        return 'The :attribute must be uppercase.';
    }
}
```

Por supuesto, se puede usar el helper `trans` desde su método `message` si desea devolver un mensaje de error desde los archivos de traducción:

```
/**  
 * Get the validation error message.  
 *  
 * @return string  
 */  
  
public function message()  
{  
    return trans('validation.uppercase');  
}
```

Una vez que se ha definido una regla, se puede asociar al validador pasando una instancia del *rule object* junto a las otras reglas:

```
use App\Rules\Uppercase;  
  
$request->validate([  
    'name' => ['required', new Uppercase],  
]);
```

Utilizar extensiones

Otra de las maneras de registrar reglas de validación personalizadas es utilizando el método `extend` del [facade Validator](#). Utilizaremos este método en un [service provider](#) para registrar una regla de validación personalizada:

```

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Validator;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function ($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

El *Closure* del validador personalizado recibe cuatro argumentos: el nombre del `$attribute` a validar, el `$value` del atributo, un *array* de los `$parameters` que son pasados a la regla, y una instancia de `Validator`.

También se puede pasar una clase y método al método `extend` en lugar de un *Closure*:

```
Validator::extend('foo', 'FooValidator@validate');
```

Definir el mensaje de error

Hay que definir un mensaje de error para la regla personalizada. Se puede hacer o pasando un mensaje concreto en el *array* o añadiendo una nueva entrada en el archivo de validación de lenguaje. Este mensaje debe incluirse en el primer nivel del *array*, nunca dentro del *array* `custom` , el cual es únicamente para mensajes de error de atributos específicos:

```
"foo" => "Your input was invalid!",  
  
"accepted" => "The :attribute must be accepted.",  
  
// The rest of the validation error messages...
```

Cuando se crea una nueva regla de validación, a veces es necesario definir reemplazos personalizados para los mensajes de error. Esto se puede hacer añadiendo un Validator personalizado como se describe a continuación y llamando al método `replacer` de la *facade Validator* . Esto se podría incluir en el método `boot` de un service provider:

```
/**  
 * Bootstrap any application services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    Validator::extend(...);  
  
    Validator::replacer('foo', function ($message, $attribute, $rule, $parameters) {  
  
        return str_replace(...);  
    });  
}
```

Extensiones implícitas

Por defecto, cuando se intenta validar un atributo que no está presente o contiene un valor vacío como se define en la regla `required` , las reglas de validacion normales, incluidas las extensiones, no se ejecutan. Por ejemplo, la regla `unique` no se ejecutara con un valor `null` :

```
$rules = ['name' => 'unique'];

$input = ['name' => null];

Validator::make($input, $rules)->passes(); // true
```

Para que una regla se ejecute incluso cuando un atributo está vacío, la regla debe presuponer que el atributo es requerido. Para crear una extensión "implícita", utilizar el método

`Validator::extendImplicit()` :

```
Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

Note: Una extensión "implícita" únicamente *implica* que el atributo es requerido.

Si se valida o no un atributo vacío queda a merced del usuario.

Errores & *Logging*

Introducción

Configuración

- # Detalles de errores
- # Almacenamiento de *logs*
- # Niveles de gravedad del *log*
- # Configuración de *Monolog* personalizada

El gestor de excepciones

- # El método *report*
- # Método *render*
- # Excepciones reportables & renderizables

Excepciones HTTP

- # Páginas de error HTTP personalizadas

Logging

Introducción

Al comenzar un nuevo proyecto con Laravel, la gestión de errores y excepciones viene ya configurada. La clase `App\Exceptions\Handler` es donde todas las excepciones activadas por su aplicación se registran y se devuelven al usuario. Profundizaremos en esta clase a lo largo de esta documentación.

Para *logging*, Laravel utiliza la biblioteca [Monolog](#), que proporciona soporte para una variedad de potentes gestores de *log*. Laravel configura varios de estos gestores, lo que le permite elegir entre un único archivo de *log*, archivos de *log* rotativos o escribir la información de errores en el *log* del sistema.

Configuración

Detalles de errores

La opción `debug` en su archivo de configuración `config/app.php` determina la cantidad de información sobre un error se debe mostrar al usuario. De forma predeterminada, esta opción está configurada para respetar el valor de la variable de entorno `APP_DEBUG`, que se almacena en su archivo `.env`.

Para el desarrollo local, debe establecer la variable `APP_DEBUG` de entorno en `true`. En su entorno de producción, este valor siempre debe ser `false`. Si el valor se ajusta a `true` en producción, corre el riesgo de exponer los valores de configuración sensibles a los usuarios finales de la aplicación.

Almacenamiento de *logs*

De serie, Laravel soporta la escritura de *logs* en archivos `individuales`, archivos `diarios`, el `syslog`, y el `errorlog`. Para configurar qué mecanismo de almacenamiento utiliza Laravel, debe modificar la opción de `log` en su archivo de configuración `config/app.php`. Por ejemplo, si desea utilizar archivos de *log* diarios en lugar de un único archivo, debería establecer el valor de `log` en su archivo de configuración de `app` a `daily`:

```
'log' => 'daily'
```

Máximo de archivos diarios de *log*

Cuando se utiliza el modo de registro `daily`, Laravel sólo conserva cinco días de archivos de *log* por defecto. Si desea ajustar el número de archivos retenidos, puede añadir un valor de configuración `log_max_files` a su archivo de configuración `app`:

```
'log_max_files' => 30
```

Niveles de gravedad del *log*

Al usar *Monolog*, los mensajes de registro pueden tener diferentes niveles de gravedad. Por defecto, Laravel guarda la información de cualquier nivel de *log*. Sin embargo, en su entorno de producción, es posible que desee configurar la gravedad mínima que se debe registrar añadiendo la opción `log_level` al archivo de configuración `app.php`.

Una vez configurada la opción, Laravel registrará todos los niveles mayores o iguales a la gravedad especificada. Por ejemplo, un `log_nivel` de `error` registrará mensajes de **error**, **críticos**, **alerta** y **emergencia**:

```
'log_level' => env('APP_LOG_LEVEL', 'error'),
```

Monolog reconoce los siguientes niveles de gravedad - de menos severo a más severo: `debug` , `info` , `notice` , `warning` , `error` , `critical` , `alert` , `emergency` .

Configuración de *Monolog* personalizada

Para tener un control completo sobre la configuración de *Monolog*, se puede utilizar el método `configureMonologUsing` de la aplicación. Este método se debe llamar en el archivo `bootstrap/app.php` justo antes de donde el archivo retorna la variable `$app` :

```
$app->configureMonologUsing(function ($monolog) {
    $monolog->pushHandler(...);
});

return $app;
```

Personalizar el nombre del canal

Por defecto, *Monolog* se instala con un nombre que coincide con el entorno actual, como `production` o `local` . Para cambiar este valor, añada la opción `log_channel` a su archivo de configuración `app.php` :

```
'log_channel' => env('APP_LOG_CHANNEL', 'my-app-name'),
```

El gestor de excepciones

El método *report*

Todas las excepciones se gestionan por la clase `App\Exceptions\Handler` . Esta clase contiene dos métodos: `report` y `render` . Se examinarán en detalle. El método `report` se utiliza para registrar excepciones o enviarlas a un servicio externo como [Bugsnag](#) o [Sentry](#). Por defecto, el método `report` simplemente pasa la excepción a la clase base donde la excepción se añade al *log*. Sin embargo, se pueden añadir al *log* tantas excepciones como se desee.

Por ejemplo, para reportar diferentes tipos de excepciones de modos diferentes, se puede utilizar el operador de comparación `instanceof`:

```
/**  
 * Report or log an exception.  
 *  
 * This is a great spot to send exceptions to Sentry, Bugsnag, etc.  
 *  
 * @param \Exception $exception  
 * @return void  
 */  
  
public function report(Exception $exception)  
{  
    if ($exception instanceof CustomException) {  
        //  
    }  
  
    return parent::report($exception);  
}
```

El helper `report`

A veces es posible que necesite reportar una excepción pero continuar manejando la solicitud actual. El helper `report` le permite notificar rápidamente una excepción utilizando el método `report` del gestor de excepciones sin mostrar una página de error:

```
public function isValid($value)  
{  
    try {  
        // Validate the value...  
    } catch (Exception $e) {  
        report($e);  
  
        return false;  
    }  
}
```

Ignorar excepciones por tipo

La propiedad `$dontReport` del gestor de excepciones contiene un *array* de excepciones que no deben ser registradas. Por ejemplo, las excepciones resultantes de errores 404, así como otros tipos de errores, no se escriben en los ficheros *log*. Puede agregar otros tipos de excepciones a este *array* según sea necesario:

```
/**  
 * A list of the exception types that should not be reported.  
 *  
 * @var array  
 */  
  
protected $dontReport = [  
    \Illuminate\Auth\AuthenticationException::class,  
    \Illuminate\Auth\Access\AuthorizationException::class,  
    \Symfony\Component\HttpKernel\Exception\HttpException::class,  
    \Illuminate\Database\Eloquent\ModelNotFoundException::class,  
    \Illuminate\Validation\ValidationException::class,  
];
```

El método `render`

El método `render` es responsable de convertir una excepción en una respuesta HTTP que debe enviarse al navegador. Por defecto, la excepción se pasa a la clase base que genera una respuesta automática. Sin embargo, se puede comprobar el tipo de la excepción o retornar una respuesta diferente:

```
/**  
 * Render an exception into an HTTP response.  
 *  
 * @param \Illuminate\Http\Request $request  
 * @param \Exception $exception  
 * @return \Illuminate\Http\Response  
 */  
  
public function render($request, Exception $exception)  
{  
    if ($exception instanceof CustomException) {  
        return response()->view('errors.custom', [], 500);  
    }  
  
    return parent::render($request, $exception);  
}
```

Excepciones *reportables* & *renderizables*

En lugar de comprobar el tipo de excepciones en los métodos `report` y `render` del gestor de excepciones, puede definir métodos `report` y `render` directamente en su excepción personalizada. Si estos métodos existen, el framework los llamará automáticamente:

```
<?php

namespace App\Exceptions;

use Exception;

class RenderException extends Exception
{
    /**
     * Report the exception.
     *
     * @return void
     */
    public function report()
    {
        //
    }

    /**
     * Render the exception into an HTTP response.
     *
     * @param \Illuminate\Http\Request
     * @return \Illuminate\Http\Response
     */
    public function render($request)
    {
        return response(...);
    }
}
```

Excepciones HTTP

Algunas excepciones describen códigos de error HTTP desde el servidor. Por ejemplo, un error "page not found" (404), un "unauthorized error" (401) o incluso un error 500 generado manualmente. Para generar una respuesta de este tipo desde cualquier punto de su aplicación, puede utilizar el helper `abort` :

```
abort(404);
```

El helper `abort` lanzará una excepción inmediatamente que será procesada por el gestor de excepciones. Opcionalmente, se puede proporcionar el texto de respuesta:

```
abort(403, 'Unauthorized action.');
```

Páginas de error HTTP personalizadas

Laravel facilita la visualización de páginas de error personalizadas para varios códigos de estado HTTP. Por ejemplo, para personalizar la página de error para el código de estado HTTP 404, crear el archivo `resources/views/errors/404.blade.php`. Este archivo se servirá en todos los errores 404 generados por su aplicación. Los nombres de las vistas de este directorio deben coincidir con el código de estado HTTP al que corresponden. La instancia de `HttpException` planteada por la función `abort` pasará a la vista como variable `$exception` :

```
<h2>{{ $exception->getMessage() }}</h2>
```

Registro – *Logging*

Laravel proporciona una capa de abstracción simple sobre la potente biblioteca [Monolog](#). De forma predeterminada, Laravel está configurado para crear un archivo de registro para su aplicación en el directorio `storage/logs`. Se puede escribir información en los *logs* utilizando la [facade Log](#) :

```

<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Support\Facades\Log;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        Log::info('Showing user profile for user: '.$id);

        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}

```

El *logger* proporciona los ocho niveles de *logging* definidos en [RFC 5424](#): **emergency, alert, critical, error, warning, notice, info y debug**.

```

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);

```

Información contextual

Se puede también pasar un *array* con datos contextuales a los métodos de *log*. Los datos contextuales se formatearán y mostrarán con el mensaje de *log*:

```
Log::info('User failed to login.', ['id' => $user->id]);
```

Acceder a la instancia *Monolog* subyacente

Monolog tiene una gran variedad de gestores adicionales que se pueden utilizar para *logging*. Si es necesario, se puede acceder a la instancia de *Monolog* subyacente que utiliza Laravel:

```
$monolog = Log::getMonolog();
```

Frontend

Plantillas Blade

- # Introducción
- # Herencia de plantillas
 - # Definir una plantilla
 - # Heredar un *layout*
- # Componentes & Slots
- # Mostrar datos
 - # Blade & Frameworks JavaScript
- # Estructuras de control
 - # Estructuras *if*
 - # Instrucción *switch*
 - # Bucles
 - # La variable *loop*
 - # Comentarios
 - # PHP
- # Incluir sub-vistas
 - # Procesar vistas para colecciones
- # Pilas – *stacks*
- # Inyección de servicios
- # Extender Blade
 - # Estructuras *if* personalizadas

Introducción

Blade es un simple pero poderoso motor de plantillas incluido con Laravel. A diferencia de otros populares motores de plantillas para PHP, Blade no limita el uso de código PHP simple en las vistas. Las vistas en Blade se compilan a código PHP y se cachean hasta que son modificadas, básicamente esto se traduce en que Blade añade sobrecarga cero a las aplicaciones. Las vistas en Blade usan la extensión `.blade.php` y normalmente se almacenan en el directorio `resources/views`.

Herencia de plantillas

Definir una plantilla

Dos de los principales beneficios del uso de Blade son la *herencia de plantillas* y las secciones. Para empezar, se va a revisar un sencillo ejemplo. Primero, examinaremos un *layout* "master". Puesto que la mayoría de aplicaciones web mantienen la misma estructura a través de sus diferentes páginas, es conveniente definir este *layout* como una única vista Blade:

```
<!-- Stored in resources/views/layouts/app.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

Como se puede observar, este archivo contiene una estructura HTML típica. Sin embargo, se puede tomar nota de las directivas `@section` y `@yield`. La directiva `@section`, como su nombre indica, define un sección de contenido, mientras que la directiva `@yield` es utilizada para mostrar el contenido de una sección.

Una vez que se tiene definido un *layout* para la aplicación, se puede definir una página hija que hereda de este *layout*.

Heredar un *layout*

Cuando defina una vista hija, utilice la directiva Blade `@extends` para especificar de qué *layout* debe "heredar". Las vistas que extienden un *layout* de Blade pueden injectar contenido en las secciones mediante las directivas `@section`. Recordar, como se ve en el

en el ejemplo anterior, el contenido de estas secciones se mostrará el *layout* utilizando `@yield`:

```
<!-- Stored in resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

En este ejemplo, la sección `sidebar` está utilizando la directiva `@parent` para anexar (más que sobrescribir) contenido al sidebar de la plantilla padre. La directiva `@parent` será reemplazada por el contenido del *layout* cuando se procese la vista.

Contrariamente al ejemplo anterior, esta sección `sidebar` termina con `@endsection` en lugar de `@show`. La directiva `@endsection` definirá únicamente una sección mientras que `@show` definirá y **enlazará inmediatamente** la sección.

Las vistas de Blade pueden ser devueltas desde rutas usando el helper global `view`:

```
Route::get('blade', function () {
    return view('child');
});
```

Componentes & Slots

Los componentes y los slots proporcionan beneficios similares a las secciones y los *layouts*; sin embargo, algunos pueden encontrar el modelo mental de componentes y *slots* más fáciles de entender. Primero, imaginemos un componente de "alerta" reutilizable que sería útil en toda nuestra aplicación:

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

La variable `{{ $slot }}` contendrá lo que deseamos inyectar en el componente. Ahora, para construir este componente, podemos usar la directiva Blade `@component` :

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

A veces es útil definir varios *slots* para un componente. Vamos a modificar nuestro componente de alerta para permitir la inyección de un "title". El contenido de los *slots* con nombre se puede mostrar simplemente "imprimiendo" (echoing) la variable que coincide con su nombre:

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

Ahora, podemos inyectar contenido en el *slot* usando la directiva `@slot`. Cualquier contenido que no esté dentro de una directiva `@slot` pasará al componente en la variable `$slot` :

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

Pasar datos adicionales a los componentes

A veces, es posible que necesite transferir datos adicionales a un componente. Por esta razón, puede pasar una matriz de datos como segundo argumento a la directiva `@componente`. Todos los datos se pondrán a disposición de la plantilla del componente como variables:

```
@component('alert', ['foo' => 'bar'])  
...  
@endcomponent
```

Mostrar datos

Puede mostrar los datos pasados a las vistas de Blade envolviendo la variable entre llaves.

Por ejemplo, por la siguiente ruta:

```
Route::get('greeting', function () {  
    return view('welcome', ['name' => 'Samantha']);  
});
```

Se puede mostrar el contenido de la variable `name` así:

```
Hello, {{ $name }}.
```

Por supuesto, el contenido que se puede mostrar no está limitado a las variables pasadas a la vista. También puede mostrar los resultados de cualquier función PHP. De hecho, poner cualquier código PHP que desee dentro de una declaración de Blade usando `{{}}`:

```
The current UNIX timestamp is {{ time() }}.
```

Las declaraciones Blade `{{ }}` se envían automáticamente a través de la función PHP `htmlspecialchars` para prevenir ataques XSS.

Mostrar datos sin escapar

Por defecto, las sentencias Blade `{{ }}` se envían automáticamente a través de la función PHP `htmlspecialchars` para prevenir ataques XSS. Para forzar una impresión de datos sin escapar, se puede utilizar la siguiente sintaxis:

```
Hello, {!! $name !!}.
```

Sea muy cuidadoso al hacer muestreo de contenido suministrado por los usuarios de su aplicación. Utilice siempre la sintaxis de doble llave para evitar ataques XSS cuando muestre los datos suministrados por el usuario.

Renderizando JSON

A veces puede pasar un *array* a su vista con la intención de renderizarlo como JSON para inicializar una variable JavaScript. Por ejemplo:

```
<script>
  var app = <?php echo json_encode($array); ?>;
</script>
```

Sin embargo, en lugar de llamar manualmente `json_encode`, puede usar la directiva Blade `@json`:

```
<script>
  var app = @json($array);
</script>
```

Blade & Frameworks JavaScript

Ya que muchos frameworks de JavaScript también utilizan "llaves" para indicar que una expresión determinada debe mostrarse en el navegador, se puede utilizar el símbolo `@` para informar al motor de procesamiento de Blade que una expresión debe permanecer intacta. Por ejemplo:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

En este ejemplo, Blade eliminará el símbolo `@`; sin embargo, la expresión `{{ name }}` permanecerá ajena al motor Blade, permitiendo que sea procesada por el framework JavaScript.

La directiva `@verbatim`

Si está mostrando variables JavaScript en una gran parte de su plantilla, puede envolver el HTML en la directiva `@verbatim` para que no tenga que prefijar cada expresión con un símbolo `@`:

```
@verbatim
<div class="container">
    Hello, {{ name }}.
</div>
@endverbatim
```

Estructuras de control

Además de la herencia de plantillas y la visualización de datos, Blade también proporciona accesos directos convenientes para las estructuras de control PHP comunes, tales como sentencias condicionales y bucles. Estos atajos proporcionan una forma muy limpia y concisa de trabajar con las estructuras de control PHP, al mismo tiempo que se mantienen familiarizados con sus contrapartes PHP.

Estructuras `if`

Se pueden construir sentencias `if` usando las directivas `@if`, `@elseif`, `@else` y `@endif`. Estas directivas funcionan idénticamente a sus contrapartes de PHP:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

Para conveniencia, Blade ofrece también una directiva `@unless`:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

Además de las directivas condicionales ya discutidas, las directivas `@isset` y `@empty` pueden ser utilizadas como atajos convenientes para sus respectivas funciones PHP:

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

Shortcuts (atajos) para autentificacion

Las directivas `@auth` y `@guest` pueden utilizarse para determinar rápidamente si el usuario actual está autenticado o es un invitado:

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

Si es necesario, puede especificar el [authentication guard](#) que se debe comprobar al utilizar las directivas `@auth` y `@guest`:

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

Instrucción switch

La instrucción *switch* pueden construirse utilizando las directivas `@switch` , `@case` , `@break` , `@default` y `@endswitch` :

```
@switch($i)
  @case(1)
    First case...
    @break

  @case(2)
    Second case...
    @break

  @default
    Default case...
@endswitch
```

Bucles (*loops*)

Además de las declaraciones condicionales, Blade proporciona directivas simples para trabajar con las estructuras de bucle de PHP. De nuevo, cada una de estas directivas funciona de forma idéntica a sus contrapartes PHP:

```
@for ($i = 0; $i < 10; $i++)
  The current value is {{ $i }}
@endfor

@foreach ($users as $user)
  <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
  <li>{{ $user->name }}</li>
@empty
  <p>No users</p>
@endforelse

@while (true)
  <p>I'm looping forever.</p>
@endwhile
```

Al hacer bucles, puedes usar la variable `loop` para obtener información valiosa sobre el bucle, como si estás en la primera o última iteración.

Al usar bucles, también se puede terminar o saltar la iteración actual:

```
@foreach ($users as $user)
@if ($user->type == 1)
@continue
@endif

<li>{{ $user->name }}</li>

@if ($user->number == 5)
@break
@endif
@endforeach
```

También puede incluir la condición con la declaración de directiva en una línea:

```
@foreach ($users as $user)
@continue($user->type == 1)

<li>{{ $user->name }}</li>

@break($user->number == 5)
@endforeach
```

La variable *loop*

Al hacer un bucle, una variable `$loop` estará disponible dentro de el. Esta variable proporciona acceso a algunos datos útiles como el índice actual y si esta es la primer o la última iteración a través del bucle:

```

@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach

```

Si está en un bucle anidado, puede acceder a la variable del bucle padre `$loop` a través de la propiedad `parent`:

```

@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is first iteration of the parent loop.
        @endif
    @endforeach
@endforeach

```

La variable `$loop` también contiene otras propiedades útiles:

Propiedad	Descripción
<code>\$loop->index</code>	El índice de la iteración del bucle actual (comienza en 0).
<code>\$loop->iteration</code>	La iteración del bucle actual (comienza en 1).
<code>\$loop->remaining</code>	La iteración que falta en el bucle.
<code>\$loop->count</code>	El número total de elementos en el array que se está iterando.
<code>\$loop->first</code>	Si esta es la primera iteración a través del bucle.
<code>\$loop->last</code>	Si esta es la última iteración a través del bucle.
<code>\$loop->depth</code>	El nivel de anidamiento del bucle actual.

Propiedad	Descripción
\$loop->parent	Cuando está en un bucle anidado, la variable de bucle del parent.

Comentarios

Blade también permite definir comentarios en las vistas. Sin embargo, a diferencia de los comentarios HTML, los comentarios Blade no se incluyen en el HTML final de la aplicación:

```
{!-- This comment will not be present in the rendered HTML --}
```

PHP

En algunas situaciones, es útil incrustar código PHP en sus vistas. Puede usar la directiva Blade `@php` para ejecutar un bloque de PHP simple dentro de su plantilla:

```
@php  
//  
@endphp
```

Si bien Blade proporciona esta función, su uso frecuente puede ser una señal de que tiene demasiada lógica incrustada en la plantilla.

Incluir Sub-Vistas

La directiva `@include` le permite incluir una vista Blade desde otra vista. Todas las variables disponibles en la vista padre se pondrán a disposición de la vista incluida:

```
<div>  
@include('shared.errors')  
  
<form>  
  <!-- Form Contents -->  
</form>  
</div>
```

Además que la vista incluida heredará todos los datos disponibles en la vista padre, se puede pasar también un *array* de datos a la vista incluida:

```
@include('view.name', ['some' => 'data'])
```

Por supuesto, si usted intenta `@include` en una vista que no existe, Laravel lanzará un error. Si desea incluir una vista que puede o no estar presente, debe utilizar la directiva `@includeIf`:

```
@includeIf('view.name', ['some' => 'data'])
```

Si se necesita usar `@include` dependiendo de una determinada condición booleana, puede utilizar la directiva `@includeWhen`:

```
@includeWhen($boolean, 'view.name', ['some' => 'data'])
```

Para incluir la primera vista que existe desde un conjunto determinado de vistas, puede utilizar la directiva `@includeFirst`:

```
@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])
```

Debe evitar usar las constantes `__DIR__` and `__FILE__` en las vistas Blade, ya que se referirán a la ubicación de la vista compilada en caché.

Procesar vistas para colecciones

Se pueden combinar bucles e inclusiones en una línea con la directiva de Blade `@each`:

```
@each('view.name', $jobs, 'job')
```

El primer argumento es la vista parcial para representar cada elemento en el *array* o colección. El segundo argumento es el *array* o colección sobre el que se desea iterar, mientras que el tercer argumento es el nombre de la variable que se asignará a la iteración actual dentro de la vista. Así, por ejemplo, si se está iterando sobre un *array* de `jobs`, normalmente se querrá acceder a cada `job` como una variable `job` en la vista parcial. La clave para la iteración actual estará disponible como variable `key` dentro de su vista parcial.

También se puede pasar un cuarto argumento a la directiva `@each`. Este argumento determina la vista que se mostrará si el *array* está vacío.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

Las vistas mostradas mediante `@each` no heredan las variables de la vista padre. Si la vista hijo requiere estas variables, debe utilizar `@foreach` e `@include`.

Stacks

Blade permite añadir elementos a *stacks* (pilas) con nombre y que se pueden renderizar en otra vista o *layout*. Esto es especialmente útil para requerir librerías JavaScript en vistas hijas:

```
@push('scripts')
<script src="/example.js"></script>
@endpush
```

Se pueden añadir tantos elementos al *stack* como se necesite. Para renderizar el contenido completo de un *stack*, únicamente hay que pasar el nombre del mismo a la directiva `@stack`:

```
<head>
  <!-- Head Contents -->

  @stack('scripts')
</head>
```

Inyección de servicios

La directiva `@inject` puede utilizarse para recuperar un servicio desde el [service container](#) de Laravel. El primer argumento pasado a `@inject` es el nombre de la variable en la que se ubicará el servicio, mientras que el segundo argumento es el nombre de la clase o interfaz del servicio que desea resolver:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

Extender Blade

Blade le permite definir sus propias directivas personalizadas utilizando el método `directive`. Cuando el compilador Blade se encuentre con la directiva personalizada, llamará a la llamada de retorno provista con la expresión que contiene la directiva.

El siguiente ejemplo crea una directiva `@datetime ($var)` que formatea una variable `$var`, que debería ser una instancia de `DateTime`:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

Como puede ver, encadenamos el método `format` en cualquier expresión que pase a la directiva. Así, en este ejemplo, el PHP final generado por esta directiva será:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

Después de actualizar la lógica de una directiva Blade, tendrá que borrar todas las vistas almacenadas en caché. Esto se hace utilizando el comando Artisan `view:clear`.

Estructuras *if* personalizadas

La programación de una directiva personalizada es a veces más compleja de lo necesario cuando se definen expresiones condicionales simples y personalizadas. Por esta razón, Blade provee un método `Blade::if` que permite definir rápidamente una directiva condicional propia utilizando *Closures*. Por ejemplo, definamos una condición personalizada que compruebe el entorno de aplicación actual. Podemos hacer esto en el método `boot` de nuestro `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    Blade::if('env', function ($environment) {
        return app()->environment($environment);
    });
}
```

Una vez que la estructura condicional se ha definido, es muy fácil utilizarla en nuestras plantillas:

```
@env('local')
    // The application is in the local environment...
@elseenv('testing')
    // The application is in the testing environment...
@else
    // The application is not in the local or testing environment...
@endenv
```

Localización – *Locale*

Introducción

Definición de cadenas de traducción

- # Utilizando claves – *short keys*
- # Usando cadenas de traducción como claves

Definición de cadenas de traducción

- # Reemplazar Parámetros en cadenas de traducción
- # Pluralización

Sobrescribir archivos de idioma de los paquetes

Introducción

La característica de localización de Laravel provee una forma muy cómoda de obtener cadenas en varios lenguajes, permitiendo soportar múltiples idiomas en una aplicación de forma muy sencilla. Las cadenas de idioma se almacenan en archivos dentro del directorio `resources/lang`. Dentro de este directorio debería haber un subdirectorio para cada idioma soportado por la aplicación:

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

Todos los archivos de idioma únicamente retornan un array de cadenas con clave. Por ejemplo:

```
<?php

return [
  'welcome' => 'Welcome to our application'
];
```

Configuración Regional

El idioma por defecto de la aplicación se almacena en el archivo de configuración `config/app.php`. Por supuesto, se puede modificar este valor para satisfacer las necesidades de la aplicación. Además se puede cambiar el idioma actual en tiempo de ejecución utilizando el método `setLocale` sobre la facade `App`:

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($locale);

    //
});
```

Puede configurar un "idioma de reserva", que se utilizará cuando el idioma activo no contenga una determinada cadena de traducción. Al igual que el idioma predeterminado, el lenguaje de reserva también se configura en el archivo `config/app.php`:

```
'fallback_locale' => 'en',
```

Determinar el idioma actual

Puede utilizar los métodos `getLocale` y `isLocale` en la facade `App` para determinar la localización actual o comprobar si la localización es un valor dado:

```
$locale = App::getLocale();

if (App::isLocale('en')) {
    //
}
```

Definición de cadenas de traducción

Utilizando claves – *short keys*

Normalmente, las cadenas de traducción se almacenan en archivos dentro del directorio `resources/lang`. Dentro de este directorio debería haber un subdirectorio para cada idioma soportado por la aplicación:

```
/resources
  /lang
    /en
      messages.php
    /es
      messages.php
```

Todos los archivos de idioma únicamente retornan un *array* de cadenas con clave. Por ejemplo:

```
<?php

// resources/lang/en/messages.php

return [
  'welcome' => 'Welcome to our application'
];
```

Usando cadenas de traducción como claves

Para aplicaciones con requisitos de traducción muy exigentes, definir cada cadena con una "short key" puede resultar muy confuso cuando se hace referencia a ellas en sus vistas. Por esta razón, Laravel también ofrece soporte para definir cadenas de traducción utilizando la traducción "por defecto" de la cadena como clave.

Los archivos de traducción que utilizan cadenas de traducción como claves se almacenan como archivos JSON en el directorio `resources/lang`. Por ejemplo, si su aplicación tiene una traducción al español, debe crear un archivo `resources/lang/es.json`:

```
{
  "I love programming.": "Me encanta programar."
}
```

Definición de cadenas de traducción

Puede recuperar líneas de los archivos de idioma utilizando el helper `__`. El método `__` acepta el archivo y la clave de la cadena de traducción como su primer argumento. Por ejemplo, para recuperar la cadena `welcome` del archivo `recursos/lang/messages.php`:

```
echo __('messages.welcome');

echo __('I love programming.');
```

Por supuesto, si está utilizando el motor de plantillas Blade, puede utilizar la sintaxis `{{ }}` para mostrar la cadena de traducción o utilizar la directiva `@lang`:

```
{{__('messages.welcome')}}

@lang('messages.welcome')
```

Si la cadena de traducción especificada no existe, la función `__` devolverá la cadena de traducción para esa clave. Por lo tanto, utilizando el ejemplo anterior, la función `__` devolverá `messages.welcome` si la cadena de traducción no existe.

Reemplazar parámetros en cadenas de traducción

Si se desea, se pueden establecer parámetros en las cadenas de traducción. Todos los parámetros contienen el prefijo `:`. Por ejemplo, se puede definir un mensaje de bienvenida con un nombre como parámetro:

```
'welcome' => 'Welcome, :name',
```

Para reemplazar los marcadores de posición cuando se recupera una cadena de traducción, pase una matriz de reemplazos como segundo argumento a la función `__`:

```
echo __('messages.welcome', ['name' => 'dayle']);
```

Si su parámetro contiene todas las letras mayúsculas, o sólo tiene su primera letra en mayúscula, el valor traducido será en mayúsculas y minúsculas:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

Pluralización

La pluralización es un problema complejo, pues diferentes idiomas tienen diferentes reglas de pluralización. Utilizando el carácter "pipe" (tubería o barra vertical), se puede distinguir entre la forma singular y plural de una cadena:

```
'apples' => 'There is one apple|There are many apples',
```

Incluso puede crear reglas de pluralización más complejas que especifiquen cadenas de traducción para múltiples rangos de números:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

Después de definir una cadena de traducción que tenga opciones de pluralización, puede utilizar la función `trans_choice` para recuperar la línea para un "conteo" determinado. En este ejemplo, puesto que el contador es mayor que uno, se retornará la forma plural:

```
echo trans_choice('messages.apples', 10);
```

Sobrescribir archivos de idioma de los paquetes

Algunos paquetes incluyen sus propios archivos de idioma. En lugar de cambiar los archivos centrales del paquete para ajustar estas líneas, puede anularlas colocando archivos en el directorio `resources/lang/vendor/{package}/{locale}` .

Así, por ejemplo, si necesita anular las cadenas de traducción al inglés en `messages.php` para un paquete llamado `skyrim/hearthfire`, debería colocar un archivo de idioma en: `resources/lang/vendor/hearthfire/en/messages.php`. Dentro de este fichero, sólo debe definir las cadenas de traducción que desea anular. Cualquier cadena de traducción que no sobreescriba se cargará desde los archivos de idioma originales del paquete.

Scaffolding JavaScript & CSS

Introducción

Escribir CSS

Escribir JavaScript

 # Escribiendo componentes Vue

 # Usando React

Introducción

Si bien Laravel no dicta qué preprocesadores JavaScript o CSS utilizar, sí proporciona un punto de partida básico utilizando [Bootstrap](#) y [Vue](#) que será útil para muchas aplicaciones. Por defecto, Laravel usa [NPM](#) para instalar ambos paquetes frontend.

CSS

[Laravel Mix](#) proporciona una API limpia y expresiva sobre la compilación de SASS o Less, que son extensiones de CSS simple que añaden variables, mezclas y otras características poderosas que hacen que trabajar con CSS sea mucho más agradable. En este documento, discutiremos brevemente la compilación CSS en general; sin embargo, debe consultar la documentación completa de [Laravel Mix](#) para obtener más información sobre la compilación de SASS o Less.

JavaScript

Laravel no requiere que utilice un framework o biblioteca de JavaScript específico para construir sus aplicaciones. De hecho, no tiene que usar JavaScript. Sin embargo, Laravel incluye algunos scaffolding básicos para facilitar el inicio de la escritura de JavaScript moderno usando la biblioteca [Vue](#). Vue proporciona una API expresiva para construir aplicaciones JavaScript robustas usando componentes. Al igual que con CSS, podemos usar Laravel Mix para compilar fácilmente componentes JavaScript en un solo archivo listo para el navegador.

Eliminar el *scaffolding* de frontend

Si desea retirar el *scaffolding* frontend de su aplicación, puede utilizar el comando Artisan `preset`. Este comando, cuando se combina con la opción `none`, eliminará el *scaffolding* Bootstrap y Vue de su aplicación, dejando sólo un archivo SASS en blanco y algunas librerías comunes de utilidades JavaScript:

```
php artisan preset none
```

Escribiendo CSS

El archivo `package.json` de Laravel incluye el paquete `bootstrap-sass` para ayudarle a empezar a crear prototipos del frontend de su aplicación usando Bootstrap. Sin embargo, no dude en añadir o eliminar paquetes del archivo `package.json` según sea necesario para su propia aplicación. Usted no está obligado a utilizar el framework Bootstrap para construir su aplicación Laravel - simplemente se proporciona como un buen punto de partida para aquellos que elijan utilizarlo.

Antes de compilar su CSS, instale las dependencias frontend de su proyecto utilizando [Node package manager \(NPM\)](#):

```
npm install
```

Una vez instaladas las dependencias usando `npm install`, puede compilar sus archivos SASS a CSS plano usando [Laravel Mix](#). El comando `npm run dev` procesará las instrucciones en su archivo `webpack.mix.js`. Normalmente, su CSS compilado se ubicará en el directorio `public/css`:

```
npm run dev
```

Por defecto el archivo `webpack.mix.js` incluido con Laravel compilará el archivo SASS `resources/assets/sass/app.scss`. Este archivo `app.scss` importa un archivo de variables SASS y carga Bootstrap, que proporciona un buen punto de partida para la mayoría de las aplicaciones. Siéntase libre de personalizar el archivo `app.scss` si usted lo desea o incluso utilizar un pre-procesador completamente diferente: [configuring Laravel Mix](#).

Escribiendo JavaScript

Todas las dependencias JavaScript requeridas por su aplicación se pueden encontrar en el archivo `package.json` en el directorio raíz del proyecto. Este archivo es similar al archivo `composer.json` excepto que especifica dependencias JavaScript en lugar de dependencias PHP. Puede instalar estas dependencias utilizando [Node package manager \(NPM\)](#):

```
npm install
```

Por defecto, el archivo `package.json` incluye algunos paquetes como `vue` y `axios` para ayudarle a empezar a crear su aplicación JavaScript. Siéntase libre de agregar o quitar paquetes necesarios para su aplicación del archivo `package.json`.

Una vez instalados los paquetes, puede usar el comando `npm run dev` para [compilar sus recursos](#). Webpack es un empaquetador de módulos para aplicaciones JavaScript modernas. Cuando ejecuta el comando `npm run dev`, Webpack ejecutará las instrucciones en su archivo `webpack.mix.js`:

```
npm run dev
```

De forma predeterminada, el archivo Laravel `webpack.mix.js` compila su SASS y el archivo `resources/assets/js/app.js`. Dentro del archivo `app.js` puede registrar sus componentes Vue o, si prefiere un framework diferente, configurar su propia aplicación JavaScript. Su JavaScript compilado se ubicará normalmente en el directorio `public/js`.

El archivo `app.js` cargará el archivo `resources/assets/js/bootstrap.js` que inicia y configura Vue, Axios, jQuery y todas las demás dependencias JavaScript. Si tiene dependencias JavaScript adicionales para configurar, puede hacerlo en este archivo.

Escribiendo componentes Vue

Por defecto, las aplicaciones de Laravel nuevas contienen un componente Vue `ExampleComponent.vue` situado en el directorio `resources/assets/js/components`. El archivo `ExampleComponent.vue` es un ejemplo de [componente vue de archivo simple](#) que define su plantilla JavaScript y HTML en el mismo archivo. Los componentes de un solo archivo proporcionan un enfoque muy conveniente para la creación de aplicaciones basadas en JavaScript. El componente de ejemplo está registrado en su archivo `app.js`:

```
Vue.component('example-component',
    require('./components/ExampleComponent.vue')
);
```

Para utilizar el componente de su aplicación, simplemente puede colocarlo en una de sus plantillas HTML. Por ejemplo, después de ejecutar el comando Artisan `make:auth` para crear las vistas de autenticación y registro de su aplicación, podría colocar el componente en la plantilla Blade `home.blade.php` :

```
@extends('layouts.app')

@section('content')
<example-component></example-component>
@endsection
```

Recuerde, debería ejecutar el comando `npm run dev` cada vez que cambie un componente Vue. O bien, puede ejecutar el comando `npm run watch` para monitorear y recompilar automáticamente sus componentes cada vez que se modifiquen.

Por supuesto, si está interesado en aprender más sobre la escritura de componentes Vue, debe leer la [Documentación Vue](#), que proporciona una visión general completa y fácil de leer de todo el framework Vue.

Usar React

Si usted prefiere usar React para construir su aplicación JavaScript, Laravel hace que sea fácil intercambiar el *scaffolding* Vue con el *scaffolding* React. En cualquier aplicación de Laravel nueva, puede usar el comando `preset` con la opción `react` :

```
php artisan preset react
```

Este comando único eliminará el *scaffolding* Vue y lo reemplazará con el *scaffolding* React, incluyendo un componente de ejemplo.

Compilando *Assets* (Laravel Mix)

- # Introducción
- # Instalación & configuración
- # Ejecutar Mix
- # Trabajar con hojas de estilo
 - # Less
 - # Sass
 - # Stylus
 - # PostCSS
 - # CSS simple
 - # Procesamiento de URL
 - # Mapas de fuentes – *Source maps*
- # Trabajar con JavaScript
 - # Extracción del *Vendor*
 - # React
 - # Vanilla JS
 - # Configuración personalizada de Webpack
- # Copiar archivos & directorios
- # Versionado/evitar caché
- # Recarga Browsersync
- # Variables de entorno
- # Notificaciones

Introducción

[Laravel Mix](#) proporciona una API fluida para definir los pasos de compilación de Webpack para su aplicación Laravel utilizando varios preprocesadores comunes de CSS y JavaScript. Encadenando métodos, puede definir con fluidez su *asset pipeline*. Por ejemplo:

```
mix.js('resources/assets/js/app.js', 'public/js')  
.sass('resources/assets/sass/app.scss', 'public/css');
```

Si alguna vez ha estado confundido y abrumado sobre cómo comenzar con la compilación de assets y Webpack, le encantará Laravel Mix. Sin embargo, no es necesario utilizarlo para el desarrollo de la aplicación. Por supuesto, puede usar cualquier herramienta para el tratamiento de assets que desee, o incluso ninguna.

Instalación & configuración

Instalar Node

Antes de activar Mix, primero debe asegurarse de que Node.js y NPM estén instalados en su máquina.

```
node -v  
npm -v
```

Por defecto, Laravel Homestead incluye todo lo que necesita; sin embargo, si no está utilizando Vagrant, puede instalar fácilmente la última versión de Node y NPM usando instaladores gráficos simples desde [su página de descarga](#).

Laravel Mix

El único paso restante es instalar Laravel Mix. Dentro de una nueva instalación de Laravel, encontrará un archivo `package.json` en la raíz de la estructura de su directorio. El archivo predeterminado `package.json` incluye todo lo que necesita para comenzar. Este archivo es similar a `composer.json`, excepto que define las dependencias de Node en lugar de PHP. Se pueden instalar estas dependencias ejecutando:

```
npm install
```

Ejecutar Mix

Mix es una capa de configuración por encima de [Webpack](#), por lo que para ejecutar las tareas de Mix solo es necesario ejecutar uno de los scripts de NPM que se incluye con el archivo Laravel predeterminado `package.json`:

```
// Run all Mix tasks...
npm run dev

// Run all Mix tasks and minify output...
npm run production
```

Observar cambios en assets

El comando `npm run watch` continuará ejecutándose en su terminal y observará todos los ficheros relevantes en busca de cambios. Webpack luego recompilará automáticamente estos *assets* cuando detecte un cambio:

```
npm run watch
```

Puede encontrar que en ciertos entornos Webpack no se actualiza cuando se modifican sus archivos. Si este es el caso en su sistema, considerar el comando `watch-poll`:

```
npm run watch-poll
```

Trabajar con hojas de estilo

El archivo `webpack.mix.js` es el punto de entrada para la compilación de todos los *assets*. Piense en ello como un contenedor de configuración ligera de Webpack. Las tareas Mix pueden encadenarse juntas para definir exactamente cómo deben compilarse los *assets*.

Less

El método `Less` se puede usar para compilar `Less` en CSS. Vamos a compilar nuestro archivo principal `app.less` a `public/css/app.css`.

```
mix.less('resources/assets/less/app.less', 'public/css');
```

Se pueden realizar varias llamadas al método `less` para compilar varios archivos:

```
mix.less('resources/assets/less/app.less', 'public/css')
    .less('resources/assets/less/admin.less', 'public/css');
```

Si desea personalizar el nombre del archivo compilado CSS, puede pasar una ruta completa del archivo como el segundo argumento para el método `less` :

```
mix.less('resources/assets/less/app.less', 'public/stylesheets/styles.css');
```

Si necesita anular las [opciones del plug-in Less subyacentes](#), puede pasar un objeto como tercer argumento para `mix.less()` :

```
mix.less('resources/assets/less/app.less', 'public/css', {  
    strictMath: true  
});
```

Sass

El método `sass` te permite compilar [Sass](#) en CSS. Puede usar el método de esta manera:

```
mix.sass('resources/assets/sass/app.scss', 'public/css');
```

De nuevo, al igual que el método `less`, puede compilar varios archivos Sass en sus respectivos archivos CSS e incluso personalizar el directorio de salida del CSS resultante:

```
mix.sass('resources/assets/sass/app.sass', 'public/css')  
.sass('resources/assets/sass/admin.sass', 'public/css/admin');
```

Se pueden proporcionar [opciones del plug-in Node-Sass](#) como tercer argumento:

```
mix.sass('resources/assets/sass/app.sass', 'public/css', {  
    precision: 5  
});
```

Stylus

Similar a Less y Sass, el método `stylus` le permite compilar [Stylus](#) en CSS:

```
mix.stylus('resources/assets/stylus/app.styl', 'public/css');
```

También puede instalar complementos adicionales de Stylus, como [Rupture](#). Primero, instale el complemento en cuestión a través de NPM (`npm install rupture`) y luego lo requiere en su llamada a `mix.styles()` :

```
mix.styles('resources/assets/stylus/app.styl', 'public/css', {
    use: [
        require('rupture')()
    ]
});
```

PostCSS

[PostCSS](#), una herramienta poderosa para transformar su CSS, se incluye con Laravel Mix de serie. De forma predeterminada, Mix aprovecha el popular *plug-in* [Autoprefixer](#) para aplicar automáticamente todos los prefijos de CSS necesarios. Sin embargo, puede agregar complementos adicionales que sean apropiados para su aplicación. Primero, instale el complemento deseado a través de NPM y luego hágalo en su archivo `webpack.mix.js` :

```
mix.sass('resources/assets/sass/app.scss', 'public/css')
    .options({
        postCss: [
            require('postcss-css-variables')()
        ]
    });
});
```

CSS simple

Si simplemente desea concatenar algunas hojas de estilos CSS simples en un solo archivo, puede usar el método `styles` .

```
mix.styles([
    'public/css/vendor/normalize.css',
    'public/css/vendor/videojs.css'
], 'public/css/all.css');
```

Procesamiento URL

Debido a que Laravel Mix se basa en Webpack, es importante comprender algunos conceptos de Webpack. Para la compilación de CSS, Webpack reescribirá y optimizará cualquier llamada `url()` dentro de sus hojas de estilo. Si bien esto inicialmente puede sonar extraño, es una funcionalidad increíblemente poderosa. Imagine que queremos compilar Sass que incluye una URL relativa a una imagen:

```
.example {  
    background: url('../images/example.png');  
}
```

Las rutas absolutas para cualquier `url()` determinada se excluirán de la reescritura de URL. Por ejemplo, `url('/images/thing.png')` o `url('http://example.com/images/thing.png')` no será modificado.

De forma predeterminada, Laravel Mix y Webpack encontrarán `example.png`, lo copiarán en su carpeta `public/images` y luego reescribirá la `url()` dentro de su hoja de estilo generada. Como tal, su CSS compilado será:

```
.example {  
    background: url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);  
}
```

Tan útil como puede ser esta característica, es posible que su estructura de carpetas existente ya esté configurada de la manera que desee. Si este es el caso, puede deshabilitar la reescritura de `url()` así:

```
mix.sass('resources/assets/app/app.scss', 'public/css')  
.options({  
    processCssUrls: false  
});
```

Con esta adición a su archivo `webpack.mix.js`, Mix ya no coincidirá con ninguna `url()` o copiará los *assets* en su directorio `public`. En otras palabras, el CSS compilado se verá exactamente como se escribió originalmente:

```
.example {  
    background: url("../images/thing.png");  
}
```

Mapas de fuentes – *Source maps*

Aunque está deshabilitado de forma predeterminada, los *source maps* se pueden activar llamando al método `mix.sourceMaps()` en su archivo `webpack.mix.js`. A pesar de que viene con un costo de compilación/rendimiento, esto proporcionará información de depuración adicional a las herramientas de desarrollo de su navegador al usar recursos compilados.

```
mix.js('resources/assets/js/app.js', 'public/js')
    .sourceMaps();
```

Trabajar con JavaScript

Mix proporciona varias funciones para ayudar a trabajar con sus archivos JavaScript, como la compilación de ECMAScript 2015, la agrupación de módulos, la minificación y simplemente la concatenación de archivos JavaScript simples. Aún mejor, todo esto funciona a la perfección, sin requerir una pizca de configuración personalizada:

```
mix.js('resources/assets/js/app.js', 'public/js');
```

Con esta única línea de código, ahora puede aprovechar:

- Sintaxis ES2015.
- Módulos
- Compilación de archivos `.vue`.
- Minificación para entornos de producción.

Extracción del *Vendor*

Una posible desventaja de agrupar todo el JavaScript específico de la aplicación con las bibliotecas de su *vendor* es que hace que el almacenamiento en caché a largo plazo sea más difícil. Por ejemplo, una sola actualización de su código de aplicación obligará al navegador a volver a descargar todas las bibliotecas de su *vendor*, incluso si no han cambiado.

Si tiene la intención de realizar actualizaciones frecuentes del JavaScript de su aplicación, debería considerar extraer todas sus librerías del *vendor* en su propio archivo. De esta forma, un cambio en el código de su aplicación no afectará el almacenamiento en caché de su gran archivo `vendor.js`. El método `extract` de Mix hace que esto sea muy sencillo:

```
mix.js('resources/assets/js/app.js', 'public/js')
    .extract(['vue'])
```

El método `extract` acepta una matriz de todas las librerías o módulos que desea extraer en un archivo `vendor.js`. Usando el fragmento de arriba como ejemplo, Mix generará los siguientes archivos:

- `public/js/manifest.js` : *El tiempo de ejecución del manifiesto de Webpack*
- `public/js/vendor.js` : *Sus librerías del vendor*
- `public/js/app.js` : *Su código de aplicación*

Para evitar errores de JavaScript, asegúrese de cargar estos archivos en el orden correcto:

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

React

Mix puede instalar automáticamente los *plug-ins* de Babel necesarios para la compatibilidad con React. Para comenzar, reemplace su llamada a `mix.js()` con `mix.react()`:

```
mix.react('resources/assets/js/app.jsx', 'public/js');
```

Mix descargará e incluirá el *plug-in* de Babel `babel-preset-react`.

Vanilla JS

Similar a la combinación de hojas de estilo con `mix.styles()`, también puedes combinar y minificar cualquier cantidad de archivos JavaScript con el método `scripts()`:

```
mix.scripts([
    'public/js/admin.js',
    'public/js/dashboard.js'
], 'public/js/all.js');
```

Esta opción es particularmente útil para proyectos antiguos donde no se requiere compilación de Webpack para su JavaScript.

Una ligera variación de `mix.scripts()` es `mix.babel()`. Su firma de método es idéntica a `scripts`; sin embargo, el archivo concatenado recibirá la compilación de Babel, que traduce cualquier código de ES2015 a *vanilla JavaScript* que todos los navegadores entenderán.

Configuración personalizada de Webpack

Entre bastidores, Laravel Mix hace referencia a un archivo `webpack.config.js` pre-configurado para que pueda comenzar a trabajar lo más rápido posible. En ocasiones, puede necesitar modificar manualmente este archivo. Es posible que tenga un cargador o *plug-in* especial al que se deba hacer referencia, o tal vez prefiera usar Stylus en lugar de Sass. En tales casos, tiene dos opciones:

Fusionar la configuración personalizada

Mix proporciona un método útil `webpackConfig` que le permite combinar cualquier anulación de configuración de Webpack. Esta es una opción particularmente atractiva, ya que no requiere que copie y mantenga su propia copia del archivo `webpack.config.js`. El método `webpackConfig` acepta un objeto, que debe contener cualquier [configuración específica de Webpack](#) que desee aplicar.

```
mix.webpackConfig({
  resolve: {
    modules: [
      path.resolve(__dirname, 'vendor/laravel/spark/resources/assets/js')
    ]
  }
});
```

Archivos de configuración personalizados

Si desea personalizar completamente la configuración de su Webpack, copie el archivo `node_modules/laravel-mix/setup/webpack.config.js` en el directorio raíz de su proyecto. A continuación, apunte todas las referencias de `--config` en su archivo `package.json` al archivo

de configuración recién copiado. Si opta por llevar este enfoque, cualquier actualización futura de del archivo `webpack.config.js` de Mix deberá fusionarse manualmente en su archivo personalizado.

Copiar archivos & directorios

El método `copy` se puede usar para copiar archivos y directorios a ubicaciones nuevas. Puede ser útil cuando un recurso particular dentro de su directorio `node_modules` necesita ser reubicado en su carpeta `public`.

```
mix.copy('node_modules/foo/bar.css', 'public/css/bar.css');
```

Al copiar un directorio, el método `copy` aplanará la estructura del directorio. Para mantener la estructura original del directorio, debe usar el método `copyDirectory` en su lugar:

```
mix.copyDirectory('assets/img', 'public/img');
```

Versionado/evitar caché

Muchos desarrolladores establecen un *timestamp* o *token* único como sufijo para sus archivos compilados para forzar a los navegadores cargar las nuevas copias en lugar de servir copias obsoletas del código. Mix gestiona esto utilizando el método `version`.

El método `version` agregará automáticamente un *hash* único a los nombres de archivo de todos los archivos compilados, lo que permite un almacenamiento en memoria caché más conveniente:

```
mix.js('resources/assets/js/app.js', 'public/js')
    .version();
```

Después de generar el archivo versionado, no sabrá el nombre exacto del archivo. Por lo tanto, debe usar la función global de Laravel `mix` en sus vistas para cargar el asset adecuado. La función `mix` determinará automáticamente el nombre actual del archivo:

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}>
```

Debido a que los archivos versionados generalmente no son necesarios en el desarrollo, puede indicarle al proceso de versiones que solo se ejecute durante `npm run production` :

```
mix.js('resources/assets/js/app.js', 'public/js');

if (mix.inProduction()) {
    mix.version();
}
```

Recarga Browsersync

[BrowserSync](#) puede supervisar automáticamente sus archivos en busca de cambios e injectar sus cambios en el navegador sin necesidad de una actualización manual. Puede habilitar el soporte llamando al método `mix.browserSync()` :

```
mix.browserSync('my-domain.dev');

// Or...

// https://browsersync.io/docs/options
mix.browserSync({
    proxy: 'my-domain.dev'
});
```

Puede pasar una cadena (proxy) u objeto (configuración BrowserSync) a este método. A continuación, inicie el servidor de desarrollo de Webpack con el comando `npm run watch` . Ahora, cuando modifique un script o un archivo PHP, observe cómo el navegador actualiza al instante la página para reflejar sus cambios.

Variables de entorno

Puede injectar variables de entorno en Mix prefijando una clave en su archivo `.env` con `MIX_` :

```
MIX_SENTRY_DSN_PUBLIC=http://example.com
```

Después de que la variable se haya definido en su archivo `.env`, puede acceder a través del objeto `process.env`. Si el valor cambia mientras está ejecutando una tarea `watch`, deberá reiniciar la tarea:

```
process.env.MIX_SENTRY_DSN_PUBLIC
```

Notificaciones

Cuando esté disponible, Mix mostrará automáticamente las notificaciones del sistema operativo para cada paquete. Esto le dará una respuesta instantánea, si la compilación fue exitosa o no. Sin embargo, puede haber instancias en las que prefiera inhabilitar estas notificaciones. Uno de esos ejemplos podría ser activar Mix en su servidor de producción. Las notificaciones se pueden desactivar mediante el método `disableNotifications`.

```
mix.disableNotifications();
```

Seguridad

Autenticación

Introducción

Consideraciones de la base de datos

Guía Rápida de Autenticación

Rutas

Vistas

Autenticación

Obtener el usuario autenticado

Protección de rutas

Intentos de login

Autenticación manual de usuarios

Recordar usuarios

Otros métodos de autenticación

Autenticación HTTP básica

Autenticación HTTP basica sin estado

Autenticación social

Añadir Guards personalizados

Añadir user providers personalizados

El contrato User Provider

El contrato Authenticatable

Eventos

Introducción

¿Quiere comenzar rápido? Simplemente ejecutar `php artisan make:auth` y `php artisan migrate` una aplicación Laravel. A continuación ya se podrá acceder a `http://tu-app.dev/register`. ¡Estos dos comandos generarán todo el *scaffolding* del sistema de autenticación!

Laravel hace que la implementación de la autenticación sea muy simple. De hecho, prácticamente todo viene configurado de serie. El archivo de configuración de autenticación se encuentra en `config/auth.php`, el cual contiene varias opciones bien documentadas para

ajustar el comportamiento de los servicios de autenticación.

Así como el core, las funciones de autenticación de Laravel se basan en "guards" y "providers". Los *guards* definen como se autentican los usuarios para cada petición. Por ejemplo, Laravel incluye un *guard* `session` que mantiene el estado utilizando el almacenamiento de sesión y cookies.

Los *providers* definen como se obtienen los usuarios de la capa de almacenamiento. Laravel incluye soporte para obtener usuarios con Eloquent y el *query builder* de la base de datos. Sin embargo, se pueden definir *providers* adicionales si la aplicación lo necesitara.

¡Todo esto puede sonar confuso ahora! La mayoría de aplicaciones nunca necesitarán modificar la configuración de autenticación por defecto.

Consideraciones de la base de datos

Por defecto, Laravel incluye un [modelo Eloquent](#) `App\User` en el directorio `app`. Este modelo se puede utilizar con el driver de autenticación de Eloquent por defecto. Si la aplicación no utiliza Eloquent, se puede especificar el driver `database` que utiliza el *query builder* de Laravel.

Al construir la estructura de base de datos para el modelo `App\User`, hay que comprobar que la columna *password* contiene al menos 60 caracteres de longitud. Mantener la longitud por defecto de esta columna de 255 caracteres sería una buena opción.

También, verificar que la tabla `users` (o equivalente) contiene una columna `remember_token` de tipo *string* que acepta valores *null* de 100 caracteres. Esta columna se utilizará para almacenar un *token* para los usuarios que seleccionen la opción "recordarme" al identificarse en la aplicación.

Guía rápida de autenticación

Laravel incluye varios controladores pre-desarrollados que se pueden encontrar en el namespace `App\Http\Controllers\Auth`. El controlador `RegisterController` gestiona el registro de nuevos usuarios, `LoginController` gestiona la autenticación, `ForgotPasswordController` se encarga del envío de e-mails para el restablecimiento de contraseñas y `ResetPasswordController` contiene la lógica para restablecer las contraseñas. Cada uno de estos controladores utiliza un *trait* para incluir los métodos necesarios. Para muchas aplicaciones, no será necesario la modificación siquiera de estos controladores.

Rutas

Laravel provee una forma rápida de generar todas las rutas y vistas necesarias para la autenticación en un simple comando:

```
php artisan make:auth
```

Este comando se debe utilizar en aplicaciones nuevas y instalará un *layout*, vistas de registro y login así como rutas para todos los *end-point* de autenticación. Se generará además un `HomeController` para gestionar las peticiones *post-login* de la aplicación.

Vistas

Como se ha mencionado anteriormente, el comando `php artisan make:auth` creará todas las vistas necesarias para la autenticación y las ubicará en el directorio `resources/views/auth`.

El comando `make:auth` creará además el directorio `resources/views/layouts` que contiene un *layout* para la aplicación. Todas estas vistas utilizan Bootstrap CSS, pero se puede personalizar al gusto.

Autenticación

Ahora que se tienen las rutas y vistas configuradas con los controladores de autenticación, se está listo para registrar y autenticar usuarios en la aplicación. Se puede acceder a la aplicación directamente desde el navegador ya que los controladores de autenticación ya tienen la lógica (a través de *traits*) para autenticar usuarios existentes y registrar nuevos en la base de datos.

Personalización de rutas

Cuando un usuario se autentica satisfactoriamente, se le redirecciona a la URI `/home`. Se puede personalizar esta redirección post-autenticación definiendo la propiedad `redirectTo` de `LoginController`, `RegisterController` y `ResetPasswordController`:

```
protected $redirectTo = '/';
```

Si la ruta de redirección precisa de alguna lógica, se puede definir un método `redirectTo` en lugar de la propiedad `redirectTo`:

```
protected function redirectTo()
{
    return '/path';
}
```

El método `redirectTo` se ejecutará preferiblemente sobre el atributo `redirectTo`.

Personalizar el nombre de usuario

Por defecto, Laravel utiliza el campo `email` para la autenticación. Para cambiar esto, se puede definir el método `username` en `LoginController`:

```
public function username()
{
    return 'username';
}
```

Personalización de *guard*

Se puede personalizar el "guard" que se utiliza para autenticar y registrar usuarios. Para comenzar, defina un método `guard` en `LoginController`, `RegisterController` y `ResetPasswordController`. Este método debe retornar una instancia de *Guard*:

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

Personalizar la validación/persistencia

Para modificar los campos del formulario que se requieren al registrar un usuario en la aplicación o personalizar cómo se almacenan los usuarios en la base de datos, se puede modificar la clase `RegisterController`. Esta clase es responsable de validar y crear nuevos usuarios en la aplicación.

El método `validator` de `RegisterController` contiene las reglas de validación para nuevos usuarios. Se puede modificar este método al gusto.

El método `create` de `RegisterController` es responsable de crear un nuevo registro `App\User` en la base de datos utilizando [Eloquent ORM](#). Se puede modificar este método de acuerdo con las necesidades de la base de datos.

Obtener el usuario autenticado

Es posible acceder al usuario autenticado a través de la facade `Auth`:

```
use Illuminate\Support\Facades\Auth;

// Get the currently authenticated user...
$user = Auth::user();

// Get the currently authenticated user's ID...
$id = Auth::id();
```

Alternativamente, una vez que un usuario está autenticado, se puede acceder al mismo a través de una instancia de `Illuminate\Http\Request`. Recordar, las clases incluidas en la firma del método serán inyectadas:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller
{
    /**
     * Update the user's profile.
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        // $request->user() returns an instance of the authenticated user...
    }
}
```

Determinar si el usuario está autenticado

Para determinar si el usuario ha iniciado sesión en la aplicación, se puede utilizar el método `check` de la facade `Auth`, el cual retornará `true` si el usuario está autenticado:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // The user is logged in...
}
```

Incluso si es posible determinar si el usuario está autenticado utilizando el método `check`, normalmente se utilizará un middleware para verificar que el usuario está autenticado antes de permitir el acceso a ciertas rutas o controladores. Para saber más sobre esto puede revisar la documentación en protección de rutas.

Protección de rutas

Los [middleware de ruta](#) se pueden utilizar para permitir el acceso a ciertas rutas únicamente a usuarios autenticados. Laravel incluye un middleware `auth`, definido en `Illuminate\Auth\Middleware\Authenticate`. Puesto que este middleware ya está registrado en el `kernel` HTTP, todo lo que hay que hacer es asignarlo a la definición de una ruta:

```
Route::get('profile', function () {
    // Only authenticated users may enter...
})->middleware('auth');
```

Por supuesto, si se están utilizando [controladores](#), se puede llamar al método `middleware` desde el constructor del controlador en lugar de asignarlo en la definición de la ruta directamente:

```
public function __construct()
{
    $this->middleware('auth');
}
```

Especificar un *Guard*

Al asignar el *middleware* `auth` a una ruta, se puede especificar el *guard* a utilizar para autenticar al usuario. El *guard* especificado corresponderá a una de las claves del array `guards` en el archivo de configuración `auth.php`:

```
public function __construct()
{
    $this->middleware('auth:api');
}
```

Intentos de login

Si está utilizando la clase `LoginController` que incluye Laravel, se incluirá directamente el *trait* `Illuminate\Foundation\Auth\ThrottlesLogins` en el controlador. De forma predeterminada, el usuario no podrá iniciar sesión durante un minuto si no se proporcionan las credenciales correctas después de varios intentos. Este bloqueo es único al nombre de usuario / email y su dirección IP.

Autenticación manual de usuarios

Por supuesto, no es obligatorio utilizar los controladores de autenticación incluidos en Laravel. Se pueden eliminar y gestionar la autenticación de usuarios utilizando las clases de autenticación de Laravel directamente. No hay que alarmarse, ¡es super sencillo!

Se accede a los servicios de autenticación de Laravel a través de la `Facade Auth`, por lo que hay que comprobar que se importa la `Facade Auth` al comienzo de la clase. A continuación se revisará el método `attempt`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @return Response
     */
    public function authenticate()
    {
        if (Auth::attempt(['email' => $email, 'password' => $password])) {
            // Authentication passed...
            return redirect()->intended('dashboard');
        }
    }
}
```

El método `attempt` acepta un *array* de pares clave/valor como primer parámetro. Los valores de este array se utilizarán para encontrar al usuario en la tabla de la base de datos. Así, en el ejemplo anterior, se obtendrá el usuario por el valor de la columna `email`. Si se encuentra el usuario, se comparará la contraseña cifrada almacenada en la base de datos con el valor `password` que se pasó a través del *array*. Nunca se debe cifrar la contraseña del campo `password`, puesto que el framework lo hará automáticamente antes de compararla con el valor cifrado en la base datos. Si las dos contraseñas cifradas coinciden se iniciará una sesión autenticada para el usuario.

El método `attempt` retornará `true` si la autenticación tuvo éxito. De lo contrario, retornará `false`.

El método `intended` redireccionará al usuario a la URL que intentaba acceder antes de ser interceptado por el *middleware* de autenticación. Se puede proporcionar una URI por defecto para cualquier otro caso.

Especificar condiciones adicionales

Si se desea, se pueden añadir condiciones extra a la consulta de autenticación además del e-mail y contraseña del usuario. Por ejemplo, se puede verificar que el usuario esté marcado como "activo":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {  
    // The user is active, not suspended, and exists.  
}
```

En estos ejemplos, `email` no es una opción requerida, sino meramente usado como ejemplo. Usted debe usar el nombre de la columna que corresponda a un "nombre de usuario" en su base de datos.

Acceder a instancias *Guard* específicas

Se puede especificar que instancia de *Guard* se ha de utilizar utilizando el método `guard` en la facade `Auth`. Esto permite separar la gestión de la autenticación para las diferentes partes de la aplicación utilizando modelos autenticables o tablas de usuarios.

El nombre del *guard* pasado al método `guard` debe corresponder con uno de los *guards* del archivo de configuración `auth.php`:

```
if (Auth::guard('admin')->attempt($credentials)) {  
    //  
}
```

Cerrar sesión – *Log out*

Para desconectar usuarios de la aplicación, se utiliza el método `logout` de la facade `Auth`. Esto borrará la información de autenticación en la sesión del usuario:

```
Auth::logout();
```

Recordar usuarios

Para proporcionar a la aplicación de la funcionalidad "recuerdame", se puede pasar un valor booleano como segundo argumento del método `attempt`, el cual mantendrá al usuario autenticado de forma indefinida, o hasta que manualmente se desconecte. Por supuesto, la

tabla `users` debe incluir la columna de cadena `remember_token`, la cual será utilizada para almacenar el token "remember me".

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {  
    // The user is being remembered...  
}
```

Si se está usando el `LoginController` incluido en Laravel, la lógica para "recordar" usuarios ya está incluida en los *traits* que utiliza el controlador.

Si se "recuerdan" usuarios, se puede utilizar el método `viaRemember` para determinar si un usuario fue autenticado a través de la cookie "remember me":

```
if (Auth::viaRemember()) {  
    //  
}
```

Otros métodos de autenticación

Autenticar una instancia de usuario

Para autenticar una instancia existente de un usuario en la aplicación, se puede utilizar el método `login` con ésta. El objeto proporcionado debe ser una implementación del [contrato Illuminate\Contracts\Auth\Authenticatable](#). Por supuesto, el modelo `App\User` proporcionado con Laravel ya implementa esta interfaz:

```
Auth::login($user);  
  
// Login and "remember" the given user...  
Auth::login($user, true);
```

Por supuesto, se puede especificar la instancia `guard` que se desee utilizar:

```
Auth::guard('admin')->login($user);
```

Autenticar un usuario por ID

Para loguear a un usuario en la aplicación por ID, se puede utilizar el método `loginUsingId` . Este método simplemente acepta la clave primaria del usuario que se desea autenticar:

```
Auth::loginUsingId(1);

// Login and "remember" the given user...
Auth::loginUsingId(1, true);
```

Autenticar un usuario una vez

Es posible utilizar el método `once` para autenticar a un usuario para una única petición. No se utilizarán sesiones ni *cookies*, por lo que puede ser útil para APIs sin estado:

```
if (Auth::once($credentials)) {
    //
}
```

Autenticación HTTP Basic

La [Autenticación HTTP Basic](#) permite autenticar usuarios en la aplicación de forma sencilla y rápida sin necesidad de una página de "login" dedicada. Para empezar, añadir el [middleware auth.basic](#) a la ruta. El middleware `auth.basic` viene de serie con Laravel framework, así que no hay que definirlo:

```
Route::get('profile', function () {
    // Only authenticated users may enter...
})->middleware('auth.basic');
```

Una vez que el *middleware* se ha adjuntado a la ruta, automáticamente se solicitarán las credenciales al acceder a la ruta en el navegador. Por defecto, el *middleware* `auth.basic` utilizará la columna `email` del usuario como "nombre de usuario".

Una nota sobre FastCGI

Si se utiliza PHP FastCGI, la autenticación HTTP Basic puede que no funcione correctamente. Se deben añadir las siguientes líneas al archivo `.htaccess` :

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Autenticación HTTP Basic sin estado

También puede utilizar autenticación HTTP Basic sin establecer una cookie de identificación de usuario en la sesión, lo que resulta particularmente útil para autenticación de API. Para ello, [definir un middleware](#) que llame al método `onceBasic`. Si no se produce una respuesta por el método `onceBasic`, la petición seguirá adelante en la aplicación:

```
<?php

namespace Illuminate\Auth\Middleware;

use Illuminate\Support\Facades\Auth;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }
}
```

Siguiente, [registrar el middleware de ruta](#) y fijarlo a una ruta:

```
Route::get('api/user', function () {
    // Only authenticated users may enter...
})->middleware('auth.basic.once');
```

Añadir Guards personalizados

Se pueden definir *guards* de autenticación propios utilizando el método `extend` de la *facade Auth*. Debe establecer la llamada al método `extend` en un service provider. Puesto que Laravel incluye un `AuthServiceProvider`, se puede incluir el código ahí mismo:

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Support\Facades\Auth;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::extend('jwt', function ($app, $name, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\Guard...

            return new JwtGuard($app['auth']->createUserProvider($config['provider']));
        });
    }
}
```

Como se puede observar en el ejemplo anterior, el *callback* que se pasa al método `extend` debe retornar una implementación de `Illuminate\Contracts\Auth\Guard`. Esta interfaz contiene los métodos que habrá que implementar para definir un *guard* personalizado. Una vez que se ha definido el *guard*, se puede configurar en la opción `guards</>` del archivo de configuración `auth.php`:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

Añadir *user providers* personalizados

Si no se está utilizando una base de datos relacional tradicional para almacenar los usuarios, será necesario extender Laravel con un proveedor de autenticación propio. Utilice el método `provider` de la facade `Auth` para definir un *user provider* personalizado:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Auth;
use App\Extensions\RiakUserProvider;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::provider('riak', function ($app, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\UserProvider...

            return new RiakUserProvider($app->make('riak.connection'));
        });
    }
}

```

Una vez que se ha registrado el proveedor utilizando el método `provider`, se puede cambiar a esta implementación en el archivo de configuración `auth.php`. Primero, definir un `provider` que utilice el nuevo driver:

```

'providers' => [
    'users' => [
        'driver' => 'riak',
    ],
],

```

Después, ya se puede utilizar ese *provider* en la sección `guards` de la configuración:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

El contrato User Provider

Las implementaciones de `Illuminate\Contracts\Auth\UserProvider` sólo son responsables de sacar una implementación de `Illuminate\Contracts\Auth\Authenticatable` fuera del sistema de almacenamiento, como MySQL, Riak, etc. Estas dos interfaces permiten a los mecanismos de autenticación de Laravel continuar funcionando independientemente de cómo se almacenan los datos del usuario o qué tipo de clase se utiliza para representarlo.

El contrato `Illuminate\Contracts\Auth\UserProvider` :

```
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider {

    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);

    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array $credentials);

}
```

La función `retrieveById` por lo general recibe una clave que representa al usuario, tal como un ID auto-incremental de una base de datos MySQL. La implementación de `Authenticatable` que concuerda con el ID se obtendrá y será devuelta por el método.

La función `retrieveByToken` obtiene un usuario por su único `$identifier` y `$token` "remember me", almacenado en el campo `remember_token`. Como el método anterior, se retornará una implementación de `Authenticatable`.

El método `updateRememberToken` actualiza el campo `remember_token` de `$user` con el nuevo `$token`. El nuevo *token* puede ser un *token* nuevo, asignado a un intento de login satisfactorio "recordarme", o cuando el usuario cierra sesión.

El método `retrieveByCredentials` recibe el *array* de credenciales que se pasan al método `Auth::attempt` cuando se intenta iniciar sesión en una aplicación. Este método es el que debe "consultar" a la capa de almacenamiento por el usuario que coincide con estos credenciales. Por lo general, este método ejecuta una consulta con una condición "where" en `$credentials['username']`. El método debe retornar una implementación de `Authenticatable`. **Este método no debe intentar hacer ninguna validación de contraseñas o autenticación.**

El método `validateCredentials` debe comparar el `$user` dado con los `$credentials` para autenticar al usuario. Por ejemplo, este método debería utilizar `Hash::check` para comparar el valor de `$user->getAuthPassword()` con `$credentials['password']`. Debe retornar `true` o `false` indicando si la contraseña es válida.

El contrato Authenticatable

Una vez que ya se conocen de los métodos del `UserProvider`, hay otro contrato para revisar `Authenticatable`. Recordar, el provider debe retornar una implementación de esta interfaz desde los métodos `retrieveById` y `retrieveByCredentials`:

```
<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable {

    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();

}
```

Esta interfaz es simple. El método `getAuthIdentifierName` debe retornar el nombre del campo "clave primaria" del usuario y `getAuthIdentifier` debe retornar la "clave primaria" del usuario. En una base de MySQL, esto se correspondería con la clave primaria auto-

incremental. El método `getAuthPassword` debería devolver la contraseña del usuario cifrada. Esta interfaz permite al sistema de autenticación trabajar con cualquier clase de usuario, independientemente de qué capa de abstracción ORM o almacenamiento de información utiliza. Por defecto, Laravel incluye una clase `User` en la carpeta `app` que implementa esta interfaz, por lo que se puede consultar esta clase para obtener un ejemplo de una implementación.

Eventos

Laravel dispara varios [eventos](#) durante el proceso de autenticación. Se pueden capturar estos eventos estableciendo listeners en el `EventServiceProvider`:

```
/**  
 * The event listener mappings for the application.  
 *  
 * @var array  
 */  
  
protected $listen = [  
    'Illuminate\Auth\Events\Registered' => [  
        'App\Listeners\LogRegisteredUser',  
    ],  
  
    'Illuminate\Auth\Events\Attempting' => [  
        'App\Listeners\LogAuthenticationAttempt',  
    ],  
  
    'Illuminate\Auth\Events\Authenticated' => [  
        'App\Listeners\LogAuthenticated',  
    ],  
  
    'Illuminate\Auth\Events\Login' => [  
        'App\Listeners\LogSuccessfulLogin',  
    ],  
  
    'Illuminate\Auth\Events\Failed' => [  
        'App\Listeners\LogFailedLogin',  
    ],  
  
    'Illuminate\Auth\Events\Logout' => [  
        'App\Listeners\LogSuccessfulLogout',  
    ],  
  
    'Illuminate\Auth\Events\Lockout' => [  
        'App\Listeners\LogLockout',  
    ],  
  
    'Illuminate\Auth\Events>PasswordReset' => [  
        'App\Listeners\LogPasswordReset',  
    ],  
];
```

Autenticación API (Passport)

- # Introducción
- # Instalación
 - # Inicio rápido Frontend
 - # Despliegue de *Passport*
- # Configuración
 - # Duración de token
- # Emisión de tokens de acceso
 - # Gestión de clientes
 - # Solicitud de tokens
 - # Refrescando Tokens
- # Tokens de concesión de contraseña
 - # Crear un cliente de concesión de contraseña
 - # Solicitud de Tokens
 - # Solicitud de todos los ámbitos
- # Fichas de donaciones implícitas
- # Tokens de concesión de credenciales de cliente
- # Tokens de acceso personal
 - # Crear un cliente de acceso personal
 - # Administrar tokens de acceso personal
- # Protección de Rutas
 - # A través de *middleware*
 - # Pasando el token de acceso
- # Alcance Token
 - # Definiendo scopes
 - # Asignación de scopes a tokens
 - # Verificación de scopes
- # Consumir su API con JavaScript
- # Eventos
- # Testing

Introducción

Laravel ya facilita la autenticación a través de los formularios de inicio de sesión tradicionales, pero ¿qué pasa con las APIs? Las APIs generalmente usan tokens para autenticar usuarios y no mantienen el estado de la sesión entre las solicitudes. Laravel hace que la autenticación de API sea muy sencilla utilizando Laravel Passport, que proporciona una implementación completa del servidor OAuth2 para su aplicación en cuestión de minutos. Passport está construido sobre el servidor [League OAuth2](#) que es mantenido por Alex Bilbie.

Esta documentación asume que usted ya está familiarizado con OAuth2. Si usted no sabe nada acerca de OAuth2, considere familiarizarse con la terminología y características generales de OAuth2 antes de continuar.

Instalación

Para empezar, instale Passport a través del gestor de paquetes Composer:

```
composer require laravel/passport
```

El proveedor de servicio Passport registra su propio directorio de migración de base de datos con el framework, por lo que debe migrar su base de datos después de registrar el proveedor. Las migraciones de Passport crearán las tablas que su aplicación necesita para almacenar clientes y tokens de acceso:

```
php artisan migrate
```

Si no va a utilizar las migraciones predeterminadas de Passport, debe llamar al método `Passport::ignoreMigrations` en el método `register` de su `AppServiceProvider`. Puede exportar las migraciones predeterminadas utilizando `php artisan vendor:publish --tag=passport-migrations`.

A continuación, debe ejecutar el comando `passport:install`. Este comando creará las claves de cifrado necesarias para generar tokens de acceso seguro. Además, el comando creará clientes "de acceso personal" y "de concesión de contraseña" que se utilizarán para generar tokens de acceso:

```
php artisan passport:install
```

Después de ejecutar el comando, agregue el *trait* `Laravel\Passport\HasApiTokens` en su modelo `App\User`. Este *trait* proporcionará algunos *helpers* a su modelo que le permitirán inspeccionar el token y alcances del usuario autenticado:

```
<?php

namespace App;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

A continuación, debe llamar al método `Passport::routes` dentro del método `boot` de su `AuthServiceProvider`. Este método registrará las rutas necesarias para emitir tokens de acceso y revocará los tokens de acceso, clientes y tokens de acceso personal:

```

<?php

namespace App\Providers;

use Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}

```

Por último, en el archivo de configuración `config/auth.php`, debe cambiar la opción `driver` de `api` a `passport`. Esto le indicará a su aplicación el uso de Passport's `TokenGuard` al autenticar las solicitudes de API entrantes:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Inicio rápido Frontend

Para poder utilizar los componentes Passport de Vue, debe estar utilizando el framework JavaScript Vue. Estos componentes también utilizan el framework CSS Bootstrap. Sin embargo, incluso si no está utilizando estas herramientas, los componentes sirven como una referencia valiosa para su propia implementación frontend.

Passport se entrega con una API JSON que puede utilizar para permitir a sus usuarios crear clientes y tokens de acceso personal. Sin embargo, puede llevar mucho tiempo codificar un frontend para interactuar con estas APIs. Por lo tanto, Passport también incluye componentes [Vue](#) preconstruidos que puede utilizar como ejemplo de implementación o punto de partida para su propia implementación.

Para publicar los componentes Vue de Passport, utilice el comando Artisan `vendor:publish` :

```
php artisan vendor:publish --tag=passport-components
```

Los componentes publicados se colocarán en el directorio `resources/assets/js/components` . Una vez que los componentes se hayan publicado, debe registrarlos en su archivo `resources/assets/js/app.js` :

```
Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue')
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue')
);
```

Después de registrar los componentes, asegúrese de ejecutar `npm run dev` para recomilar sus recursos. Una vez que haya recompilado sus recursos, puede colocar los componentes en una de las plantillas de su aplicación para empezar a crear clientes y tokens de acceso personal:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Despliegue de *Passport*

Al implementar Passport en sus servidores de producción por primera vez, es probable que tenga que ejecutar el comando `passport:keys`. Este comando genera las claves de encriptación que necesita Passport para generar el token de acceso. Las claves generadas no suelen mantenerse en el control de origen:

```
php artisan passport:keys
```

Configuración

Duración de token

De forma predeterminada, Passport emite tokens de acceso de larga duración que nunca necesitan actualizarse. Si desea configurar una vida útil más corta, puede utilizar los métodos `tokensExpireIn` y `refreshTokensExpireIn`. Estos métodos deben ser llamados desde el método `boot` de su `AuthServiceProvider`:

```
use Carbon\Carbon;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(Carbon::now()->addDays(15));

    Passport::refreshTokensExpireIn(Carbon::now()->addDays(30));
}
```

Emisión de tokens de acceso

El uso de OAuth2 con códigos de autorización es como la mayoría de los desarrolladores que están familiarizados con OAuth2 lo usan. Cuando se utilizan códigos de autorización, una aplicación cliente redirigirá a un usuario a su servidor donde aprobará o denegará la solicitud de emisión de un token de acceso al cliente.

Gestión de clientes

En primer lugar, los desarrolladores que necesitan interactuar con la API de su aplicación necesitarán registrar su aplicación con la suya creando un "cliente". Normalmente, esto consiste en proporcionar el nombre de su aplicación y una URL a la que su aplicación puede redirigir después de que los usuarios aprueben su solicitud de autorización.

El comando `passport:client`

La forma más sencilla de crear un cliente es utilizando el comando Artisan `passport:client`. Este comando puede ser utilizado para crear sus propios clientes para probar su funcionalidad OAuth2. Cuando ejecute el comando `client`, Passport le pedirá más información sobre su cliente y le proporcionará un ID de cliente y un valor secreto:

```
php artisan passport:client
```

API JSON

Dado que sus usuarios no podrán utilizar el comando `client`, Passport proporciona una API JSON que puede utilizar para crear clientes. Esto le ahorra la molestia de tener que codificar manualmente los controladores para crear, actualizar y eliminar clientes.

Sin embargo, usted necesitará emparejar la API JSON de Passport con su propio frontend para proporcionar un panel de control para que sus usuarios administren sus clientes. A continuación, revisaremos todos los endpoints de API para administrar clientes. Para mayor comodidad, utilizaremos [Axios](#) para demostrar cómo realizar solicitudes HTTP en los puntos finales.

Si no desea implementar el frontend de administración de cliente completo usted mismo, puede utilizar el frontend quickstart para tener un frontend completamente funcional en cuestión de minutos.

GET `/oauth/clients`

Esta ruta devuelve todos los clientes para el usuario autenticado. Esto es principalmente útil para listar todos los clientes del usuario para que puedan editarlos o borrarlos:

```
axios.get('/oauth/clients')
  .then(response => {
    console.log(response.data);
 });
```

POST `/oauth/clients`

Esta ruta se utiliza para crear nuevos clientes. Requiere dos tipos de datos: el nombre del cliente `name` y una URL de redirección `redirect`. La URL de redirección `redirect` es donde el usuario será redirigido después de aprobar o rechazar una solicitud de autorización.

Cuando se crea un cliente, se emitirá un ID y un valor secreto. Estos valores se utilizarán cuando solicite tokens de acceso desde su aplicación. La ruta de creación del cliente devolverá la nueva instancia del cliente:

```
const data = {
  name: 'Client Name',
  redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
  .then(response => {
    console.log(response.data);
  })
  .catch(response => {
    // List errors on response...
  });

```

PUT /oauth/clients/{client-id}

Esta ruta se utiliza para actualizar clientes. Requiere dos tipos de datos: el nombre del cliente `name` y una URL de redirección `redirect`. La URL de redirección `redirect` es donde el usuario será redirigido después de aprobar o rechazar una solicitud de autorización. La ruta devolverá la instancia de cliente actualizada:

```
const data = {
  name: 'New Client Name',
  redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
  .then(response => {
    console.log(response.data);
  })
  .catch(response => {
    // List errors on response...
  });

```

DELETE /oauth/clients/{client-id}

Esta ruta se utiliza para eliminar clientes:

```
axios.delete('/oauth/clients/' + clientId)
  .then(response => {
    //
  });
}
```

Solicitud de tokens

Redireccionando para autenticación

Una vez que se ha creado un cliente, los desarrolladores pueden utilizar su ID de cliente y el valor secreto para solicitar un código de autorización y un token de acceso desde su aplicación. En primer lugar, la aplicación consumidora debe hacer una solicitud de redireccionamiento a la ruta `/oauth/authorize` de su aplicación así:

```
Route::get('/redirect', function () {
  $query = http_build_query([
    'client_id' => 'client-id',
    'redirect_uri' => 'http://example.com/callback',
    'response_type' => 'code',
    'scope' => '',
  ]);

  return redirect('http://your-app.com/oauth/authorize?' . $query);
});
```

Recuerde, la ruta `/oauth/authorize` ya está definida por el método `Pasaporte::route`. No es necesario definir manualmente esta ruta.

Aprobar la solicitud

Al recibir solicitudes de autorización, Passport mostrará automáticamente una plantilla al usuario para que éste pueda aprobar o denegar la solicitud de autorización. Si se aprueba la solicitud, será redirigido de vuelta al `redirect_uri` especificado por la aplicación consumidora. El `redirect_uri` debe coincidir con la URL de redirección especificada `redirect` de cuando se creó el cliente.

Si desea personalizar la pantalla de autorización, puede publicar las vistas de Passport utilizando el comando Artisan `vendor:publish`. Las vistas publicadas se colocarán en `resources/views/vendor/passport`:

```
php artisan vendor:publish --tag=passport-views
```

Conversión de códigos de autorización para acceder a los Tokens

Si el usuario autoriza la solicitud, se redireccionará a la aplicación consumidora. El consumidor debe entonces enviar una solicitud `POST` a la aplicación para solicitar un token de acceso. La solicitud debe incluir el código de autorización emitido por la aplicación. En este ejemplo, usaremos la librería Guzzle HTTP para hacer la petición `POST`:

```
Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);
    return json_decode((string) $response->getBody(), true);
});
```

Esta ruta `/oauth/token` devuelve una respuesta JSON que contiene los atributos `access_token`, `refresh_token`, y `expires_in`. El atributo `expires_in` contiene el número de segundos hasta que expira el token de acceso.

Como la ruta `/oauth/authorize`, la ruta `/oauth/token` se define por el método `Passport::routes`. No es necesario definir manualmente esta ruta.

Refrescando Tokens

Si su aplicación emite tokens de acceso de corta duración, los usuarios tendrán que refrescar sus tokens mediante el token de actualización que se les proporcionó cuando se emitió el token de acceso. En este ejemplo, usaremos la librería Guzzle HTTP para hacer la petición POST:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

La ruta `/oauth/token` devuelve una respuesta JSON que contiene los atributos `access_token`, `refresh_token`, y `expires_in`. El atributo `expires_in` contiene el número de segundos hasta que expira el token de acceso.

Tokens de concesión de contraseña

La concesión de la contraseña OAuth2 permite a sus otros clientes, como una aplicación móvil, obtener una clave de acceso utilizando una dirección de correo electrónico / nombre de usuario y contraseña. Esto le permite emitir tokens de acceso de forma segura a sus clientes de primera mano sin requerir que sus usuarios pasen por todo el flujo de redirecccionamiento del código de autorización de OAuth2.

Crear un cliente de concesión de contraseña

Antes de que su aplicación pueda emitir tokens a través de la concesión de contraseña, deberá crear un cliente apropiado para ello. Puede hacerlo utilizando el comando `passport:client` con la opción `--password`. Si ya ha ejecutado el comando `passport:install`, no necesita ejecutar este comando:

```
php artisan passport:client --password
```

Solicitud de tokens

Una vez que haya creado un cliente de concesión de contraseña, puede solicitar un token de acceso enviando una solicitud `POST` a la ruta `/oauth/token` con la dirección de correo electrónico y contraseña del usuario. Recuerde, esta ruta ya está registrada por el método `Passport::routes` para que no sea necesario definirla manualmente. Si la solicitud tiene éxito, recibirá un `access_token` y `refresh_token` en la respuesta JSON del servidor:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

Recuerde, los tokens de acceso son de larga duración por defecto. Sin embargo, usted es libre de configurar la vida útil máxima del token de acceso si es necesario.

Solicitud de todos los ámbitos

Cuando utilice la concesión de contraseña, puede que desee autorizar el token para todos los ámbitos soportados por su aplicación. Puede hacerlo solicitando el scope `*`. Si usted solicita el scope `*`, el método `can` en la instancia del token siempre regresará `true`. Este alcance sólo puede asignarse a un token que se emita utilizando la concesión de `password`:

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
])
```

Implicit Grant Tokens

La concesión implícita es similar a la concesión del código de autorización; sin embargo, el token se devuelve al cliente sin cambiar un código de autorización. Esta concesión se utiliza más comúnmente para aplicaciones JavaScript o móviles en las que las credenciales del cliente no se pueden almacenar de forma segura. Para habilitar la concesión, se debe llamar al método `enableImplicitGrant` en su `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Una vez que se ha habilitado una concesión, los desarrolladores pueden utilizar su ID de cliente para solicitar un token de acceso desde su aplicación. La aplicación consumidora debería hacer una petición de redireccionamiento a la ruta `/oauth/authorize` de su aplicación:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

Recuerde, la ruta `/oauth/authorize` ya está definida por el método `Passport::routes`. No es necesario definir manualmente esta ruta.

Tokens de concesión de credenciales de cliente

La concesión de credenciales de cliente es adecuada para la autenticación máquina a máquina. Por ejemplo, puede utilizar esta concesión en un trabajo programado que esté realizando tareas de mantenimiento a través de una API. Para usar este método primero necesita agregar un nuevo middleware a su `$routeMiddleware` en `app/Http/Kernel.php`:

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];
```

A continuación, conecte este middleware a una ruta:

```
Route::get('/user', function(Request $request) {
    ...
})->middleware('client');
```

Para recuperar un token, haga una solicitud al endpoint `oauth/token`:

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

return json_decode((string) $response->getBody(), true)['access_token'];
```

Tokens de acceso personal

A veces, sus usuarios pueden querer emitir tokens de acceso para ellos mismos sin pasar por el típico flujo de redireccionamiento de código de autorización. Permitir a los usuarios emitir tokens para sí mismos a través de la interfaz de usuario de su aplicación puede ser útil para permitir a los usuarios experimentar con su API o puede servir como un enfoque más simple para emitir tokens de acceso en general.

Los tokens de acceso personal son siempre duraderos. Su vida útil no se modifica cuando se utilizan los métodos `tokensExpireIn` o `refreshTokensExpireIn`.

Crear un cliente de acceso personal

Antes de que su aplicación pueda emitir tokens de acceso personal, deberá crear un cliente de acceso personal. Puede hacerlo utilizando el comando `passport:client` con la opción `--personal`. Si ya ha ejecutado el comando `passport:install`, no necesita ejecutar este comando:

```
php artisan passport:client --personal
```

Administrar tokens de acceso personal

Una vez que haya creado un cliente de acceso personal, puede emitir tokens para un usuario determinado utilizando el método `createToken` en la instancia de modelo de `User`. El método `createToken` acepta el nombre del token como primer argumento y una matriz opcional de [scopes](#) como segundo argumento:

```
$user = App\User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

API JSON

Passport también incluye una API JSON para la gestión de tokens de acceso personal. Usted puede asociar esto con su propio frontend para ofrecer a sus usuarios un panel de control para gestionar los tokens de acceso personal. A continuación, revisaremos todos los endpoints de la API para administrar los tokens de acceso personal. Para mayor comodidad, utilizaremos [Axios](#) para demostrar cómo realizar solicitudes HTTP en los puntos finales.

Si no desea implementar el frontend de acceso personal usted mismo, puede utilizar el quickstart del frontend para tener un frontend totalmente funcional en cuestión de minutos.

GET /oauth/scopes

Esta ruta devuelve todos los [alcances \(scopes\)](#) definidos para su aplicación. Puede utilizar esta ruta para listar los alcances que un usuario puede asignar a un token de acceso personal:

```
axios.get('/oauth/scopes')
  .then(response => {
    console.log(response.data);
 });
```

GET /oauth/personal-access-tokens

Esta ruta devuelve todos los tokens de acceso personal creados por el usuario autenticado. Esto es principalmente útil para listar todos los tokens del usuario para que puedan editarlos o borrarlos:

```
axios.get('/oauth/personal-access-tokens')
  .then(response => {
    console.log(response.data);
  });
});
```

POST /oauth/personal-access-tokens

Esta ruta crea nuevos tokens de acceso personal. Requiere dos piezas de datos: `name` y `scopes` que deben asignarse al token:

```
const data = {
  name: 'Token Name',
  scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
  .then(response => {
    console.log(response.data.accessToken);
  })
  .catch(response => {
    // List errors on response...
  });
});
```

DELETE /oauth/personal-access-tokens/{token-id}

Esta ruta se puede utilizar para borrar los tokens de acceso personal:

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

Protección de Rutas

A través de *middleware*

Passport incluye un [authentication guard \(protector de autenticación\)](#) que validará los tokens de acceso en las solicitudes entrantes. Una vez que haya configurado el `api` guard para usar el controlador de `passport`, sólo necesita especificar el middleware `auth:api` en cualquier ruta que requiera un token de acceso válido:

```
Route::get('/user', function () {
    //
})->middleware('auth:api');
```

Pasando el token de acceso

Al llamar a rutas protegidas por Passport, los consumidores de la API de su aplicación deben especificar su token de acceso como `Bearer` token en el encabezado `Authorization` de su solicitud. Por ejemplo, cuando se utiliza la biblioteca Guzzle HTTP:

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer ' . $accessToken,
    ],
]);
```

Alcance Token

Definiendo scopes

Los scopes (alcances) permiten a sus clientes API obtener un conjunto específico de permisos cuando solicitan autorización para acceder a una cuenta. Por ejemplo, si está creando una aplicación de comercio electrónico, no todos los consumidores de la API necesitarán poder realizar pedidos. En su lugar, puede permitir que los consumidores sólo soliciten autorización para acceder al estatus de envío de pedidos. En otras palabras, los alcances permiten a los usuarios de su aplicación limitar las acciones que una aplicación de terceros puede realizar en su nombre.

Puede definir los alcances de su API usando el método `Passport::tokensCan` en el método `boot` de su `AuthServiceProvider`. El método `tokensCan` acepta una matriz de nombres y descripciones de alcance. La descripción del alcance puede ser cualquier cosa que desee y se mostrará a los usuarios en la pantalla de autorización:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);
```

Asignación de scopes a tokens

Cuando solicita códigos de autorización

Cuando solicita un token de acceso utilizando la concesión de código de autorización, los consumidores deben especificar sus alcances deseados como parámetro de cadena de consulta de `scope`. El parámetro `scope` debe ser una lista de alcances delimitada por el espacio:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

Cuando se emitan Tokens de acceso personal

Si está emitiendo tokens de acceso personal utilizando el método `createToken` del modelo `User`, puede pasar la matriz de alcances deseados como segundo argumento al método:

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Verificación de scopes

Passport incluye dos middleware que se pueden utilizar para verificar que una solicitud entrante se autentica con un token al que se le ha concedido un alcance dado. Para empezar, agregue el siguiente middleware a la propiedad `$routeMiddleware` de su archivo `app/Http/Kernel.php`:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,  
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

Verifique todos los alcances

El middleware `scopes` puede ser asignado a una ruta para verificar que el token de acceso de la solicitud entrante tiene *todos* los alcances listados:

```
Route::get('/orders', function () {  
    // Access token has both "check-status" and "place-orders" scopes...  
})->middleware('scopes:check-status,place-orders');
```

Verifique cualquier alcance

El middleware `scope` puede asignarse a una ruta para verificar que el token de acceso de la solicitud entrante tenga *al menos uno* de los alcances listados:

```
Route::get('/orders', function () {  
    // Access token has either "check-status" or "place-orders" scope...  
})->middleware('scope:check-status,place-orders');
```

Comprobación de ámbitos en una instancia de un Token

Una vez que una solicitud autenticada de token de acceso ha entrado en su aplicación, puede comprobar si el token tiene un alcance determinado utilizando el método `tokenCan` en la instancia de `Usuario` autenticado:

```
use Illuminate\Http\Request;  
  
Route::get('/orders', function (Request $request) {  
    if ($request->user()->tokenCan('place-orders')) {  
        //  
    }  
});
```

Consumir su API con JavaScript

Al crear una API, puede ser extremadamente útil poder consumir su propia API desde su aplicación JavaScript. Este enfoque al desarrollo de API permite que su propia aplicación consuma la misma API que usted está compartiendo con el mundo. La misma API puede ser consumida por su aplicación web, aplicaciones móviles, aplicaciones de terceros y cualquier SDK que pueda publicar en varios administradores de paquetes.

Normalmente, si desea consumir su API desde su aplicación JavaScript, debería enviar manualmente un token de acceso a la aplicación y pasarlo con cada solicitud. Sin embargo, Passport incluye un middleware que puede manejar esto por usted. Todo lo que necesita hacer es añadir el middleware `CreateFreshApiToken` al grupo de middleware `web` :

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

Este middleware Passport adjuntará una cookie `laravel_token` a sus respuestas salientes. Esta cookie contiene un JWT cifrado que Passport utilizará para autenticar las solicitudes API de su aplicación JavaScript. Ahora, puede realizar solicitudes a la API de su aplicación sin pasar explícitamente un token de acceso:

```
axios.get('/api/user')
  .then(response => {
    console.log(response.data);
 });
```

Cuando se utiliza este método de autenticación, el scaffolding JavaScript predeterminado de Laravel instruye a Axios para que siempre envíe en el encabezado el `X-CSRF-TOKEN` y el `X-Requested-With`. Sin embargo, debe asegurarse de incluir su token CSRF en un [meta tag HTML](#):

```
window.axios.defaults.headers.common = {
  'X-Requested-With': 'XMLHttpRequest',
};
```

Si está utilizando un framework JavaScript diferente, debe asegurarse de que está configurado para enviar en los encabezados el `X-CSRF-TOKEN` y `X-Requested-With` con cada solicitud saliente.

Eventos

Passport levanta eventos cuando se emiten tokens de acceso y tokens de actualización. Usted puede usar estos eventos para limpiar o revocar otros tokens de acceso en su base de datos. Puede adjuntar oyentes a estos eventos en el `EventServiceProvider` de su aplicación:

```
/**  
 * The event listener mappings for the application.  
 *  
 * @var array  
 */  
  
protected $listen = [  
    'Laravel\Passport\Events\AccessTokenCreated' => [  
        'App\Listeners\RevokeOldTokens',  
    ],  
  
    'Laravel\Passport\Events\RefreshTokenCreated' => [  
        'App\Listeners\PruneOldTokens',  
    ],  
];
```

Testing

El método `actingAs` de Passport se puede utilizar para especificar el usuario actualmente autenticado, así como sus alcances. El primer argumento dado al método `actingAs` es la instancia de usuario y el segundo es un array de alcances que debe ser otorgado al token del usuario:

```
public function testServerCreation()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(200);
}
```

Autorización

Introducción

Gates (puertas)

- # Escribir puertas
- # Autorizar acciones

Creación de políticas

- # Generación de políticas
- # Registro de políticas

Definición políticas

- # Métodos de políticas
- # Métodos sin modelos
- # Filtros de políticas

Autorizar acciones usando políticas

- # Vía el modelo User
- # A través de *middleware*
- # A través de *helper* de controladores
- # A través de plantillas Blade

Introducción

Además de contar con servicios de [authentication](#) desde la instalación, Laravel también provee una manera simple de autorizar las acciones de los usuarios contra un recurso determinado. Como en la autenticación, el enfoque de Laravel hacia la autorización es simple, y existen principalmente dos maneras de autorizar acciones: *gates* y políticas.

Piense en *gates* y en las políticas como las rutas y los controladores. *Gates* ofrecen una manera simple de autorización basada en funciones anónimas, mientras que las políticas, como los controladores, agrupan la lógica alrededor de un modelo o recurso particular. Primero veremos las *gates* y luego las políticas.

Cuando se construye una aplicación, no es necesario elegir entre usar exclusivamente *gates* o exclusivamente políticas. La mayoría de las aplicaciones probablemente contienen una mezcla de *gates* y políticas ¡y eso es perfectamente correcto! Los *Gates* son más aplicables a

las acciones que no están relacionadas con ningun modelo o recurso, tal como ver el dashboard de administración. Por el contrario, las políticas se deberían utilizar cuando se desea autorizar una acción para un modelo o recurso particular.

Gates

Escribir Gates

Los *Gates* son funciones anónimas que determinan si un usuario está autorizado para realizar una determinada acción y se definen típicamente en la clase

`App\Providers\AuthServiceProvider` usando el *facade* `Gate`. Los *Gates* siempre reciben una instancia del usuario como primer argumento, y pueden recibir opcionalmente argumentos adicionales como un modelo Eloquent relevante:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function ($user, $post) {
        return $user->id == $post->user_id;
    });
}
```

Los *Gates* también se pueden ser definidos usando el estilo de *callback* `Class@method`, como en los controladores:

```

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', 'PostPolicy@update');
}

```

Gates de recursos

Se pueden definir multiples habilidades del *Gates* usando el método `resource` :

```
Gate::resource('posts', 'PostPolicy');
```

Esto es idéntico a definir manualmente las siguientes definiciones de *Gate*:

```

Gate::define('posts.view', 'PostPolicy@view');
Gate::define('posts.create', 'PostPolicy@create');
Gate::define('posts.update', 'PostPolicy@update');
Gate::define('posts.delete', 'PostPolicy@delete');

```

Por defecto, se definirán las habilidades `view` , `create` , `update` , y `delete` . Usted puede sobreescribir o agregar a las habilidades por defecto pasando una matriz como tercer argumento al método `resource` . Las claves de la matriz definen los nombres de las habilidades, mientras los valores definen los nombres de los métodos. Por ejemplo, el siguiente código crea las definiciones de dos nuevos *Gates* - `posts.image` y `posts.photo` :

```

Gate::resource('posts', 'PostPolicy', [
    'image' => 'updateImage',
    'photo' => 'updatePhoto',
]);

```

Autorizar acciones

Para autorizar acciones usando *Gates*, se deben usar los métodos `allows` o `denies`. Nótese que no es necesario enviar a los métodos el usuario que se encuentra autenticado actualmente. Laravel se encargará automáticamente de pasar el usuario a la función anónima del *Gate*:

```
if (Gate::allows('update-post', $post)) {  
    // The current user can update the post...  
}  
  
if (Gate::denies('update-post', $post)) {  
    // The current user can't update the post...  
}
```

Si se quisiera determinar si un usuario particular está autorizado a realizar una acción, se puede usar el método `forUser` en la *facade Gate*:

```
if (Gate::forUser($user)->allows('update-post', $post)) {  
    // The user can update the post...  
}  
  
if (Gate::forUser($user)->denies('update-post', $post)) {  
    // The user can't update the post...  
}
```

Creación de políticas

Generación de políticas

Las Políticas son clases que organizan la lógica de autorización alrededor de un modelo o recurso particular. Por ejemplo, si la aplicación es un blog, se tendría un modelo `Post` y la correspondiente `PostPolicy` para autorizar acciones de los usuarios tales como crear o modificar posts.

Se puede generar una política utilizando el [comando de artisan](#) `make:policy`. La política generada se colocará en el directorio `app/Policies`. Si el directorio no existe en la aplicación, Laravel lo creará:

```
php artisan make:policy PostPolicy
```

El comando `make:policy` genera una clase de política vacía. Si se quiere generar una clase con los métodos de políticas básicos de "CRUD" ya incluidos, se puede especificar la bandera `--model` cuando se ejecute el comando:

```
php artisan make:policy PostPolicy --model=Post
```

Todas las políticas se resuelven vía el service container de Laravel, lo que permite hacer un *type-hint* de cualquier dependencia necesaria en el constructor de la política e inyectarlos automáticamente.

Registro de políticas

Una vez que la política existe, es necesario que se registre. El `AuthServiceProvider` incluído en las aplicaciones de Laravel contiene una propiedad `policies` la cual mapea los modelos Eloquent con sus políticas correspondientes. Registrar una política le dice a Laravel cuál política utilizar para la autorización de acciones sobre un modelo dado:

```
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        //
    }
}
```

Definir políticas

Métodos de políticas

Una vez que se ha registrado la política, se pueden incluir métodos para cada una de las acciones que se autorizan. Por ejemplo, se define un método `update` en la política `PostPolicy` la cual determina si un `User` dado puede actualizar una instancia de `Post`.

El método `update` recibirá como argumentos un `User` y una instancia de `Post`, y debería retornar `true` o `false` indicando si el usuario está autorizado a editar el `Post` dado. Así, para este ejemplo, se verifica que el `id` del usuario coincida con el `user_id` del post:

```
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param \App\User $user
     * @param \App\Post $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

Se puede continuar definiendo métodos adicionales en la política a medida que sean necesarios por las distintas acciones a autorizar. Por ejemplo, se podrían definir métodos `view` o `delete` para autorizar las distintas acciones sobre los `Post`, pero hay que recordar que se puede dar cualquier nombre que se desee a los métodos.

Si se usa la opción `--model` cuando se crea la política a través de la cónsola de Artisan, esta ya contendrá los métodos para las acciones de `view`, `create`, `update`, y `delete`.

Métodos sin modelos

Algunos métodos de las políticas recibirán únicamente el usuario autenticado y no una instancia del modelo al cual autorizan. Esta situación es más común cuando se autorizan acciones de `create`. Por ejemplo, si se está creando un blog, es posible que se quiera verificar si el usuario tiene permisos para crear posts.

Cuando se definen métodos de políticas que no van a recibir una instancia de un modelo, tal como el método `create`, no se recibirá la instancia del modelo. En su lugar, se deben definir métodos que solo esperen el usuario autenticado:

```
/**  
 * Determine if the given user can create posts.  
 *  
 * @param \App\User $user  
 * @return bool  
 */  
  
public function create(User $user)  
{  
    //  
}
```

Filtros de políticas

Para ciertos usuarios, se puede querer autorizar todas las acciones dentro de una política dada. Para conseguir esto, se define un método `before` en la política. El método `before` se ejecuta antes de cualquier otro método en la política, dando una oportunidad para autorizar la acción antes de la llamada al método particular. Esta característica se usa más comúnmente para autorizar a los administradores de la aplicación a realizar cualquier acción:

```
public function before($user, $ability)  
{  
    if ($user->isSuperAdmin()) {  
        return true;  
    }  
}
```

Si se quiere denegar todas las autorizaciones para un usuario, se debe retornar `false` en el método `before`. Si se retorna `null`, la autorización se delegará al método que se ha llamado.

nota El método `before` de una clase política no se ejecuta si la clase no contiene un método con un nombre que coincida con el nombre de la habilidad consultada.

Autorizar acciones usando políticas

Vía el modelo User

El modelo `User` que está incluído en la aplicación Laravel cuenta con dos métodos útiles para autorizar acciones: `can` y `cant`. El método `can` recibe la acción que se desea autorizar y el modelo relevante. Por ejemplo, para determinar si un usuario está autorizado a editar un modelo `Post` dado:

```
if ($user->can('update', $post)) {  
    //  
}
```

Si una [política está registrada](#) para el modelo dado, el método `can` llamará automáticamente la política apropiada y devolverá un resultado *boolean*. Si no hay una política registrada para el modelo, el método `can` intentará llamar al Gate y a la función anónima que coincidan con el nombre de la acción dada.

Acciones que no requieren modelos

Hay que recordar, que algunas acciones como `create` pueden no requerir una instancia de un modelo. En estas situaciones, se puede pasar el nombre de una clase al método `can`. El nombre de la clase se usará para determinar cuál política utilizar para autorizar la acción:

```
use App\Post;  
  
if ($user->can('create', Post::class)) {  
    // Executes the "create" method on the relevant policy...  
}
```

Via middleware

Laravel incluye un *middleware* que puede autorizar aun antes de que la petición entrante llegue a las rutas o controladores. Por defecto, el *middleware*

`Illuminate\Auth\Middleware\Authorize` se le asigna la clave `can` en la clase `App\Http\Kernel`. Exploraremos un ejemplo del uso del *middleware* `can` para autorizar que un usuario puede editar un post del blog:

```
use App\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

En este ejemplo, se están enviando dos argumentos al *middleware* `can`. El primero es el nombre de la acción que se desea autorizar y el segundo es el parámetro en la ruta que se quiere pasar al método de la política. En este caso, dado que se está usando [implicit model binding](#), un modelo `Post` se enviará al método de la política. Si el usuario no está autorizado para realizar la acción dada, una respuesta HTTP con un código de estado `403` será generada por el middleware.

Acciones que no requieren modelos

De nuevo, algunas acciones como `create` pueden no requerir una instancia de modelo. En estas situaciones, se puede pasar el nombre de una clase al *middleware*. El nombre de la clase se usará para determinar cuál política se usa para autorizar la acción:

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Post');
```

A través de helper de controladores

Adicional a los útiles métodos que provee el modelo `User`, Laravel provee un útil método `authorize` a todos los controladores que extienden de la clase base `App\Http\Controllers\Controller`. Como el método `can`, este método acepta el nombre de la acción que se quiere autorizar y el modelo relevante. Si la acción no está autorizada, el método `authorize` lanzará una `Illuminate\Auth\Access\AuthorizationException`, la cual el manejador de excepciones de Laravel convertirá en una respuesta HTTP con código de estado `403`:

```
<?php

namespace App\Http\Controllers;

use App\Post;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @param Request $request
     * @param Post $post
     * @return Response
     */
    public function update(Request $request, Post $post)
    {
        $this->authorize('update', $post);

        // The current user can update the blog post...
    }
}
```

Acciones que no requieren modelos

Como se discutió previamente, que algunas acciones como `create` pueden no requerir una instancia de un modelo. En estas situaciones, se puede pasar el nombre de una clase al método `authorize`. El nombre de la clase se usará para determinar cuál política utilizar para autorizar la acción:

```

/**
 * Create a new blog post.
 *
 * @param Request $request
 * @return Response
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}

```

A través de plantillas Blade

Cuando se escriben plantillas de Blade, es posible que se quiera mostrar una sección de la página sólo si el usuario está autorizado a ejecutar una acción determinada. Por ejemplo, se puede querer mostrar un formulario de edición para un post del blog sólo si el usuario puede, en efecto, actualizar el post. En esta situación, se puede usar la familia de directivas `@can` y `@cannot`:

```

@can('update', $post)
    <!-- The Current User Can Update The Post -->
@elsecan('create', App\Post::class)
    <!-- The Current User Can Create New Post -->
@endcan

@cannot('update', $post)
    <!-- The Current User Can't Update The Post -->
@elsecannot('create', App\Post::class)
    <!-- The Current User Can't Create New Post -->
@endcannot

```

Estas directivas son atajos convenientes para escribir sentencias `@if` y `@unless`. Las sentencias `@can` y `@cannot` de arriba se traducen respectivamente en las siguientes sentencias:

```
@if (Auth::user()->can('update', $post))  
    <!-- The Current User Can Update The Post -->  
@endif  
  
@unless (Auth::user()->can('update', $post))  
    <!-- The Current User Can't Update The Post -->  
@endunless
```

Acciones que no requieren modelos

Como la mayoría de los otros métodos de autorización, se puede pasar el nombre de una clase a las directivas `@can` y `@cannot` si la acción no requiere la instancia de un modelo:

```
@can('create', App\Post::class)  
    <!-- The Current User Can Create Posts -->  
@endcan  
  
@cannot('create', App\Post::class)  
    <!-- The Current User Can't Create Posts -->  
@endcannot
```

Encriptación

- # Introducción
 - # Configuración
 - # Cómo usar el encriptador
-

Introducción

El encriptador de Laravel utiliza OpenSSL para proporcionar encriptación AES-256 y AES-128. Se le recomienda encarecidamente que utilice las funciones de encriptación incorporadas de Laravel y que no intente utilizar sus propios algoritmos de encriptación "caseros". Todos los valores encriptados de Laravel son firmados usando un código de autenticación de mensaje (MAC) para que su valor subyacente no pueda ser modificado una vez encriptado.

Configuración

Antes de usar el encriptador de Laravel, debe establecer una opción `key` en su archivo de configuración `config/app.php`. Debe usar el comando Artisan `php artisan key:generate` para generar esta clave ya que este comando usará el generador seguro de bytes aleatorios de PHP para construir su clave. Si este valor no se establece correctamente, los valores cifrados por Laravel serán inseguros.

Cómo usar el encriptador

Cifrar un valor

Puede encriptar un valor usando el helper `encrypt`. Todos los valores se encriptan utilizando OpenSSL y el algoritmo de cifrado `AES-256-CBC`. Además, todos los valores cifrados se firman con un código de autenticación de mensaje (MAC) para detectar cualquier modificación de la cadena cifrada:

```

<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Store a secret message for the user.
     *
     * @param Request $request
     * @param int $id
     * @return Response
     */
    public function storeSecret(Request $request, $id)
    {
        $user = User::findOrFail($id);

        $user->fill([
            'secret' => encrypt($request->secret)
        ])->save();
    }
}

```

Encriptación sin serialización

Los valores codificados se transmiten a través de `serialize` durante el encriptado, lo que permite la encriptación de objetos y matrices. Por lo tanto, los clientes no PHP que reciban valores cifrados necesitarán usar `unserialize` para los datos. Si desea encriptar y desencriptar valores sin serialización, puede utilizar los métodos `encryptString` y `decryptString` de la facade `Crypt`:

```

use Illuminate\Support\Facades\Crypt;

$encrypted = Crypt::encryptString('Hello world.');

$decrypted = Crypt::decryptString($encrypted);

```

Desencriptar un valor

Puede desencriptar valores utilizando el helper `decrypt` . Si no se puede desencriptar correctamente, como cuando la MAC es inválida, se lanzará una `Illuminate\Contracts\Encryption\DecryptException` :

```
use Illuminate\Contracts\Encryption\DecryptException;

try {
    $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
}
```

Cifrado – *Hashing*

- # **Introducción**
 - # **Uso básico**
-

Introducción

La [facade Hash](#) de Laravel proporciona un cifrado seguro para almacenar contraseñas de usuario. Si está utilizando las clases [LoginController](#) y [RegisterController](#) incorporadas en su aplicación Laravel, utilizarán automáticamente *Bcrypt* para el registro y la autenticación.

Bcrypt es una gran opción para el cifrado de contraseñas ya que su "factor de trabajo" es ajustable, lo que significa que el tiempo necesario para generar un cifrado se puede aumentar a medida que aumenta la potencia del hardware.

Uso básico

Se puede cifrar una contraseña llamando al método [make](#) de la [facade Hash](#) :

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Http\Controllers\Controller;

class UpdatePasswordController extends Controller
{
    /**
     * Update the password for the user.
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        // Validate the new password length...

        $request->user()->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();
    }
}

```

El método `make` también le permite administrar el factor de trabajo del algoritmo de cifrado de *bcrypt* usando la opción `rounds`; sin embargo, el valor predeterminado es aceptable para la mayoría de las aplicaciones:

```

$hashed = Hash::make('password', [
    'rounds' => 12
]);

```

Verificar una contraseña contra un cifrado

El método `check` permite verificar que una cadena corresponde con un cifrado concreto. Sin embargo, si está usando [LoginController](#) [incluido con Laravel](#), probablemente no necesitará usar esto directamente, ya que este controlador llama automáticamente a este método:

```
if (Hash::check('plain-text', $hashedPassword)) {  
    // The passwords match...  
}
```

Comprobar si una contraseña requiere re-cifrado

La función `needsRehash` permite determinar si el factor de trabajo utilizado por el cifrado ha cambiado desde que se cifró la contraseña:

```
if (Hash::needsRehash($hashed)) {  
    $hashed = Hash::make('plain-text');  
}
```

Restablecer contraseñas

- # **Introducción**
 - # **Consideraciones de la base de datos**
 - # **Rutas**
 - # **Vistas**
 - # **Tras restablecer contraseñas**
 - # **Personalizacion**
-

Introducción

¿Quiere comenzar rápidamente? Ejecute `php artisan make:auth` en una aplicación de Laravel recién instalada y visite `http://tu-app.dev/register` o cualquier otra URL que esté asignada a su aplicación. Este simple comando gestionará todo el *scaffolding* del sistema de autenticación, ¡incluso el restablecimiento de contraseñas!

La mayoría de las aplicaciones web proporcionan un medio para el restablecimiento de contraseñas olvidadas. En lugar de forzar la re-implementación en cada aplicación, Laravel provee de métodos para enviar recordatorios y restablecimiento de contraseñas.

Antes de usar la característica de Laravel de restablecimiento de contraseñas, su usuario debe utilizar el *trait* `Illuminate\Notifications\Notifiable`.

Consideraciones de la Base de Datos

Para comenzar, verificar que el modelo `App\User` implementa el contrato `Illuminate\Contracts\Auth\CanResetPassword`. Por supuesto, el modelo `App\User` incluido con el framework ya implementa esta interfaz y utiliza el *trait* `Illuminate\Auth\Passwords\CanResetPassword` para incluir los métodos necesarios para cumplir la interfaz.

Generar la migración de la tabla de *tokens* para el restablecimiento

A continuación, se debe crear una tabla para almacenar los *tokens* de restablecimiento de contraseña. La migración de esta tabla se incluye con Laravel y se encuentra en el directorio `database/migrations`. Todo lo que hay que hacer es ejecutar las migraciones de la base de datos:

```
php artisan migrate
```

Rutas

Laravel incluye las clases `Auth\ForgotPasswordController` y `Auth\ResetPasswordController` que contienen la lógica necesaria para enviar por correo los enlaces de restablecimiento de contraseñas y su propio restablecimiento. Todas las rutas necesarias para llevar a cabo la acción se pueden generar utilizando el comando de Artisan `make:auth`:

```
php artisan make:auth
```

Vistas

De nuevo, Laravel generará todas las vistas necesarias para el restablecimiento de contraseñas al ejecutar el comando `make:auth`. Las vistas se almacenarán en `resources/views/auth/passwords`. Se es libre de personalizarlas como se necesite en su aplicación.

Tras restablecer contraseñas

Una vez que ha definido las rutas y vistas para restablecer las contraseñas de los usuarios, se puede acceder a través del navegador en `/password/reset`. El `ForgotPasswordController` que incluye el framework ya proporciona la lógica para enviar los e-mails de restablecimiento, mientras que `ResetPasswordController` incluye la lógica del restablecimiento en sí.

Después de que una contraseña se haya restablecido, el usuario será identificado en la aplicación y redirigido `/home`. Se puede personalizar la redirección tras el restablecimiento de contraseña definiendo la propiedad `redirectTo` en `ResetPasswordController`:

```
protected $redirectTo = '/dashboard';
```

Por defecto, los *tokens* de restablecimiento de contraseña expiran en una hora. Se puede cambiar esto a través de la opción `expire` del archivo `config/auth.php`.

Personalizacion

Personalización del *Guard* de autenticación

En su archivo de configuración `auth.php` se pueden configurar varios "guards", que pueden utilizarse para definir el comportamiento de autenticación para varias tablas de usuarios. Se puede personalizar el `ResetPasswordController` incluido para utilizar el *guard* de su elección reemplazando el método `guard` en el controlador. Este método debe retornar una instancia de un *guard*:

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

Personalizar el *password broker*

En el archivo de configuración `auth.php`, puede configurar varios "brokers" que se podrán utilizar para restablecer contraseñas en sistemas con varias tablas de usuarios. Se puede personalizar `ForgotPasswordController` y `ResetPasswordController` que se incluyen con Laravel para utilizar el *broker* de su elección sobrescribiendo el método `broker`:

```
use Illuminate\Support\Facades\Password;

/**
 * Get the broker to be used during password reset.
 *
 * @return PasswordBroker
 */
protected function broker()
{
    return Password::broker('name');
}
```

Personalizar el e-mail de restablecimiento

Se puede modificar la clase de notificación a utilizar para enviar el enlace de restablecimiento de contraseña al usuario. Para comenzar, sobrescriba el método `sendPasswordResetNotification` en el modelo `User`. En este método puede enviar la notificación utilizando cualquier clase de notificación que desee. El `$token` de restablecimiento de contraseña es el primer parámetro que recibe el método:

```
/**
 * Send the password reset notification.
 *
 * @param string $token
 * @return void
 */
public function sendPasswordResetNotification($token)
{
    $this->notify(new ResetPasswordNotification($token));
}
```

En profundidad

Consola Artisan

Introducción

Crear comandos

- # Generar comandos
- # Estructura de un comando
- # Comandos en funciones anónimas (closure)

Definición de expectativas de entrada

- # Argumentos
- # Opciones
- # Matrices de entradas
- # Descripciones de entrada

E/S del comando

- # Obtener la entrada
- # Solicitud de parámetros de entrada
- # Generar la salida

Registro de los comandos

Programando la ejecución de comandos

- # Ejecutar Comandos desde otros comandos

Introducción

Artisan es el nombre de la interfaz de línea de comandos incluída en Laravel. Ésta proporciona una serie de comandos útiles que podrán ayudar mientras desarrolla su aplicación. Para conocer la lista de todos los comandos Artisan disponibles, usted puede utilizar el comando `list` :

```
php artisan list
```

Cada comando incluye además una ayuda en pantalla que muestra y describe los parámetros y argumentos disponibles para el comando. Para visualizar la ayuda en pantalla, simplemente precede el nombre del comando con `help` :

```
php artisan help migrate
```

Laravel REPL

Todas las aplicaciones Laravel incluyen *Tinker*, una consola (REPL) utilizando el motor del paquete [PsySH](#). Tinker le permite interactuar con toda la aplicación Laravel a través de la línea de comandos, incluyendo *Eloquent ORM*, trabajos, eventos, y más. Para entrar en el entorno Tinker ejecute el comando Artisan `tinker` :

```
php artisan tinker
```

Escribir comandos

Además de los comandos que provee Artisan, usted puede crear también sus propios comandos personalizados. Los comandos se almacenan normalmente en el directorio [app\Console\Commands](#) ; sin embargo, usted es libre de elegir su propia ubicación de almacenamiento siempre y cuando sus comandos puedan ser cargados por *Composer*.

Generar comandos

Para crear un nuevo comando utilice el comando de Artisan `make:command` . Este comando creará una nueva clase *command* en el directorio [app\Console\Commands](#) . No se preocupe si este directorio no existe en la aplicación, ya que éste se creará la primera vez que ejecute el comando Artisan `make:command` . El comando generado incluirá el conjunto predeterminado de propiedades y métodos que están presentes en todos los comandos:

```
php artisan make:command SendEmails
```

Estructura de un comando

Tras generar el comando usted debería llenar las propiedades `signature` y `description` de la clase, las cuales serán usadas cuando se muestre el comando en la lista que obtenemos al ejecutar el comando Artisan `list` . El método `handle` será llamado cuando el comando es ejecutado. Puede colocar la lógica del comando dentro de este método.

Para facilitar la reutilización de código, es una buena práctica mantener los comandos de consola ligeros y permitir delegar la lógica para cumplir sus tareas a los servicios de la aplicación. En el ejemplo que mostramos más abajo se inyecta una clase servicio para hacer la "parte pesada" del envío de los correos electrónicos.

Veamos un comando de ejemplo. Tener en cuenta que somos capaces de inyectar cualquier dependencia que necesitemos dentro del constructor del comando. El [service container](#) de Laravel inyectará automáticamente, en el constructor, todas las dependencias que estén cualificadas por su tipo:

```
<?php

namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'email:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';

    /**
     * The drip e-mail service.
     *
     * @var DripEmailer
     */
    protected $drip;

    /**
     * Create a new command instance.
     *
     * @param DripEmailer $drip
     * @return void
     */
    public function __construct(DripEmailer $drip)
    {
        parent::__construct();
        $this->$drip = $drip;
    }
}
```

```

    }

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $this->drip->send(User::find($this->argument('user')));
    }
}

```

Comandos definidos como *funciones anónimas* (Closure)

Los comandos basados en una función anónima (*Closure*) son una alternativa a definir comandos de consola como clases. Del mismo modo que las rutas definidas como funciones anónimas son la alternativa al uso de controladores, piense que los comandos definidos así son una alternativa a los comandos definidos en una clase. En el interior del método `commands` del fichero `app/Console/Kernel.php`, Laravel carga el fichero `routes/console.php`:

```

    /**
     * Register the Closure based commands for the application.
     *
     * @return void
     */
    protected function commands()
    {
        require base_path('routes/console.php');
    }
}

```

Aunque este fichero no define rutas HTTP, éstas definen puntos de entrada a la consola (rutas) dentro de la aplicación. En el interior de este fichero se pueden definir todos los comandos definidos como funciones anónimas usando el método `Artisan::command`. El método `command` acepta dos argumentos: La [firma del comando](#) y una función anónima que recibe los argumentos y opciones del comando:

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
});
```

La función anónima está enlazada con la instancia de un comando, por tanto, tiene acceso total a todos los métodos de ayuda que tendría normalmente accesibles en una clase de comando.

Sugerencias de tipo para las dependencias

Además de recibir los argumentos y opciones del comando, las funciones anónimas pueden también establecer los tipos para las dependencias adicionales que quisieran ser resueltas a través del [service container](#):

```
use App\User;
use App\DripEmailer;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

Descripciones de comandos definidos como funciones anónimas

Cuando se define un comando a través de una función anónima, usted puede usar el método `describe` para añadir la descripción al comando. Esta descripción será mostrada cuando ejecuten los comandos `php artisan list` o `php artisan help`:

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

Definición de las entradas esperadas

Al escribir comandos de consola, es bastante común obtener datos de entrada desde el usuario a través de argumentos u opciones. Laravel simplifica la definición de las entradas requeridas por el usuario utilizando la propiedad `signature` del comando. La propiedad `signature` le permite definir el nombre, argumentos y opciones del comando con una sintaxis simple y expresiva, similar a la de las rutas.

Argumentos

Todos los argumentos y opciones introducidos por el usuario irán entre llaves. En el siguiente ejemplo, el comando define un argumento obligatorio **required**: `user` :

```
/**  
 * The name and signature of the console command.  
 *  
 * @var string  
 */  
protected $signature = 'email:send {user}';
```

Puede crear argumentos opcionales y definir valores por defecto para los argumentos:

```
// Optional argument...  
email:send {user?}  
  
// Optional argument with default value...  
email:send {user=foo}
```

Opciones

Las opciones, al igual que los argumentos, son otra forma de recoger valores de entrada aportados por el usuario. Las opciones son precedidas por dos guiones (`--`) cuando se especifican en la línea de comandos. Hay dos tipos de opciones: aquellas que reciben un valor y las que no. Las opciones que no reciben un valor funcionan como un "interruptor" lógico. Mire el siguiente ejemplo de este tipo de opción:

```
/**  
 * The name and signature of the console command.  
 *  
 * @var string  
 */  
protected $signature = 'email:send {user} {--queue}';
```

En este ejemplo, la opción `--queue` se puede especificar directamente cuando se ejecuta el comando Artisan. Si se pasa el opción `--queue`, el valor de la opción será `true`. De lo contrario, el valor será `false` :

```
php artisan email:send 1 --queue
```

Opciones con Valores

Ahora, veamos una opción que espera un valor. Si el usuario debe especificar un valor para una opción, ponga = como sufijo de la opción:

```
/**  
 * The name and signature of the console command.  
 *  
 * @var string  
 */  
  
protected $signature = 'email:send {user} {--queue=}';
```

En este ejemplo, el usuario puede pasar un valor para la opción:

```
php artisan email:send 1 --queue=default
```

Puede asignar valores por defecto a las opciones especificándolo después del nombre de la opción. Si el usuario no se pasa ningún valor se usará el el valor por defecto:

```
email:send {user} {--queue=default}
```

Atajos para las opciones

Para asignar un atajo cuando se define una opción, usted puede especificarla antes del nombre de la opción y utilizando el delimitador | para separar el atajo del nombre completo de la opción:

```
email:send {user} {--Q|queue}
```

Matrices de entrada

Si quisiera definir argumentos u opciones que esperan una entrada en forma de matriz, puedes usar el carácter *. Primero, veamos un ejemplo que especifica una matriz como argumento:

```
email:send {user*}
```

Cuando se llame este método, los argumentos de `user` pueden ser pasados, en orden, a la línea de comandos. Por ejemplo, el siguiente comando asignará `['foo', 'bar']` como valor de `user`:

```
php artisan email:send foo bar
```

Cuando definimos una opción que espera una matriz como entrada, cada valor de opción pasado al comando debería ser precedido por el nombre de opción:

```
email:send {user} {--id=*}
```

```
php artisan email:send --id=1 --id=2
```

Descripciones de la entrada

Usted puede asignar descripciones a argumentos y opciones de entrada separando el parámetro de la descripción usando los dos puntos (:). Si necesita un poco más de espacio para definir su comando, siéntase libre de separar la definición en varias líneas:

```
/**  
 * The name and signature of the console command.  
 *  
 * @var string  
 */  
  
protected $signature = 'email:send  
    {user : The ID of the user}  
    {--queue= : Whether the job should be queued}';
```

Comandos de I/O (E/S)

Obtener la entrada

Mientras que se está ejecutando el comando, usted puede, obviamente, acceder a los valores de los argumentos y opciones aceptados por su comando. Para ello, usted puede utilizar los métodos `argument` y `option`:

```
/**  
 * Execute the console command.  
 *  
 * @return mixed  
 */  
public function handle()  
{  
    $userId = $this->argument('user');  
  
    //  
}
```

Si necesita obtener todos los argumentos como una `matriz`, use el método `arguments`:

```
$arguments = $this->arguments();
```

Las opciones pueden ser obtenidas tan fácilmente como los argumentos usando el método `option`. Para obtener las opciones como una matriz, llama al método `options`:

```
// Retrieve a specific option...  
$queueName = $this->option('queue');  
  
// Retrieve all options...  
$options = $this->options();
```

Si el argumento u opción no existe, se retornará `null`.

Solicitud de los parámetros de entrada

Además de mostrar la salida, usted puede, además, solicitar información al usuario durante la ejecución de su comando. El método `ask` preguntará al usuario por información, aceptará sus entradas y entregará esta información al comando:

```
/**  
 * Execute the console command.  
 *  
 * @return mixed  
 */  
public function handle()  
{  
    $name = $this->ask('What is your name?');  
}
```

El método `secret` es parecido a `ask`, pero la entrada del usuario no será visible mientras se escribe en la consola. Este método resulta útil cuando se solicita información sensible como contraseñas:

```
$password = $this->secret('What is the password?');
```

Solicitar confirmación

Si se necesita pedirle al usuario una simple confirmación, puede utilizar el método `confirm`. Por defecto, este método retornará `false`. Si embargo, si el usuario introduce `y` o `yes` como respuesta a la solicitud desde la consola de comandos, el método retornará `true`.

```
if ($this->confirm('Do you wish to continue?')) {  
    //  
}
```

Autocompletado

El método `anticipate` puede ser usado para proveer autocompletado para posibles elecciones. El usuario podrá elegir cualquier respuesta, independientemente de las opciones que se le ofrecen como autocompletado:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Preguntas de elección múltiple

Si necesita dar al usuario un conjunto predeterminado de elecciones, puede usar el método `choice`. Puede especificar el valor por defecto para ser retornado si ninguna opción es seleccionada:

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $default);
```

Generar salida

Para enviar contenido de la salida a la consola, utilice los métodos `line`, `info`, `comment`, `question` y `error`. Cada uno de estos métodos usará colores ANSI apropiados para su cometido. Por ejemplo, permite mostrar información general al usuario. Normalmente, el método `info` mostrará texto verde en la consola:

```
/**  
 * Execute the console command.  
 *  
 * @return mixed  
 */  
  
public function handle()  
{  
    $this->info('Display this on the screen');  
}
```

Para mostrar un mensaje de error, utilice el método `error`. El mensaje de error, normalmente, será mostrado en rojo:

```
$this->error('Something went wrong!');
```

Si desearía mostrar texto plano, sin colorear en la salida de la consola, utilice el método `line`:

```
$this->line('Display this on the screen');
```

Diseño de tablas

El método `table` facilita el correcto formato de dato en múltiples filas / columnas. Simplemente pasa los encabezados y filas al método. El ancho y alto será calculado de forma dinámica en función de los datos:

```
$headers = ['Name', 'Email'];

$users = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $users);
```

Barras de progreso

Para tareas largas, podría ser útil mostrar indicadores de progreso. Utilizando el objeto de salida, podemos comenzar, avanzar y detener la barra de progreso. Primero, define el número total de pasos que recorrer el proceso. Entonces, avance la barra de progreso tras procesar cada paso:

```
$users = App\User::all();

$bar = $this->output->createProgressBar(count($users));

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

Para opciones más avanzadas, eche un vistazo a la [documentación del componente de la Barra de Progreso de Symfony](#).

Registro de comandos

Debido a que el método `load` es llamado en el método `commands` del *kernel* de la consola, todos los comandos en el directorio `app\Console\Commands` serán registrados automáticamente en Artisan. De hecho, usted es libre de hacer llamadas adicionales al método `load` para escanear otros directorios donde tenga comandos para añadir a Artisan:

```
/**  
 * Register the commands for the application.  
 *  
 * @return void  
 */  
  
protected function commands()  
{  
    $this->load(__DIR__ . '/Commands');  
    $this->load(__DIR__ . '/MoreCommands');  
  
    // ...  
}
```

Puede también registrar comandos manualmente añadiendo el nombre de la clase a la propiedad `$commands` del archivo `app\Console\Kernel.php`. Cuando Artisan arranca, todos los comandos enumerados en esta propiedad serán resueltos por el [service container](#) y registrados con Artisan:

```
protected $commands = [  
    Commands\SendEmails::class  
];
```

Programando la ejecución de comandos

A veces puede desear que se ejecute un comando Artisan desde fuera de la consola CLI. Por ejemplo, puede disparar un comando Artisan desde una ruta o controlador. Se puede hacer utilizando el método `call` de la facade `Artisan`. El método `call` acepta el nombre del comando como primer argumento y una matriz de parámetros como segundo argumento. El código de salida será retornado:

```
Route::get('/foo', function () {  
    $exitCode = Artisan::call('email:send', [  
        'user' => 1, '--queue' => 'default'  
    ]);  
  
    //  
});
```

Usando el método `queue` de la *facade Artisan*, puede incluso poner en cola comandos de Artisan que serán procesados en segundo plano por sus [procesadores de colas](#). Antes de usar este método, asegúrese de tener configurado su *cola de trabajos* y de que esté corriendo un procesador de colas:

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});
```

Puede también especificar la conexión o cola a la que el comando de Artisan debe ser lanzado:

```
Artisan::queue('email:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

Pasar matrices como valor

Si su comando define una opción que acepta una matriz, debe simplemente pasar una matriz de valores a esa opción:

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--id' => [5, 13]
    ]);
});
```

Pasar valores booleanos

Si necesita especificar el valor de una opción que no acepta cadenas de texto como valor, tal como el marcador `--force` en el comando `migrate:refresh` usted debería pasar `true` o `false`:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

Ejecutar Comandos desde otros comandos

En ocasiones puede requerir llamar otros comandos desde un comando Artisan existente. Usted puede hacerlo utilizando el método `call`. Este método `call` acepta el nombre del comando y una matriz de parámetros del comando:

```
/**  
 * Execute the console command.  
 *  
 * @return mixed  
 */  
  
public function handle()  
{  
    $this->call('email:send', [  
        'user' => 1, '--queue' => 'default'  
    ]);  
  
    //  
}
```

Si quisiera llamar a otro comando y eliminar cualquier salida que pueda generar, puede utilizar el método `callSilent`. El método `callSilent` tiene la misma firma que el método `call`:

```
$this->callSilent('email:send', [  
    'user' => 1, '--queue' => 'default'  
]);
```

Broadcasting

Introduction

- # Configuration
- # Driver Prerequisites

Concept Overview

- # Using An Example Application

Defining Broadcast Events

- # Broadcast Name
- # Broadcast Data
- # Broadcast Queue
- # Broadcast Conditions

Authorizing Channels

- # Defining Authorization Routes
- # Defining Authorization Callbacks

Broadcasting Events

- # Only To Others

Receiving Broadcasts

- # Installing Laravel Echo
- # Listening For Events
- # Leaving A Channel
- # Namespaces

Presence Channels

- # Authorizing Presence Channels
- # Joining Presence Channels
- # Broadcasting To Presence Channels

Client Events

Notifications

Introduction

In many modern web applications, WebSockets are used to implement realtime, live-updating user interfaces. When some data is updated on the server, a message is typically sent over a WebSocket connection to be handled by the client. This provides a more robust,

efficient alternative to continually polling your application for changes.

To assist you in building these types of applications, Laravel makes it easy to "broadcast" your [events](#) over a WebSocket connection. Broadcasting your Laravel events allows you to share the same event names between your server-side code and your client-side JavaScript application.

Before diving into event broadcasting, make sure you have read all of the documentation regarding Laravel events and listeners.

Configuration

All of your application's event broadcasting configuration is stored in the `config/broadcasting.php` configuration file. Laravel supports several broadcast drivers out of the box: [Pusher](#), [Redis](#), and a `log` driver for local development and debugging. Additionally, a `null` driver is included which allows you to totally disable broadcasting. A configuration example is included for each of these drivers in the `config/broadcasting.php` configuration file.

Broadcast Service Provider

Before broadcasting any events, you will first need to register the `App\Providers\BroadcastServiceProvider`. In fresh Laravel applications, you only need to uncomment this provider in the `providers` array of your `config/app.php` configuration file. This provider will allow you to register the broadcast authorization routes and callbacks.

CSRF Token

[Laravel Echo](#) will need access to the current session's CSRF token. You should verify that your application's `head` HTML element defines a `meta` tag containing the CSRF token:

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

Driver Prerequisites

Pusher

If you are broadcasting your events over [Pusher](#), you should install the Pusher PHP SDK using the Composer package manager:

```
composer require pusher/pusher-php-server "~3.0"
```

Next, you should configure your Pusher credentials in the [config/broadcasting.php](#) configuration file. An example Pusher configuration is already included in this file, allowing you to quickly specify your Pusher key, secret, and application ID. The [config/broadcasting.php](#) file's `pusher` configuration also allows you to specify additional `options` that are supported by Pusher, such as the cluster:

```
'options' => [
    'cluster' => 'eu',
    'encrypted' => true
],
```

When using Pusher and [Laravel Echo](#), you should specify `pusher` as your desired broadcaster when instantiating the Echo instance in your `resources/assets/js/bootstrap.js` file:

```
import Echo from "laravel-echo"

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
});
```

Redis

If you are using the Redis broadcaster, you should install the Predis library:

```
composer require predis/predis
```

The Redis broadcaster will broadcast messages using Redis' pub / sub feature; however, you will need to pair this with a WebSocket server that can receive the messages from Redis and broadcast them to your WebSocket channels.

When the Redis broadcaster publishes an event, it will be published on the event's specified channel names and the payload will be a JSON encoded string containing the event name, a `data` payload, and the user that generated the event's socket ID (if applicable).

Socket.IO

If you are going to pair the Redis broadcaster with a Socket.IO server, you will need to include the Socket.IO JavaScript client library in your application's `head` HTML element. When the Socket.IO server is started, it will automatically expose the client JavaScript library at a standard URL. For example, if you are running the Socket.IO server on the same domain as your web application, you may access the client library like so:

```
<script src="//{{ Request::getHost() }}:6001/socket.io/socket.io.js"></script>
```

Next, you will need to instantiate Echo with the `socket.io` connector and a `host`.

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'socket.io',
    host: window.location.hostname + ':6001'
});
```

Finally, you will need to run a compatible Socket.IO server. Laravel does not include a Socket.IO server implementation; however, a community driven Socket.IO server is currently maintained at the [tlaverdure/laravel-echo-server](#) GitHub repository.

Queue Prerequisites

Before broadcasting events, you will also need to configure and run a [queue listener](#). All event broadcasting is done via queued jobs so that the response time of your application is not seriously affected.

Concept Overview

Laravel's event broadcasting allows you to broadcast your server-side Laravel events to your client-side JavaScript application using a driver-based approach to WebSockets. Currently, Laravel ships with [Pusher](#) and Redis drivers. The events may be easily consumed on the client-side using the [Laravel Echo](#) Javascript package.

Events are broadcast over "channels", which may be specified as public or private. Any visitor to your application may subscribe to a public channel without any authentication or authorization; however, in order to subscribe to a private channel, a user must be authenticated and authorized to listen on that channel.

Using An Example Application

Before diving into each component of event broadcasting, let's take a high level overview using an e-commerce store as an example. We won't discuss the details of configuring [Pusher](#) or [Laravel Echo](#) since that will be discussed in detail in other sections of this documentation.

In our application, let's assume we have a page that allows users to view the shipping status for their orders. Let's also assume that a `ShippingStatusUpdated` event is fired when a shipping status update is processed by the application:

```
event(new ShippingStatusUpdated($update));
```

The `ShouldBroadcast` Interface

When a user is viewing one of their orders, we don't want them to have to refresh the page to view status updates. Instead, we want to broadcast the updates to the application as they are created. So, we need to mark the `ShippingStatusUpdated` event with the `ShouldBroadcast` interface. This will instruct Laravel to broadcast the event when it is fired:

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ShippingStatusUpdated implements ShouldBroadcast
{
    /**
     * Information about the shipping status update.
     *
     * @var string
     */
    public $update;
}
```

The `ShouldBroadcast` interface requires our event to define a `broadcastOn` method. This method is responsible for returning the channels that the event should broadcast on. An empty stub of this method is already defined on generated event classes, so we only need to fill in its details. We only want the creator of the order to be able to view status updates, so we will broadcast the event on a private channel that is tied to the order:

```
/**  
 * Get the channels the event should broadcast on.  
 *  
 * @return array  
 */  
  
public function broadcastOn()  
{  
    return new PrivateChannel('order.'.$this->update->order_id);  
}
```

Authorizing Channels

Remember, users must be authorized to listen on private channels. We may define our channel authorization rules in the `routes/channels.php` file. In this example, we need to verify that any user attempting to listen on the private `order.1` channel is actually the creator of the order:

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {  
    return $user->id === Order::findOrNew($orderId)->user_id;  
});
```

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Listening For Event Broadcasts

Next, all that remains is to listen for the event in our JavaScript application. We can do this using Laravel Echo. First, we'll use the `private` method to subscribe to the private channel. Then, we may use the `listen` method to listen for the `ShippingStatusUpdated` event. By default, all of the event's public properties will be included on the broadcast event:

```
Echo.private(`order.${orderId}`)
    .listen('ShippingStatusUpdated', (e) => {
        console.log(e.update);
    });
});
```

Defining Broadcast Events

To inform Laravel that a given event should be broadcast, implement the [Illuminate\Contracts\Broadcasting\ShouldBroadcast](#) interface on the event class. This interface is already imported into all event classes generated by the framework so you may easily add it to any of your events.

The [ShouldBroadcast](#) interface requires you to implement a single method: [broadcastOn](#). The [broadcastOn](#) method should return a channel or array of channels that the event should broadcast on. The channels should be instances of [Channel](#), [PrivateChannel](#), or [PresenceChannel](#). Instances of [Channel](#) represent public channels that any user may subscribe to, while [PrivateChannels](#) and [PresenceChannels](#) represent private channels that require [channel authorization](#):

```

<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    public $user;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct(User $user)
    {
        $this->user = $user;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('user.' . $this->user->id);
    }
}

```

Then, you only need to [fire the event](#) as you normally would. Once the event has been fired, a [queued job](#) will automatically broadcast the event over your specified broadcast driver.

Broadcast Name

By default, Laravel will broadcast the event using the event's class name. However, you may customize the broadcast name by defining a `broadcastAs` method on the event:

```
/**  
 * The event's broadcast name.  
 *  
 * @return string  
 */  
public function broadcastAs()  
{  
    return 'server.created';  
}
```

If you customize the broadcast name using the `broadcastAs` method, you should make sure to register your listener with a leading `.` character. This will instruct Echo to not prepend the application's namespace to the event:

```
.listen('.server.created', function (e) {  
    ....  
});
```

Broadcast Data

When an event is broadcast, all of its `public` properties are automatically serialized and broadcast as the event's payload, allowing you to access any of its public data from your JavaScript application. So, for example, if your event has a single public `$user` property that contains an Eloquent model, the event's broadcast payload would be:

```
{  
    "user": {  
        "id": 1,  
        "name": "Patrick Stewart"  
        ....  
    }  
}
```

However, if you wish to have more fine-grained control over your broadcast payload, you may add a `broadcastWith` method to your event. This method should return the array of data that you wish to broadcast as the event payload:

```
/**  
 * Get the data to broadcast.  
 *  
 * @return array  
 */  
public function broadcastWith()  
{  
    return ['id' => $this->user->id];  
}
```

Broadcast Queue

By default, each broadcast event is placed on the default queue for the default queue connection specified in your `queue.php` configuration file. You may customize the queue used by the broadcaster by defining a `broadcastQueue` property on your event class. This property should specify the name of the queue you wish to use when broadcasting:

```
/**  
 * The name of the queue on which to place the event.  
 *  
 * @var string  
 */  
public $broadcastQueue = 'your-queue-name';
```

If you want to broadcast your event using the `sync` queue instead of the default queue driver, you can implement the `ShouldBroadcastNow` interface instead of `ShouldBroadcast`:

```
<?php  
  
use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;  
  
class ShippingStatusUpdated implements ShouldBroadcastNow  
{  
    //  
}
```

Broadcast Conditions

Sometimes you want to broadcast your event only if a given condition is true. You may define these conditions by adding a `broadcastWhen` method to your event class:

```
/**  
 * Determine if this event should broadcast.  
 *  
 * @return bool  
 */  
public function broadcastWhen()  
{  
    return $this->value > 100;  
}
```

Authorizing Channels

Private channels require you to authorize that the currently authenticated user can actually listen on the channel. This is accomplished by making an HTTP request to your Laravel application with the channel name and allowing your application to determine if the user can listen on that channel. When using [Laravel Echo](#), the HTTP request to authorize subscriptions to private channels will be made automatically; however, you do need to define the proper routes to respond to these requests.

Defining Authorization Routes

Thankfully, Laravel makes it easy to define the routes to respond to channel authorization requests. In the `BroadcastServiceProvider` included with your Laravel application, you will see a call to the `Broadcast::routes` method. This method will register the `/broadcasting/auth` route to handle authorization requests:

```
Broadcast::routes();
```

The `Broadcast::routes` method will automatically place its routes within the `web` middleware group; however, you may pass an array of route attributes to the method if you would like to customize the assigned attributes:

```
Broadcast::routes($attributes);
```

Defining Authorization Callbacks

Next, we need to define the logic that will actually perform the channel authorization. This is done in the `routes/channels.php` file that is included with your application. In this file, you may use the `Broadcast::channel` method to register channel authorization callbacks:

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

The `channel` method accepts two arguments: the name of the channel and a callback which returns `true` or `false` indicating whether the user is authorized to listen on the channel.

All authorization callbacks receive the currently authenticated user as their first argument and any additional wildcard parameters as their subsequent arguments. In this example, we are using the `{orderId}` placeholder to indicate that the "ID" portion of the channel name is a wildcard.

Authorization Callback Model Binding

Just like HTTP routes, channel routes may also take advantage of implicit and explicit [route model binding](#). For example, instead of receiving the string or numeric order ID, you may request an actual `Order` model instance:

```
use App\Order;

Broadcast::channel('order.{order}', function ($user, Order $order) {
    return $user->id === $order->user_id;
});
```

Broadcasting Events

Once you have defined an event and marked it with the `ShouldBroadcast` interface, you only need to fire the event using the `event` function. The event dispatcher will notice that the event is marked with the `ShouldBroadcast` interface and will queue the event for broadcasting:

```
event(new ShippingStatusUpdated($update));
```

Only To Others

When building an application that utilizes event broadcasting, you may substitute the `event` function with the `broadcast` function. Like the `event` function, the `broadcast` function dispatches the event to your server-side listeners:

```
broadcast(new ShippingStatusUpdated($update));
```

However, the `broadcast` function also exposes the `toOthers` method which allows you to exclude the current user from the broadcast's recipients:

```
broadcast(new ShippingStatusUpdated($update))->toOthers();
```

To better understand when you may want to use the `toOthers` method, let's imagine a task list application where a user may create a new task by entering a task name. To create a task, your application might make a request to a `/task` end-point which broadcasts the task's creation and returns a JSON representation of the new task. When your JavaScript application receives the response from the end-point, it might directly insert the new task into its task list like so:

```
axios.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
 });
```

However, remember that we also broadcast the task's creation. If your JavaScript application is listening for this event in order to add tasks to the task list, you will have duplicate tasks in your list: one from the end-point and one from the broadcast.

You may solve this by using the `toOthers` method to instruct the broadcaster to not broadcast the event to the current user.

Configuration

When you initialize a Laravel Echo instance, a socket ID is assigned to the connection. If you are using [Vue](#) and [Axios](#), the socket ID will automatically be attached to every outgoing request as a `X-Socket-ID` header. Then, when you call the `toOthers` method, Laravel will extract the socket ID from the header and instruct the broadcaster to not broadcast to any connections with that socket ID.

If you are not using Vue and Axios, you will need to manually configure your JavaScript application to send the `X-Socket-ID` header. You may retrieve the socket ID using the `Echo.socketId` method:

```
var socketId = Echo.socketId();
```

Receiving Broadcasts

Installing Laravel Echo

Laravel Echo is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by Laravel. You may install Echo via the NPM package manager. In this example, we will also install the `pusher-js` package since we will be using the Pusher broadcaster:

```
npm install --save laravel-echo pusher-js
```

Once Echo is installed, you are ready to create a fresh Echo instance in your application's JavaScript. A great place to do this is at the bottom of the `resources/assets/js/bootstrap.js` file that is included with the Laravel framework:

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key'
});
```

When creating an Echo instance that uses the `pusher` connector, you may also specify a `cluster` as well as whether the connection should be encrypted:

```
window.Echo = new Echo({
  broadcaster: 'pusher',
  key: 'your-pusher-key',
  cluster: 'eu',
  encrypted: true
});
```

Listening For Events

Once you have installed and instantiated Echo, you are ready to start listening for event broadcasts. First, use the `channel` method to retrieve an instance of a channel, then call the `listen` method to listen for a specified event:

```
Echo.channel('orders')
  .listen('OrderShipped', (e) => {
    console.log(e.order.name);
 });
```

If you would like to listen for events on a private channel, use the `private` method instead. You may continue to chain calls to the `listen` method to listen for multiple events on a single channel:

```
Echo.private('orders')
  .listen(...)
  .listen(...)
  .listen(...);
```

Leaving A Channel

To leave a channel, you may call the `leave` method on your Echo instance:

```
Echo.leave('orders');
```

Namespaces

You may have noticed in the examples above that we did not specify the full namespace for the event classes. This is because Echo will automatically assume the events are located in the `App\Events` namespace. However, you may configure the root namespace when you instantiate Echo by passing a `namespace` configuration option:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-key',
    namespace: 'App.Other.Namespace'
});
```

Alternatively, you may prefix event classes with a `.` when subscribing to them using Echo. This will allow you to always specify the fully-qualified class name:

```
Echo.channel('orders')
    .listen('.Namespace.Event.Class', (e) => {
        //
});
```

Presence Channels

Presence channels build on the security of private channels while exposing the additional feature of awareness of who is subscribed to the channel. This makes it easy to build powerful, collaborative application features such as notifying users when another user is viewing the same page.

Authorizing Presence Channels

All presence channels are also private channels; therefore, users must be [authorized to access them](#). However, when defining authorization callbacks for presence channels, you will not return `true` if the user is authorized to join the channel. Instead, you should return an array of data about the user.

The data returned by the authorization callback will be made available to the presence channel event listeners in your JavaScript application. If the user is not authorized to join the presence channel, you should return `false` or `null`:

```
Broadcast::channel('chat.{roomId}', function ($user, $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

Joining Presence Channels

To join a presence channel, you may use Echo's `join` method. The `join` method will return a `PresenceChannel` implementation which, along with exposing the `listen` method, allows you to subscribe to the `here`, `joining`, and `leaving` events.

```
Echo.join(`chat.${roomId}`)
.here((users) => {
    //
})
.joining((user) => {
    console.log(user.name);
})
.leaving((user) => {
    console.log(user.name);
});
```

The `here` callback will be executed immediately once the channel is joined successfully, and will receive an array containing the user information for all of the other users currently subscribed to the channel. The `joining` method will be executed when a new user joins a channel, while the `leaving` method will be executed when a user leaves the channel.

Broadcasting To Presence Channels

Presence channels may receive events just like public or private channels. Using the example of a chatroom, we may want to broadcast `NewMessage` events to the room's presence channel. To do so, we'll return an instance of `PresenceChannel` from the event's `broadcastOn` method:

```
/**  
 * Get the channels the event should broadcast on.  
 *  
 * @return Channel[]  
 */  
  
public function broadcastOn()  
{  
    return new PresenceChannel('room.'.$this->message->room_id);  
}
```

Like public or private events, presence channel events may be broadcast using the `broadcast` function. As with other events, you may use the `toOthers` method to exclude the current user from receiving the broadcast:

```
broadcast(new NewMessage($message));  
  
broadcast(new NewMessage($message))->toOthers();
```

You may listen for the join event via Echo's `listen` method:

```
Echo.join(`chat.${roomId}`)  
    .here(...)  
    .joining(...)  
    .leaving(...)  
    .listen('NewMessage', (e) => {  
        //  
    });
```

Client Events

Sometimes you may wish to broadcast an event to other connected clients without hitting your Laravel application at all. This can be particularly useful for things like "typing" notifications, where you want to alert users of your application that another user is typing a message on a given screen. To broadcast client events, you may use Echo's `whisper` method:

```
Echo.channel('chat')
    .whisper('typing', {
        name: this.user.name
   });
```

To listen for client events, you may use the `listenForWhisper` method:

```
Echo.channel('chat')
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
   });
```

Notifications

By pairing event broadcasting with [notifications](#), your JavaScript application may receive new notifications as they occur without needing to refresh the page. First, be sure to read over the documentation on using [the broadcast notification channel](#).

Once you have configured a notification to use the broadcast channel, you may listen for the broadcast events using Echo's `notification` method. Remember, the channel name should match the class name of the entity receiving the notifications:

```
Echo.private(`App.User.${userId}`)
    .notification((notification) => {
        console.log(notification.type);
   });
```

In this example, all notifications sent to `App\User` instances via the `broadcast` channel would be received by the callback. A channel authorization callback for the `App.User.{id}` channel is included in the default `BroadcastServiceProvider` that ships with the Laravel framework.

Cache

Configuración

Prerrequisitos del *driver*

Uso de Caché

Obtener una instancia de Caché

Recuperar elementos de la Caché

Almacenar elementos en Caché

Quitar elementos de Caché

El helper Caché

Etiquetas de Caché

Almacenar elementos etiquetados en caché

Acceder a elementos etiquetados en caché

Eliminar elementos etiquetados en caché

Añadir *drivers* personalizados

Escribir el *driver*

Registrar el *driver*

Eventos

Configuración

Laravel incorpora un API unificado para varios sistemas de caché. La configuración de caché se encuentra en `config/cache.php`. En este archivo se puede especificar qué *driver* utilizar por defecto para la aplicación. Laravel soporta de serie los sistemas de caché más populares como [Memcached](#) y [Redis](#).

El archivo de configuración de caché incluye además otras opciones, las cuales se encuentran documentadas y es recomendable leer. Por defecto, Laravel viene configurado para utilizar el *driver* `file`, el cual almacena objetos serializados en el sistema de archivos. Para aplicaciones más grandes, se recomienda utilizar un controlador más robusto como Memcached o Redis. Se pueden establecer varias configuraciones para un mismo *driver*.

Prerrequisitos del *driver*

Base de datos

Cuando se utiliza el *driver database*, es necesario crear una tabla que contendrá los elementos cacheados. Este es un ejemplo del *Schema* de esta tabla:

```
Schema::create('cache', function ($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

También se puede usar el comando de Artisan `php artisan cache:table` para generar una migración con el esquema apropiado.

Memcached

El uso del controlador Memcached requiere la instalación del paquete [Memcached PECL](#).

Puede listar todos sus servidores Memcached en el archivo de configuración

`config/cache.php` :

```
'memcached' => [
    [
        'host' => '127.0.0.1',
        'port' => 11211,
        'weight' => 100
    ],
],
```

También se puede establecer la opción `host` a un socket UNIX. De ser así, la opción `port` debe establecerse en ``:

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

Redis

Antes de utilizar Reids con Laravel, es necesario instalar el paquete `predis/predis` (~1.0) a través de Composer o instalar la extensión de PHP PhpRedis vía PECL.

Para más información sobre cómo configurar Redis, consultar su [documentación en Laravel](#).

Uso de Caché

Obtener una Instancia de Caché

Los [contracts](#) `Illuminate\Contracts\Cache\Factory` y `Illuminate\Contracts\Cache\Repository` proporcionan acceso a los servicios de caché de Laravel. El contrato `Factory` permite acceder a todos los *drivers* de caché definidos en la aplicación. El contrato `Repository` es normalmente una implementación del *driver* de caché por defecto de la aplicación especificado por el archivo de configuración `cache`.

Sin embargo, también se puede utilizar la *facade* `Cache`, la cual se utilizará a lo largo de esta documentación. La *facade* `Cache` proporciona un acceso cómodo y preciso a las implementaciones subyacentes de los contratos de caché de Laravel:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

Acceder a varios sistemas de caché

Utilizando la *facade Cache*, se puede acceder a los diferentes sistemas de caché utilizando el método `store`. La clave pasada al método `store` corresponderá con una de las listadas en el array `stores` del archivo de configuración `cache`:

```
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 10);
```

Recuperar elementos de la Caché

El método `get` de la *facade Cache* se utiliza para obtener elementos desde la caché. Si el elemento no existe en la caché, se retornará `null`. Si lo desea, puede pasar un segundo argumento al método `get` especificando el valor predeterminado que desea que se devuelva si el elemento no existe:

```
$value = Cache::get('key');

$value = Cache::get('key', 'default');
```

Se puede incluso pasar un *Closure* como valor por defecto. Se retornará el resultado de este *Closure* si el elemento especificado no existe en la caché. Pasar un *Closure* permite recuperar valores desde una base de datos o incluso un servicio externo:

```
$value = Cache::get('key', function () {
    return DB::table(...)->get();
});
```

Comprobar la existencia de un elemento

Se puede utilizar el método `has` para determinar si existe un elemento en la caché. Este método devuelve `false` si el valor es `null` o `false`:

```
if (Cache::has('key')) {
    //
}
```

Incrementar/decrementar valores

Los métodos `increment` y `decrement` se utilizan para ajustar el valor de elementos enteros en la caché. Ambos métodos aceptan un segundo argumento opcional que indica la cantidad por la cual se debe incrementar o disminuir el valor del elemento:

```
Cache::increment('key');  
Cache::increment('key', $amount);  
Cache::decrement('key');  
Cache::decrement('key', $amount);
```

Recuperar & almacenar

A veces se desea recuperar un elemento de la caché, pero a su vez almacenar un valor por defecto si ese elemento no existe. Por ejemplo, obtener todos los usuarios desde la caché o, si no existen, recuperarlos de la base de datos y añadirlos a la caché. Para ello se puede utilizar el método `Cache::remember`:

```
$value = Cache::remember('users', $minutes, function () {  
    return DB::table('users')->get();  
});
```

Si el elemento no existe en la caché, el `Closure` pasado al método `remember` se ejecutará y se almacenará su resultado.

Puede utilizar el método `rememberForever` para recuperar un elemento de la caché o guardarlo para siempre:

```
$value = Cache::rememberForever('users', function() {  
    return DB::table('users')->get();  
});
```

Recuperar & borrar

Si necesita recuperar un elemento de la caché y luego eliminarlo, puede utilizar el método `pull`. Como el método `get`, retornará `null` si el elemento no existe:

```
$value = Cache::pull('key');
```

Almacenar elementos en caché

Para almacenar elementos en la caché se puede utilizar el método `put` de la facade `Cache`. Cuando se coloca un elemento en la caché, es necesario especificar el número de minutos que el valor debe estar almacenado:

```
Cache::put('key', 'value', $minutes);
```

En lugar de pasar el número de minutos como un entero, también puede pasar una instancia `DateTime` representando la hora/fecha de caducidad del elemento almacenado en caché:

```
$expiresAt = Carbon::now()->addMinutes(10);

Cache::put('key', 'value', $expiresAt);
```

Almacenar si no está presente

El método `add` únicamente añadirá un elemento a la caché si no existe previamente. Este método retornará `true` si el elemento ya existe en la caché. De otro modo, retornará `false`:

```
Cache::add('key', 'value', $minutes);
```

Almacenar elementos indefinidamente

El método `forever` puede utilizarse para almacenar un elemento en la caché permanentemente. Dado que estos elementos no caducarán, deben eliminarse manualmente de la caché utilizando el método `forget`:

```
Cache::forever('key', 'value');
```

Si está utilizando el driver Memcached, los elementos que se almacenan "para siempre" pueden eliminarse cuando la caché alcanza su límite de tamaño.

Borrar elementos de la Caché

Puede eliminar elementos de la caché utilizando el método `forget`:

```
Cache::forget('key');
```

Y además se puede limpiar por completo la caché utilizando el método `flush`:

```
Cache::flush();
```

La purga de la caché no respeta el prefijo de caché y eliminará todas las entradas. Tenga mucha precaución al limpiar una caché que pueda estar siendo compartida por otras aplicaciones.

El helper Caché

Además de utilizar la *facade* `Cache` o el [contrato caché](#), también puede utilizar la función global `cache` para recuperar y almacenar datos. Cuando se llama a la función `cache` con una cadena como único argumento, retornará el valor para esa clave:

```
$value = cache('key');
```

Si se proporciona un *array* de pares clave/valor y un tiempo de caducidad a la función, almacenará los valores en la caché durante el tiempo especificado:

```
cache(['key' => 'value'], $minutes);  
  
cache(['key' => 'value'], Carbon::now()->addSeconds(10));
```

Al "testear" la llamada a la función global `cache`, puede utilizar el método `Cache::shouldReceive` como si estuviera testeando un facade.

Etiquetas de Caché

Las etiquetas de caché no son compatibles cuando se utilizan los *drivers* de caché `file` o `database`. Además, cuando se utilizan varias etiquetas con cachés que se almacenan "forever", el rendimiento aumenta con *drivers* como `memcached`, que automáticamente purgan registros obsoletos.

Almacenar elementos etiquetados en caché

Las etiquetas de caché permiten etiquetar elementos relacionados en la caché y luego eliminar todos los valores almacenados que han sido asignados a una etiqueta determinada. Se puede acceder a una caché etiquetada pasando un *array* ordenado de nombres de etiquetas. Por ejemplo, para acceder a una caché etiquetada y hacer `put` (almacenar) de un valor en la caché:

```
Cache::tags(['people', 'artists'])->put('John', $john, $minutes);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $minutes);
```

Acceder a elementos etiquetados en caché

Para recuperar un elemento de caché etiquetado, pase la misma lista ordenada de etiquetas al método `tags` y llame al método `get` con la clave que desee recuperar:

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

Eliminar elementos etiquetados en caché

Se pueden eliminar todos los elementos asignados a una etiqueta o lista de etiquetas. Por ejemplo, esta declaración eliminaría todas las cachés etiquetadas con `people`, `authors` o ambos. Por lo que tanto `Anne` como `John` se eliminarían de la caché:

```
Cache::tags(['people', 'authors'])->flush();
```

En contraste, esta declaración eliminaría sólo las cachés marcadas con `authors`, por lo que `Anne` se eliminaría, pero no `John`:

```
Cache::tags('authors')->flush();
```

Añadir drivers personalizados

Escribir el *driver*

Para crear un controlador de caché personalizado, es necesario implementar el [contrato Illuminate\Contracts\Cache\Store](#). Por lo que una implementación de caché con MongoDB tendría este aspecto:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys);
    public function put($key, $value, $minutes) {}
    public function putMany(array $values, $minutes);
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

Únicamente es necesario implementar cada uno de estos métodos utilizando una conexión MongoDB. Para ver un ejemplo de cómo implementar cada uno de estos métodos, revise [Illuminate\Cache\MemcachedStore](#) en el código fuente del framework. Una vez que la implementación esté completa, se puede registrar el *driver* personalizado.

```
Cache::extend('mongo', function ($app) {
    return Cache::repository(new MongoStore());
});
```

Si se está preguntando dónde colocar el código de su *driver* personalizado, puede crear un *namespace* llamado `Extensions` dentro de su directorio `app`. Tenga en cuenta que Laravel no posee una estructura de aplicación rígida y que se es libre de organizar la aplicación como de acuerdo con sus preferencias.

Registrar el *driver*

Para registrar el *driver* de caché personalizado en Laravel puede utilizar el método `extend` en la *facade* `Cache`. La llamada a `Caché::extend` podría hacerse en el método `boot` de `App\Providers\AppServiceProvider` que se incluye con las nuevas aplicaciones de Laravel, o puede crear su propio *service provider* para alojar la extensión - no olvide registrar el *provider* en el *array* de *providers* del archivo `config/app.php`:

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function ($app) {
            return Cache::repository(new MongoStore());
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

El primer argumento del método `extend` será el nombre del *driver*. Este además debe corresponder con la opción `driver` del archivo de archivo de configuración `config/cache.php`. El segundo argumento será un *Closure* que debe devolver una instancia de `Illuminate\Cache\Repository`. El *Closure* recibirá una instancia `$app`, que es una instancia del service container.

Una vez que su extensión esté registrada, simplemente actualice la opción `driver` en el archivo de configuración `config/cache.php` con el nombre de su extensión.

Eventos

Para ejecutar código en cada operación de caché, se pueden capturar los eventos lanzados por la caché. Normalmente, estos *listeners* se suelen ubicar en el `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
        'App\Listeners\LogCacheHit',
    ],
    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],
    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],
    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];
```

Colecciones

Introducción

- # Crear colecciones
- # Extender colecciones

Métodos disponibles

Nivel superior de mensajes

Introducción

La clase `Illuminate\Support\Collection` proporciona un encapsulador muy fluido para trabajar con *arrays* de datos. Por ejemplo, revisar el siguiente código. Se utiliza el helper `collect` para crear una nueva instancia de collection desde un *array*, ejecutar la función `strtoupper` sobre cada elemento y eliminar cualquier elemento vacío:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {  
    return strtoupper($name);  
})  
->reject(function ($name) {  
    return empty($name);  
});
```

Como se puede apreciar, la clase `Collection` permite encadenar estos métodos para llevar a cabo mapeos y reducción del *array* de datos subyacente a la colección. En general, las colecciones son inmutables, lo que quiere decir que cada método de `Collection` retorna una instancia completamente nueva de `Collection`.

Crear colecciones

Como ya se ha mencionado, el helper `collect` retorna una nueva instancia de `Illuminate\Support\Collection` para el *array* dado. Por lo tanto, crear una colección es tan simple como:

```
$collection = collect([1, 2, 3]);
```

Los resultados de consultas Eloquent se devuelven siempre como instancias de `Collection`.

Extender colecciones

Las colecciones son "macroables", por lo que permiten añadir métodos adicionales a la clase `Collection` en tiempo de ejecución. Por ejemplo, el siguiente código añade el método `toUpper` a la clase `Collection`:

```
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function ($value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Normalmente se deben declarar las *macros* en un [service provider](#).

Métodos disponibles

Para el resto de este documento, se describirá cada uno de los métodos disponibles para la clase `Collection`. Recordar, todos estos métodos se pueden encadenar para manipular el array subyacente. Además, en prácticamente cada método se retorna una instancia de `Collection`, permitiendo preservar la copia original de la colección cuando sea necesario:

[all](#)

[average](#)

[avg](#)

[chunk](#)

[collapse](#)

[combine](#)

[concat](#)

[contains](#)

[containsStrict](#)

count	macro	shuffle
crossJoin	make	slice
dd	map	sort
diff	mapInto	sortBy
diffAssoc	mapSpread	sortByDesc
diffKeys	mapToGroups	splice
dump	mapWithKeys	split
each	max	sum
eachSpread	median	take
every	merge	tap
except	min	times
filter	mode	toArray
first	nth	toJson
flatMap	only	transform
flatten	pad	union
flip	partition	unique
forget	pipe	uniqueStrict
forPage	pluck	unless
get	pop	unwrap
groupBy	prepend	values
has	pull	when
implode	push	where
intersect	put	whereStrict
intersectByKeys	random	whereIn
isEmpty	reduce	whereInStrict
isNotEmpty	reject	whereNotIn
keyBy	reverse	whereNotInStrict
keys	search	wrap
last	shift	zip

Lista de métodos

[all\(\)](#)

El método `all` retorna el array subyacente que representa la colección:

```
collect([1, 2, 3])->all();
```

```
// [1, 2, 3]
```

average()

Alias para el método [avg](#).

avg()

El método [avg](#) retorna la [media aritmética](#) de una clave concreta:

```
$average = collect(['foo' => 10, 'foo' => 10, 'foo' => 20, 'foo' => 40])->avg('foo');  
// 20  
  
$average = collect([1, 1, 2, 4])->avg();  
// 2
```

chunk()

El método [chunk](#) divide la colección en varias colecciones más pequeñas del tamaño especificado:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);  
  
$chunks = $collection->chunk(4);  
  
$chunks->toArray();  
  
// [[1, 2, 3, 4], [5, 6, 7]]
```

Este método es especialmente útil en [vistas](#) cuando se trabaja con un sistema de *grid* como [Bootstrap](#). Imaginar que se tiene una colección de modelos [Eloquent](#) y se quiere mostrar en un grid:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

collapse()

El método `collapse` colapsa una colección de *arrays* en una colección simple y plana:

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

combine()

El método `combine` combina las claves de la colección con valores de otro *array* o colección:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

concat()

El método `concat` anexa los valores del *array* o colección dados al final de la colección:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

contains()

El método `contains` determina si la colección contiene un elemento concreto:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

También se puede pasar un par de clave / valor al método `contains`, el cual determinará si el par existe en la colección:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

Finalmente, se puede pasar un *callback* al método `contains` para realizar un test propio:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->contains(function ($value, $key) {  
    return $value > 5;  
});  
  
// false
```

El método `contains` utiliza comparaciones "débiles" al comprobar los valores de los elementos, por lo que una cadena con un valor entero se considerará igual a un entero del mismo valor. Utilizar el método `containsStrict` para filtrar utilizando comparaciones "estrictas".

containsStrict()

Este método tiene la misma firma que el método `contains` ; sin embargo, todos los valores se comparan utilizando operadores "estrictos".

count()

El método `count` retorna el total de elementos en la colección:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->count();  
  
// 4
```

crossJoin()

El método `crossJoin` mezclará los valores de la colección con el *array* o colección dado, retornando un producto cartesiano con todas las permutaciones posibles:

```

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/*
[
    [1, 'a'],
    [1, 'b'],
    [2, 'a'],
    [2, 'b'],
]
*/

```



```

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
[
    [1, 'a', 'I'],
    [1, 'a', 'II'],
    [1, 'b', 'I'],
    [1, 'b', 'II'],
    [2, 'a', 'I'],
    [2, 'a', 'II'],
    [2, 'b', 'I'],
    [2, 'b', 'II'],
]
*/

```

dd()

El método `dd` vuelca el contenido de la colección y finaliza con la ejecución del script:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/
```

Si no quiere terminar con la ejecución del script, utilice el método [dump](#) .

diff()

El método [diff](#) compara la colección con otra colección o un [array](#) de PHP nativo basado en sus valores. Esto método retornará los valores de la colección original que no estén presentes en la colección dada:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

diffAssoc()

El método [diffAssoc](#) compara la colección con otra colección o un [array](#) de PHP nativo basado en sus claves y valores. Este método retornará el par de clave/valor en la colección original que no están presentes en la colección dada:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

diffKeys()

El método `diffKeys` compara la colección con otra colección o un `array` de PHP nativo basado en sus claves. Este método retornará el par de clave/valor en la colección original que no están presentes en la colección dada:

```

$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]

```

dump()

El método `dump` vuelca el contenido de los elementos de la colección:

```

$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/

```

Si se desea dejar de ejecutar un *script* tras volcar la colección, utilizar el método `dd` en su lugar.

each()

El método `each` itera sobre los elementos de la colección y pasa cada uno de ellos por un *callback*:

```
$collection = $collection->each(function ($item, $key) {  
    //  
});
```

Para detener la iteración de elementos, se puede devolver `false` desde el *callback*:

```
$collection = $collection->each(function ($item, $key) {  
    if /* some condition */ {  
        return false;  
    }  
});
```

eachSpread()

El método `eachSpread` itera sobre los elementos de la colección, pasando el valor de cada elemento hijo al *callback* dado:

```
$collection = collect(['John Doe', 35], ['Jane Doe', 33]);  
  
$collection->eachSpread(function ($name, $age) {  
    //  
});
```

Se puede detener la iteración de elementos retornando `false` desde el *callback*:

```
$collection->eachSpread(function ($name, $age) {  
    return false;  
});
```

every()

El método `every` se puede utilizar para verificar que todos los elementos de una colección pasan una prueba de verdad:

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});

// false
```

except()

El método `except` retorna todos los elementos de la colección excepto aquellos especificados:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

El inverso del método `except` es el método `only`.

filter()

El método `filter` filtra la colección utilizando un *callback*, únicamente manteniendo aquellos elementos que pasan una prueba de verdad:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

Si no se proporciona un *callback*, todas las entradas de la colección que sean equivalentes a `false` serán eliminadas:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);  
  
$collection->filter()->all();  
  
// [1, 2, 3]
```

Para el inverso de `filter`, ver el método [reject](#).

first()

El método `first` retorna el primer elemento de la colección que cumpla la condición:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {  
    return $value > 2;  
});  
  
// 3
```

También se puede llamar a este método `first` sin argumentos para obtener el primer elemento de la colección. Si la colección está vacía se retornará `null`:

```
collect([1, 2, 3, 4])->first();  
  
// 1
```

flatMap()

El método `flatMap` itera la colección y pasa cada valor por un *callback* concreto. El *callback* es libre de modificar el elemento y retornarlo, permitiendo formar así una nueva colección de elementos modificados. A continuación el *array* se aplana un nivel:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten()

El método `flatten` aplana una colección multidimensional en una única dimensión:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

Opcionalmente se puede pasar un argumento de "profundidad":

```

$collection = collect([
    'Apple' => [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ],
    'Samsung' => [
        ['name' => 'Galaxy S7', 'brand' => 'Samsung']
    ],
]);

```

```

$products = $collection->flatten(1);

$products->values()->all();

/*
[
    ['name' => 'iPhone 6S', 'brand' => 'Apple'],
    ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
]
*/

```

En este ejemplo, llamar a `flatten` sin proporcionar la profundidad aplanará también los *arrays* anidados, resultando en `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Proporcionar la profundidad permite restringir los niveles de *arrays* anidados a aplanar.

flip()

El método `flip` alterna las claves de la colección con sus valores correspondientes:

```

$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']

```

forget()

El método `forget` elimina un elemento de la colección por su clave:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

collectionView->forget('name');

collectionView->all();

// ['framework' => 'laravel']
```

Al contrario que la mayoría de los métodos de las colecciones, `forget` no retorna una colección nueva; modifica la colección en la que se llama.

forPage()

El método `forPage` retorna una nueva colección conteniendo los elementos que estarían presentes en una página concreta. El método acepta el número de página como primer argumento y el número de elementos a mostrar por página como segundo parámetro:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

get()

El método `get` retorna el elemento de una clave determinada. Si la clave no existe, retorna `null`:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

Opcionalmente se puede pasar el valor por defecto como segundo parámetro:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('foo', 'default-value');

// default-value
```

Incluso se puede pasar un *callback* como valor por defecto. El resultado del *callback* se retornará si la clave no existe:

```
$collection->get('email', function () {
    return 'default-value';
});

// default-value
```

groupBy()

El método `groupBy` agrupa los elementos de la colección por una clave dada:

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->toArray();

/*
[
    'account-x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'account-x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/
```

Además de pasar una cadena `clave`, se puede pasar un *callback*. El *callback* retornará el valor de la clave para agrupar:

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});

$grouped->toArray();

/*
[
    'x10' => [
        ['account_id' => 'account-x10', 'product' => 'Chair'],
        ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ],
    'x11' => [
        ['account_id' => 'account-x11', 'product' => 'Desk'],
    ],
]
*/
```

has()

El método `has` determina si una clave existe en la colección:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);

$collection->has('product');

// true
```

implode()

El método `implode` unirá los elementos en una colección. Sus argumentos dependen del tipo de elementos en la colección. Si la colección contiene arrays u objetos, se debe pasar la clave de los atributos a unir y la cadena de "unión" que se establecerá entre los valores:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
])];

$collection->implode('product', ', ');

// Desk, Chair
```

Si la colección contiene únicamente cadenas o valores numéricos, simplemente hay que pasar la "unión" como único argumento:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

intersect()

El método `intersec` elimina cualquier valor de la colección original que no esté presente en un `array` o colección dado. La colección resultante preservará las claves de la colección original:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

intersectByKeys()

El método `intersectByKeys` elimina cualquier clave de la colección original que no esté presente en el `array` o colección dados:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009
]);

$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

isEmpty()

El método `isEmpty` retornará `true` si la colección está vacía, de otro modo, retornará `false`:

```
collect([])->isEmpty();

// true
```

isNotEmpty()

El método `isNotEmpty` reotrnará `true` si la colección no está vacía; de otro modo, `false`:

```
collect([])->isNotEmpty();

// false
```

keyBy()

El método `keyBy` establece la clave de la colección por una clave concreta. Si hay varios elementos que tienen la misma clave, solo una aparecerá en la nueva colección:

```

$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);
$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
[
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/

```

También se puede pasar un *callback* al método. El *callback* debe retornar el valor de la clave que se desea establecer:

```

$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
[
    'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]
*/

```

keys()

El método `keys` retorna todas las claves de la colección:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);
$keys = $collection->keys();
$keys->all();
// ['prod-100', 'prod-200']
```

last()

El método `last` retorna el último elemento de la colección en cumplir el filtro:

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});
// 2
```

También se puede llamar al método `last` sin argumentos para obtener el último elemento de la colección. Si la colección está vacía se retornará `null`:

```
collect([1, 2, 3, 4])->last();
// 4
```

macro()

El método estático `macro` permite añadir métodos a la clase `Collection` en tiempo de ejecución. Diríjase a la documentación de [extender colecciones](#) para más información.

make()

El método estático `make` crea una nueva instancia de colección. Ver la sección [crear colecciones](#).

map()

El método `map` itera la colección y pasa cada elemento al *callback* proporcionado. El *callback* es capaz de modificar el elemento y retornarlo, formando así una nueva colección de elementos modificados:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$multiplied = $collection->map(function ($item, $key) {  
    return $item * 2;  
});  
  
$multiplied->all();  
  
// [2, 4, 6, 8, 10]
```

Como la mayoría de los métodos de la colección, `map` retorna una nueva instancia de la colección; no modifica la colección en la que se ha llamado. Si se desea transformar la colección original, utilizar le método `transform`.

mapInto()

El método `mapInto` itera la colección, creando una nueva instancia de la clave dada pasando el valor al constructor:

```

class Currency
{
    /**
     * Create a new currency instance.
     *
     * @param string $code
     * @return void
     */
    function __construct(string $code)
    {
        $this->code = $code;
    }
}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]

```

mapSpread()

El método `mapSpread` itera la colección pasando el valor de cada hijo al *callback* proporcionado. El *callback* es capaz de modificar el elemento y retornarlo, formando así una nueva colección de elementos modificados:

```

$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($odd, $even) {
    return $odd + $even;
});

$sequence->all();

// [1, 5, 9, 13, 17]

```

mapToGroups()

El método `mapToGroups` agrupa los elementos de la colección por un *callback*. El *callback* debe retornar un *array* asociativo contenido un par clave/valor, formando así una nueva colección de valores agrupados:

```
$collection = collect([
    [
        'name' => 'John Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Jane Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Johnny Doe',
        'department' => 'Marketing',
    ]
]);

$grouped = $collection->mapToGroups(function ($item, $key) {
    return [$item['department'] => $item['name']];
});

$grouped->toArray();

/*
[
    'Sales' => ['John Doe', 'Jane Doe'],
    'Marketing' => ['Johnny Doe'],
]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']
```

mapWithKeys()

El método `mapWithKeys` itera la colección y pasa cada valor al *callback* dado. El *callback* debe retornar un *array* asociativo conteniendo un único par de clave/valor:

```
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com'
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com'
    ]
]);

$keyed = $collection->mapWithKeys(function ($item) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
[
    'john@example.com' => 'John',
    'jane@example.com' => 'Jane',
]
*/
```

max()

El método `max` retorna el valor máximo de una clave dada:

```
$max = collect([['foo' => 10], ['foo' => 20]])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

median()

El método `median` retorna el valor medio de una clave concreta:

```
$median = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->median('foo')  
  
// 15  
  
$median = collect([1, 1, 2, 4])->median();  
  
// 1.5
```

merge()

El método `merge` unifica el *array* o colección dado con la colección original. Si alguna cadena de clave coincide con otra de la colección original, el valor de los elementos se sobrescribirá por el valor de la colección original:

```
$collection = collect(['product_id' => 1, 'price' => 100]);  
  
$merged = $collection->merge(['price' => 200, 'discount' => false]);  
  
$merged->all();  
  
// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

Si las claves de los elementos proporcionados son numéricas, los valores se añadirán al final de la colección:

```
$collection = collect(['Desk', 'Chair']);  
  
$merged = $collection->merge(['Bookcase', 'Door']);  
  
$merged->all();  
  
// ['Desk', 'Chair', 'Bookcase', 'Door']
```

min()

El método `min` retorna el valor mínimo de una clave concreta:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

mode()

El método `mode` retorna el valor de moda de una clave concreta:

```
$mode = collect([['foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40]])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]
```

nth()

El método `nth` crea una nueva colección basado en la serie dada:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

También se puede pasar un <0>offset como segundo argumento:

```
$collection->nth(4, 1);

// ['b', 'f']
```

only()

El método `only` retorna los elementos de la colección con las claves pasadas:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => false]);  
  
$filtered = $collection->only(['product_id', 'name']);  
  
$filtered->all();  
  
// ['product_id' => 1, 'name' => 'Desk']
```

El inverso de `only` es el método [except](#).

pad()

El método `pad` rellenará el *array* con el valor dado hasta que el *array* llegue a un tamaño concreto. Este método actua justo como la función de PHP [array_pad](#).

Para llenar hacia la izquierda, se debe especificar un tamaño negativo. No se hará ningún relleno si el valor absoluto proporcionado es menor o igual a la longitud del *array*:

```
$collection = collect(['A', 'B', 'C']);  
  
$filtered = $collection->pad(5, 0);  
  
$filtered->all();  
  
// ['A', 'B', 'C', 0, 0]  
  
$filtered = $collection->pad(-5, 0);  
  
$filtered->all();  
  
// [0, 0, 'A', 'B', 'C']
```

partition()

El método `partition` se puede combinar con la función `lists` de PHP para separar elementos que pasan una prueba de verdad de aquellos que no lo hacen:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

list($underThree, $aboveThree) = $collection->partition(function ($i) {
    return $i < 3;
});
```

pipe()

El método `pipe` pasa la colección por un *callback* y retorna el resultado:

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

pluck()

El método `pluck` obtiene todos los valores para una clave dada:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

Además se puede especificar qué clave se debe utilizar para la colección resultante:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop()

El método `pop` elimina y retorna el último elemento de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

prepend()

El método `prepend` añade un elemento al principio de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

También se puede pasar un segundo parámetro para establecer la clave del elemento a añadir:

```
$collection = collect(['one' => 1, 'two' => 2]);  
  
$collection->prepend(0, 'zero');  
  
$collection->all();  
  
// ['zero' => 0, 'one' => 1, 'two' => 2]
```

pull()

El método `pull` elimina y retorna un elemento de la colección por su clave:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);  
  
$collection->pull('name');  
  
// 'Desk'  
  
$collection->all();  
  
// ['product_id' => 'prod-100']
```

push()

El método `push` añade un elemento al final de la colección:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->push(5);  
  
$collection->all();  
  
// [1, 2, 3, 4, 5]
```

put()

El método `put` establece una clave y valor en la colección:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

collectionView->put('price', 100);

collectionView->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random()

El método `random` retorna un elemento aleatorio de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);

collectionView->random();

// 4 - (retrieved randomly)
```

De forma opcional, se puede pasar un entero a `random` para especificar cuantos elementos se desea obtener de forma aleatoria. Siempre se retornará una colección de elementos cuando explícitamente se pase el número de elementos que se desea obtener:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

reduce()

El método `reduce` reduce la colección a un único valor, pasando el resultado de cada iteración a la iteración subsecuente:

```
$collection = collect([1, 2, 3]);  
  
$total = $collection->reduce(function ($carry, $item) {  
    return $carry + $item;  
});  
  
// 6
```

El valor de `$carry` en la primera iteración es `null`; sin embargo, se puede especificar su valor inicial pasándole un segundo argumento a `reduce`:

```
$collection->reduce(function ($carry, $item) {  
    return $carry + $item;  
}, 4);  
  
// 10
```

reject()

El método `reject` filtra la colección utilizando un *callback*. El *callback* debe retornar `true` si el elemento debe eliminarse de la colección resultante:

```
$collection = collect([1, 2, 3, 4]);  
  
$filtered = $collection->reject(function ($value, $key) {  
    return $value > 2;  
});  
  
$filtered->all();  
  
// [1, 2]
```

El inverso del método `reject` es el método `filter`.

reverse()

El método `reverse` invierte el orden de los elementos de la colección, preservando las claves originales:

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/*
[
    4 => 'e',
    3 => 'd',
    2 => 'c',
    1 => 'b',
    0 => 'a',
]
*/

```

search()

El método de `search` busca en la colección el valor dado y devuelve su clave si la encuentra. Si el elemento no se encuentra, se devuelve `false`.

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

La búsqueda se realiza utilizando una comparación "débil" (*loose*), por lo que una cadena con un entero se considerará igual que un entero del mismo valor. Para utilizar una comparación "estricta", indicar `true` como segundo parámetro:

```
$collection->search('4', true);

// false
```

De forma alternativa, se puede pasar un *callback* propio a la búsqueda para encontrar el primer elemento que cumpla la condición:

```
$collection->search(function ($item, $key) {  
    return $item > 5;  
});  
  
// 2
```

shift()

El método `shift` elimina y retorna el primer elemento de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->shift();  
  
// 1  
  
$collection->all();  
  
// [2, 3, 4, 5]
```

shuffle()

El método `shuffle` mezcla al azar los elementos de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$shuffled = $collection->shuffle();  
  
$shuffled->all();  
  
// [3, 2, 5, 1, 4] - (generated randomly)
```

slice()

El método `slice` retorna una parte de la colección comenzando por un índice dado:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
  
$slice = $collection->slice(4);  
  
$slice->all();  
  
// [5, 6, 7, 8, 9, 10]
```

Se puede pasar el tamaño como segundo argumento al método:

```
$slice = $collection->slice(4, 2);  
  
$slice->all();  
  
// [5, 6]
```

La porción resultante preservará las claves por defecto. Si no se desea preservar las claves originales, se puede utilizar el método [values](#) para *reindexar* la colección.

sort()

El método [sort](#) ordena la colección. La colección ordenada mantiene las claves originales, por lo que en este ejemplo se utilizará el método [values](#) para restablecer las claves a índices consecutivos:

```
$collection = collect([5, 3, 1, 2, 4]);  
  
$sorted = $collection->sort();  
  
$sorted->values()->all();  
  
// [1, 2, 3, 4, 5]
```

Si las necesidades de orden son más avanzadas, se puede pasar un *callback* a [sort](#) con un algoritmo propio. Refiérase a la documentación de PHP sobre [uasort](#), la cual usa el método [sort](#) bajo el capó.

Si se necesita ordenar una colección de *arrays* u objetos anidados, ver los métodos `sortBy` y `sortByDesc`.

sortBy()

El método `sortBy` ordena la colección por una clave dada. La colección ordenada mantiene las claves originales, por lo que en este ejemplo se utilizará el método `values` para restablecer las claves a índices consecutivos:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
[
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
    ['name' => 'Desk', 'price' => 200],
]
*/
```

También se puede pasar un *callback* propio para determinar como ordenar los valores de la colección:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);
$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});
$sorted->values()->all();
/*
[
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]
*/
```

sortByDesc()

Este método tiene la misma firma que el método de [sortBy](#), pero ordenará a la colección en orden inverso.

splice()

El método [splice](#) elimina y retorna una porción de elementos comenzando desde un índice específico:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2);  
  
$chunk->all();  
  
// [3, 4, 5]  
  
$collection->all();  
  
// [1, 2]
```

Se puede pasar un segundo argumento para limitar el tamaño de la porción resultante:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2, 1);  
  
$chunk->all();  
  
// [3]  
  
$collection->all();  
  
// [1, 2, 4, 5]
```

Además, un tercero que contiene los nuevos elementos para reemplazar a los eliminados de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2, 1, [10, 11]);  
  
$chunk->all();  
  
// [3]  
  
$collection->all();  
  
// [1, 2, 10, 11, 4, 5]
```

split()

El método `split` divide la colección en tantos grupos como se indique:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$groups = $collection->split(3);  
  
$groups->toArray();  
  
// [[1, 2], [3, 4], [5]]
```

sum()

El método `sum` suma todos los elementos de la colección:

```
collect([1, 2, 3, 4, 5])->sum();  
  
// 15
```

Si la colección contiene *arrays* anidados u objetos, se debe pasar una clave para determinar qué valores se han de sumar:

```
$collection = collect([  
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],  
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],  
]);  
  
$collection->sum('pages');  
  
// 1272
```

Además, se puede pasar un *callback* propio para determinar qué elementos sumar:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
});

// 6
```

take()

El método `take` retorna una colección con un número específico de elementos:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

También se puede pasar un valor negativo para tomar esa cantidad de elementos desde el final de la colección:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

tap()

El método `tap` pasa la colección por un *callback* proporcionado, permitiendo "tocar/alterar" la colección en un punto concreto y hacer algo con sus elementos sin afectar a la colección en sí misma:

```
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function ($collection) {
        Log::debug('Values after sorting', $collection->values()->toArray());
    })
    ->shift();

// 1
```

times()

El método estático `times` crea una nueva colección invocando el *callback* la cantidad de veces dada:

```
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

Este método puede ser útil al combinarlo con factorías para crear modelos [Eloquent](#):

```
$categories = Collection::times(3, function ($number) {
    return factory(Category::class)->create(['name' => 'Category #' . $number]);
});

(categories->all();

/*
[
    ['id' => 1, 'name' => 'Category #1'],
    ['id' => 2, 'name' => 'Category #2'],
    ['id' => 3, 'name' => 'Category #3'],
]
*/

```

toArray()

El método `toArray` convierte la colección en un `array` PHP plano. Si los elementos de la colección son modelos [Eloquent](#), los modelos también se convertirán a `arrays`:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
[
    ['name' => 'Desk', 'price' => 200],
]
*/

```

`toArray` convierte además los objetos anidados a un `array`. Si se desea obtener el `array` subyacente, utilizar el método `all` en su lugar.

toJson()

El método `toJson` convierte la colección en una cadena JSON *serializada*:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);  
  
$collection->toJson();  
  
// '{"name": "Desk", "price": 200}'
```

transform()

El método `transform` itera sobre todos los elementos de la colección y llama al *callback* con cada uno de ellos. Los elementos de la colección se reemplazarán con los valores retornados por el *callback*:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->transform(function ($item, $key) {  
    return $item * 2;  
});  
  
$collection->all();  
  
// [2, 4, 6, 8, 10]
```

Al contrario que la mayoría de los métodos de las colecciones, `transform` modifica la colección en si misma. Para crear una nueva colección, utilizar el método `map` en su lugar.

union()

El método `union` añade el *array* proporcionado a la colección. Si el *array* contiene claves que ya existen en la colección, se preferirán los valores de la colección original:

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['b']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

unique()

El método `unique` retorna todos los elementos únicos de la colección. La colección resultante mantendrá las claves originales, por lo que en este ejemplo se utilizará el método `values` para restablecer las claves a valores consecutivos:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

Cuando se trata con *arrays* anidados u objetos, se puede especificar la clave que determina la unicidad:

```

$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

```

```

$unique = $collection->unique('brand');

$unique->values()->all();

/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
]
*/

```

También se puede pasar un *callback* propio para determinar la unicidad:

```

$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
[
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]
*/

```

El método `unique` utiliza una comparación "débil" (*loose*) al comprobar los valores de los elementos, por lo que una cadena con un valor entero se considerará igual que un entero del mismo valor. Utilice el método `uniqueStrict` para filtrar utilizando comparaciones "estrictas".

`uniqueStrict()`

Este método tiene la misma firma que el método [unique](#) ; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

`unless()`

El método `unless` ejecutará el *callback* dado a no ser que el primer argumento del método se resuelva a `true` :

```
$collection = collect([1, 2, 3]);  
  
$collection->unless(true, function ($collection) {  
    return $collection->push(4);  
});  
  
$collection->unless(false, function ($collection) {  
    return $collection->push(5);  
});  
  
$collection->all();  
  
// [1, 2, 3, 5]
```

El inverso de `unless` sería el método [when](#) .

`unwrap()`

El método estático `unwrap` retorna los elementos subyacentes de la colección para el valor dado cuando aplique:

```
Collection::unwrap(collect('John Doe'));

// ['John Doe']

Collection::unwrap(['John Doe']);

// ['John Doe']

Collection::unwrap('John Doe');

// 'John Doe'
```

values()

El método `values` retorna una colección con las claves restablecidas a enteros consecutivos:

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200]
]);

$values = $collection->values();

$values->all();

/*
[
    0 => ['product' => 'Desk', 'price' => 200],
    1 => ['product' => 'Desk', 'price' => 200],
]
*/
```

when()

El método `when` ejecutará el *callback* proporcionado cuando el primer argumento se resuelva a `true`:

```

$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
    return $collection->push(4);
});

$collection->when(false, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 4]

```

El inverso de `when` sería el método `unless`.

where()

El método `where` filtra la colección por un par de clave / valor:

```

$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/

```

El método `where` utiliza una comparación "débil" (*loose*) al comprobar los valores de los elementos, por lo que una cadena con un valor entero se considerará igual que un entero del mismo valor. Utilice el método `whereStrict` para filtrar utilizando comparaciones "estrictas".

whereStrict()

Este método tiene la misma firma que el método `where`; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

whereIn()

El método `whereIn` filtra la colección por un par clave/valor dado contenido dentro del array de elementos:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
[
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Desk', 'price' => 200],
]
*/
```

El método `whereIn` utiliza una comparación "débil" (*loose*) al comprobar los valores de los elementos, por lo que una cadena con un valor entero se considerará igual que un entero del mismo valor. Utilice el método `whereInStrict` para filtrar utilizando comparaciones "estrictas".

whereInStrict()

Este método tiene la misma firma que el método [whereIn](#) ; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

whereNotIn()

El método [whereNotIn](#) filtra la colección por un par clave/valor dado que no exista dentro del *array* de elementos:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Door', 'price' => 100],
]
*/
```

El método [whereNotIn](#) utiliza una comparación "débil" (*loose*) al comprobar los valores de los elementos, por lo que una cadena con un valor entero se considerará igual que un entero del mismo valor. Utilice el método [whereNotInStrict](#) para filtrar utilizando comparaciones "estrictas".

whereNotInStrict()

Este método tiene la misma firma que el método [whereNotIn](#) ; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

wrap()

El método `wrap` envuelve el valor proporcionado en una colección si es posible:

```
$collection = Collection::wrap('John Doe');

$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

zip()

El método `zip` mezcla juntos los valores del *array* dado con los valores de la colección original basado en su índice:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Nivel superior de mensajes

Las colecciones soportan además "niveles superiores de mensajes" o "Higher Order Messages (Hom)", los cuales son atajos para ejecutar acciones comunes en colecciones. Los métodos de la colección que soportan esta característica son: `average` , `avg` , `contains` , `each` , `every` ,

```
filter , first , flatMap , map , partition , reject , sortBy , sortByDesc , y sum .
```

Cada mensaje se puede acceder como una propiedad dinámica en una instancia de una colección. Por ejemplo, el mensaje `each` se puede utilizar para llamar a un método por cada elemento en la colección:

```
$users = User::where('votes', '>', 500)->get();  
  
$users->each->markAsVip();
```

Del mismo modo, se puede utilizar el mensaje de nivel superior `sum` para calcular el número total de votos de una colección de usuarios:

```
$users = User::where('group', 'Development')->get();  
  
return $users->sum->votes;
```

Eventos

Introducción

Registrar eventos & *listeners*

- # Generar eventos & *listeners*
- # Registrar eventos manualmente

Definir eventos

Definir *listeners*

Colas de *listeners*

- # Acceder a la cola manualmente
- # Gestionar trabajos (*jobs*) fallidos

Disparar eventos

Suscriptores de eventos

- # Escribir suscriptores de eventos
- # Registrar suscriptores de eventos

Introducción

Los eventos de Laravel proveen una implementación *observer*, permitiendo suscribir y capturar varios eventos que ocurren en la aplicación. Las clases de eventos se almacenan normalmente en `app/Events` , mientras que sus *listeners* (escuchadores) se almacenan en `app/Listeners` . No se preocupe si no encuentra estos directorios en la aplicación, puesto que se crearán tan pronto como comience a generar eventos y *listeners* utilizando los comandos de Artisan.

Los eventos son una buena forma de desacoplar varios aspectos de la aplicación, puesto que un único evento puede tener varios *listeners* que no dependan de otros. Por ejemplo, se puede enviar una notificación de Slack a un usuario cada vez que se envía un pedido. En lugar de acoplar el procesamiento del pedido a las notificaciones de Slack, se puede simplemente lanzar un evento `OrderShipped` , el cual puede recibir un *listener* y transformarlo en una notificación de Slack.

Registrar eventos & *listeners*

El `EventServiceProvider` que incluye Laravel es un buen lugar para registrar los *listeners* de eventos de toda su aplicación. La propiedad `listen` contiene un *array* de todos los eventos (claves) y sus *listeners* (valores). Por supuesto, se pueden añadir tantos eventos al *array* como sea necesario. Por ejemplo, para añadir un evento `OrderShipped` :

```
/**  
 * The event listener mappings for the application.  
 *  
 * @var array  
 */  
  
protected $listen = [  
    'App\\Events\OrderShipped' => [  
        'App\Listeners\SendShipmentNotification',  
    ],  
];
```

Generar eventos & *listeners*

Por supuesto, la creación de los archivos para eventos y *listeners* resulta tediosa. En su lugar, simplemente añade *listeners* y eventos al `EventServiceProvider` y ejecuta el comando `event:generate`. Este comando generará cualquier evento o *listener* listado en el `EventServiceProvider`. Por supuesto, los eventos y *listeners* que ya existan quedarán intactos:

```
php artisan event:generate
```

Registrar eventos manualmente

Normalmente, los eventos se registrarán a través del *array* `$listen` en `EventServiceProvider`; sin embargo, se puede registrar eventos basados en *Closures* de forma manual en el método `boot` del `EventServiceProvider`:

```
/**  
 * Register any other events for your application.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    parent::boot();  
  
    Event::listen('event.name', function ($foo, $bar) {  
        //  
    });  
}
```

Comodín de captura de eventos

Se pueden registrar *listeners* utilizando `*` como parámetro comodín, permitiendo capturar varios eventos en el mismo *listener*. Los *listeners* con comodín reciben el nombre del evento como primer parámetro y el *array* de datos del array como segundo argumento:

```
Event::listen('event.*', function ($eventName, array $data) {  
    //  
});
```

Definir eventos

Una clase de evento es simplemente un contenedor de datos que incluye información relacionada con el evento. Por ejemplo, asuma que el evento `OrderShipped` generado recibe un objeto [Eloquent](#):

```

<?php

namespace App\Events;

use App\Order;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use SerializesModels;

    public $order;

    /**
     * Create a new event instance.
     *
     * @param Order $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
}

```

Como puede observar, el evento no contiene lógica. Es un contenedor simple para la instancia de `Order` que se compró. El trait `SerializesModels` utilizado por el evento *serializará* cualquier modelo Eloquent si el objeto del evento se *serializa* utilizando la función de PHP `serialize`.

Definir *listeners*

A continuación, echemos un vistazo al *listener* del evento de ejemplo. Los *listeners* reciben la instancia del evento en el método `handle`. El comando `event:generate` importará automáticamente la clase adecuada y incluirán el *type-hint* del evento en el método `handle`. En el método `handle` se puede ejecutar cualquier acción necesaria para responder al evento:

```

<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param OrderShipped $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        // Access the order using $event->order...
    }
}

```

Los *listeners* de eventos pueden además incluir *type-hints* de cualquier dependencia que necesiten en sus constructores. Todos los *listeners* se resuelven a través del service container de Laravel, por lo que las dependencias se inyectarán automáticamente.

Detener la propagación de un evento

A veces, es necesario detener la propagación de un evento a otros *listeners*. Se puede hacer retornando `false` desde el método `handle` del *listener*.

Colas de *listeners*

Añadir la ejecución de un *listener* a una cola puede ser beneficioso si el *listener* va a ejecutar alguna tarea lenta como enviar un correo electrónico o peticiones HTTP. Antes de comenzar con colas de *listeners*, asegúrese de [configurar su cola](#) y comenzar un *queue listener* en su servidor local o entorno de desarrollo.

Para especificar que un *listener* debe incluirse en una cola, debe añadir la interfaz `ShouldQueue` a la clase del mismo. Los *listeners* generados por el comando Artisan `event:generate` ya tienen esta interfaz importada en el *namespace* y pueden utilizarla directamente:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //
}
```

¡Eso es todo! Ahora cuando se llame a un *listener* para un evento, se añadirá directamente a una cola por el *event dispatcher* (disparador de eventos) utilizando el [sistema de colas](#) (*queue system*) de Laravel. Si no se lanza ninguna excepción durante la ejecución del *listener*, el trabajo se eliminará de la cola una vez que haya concluido su procesamiento.

Personalizar la conexión de colas & nombre de cola

Si desea personalizar la conexión para la cola y el nombre que utiliza un *listener*, se pueden definir las propiedades `$connection` y `$queue` de la clase *listener*:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * The name of the connection the job should be sent to.
     *
     * @var string|null
     */
    public $connection = 'sq';

    /**
     * The name of the queue the job should be sent to.
     *
     * @var string|null
     */
    public $queue = 'listeners';
}
```

Acceder a la cola manualmente

Si necesita acceder a los métodos `delete` y `release` de la cola de trabajo subyacente del *listener*, puede hacerlo utilizando el *trait* `Illuminate\Queue\InteractsWithQueue`. Este *trait* se importa por defecto en los *listeners* generados y proporciona acceso a estos métodos:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param \App\Events\OrderShipped $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

Gestionar trabajos (*jobs*) fallidos

En ocasiones, las colas de *listeners* pueden fallar. Si un *listener* de una cola supera el número máximo de intentos definidos por el *queue worker*, se ejecutará el método `failed` del *listener*. El método `failed` recibe la instancia del evento y la excepción que causó el fallo:

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param \App\Events\OrderShipped $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        //
    }

    /**
     * Handle a job failure.
     *
     * @param \App\Events\OrderShipped $event
     * @param \Exception $exception
     * @return void
     */
    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

Disparar eventos

Para lanzar un evento, se puede pasar una instancia del evento al helper `event`. El helper disparará el evento a todos los *listeners* registrados. Puesto que el helper `event` está disponible globalmente, se puede llamar desde cualquier parte de la aplicación:

```
<?php

namespace App\Http\Controllers;

use App\Order;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param int $orderId
     * @return Response
     */
    public function ship($orderId)
    {
        $order = Order::findOrFail($orderId);

        // Order shipment logic...

        event(new OrderShipped($order));
    }
}
```

Cuando se ejecutan *tests*, puede ser útil afirmar (*assert*) que se dispararon ciertos eventos sin necesidad de ejecutar sus *listeners*. Los helpers incluidos hacen esto muy sencillo.

Suscriptores de Eventos – *Subscribers*

Escribir suscriptores de eventos

Los suscriptores de eventos son clases que pueden suscribirse a varios eventos desde la propia clase, permitiendo definir varios controladores de eventos en una misma clase. Los suscriptores deben definir un método `subscribe`, el cual recibirá una instancia de un *event dispatcher*. Se puede llamar al método `listen` en el *dispatcher* dado para registrar *listeners* (capturadores/escuchadores) de eventos:

```
<?php

namespace App\Listeners;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function onUserLogin($event) {}

    /**
     * Handle user logout events.
     */
    public function onUserLogout($event) {}

    /**
     * Register the listeners for the subscriber.
     *
     * @param Illuminate\Events\Dispatcher $events
     */
    public function subscribe($events)
    {
        $events->listen(
            'Illuminate\Auth\Events\Login',
            'App\Listeners\UserEventSubscriber@onUserLogin'
        );

        $events->listen(
            'Illuminate\Auth\Events\Logout',
            'App\Listeners\UserEventSubscriber@onUserLogout'
        );
    }
}
```

Registrar suscriptores de eventos

Tras escribir el *subscriber*, ya está listo para registrarlo con el *event dispatcher* (disparador de eventos). Puede registrar *subscribers* utilizando la propiedad `$subscribe` en `EventServiceProvider`. Por ejemplo, para añadir el `UserEventSubscriber` a la lista:

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}
```

Almacenamiento de archivos – *File storage*

Introducción

Configuración

- # El disco *public*
- # El *driver* local
- # Prerrequisitos del *driver*

Obtener Instancias de discos

Obtener archivos

- # URLs de archivos
- # Metadatos de archivos

Almacenar archivos

- # Subida de archivos
- # Visibilidad de archivos

Eliminar archivos

Directories

Filesystems personalizados

Introducción

Laravel provee una potente abstracción para *filesystem* (manejo de archivos) gracias al paquete de PHP de Frank de Jonge [Flysystem](#). La integración de Flysystem de Laravel provee *drivers* sencillos para trabajar con sistemas de archivos locales, Amazon S3, y Rackspace Cloud. Incluso mejor, es increíblemente sencillo alternar entre estas opciones de almacenamiento, pues el API se mantiene constante para cada sistema.

Configuración

La configuración de *filesystem* se encuentra en el fichero `config/filesystems.php`. En este archivo se pueden configurar todos los "discos". Cada disco representa un *driver* de almacenamiento y una ubicación en particular. En el archivo de configuración se incluyen varios ejemplos para cada *driver* soportado. Así pues, simplemente hay que modificar la configuración para reflejar las preferencias de almacenamiento y credenciales.

Por supuesto, se pueden configurar tantos discos como sea necesario, e incluso varios discos para un mismo *driver*.

El disco *public*

El disco `public` está previsto para archivos que van a ser de acceso público. Por defecto, el uso del disco `public` usa el *driver* `local` y almacena estos archivos en `storage/app/public`. Para acceder desde la web, debe crear un enlace simbólico desde `public/storage` a `Storage/app/public`. Esta convención mantendrá sus archivos de acceso público en un directorio que se puede compartir fácilmente en las implementaciones cuando se utilizan sistemas de implementación sin tiempo de inactividad como [Envoyer](#).

Para crear el link simbólico, puede usar el comando Artisan `storage:link`:

```
php artisan storage:link
```

Por supuesto, una vez que se ha almacenado un archivo y se ha creado el link simbólico, puede crear una URL para los archivos utilizando el *helper* `asset`:

```
echo asset('storage/file.txt');
```

El *driver local*

Cuando se utiliza el *driver* `local`, hay que tener en cuenta que todas las operaciones son relativas al directorio `root` definido en el archivo de configuración. Por defecto, este valor se establece en el directorio `storage/app`. Por lo tanto, el siguiente comando almacenará un archivo en `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Pre requisitos del *driver*

Paquetes composer

Antes de utilizar los *drivers* S3 o Rackspace, es necesario instalar el paquete apropiado vía Composer:

- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`

- Rackspace: `league/flysystem-rackspace ~1.0`

Configuración del *driver S3*

La información de configuración del *driver S3* se encuentra localizada en el archivo de configuración `config/database.php`. Este archivo contiene un ejemplo de *array* de configuración para el *driver S3*. Podrá modificar este *array* con su propia configuración y credenciales de S3. Por conveniencia, estas variables de entorno coinciden con la convención de nomenclatura utilizada por AWS CLI.

Configuración FTP

Las integraciones de Laravel en Flysystem funcionan muy bien con FTP; sin embargo, no se incluye ninguna configuración de ejemplo en el archivo de configuración `filesystems.php`. Si necesita configurar el *filesystem* FTP, puede usar el siguiente ejemplo de configuración:

```
'ftp' => [
    'driver'    => 'ftp',
    'host'      => 'ftp.example.com',
    'username'  => 'your-username',
    'password'  => 'your-password',

    // Optional FTP Settings...
    // 'port'      => 21,
    // 'root'      => '',
    // 'passive'   => true,
    // 'ssl'       => true,
    // 'timeout'  => 30,
],
]
```

Configuración del *driver Rackspace*

Las integraciones Flysystem de Laravel funcionan muy bien con Rackspace; sin embargo, no se incluye una configuración de ejemplo en el archivo de configuración `filesystems.php` del framework. Si necesita configurar el *filesystem* Rackspace, puede usar el siguiente ejemplo de configuración:

```
'rackspace' => [
    'driver'      => 'rackspace',
    'username'    => 'your-username',
    'key'         => 'your-key',
    'container'   => 'your-container',
    'endpoint'    => 'https://identity.api.rackspacecloud.com/v2.0/',
    'region'      => 'IAD',
    'url_type'   => 'publicURL',
],
`
```

Obtener Instancias de discos

La *facade Storage* se puede utilizar para interactuar con cualquiera de los discos configurados. Por ejemplo, se puede utilizar el método `put` de la *facade* para almacenar un avatar en el disco por defecto. Si se llaman a métodos en la *facade Storage* sin haber llamado primero al método `disk`, la llamada al método se pasará automáticamente al disco por defecto:

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $fileContents);
```

Si la aplicación interacciona con varios discos, se puede utilizar el método `disk` en la *facade Storage* para trabajar con los archivos de un disco en particular:

```
Storage::disk('s3')->put('avatars/1', $fileContents);
```

Obtener archivos

El método `get` se utiliza para obtener el contenido de un archivo. El método devolverá una cadena sin formato con el contenido del archivo. Recordar, todas las rutas a archivos deben especificarse de forma relativa a la "raíz" configurada para el disco:

```
$contents = Storage::get('file.jpg');
```

El método `exists` puede utilizarse para determinar si existe un archivo en el disco:

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

URLs de archivos

Puede usar el método `url` para obtener la URL para un archivo dado. Si está usando el *driver local*, normalmente antepondrá `/storage` a la ruta y retornará la URL relativa del archivo. Si está usando el *driver s3* ó `rackspace`, se retornará la URL remota completa:

```
use Illuminate\Support\Facades\Storage;  
  
$url = Storage::url('file1.jpg');
```

nota Recuerde, si está usando el *driver local*, todos los archivos que deberían ser de acceso público deben colocarse en el directorio `storage/app/public`. Además, deberá [crear un enlace simbólico](#the-public-disk) que apuntará al directorio `storage/app/public`.

URLs temporales

Para archivos almacenados usando los *driver s3* o `rackspace`, puede crearse una URL temporal para un archivo dado usando el método `temporaryUrl`. Este método acepta una ruta e instancia `DateTime` especificando cuando debería expirar la URL:

```
$url = Storage::temporaryUrl(  
    'file1.jpg', Carbon::now()->addMinutes(5)  
>;
```

Personalizar la URL del servidor local

Si quiere predefinir el *host* para los archivos almacenados en un disco usando el *driver local*, puede añadir la opción `url` al array de configuración del disco:

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

Metadatos de los archivos

Además de leer y escribir archivos, Laravel permite obtener información sobre los propios archivos. Por ejemplo, el método `size` puede utilizarse para obtener el tamaño del archivo en bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file1.jpg');
```

El método `lastModified` retorna el UNIX timestamp de la última vez que el archivo fue modificado:

```
$time = Storage::lastModified('file1.jpg');
```

Almacenar archivos

El método `put` puede usarse para almacenar el contenido de archivos raw en un disco. Además se puede pasar un `recurso` de PHP al método `put`, el cual utilizará el soporte para *stream* subyacente en Flysystem. Es recomendable utilizar *streams* cuando se trabaja con archivos grandes:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

Streaming automático

Si desea que Laravel administre automáticamente el *streaming* de un archivo concreto en una ubicación de almacenamiento, puede utilizar el método `putFile` o el método `putFileAs`. Este método acepta tanto una instancia `Illuminate\Http\File` como de `Illuminate\Http\UploadedFile` y automáticamente enviará el archivo a la ubicación deseada:

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

// Automatically generate a unique ID for file name...
Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a file name...
Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg');
```

Hay varias cuestiones importantes a tener en cuenta sobre el método `putFile`: Tenga en cuenta que únicamente hay que especificar el nombre de un directorio, no el nombre de un archivo. Por defecto, el método `putFile` generará un ID único equivalente al nombre del archivo. El método `putFile` devolverá la ruta del archivo, incluyendo el nombre del archivo, por lo que podría utilizarla para almacenarla en su base de datos.

Los métodos `putFile` y `putFileAs` también aceptan un argumento para especificar la "visibilidad" del archivo almacenado. Esto es particularmente útil si se está almacenando el archivo en un disco en la nube como S3 y se necesita que dicho archivo sea accesible públicamente:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

Añadir datos al principio o al final de archivos

Los métodos `prepend` y `append` permiten escribir al principio o al final de un archivo:

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

Copiar & mover archivos

El método `copy` puede usarse para copiar un archivo existente en una nueva ubicación del disco, mientras que el método `move` puede utilizarse para renombrar o mover un archivo existente a una nueva ubicación:

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

Subida de archivos

En las aplicaciones web, uno de los casos de uso más comunes de almacenamiento de archivos es almacenar los archivos subidos por los usuarios, tales como imágenes, fotos y documentos. Laravel permite almacenar los archivos subidos fácilmente usando el método `store` para una instancia de archivo subido. Simplemente hay que llamar al método `store` con la trayectoria en la que deseas almacenar el archivo subido:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     *
     * @param Request $request
     * @return Response
     */
    public function update(Request $request)
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

Hay varias cuestiones importantes a tener en cuenta en este ejemplo. Nótese que sólo se ha especificado un nombre de directorio, no un nombre de archivo. Por defecto, el método `store` generará un ID único equivalente al nombre del archivo. El método `store` devolverá la ruta del archivo, y así podrá utilizarse esta, incluyendo el nombre de archivo generado, para almacenarla en su base de datos.

También puede llamar al método `putFile` de la *facade Storage* para realizar la misma manipulación del archivo como en el siguiente ejemplo:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Especificar un nombre de archivo

Si no quiere asignar un nombre automáticamente al archivo almacenado, puede utilizar el método `storeAs`, el cual recibe la ruta, el nombre del archivo y (opcionalmente) el disco como argumentos:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

Por supuesto, puede usar el método `putFileAs` de la *facade Storage*, el cual permitirá la misma manipulación de archivos como en el ejemplo anterior:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

Especificar un disco

Por defecto, este método usará el disco predeterminado. Si quiere especificar otro disco, debe pasar el nombre del disco como segundo argumento en el método `store`:

```
$path = $request->file('avatar')->store(
    'avatars/'. $request->user()->id, 's3'
);
```

Visibilidad de archivos

En la integración de Flysystem en Laravel, "visibilidad" es una abstracción de los permisos de archivo para múltiples plataformas. Los archivos pueden ser declarados `public` o `private`. Cuando un archivo es declarado `public`, se está indicando que el archivo debería ser accesible para otros. Por ejemplo, cuando se usa el *driver* de S3, puede obtener las URLs de los archivos `public`.

Puede seleccionar la visibilidad cuando configura el archivo a través del método `put` :

```
use Illuminate\Support\Facades\Storage;  
  
Storage::put('file.jpg', $contents, 'public');
```

Si el archivo ya ha sido almacenado, su visibilidad puede obtenerse o seleccionarse a través de los métodos `getVisibility` y `setVisibility` respectivamente:

```
$visibility = Storage::getVisibility('file.jpg');  
  
Storage::setVisibility('file.jpg', 'public')
```

Eliminar archivos

El método `delete` acepta un archivo o un *array* de archivos a eliminar del disco:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::delete('file.jpg');  
  
Storage::delete(['file1.jpg', 'file2.jpg']);
```

Si es necesario, se puede especificar el disco en el que se debería eliminar el archivo:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::disk('s3')->delete('folder_path/file_name.jpg');
```

Directorios

Obtener todos los archivos de un directorio

El método `files` retorna un *array* de todos los archivos en un directorio concreto. Para recuperar una lista de todos los archivos de un directorio, incluyendo sus sub-directorios, se puede utilizar el método `allFiles` :

```
use Illuminate\Support\Facades\Storage;  
  
$files = Storage::files($directory);  
  
$files = Storage::allFiles($directory);
```

Obtener todos los directorios de un directorio

El método `directories` retorna un *array* de todos los directorios dentro de un directorio. Además, se puede utilizar el método `allDirectories` para obtener una lista de todos los directorios de un directorio y sus sub-directorios:

```
$directories = Storage::directories($directory);  
  
// Recursive...  
$directories = Storage::allDirectories($directory);
```

Crear un directorio

El método `makeDirectory` creará el directorio, incluyendo cualquier sub-directorio que se necesite:

```
Storage::makeDirectory($directory);
```

Eliminar un directorio

Finalmente, se puede usar `deleteDirectory` para eliminar un directorio y todos sus archivos:

```
Storage::deleteDirectory($directory);
```

Filesystems personalizados

La integración de Flysystem de Laravel provee *drivers* para varios "drivers"; sin embargo, Flysystem no está limitado a estos y posee adaptadores para otros sistemas de almacenamiento. Se puede crear un *driver* personalizado si se necesita alguno de estos adaptadores adicionales en una aplicación Laravel.

Para configurar un sistema de archivos a medida se necesitará un adaptador Flysystem. Se puede añadir un adaptador de Dropbox mantenido por la comunidad a un proyecto utilizando:

```
composer require spatie/flysystem-dropbox
```

Puede crear un [service provider](#) llamado `DropboxServiceProvider`. En el método `boot` del *provider*, se puede utilizar el método `extend` de la *facade* `Storage` para definir el *driver*:

```
<?php

namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Illuminate\Support\ServiceProvider;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class DropboxServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function ($app, $config) {
            $client = new DropboxClient(
                $config['authorizationToken']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

El primer parámetro del método `extend` es el nombre del *driver* y el segundo un *Closure* que recibirá las variables `$app` y `$config`. El resultado del *Closure* debe retornar una instancia de `League\Flysystem\Filesystem`. La variable `$config` contiene los valores definidos en `config/filesystems.php` para el disco especificado.

Una vez creado el *service provider*, para registrar la extensión se puede utilizar el *driver* `dropbox` en el archivo de configuración `config/filesystems.php`.

Helpers

Introducción

Métodos disponibles

Introducción

Laravel incluye una gran variedad de funciones "*helper*". Muchas de estas funciones son utilizadas por el propio framework; sin embargo, es libre de usarlas en sus propias aplicaciones si lo considera conveniente.

Métodos disponibles

Arrays & objetos

[array_add](#)

[array-collapse](#)

[array_divide](#)

[array_dot](#)

[array_except](#)

[array_first](#)

[array_flatten](#)

[array_forget](#)

[array_get](#)

[array_has](#)

[array_last](#)

[array_only](#)

[array_pluck](#)

[array-prepend](#)

[array_pull](#)

[array_random](#)

[array_set](#)

[array_sort](#)

[array_sort_recursive](#)

[array_where](#)

[array_wrap](#)

[data_fill](#)

[data_get](#)

[data_set](#)

[head](#)

[last](#)

Rutas

[app_path](#)

[base_path](#)

[config_path](#)

[database_path](#)

[mix](#)

[public_path](#)

[resource_path](#)

[storage_path](#)

Cadenas

\	str_after	str_replace_first
camel_case	str_before	str_replace_last
class_basename	str_contains	str_singular
e	str_finish	str_slug
ends_with	str_is	str_start
kebab_case	str_limit	studly_case
preg_replace_array	str_plural	title_case
snake_case	str_random	trans
starts_with	str_replace_array	trans_choice

URLs

action	secure_asset	secure_url
asset	route	url

Varios

abort	decrypt	report
abort_if	dispatch	request
abort_unless	dispatch_now	rescue
app	dump	resolve
auth	encrypt	response
back	env	retry
bcrypt	event	session
blank	factory	tap
broadcast	filled	today
cache	info	throw_if
class_uses_recursive	logger	throw_unless
collect	method_field	trait_uses_recursive
config	now	transform
cookie	old	validator
csrf_field	optional	value
csrf_token	policy	view
dd	redirect	with

Lista de métodos

Arrays & objetos

array_add()

La función `array_add` añade un par clave/valor dado a un *array* si la clave dada no existe ya en el mismo:

```
$array = array_add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

array_collapse()

La función `array_collapse` colapsa un *array* de *arrays* en un solo *array*:

```
$array = array_collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

array_divide()

La función `array_divide` devuelve dos *arrays*, uno que contiene las claves y el otro que contiene los valores del *array* en cuestión:

```
list($keys, $values) = array_divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

array_dot()

La función `array_dot` convierte un *array* multidimensional en un *array* de un único nivel utilizando la notación de "puntos" para indicar la profundidad:

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
$flattened = array_dot($array);  
  
// ['products.desk.price' => 100]
```

array_except()

La función `array_except` elimina los pares clave/valor dados de un *array*:

```
$array = ['name' => 'Desk', 'price' => 100];  
  
$filtered = array_except($array, ['price']);  
  
// ['name' => 'Desk']
```

array_first()

El método `array_first` devuelve el primer elemento de un *array* que cumpla una condición dada:

```
$array = [100, 200, 300];  
  
$first = array_first($array, function ($value, $key) {  
    return $value >= 150;  
});  
  
// 200
```

Se puede pasar un valor por defecto como tercer parámetro al método. Este valor se devolverá si no hay un valor que cumpla la condición:

```
$first = array_first($array, $callback, $default);
```

array_flatten()

La función `array_flatten` aplana un *array* multidimensional en un *array* de un solo nivel:

```
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];  
  
$flattened = array_flatten($array);  
  
// ['Joe', 'PHP', 'Ruby']
```

array_forget()

El método `array_forget` elimina un par clave/valor dado de un *array* anidado utilizando la notación de "puntos":

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
array_forget($array, 'products.desk');  
  
// ['products' => []]
```

array_get()

El método `array_get` recupera un valor de un *array* anidado usando la notación de "puntos":

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
$price = array_get($array, 'products.desk.price');  
  
// 100
```

La función `array_get` también acepta un valor predeterminado, que será devuelto si no se encuentra la clave especificada:

```
$discount = array_get($array, 'products.desk.discount', 0);  
  
// 0
```

array_has()

La función `array_has` comprueba si un elemento o elementos determinados existen en un *array* utilizando la notación de "puntos":

```
$array = ['product' => ['name' => 'Desk', 'price' => 100]];  
  
$contains = array_has($array, 'product.name');  
  
// true  
  
$contains = array_has($array, ['product.price', 'product.discount']);  
  
// false
```

`array_last()`

La función `array_last` devuelve el último elemento de un *array* que pasa una prueba de verdad concreta:

```
$array = [100, 200, 300, 110];  
  
$last = array_last($array, function ($value, $key) {  
    return $value >= 150;  
});  
  
// 300
```

Se puede pasar un valor predeterminado como tercer argumento al método. Este valor se devolverá si ningún valor pasa la prueba de la verdad:

```
$last = array_last($array, $callback, $default);
```

`array_only()`

La función `array_only` devuelve sólo los pares clave/valor especificados del *array* dado:

```
$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];  
  
$slice = array_only($array, ['name', 'price']);  
  
// ['name' => 'Desk', 'price' => 100]
```

array_pluck()

La función `array_pluck` recupera todos los valores de una clave determinada de un *array*:

```
$array = [  
    ['developer' => ['id' => 1, 'name' => 'Taylor']],  
    ['developer' => ['id' => 2, 'name' => 'Abigail']],  
];  
  
$names = array_pluck($array, 'developer.name');  
  
// ['Taylor', 'Abigail']
```

También se puede especificar que clave obtendrá el resultado:

```
$names = array_pluck($array, 'developer.name', 'developer.id');  
  
// [1 => 'Taylor', 2 => 'Abigail']
```

array-prepend()

La función `array-prepend` añadirá un elemento al principio del *array*:

```
$array = ['one', 'two', 'three', 'four'];  
  
$array = array_prepend($array, 'zero');  
  
// ['zero', 'one', 'two', 'three', 'four']
```

Si es necesario, puede especificar la clave que debe utilizarse para el valor:

```
$array = ['price' => 100];  
  
$array = array_prepend($array, 'Desk', 'name');  
  
// ['name' => 'Desk', 'price' => 100]
```

array_pull()

La función `array_pull` devuelve y elimina un par clave/valor del *array*:

```
$array = ['name' => 'Desk', 'price' => 100];  
  
$name = array_pull($array, 'name');  
  
// $name: Desk  
  
// $array: ['price' => 100]
```

Se puede pasar un valor predeterminado como tercer argumento al método. Este valor se devolverá si la clave no existe:

```
$value = array_pull($array, $key, $default);
```

array_random()

La función `array_random` devuelve un valor aleatorio del *array*:

```
$array = [1, 2, 3, 4, 5];  
  
$random = array_random($array);  
  
// 4 - (retrieved randomly)
```

También puede especificar el número de elementos a devolver como segundo argumento opcional. Tenga en cuenta que al proporcionar este argumento se devolverá un *array*, incluso si sólo se desea un elemento:

```
$items = array_random($array, 2);  
  
// [2, 5] - (retrieved randomly)
```

array_set()

El método `array_set` establece un valor dentro de un *array* anidado en profundidad mediante la notación de "puntos":

```
$array = ['products' => ['desk' => ['price' => 100]]];  
  
array_set($array, 'products.desk.price', 200);  
  
// ['products' => ['desk' => ['price' => 200]]]
```

array_sort()

La función `array_sort` ordena un *array* por sus valores:

```
$array = ['Desk', 'Table', 'Chair'];  
  
$sorted = array_sort($array);  
  
// ['Chair', 'Desk', 'Table']
```

También puede ordenar el *array* por los resultados del *Closure* dado:

```

$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = array_values(array_sort($array, function ($value) {
    return $value['name'];
}));
```

/*

```

[
    ['name' => 'Chair'],
    ['name' => 'Desk'],
    ['name' => 'Table'],
]
*/

```

array_sort_recursive()

La función `array_sort_recursive` un *array* de forma recursiva usando la función `sort` :

```

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
];

$sorted = array_sort_recursive($array);

/*
[
    ['Li', 'Roman', 'Taylor'],
    ['JavaScript', 'PHP', 'Ruby'],
]
*/

```

array_where()

La función `array_where` filtra un *array* usando el *Closure* dado:

```
$array = [100, '200', 300, '400', 500];

$filtered = array_where($array, function ($value, $key) {
    return is_string($value);
});

// [1 => 200, 3 => 400]
```

array_wrap()

La función `array_wrap` envuelve el valor dado en un *array*. Si el valor dado ya es un *array* no se modificará:

```
$string = 'Laravel';

$array = array_wrap($string);

// ['Laravel']
```

data_fill()

La función `data_fill` establece un valor faltante dentro del *array* anidado u objeto usando la notación de "punto":

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_fill($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 100]]]

data_fill($data, 'products.desk.discount', 10);

// ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]
```

Esta función también acepta asteriscos como comodines y rellenará el objetivo en consecuencia:

```

$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2'],
    ],
];

data_fill($data, 'products.*.price', 200);

/*
[
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2', 'price' => 200],
    ],
]
*/

```

`data_get()`

La función `data_get` recupera un valor de un *array* anidado u objeto utilizando la notación de "puntos":

```

$data = ['products' => ['desk' => ['price' => 100]]];

$price = data_get($data, 'products.desk.price');

// 100

```

La función `data_get` también acepta un valor predeterminado, que se devolverá si no se encuentra la clave especificada:

```

$discount = data_get($data, 'products.desk.discount', 0);

// 0

```

`data_set()`

La función `data_set` establece un valor dentro de un *array* anidado u objeto usando la notación de "punto":

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_set($data, 'products.desk.price', 200);  
  
// ['products' => ['desk' => ['price' => 200]]]
```

Esta función también acepta comodines y fijará los valores en el objetivo en consecuencia:

```
$data = [  
    'products' => [  
        ['name' => 'Desk 1', 'price' => 100],  
        ['name' => 'Desk 2', 'price' => 150],  
    ],  
];  
  
data_set($data, 'products.*.price', 200);  
  
/*  
[  
    'products' => [  
        ['name' => 'Desk 1', 'price' => 200],  
        ['name' => 'Desk 2', 'price' => 200],  
    ],  
]
```

Por defecto, todos los valores existentes se sobrescriben. Si sólo desea establecer un valor si no existe, puede pasar `false` como tercer argumento:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_set($data, 'products.desk.price', 200, false);  
  
// ['products' => ['desk' => ['price' => 100]]]
```

head()

La función `head` devuelve el primer elemento del *array*:

```
$array = [100, 200, 300];  
  
$first = head($array);  
  
// 100
```

last()

La función `last` devuelve el último elemento del *array* especificado:

```
$array = [100, 200, 300];  
  
$last = last($array);  
  
// 300
```

Rutas

app_path()

La función `app_path` devuelve la ruta cualificada al directorio `app`. También puede utilizar la función `app_path` para generar una ruta totalmente cualificada a un archivo relativo al directorio `app`:

```
$path = app_path();  
  
$path = app_path('Http/Controllers/Controller.php');
```

base_path()

La función `base_path` devuelve la ruta cualificada a la raíz del proyecto. También puede utilizar la función `base_path` para generar una ruta totalmente cualificada a un archivo determinado en relación con el directorio raíz del proyecto:

```
$path = base_path();  
  
$path = base_path('vendor/bin');
```

config_path()

La función `config_path` devuelve la ruta totalmente cualificada al directorio `config`. También puede utilizar la función `config_path` para generar una ruta totalmente cualificada a un archivo determinado dentro del directorio `config`:

```
$path = config_path();  
  
$path = config_path('app.php');
```

database_path()

La función `database_path` devuelve la ruta totalmente cualificada al directorio `database`. También puede utilizar la función `database_path` para generar una ruta totalmente cualificada a un archivo determinado dentro del directorio `database`:

```
$path = database_path();  
  
$path = database_path('factories/UserFactory.php');
```

mix()

La función `mix` devuelve la ruta a un [archivo de Mix versionado](#):

```
$path = mix('css/app.css');
```

public_path()

La función `public_path` devuelve la ruta totalmente cualificada al directorio `public`. También puede utilizar la función `public_path` para generar una ruta totalmente cualificada a un archivo determinado dentro del directorio `public`:

```
$path = public_path();  
  
$path = public_path('css/app.css');
```

resource_path()

La función `resource_path` devuelve la ruta totalmente cualificada al directorio `resources`. También puede utilizar la función `resource_path` para generar una ruta totalmente cualificada a un archivo determinado dentro del directorio `resources`:

```
$path = resource_path();  
  
$path = resource_path('assets/sass/app.scss');
```

storage_path()

La función `storage_path` devuelve la ruta totalmente cualificada al directorio `storage`. También puede utilizar la función `storage_path` para generar una ruta totalmente cualificada a un archivo determinado dentro del directorio `storage`:

```
$path = storage_path();  
  
$path = storage_path('app/file.txt');
```

Cadenas

__()

La función `__` traduce la cadena de traducción o la clave de traducción utilizando sus [ficheros de localización](#):

```
echo __('Welcome to our application');  
  
echo __('messages.welcome');
```

Si la cadena o clave de traducción especificada no existe, la función `__` simplemente devolverá el valor proporcionado. Por lo tanto, utilizando el ejemplo anterior, la función `__` devolvería `messages.welcome` si esa clave de conversión no existe.

camel_case()

La función `camel_case` convierte la cadena dada en `camelCase`:

```
$converted = camel_case('foo_bar');

// fooBar
```

class_basename()

`class_basename` devuelve el nombre de la clase especificada sin el *namespace* de la propia clase:

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

e()

La función `e` ejecuta la función `htmlspecialchars` de PHP con la opción `double_encode` ajustada a `false`:

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

ends_with()

La función `ends_with` determina si la cadena dada termina con un valor especificado:

```
$result = ends_with('This is my name', 'name');

// true
```

kebab_case()

La función `kebab_case` convierte la cadena dada a `kebab-case` :

```
$converted = kebab_case('fooBar');

// foo-bar
```

preg_replace_array()

La función `preg_replace_array` reemplaza un patrón dado en la cadena de forma secuencial utilizando un `array`:

```
$string = 'The event will take place between :start and :end';

$replaced = preg_replace_array('/:[a-z_]+/', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

snake_case()

La función `snake_case` convierte la cadena dada en formato `snake_case` :

```
$converted = snake_case('fooBar');

// foo_bar
```

starts_with()

La función `starts_with` determina si la cadena dada comienza con el valor especificado:

```
$result = starts_with('This is my name', 'This');

// true
```

str_after()

La función `str_after` devuelve todo después del valor dado en una cadena:

```
$slice = str_after('This is my name', 'This is');

// ' my name'
```

str_before()

La función `str_before` devuelve todo antes del valor dado en una cadena:

```
$slice = str_before('This is my name', 'my name');

// 'This is '
```

str_contains()

La función `str_contains` determina si la cadena dada contiene el valor especificado:

```
$contains = str_contains('This is my name', 'my');

// true
```

También puede pasar un *array* de valores para determinar si la cadena contiene alguno de ellos:

```
$contains = str_contains('This is my name', ['my', 'foo']);

// true
```

str_finish()

La función `str_finish` añade una sola instancia del valor dado a una cadena si no termina ya con el valor:

```
$adjusted = str_finish('this/string', '/');

// this/string

$adjusted = str_finish('this/string/', '/');

// this/string/
```

str_is()

La función `str_is` determina si una cadena dada coincide con un patrón determinado. Se pueden utilizar asteriscos para indicar comodines:

```
$matches = str_is('foo*', 'foobar');

// true

$matches = str_is('baz*', 'foobar');

// false
```

str_limit()

La función `str_limit` interrumpe la cadena en la longitud especificada:

```
$truncated = str_limit('The quick brown fox jumps over the lazy dog', 20);

// The quick brown fox...
```

También puede pasar un tercer argumento para cambiar la cadena que se agregará al final:

```
$truncated = str_limit('The quick brown fox jumps over the lazy dog', 20, ' (...)');

// The quick brown fox (...)
```

str_plural()

La función `str_plural` convierte una cadena a su forma plural. Esta función actualmente sólo soporta el idioma inglés:

```
$plural = str_plural('car');

// cars

$plural = str_plural('child');

// children
```

Se puede proporcionar a la función un número entero como segundo argumento para recuperar la forma singular o plural de la cadena:

```
$plural = str_plural('child', 2);

// children

$plural = str_plural('child', 1);

// child
```

str_random()

La función `str_random` genera una cadena aleatoria de la longitud especificada. Utiliza la función de PHP `random_bytes`:

```
$random = str_random(40);
```

str_replace_array()

La función `str_replace_array` reemplaza un valor dado en la cadena secuencialmente usando un `array`:

```
$string = 'The event will take place between ? and ?';  
  
$replaced = str_replace_array('?', ['8:30', '9:00'], $string);  
  
// The event will take place between 8:30 and 9:00
```

`str_replace_first()`

La función `str_replace_first` reemplaza la primera aparición de un valor dado en una cadena:

```
$replaced = str_replace_first('the', 'a', 'the quick brown fox jumps over the lazy dog');  
  
// a quick brown fox jumps over the lazy dog
```

`str_replace_last()`

La función `str_replace_last` reemplaza la última aparición de un valor dado en una cadena:

```
$replaced = str_replace_last('the', 'a', 'the quick brown fox jumps over the lazy dog');  
  
// the quick brown fox jumps over a lazy dog
```

`str_singular()`

La función `str_singular` convierte una cadena a su forma singular. Esta función actualmente sólo soporta el idioma inglés:

```
$singular = str_singular('cars');

// car

$singular = str_singular('children');

// child
```

str_slug()

La función `str_slug` genera un "slug" amigable para la URL de la cadena especificada:

```
$slug = str_slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

str_start()

La función `str_start` añade una sola instancia del valor dado a una cadena si no comienza ya con el valor:

```
$adjusted = str_start('this/string', '/');

// /this/string

$adjusted = str_start('/this/string/', '/');

// /this/string
```

studly_case()

La función `studly_case` convierte la cadena dada en formato `StudlyCase`:

```
$converted = studly_case('foo_bar');

// FooBar
```

`title_case()`

La función `title_case` convierte la cadena dada a `Title Case`:

```
$converted = title_case('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

`trans()`

La función `trans` traduce la clave de conversión dada utilizando los [archivos de localización](#):

```
echo trans('messages.welcome');
```

Si la clave de conversión especificada no existe, la función `trans` simplemente devolverá la clave dada. Por lo tanto, utilizando el ejemplo anterior, la función `trans` devolvería `messages.welcome` si la clave de traducción no existe.

`trans_choice()`

La función `trans_choice` traduce la clave de conversión dada con inflexión:

```
echo trans_choice('messages.notifications', $unreadCount);
```

Si la clave de conversión especificada no existe, la función `trans_choice` simplemente devolverá la clave dada. Por lo tanto, utilizando el ejemplo anterior, la función `trans_choice` devolvería `messages.notifications` si no existe la clave de conversión.

URLs

`action()`

La función `action` genera una dirección URL para una acción determinada de un controlador. No es necesario especificar el *namespace* completo del controlador. En vez de eso, hay que pasar el nombre de la clase relativo al *namespace* `App\Http\Controllers`:

```
$url = action('HomeController@index');
```

Si el método acepta parámetros de ruta, se pueden pasar como segundo argumento al método:

```
$url = action('UserController@profile', ['id' => 1]);
```

asset()

La función `asset` genera una URL para un *asset* utilizando el esquema actual de la petición (HTTP o HTTPS):

```
$url = asset('img/photo.jpg');
```

secure_asset()

La función `secure_asset` genera una URL para un *asset* usando HTTPS:

```
$url = secure_asset('img/photo.jpg');
```

route()

La función `route` genera una URL para un nombre de ruta dado:

```
$url = route('routeName');
```

Si la ruta acepta parámetros, se pueden pasar como segundo argumento al método:

```
$url = route('routeName', ['id' => 1]);
```

De forma predeterminada, la función `route` genera una URL absoluta. Si desea generar una URL relativa, puede pasar `false` como tercer argumento:

```
$url = route('routeName', ['id' => 1], false);
```

`secure_url()`

La función `secure_url` genera una URL HTTPS totalmente cualificada a la ruta dada:

```
$url = secure_url('user/profile');

$url = secure_url('user/profile', [1]);
```

`url()`

La función `url` genera una URL completa a la ruta dada:

```
$url = url('user/profile');

$url = url('user/profile', [1]);
```

Si no se proporciona ninguna ruta, se devuelve una instancia de

`Illuminate\Routing\UrlGenerator`:

```
$current = url()->current();

$full = url()->full();

$previous = url()->previous();
```

Varios

`abort()`

La función `abort` lanza una excepción HTTP que será mostrada por el gestor de excepciones:

```
abort(403);
```

También puede proporcionar el texto de respuesta de la excepción y encabezados de respuesta personalizados:

```
abort(403, 'Unauthorized.', $headers);
```

abort_if()

La función `abort_if` lanza una excepción HTTP si una expresión *booleana* especificada se evalúa a `true` :

```
abort_if(! Auth::user()->isAdmin(), 403);
```

Al igual que el método `abort` , también puede proporcionar el texto de respuesta de la excepción como tercer argumento y un *array* de cabeceras de respuesta personalizadas como cuarto argumento.

abort_unless()

La función `abort_unless` lanza una excepción HTTP si una expresión *booleana* dada evalúa `false` :

```
abort_unless(Auth::user()->isAdmin(), 403);
```

Al igual que el método `abort` , también puede proporcionar el texto de respuesta de la excepción como tercer argumento y una matriz de encabezados de respuesta personalizados como cuarto argumento.

app()

La función `app` devuelve una instancia de [service container](#):

```
$container = app();
```

Puede pasar un nombre de clase o interfaz para resolverlo desde el contenedor:

```
$api = app('HelpSpot\API');
```

auth()

La función `auth` devuelve una instancia de [authenticator](#). Puede utilizarlo en lugar de la *facade* `Auth` por conveniencia:

```
$user = auth()->user();
```

Si es necesario, puede especificar a qué instancia de *guard* desea acceder:

```
$user = auth('admin')->user();
```

back()

La función `back` genera una [respuesta HTTP redireccionada](#) a la ubicación anterior del usuario:

```
return back($status = 302, $headers = [], $fallback = false);  
  
return back();
```

bcrypt()

La función `bcrypt` [hashea](#) el valor dado utilizando Bcrypt. Se puede utilizar como alternativa a la *facade* `Hash`:

```
$password = bcrypt('my-secret-password');
```

broadcast()

La función `broadcast` [difunde \("broadcasts"\) el evento](#) dado a sus *listeners*:

```
broadcast(new UserRegistered($user));
```

blank()

La función `blank` devuelve si el valor dado está "en blanco":

```
blank('');  
blank('  ');  
blank(null);  
blank(collect());  
  
// true  
  
blank(0);  
blank(true);  
blank(false);  
  
// false
```

Para el inverso de `blank`, vea el método [filled](#).

cache()

La función `cache` puede usarse para obtener valores de la [caché](#). Si la clave dada no existe en la caché, se devolverá un valor predeterminado opcional:

```
$value = cache('key');  
  
$value = cache('key', 'default');
```

Puede añadir elementos a la caché pasando una matriz de pares clave/valor a la función. También debe pasar el número de minutos o la duración que el valor almacenado en caché debe considerarse válido:

```
cache(['key' => 'value'], 5);  
  
cache(['key' => 'value'], Carbon::now()->addSeconds(10));
```

class_uses_recursive()

La función `class_uses_recursive` devuelve todos los *traits* usados por una clase, incluyendo los *traits* usados por cualquier subclase:

```
$traits = class_uses_recursive(App\User::class);
```

collect()

La función `collect` crea una instancia de [collection](#) a partir del valor dado:

```
$collection = collect(['taylor', 'abigail']);
```

config()

La función `config` obtiene el valor de una variable de [configuración](#). Los valores de configuración pueden accederse mediante una sintaxis de "puntos", que incluye el nombre del archivo y la opción a acceder. Se puede especificar un valor por defecto para ser devuelto si no existe la opción de configuración:

```
$value = config('app.timezone');  
  
$value = config('app.timezone', $default);
```

Puede establecer variables de configuración en tiempo de ejecución pasando un *array* de pares clave/valor:

```
config(['app.debug' => true]);
```

cookie()

La función `cookie` crea una nueva instancia de [cookie](#):

```
$cookie = cookie('name', 'value', $minutes);
```

csrf_field()

La función `csrf_field` genera un campo HTML input `hidden` que contiene el valor del *token* CSRF. Por ejemplo, utilizando [sintaxis de Blade](#):

```
 {{ csrf_field() }}
```

csrf_token()

La función `csrf_token` recupera el valor del *token CSRF* actual:

```
$token = csrf_token();
```

dd()

La función `dd` muestra información sobre las variables dadas y termina la ejecución del script:

```
dd($value);  
  
dd($value1, $value2, $value3, ...);
```

Si no desea detener la ejecución de su script, utilice la función `dump`.

decrypt()

La función `decrypt` desencripta el valor dado utilizando el [encrypter](#) de Laravel:

```
$decrypted = decrypt($encrypted_value);
```

dispatch()

La función `dispatch` añade el [trabajo](#) dado a la [cola de trabajo](#) de Laravel:

```
dispatch(new App\Jobs\SendEmails);
```

dispatch_now()

La función `dispatch_now` ejecuta el [trabajo](#) inmediatamente y devuelve el valor de su método `handle`:

```
$result = dispatch_now(new App\Jobs\SendEmails);
```

dump()

La función `dump` muestra información sobre las variables dadas:

```
dump($value);  
  
dump($value1, $value2, $value3, ...);
```

Si desea detener la ejecución del script después de haber mostrado información sobre las variables, utilice la función `dd` en su lugar.

encrypt()

La función `encrypt` encripta el valor dado usando el [encrypter](#) de Laravel:

```
$encrypted = encrypt($unencrypted_value);
```

env()

La función `env` recupera el valor de una [variable del entorno](#) o devuelve un valor por defecto:

```
$env = env('APP_ENV');  
  
// Returns 'production' if APP_ENV is not set...  
$env = env('APP_ENV', 'production');
```

event()

La función `event` lanza un [evento](#) a sus *listeners*:

```
event(new UserRegistered($user));
```

factory()

La función `factory` crea un *model factory builder* para una clase, nombre y cantidad. Puede ser utilizado durante testing (pruebas) o seeding (poblado):

```
$user = factory(App\User::class)->make();
```

filled()

La función `filled` devuelve si el valor dado no está "en blanco":

```
filled(0);
filled(true);
filled(false);

// true

filled('');
filled(' ');
filled(null);
filled(collect());

// false
```

Para el inverso de `filled`, ver el método `blank`.

info()

La función `info` escribirá información en el `log`:

```
info('Some helpful information');
```

También se puede transferir a la función un array de datos contextuales:

```
info('User login attempt failed.', ['id' => $user->id]);
```

logger()

La función `logger` puede utilizarse para escribir un mensaje de nivel `debug` en el [log](#):

```
logger('Debug message');
```

También se puede transferir a la función un *array* de datos contextuales:

```
logger('User has logged in.', ['id' => $user->id]);
```

Se devolverá una instancia de [logger](#) si no se pasa ningún valor a la función:

```
logger()->error('You are not allowed here.');
```

method_field()

La función `method_field` genera un campo HTML `input hidden` que almacena el valor del verbo HTTP del formulario. Por ejemplo, utilizando la [sintaxis Blade](#):

```
<form method="POST">
{{ method_field('DELETE') }}
</form>
```

now()

La función `now` crea una nueva instancia de `Illuminate\Support\Carbon` para la hora actual:

```
$now = now();
```

old()

La función `old` [recupera](#) un valor de [entrada antiguo](#) de la sesión:

```
$value = old('value');

$value = old('value', 'default');
```

optional()

La función `optional` acepta cualquier argumento y le permite acceder a las propiedades o métodos de llamada de ese objeto. Si el objeto dado es `null`, las propiedades y métodos simplemente devolverán `null` en lugar de causar un error:

```
return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}
```

policy()

El método `policy` recupera una instancia de `policy` para una clase determinada:

```
$policy = policy(App\User::class);
```

redirect()

La función `redirect` devuelve una [respuesta HTTP de redirección](#), o devuelve la instancia del redireccionador si se llama sin argumentos:

```
return redirect($to = null, $status = 302, $headers = [], $secure = null);

return redirect('/home');

return redirect()->route('route.name');
```

report()

La función `report` reportará una excepción usando el método `report` de su [manejador de excepciones](#):

```
report($e);
```

request()

La función `request` devuelve la instancia actual de `request` u obtiene un elemento de entrada:

```
$request = request();  
  
$value = request('key', $default = null);
```

rescue()

La función `rescue` ejecuta el *Closure* dado y captura cualquier excepción que ocurra durante su ejecución. Todas las excepciones que sean capturadas se enviarán al método `report` del [gestor de excepciones](#); sin embargo, la solicitud continuará procesándose:

```
return rescue(function () {  
    return $this->method();  
});
```

También puede pasar un segundo argumento a la función `rescue`. Este argumento será el valor "por defecto" que debe devolverse si se produce una excepción durante la ejecución del *Closure*:

```
return rescue(function () {  
    return $this->method();  
}, false);  
  
return rescue(function () {  
    return $this->method();  
}, function () {  
    return $this->failure();  
});
```

resolve()

La función `resolve` resuelve un determinado nombre de clase o interfaz a su instancia utilizando el [contenedor de servicio](#):

```
$api = resolve('HelpSpot\API');
```

response()

La función `response` crea una instancia de [response](#) u obtiene una instancia de una factoría `response`:

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

retry()

La función `retry` intenta ejecutar la llamada de retorno hasta que se alcanza el umbral de intento máximo. Si la devolución de llamada no lanza una excepción, se devolverá su valor de retorno. Si la devolución de llamada lanza una excepción, se volverá a repetir automáticamente. Si se excede el número máximo de intentos, la excepción será lanzada:

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms in between attempts...
}, 100);
```

session()

La función `session` se puede utilizar para obtener o establecer valores de [sesión](#):

```
$value = session('key');
```

Se pueden establecer valores pasando un `array` con el par clave/valor a la función:

```
session(['chairs' => 7, 'instruments' => 3]);
```

La sesión almacenada se retornará si no se pasa ningún valor a la función:

```
$value = session()->get('key');
```

```
session()->put('key', $value);
```

tap()

La función `tap` acepta dos argumentos: un valor `$value` y un *Closure*. El `$value` pasará al *Closure* y luego será devuelto por la función `tap`. El valor de retorno del *Closure* es irrelevante:

```
$user = tap(User::first(), function ($user) {
    $user->name = 'taylor';

    $user->save();
});
```

Si no se pasa ningún *Closure* a la función `tap`, puede llamar a cualquier método en el `$value` dado. El valor de retorno del método que llama siempre será `$value`, independientemente de lo que el método devuelva realmente en su definición. Por ejemplo, el método Eloquent `update` normalmente devuelve un entero. Sin embargo, se puede forzar al método a devolver el modelo mismo encadenando la llamada `update` a través de la función `tap`:

```
$user = tap($user)->update([
    'name' => $name,
    'email' => $email,
]);
```

today()

La función `today` crea una nueva instancia de `Illuminate\Support\Carbon` para la fecha actual:

```
$today = today();
```

throw_if()

La función `throw_if` lanza la excepción dada si una expresión booleana determinada evalúa `true`:

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw_if(
    ! Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page'
);
```

throw_unless()

La función `throw_unless` lanza la excepción dada si una expresión booleana dada evalúa `false`:

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw_unless(
    Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page'
);
```

trait_uses_recursive()

La función `trait_uses_recursive` devuelve todos los *traits* usados por un *trait*:

```
$traits = trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

transform()

La función `transform` ejecuta un `Closure` en un valor dado si no está en blanco y devuelve el resultado del `Closure`:

```
$callback = function ($value) {
    return $value * 2;
};

$result = transform(5, $callback);

// 10
```

También se puede pasar al método un valor predeterminado o [Closure](#) como tercer parámetro. Este valor se devolverá si el valor dado está en blanco:

```
$result = transform(null, $callback, 'The value is blank');

// The value is blank
```

validator()

La función [validator](#) crea una nueva instancia de [validator](#) con los argumentos dados. Puede utilizarlo en lugar de la facade [Validador](#) por conveniencia:

```
$validator = validator($data, $rules, $messages);
```

value()

La función [value](#) devuelve el valor indicado. Sin embargo, si se proporciona un [Closure](#) a la función, se devolverá el resultado del [Closure](#):

```
$result = value(true);

// true

$result = value(function () {
    return false;
});

// false
```

view()

La función `view` retorna una instancia de `view`:

```
return view('auth.login');
```

with()

La función `with` devuelve el valor indicado. Si se pasa un `Closure` como segundo argumento a la función, se ejecutará `Closure` y se devolverá el resultado:

```
$callback = function ($value) {
    return (is_numeric($value)) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

Correo

Introducción

Pre-requisitos del driver

Generación de Mailables

Escribiendo Mailables

Configurando el Remitente

Configurando la Vista

Ver datos

Archivos Adjuntos

Archivos Adjuntos en Línea

Customizing The SwiftMailer Message

Archivos de Markdown

Generando Markdown Mailables

Escribiendo Mensajes Markdown

Personalización de los componentes

Previewing Mailables In The Browser

Enviar Correo

Cola de Correos

Correo & Desarrollo Local

Eventos

Introducción

Laravel proporciona una API clara y simple sobre la popular librería [SwiftMailer](#) con drivers para SMTP, Mailgun, SparkPost, Amazon SES, para la función `mail` de PHP y `sendmail`, permitiéndole comenzar a enviar correos rápidamente a través de un servicio local o en la nube de su elección.

Pre-requisitos del driver

Los drivers basados en API's como Mailgun y Mandril son a menudo más simples y rápidos que los servidores SMTP. Siempre que sea posible, debe usar uno de estos drivers. Todos los drivers del API requieren la librería Guzzle HTTP, la cual puede instalarse a través del gestor

de paquetes Composer:

```
composer require guzzlehttp/guzzle
```

Driver Mailgun

Para utilizar el driver de Mailgun, se debe instalar primero Guzzle, y luego colocar `mailgun` en la opción de `driver` en el archivo de configuración ubicado en `config/mail.php`. A continuación, se verifica que en el archivo `config/services.php` se encuentren las siguientes opciones:

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

Driver SparkPost

Para usar el driver SparkPost, instale primero Guzzle, luego configure la opción `driver` en el archivo de configuración `config/mail.php` con el valor `sparkpost`. A continuación, se verifica que en el archivo `config/services.php` se encuentren las siguientes opciones:

```
'sparkpost' => [
    'secret' => 'your-sparkpost-key',
],
```

Driver SES

Para usar el driver Amazon SES, primero debe instalar el SDK Amazon AWS para PHP. Puede instalar esta librería añadiendo la siguiente línea a la sección `require` del archivo `composer.json` y ejecutando el comando `composer update`:

```
"aws/aws-sdk-php": "~3.0"
```

A continuación, configure la opción `driver` en el archivo de configuración `config/mail.php` con el valor `ses` y verifique que el archivo de configuración `config/services.php` contiene las siguientes opciones:

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region', // e.g. us-east-1
],
```

Generación de Mailables

En Laravel, cada tipo de correo electrónico enviado por su aplicación se representa como una clase "mailable". Todas estas clases se almacenan en el directorio `app/Mail`. No se preocupe si no ve este directorio en su aplicación, se creará para usted cuando cree su primera clase mailable usando el comando `make:mail`:

```
php artisan make:mail OrderShipped
```

Escribiendo Mailables

Toda la configuración de una clase mailable se realiza en el método `build`. Desde este método puede llamar a varios métodos como `from`, `subject`, `view` y `attach` para configurar la presentación y entrega del correo electrónico.

Configurando el Remitente

El método `from`

Primero, exploremos la configuración del remitente del correo electrónico. O, en otras palabras, quién se configurará en el método "from". Hay dos formas de configurar el remitente. La primera, puede usar el método `from` dentro del método `build` de su clase mailable:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
  
public function build()  
{  
    return $this->from('example@example.com')  
        ->view('emails.orders.shipped');  
}
```

Usar una Dirección de Remitente Global con `from`

Sin embargo, si la aplicación utiliza la misma dirección "from" para todos los correos electrónicos, puede resultar engoroso llamar al método `from` en cada clase de mailable que se genere. En su lugar, se puede especificar una dirección "from" en el archivo de configuración `config/mail.php`. Esta dirección se usará si no se especifica ninguna otra dirección "from" dentro de la clase mailable:

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

Configurando la Vista

Dentro del método `build` de clase mailable, se puede usar el método `view` para especificar qué plantilla se debe usar para representar el contenido del correo electrónico. Dado que cada correo electrónico normalmente usa una [plantilla de Blade](#) para representar el contenido, se cuenta con toda la potencia y la comodidad del motor de plantillas para construir el HTML del correo electrónico:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
  
public function build()  
{  
    return $this->view('emails.orders.shipped');  
}
```

Se puede crear un directorio `resources/views/emails` para guardar todas las vistas de correo electrónico; sin embargo, se pueden colocar donde se desee dentro del directorio `resources/views`.

Emails de texto plano

Si se desea definir una versión de texto sin formato de correo electrónico, se puede usar el método `text`. Al igual que el método `view`, el método `text` acepta un nombre de plantilla que se utilizará para representar el contenido del correo electrónico. Se es libre de definir una versión HTML y de texto sin formato del mensaje:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
  
public function build()  
{  
    return $this->view('emails.orders.shipped')  
        ->text('emails.orders.shipped_plain');  
}
```

Ver datos

Vía Propiedades Públicas

Por lo general, se querrá pasar algunos datos a una vista que se puede utilizar al representar el HTML de un correo electrónico. Hay dos maneras en que puede hacer que los datos estén disponibles para una vista. Primero, cualquier propiedad pública definida en la clase mailable se pondrá automáticamente a disposición de la vista. Entonces, por ejemplo, se puede pasar datos al constructor de la clase mailable y establecer esos datos a propiedades públicas definidas dentro de la clase:

```
<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var Order
     */
    public $order;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped');
    }
}
```

Una vez que los datos se han establecido en una propiedad pública, estarán automáticamente disponibles en la vista, por lo que puede acceder a ella como si tuviera acceso a cualquier otro dato en las plantillas Blade:

```
<div>
    Price: {{ $order->price }}
</div>
```

Vía el método `with` :

Si se desea personalizar el formato de los datos del correo electrónico antes de enviarlo a la plantilla, se puede pasar manualmente los datos a la vista a través del método `with` . Por lo general, se pasarán datos a través del constructor de la clase mailable; sin embargo, para que los datos no estén disponibles automáticamente en la plantilla se debe establecer estos datos como propiedades `protected` ó `private` . Luego, cuando se llame al método `with` , se debe pasar un array de los datos que se desee poner a disposición de la plantilla:

```
<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var Order
     */
    protected $order;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped')
            ->with([
                'orderName' => $this->order->name,
                'orderPrice' => $this->order->price,
            ]);
    }
}
```

```
    }  
}
```

Una vez que los datos se pasen con el método `with`, estarán automáticamente disponibles en la vista, por lo que puede acceder a ella como si tuviera acceso a cualquier otro dato en las plantillas Blade:

```
<div>  
    Price: {{ $orderPrice }}  
</div>
```

Archivos Adjuntos

Para agregar archivos adjuntos a un correo electrónico, use el método `attach` dentro del método `build` de la clase mailable. El método `attach` acepta la ruta entera del archivo como primer argumento:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
  
public function build()  
{  
    return $this->view('emails.orders.shipped')  
        ->attach('/path/to/file');  
}
```

Al adjuntar archivos a un mensaje, también se puede especificar el nombre de visualización y/o tipo MIME pasando un `array` como el segundo argumento del método `attach`:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file', [
            'as' => 'name.pdf',
            'mime' => 'application/pdf',
        ]);
}
```

Raw Data Attachments

El método `attachData` se puede usar para adjuntar una cadena de bytes sin formato como un archivo adjunto. Por ejemplo, se puede usar este método si ha generado un PDF en la memoria y desea adjuntarlo al correo electrónico sin escribirlo en el disco. El método `attachData` acepta los bytes de datos brutos como su primer argumento, el nombre del archivo como su segundo argumento y un array de opciones como su tercer argumento:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attachData($this->pdf, 'name.pdf', [
            'mime' => 'application/pdf',
        ]);
}
```

Archivos Adjuntos en Línea

La incorporación de imágenes en línea en correos electrónicos suele ser un proceso engorroso; sin embargo, Laravel proporciona una manera práctica para adjuntar imágenes a los correos y recuperar el CID apropiado. Para incluir una imagen en linea, se utiliza el método

`embed` de la variable `$message` en la vista del e-mail. Laravel automáticamente hace que la variable `$message` esté disponible para todas las plantillas de correo electrónico, por lo que no hay que preocuparse por pasarlal manualmente:

```
<body>  
    Here is an image:  
  
    embedData($data, $name) }}>  
</body>
```

Personalizar el mensaje de SwiftMailer

El método `withSwiftMessage` de la clase base `Mailable` permite registrar un *callback* que se invocará con la instancia de mensaje de SwiftMailer sin procesar antes de enviar el mensaje. Esto le da la oportunidad de personalizar el mensaje antes de que se entregue:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    $this->view('emails.orders.shipped');

    $this->withSwiftMessage(function ($message) {
        $message->getHeaders()
            ->addTextHeader('Custom-Header', 'HeaderValue');
    });
}
```

Markdown Mailables

Los mensajes mailers de Markdown permiten aprovechar las plantillas y los componentes precompilados de las notificaciones por correo en los mails. Dado que los mensajes están escritos en Markdown, Laravel es capaz de generar hermosas plantillas HTML para los mensajes, mientras que también genera automáticamente una contraparte de texto plano.

Generando Markdown Mailables

Para generar una mailable con una plantilla de Markdown correspondiente, puede utilizar la opción `--markdown` del comando Artisan `make:mail` :

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

Luego, al configurar el mailable dentro de su método `build`, se llama al método `markdown` en lugar del método `view`. Los métodos `markdown` aceptan el nombre de la plantilla de Markdown y una array opcional de datos para poner a disposición de la plantilla:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
public function build()  
{  
    return $this->from('example@example.com')  
        ->markdown('emails.orders.shipped');  
}
```

Escribiendo Mensajes Markdown

Los mailables de Markdown utilizan una combinación de componentes Blade y sintaxis de Markdown que le permiten construir fácilmente mensaje de correo aprovechando los componentes predefinidos de Laravel:

```
@component('mail::message')  
# Order Shipped  
  
Your order has been shipped!  
  
@component('mail::button', ['url' => $url])  
View Order  
@endcomponent  
  
Thanks,  
{{ config('app.name') }}  
@endcomponent
```

No usar sangría excesiva al escribir correos electrónicos de Markdown. Los analizadores de Markdown renderizarán contenido sangrado como bloques de código.

Componente `button`

El componente `button` muestra un enlace de botón centrado. El componente acepta dos argumentos, `url` y opcional `color`. Los colores admitidos son azul, verde y rojo (`blue` , `green` , and `red`). Puede añadir tantos componentes `button` a un mensaje como desee:

```
@component('mail::button', ['url' => $url, 'color' => 'green'])  
View Order  
@endcomponent
```

Componente *panel*

El componente *panel* representa el bloque de texto dado en un panel que tiene un color de fondo ligeramente diferente que el resto del mensaje. Esto le permite llamar la atención sobre un bloque de texto dado:

```
@component('mail::panel')  
This is the panel content.  
@endcomponent
```

Componente *table*

El componente *table* le permite transformar una tabla Markdown en una tabla HTML. El componente acepta la tabla de Markdown como contenido. La alineación de la columna de la tabla se soporta utilizando la sintaxis de alineación de la tabla Markdown por defecto:

```
@component('mail::table')  
| Laravel | Table | Example |  
| ----- | :-----: | -----: |  
| Col 2 is | Centered | $10 |  
| Col 3 is | Right-Aligned | $20 |  
@endcomponent
```

Personalización de los componentes

Puede exportar todos los componentes de Mail Markdown a su propia aplicación para personalizarlos. Para exportar los componentes, utilice el comando Artisan `vendor:publish` para publicar la etiqueta `laravel-mail`:

```
php artisan vendor:publish --tag=laravel-mail
```

Este comando publicará los componentes de correo de Markdown en el directorio `resources/views/vendor/mail`. El directorio `mail` contendrá un directorio `html` y un directorio `markdown`, cada uno de los cuales contendrá sus respectivas representaciones de cada

componente disponible. Es libre de personalizar estos componentes como quiera.

Personalizar el CSS

Después de exportar los componentes, el directorio `resources/views/vendor/mail/html/themes` contendrá un archivo `default.css`. Puede personalizar el CSS en este archivo y sus estilos estarán automáticamente incrustados dentro de las representaciones HTML de sus mensajes de mail Markdown.

Si desea crear un tema completamente nuevo para los componentes de Markdown, simplemente escriba un nuevo archivo CSS dentro del directorio `html/themes` y cambie la opción `theme` de su archivo de configuración `mail`.

Vista previa de Mailables en el navegador

Al diseñar una plantilla de mailable, es conveniente hacer una vista previa rápida del mial procesado en el navegador como una típica plantilla Blade. Por esta razón, Laravel permite devolver cualquier mailable directamente desde una ruta anónima o un controlador. Cuando se devuelve un mailable, se procesará y se mostrará en el navegador, lo que permite obtener una vista previa rápida de su diseño sin necesidad de enviarlo a una dirección de correo electrónico real:

```
Route::get('/mailable', function () {
    $invoice = App\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

Enviar Correo

Para enviar un mensaje, se usa el método `to` en el `Mail facade`. El método `to` acepta una dirección de correo electrónico, una instancia de usuario o una colección de usuarios. Si se pasa un objeto o una colección de objetos, el remitente utilizará automáticamente las propiedades `email` y `name` al configurar los destinatarios del correo electrónico, así que hay que asegurarse de que estos atributos estén disponibles en los objetos. Una vez que haya especificado los destinatarios, se puede pasar una instancia de su clase mailable al método `send`:

```

<?php

namespace App\Http\Controllers;

use App\Order;
use App\Mail\OrderShipped;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
use App\Http\Controllers\Controller;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param Request $request
     * @param int $orderId
     * @return Response
     */
    public function ship(Request $request, $orderId)
    {
        $order = Order::findOrFail($orderId);

        // Ship order...

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}

```

Por supuesto, no está limitado a especificar los destinatarios "to" al enviar un mensaje. Se es libre de configurar los destinatarios "to", "cc" y "bcc", todo dentro de una única llamada a un método encadenado:

```

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));

```

Cola de Correos

Encolar un Correo

El envío de correo electrónico puede alargar drásticamente el tiempo de respuesta de la aplicación, muchos desarrolladores eligen hacer una cola de los mensajes para ser enviados en segundo plano. Laravel lo hace fácil a través del uso de su [API de colas unificada](#). Para poner en cola un mensaje de correo, se usa el método `queue` en la facade `Mail` después de especificar los destinatarios del mensaje:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

Este método se encargará automáticamente de empujar un trabajo en la cola para enviar el mensaje de correo en segundo plano. Por supuesto, hay que [configurar las colas](#) antes de utilizar esta función.

Encolar Mensajes con Retraso

Si se desea retrasar la entrega de un correo electrónico en cola, se puede usar el método `later`. Como primer argumento, el método `later` acepta una instancia de `DateTime` que indica cuándo se debe enviar el mensaje:

```
$when = Carbon\Carbon::now()->addMinutes(10);

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later($when, new OrderShipped($order));
```

Incluir a Colas Específicas

Como todas las clases mailable generadas utilizando el comando `make:mail` hacen uso del trait `Illuminate\Bus\Queueable`, se pueden llamar los métodos `onQueue` y `onConnection` en cualquier instancia de clase mailable, lo que permite especificar la conexión y el nombre de la cola para el mensaje:

```
$message = (new OrderShipped($order))
    ->onConnection('sqS')
    ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

Queueing por defecto

Si tiene clases mailable que desea que estén siempre en cola, se puede implementar el contrato `ShouldQueue` en la clase. Ahora, incluso si llama al método `send` cuando se envía, el mailable seguirá en cola ya que implementa el contrato:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    //
}
```

Correo & Desarrollo Local

Al desarrollar una aplicación que envía e-mails, es probable que no se desee enviar mensajes a direcciones de correo reales. Laravel ofrece varias maneras de "desactivar" el envío de mensajes de correo electrónico.

Log Driver

En lugar de enviar los correos electrónicos, el driver de correo `log` escribirá todos los mensajes de correo electrónico en los logs para su inspección. Para más información de como configurar una aplicación para el desarrollo, consultar el apartado de [documentación de configuración](#).

Recipiente Universal

Otra solución que proporciona Laravel es crear un receptor universal de todos los correos electrónicos enviados por el framework. De esta manera, todos los correos electrónicos generados por la aplicación serán enviados a una dirección específica, en lugar de la dirección especificada cuando se envía el mensaje. Esto se puede hacer a través de la opción `to` del archivo de configuración `config/mail.php`:

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

Mailtrap

Por último, es posible utilizar un servicio como [Mailtrap](#) y el driver `smtp` para enviar mensajes de correo electrónico a un buzón "ficticio" el cual se puede ver como un verdadero cliente de correo. Este enfoque tiene la ventaja de que permite inspeccionar realmente los correos electrónicos finales en el visor de mensajes de Mailtrap.

Eventos

Laravel dispara dos eventos durante el proceso de envío de mensajes de correo. El evento `MessageSending` se dispara antes de que se envíe un mensaje, mientras que el evento `MessageSent` se dispara después de que un mensaje ha sido enviado. Recuerde, estos eventos se disparan cuando el correo se *envía*, no cuando está en cola. Puede registrar un listener para este evento en su `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSendingMessage',
    ],
    'Illuminate\Mail\Events\MessageSent' => [
        'App\Listeners\LogSentMessage',
    ],
];
```


Notificaciones

Introducción

Crear notificaciones

Enviar notificaciones

- # Uso del *trait Notifiable*
- # Uso de la *facade Notification*
- # Especificar de canales de entrega
- # Cola de notificaciones
- # Notificaciones bajo demanda

Notificación por correo electrónico

- # Formatear mensajes de correo electrónico
- # Personalizar el destinatario
- # Personalizar el tema
- # Personalizar plantillas (*templates*)

Notificaciones de correo Markdown

- # Generación del mensaje
- # Construir el mensaje
- # Personalización de los componentes

Notificaciones de base de datos

- # Requisitos previos
- # Formatear notificaciones de bases de datos
- # Acceso a notificaciones
- # Marcar notificaciones como leidas

Transmitir notificaciones

- # Requisitos previos
- # Formatear notificaciones de difusión (*broadcasting*)
- # Cómo escuchar notificaciones

Notificaciones por SMS

- # Requisitos previos
- # Formatear notificaciones SMS
- # Personalizar el número de origen
- # Enrutar notificaciones SMS

Notificaciones de Slack

- # Requisitos previos
- # Formatear notificaciones de Slack

```
# Adjuntos en Slack  
# Enrutar notificaciones de Slack  
# Eventos de notificaciones  
# Canales personalizados
```

Introducción

Además del soporte para [enviar correo electrónico](#), Laravel proporciona soporte para el envío de notificaciones a través de una variedad de canales de entrega, incluyendo correo, SMS (via [Nexmo](#)), y [Slack](#). Las notificaciones también pueden almacenarse en una base de datos para que se muestren en su web.

Normalmente, las notificaciones deben ser breves, mensajes informativos que avisan a los usuarios de algo que ocurrió en su aplicación. Por ejemplo, si está escribiendo una aplicación de facturación, puede enviar una notificación de "factura pagada" a sus usuarios a través de los canales de correo electrónico y SMS.

Crear notificaciones

En Laravel, cada notificación está representada por una única clase (normalmente almacenada en el directorio [app/Notifications](#)). No hay que preocuparse si no se encuentra el directorio, se creará al ejecutar el comando de Artisan `make:notification` :

```
php artisan make:notification InvoicePaid
```

Este comando colocará una nueva clase de notificación en el directorio [app/Notifications](#) . Cada clase de notificación contiene un método `via` y un número variable de métodos de creación de mensajes (como `toMail` o `toDatabase`) que convierten la notificación en un mensaje optimizado para ese canal en particular.

Enviar notificaciones

Uso del *trait Notifiable*

Las notificaciones se pueden enviar de dos formas: usando el método `notify` del *trait* `Notifiable` o usando la *facade* `Notification`. Primero, exploremos usando el método del *trait*:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;
}
```

Este *trait* se incluye en el modelo `App\User` por defecto y contiene un método que puede usarse para enviar notificaciones: `notify`. El método `notify` espera recibir una instancia de una notificación:

```
use App\Notifications\InvoicePaid;

$user->notify(new InvoicePaid($invoice));
```

Recuerde que puede utilizar el *trait* `Illuminate\Notifications\Notifiable` en cualquiera de sus modelos. No se limite a incluirlo únicamente en su modelo `User`.

Uso de la *facade* `Notification`

Alternativamente, puede enviar notificaciones a través de la *facade* `Notification`. Esto es útil principalmente cuando necesita enviar una notificación a varias entidades notificables, como una colección de usuarios. Para enviar notificaciones utilizando la *facade*, pase todas las entidades notificables y la instancia de notificación al método `send`:

```
Notification::send($users, new InvoicePaid($invoice));
```

Especificación de canales de entrega

Cada clase de notificación tiene un método `via` que determina en qué canales se entregará la notificación. De serie, las notificaciones se pueden enviar a los canales `mail`, `database`, `broadcast`, `nexmo` y `slack`.

Si desea utilizar otros canales de entrega como Telegram o Pusher, consulte el sitio web Laravel Notification Channels website.

El método `via` recibe una instancia `$notifiable`, que será una instancia de la clase a la que se envía la notificación. Puede utilizar `$notifiable` para determinar en qué canales se debe entregar la notificación:

```
/**  
 * Get the notification's delivery channels.  
 *  
 * @param mixed $notifiable  
 * @return array  
 */  
  
public function via($notifiable)  
{  
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];  
}
```

Cola de notificaciones

Antes de hacer la cola de notificaciones, debe configurarla e iniciar un *worker*.

El envío de notificaciones puede tomar tiempo, especialmente si el canal necesita una llamada externa de una API para entregar la notificación. Para acelerar el tiempo de respuesta de su aplicación, deje que su notificación se coloque en la cola añadiendo la interfaz `ShouldQueue` y el trait `Queueable` a su clase. La interfaz y el trait ya se importan para todas las notificaciones generadas con `make:notification`, por lo que pueden añadirse inmediatamente a la clase de notificación:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification implements ShouldQueue
{
    use Queueable;

    // ...
}
```

Una vez que la interfaz `ShouldQueue` haya sido agregada, puede enviar la notificación normalmente. Laravel detectará la interfaz `ShouldQueue` en la clase y automáticamente pondrá en cola la entrega de la notificación:

```
$user->notify(new InvoicePaid($invoice));
```

Si desea retrasar la entrega de la notificación, puede encadenar el método `delay` en su instancia:

```
$when = Carbon::now()->addMinutes(10);

$user->notify((new InvoicePaid($invoice))->delay($when));
```

Notificaciones bajo demanda

A veces, es posible que necesite enviar una notificación a alguien que no esté almacenado como "usuario" de su aplicación. Utilizando el método `Notification::route`, puede especificar la información de enrutamiento de la notificación *ad-hoc* antes de enviar la notificación:

```
Notification::route('mail', 'taylor@laravel.com')
    ->route('nexmo', '5555555555')
    ->notify(new InvoicePaid($invoice));
```

Notificación por correo electrónico

Formatear mensajes de correo electrónico

Si una notificación soporta que se envíe como un correo electrónico, debe definir un método `toMail` en la clase de notificación. Este método recibirá una entidad `$notifiable` y devolverá una instancia de `\Illuminate\Notifications\Messages\MailMessage`. Los mensajes de correo pueden contener líneas de texto así como una "llamada a la acción". A continuación se muestra un ejemplo del método `toMail`:

```
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Messages\MailMessage  
 */  
  
public function toMail($notifiable)  
{  
    $url = url('/invoice/'.$this->invoice->id);  
  
    return (new MailMessage)  
        ->greeting('Hello!')  
        ->line('One of your invoices has been paid!')  
        ->action('View Invoice', $url)  
        ->line('Thank you for using our application!');  
}
```

Tenga en cuenta que se está usando `$this->invoice->id` en el método `toMail`. Puede pasar cualquier dato que necesite su notificación para generar su mensaje en el constructor.

En este ejemplo se registra un saludo, una línea de texto, una llamada a la acción, y luego otra línea de texto. Estos métodos proporcionados por el objeto `MailMessage` hacen que formatear pequeños correos electrónicos transaccionales sea fácil y rápido. El canal de correo traducirá los componentes del mensaje en una plantilla de correo electrónico HTML agradable y receptiva con una contraparte de texto plano. A continuación se muestra un ejemplo de un correo electrónico generado por el canal `mail`:

Hello!

One of your invoices has been paid!

[View Invoice](#)

Thank you for using our application!

Regards,
Laravel

If you're having trouble clicking the "View Invoice" button, copy and paste the URL below into your web browser:

<https://example.com/invoice/1>

© 2016 [Laravel](#). All rights reserved.

Cuando envíe notificaciones por correo, asegúrese de establecer el valor `name` en su archivo de configuración `config/app.php`. Este valor se utilizará en el encabezado y pie de página de sus mensajes de notificación de correo.

Otras opciones de formato para las notificaciones

En lugar de definir las "líneas" del texto en la clase de notificación, puede utilizar el método `view` para especificar una plantilla personalizada que debería utilizarse para mostrar el correo electrónico de notificación:

```
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Messages\MailMessage  
 */  
  
public function toMail($notifiable)  
{  
    return (new MailMessage)->view(  
        'emails.name', ['invoice' => $this->invoice]  
    );  
}
```

Además, puede devolver un objeto [mailable](#) desde el método `toMail`:

```
use App\Mail\InvoicePaid as Mailable;  
  
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return Mailable  
 */  
  
public function toMail($notifiable)  
{  
    return (new Mailable($this->invoice))->to($this->user->email);  
}
```

Mensajes de error

Algunas notificaciones informan a los usuarios de errores, como el pago fallido de facturas. Puede indicar que un mensaje de correo electrónico se refiere a un error llamando al método `error` al crear su mensaje. Cuando se utiliza el método `error` en un mensaje de correo electrónico, el botón de llamada a la acción estará rojo en lugar de azul:

```
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Message  
 */  
  
public function toMail($notifiable)  
{  
    return (new MailMessage)  
        ->error()  
        ->subject('Notification Subject')  
        ->line('...');  
}
```

Personalizar el destinatario

Cuando envíe notificaciones a través del canal `mail`, el sistema de notificaciones buscará automáticamente una propiedad `email` en su entidad. Puede personalizar qué dirección de correo electrónico se utiliza para enviar la notificación definiendo un método `routeNotificationForMail` en la entidad:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the mail channel.
     *
     * @return string
     */
    public function routeNotificationForMail()
    {
        return $this->email_address;
    }
}
```

Personalizar el tema

De forma predeterminada, el asunto del correo electrónico es el nombre de la clase de la notificación con formato "title case". Por lo tanto, si su clase de notificación se llama `InvoicePaid`, el asunto del correo electrónico será `Invoice Paid`. Si desea especificar un asunto explícito para el mensaje, puede llamar al método `subject` al crear el mensaje:

```
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Messages\MailMessage  
 */  
  
public function toMail($notifiable)  
{  
    return (new MailMessage)  
        ->subject('Notification Subject')  
        ->line('...');  
}
```

Personalizar plantillas (*templates*)

Puede modificar la plantilla HTML y de texto plano utilizada en las notificaciones por correo electrónico publicando los recursos del paquete de notificación. Después de ejecutar este comando, las plantillas de notificación de correo se ubicarán en el directorio

`resources/views/vendor/notifications :`

```
php artisan vendor:publish --tag=laravel-notifications
```

Notificaciones de correo de Markdown

Las notificaciones de correo de Markdown le permiten aprovechar las plantillas prediseñadas de las notificaciones de correo electrónico, mientras que dan más libertad para escribir mensajes más largos y personalizados. Dado que los mensajes están escritos en Markdown, Laravel es capaz de generar plantillas HTML amigables para los mensajes, mientras que genera a su vez una contraparte de texto plano.

Generación del mensaje

Para generar una notificación con una plantilla de Markdown correspondiente, puede utilizar la opción `--markdown` del comando Artisan `make:notification` :

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

Al igual que todas las demás notificaciones de correo, las notificaciones que utilizan plantillas de Markdown deben definir un método `toMail` en su clase. Sin embargo, en lugar de utilizar los métodos `line` y `action` para construir la notificación, utilice el método `markdown` para especificar el nombre de la plantilla Markdown que debe utilizarse:

```
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return \Illuminate\Notifications\Messages\MailMessage  
 */  
  
public function toMail($notifiable)  
{  
    $url = url('/invoice/'.$this->invoice->id);  
  
    return (new MailMessage)  
        ->subject('Invoice Paid')  
        ->markdown('mail.invoice.paid', ['url' => $url]);  
}
```

Construir el mensaje

Las notificaciones de correo de Markdown utilizan una combinación de componentes Blade y sintaxis de Markdown que le permiten construir fácilmente notificaciones aprovechando los componentes predefinidos de Laravel:

```
@component('mail::message')  
# Invoice Paid  
  
Your invoice has been paid!  
  
@component('mail::button', ['url' => $url])  
View Invoice  
@endcomponent  
  
Thanks,<br>  
{{ config('app.name') }}  
@endcomponent
```

Componente `button`

El componente `button` muestra un enlace como un botón centrado. El componente acepta dos argumentos, `url` y un `color` opcional. Los colores admitidos son azul, verde y rojo (`blue`, `green`, and `red`). Puede añadir tantos componentes `button` a una notificación como desee:

```
@component('mail::button', ['url' => $url, 'color' => 'green'])  
View Invoice  
@endcomponent
```

Componente `panel`

El componente `panel` representa el bloque de texto dado en un panel que tiene un color de fondo ligeramente diferente al del resto de la notificación. Esto le permite llamar la atención sobre un bloque de texto dado:

```
@component('mail::panel')  
This is the panel content.  
@endcomponent
```

Componente `table`

El componente `table` le permite transformar una tabla Markdown en una tabla HTML. El componente acepta la tabla de Markdown como contenido. La alineación de la columna de la tabla se soporta utilizando la sintaxis de alineación de la tabla Markdown por defecto:

```
@component('mail::table')  
| Laravel | Table | Example |  
| ----- | :-----: | -----: |  
| Col 2 is | Centered | $10 |  
| Col 3 is | Right-Aligned | $20 |  
@endcomponent
```

Personalización de los componentes

Puede exportar todos los componentes de notificación Markdown a su propia aplicación para personalizarlos. Para exportar los componentes, utilice el comando Artisan `vendor:publish` para publicar la etiqueta `laravel-mail`:

```
php artisan vendor:publish --tag=laravel-mail
```

Este comando publicará los componentes de correo de Markdown en el directorio `resources/views/vendor/mail`. El directorio `mail` contendrá un directorio `html` y un directorio `markdown`, cada uno de los cuales contendrá sus respectivas representaciones de cada componente disponible. Es libre de personalizar estos componentes como quiera.

Personalizar el CSS

Después de exportar los componentes, el directorio `resources/views/vendor/mail/html/themes` contendrá un archivo `default.css`. Puede personalizar el CSS en este archivo y sus estilos estarán automáticamente incrustados dentro de las representaciones HTML de sus notificaciones Markdown.

Si desea crear un tema completamente nuevo para los componentes de Markdown, simplemente escriba un nuevo archivo CSS dentro del directorio `html/themes` y cambie la opción `theme` de su archivo de configuración `mail`.

Notificaciones de base de datos

Requisitos previos

El canal de notificación `database` almacena la información en una tabla de base de datos. Esta tabla contendrá información como el tipo de notificación y los datos JSON personalizados que describen la notificación.

Puede consultar la tabla para mostrar las notificaciones en la interfaz de usuario de la aplicación. Pero, antes de que pueda hacerlo, necesitará crear una tabla de base de datos para guardar sus notificaciones. Puede utilizar el comando `notifications:table` para generar una migración con el esquema de tabla adecuado:

```
php artisan notifications:table  
  
php artisan migrate
```

Formatear notificaciones de bases de datos

Si una notificación soporta que se almacene en una tabla de la base de datos, debe definir un método `toDatabase` o `toArray` en la clase de notificación. Este método recibirá una entidad `$notifiable` y debería devolver un *array* PHP simple. El *array* devuelto se transformará a JSON y se guardará en la columna `data` de la tabla `notifications`. Veamos un ejemplo del método `toArray`:

```
/**  
 * Get the array representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return array  
 */  
  
public function toArray($notifiable)  
{  
    return [  
        'invoice_id' => $this->invoice->id,  
        'amount' => $this->invoice->amount,  
    ];  
}
```

toDatabase versus toArray

El método `toArray` también se utiliza por el canal `broadcast` para determinar qué datos transmitir a su cliente JavaScript. Si desea tener dos representaciones de *array* diferentes para los canales `database` y `broadcast`, debe definir un método `toDatabase` en lugar de un método `toArray`.

Acceso a notificaciones

Una vez que las notificaciones se almacenan en la base de datos, necesita una forma conveniente de acceder a ellas desde sus entidades *notifiables*. El *trait* `Illuminate\Notifications\Notifiable`, que se incluye en el modelo predeterminado de `App\User`, incluye una relación Eloquent con `notifications` que devuelve las notificaciones de la entidad. Para obtener notificaciones, puede acceder a este método como cualquier otra relación Eloquent. De forma predeterminada, las notificaciones se ordenarán por el campo de fecha `created_at`:

```
$user = App\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

Si desea recuperar sólo las notificaciones "no leídas", puede utilizar la relación `unreadNotifications`. De nuevo, estas notificaciones se clasificarán por el campo de fecha `created_at`:

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

Para acceder a las notificaciones desde su cliente JavaScript, debe definir un controlador para su aplicación que devuelva las notificaciones de una entidad *notifiable*, como el usuario actual. A continuación, puede realizar una solicitud HTTP a la URI de ese controlador desde su cliente JavaScript.

Marcar notificaciones como leidas

Normalmente, querrá marcar una notificación como "leída" cuando un usuario la vea. El *trait* `Illuminate\Notifications\Notifiable` proporciona un método `markAsRead`, que actualiza la columna `read_at` en el registro de la base de datos de notificación:

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

Sin embargo, en lugar de pasar por cada notificación, puede utilizar el método `markAsRead` directamente en una colección de notificaciones:

```
$user->unreadNotifications->markAsRead();
```

También puede utilizar una consulta de actualización en masa para marcar todas las notificaciones como leídas sin recuperarlas de la base de datos:

```
$user = App\User::find(1);

$user->unreadNotifications()->update(['read_at' => Carbon::now()]);
```

Por supuesto, puede borrar las notificaciones para eliminarlas completamente de la tabla:

```
$user->notifications()->delete();
```

Transmitir notificaciones

Requisitos previos

Antes de hacer difusión (*broadcasting*) de las notificaciones, deberá configurar y familiarizarse con los servicios de [difusión de eventos](#) de Laravel. La difusión de eventos proporciona una forma de reaccionar a eventos lanzados por Laravel del lado del servidor desde su cliente JavaScript.

Formatear notificaciones de difusión (*broadcasting*)

El canal `broadcast` difunde notificaciones usando el servicio de [difusión de eventos](#) permitiendo a su cliente JavaScript responder a las notificaciones en tiempo real. Si una notificación soporta difusión (*broadcasting*), deberá definir un método `toBroadcast` en la clase de notificación. Este método recibirá una entidad `$notifiable` y deberá devolver una instancia de `BroadcastMessage`. Los datos devueltos serán codificados como JSON y se entregaran a tu cliente JavaScript. Veamos un ejemplo del método `toBroadcast` :

```

use Illuminate\Notifications\Messages\BroadcastMessage;

/**
 * Get the broadcastable representation of the notification.
 *
 * @param mixed $notifiable
 * @return BroadcastMessage
 */
public function toBroadcast($notifiable)
{
    return new BroadcastMessage([
        'invoice_id' => $this->invoice->id,
        'amount' => $this->invoice->amount,
    ]);
}

```

Configuración de la cola de difusión (*Broadcast Queue*)

Todas las notificaciones *broadcast* se colocan en una cola para su difusión. Si desea configurar la conexión de colas (*queue connection*) o el nombre de cola que se utiliza para la operación de *broadcast*, puede utilizar los métodos `onConnection` y `onQueue` de `BroadcastMessage` :

```

return (new BroadcastMessage($data))
    ->onConnection('sqS')
    ->onQueue('broadcasts');

```

Además de los datos que especifique, las notificaciones *broadcast* también contendrán un campo `type` que contendrá el nombre de clase de la notificación.

Cómo capturar notificaciones

Las notificaciones serán difundidas en un canal privado formateado usando la convención `{notifiable}.{id}` . Por tanto, si envía una notificación a una instancia de `App\User` con un ID de `1` , la notificación será difundida sobre el canal privado `App.User.1` . Cuando use [Laravel Echo](#) podrá escuchar fácilmente a las notificaciones en un canal usando el *helper* `notification` :

```
Echo.private('App.User.' + userId)
    .notification((notification) => {
        console.log(notification.type);
    });
});
```

Personalizar el canal de notificación

Si desea personalizar los canales en los cuales una entidad *notifiable* recibirá sus notificaciones de difusión (*broadcast notifications*), deberá definir un método `receivesBroadcastNotificationsOn` en la entidad *notifiable*:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The channels the user receives notification broadcasts on.
     *
     * @return string
     */
    public function receivesBroadcastNotificationsOn()
    {
        return 'users.'.$this->id;
    }
}
```

Notificaciones por SMS

Requisitos previos

El envío de notificaciones SMS en Laravel se opera por [Nexmo](#). Antes de que pueda enviar notificaciones a través de Nexmo, necesita instalar el paquete de Composer `nexmo/client` y añadir algunas opciones de configuración a su archivo `config/services.php`. Puede copiar el ejemplo de configuración a continuación para comenzar:

```
'nexmo' => [
    'key' => env('NEXMO_KEY'),
    'secret' => env('NEXMO_SECRET'),
    'sms_from' => '15556666666',
],
```

La opción `sms_from` es el número de teléfono desde el que se enviarán sus mensajes SMS. Debe generar un número de teléfono para su aplicación en el panel de control de Nexmo.

Formatear notificaciones SMS

Si una notificación soporta el ser enviada como SMS, debe definir un método `toNexmo` en la clase de notificación. Este método recibirá una entidad `$notifiable` y deberá devolver una instancia de `Illuminate\Notifications\Messages\NexmoMessage`:

```
/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your SMS message content');
}
```

Contenido Unicode

Si su mensaje SMS contiene caracteres unicode, debe llamar al método `unicode` al construir la instancia `NexmoMessage`:

```

/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your unicode message')
        ->unicode();
}

```

Personalizar el número de origen

Si desea enviar algunas notificaciones desde un número de teléfono diferente al especificado en su archivo `config/services.php`, puede utilizar el método `from` en una instancia `NexmoMessage`:

```

/**
 * Get the Nexmo / SMS representation of the notification.
 *
 * @param mixed $notifiable
 * @return NexmoMessage
 */
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your SMS message content')
        ->from('15554443333');
}

```

Enrutar notificaciones SMS

Al enviar notificaciones a través del canal `nexmo`, el sistema de notificaciones buscará automáticamente un atributo `phone_number` en la entidad `notifiable`. Si desea personalizar el número de teléfono al que se envía la notificación, defina un método `routeNotificationForNexmo` en la entidad:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Nexmo channel.
     *
     * @return string
     */
    public function routeNotificationForNexmo()
    {
        return $this->phone;
    }
}
```

Notificaciones de Slack

Requisitos previos

Antes de poder enviar notificaciones a través de Slack, debe instalar la librería *Guzzle HTTP* a través de Composer:

```
composer require guzzlehttp/guzzle
```

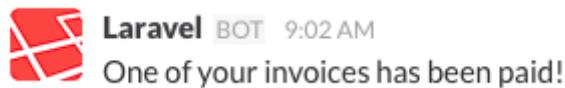
También necesitará configurar una integración "[Incoming Webhook](#)" en su equipo de Slack. Esta integración le proporcionará una URL que puede utilizar para [erutar las notificaciones de Slack](#).

Formatear notificaciones de Slack

Si una notificación soporta el envío de un mensaje a Slack, debe definir un método `toSlack` en la clase. Este método recibirá una entidad `$notifiable` y devolverá una instancia de `Illuminate\Notifications\Messages\MailMessage`. Los mensajes de Slack pueden contener texto así como un "archivo adjunto" que formatea texto adicional o una `array` de campos. Veamos un ejemplo básico de `toSlack`:

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
    return (new SlackMessage)  
        ->content('One of your invoices has been paid!');  
}
```

En este ejemplo sólo estamos enviando una única línea de texto a Slack, que creará un mensaje parecido al siguiente:



Personalización del remitente & destinatario

Puede utilizar los métodos `from` y `to` para personalizar el remitente y el destinatario. El método `from` acepta un nombre de usuario e identificador emoji, mientras que el método `to` acepta un canal o nombre de usuario:

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
    return (new SlackMessage)  
        ->from('Ghost', ':ghost:')  
        ->to('#other')  
        ->content('This will be sent to #other');  
}
```

También puede utilizar una imagen como su logotipo en lugar de un emoji:

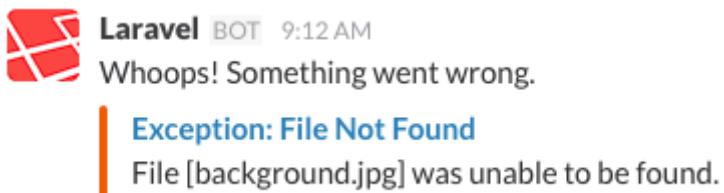
```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
    return (new SlackMessage)  
        ->from('Laravel')  
        ->image('https://laravel.com/favicon.png')  
        ->content('This will display the Laravel logo next to the message');  
}
```

Adjuntos en Slack

También puede agregar "adjuntos" a los mensajes de Slack. Los archivos adjuntos proporcionan opciones de formato más completas que los mensajes de texto simples. En este ejemplo, enviaremos una notificación de error sobre una excepción ocurrida en una aplicación, incluyendo un enlace para ver más detalles sobre la excepción:

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
    $url = url('/exceptions/'.$this->exception->id);  
  
    return (new SlackMessage)  
        ->error()  
        ->content('Whoops! Something went wrong.')  
        ->attachment(function ($attachment) use ($url) {  
            $attachment->title('Exception: File Not Found', $url)  
                ->content('File [background.jpg] was not found.');        });  
}
```

El ejemplo anterior generará un mensaje de Slack que se parecerá al siguiente:



Los adjuntos también le permiten especificar un *array* de datos que deben presentarse al usuario. Los datos proporcionados se presentarán en un formato de tabla para facilitar la lectura:

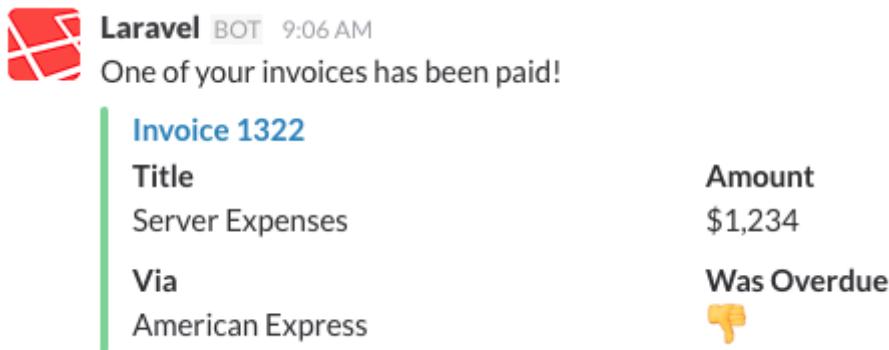
```

/**
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
{
    $url = url('/invoices/'.$this->invoice->id);

    return (new SlackMessage)
        ->success()
        ->content('One of your invoices has been paid!')
        ->attachment(function ($attachment) use ($url) {
            $attachment->title('Invoice 1322', $url)
            ->fields([
                'Title' => 'Server Expenses',
                'Amount' => '$1,234',
                'Via' => 'American Express',
                'Was Overdue' => '::-1:',
            ]);
        });
}

```

El ejemplo anterior creará un mensaje de Slack que se parecerá al siguiente:



Contenido *Markdown* adjunto

Si algunos de los campos de datos adjuntos contienen Markdown, puede utilizar el método `markdown` para indicar a Slack que analice y muestre los campos de datos adjuntos como texto con formato Markdown. Los valores aceptados por este método son: `pretext` , `text` , y /

o `fields`. Para obtener más información sobre el formato de Slack para adjuntar archivos consulte la documentación [Slack API documentation](#):

```
/**  
 * Get the Slack representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return SlackMessage  
 */  
  
public function toSlack($notifiable)  
{  
    $url = url('/exceptions/'. $this->exception->id);  
  
    return (new SlackMessage)  
        ->error()  
        ->content('Whoops! Something went wrong.')  
        ->attachment(function ($attachment) use ($url) {  
            $attachment->title('Exception: File Not Found', $url)  
                ->content('File [background.jpg] was *not found*.')  
                ->markdown(['text']);  
        });  
}
```

Enrutar notificaciones de Slack

Para enrutar las notificaciones de Slack a la ubicación apropiada, defina un método `routeNotificationForSlack` en su entidad *notifiable*. Esto debería devolver la URL del *webhook* al que se debe enviar la notificación. Las URLs de *webhook* se pueden generar añadiendo un servicio "Incoming Webhook" a su equipo de Slack:

```

<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Route notifications for the Slack channel.
     *
     * @return string
     */
    public function routeNotificationForSlack()
    {
        return $this->slack_webhook_url;
    }
}

```

Eventos de notificación

Cuando se envía una notificación, el sistema de notificaciones activa el evento `Illuminate\Notifications\Events\NotificationSent`. Contiene la entidad "notifiable" y la propia instancia de notificación. Se pueden registrar escuchas para este evento en `EventServiceProvider` :

```

/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Notifications\Events\NotificationSent' => [
        'App\Listeners\LogNotification',
    ],
];

```

Después de registrar a los escuchas en su `EventServiceProvider`, utilice el comando Artisan `event:generate` para generar rápidamente los *listeners* para estos eventos.

Dentro de un receptor de eventos, puede acceder a las propiedades `notifiable`, `notification`, y `channel` en el evento para obtener más información sobre el destinatario de la notificación o sobre la notificación en sí:

```
/**  
 * Handle the event.  
 *  
 * @param NotificationSent $event  
 * @return void  
 */  
  
public function handle(NotificationSent $event)  
{  
    // $event->channel  
    // $event->notifiable  
    // $event->notification  
}
```

Canal personalizado

Laravel incluye varios canales de notificación, pero es posible programar sus propios *drivers* para entregar notificaciones a través de otros canales. Laravel lo hace sencillo. Para empezar, defina una clase que contenga un método `send`. El método debe recibir dos argumentos:

`$notifiable` y `$notification`:

```
<?php

namespace App\Channels;

use Illuminate\Notifications\Notification;

class VoiceChannel
{
    /**
     * Send the given notification.
     *
     * @param mixed $notifiable
     * @param \Illuminate\Notifications\Notification $notification
     * @return void
     */
    public function send($notifiable, Notification $notification)
    {
        $message = $notification->toVoice($notifiable);

        // Send notification to the $notifiable instance...
    }
}
```

Una vez que su clase de canal de notificación ha sido definida, simplemente puede devolver el nombre de la clase desde el método `via` de cualquiera de sus notificaciones:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use App\Channels\VoiceChannel;
use App\Channels\Messages\VoiceMessage;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;

class InvoicePaid extends Notification
{
    use Queueable;

    /**
     * Get the notification channels.
     *
     * @param mixed $notifiable
     * @return array|string
     */
    public function via($notifiable)
    {
        return [VoiceChannel::class];
    }

    /**
     * Get the voice representation of the notification.
     *
     * @param mixed $notifiable
     * @return VoiceMessage
     */
    public function toVoice($notifiable)
    {
        // ...
    }
}
```

Desarrollo de paquetes

Introducción

Nota sobre *facades*

Descubrir paquetes

Service Providers

Recursos – Resources

Configuración

Migraciones

Rutas

Traducciones

Vistas

Comandos

Publicar assets

Publicar grupos de ficheros

Introducción

Los paquetes son el método principal para añadir funcionalidad a Laravel. Pueden ser cualquier cosa, desde una gran manera de trabajar con fechas como [Carbon](#), o un *framework* BDD completo como [Behat](#).

Por supuesto, hay diferentes tipos de paquetes. Algunos paquetes son independientes, lo que significa que funcionan con cualquier framework PHP. Carbon y Behat son ejemplos de paquetes independientes. Cualquiera de estos paquetes puede utilizarse en Laravel con tan solo añadirlos en el archivo `composer.json`.

Por otro lado, otros paquetes son específicos, a propósito, para utilizarlos con Laravel. Estos paquetes pueden tener rutas, controladores, vistas y configuración específicamente desarrollada para una aplicación Laravel. Esta guía principalmente cubre el desarrollo de esos paquetes específicos para Laravel.

Nota sobre *facades*

Al escribir una aplicación de Laravel, generalmente no importa si usa contratos o *facades* ya que ambos proporcionan niveles esencialmente iguales de *testing*. Sin embargo, su paquete no tendrá acceso a todos los *helpers* de *testing* de Laravel. Si desea poder escribir sus *tests* como si existieran dentro de una aplicación típica de Laravel, puede utilizar el paquete [Orchestral Testbench](#).

Descubrir paquetes

En el fichero de configuración `config/app.php` de Laravel, la opción `providers` define una lista de *service providers* que Laravel debe cargar. Cuando alguien instala su paquete, normalmente querrá que su *service provider* se incluya en esta lista. En lugar de exigir a los usuarios que agreguen manualmente su *service provider* a la lista, puede definirlo en la sección `extra` del archivo `composer.json` de su paquete. Además de los *service provider*, también puede listar cualquier [`fascade`](#) que deseé registrar:

```
"extra": {  
    "laravel": {  
        "providers": [  
            "Barryvdh\\Debugbar\\ServiceProvider"  
        ],  
        "aliases": {  
            "Debugbar": "Barryvdh\\Debugbar\\Facade"  
        }  
    }  
},
```

Una vez que su paquete ha sido configurado para ser descubierto, Laravel registrará automáticamente sus *service providers* y *facades* cuando se instale, creando una experiencia de instalación sencillísima para los usuarios de su paquete.

Optar por no descubrir paquetes

Si usted es el consumidor de un paquete y desea deshabilitar esta característica, puede listar el nombre del paquete en la sección `extra` del archivo `composer.json` de su aplicación:

```
"extra": {  
    "laravel": {  
        "dont-discover": [  
            "barryvdh/laravel-debugbar"  
        ]  
    }  
},
```

Puede deshabilitarlo para todos los paquetes utilizando el carácter * dentro de la directiva `dont-discover` de su aplicación:

```
"extra": {  
    "laravel": {  
        "dont-discover": [  
            "*"  
        ]  
    }  
},
```

Service Providers

Los [Service Providers](#) son el punto de conexión entre el paquete y Laravel. Un *service provider* es responsable de incluir cosas en el [service container](#) de Laravel e informar al framework de donde cargar recursos como vistas, configuración o archivos de idioma.

Un *service provider* hereda de la clase `Illuminate\Support\ServiceProvider` y contiene dos métodos: `register` y `boot`. La clase base `ServiceProvider` se encuentra en el paquete de Composer `illuminate/support`, el cual debería agregarse a sus dependencias del propio paquete. Para obtener más información sobre la estructura y el propósito de los *service providers*, consulte [su documentación](#).

Recursos – Resources

Configuración

Normalmente, necesitará publicar el archivo de configuración de su paquete en el propio directorio `config` de la aplicación. Esto permite a los usuarios modificar fácilmente las opciones por defecto de configuración del paquete. Para permitir que sus archivos de configuración se publiquen, llame al método `publishes` desde el método `boot` de su `service provider`:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__ . '/path/to/config/courier.php' => config_path('courier.php'),
    ]);
}
```

Ahora, cuando los usuarios de su paquete ejecuten el comando `vendor:publish` de Laravel, su archivo se copiará en la ubicación de publicación especificada. Por supuesto, una vez que su configuración ha sido publicada, se puede acceder a sus valores como cualquier otro archivo de configuración:

```
$value = config('courier.option');
```

No debe definir *Closures* en los archivos de configuración. No se pueden "serializar" correctamente cuando los usuarios ejecutan el comando Artisan `config:cache`.

Configuración de paquetes por defecto

También puede fusionar su propio archivo de configuración de paquetes con la copia publicada de la aplicación. Esto permitirá a sus usuarios definir sólo las opciones que realmente desean anular en la copia publicada de la configuración. Para combinar las configuraciones, se usa el método `mergeConfigFrom` dentro del método `register` del `service provider`:

```
/**  
 * Register bindings in the container.  
 *  
 * @return void  
 */  
  
public function register()  
{  
    $this->mergeConfigFrom(  
        __DIR__ . '/path/to/config/courier.php', 'courier'  
    );  
}
```

Este método sólo fusiona el primer nivel del *array* de configuración. Si sus usuarios definen parcialmente un *array* de configuración multidimensional, las opciones que faltan no se fusionarán.

Rutas

Si su paquete contiene rutas, puede cargarlas usando el método `loadRoutesFrom`. Este método determinará automáticamente si las rutas de la aplicación están almacenadas en caché y no cargará el archivo de ser así:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    $this->loadRoutesFrom(__DIR__ . '/routes.php');  
}
```

Migraciones

Si su paquete contiene [migraciones](#), puede utilizar el método `loadMigrationsFrom` para informar a Laravel cómo cargarlas. El método `loadMigrationsFrom` acepta la ruta a las migraciones de su paquete como su único argumento:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    $this->loadMigrationsFrom(__DIR__ . '/path/to/migrations');  
}
```

Una vez registradas las migraciones de su paquete, se ejecutarán automáticamente cuando se ejecute el comando `php artisan migrate`. No es necesario exportarlos al directorio principal de la aplicación `database/migrations`.

Traducciones

Si el paquete contiene [archivos de traducción](#), tiene disponible el método `loadTranslationsFrom` para informar a Laravel como cargarlos. Por ejemplo, si su paquete se llama `courier`, debería añadir lo siguiente al método `boot` de su *service provider*:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    $this->loadTranslationsFrom(__DIR__ . '/path/to/translations', 'courier');  
}
```

Las traducciones de paquetes son referenciadas usando la convención de sintaxis `paquete::file.line`. Por lo tanto, se puede cargar del paquete `courier` la línea `welcome` del archivo `messages` de la siguiente forma:

```
echo trans('courier::messages.welcome');
```

Publicar traducciones

Si se quiere publicar las traducciones de un paquete dentro del directorio `resources/lang/vendor` de la aplicación, se puede usar el método `publishes` del service provider. El método `publishes` acepta un *array* de rutas de paquetes y sus ubicaciones de publicación deseadas. Por ejemplo, para publicar los archivos de traducción del paquete `courier` puede hacer lo siguiente:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    $this->loadTranslationsFrom(__DIR__ . '/path/to/translations', 'courier');  
  
    $this->publishes([  
        __DIR__ . '/path/to/translations' => resource_path('lang/vendor/courier'),  
    ]);  
}
```

Ahora, cuando los usuarios de su paquete ejecuten el comando Artisan `vendor:publish` de Laravel, las traducciones de su paquete se publicarán en la ubicación de publicación especificada.

Vistas

Para registrar las `vistas` del paquete con Laravel, es necesario informar al framework donde se encuentran. Esto se puede hacer utilizando el método `loadViewsFrom` del service provider. El método `loadViewsFrom` acepta dos argumentos: la ruta a las vistas y el nombre del paquete. Por ejemplo, si el nombre de su paquete es `courier`, puede añadir lo siguiente al método `boot` de su service provider:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    $this->loadViewsFrom(__DIR__ . '/path/to/views', 'courier');  
}
```

Las vistas de paquetes se referencian usando la convención de sintaxis `package::view`. Por lo tanto, una vez que su ruta de vista está registrada en un *service provider*, puede cargar la vista `admin` desde el paquete `courier`:

```
Route::get('admin', function () {  
    return view('courier::admin');  
});
```

Sobrescribir las vistas de un paquete

Cuando utiliza el método `loadViewsFrom`, Laravel registra dos ubicaciones para sus vistas: el directorio `resources/views/vendor` de la aplicación y el directorio que especifique. Por lo tanto, usando el ejemplo `courier`, Laravel primero comprobará si el desarrollador ha proporcionado una versión personalizada de la vista en `resources/views/vendor/vendor/courier`. Si no se encuentra, Laravel buscará la vista en el directorio que se especificó en el método `loadViewsFrom`. Esto hace que sea fácil para los usuarios de paquetes personalizar / anular las vistas de su paquete.

Publicar las vistas

Si se quiere publicar las vistas de un paquete dentro del directorio `resources/views/vendor` de la aplicación, se puede usar el método `publishes` del *service provider*. El método `publishes` acepta un conjunto de rutas de vista de paquetes y sus ubicaciones de publicación deseadas:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    $this->loadViewsFrom(__DIR__.'/path/to/views', 'courier');  
  
    $this->publishes([  
        __DIR__.'/path/to/views' => resource_path('views/vendor/courier'),  
    ]);  
}
```

Ahora, cuando los usuarios de su paquete ejecuten el comando Artisan `vendor:publish` de Laravel, las vistas de su paquete se copiarán en la ubicación de publicación especificada.

Comandos

Para registrar los comandos Artisan de su paquete con Laravel, puede usar el método `commands`. Este método espera un *array* de nombres de clases de comando. Una vez registrados los comandos, puede ejecutarlos utilizando [Artisan CLI](#):

```
/**  
 * Bootstrap the application services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    if ($this->app->runningInConsole()) {  
        $this->commands([  
            FooCommand::class,  
            BarCommand::class,  
        ]);  
    }  
}
```

Publicar assets

Su paquete puede tener recursos como JavaScript, CSS e imágenes. Para publicar estos recursos dentro del directorio `public` de la aplicación, se utiliza el método `publishes` del *service provider*. En este ejemplo, añadiremos también una etiqueta `public`, la cual puede utilizarse para publicar grupos de recursos relacionados:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    $this->publishes([  
        __DIR__ . '/path/to/assets' => public_path('vendor/courier'),  
    ], 'public');  
}
```

Ahora, cuando los usuarios del paquete ejecuten el comando `vendor:publish`, sus activos se copiarán en la ubicación de publicación especificada. Como normalmente necesitará sobrescribir los activos cada vez que actualice el paquete, puede utilizar el indicador `--force`:

```
php artisan vendor:publish --tag=public --force
```

Publicar grupos de ficheros

Puedes querer publicar un grupo de recursos del paquete de forma separada. Por ejemplo, es posible que desee permitir a sus usuarios publicar los archivos de configuración de su paquete sin verse forzado a publicar los recursos. Puede hacerlo "etiquetándolos" cuando llame al método `publishes` del proveedor de servicios de un paquete. Por ejemplo, usemos etiquetas para definir dos grupos de publicación en el método `boot` de un *service provider* de un paquete:

```
/**  
 * Perform post-registration booting of services.  
 *  
 * @return void  
 */  
  
public function boot()  
{  
    $this->publishes([  
        __DIR__ . '/../config/package.php' => config_path('package.php')  
    ], 'config');  
  
    $this->publishes([  
        __DIR__ . '/../database/migrations/' => database_path('migrations')  
    ], 'migrations');  
}
```

Ahora los usuarios pueden publicar estos grupos por separado haciendo referencia a su etiqueta al ejecutar el comando `vendor:publish` :

```
php artisan vendor:publish --tag=config
```

Colas – Queues

Introducción

- # Conexiones y Colas
- # Prerrequisitos del driver

Creando los trabajos(Jobs)

- # Generando las Clases Trabajo
- # Estructura de la Clase

Enviar trabajos a la cola

- # Retrasar el envío
- # Encadenado de trabajos
- # Customizing The Queue & Connection
- # Especificar mayor cantidad de intentos de trabajo / tiempo de espera
- # Límites de velocidad
- # Manejo de Errores

Ejecutar el Procesador de cola

- # Priorización de colas
- # Queue Workers & Deployment
- # Job Expirations & Timeouts

Supervisor Configuration

Gestionar trabajos fallidos

- # Cleaning Up After Failed Jobs
- # Eventos de trabajos fallidos
- # Reintentar trabajos fallidos

Eventos de trabajos

Introducción

Laravel ahora ofrece Horizon, un hermoso *dashboard* y sistema de configuración para sus colas de Redis. Consulte la documentación completa de Horizon para obtener más información.

Las colas de Laravel proporcionan una API unificada en una variedad de backends de cola diferentes, como Beanstalk, Amazon SQS, Redis o incluso una base de datos relacional. *Queues* permite posponer el procesamiento de una tarea que consume mucho tiempo, como enviar un correo electrónico, hasta un momento posterior. Aplazar estas tareas que consumen mucho tiempo acelera drásticamente las peticiones web de la aplicación.

El archivo de configuración de *queue* de trabajo se almacena en `config.queue.php`. En este archivo se encuentran las configuraciones de conexión para cada uno de los *queue drivers* incorporados con el framework, que incluye una base de datos, [Beanstalkd](#), [Amazon SQS](#), [Redis](#), y un driver síncrono que ejecutará trabajos inmediatamente (para uso local). Un *queue driver* `null` también está incluido, el cual simplemente descarta los trabajos.

Conexiones y Colas

Antes de comenzar con las colas en Laravel, es importante comprender la distinción entre "conexiones" (*connections*) y "colas" (*queues*). En el archivo de configuración `config/queue.php`, hay una opción de configuración `connections`. Esta opción define una conexión particular a un servicio *backend* como Amazon SQS, Beanstalk o Redis. Sin embargo, cualquier *queue connection* puede tener múltiples "queue" que pueden considerarse como diferentes pilas o montones de trabajos en cola.

Tenga en cuenta que cada ejemplo de configuración de conexión en el archivo de configuración `queue` contiene un atributo `queue`. Esta es la *queue* predeterminada a la que se le van a mandar los trabajos cuando se envíen a una conexión dada. En otras palabras, si envía un trabajo sin definir explícitamente a qué *queue* debe enviarse, el trabajo se colocará en la *queue* definida en el atributo `queue` de la configuración de conexión:

```
// This job is sent to the default queue...
Job::dispatch();

// This job is sent to the "emails" queue...
Job::dispatch()->onQueue('emails');
```

Es posible que algunas aplicaciones no necesiten insertar trabajos en varias *queues*, sino que prefieran tener una simple *queue*. Sin embargo, enviar trabajos a varias colas puede ser especialmente útil para las aplicaciones que desean priorizar o segmentar cómo se procesan los trabajos, ya que el *queue worker*(gestor de trabajos de cola) de Laravel le permite especificar cual cola debe procesar según su prioridad. Por ejemplo, si envía trabajos a una cola `high` (prioridad alta), se ejecutara un *worker* que les dé una mayor prioridad de procesamiento:

```
php artisan queue:work --queue=high,default
```

Pre requisitos del driver

Base de datos

Para usar el *queue driver* `database`, se necesita una base de datos para guardar los trabajos. Para generar una migración que cree la tabla, es necesario ejecutar el comando de Artisan `queue:table`. Una vez creada la migración, ya puede migrar la base de datos usando el comando `migrate`:

```
php artisan queue:table
```

```
php artisan migrate
```

Redis

Para usar el *queue driver* de `redis`, debe configurar la conexión a la base de datos de Redis en el fichero de configuración `config/database.php`.

Si la *queue connection* de Redis usa *Redis Cluster*, el nombre de la cola debe contener key. hash tag. Esto es necesario para garantizar que todas las *Redis keys* para una cola determinada se coloquen en el mismo *hash slot*:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

Otros Pre requisitos

Las siguientes son dependencias necesarias para los *drivers* mencionados:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~3.0`
- Redis: `predis/predis ~1.0`

Creando los trabajos(Jobs)

Generando las Clases Trabajo

De forma predeterminada, todos los trabajos encolables en su aplicación se almacenan en el directorio `app/Jobs`. Si el directorio `app/Jobs` no existe, se creará cuando se ejecute el comando de *Artisan* `make:job`. Puede generar un nuevo trabajo en cola utilizando la CLI de *Artisan*:

```
php artisan make:job ProcessPodcast
```

La clase generada implementara la interfaz `Illuminate\Contracts\Queue\ShouldQueue`, lo que indicará a Laravel que el trabajo debe colocarse en la cola para ejecutarse de forma asíncrona.

Estructura de la Clase

Las *job classes* son muy simples, normalmente contienen el método `handle`, el cual es llamado cuando el trabajo es procesado por la cola. Para comenzar, echemos un vistazo a una clase de trabajo (*job class*) de ejemplo. En este ejemplo, pretendemos que administramos un servicio de publicación de podcasts y necesitamos procesar los archivos de podcasts subidos antes de que se publiquen:

```
<?php

namespace App\Jobs;

use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;

class ProcessPodcast implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param AudioProcessor $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }
}
```

En este ejemplo se nota que es posible enviar un [Eloquent model](#) directamente al constructor del trabajo. Debido al *trait* [SerializesModels](#) que la *job class* usa, los modelos Eloquent se serializan y deserializan elegantemente cuando el trabajo se está procesando. Si el trabajo en la cola acepta un modelo Eloquent en su constructor, únicamente el identificador del modelo sera serializado en la cola. Cuando el trabajo se ejecute, el sistema de cola buscará automáticamente en la base de datos la instancia completa del modelo. Todo es totalmente transparente para la aplicación y previene inconvenientes que pueden producirse al serializar instancias completas de modelos.

El método `handle` se llama cuando la cola procesa el trabajo. Es de tener en cuenta que es posible determinar el tipo de las dependencias en el método `handle` del trabajo. El [service container](#) de Laravel automáticamente inyecta estas dependencias.

Los datos binarios, como los contenidos de imágenes sin formato, se deben pasar a través de la función `base64_encode` antes de pasarlo a un trabajo en cola. De lo contrario, es posible que el trabajo no se serialice correctamente en formato JSON cuando se coloca en la cola.

Enviar trabajos a la cola

Una vez que haya escrito su *job class*, puede enviarla utilizando el método `dispatch` de la misma clase. Los argumentos pasados al método `dispatch` se le darán al constructor de la clase *job*:

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast);
    }
}
```

Retrasar el envío

Si desea retrasar la ejecución de un trabajo en cola, puede usar el método `delay` al enviar un trabajo. Por ejemplo, especifiquemos que un trabajo no debería estar disponible para su procesamiento hasta 10 minutos después de su envío:

```

<?php

namespace App\Http\Controllers;

use Carbon\Carbon;
use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast)
            ->delay(Carbon::now()->addMinutes(10));
    }
}

```

El servicio de cola de Amazon SQS permite un retraso máximo de 15 minutos.

Encadenado de trabajos

La cadena de trabajos permite especificar una lista de los trabajos en cola que deben ejecutarse en secuencia. Si falla un trabajo de la secuencia, el resto de los trabajos no se ejecutarán. Para ejecutar una cadena de trabajos en cola, puede usar el método `withChain` en cualquiera de sus trabajos distribuibles:

```

ProcessPodcast::withChain([
    new OptimizePodcast,
    new ReleasePodcast
])->dispatch();

```

Customizing The Queue & Connection

Enviar trabajo a una cola determinada

Al enviar trabajos a diferentes colas, puede "categorizar" sus trabajos de cola e incluso priorizar la cantidad de procesadores que asigna a varias colas. Tenga en cuenta que esto no envía los trabajos a diferentes "conexiones" de cola definidas por su archivo de configuración de cola, sino solo a colas específicas dentro de una única conexión. Para especificar la cola, se usa el método `onQueue` al enviar el trabajo:

```
<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onQueue('processing');
    }
}
```

Enviar trabajo a una conexión determinada

Si está trabajando con múltiples conexiones de colas, puede especificar a cual conexión envía el trabajo. Para especificar la conexión, use el método `onConnection` cuando envié el trabajo:

```

<?php

namespace App\Http\Controllers;

use App\Jobs\ProcessPodcast;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class PodcastController extends Controller
{
    /**
     * Store a new podcast.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Create podcast...

        ProcessPodcast::dispatch($podcast)->onConnection('sqS');
    }
}

```

Puede encadenar los métodos `onConnection` y `onQueue` para especificar la conexión y la cola para el trabajo:

```

ProcessPodcast::dispatch($podcast)
    ->onConnection('sqS')
    ->onQueue('processing');

```

Especificar mayor cantidad de intentos de trabajo / tiempo de espera

Mayor cantidad de intentos

Un enfoque para especificar el número máximo de veces que se puede intentar un trabajo es a través del interruptor `--tries` en la línea de comando de Artisan:

```
php artisan queue:work --tries=3
```

Sin embargo, puede tomar un enfoque más específico al definir la cantidad máxima de intentos en la *job class*. Si se especifica la cantidad máxima de intentos en *job class*, tendrá prioridad sobre el valor proporcionado en la línea de comando:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of times the job may be attempted.
     *
     * @var int
     */
    public $tries = 5;
}
```

Intentos basados en tiempo

Como alternativa para definir cuántos intentos debe hacer un trabajo antes de que falle, puede definir un tiempo en la que el trabajo debe expirar. Esto permite que un trabajo se intente varias veces en un tiempo determinado. Para definir la hora a la que un trabajo debe expirar, agregue un método `retryUntil` a su clase de trabajo:

```
/**
 * Determine the time at which the job should timeout.
 *
 * @return \DateTime
 */
public function retryUntil()
{
    return now()->addSeconds(5);
}
```

También puede definir un método `retryUntil` en sus *queued event listeners*.

Tiempo límite

La característica `timeout` está optimizada para PHP 7.1+ y la extensión de PHP `pcntl`.

Del mismo modo, la cantidad máxima de segundos que se pueden ejecutar trabajos se puede especificar utilizando `--timeout` en la línea de comandos de Artisan:

```
php artisan queue:work --timeout=30
```

Sin embargo, también puede definir en la *job class* la cantidad máxima de segundos que se debe permitir que un trabajo se ejecute. Si el tiempo de espera se especifica en la *job class*, tendrá prioridad sobre cualquier tiempo de espera especificado en la línea de comando:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}
```

Límites de velocidad

Esta función requiere que su aplicación pueda interactuar con un Redis server.

Si su aplicación interactúa con Redis, pondría limitar sus trabajos en cola por tiempo o concurrencia. Esta función puede ser útil cuando tus trabajos en cola interactúen con API que también tienen un límite de velocidad. Por ejemplo, con el método `throttle`, puede limitar un tipo de trabajo determinado para que se ejecute solo 10 veces cada 60 segundos. Si no se puede obtener un bloqueo, normalmente debe volver a liberar el trabajo en la cola para que pueda volver a intentarlo más adelante:

```
Redis:::throttle('key')->allow(10)->every(60)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

    return $this->release(10);
});
```

En el ejemplo anterior, `key` puede ser cualquier cadena que identifique de forma única el tipo de trabajo al que le gustaría calificar el límite. Por ejemplo, es posible que desee construir la `key` según el nombre de la clase del trabajo y los ID de los modelos Eloquent en los que opera.

Alternativamente, puede especificar la cantidad máxima de procesadores que pueden tratar simultáneamente un trabajo determinado. Esto puede ser útil cuando un trabajo en cola está modificando un recurso que solo debería ser modificado por un trabajo a la vez. Por ejemplo, con el método `funnel`, puede limitar trabajos de un tipo determinado para que solo los trate un procesador a la vez:

```
Redis:::funnel('key')->limit(1)->then(function () {
    // Job logic...
}, function () {
    // Could not obtain lock...

    return $this->release(10);
});
```

Al usar la limitación de velocidad, es difícil determinar la cantidad de intentos que su trabajo necesitará para ejecutar con éxito. Por lo tanto, es útil combinar la limitación de velocidad con time based attempts.

Manejo de Errores

Si se lanza una excepción mientras se está procesando el trabajo, el trabajo se volverá a liberar automáticamente en la cola para que se pueda intentar nuevamente. El trabajo continuará siendo liberado hasta que se haya intentado la cantidad máxima de veces permitida por su aplicación. El número máximo de intentos se define mediante `--tries`

utilizado en el comando `queue:work` de Artisan. Alternativamente, la cantidad máxima de intentos puede definirse en la *job class*. Más información sobre cómo ejecutar el procesador de cola [en siguientes secciones](#).

Ejecutar el Procesador de cola

Laravel incluye un procesador de cola que tratará los nuevos trabajos a medida que se colocan en la cola. Puede ejecutar el procesador utilizando el comando de Artisan `queue: work`. Tenga en cuenta que una vez que el comando `queue: work` haya comenzado, continuará ejecutándose hasta que se detenga manualmente o cierre su terminal:

```
php artisan queue:work
```

Para mantener el proceso `queue: work` ejecutándose permanentemente en segundo plano, debe usar un supervisor de procesos como Supervisor para asegurarse de que el procesador de cola no deje de ejecutarse.

Recuerde, los procesadores de cola son procesos de larga duración y almacenan el estado de inicio de la aplicación en memoria. Como resultado, no notarán cambios en su código una vez que se hayan iniciado. Por lo tanto, durante el proceso de implementación, asegúrese de [restart your queue workers](#).

Processing A Single Job

La opción `--once` se puede usar para indicar al procesador de cola que solo procese un único trabajo de la cola:

```
php artisan queue:work --once
```

Specifying The Connection & Queue

También puede especificar qué conexión de cola debe utilizar el procesador de cola. El nombre de conexión pasado al comando `work` debe corresponder a una de las conexiones definidas en su archivo de configuración `config/queue.php`:

```
php artisan queue:work redis
```

Puede personalizar aún más su procesadores de cola para que trate colas específicas para una conexión determinada. Por ejemplo, si todos sus correos electrónicos se procesan en la cola `emails` de la conexión de cola `redis`, puede ejecutar el siguiente comando para iniciar un procesador que solo trate esa cola:

```
php artisan queue:work redis --queue=emails
```

Resource Considerations

El *Daemon* de los procesadores de cola, no reinicia el *framework* antes de procesar cada trabajo. Por lo tanto, debe liberar todos los recursos pesados después de cada trabajo. Por ejemplo, si está manipulando imágenes con la biblioteca GD, debe liberar la memoria con `imagedestroy` cuando haya terminado.

Prioridades De Colas

En ocasiones, es posible que desee priorizar cómo se procesan sus colas. Por ejemplo, en su `config/queue.php` puede establecer la cola por defecto `queue` para su conexión a `redis` con prioridad `low` (baja). Sin embargo, de vez en cuando es posible que desee insertar un trabajo en una cola de prioridad `high` (alta) como la siguiente:

```
dispatch((new Job)->onQueue('high'));
```

Para iniciar un procesador de cola que verifique que todas las tareas de cola de prioridad `high` se procesen antes de continuar con cualquier trabajo en la cola `low`, pase una lista delimitada por comas de nombres de cola al comando `work`:

```
php artisan queue:work --queue=high,low
```

Queue Workers & Deployment

Como los procesadores de cola son procesos de larga duración, no recibirán cambios en su código sin reiniciarse. Por lo tanto, la forma más sencilla de implementar una aplicación utilizando los procesadores de cola es reiniciar los procesadores durante el proceso de implementación. Puede reiniciar fácilmente todos los procesadores usando el comando `queue:restart`:

```
php artisan queue:restart
```

Este comando indicará a todos los procesadores de cola que "mueran" después de que terminen de procesar su trabajo actual para que no se pierdan trabajos existentes. Como los procesadores de cola finalizarán cuando se ejecuta el comando `queue:restart`, debe ejecutar un administrador de procesos como [Supervisor](#) para reiniciar automáticamente los procesadores de cola.

La cola usa la caché para almacenar las señales de reinicio, así que antes de utilizar esta característica, debería verificar que el *driver* de la caché está correctamente configurado para tu aplicación.

Job Expirations & Timeouts

Job Expiration

En su archivo de configuración `config/queue.php`, cada conexión de cola define una opción `retry_after`. Esta opción especifica cuántos segundos debe esperar la conexión de cola antes de reintentar un trabajo que se está procesando. Por ejemplo, si el valor de `retry_after` se establece en `90`, el trabajo se devolverá a la cola si se ha procesado durante 90 segundos sin haber sido eliminado. Por lo general, debe establecer el valor `retry_after` en la cantidad máxima de segundos que sus trabajos deben razonablemente tomar para completar el procesamiento.

La única conexión de cola que no contiene un valor `retry_after` es Amazon SQS. SQS volverá a intentar el trabajo según el Default Visibility Timeout que se administra en la consola de AWS.

Worker Timeouts

El comando de Artisan `queue:work` expone una opción `--timeout`. La opción `--timeout` especifica cuánto esperará el proceso maestro de cola de Laravel antes de matar un procesador de cola hijo que está tratando un trabajo. En ocasiones, un proceso de cola hijo puede "congelarse" por varias razones, como una llamada HTTP externa que no responde. La opción `--timeout` elimina los procesos inmovilizados que han excedido el límite de tiempo especificado:

```
php artisan queue:work --timeout=60
```

La opción de configuración `retry_after` y la opción CLI `--timeout` son diferentes, pero funcionan juntas para garantizar que no se pierdan trabajos y que los trabajos solo se procesen satisfactoriamente una vez.

El valor `--timeout` siempre debe ser al menos varios segundos más corto que su valor de configuración `retry_after`. Esto asegurará que un procesador de cola que procesa un trabajo determinado siempre muera antes de volver a intentarlo. Si su opción `--timeout` es mayor que el valor de configuración `retry_after`, sus trabajos pueden procesarse dos veces.

Worker Sleep Duration

Cuando hay trabajos disponibles en la cola, el procesador de cola seguirá procesando trabajos sin demora entre ellos. Sin embargo, la opción `sleep` determina cuánto tiempo "dormirá" el procesador si no hay nuevos trabajos disponibles. Mientras duerme, el procesador de cola no procesará ningún trabajo nuevo; los trabajos se procesarán después de que el procesador se despierte nuevamente.

```
php artisan queue:work --sleep=3
```

Configuración de Supervisor

Installing Supervisor

Supervisor es un monitor de procesos para el sistema operativo Linux y reiniciará automáticamente su proceso `queue:work` si falla. Para instalar Supervisor en Ubuntu, se puede usar el siguiente comando:

```
sudo apt-get install supervisor
```

Si configurar el supervisor por ud mismo suena abrumador, considere usar Laravel Forge, que instalará y configurará automáticamente el supervisor para sus proyectos de Laravel.

Configuring Supervisor

Los archivos de configuración de Supervisor típicamente se almacenan en el directorio `/etc/supervisor/conf.d`. Dentro de este directorio, se pueden crear cualquier número de archivos de configuración que le indiquen a Supervisor cómo deberían ser monitoreados los procesos. Por ejemplo, creamos el archivo `laravel-worker.conf` que inicia y monitorea un proceso `queue:work`:

```
[program:laravel-worker]
process_name=%(program_name)s_%(process_num)02d
command=php /home/forge/app.com/artisan queue:work sqs --sleep=3 --tries=3
autostart=true
autorestart=true
user=forge
numprocs=8
redirect_stderr=true
stdout_logfile=/home/forge/app.com/worker.log
```

En este ejemplo, la directiva `numprocs` instruye a Supervisor para que ejecute 8 procesos `queue:work` y los monitoree a todos, reiniciándolos si alguno falla. Por supuesto, debe cambiar la sección `queue:work sqs` de la instrucción `command` para reflejar su conexión de cola deseada.

Starting Supervisor

Una vez que se ha creado el archivo de configuración, se puede actualizar la configuración de Supervisor e iniciar los procesos usando los siguientes comandos:

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start laravel-worker:*
```

Para obtener más información sobre Supervisor, consulte la [Supervisor documentation](#).

Gestión de Trabajos Fallidos

Algunas veces sus trabajos en cola fallarán. ¡No te preocupes, las cosas no siempre salen según lo planeado! Laravel incluye una forma conveniente de especificar la cantidad máxima de intentos de trabajo. Después de que un trabajo haya excedido esta cantidad de intentos, se insertará en la tabla de la base de datos `failed_jobs`. Para crear una migración para la tabla `failed_jobs`, puede usar el comando `queue:failed-table`:

```
php artisan queue:failed-table  
  
php artisan migrate
```

Luego, al ejecutar el `queue worker`, debe especificar la cantidad máxima de veces que se debe intentar un trabajo utilizando `--tries` en el comando `queue:work`. Si no especifica un valor para la opción `--tries`, los trabajos se intentarán indefinidamente:

```
php artisan queue:work redis --tries=3
```

Cleaning Up After Failed Jobs

Puede definir un método `failed` directamente en su clase de trabajo, lo que le permite realizar tareas de limpieza específicas cuando ocurre una falla. Esta es la ubicación perfecta para enviar una alerta a sus usuarios o revertir cualquier acción realizada por el trabajo. La `Exception` que provocó la falla del trabajo se pasará al método `failed`:

```
<?php

namespace App\Jobs;

use Exception;
use App\Podcast;
use App\AudioProcessor;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class ProcessPodcast implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    protected $podcast;

    /**
     * Create a new job instance.
     *
     * @param Podcast $podcast
     * @return void
     */
    public function __construct(Podcast $podcast)
    {
        $this->podcast = $podcast;
    }

    /**
     * Execute the job.
     *
     * @param AudioProcessor $processor
     * @return void
     */
    public function handle(AudioProcessor $processor)
    {
        // Process uploaded podcast...
    }

    /**
     * The job failed to process.
     */
}
```

```
*  
* @param Exception $exception  
* @return void  
*/  
public function failed(Exception $exception)  
{  
    // Send user notification of failure, etc...  
}  
}
```

Eventos de Trabajos Fallidos

Si desea registrar un evento que será invocado cuando falla un trabajo, puede usar el método `Queue::failing`. Este evento es una gran oportunidad para notificar a su equipo por correo electrónico o por [HipChat](#). Por ejemplo, es posible adjuntar un *callback* a este evento desde el `AppServiceProvider` incluido con Laravel:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Queue\Events\JobFailed;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::failing(function (JobFailed $event) {
            // $event->connectionName
            // $event->job
            // $event->exception
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

Reintentar Trabajos Fallidos

Para ver todos sus trabajos fallidos que se han insertado en su tabla de base de datos `failed_jobs`, puede usar el comando de Artisan `queue:failed`:

```
php artisan queue:failed
```

El comando `queue:failed` mostrará la identificación del trabajo, la conexión, la cola y el tiempo de falla. El ID del trabajo se puede usar para volver a intentar el trabajo fallido. Por ejemplo, para volver a intentar un trabajo fallido que tiene una ID de `5`, ejecute el siguiente comando:

```
php artisan queue:retry 5
```

Para volver a intentar todos sus trabajos fallidos, ejecute el comando `queue:retry` y pase `all` como ID:

```
php artisan queue:retry all
```

Si desea eliminar un trabajo fallido, puede usar el comando `queue:forget`:

```
php artisan queue:forget 5
```

Para eliminar todos sus trabajos fallidos, puede usar el comando `queue:flush`:

```
php artisan queue:flush
```

Eventos de Tabajos

Utilizando los métodos `before` y `after` en la Queue [facade](#), puede especificar los *callbacks* para que se ejecuten antes o después de una trabajo en cola es procesado. Estos *callbacks* son una gran oportunidad para realizar estadísticas de incremento o registro adicionales para un *dashboard*. Por lo general, debe llamar a estos métodos desde un [service provider](#). Por ejemplo, podemos usar el [AppServiceProvider](#) que se incluye con Laravel:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Queue;
use Illuminate\Support\ServiceProvider;
use Illuminate\Queue\Events\JobProcessed;
use Illuminate\Queue\Events\JobProcessing;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Queue::before(function (JobProcessing $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });

        Queue::after(function (JobProcessed $event) {
            // $event->connectionName
            // $event->job
            // $event->job->payload()
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Al utilizar el método `looping` en la `Queue facade</1`, puede especificar los *callbacks* que se ejecutan antes de que el procesador de cola intente recuperar un trabajo de una cola. Por ejemplo, puede registrar un *Closure* para deshacer cualquier transacción que haya quedado abierta por un trabajo que falló:

```
Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```

Programación de Tareas – *Task Scheduling*

Introducción

Definición de tiempos

- # Programación de comandos de Artisan
- # Programación de colas de trabajo
- # Programación de comandos del terminal
- # Opciones de frecuencia de la programación
- # Prevención de solapamientos de tareas
- # Modo de mantenimiento

Salida generada

Hooks de tareas

Introducción

En el pasado, es posible que haya generado una entrada Cron para cada tarea que necesite programar en su servidor. Sin embargo, esto puede convertirse rápidamente en una molestia, porque su programador de tareas ya no está en su origen y debe usar SSH en su servidor para agregar entradas adicionales de Cron.

El programador de comandos de Laravel le permite definir con fluidez y expresividad su programación de comandos dentro del propio Laravel. Cuando utilice el programador, sólo se necesita una única entrada Cron en su servidor. La tarea programada se define en el método `schedule` en el archivo `app\Console\Kernel.php`. Para ayudarle a empezar, se incluye un ejemplo sencillo con el método.

Iniciando el programador

Cuando utilice el programador, sólo necesita añadir la siguiente entrada Cron a su servidor. Si no sabe cómo agregar entradas Cron a su servidor, considere utilizar un servicio como [Laravel Forge](#) que puede administrar las entradas Cron por usted:

```
* * * * * php /path-to-your-project/artisan schedule:run >> /dev/null 2>&1
```

Este Cron llamará al programador de Laravel cada minuto. Cuando se ejecuta el comando `schedule:run` , Laravel evaluará las tareas programadas y ejecutará las que se deben ejecutar.

Definición de tiempos

Puede definir todas las tareas en el método de `schedule` de la clase `App\Console\Kernel` . Para comenzar, veamos un ejemplo de coordinar una tarea. En este ejemplo, coordinamos un `Closure` que se llamará todos los días a medianoche. Dentro del `Closure` ejecutaremos una consulta de base de datos para borrar una tabla:

```

<?php

namespace App\Console;

use DB;
use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        \App\Console\Commands\Inspire::class,
    ];

    /**
     * Define the application's command schedule.
     *
     * @param \Illuminate\Console\Scheduling\Schedule $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {
        $schedule->call(function () {
            DB::table('recent_users')->delete();
        })->daily();
    }
}

```

Programación de comandos de Artisan

Además de programar *Closures*, también puede programar [comandos de Artisan](#) y comandos del sistema operativo. Por ejemplo, puede usar el método `command` para programar un comando Artisan usando el nombre o la clase del comando:

```
$schedule->command('emails:send --force')->daily();  
  
$schedule->command(EmailsCommand::class, ['--force'])->daily();
```

Programación de colas de trabajo

El método `job` se puede utilizar para programar un [trabajo en cola \(queued job\)](#). Este método proporciona una manera conveniente de programar trabajos sin usar el método `call` para crear cierres manuales al colocarlo en la cola:

```
$schedule->job(new Heartbeat)->everyFiveMinutes();
```

Programación de comandos del terminal

El método `exec` puede usarse para enviar una orden al sistema operativo:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

Opciones de frecuencia de la programación

Por supuesto, hay una variedad de horarios que usted puede asignar a su tarea:

Método	Descripción
<code>->cron('* * * * *');</code>	Ejecutar la tarea en un cronograma personalizado de Cron
<code>->everyMinute();</code>	Ejecutar la tarea cada minuto
<code>->everyFiveMinutes();</code>	Ejecutar la tarea cada cinco minutos
<code>->everyTenMinutes();</code>	Ejecutar la tarea cada diez minutos
<code>->everyFifteenMinutes();</code>	Ejecutar la tarea cada quince minutos
<code>->everyThirtyMinutes();</code>	Ejecutar la tarea cada treinta minutos

Método	Descripción
<code>->hourly();</code>	Ejecutar la tarea cada hora
<code>->hourlyAt(17);</code>	Ejecutar la tarea cada hora a los 17 minutos después de la hora en punto
<code>->daily();</code>	Ejecutar la tarea todos los días a medianoche
<code>->dailyAt('13:00');</code>	Ejecutar la tarea todos los días a las 13:00
<code>->twiceDaily(1, 13);</code>	Ejecutar la tarea diariamente a la 1:00 y a las 13:00
<code>->weekly();</code>	Ejecutar la tarea cada semana
<code>->monthly();</code>	Ejecutar la tarea cada mes
<code>->monthlyOn(4, '15:00');</code>	Ejecutar la tarea cada mes el día 4 a las 15:00
<code>->quarterly();</code>	Ejecutar la tarea cada trimestre
<code>->yearly();</code>	Ejecutar la tarea cada año
<code>->timezone('America/New_York');</code>	Configurar la zona horaria

Estos métodos pueden combinarse con restricciones adicionales para crear programas aún más ajustados que sólo funcionan en ciertos días de la semana. Por ejemplo, para programar un comando que se ejecute semanalmente el lunes:

```
// Run once per week on Monday at 1 PM...
$schedule->call(function () {
    //
})->weekly()->mondays()->at('13:00');

// Run hourly from 8 AM to 5 PM on weekdays...
$schedule->command('foo')
    ->weekdays()
    ->hourly()
    ->timezone('America/Chicago')
    ->between('8:00', '17:00');
```

A continuación se muestra una lista de las restricciones adicionales del horario:

Método	Descripción
->weekdays();	Limita la tarea a días de la semana
->sundays();	Limita la tarea al domingo
->mondays();	Limita la tarea al lunes
->tuesdays();	Limita la tarea al martes
->wednesdays();	Limita la tarea al miércoles
->thursdays();	Limita la tarea al jueves
->fridays();	Limita la tarea al viernes
->saturdays();	Limita la tarea al sábado
->between(\$start, \$end);	Limita la tarea a ejecutarse entre las horas de inicio y final
->when(Closure);	Limita la tarea basada en una prueba de verdad

Restricciones de tiempo

Se puede utilizar el método `between` para limitar la ejecución de una tarea basada en la hora del día:

```
$schedule->command('reminders:send')
    ->hourly()
    ->between('7:00', '22:00');
```

Del mismo modo, el método `unlessBetween` puede utilizarse para excluir la ejecución de una tarea durante un período de tiempo:

```
$schedule->command('reminders:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

Restricciones con pruebas de verdad

El método `when` puede utilizarse para limitar la ejecución de una tarea basada en el resultado de una prueba de verdad dada. En otras palabras, si el `Closure` devuelve `true`, la tarea se ejecutará siempre y cuando ninguna otra condición de restricción impida que la tarea se ejecute:

```
$schedule->command('emails:send')->daily()->when(function () {
    return true;
});
```

El método `skip` puede verse como el inverso de `when`. Si el método `skip` devuelve `true`, la tarea programada no se ejecutará:

```
$schedule->command('emails:send')->daily()->skip(function () {
    return true;
});
```

Cuando se utilizan métodos encadenados `when`, el comando programado sólo se ejecutará si todos devuelven `true`.

Prevención de solapamientos de tareas

De forma predeterminada, las tareas programadas se ejecutarán aunque la instancia anterior de la tarea siga en ejecución. Para evitarlo, puede utilizar el método `withoutOverlapping`:

```
$schedule->command('emails:send')->withoutOverlapping();
```

En este ejemplo, el [comando Artisan `emails:send`](#) se ejecutará cada minuto si no está funcionando. El método `withoutOverlapping` es especialmente útil si tiene tareas que varían drásticamente en su tiempo de ejecución, impidiéndole predecir exactamente cuánto tiempo tomará una tarea dada.

Modo de mantenimiento

Las tareas programadas de Laravel no se ejecutarán cuando Laravel está en [modo mantenimiento](#), ya que no es deseable que las tareas interfieran con cualquier mantenimiento inacabado que pueda estar realizándose en el servidor. Sin embargo, si desea

forzar una tarea para que se ejecute incluso en modo de mantenimiento, puede utilizar el método `evenInMaintenanceMode` :

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

Salida generada

El programador de Laravel proporciona varios métodos convenientes para trabajar con la salida generada por las tareas programadas. En primer lugar, utilizando el método `sendOutputTo`, puede enviar la salida a un fichero para su posterior inspección:

```
$schedule->command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

Si desea añadir la salida a un archivo determinado, puede utilizar el método `appendOutputTo`:

```
$schedule->command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

Utilizando el método `emailOutputTo`, puede enviar la salida a una dirección de correo electrónico de su elección. Antes de enviar por correo electrónico la salida de una tarea, debe configurar los [servicios de correo electrónico](#) de Laravel:

```
$schedule->command('foo')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('foo@example.com');
```

Los métodos `emailOutputTo`, `sendOutputTo` y `appendOutputTo` son exclusivos del método `command` y no son compatibles con `call`.

Hooks de tareas

Utilizando los métodos `before` y `after`, puede especificar el código a ejecutar antes y después de que la tarea programada haya finalizado:

```
$schedule->command('emails:send')
    ->daily()
    ->before(function () {
        // Task is about to start...
    })
    ->after(function () {
        // Task is complete...
    });
});
```

Ping a URLs

Utilizando los métodos `pingBefore` y `thenPing`, el programador puede hacer *ping* automáticamente a una URL determinada antes o después de que se complete una tarea. Este método es útil para notificar a un servicio externo, como [Laravel Envoyer](#), que su tarea programada está comenzando o ha finalizado:

```
$schedule->command('emails:send')
    ->daily()
    ->pingBefore($url)
    ->thenPing($url);
```

El uso de la característica `pingBefore($url)` o `thenPing($url)` requiere la biblioteca Guzzle HTTP. Puedes añadir Guzzle a tu proyecto usando el gestor de paquetes Composer:

```
composer require guzzlehttp/guzzle
```

Base de datos

Bases de datos: Comienzo

Introducción

- # Configuración
- # Conexiones lectura & escritura
- # Usando varias conexiones de base de datos

Ejecutar consultas en SQL

- # Capturar eventos de consultas

Transacciones de la base de datos

Introducción

Laravel realiza interacciones con bases de datos y ejecuta consultas de forma muy simple a través de varias configuraciones de base de datos utilizando SQL puro, el [generador de consultas Fluent](#) y el [ORM Eloquent](#). Actualmente, Laravel soporta cuatro sistemas de bases de datos:

- MySQL
- PostgreSQL
- SQLite
- SQL Server

Configuración

La configuración de la base de datos para la aplicación se encuentra en [config/database.php](#) . En este archivo se pueden definir todas las conexiones de base de datos, así como especificar qué conexión se debe utilizar por defecto. Este archivo proporciona ejemplos para la mayoría los sistemas de base de datos soportados.

Por defecto, la [configuración de entorno](#) de ejemplo de Laravel está preparada para ser usada con [Laravel Homestead](#), que es una máquina virtual adecuada para desarrollar con Laravel en un equipo en local. Por supuesto, hay libertad para modificar esta configuración cuando sea necesario para una base de datos en local.

Configuración de SQLite

Después de crear una nueva base de datos SQLite usando un comando como `touch database/database.sqlite`, se puede configurar fácilmente las variables de entorno para que apunten a esta base de datos recién creada utilizando la ruta absoluta de la base de datos:

```
DB_CONNECTION=sqlite  
DB_DATABASE=/absolute/path/to/database.sqlite
```

Conexiones lectura & escritura

A veces se puede usar una conexión a una base de datos para las sentencias SELECT y otra conexión para las sentencias INSERT, UPDATE y DELETE. Laravel hace las cosas sencillas, y se usarán las conexiones adecuadas tanto si se utilizan consultas en SQL puro, el generador de consultas o el ORM Eloquent.

En el siguiente ejemplo, se muestra como deberían estar configuradas las conexiones de lectura / escritura:

```
'mysql' => [  
    'read' => [  
        'host' => '192.168.1.1',  
    ],  
    'write' => [  
        'host' => '196.168.1.2'  
    ],  
    'sticky' => true,  
    'driver' => 'mysql',  
    'database' => 'database',  
    'username' => 'root',  
    'password' => '',  
    'charset' => 'utf8mb4',  
    'collation' => 'utf8mb4_unicode_ci',  
    'prefix' => ''  
],
```

Hay que observar que se han añadido tres claves a la configuración del array: `read`, `write` y `sticky`. Las claves `read` y `write` contienen un array con una sola clave: `host`. El resto de las opciones de base de datos para las conexiones de `read` y `write` se mezclarán desde el array principal `mysql`.

De esta forma, solo se tienen que colocar elementos en `read` y `write` si se desea sobrescribir los valores del *array* principal. Por lo tanto, en este caso, `192.168.1.1` será utilizado para la conexión "read" mientras que `192.168.1.2` será utilizado para la conexión "write". Las credenciales, prefijos, configuración de caracteres y el resto de las opciones del *array* principal `mysql` serán compartidas por ambas conexiones.

La opción `sticky`

La opción `sticky` es un valor *optional* que puede ser usado para permitir la lectura inmediata de grabaciones que han sido escritas en la base de datos durante el ciclo de solicitud actual. Si la opción `sticky` está habilitada y se ha realizado una operación de "escritura" contra la base de datos durante el ciclo de solicitud actual, cualquier operación adicional de "lectura" utilizará la conexión de "escritura". Esto garantiza que cualquier dato escrito durante el ciclo de solicitud pueda leerse inmediatamente desde la base de datos durante la misma solicitud. Depende de usted decidir si este es el comportamiento deseado para su aplicación.

Usando múltiples conexiones de base de datos

Cuando se usan múltiples conexiones, se puede acceder a cada conexión con el método `connection` de la fachada `DB`. El `nombre` pasado al método `connection` debería corresponderse con uno de las conexiones listadas en el archivo de configuración `config/database.php`:

```
$users = DB::connection('foo')->select(...);
```

Se puede acceder a la instancia PDO subyacente utilizando el método `getPdo` en la instancia de una conexión:

```
$pdo = DB::connection()->getPdo();
```

Ejecutar Consultas en SQL Puro

Una vez configurada la conexión a la base de datos, se pueden ejecutar consultas usando la *facade* `DB`. La *facade* `DB` proporciona métodos para cada tipo de consulta: `select` , `update` , `insert` , `delete` , y `statement` .

Ejecutando Una Consulta Select

Para ejecutar una consulta básica, se puede utilizar el método `select` de la facade `DB`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active = ?', [1]);

        return view('user.index', ['users' => $users]);
    }
}
```

El primer argumento pasado al método `select` es la consulta en SQL puro, mientras que el segundo argumento son los enlaces a los parámetros considerados obligatorios en la consulta. En general, estos son los valores de las restricciones de la cláusula `where`. Los parámetros enlazados proporcionan protección contra la inyección SQL.

El método `select` siempre devolverá un `array` de resultados. Cada resultado dentro del `array` será un objeto de PHP `StdClass`, que permitirá acceder a los valores de los resultados:

```
foreach ($users as $user) {
    echo $user->name;
}
```

Uso de Enlaces con Nombre

En vez de usar `?` para representar enlaces a los parámetros, se puede utilizar una consulta utilizando enlaces con nombres:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Ejecución de una Sentencia de Inserción

Para ejecutar una sentencia `insert`, se debe utilizar el método `insert` de la `facade DB`.

Como en el `select`, este método tiene la consulta en SQL puro como primer parámetro y los enlaces como segundo argumento:

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Ejecución de una Sentencia Update

El método `update` se utiliza para actualizar los registros existentes en la base de datos. El método devolverá el número de filas afectadas por la sentencia:

```
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

Ejecución de una Sentencia Delete

El método `delete` se debería usar para borrar registros de la base de datos. Al igual que `update`, devolverá el número de filas eliminadas:

```
$deleted = DB::delete('delete from users');
```

Ejecución de una Sentencia General

Algunas sentencias de bases de datos no devuelven ningún valor. Para este tipo de operaciones, se debe utilizar el método `statement` de la `facade DB`:

```
DB::statement('drop table users');
```

Capturar Eventos de Consultas

Si se desea recibir cada consulta SQL que es ejecutada en la aplicación, se debe usar el método `listen`. Este método es útil para realizar el registro o la depuración de consultas. Se puede registrar un capturador de consultas en un [service provider](#):

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function ($query) {
            // $query->sql
            // $query->bindings
            // $query->time
        });
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

Transacciones de la base de datos

Se puede usar el método `transaction` en la facde `DB` para ejecutar un conjunto de operaciones dentro de una transacción de base de datos. Si se lanza una excepción dentro del `Closure` de la transacción, esta realizará automáticamente un rollback. Si el `Closure` se

ejecuta de forma satisfactoria, la transacción realizará automáticamente un commit. No hay que preocuparse de hacer manualmente rollbacks o commits mientras se utilice el método `transaction` :

```
DB::transaction(function () {  
    DB::table('users')->update(['votes' => 1]);  
  
    DB::table('posts')->delete();  
});
```

Manejo de interbloqueos

El método `transaction` acepta un segundo argumento opcional que define el número de veces que una transacción debe ser re-intentada cuando ocurre un interbloqueo. Una vez que estos intentos se hayan agotado, se lanzará una excepción:

```
DB::transaction(function () {  
    DB::table('users')->update(['votes' => 1]);  
  
    DB::table('posts')->delete();  
}, 5);
```

Usar Transacciones Manualmente

Si se desea comenzar una transacción manual y tener el control sobre rollbacks y commits, se debe utilizar el método `beginTransaction` de la facade `DB` :

```
DB::beginTransaction();
```

Se puede hacer rollback en la transacción utilizando el método `rollBack` :

```
DB::rollBack();
```

Por último, se puede hacer un commit de la transacción utilizando el método `commit` :

```
DB::commit();
```

Los métodos de transacción del *facade* `DB` controlan las transacciones para el query builder y Eloquent ORM.

Base de datos: *Query Builder*

- # Introducción
- # Obtener resultados
 - # Fragmentar resultados
 - # Funciones agregadas
- # Selecciones (Select)
- # Raw Expressions
- # Joins
- # Unions
- # Cláusulas *where*
 - # Agrupar parámetros
 - # Sentencias *where exists*
 - # Sentencias JSON *where*
- # Ordenar, agrupar, limitar & offset
- # Cláusulas condicionales
- # Inserts
- # Updates
 - # Actualizar columnas JSON
 - # Incrementar & decrementar
- # Deletes
- # Bloqueos persistentes

Introducción

El constructor de consultas o *query builder* de Laravel es una interfaz conveniente y fluida para ejecutar consultas de base de datos. Este puede usarse para realizar la mayoría de las operaciones de bases de datos de la aplicación y funciona para todos los sistemas de bases de datos soportados.

El generador de consultas de Laravel utiliza el enlace de parámetros de PDO para proteger la aplicación contra los ataques de inyección de SQL. No es necesario limpiar las cadenas que se pasan como enlaces.

Obtener resultados

Obtener todas las filas de una tabla

Puede usar el método `table` de la facade `DB` para comenzar una consulta. El método `table` devuelve una instancia del generador de consultas fluida para la tabla dada, lo que le permite encadenar más restricciones a la consulta y finalmente obtener los resultados utilizando el método `get</0>`:

```
get();

    return view('user.index', ['users' => $users]);
}

`
```

El método `get` devuelve una `Illuminate\Support\Collection` que contiene los resultados donde cada resultado es una instancia del objeto PHP `StdClass`. Es posible acceder a cada valor de la columna mediante el acceso a la columna como una propiedad del objeto:

```
foreach ($users as $user) {
    echo $user->name;
}
```

Recuperando un Solo Registro / Columna desde una Tabla

Si solo necesita obtener una fila de una tabla de la base de datos, tu puedes utilizar el método `first`. este método devolverá un solo objeto de `StdClass` :

```
$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;
```

Si usted no necesita el valor completo de una fila, puede obtener un solo valor del registro obtenido con el método `value`. Este método le permite obtener el valor directo de dicha columna:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Recuperando una lista de valores de una columna

Si deseas recuperar una Collection que contiene los valores de una sola columna, puedes usar el método `pluck`. En este ejemplo, recuperaremos una Collection con los títulos de la tabla `roles`:

```
$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

Puedes especificar una clave de columna personalizada para la Collection devuelta:

```
$roles = DB::table('roles')->pluck('title', 'name');

foreach ($roles as $name => $title) {
    echo $title;
}
```

Fragmentar Resultados

Si tu necesitas trabajar con miles de registros de base de datos, considera utilizar el metodo `chunk`. Este método recupera una pequeña porción o bloque de resultados cada vez en un `Closure` para su procesado. Este método es muy utilizado para crear [Artisan commands](#) que procesen miles de registros. Por ejemplo, vamos a trabajar con la tabla `users` en bloques de 100 registros a la vez:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

Puedes parar el proceso de bloques cuando están siendo procesados devolviendo `false` desde la `Closure`:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {  
    // Process the records...  
  
    return false;  
});
```

Funciones agregadas

El constructor de consultas también provee una variedad de métodos agregados tales como `count` , `max` , `min` , `avg` y `sum` . Puedes llamar cualquiera de esos métodos después de construir tu consulta:

```
$users = DB::table('users')->count();  
  
$price = DB::table('orders')->max('price');
```

Por supuesto, puedes combinar esos métodos con otras cláusulas para construir su consulta:

```
$price = DB::table('orders')  
    ->where('finalized', 1)  
    ->avg('price');
```

Selecciones (Select)

Especificando una cláusula Select

Por supuesto, puede que no siempre quiera seleccionar todas las columnas de una tabla de la base de datos. Usando el método `select` , podrá especificar una cláusula a medida para la consulta:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

El método `distinct` permite forzar la consulta para retornar resultados diferentes (elimina los que son iguales):

```
$users = DB::table('users')->distinct()->get();
```

Si usted ya tiene una instancia del constructor de consultas, y desea agregar una columna a la cláusula Select existente, puede utilizar el método `addSelect` :

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Expresiones en crudo

Algunas veces necesitarás utilizar una expresión raw en una consulta. Para crear una expresión raw debes utilizar el método `DB::raw` :

```
$users = DB::table('users')
    ->select(DB::raw('count(*) as user_count, status'))
    ->where('status', '<>', 1)
    ->groupBy('status')
    ->get();
```

Los estamentos raw se inyectarán en las consultas como cadenas, por lo que debes ser extremadamente cuidadoso para no crear vulnerabilidades de inyección SQL.

Métodos raw

En lugar de utilizar `DB::raw` , puedes utilizar también los siguientes métodos para insertar expresiones raw en diversas partes de tus consultas:

`selectRaw`

El método `selectRaw` puede utilizarse en lugar de `select(DB::raw(...))` . Este método acepta un array opcional de enlaces como su segundo argumento:

```
$orders = DB::table('orders')
    ->selectRaw('price * ? as price_with_tax', [1.0825])
    ->get();
```

`whereRaw / orWhereRaw`

Los métodos `whereRaw` y `orWhereRaw` pueden utilizarse para injectar un cláusula raw `where` en tu consulta. Estos métodos aceptan un array de enlaces opcional como su segundo argumento:

```
$orders = DB::table('orders')
    ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
    ->get();
```

`havingRaw / orHavingRaw`

Los métodos `havingRaw` y `orHavingRaw` pueden utilizarse para configurar una cadena raw como el valor de la cláusula `having`:

```
$orders = DB::table('orders')
    ->select('department', DB::raw('SUM(price) as total_sales'))
    ->groupBy('department')
    ->havingRaw('SUM(price) > 2500')
    ->get();
```

`orderByRaw`

El método `orderByRaw` puede usarse para configurar una cadena raw como el valor de la cláusula `order by`:

```
$orders = DB::table('orders')
    ->orderByRaw('updated_at - created_at DESC')
    ->get();
```

Uniones

Inner Join Clause

El constructor de consultas también puede ser utilizado para escribir declaraciones de union (joins). Para realizar una consulta básica "inner join", puedes usar el método `join` en una instancia del constructor de consultas. El primer argumento pasado al método `join` es el

nombre de la tabla que necesita unir, mientras que los argumentos restantes especifican las restricciones de columnas para la unión. Por supuesto, como puede ver, puede unir múltiples tablas en una sola consulta:

```
$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

Left Join Clause

Si quisiera realizar un "left join" en vez de un "inner join", utilice el método `leftJoin`. El método `leftJoin` tiene la misma firma que el método `join`:

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

Cross Join Clause

Para realizar un "cross join" use el método `crossJoin` con el nombre de la tabla con la que quiere realizar una combinación cruzada. Las combinaciones cruzadas generan un producto cartesiano entre la primera tabla y la tabla unida:

```
$users = DB::table('sizes')
    ->crossJoin('colours')
    ->get();
```

Advanced Join Clauses

Puede también especificar cláusulas de unión más avanzadas. Para empezar pasamos una `Closure` como segundo argumento en el método `join`. La `Closure` recibirá un objeto `JoinClause` el cual le permite especificar restricciones en la clausa `join`:

```
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
    })
    ->get();
```

Si quisiera usar una cláusula estilo "where" en sus uniones, puede usar los métodos `where` y `orWhere` en una unión. En vez de comparar dos columnas, estos métodos compararán la columna contra un valor:

```
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')
            ->where('contacts.user_id', '>', 5);
    })
    ->get();
```

Fusiones (Unions)

El constructor de consultas también provee una manera rápida de "unir" dos consultas. Por ejemplo, puedes crear una consulta inicial, y entonces utilizar el método `union` para fusionarla con la segunda consulta:

```
$first = DB::table('users')
    ->whereNull('first_name');

$users = DB::table('users')
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

El método `unionAll` también está disponible y tiene la misma forma que `union`.

Cláusulas Where

Cláusulas Where Simples

Se puede usar el método `where` en una instancia del *query builder* para agregar cláusulas de `where` a la consulta. La llamada más básica a `where` requiere de tres argumentos. El primer argumento es el nombre de la columna. El segundo argumento es un operador, el cuál puede ser cualquiera de los operadores soportados por la base de datos. Finalmente, el tercer argumento es el valor a evaluar contra la columna.

Por ejemplo, esta es una consulta que verifica el valor que la columna "votes" es igual a 100:

```
$users = DB::table('users')->where('votes', '=', 100)->get();
```

Por conveniencia, si simplemente quiere verificar que la columna es igual a un valor dado, puede pasar el valor directamente como segundo parámetro al método `where`:

```
$users = DB::table('users')->where('votes', 100)->get();
```

Por supuesto, puede usar una variedad de operadores cuando escribe cláusulas `where`:

```
$users = DB::table('users')
    ->where('votes', '>=', 100)
    ->get();

$users = DB::table('users')
    ->where('votes', '<>', 100)
    ->get();

$users = DB::table('users')
    ->where('name', 'like', 'T%')
    ->get();
```

También puede pasar un array de condiciones a la función `where`:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

Declaraciones O (Or)

Puede encadenar restricciones `where`, así como agregar cláusulas `or` a la consulta. El método `orWhere` acepta los mismos argumentos que el método `where`:

```
$users = DB::table('users')
    ->where('votes', '>', 100)
    ->orWhere('name', 'John')
    ->get();
```

Agregar cláusulas *Where* adicionales

whereBetween

El método `whereBetween` verifica que el valor de la columna se encuentre entre dos valores:

```
$users = DB::table('users')
    ->whereBetween('votes', [1, 100])->get();
```

whereNotBetween

El método `whereNotBetween` verifica que el valor de la columna se encuentre fuera de los dos valores:

```
$users = DB::table('users')
    ->whereNotBetween('votes', [1, 100])
    ->get();
```

whereIn / whereNotIn

El método `whereIn` verifica que el valor de una columna dada se encuentre dentro de los valores del array:

```
$users = DB::table('users')
    ->whereIn('id', [1, 2, 3])
    ->get();
```

El método `whereNotIn` verifica que el valor de una columna dada **no** se encuentre dentro de los valores del array:

```
$users = DB::table('users')
    ->whereNotIn('id', [1, 2, 3])
    ->get();
```

whereNull / whereNotNull

El método `whereNull` verifica que el valor de la columna dada sea `NULL`:

```
$users = DB::table('users')
    ->whereNull('updated_at')
    ->get();
```

El método `whereNotNull` verifica que el valor de la columna no sea `NULL` :

```
$users = DB::table('users')
    ->whereNotNull('updated_at')
    ->get();
```

whereDate / whereMonth / whereDay / whereYear

El método `whereDate` se puede usar para comparar el valor de una columna con una fecha:

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-12-31')
    ->get();
```

El método `whereMonth` se puede usar para comparar el valor de una columna con un mes específico del año:

```
$users = DB::table('users')
    ->whereMonth('created_at', '12')
    ->get();
```

El método `whereDay` se puede usar para comparar el valor de una columna con un día específico del mes:

```
$users = DB::table('users')
    ->whereDay('created_at', '31')
    ->get();
```

El método `whereYear` se puede usar para comparar el valor de una columna con un año específico:

```
$users = DB::table('users')
    ->whereYear('created_at', '2016')
    ->get();
```

whereColumn

El método `whereColumn` se puede usar para verificar que dos columnas son iguales:

```
$users = DB::table('users')
    ->whereColumn('first_name', 'last_name')
    ->get();
```

También se puede pasar un operador de comparación al método:

```
$users = DB::table('users')
    ->whereColumn('updated_at', '>', 'created_at')
    ->get();
```

Al método `whereColumn` también se le puede pasar un array con múltiples condiciones. Estas condiciones se unirán utilizando el operador `and`:

```
$users = DB::table('users')
    ->whereColumn([
        ['first_name', '=', 'last_name'],
        ['updated_at', '>', 'created_at']
    ])->get();
```

Agrupar parámetros

A veces se puede necesitar crear cláusulas `where` más avanzadas como cláusulas "where exists" o agrupaciones de parámetros anidados. El query builder de Laravel también puede manejar estos. To get started, let's look at an example of grouping constraints within parenthesis:

```
DB::table('users')
    ->where('name', '=', 'John')
    ->orWhere(function ($query) {
        $query->where('votes', '>', 100)
            ->where('title', '!=', 'Admin');
    })
    ->get();
```

As you can see, passing a `Closure` into the `orWhere` method instructs the query builder to begin a constraint group. The `Closure` will receive a query builder instance which you can use to set the constraints that should be contained within the parenthesis group. The example above will produce the following SQL:

```
select * from users where name = 'John' or (votes > 100 and title <> 'Admin')
```

Where Exists Clauses

The `whereExists` method allows you to write `where exists` SQL clauses. The `whereExists` method accepts a `Closure` argument, which will receive a query builder instance allowing you to define the query that should be placed inside of the "exists" clause:

```
DB::table('users')
    ->whereExists(function ($query) {
        $query->select(DB::raw(1))
            ->from('orders')
            ->whereRaw('orders.user_id = users.id');
    })
    ->get();
```

The query above will produce the following SQL:

```
select * from users
where exists (
    select 1 from orders where orders.user_id = users.id
)
```

JSON Where Clauses

Laravel also supports querying JSON column types on databases that provide support for JSON column types. Currently, this includes MySQL 5.7 and PostgreSQL. To query a JSON column, use the `->` operator:

```
$users = DB::table('users')
    ->where('options->language', 'en')
    ->get();

$users = DB::table('users')
    ->where('preferences->dining->meal', 'salad')
    ->get();
```

Ordenación, Agrupamientos, Límites y Desplazamiento

orderBy

The `orderBy` method allows you to sort the result of the query by a given column. The first argument to the `orderBy` method should be the column you wish to sort by, while the second argument controls the direction of the sort and may be either `asc` or `desc`:

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->get();
```

latest / oldest

The `latest` and `oldest` methods allow you to easily order results by date. By default, result will be ordered by the `created_at` column. Or, you may pass the column name that you wish to sort by:

```
$user = DB::table('users')
    ->latest()
    ->first();
```

inRandomOrder

The `inRandomOrder` method may be used to sort the query results randomly. For example, you may use this method to fetch a random user:

```
$randomUser = DB::table('users')
    ->inRandomOrder()
    ->first();
```

groupBy / having

The `groupBy` and `having` methods may be used to group the query results. The `having` method's signature is similar to that of the `where` method:

```
$users = DB::table('users')
    ->groupBy('account_id')
    ->having('account_id', '>', 100)
    ->get();
```

For more advanced `having` statements, see the [havingRaw](#) method.

skip / take

Para limitar el número de resultados devueltos por una consulta, o saltar un número dado de resultados en la misma, puedes utilizar los métodos `skip` y `take`:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternativamente, puedes usar los métodos `limit` y `offset`:

```
$users = DB::table('users')
    ->offset(10)
    ->limit(5)
    ->get();
```

Conditional Clauses

Sometimes you may want clauses to apply to a query only when something else is true. For instance you may only want to apply a `where` statement if a given input value is present on the incoming request. You may accomplish this using the `when` method:

```
$role = $request->input('role');

$users = DB::table('users')
    ->when($role, function ($query) use ($role) {
        return $query->where('role_id', $role);
    })
    ->get();
```

The `when` method only executes the given Closure when the first parameter is `true`. If the first parameter is `false`, the Closure will not be executed.

You may pass another Closure as the third parameter to the `when` method. This Closure will execute if the first parameter evaluates as `false`. To illustrate how this feature may be used, we will use it to configure the default sorting of a query:

```
$sortBy = null;

$users = DB::table('users')
    ->when($sortBy, function ($query) use ($sortBy) {
        return $query->orderBy($sortBy);
    }, function ($query) {
        return $query->orderBy('name');
    })
    ->get();
```

Inserciones

The query builder also provides an `insert` method for inserting records into the database table. The `insert` method accepts an array of column names and values:

```
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

You may even insert several records into the table with a single call to `insert` by passing an array of arrays. Each array represents a row to be inserted into the table:

```
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the `insertGetId` method to insert a record and then retrieve the ID:

```
$id = DB::table('users')->insertGetId([  
    'email' => 'john@example.com', 'votes' => 0  
]);
```

When using PostgreSQL the `insertGetId` method expects the auto-incrementing column to be named `id`. If you would like to retrieve the ID from a different "sequence", you may pass the column name as the second parameter to the `insertGetId` method.

Actualizaciones

Por supuesto, además de insertar registros en la base de datos, el *query builder* también puede actualizar los registros existentes usando el método `update`. El método `update`, como el método `insert`, acepta un array de columnas y pares de valores que contienen las columnas que se actualizarán. Puede restringir la consulta de `update` usando `where` en las cláusulas:

```
DB::table('users')  
    ->where('id', 1)  
    ->update(['votes' => 1]);
```

Actualizar columnas JSON

Al actualizar una columna JSON, se debe usar la sintaxis `->` para acceder a la clave adecuada en el objeto JSON. Esta operación solo se admite en bases de datos que admiten columnas JSON:

```
DB::table('users')  
    ->where('id', 1)  
    ->update(['options->enabled' => true]);
```

Incrementar & decrementar

El generador de consultas también proporciona métodos convenientes para incrementar o disminuir el valor de una columna determinada. Esto es simplemente un acceso directo, que proporciona una interfaz más expresiva y concisa en comparación con la escritura manual de la declaración `update`.

Ambos métodos aceptan al menos un argumento: la columna para modificar. Y opcionalmente, se puede pasar un segundo argumento la cantidad por la cual la columna debe incrementarse o decrementarse:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

También puede especificar columnas adicionales para actualizar durante la operación:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Borrados

El *query builder* también se puede usar para eliminar registros de la tabla a través del método `delete`. Se puede restringir las sentencias `delete` añadiendo `where` a las cláusulas antes de llamar al método `delete`:

```
DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();
```

Si desea truncar toda la tabla, lo que eliminará todas las filas y restablecerá el ID incremental a cero, se puede usar el método `truncate`:

```
DB::table('users')->truncate();
```

Bloqueos persistentes

The query builder also includes a few functions to help you do "pessimistic locking" on your `select` statements. To run the statement with a "shared lock", you may use the `sharedLock` method on a query. A shared lock prevents the selected rows from being modified until your transaction commits:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the rows from being modified or from being selected with another shared lock:

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

Database: Paginación

Introducción

Uso básico

- # Paginar resultados del **query builder**

- # Paginar resultados de Eloquent

- # Crear un *paginador* manualmente

Mostrar los resultados de la paginación

- # Convertir resultados a JSON

Personalizar la vista de la paginación

Métodos de la instancia de paginación

Introducción

En otros frameworks, la paginación puede ser muy difícil. El *paginador* de Laravel está integrado con [query builder](#) y [Eloquent ORM](#), proporcionando una paginación conveniente y fácil de usar de los resultados de base de datos. El HTML generado por el *paginador* es compatible con el framework [Bootstrap CSS](#).

Uso básico

Paginar resultados del **query builder**

Hay varias formas para paginar elementos. La más simple es mediante el método `paginate` en el [query builder](#) o una [consulta de Eloquent](#). El método `paginate` se encarga automáticamente de fijar el límite y el `offset` adecuados en función de la página actual que esté viendo el usuario. De forma predeterminada, la página actual se detecta por el valor del argumento `page` que es una cadena de consulta en la petición HTTP. Por supuesto, este valor es automáticamente detectado por Laravel y se inserta automáticamente en los enlaces generados por el *paginador*.

En este ejemplo, el único argumento que se pasa al método `paginate` es el número de elementos que desea que se muestren "por página". En este caso, se especifica que se quieren mostrar 15 elementos por página:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show all of the users for the application.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->paginate(15);

        return view('user.index', ['users' => $users]);
    }
}
```

Actualmente, las operaciones de paginación que utilizan una sentencia `groupBy` pueden ser ejecutadas por Laravel de forma eficiente. Si se necesita utilizar `groupBy` con un conjunto de resultados paginado, es aconsejable realizar la consulta a la base de datos y crear un *paginador* manual.

"Paginación simple"

Si sólo necesita mostrar los enlaces simples "Siguiente" y "Anterior" en su vista de paginación, puede utilizar el método `simplePaginate` para realizar una consulta más eficiente. Esto es muy útil para grandes conjuntos de datos cuando no necesita mostrar un enlace para cada número de página al renderizar su vista:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginar resultados de Eloquent

También se pueden paginar consultas de [Eloquent](#). En este ejemplo, se paginará el modelo `User` con `15` elementos por página. Como se puede ver, la sintaxis es casi idéntica a la paginación de resultados con :

```
$users = App\User::paginate(15);
```

Por supuesto, se puede llamar a `paginate` después de establecer otras limitaciones en la consulta, tales como cláusulas `where` :

```
$users = User::where('votes', '>', 100)->paginate(15);
```

También se puede utilizar el método `simplePaginate` cuando se paginan modelos de Eloquent:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Crear un *paginador* manualmente

Se puede crear una paginación manualmente, para esto hay que pasar un array de elementos. Se puede hacer mediante la creación de una instancia de `Illuminate\Pagination\Paginator` o `Illuminate\Pagination\LengthAwarePaginator`, dependiendo de las necesidades.

La clase `Paginator` no necesita saber el número total de elementos del conjunto de resultados; sin embargo, debido a esto, no tiene métodos para recuperar el índice de la última página. El `LengthAwarePaginator` acepta casi los mismos parámetros que `Paginator`; sin embargo, requiere contar el número total de elementos del conjunto de resultados.

En otras palabras, `Paginator` corresponde al método `simplePaginate` en Eloquent, mientras que `LengthAwarePaginator` corresponde al método `paginate`.

Al crear manualmente una instancia de paginator, se debe "acortar" de forma manual el *array* de resultados que se pasa a paginator. Si no se está seguro sobre cómo hacerlo, se puede revisar la función `array_slice` de PHP.

Mostrar los resultados de la paginación

Al llamar al método `paginate`, se recibirá una instancia de `Illuminate\Pagination\LengthAwarePaginator`. Al llamar al método `simplePaginate`, se recibirá una instancia de `Illuminate\Pagination\Paginator`. Estos objetos proporcionan varios métodos que describen el conjunto de resultados. Además de estos métodos de ayuda, las instancias de *paginator* son iterables y pueden ser recorridas como un *array*. Por lo tanto, una vez que se han obtenido los resultados, se pueden mostrar y crear los enlaces de las páginas usando [Blade](#):

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

El método `links` mostrará los enlaces al resto de las páginas del conjunto de resultados. Cada uno de estos enlaces ya contendrá la variable apropiada `page` de la cadena de consulta. Recuerde, el HTML generado por el método `links` es compatible con el [Bootstrap CSS framework](#).

Personalizar la URI del *paginator*

El método `withPath` permite personalizar la URI utilizada por el *paginator* al generar enlaces. Por ejemplo, si desea que el *paginator* genere enlaces como `http://example.com/custom/url?page=N`, debe pasar `custom/url` al método `withPath`:

```
Route::get('users', function () {
    $users = App\User::paginate(15);

    $users->withPath('custom/url');

    //
});
```

Añadir variables a los links de paginación

Se puede agregar información a la cadena de consulta de los enlaces de la paginación mediante el método `appends`. Por ejemplo, para añadir `sort=votes` a cada enlace de la paginación, se debe hacer la siguiente llamada a `appends`:

```
 {{ $users->appends(['sort' => 'votes'])->links() }}
```

Si se desea agregar un ancla a las direcciones URL del *paginator*, se puede utilizar el método `fragment`. Por ejemplo, para agregar `#foo` al final de cada enlace de la paginación, se debe realizar la siguiente llamada al método `fragment`:

```
 {{ $users->fragment('foo')->links() }}
```

Convertir resultados a JSON

Las clases de resultados del *paginator* de Laravel implementan la interfaz `Illuminate\Contracts\Support\Jsonable` y exponen el método `toJson`, por lo que es muy fácil convertir los resultados de paginación a JSON. También se puede convertir una instancia *paginator* a JSON simplemente retornándola desde una ruta o acción de controlador:

```
Route::get('users', function () {
    return App\User::paginate();
});
```

El JSON del *paginator* incluirá información como `total`, `current_page`, `last_page` y más. Los objetos del resultado actual estarán disponibles a través de la clave `data` en el array JSON. Este es un ejemplo de JSON generado al retornar una instancia de paginator en una ruta:

```
{  
    "total": 50,  
    "per_page": 15,  
    "current_page": 1,  
    "last_page": 4,  
    "first_page_url": "http://laravel.app?page=1",  
    "last_page_url": "http://laravel.app?page=4",  
    "next_page_url": "http://laravel.app?page=2",  
    "prev_page_url": null,  
    "path": "http://laravel.app",  
    "from": 1,  
    "to": 15,  
    "data": [  
        {  
            // Result Object  
        },  
        {  
            // Result Object  
        }  
    ]  
}
```

Personalizar la vista de la paginación

Por defecto, las vistas usadas para mostrar los enlaces de paginación son compatibles con el framework Bootstrap CSS. Sin embargo, si no está usando Bootstrap, es libre de definir sus propias vistas para renderizar estos enlaces. Cuando llame al método `links` en una instancia de `paginator`, pase el nombre de la vista como primer argumento al método:

```
{{ $paginator->links('view.name') }}  
  
// Passing data to the view...  
{{ $paginator->links('view.name', ['foo' => 'bar']) }}
```

Sin embargo, la forma más fácil de personalizar las vistas de paginación es exportarlas a su directorio `resources/views/vendor` utilizando el comando Artisan `vendor:publish`:

```
php artisan vendor:publish --tag=laravel-pagination
```

Este comando colocará las vistas en el directorio `resources/views/vendor/pagination`. El archivo `default.blade.php` dentro de este directorio corresponde a la vista de paginación predeterminada. Simplemente edite este archivo para modificar el HTML de paginación.

Métodos de la instancia de paginación

Cada instancia de `paginator` proporciona información adicional de paginación a través de los siguientes métodos:

- `$results->count()`
- `$results->currentPage()`
- `$results->firstItem()`
- `$results->hasMorePages()`
- `$results->lastItem()`
- `$results->lastPage()` (Not available when using `simplePaginate`)
- `$results->nextPageUrl()`
- `$results->perPage()`
- `$results->previousPageUrl()`
- `$results->total()` (Not available when using `simplePaginate`)
- `$results->url($page)`

Base de Datos: Migraciones

Introducción

Generar Migraciones

Estructura de Una Migración

Ejecutar Migraciones

Revertir Migraciones

Tablas

Crear Tablas

Renombrando / Borrando Tablas

Columnas

Crear Columnas

Modificadores de columnas

Modificar Columnas

Borrar Columnas

Índices

Crear Índices

Borrar Índices

Restricciones de Claves Ajenas

Introducción

Migraciones son como un control de versión para tu base de datos, permitiéndole a tu equipo de manera sencilla modificar y compartir el esquema de la base de datos de tu aplicación. Las migraciones están normalmente emparejadas al constructor de esquemas de Laravel para construir fácilmente el esquema de la base de datos de una aplicación. Si alguna vez has tenido que decirle a un compañero de equipo que agregara manualmente una columna a su esquema de base de datos local, te has enfrentado a un problema que las migraciones de base de datos pueden resolver.

El [Schema facade](#) de laravel proporciona soporte de base de datos independiente para crear y manipular tablas en todos los sistemas de bases de datos admitidos por Laravel.

Generar Migraciones

Para crear una migración, usa el [comando de Artisan](#) `make:migration` :

```
php artisan make:migration create_users_table
```

Las nuevas migraciones serán almacenadas en el directorio `database/migrations`. Cada nombre de cada archivo de migración contiene un timestamp que permite a Laravel determinar el orden de las migraciones.

Las opciones `--table` and `--create` pueden utilizarse para indicar el nombre de la tabla y si la migración va a crear una nueva tabla. Estas opciones simplemente rellenan el archivo stub de la migración generado con la tabla especificada:

```
php artisan make:migration create_users_table --create=users
```

```
php artisan make:migration add_votes_to_users_table --table=users
```

Si se desea especificar una ruta de salida personalizada para la migración generada, se puede utilizar la opción `--path` al ejecutar el comando `make: migration`. La ruta de acceso proporcionada debe ser relativa a la ruta base de la aplicación.

Estructura de una Migración

Una clase migración contiene dos métodos: `up` y `down`. El método `up` se utiliza para agregar nuevas tablas, columnas o índices a la base de datos, mientras que el método `down` debería simplemente invertir la operación realizada por el método `up`.

Dentro de estos métodos se puede utilizar el schema builder de Laravel para crear y modificar las tablas expresivamente. Para aprender más sobre los métodos disponibles en el [Schema builder](#), [revisar la documentación](#). Por ejemplo, esta migración crea una tabla `flights`:

```

<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}

```

Ejecutar Migraciones

Para correr todas las migraciones, ejecuta el comando de Artisan `migrate` :

```
php artisan migrate
```

Si está usando la Máquina Virtual Homestead, debe ejecutar este comando dentro de su maquina virtual.

Forzando Migraciones para Ejecutarlas en Producción

Algunas operaciones de migración son destructivas, esto quiere decir que puede causar pérdida de datos. Para protegerte de ejecutar estos comandos contra tu base de datos de producción, se te pedirá confirmación antes de ejecutar los comandos. Para forzar la ejecución de estos comandos sin confirmación, usa el parámetro `--force` :

```
php artisan migrate --force
```

Revertir Migraciones

Para revertir la última migración, debes usar el comando `rollback`. Este comando regresa al último "batch" de migraciones, que puede incluir múltiples archivos de migración:

```
php artisan migrate:rollback
```

Puedes revertir un número limitado de migraciones colocando la opción `step` al comando `rollback`. Por ejemplo, el siguiente comando revertirá las últimas cinco migraciones:

```
php artisan migrate:rollback --step=5
```

El comando `migrate:reset` deshará todas las migraciones de tu aplicación:

```
php artisan migrate:reset
```

Revertir y Migrar en un Solo Comando

El comando `migrate:refresh` revertirá todas tus migraciones y luego ejecutará el comando `migrate`. Este comando re-creará efectivamente toda la base de datos:

```
php artisan migrate:refresh
```

```
// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

Puede revertir & y volver a migrar un limitado número de migraciones proporcionando la opción `step` en el comando `refresh`. Por ejemplo, el siguiente comando revertirá & y volverá a migrar las últimas cinco migraciones:

```
php artisan migrate:refresh --step=5
```

Eliminar Todas Las Tablas y Migrar

El comando `migrate:fresh` eliminará todas las tablas de la base de datos y luego ejecutará el comando `migrate`:

```
php artisan migrate:fresh
```

```
php artisan migrate:fresh --seed
```

Tablas

Crear Tablas

Para crear una nueva tabla en la base de datos, usa el método `create` del facade `Schema`. El método `create` acepta dos argumentos. El primero es el nombre de la tabla, el segundo es un `Closure` que recibe un objeto `Blueprint` usado para definir la nueva tabla:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

Por supuesto, al crear una tabla, se puede hacer uso de los [métodos de columna](#) del schema builder para definir las columnas de la tabla.

Comprobar la Existencia de Tablas / Columnas

Para comprobar la existencia de una tabla o columna se pueden utilizar los métodos `hasTable` y `hasColumn`:

```
if ($Schema::hasTable('users')) {  
    //  
}  
  
if ($Schema::hasColumn('users', 'email')) {  
    //  
}
```

Conexión y Motor de almacenamiento

Si usted quiere realizar una operación en el esquema de una conexión a la base de datos que no es la conexión por defecto, utilice el método `connection` :

```
$Schema::connection('foo')->create('users', function (Blueprint $table) {  
    $table->increments('id');  
});
```

Puedes usar la propiedad `engine` en el schema builder para definir el mecanismo de almacenamiento para la tabla:

```
$Schema::create('users', function (Blueprint $table) {  
    $table->engine = 'InnoDB';  
  
    $table->increments('id');  
});
```

Renombrando / Borrando Tablas

Para renombrar una tabla de la base de datos existente, puedes usar el método `rename` :

```
$Schema::rename($from, $to);
```

Para borrar una tabla existente, puedes usar los métodos `drop` o `dropIfExists` :

```
$Schema::drop('users');  
  
$Schema::dropIfExists('users');
```

Renombrando Tablas con Claves Foráneas

Antes de renombrar una tabla, debes verificar que cualquier restricción de la llave foránea en la tabla tiene un nombre explícito en sus archivos de migración en lugar de dejar que Laravel asigne un nombre basado en la convención. De lo contrario, el nombre de la restricción de la llave foránea se referirá al nombre de la tabla anterior.

Columnas

Crear Columnas

El metodo `table` del `Schema` facade puede ser usado para actualizar las tablas existentes. Al igual que el método `create`, el método `table` acepta dos argumentos: el nombre de la tabla y una `Closure` que recibe una instancia `Blueprint` que puede usar para agregar columnas a la tabla:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email');
});
```

Tipos de datos permitidos

Por supuesto, el Schema builder contiene una variedad de tipos de columna que puede especificar cuando este creando sus tablas:

Comando	Descripción
<code>\$table->bigIncrements('id');</code>	Equivalente a un BIGINT SIN SIGNO (llave primaria) Auto-Incremento.
<code>\$table->bigInteger('votes');</code>	Equivalente a un BIGINT.
<code>\$table->binary('data');</code>	Equivalente a un BLOB.
<code>\$table->boolean('confirmed');</code>	Equivalente a un BOOLEAN.
<code>\$table->char('name', 100);</code>	Equivalente a un CHAR con longitud opcional.

Comando	Descripción
\$table->date('created_at');	Equivalente a un DATE.
\$table->dateTime('created_at');	Equivalente a un DATETIME.
\$table->dateTimeTz('created_at');	Equivalente a un DATETIME (con zona horaria).
\$table->decimal('amount', 8, 2);	Equivalente a un DECIMAL con precisión (dígitos totales) y escala (dígitos decimales).
\$table->double('amount', 8, 2);	Equivalente a un DOUBLE con precisión (dígitos totales) y escala (dígitos decimales).
\$table->enum('level', ['easy', 'hard']);	Equivalente a un ENUM.
\$table->float('amount', 8, 2);	Equivalente a un FLOAT con precisión (dígitos totales) y escala (dígitos decimales).
\$table->geometry('positions');	Equivalente a un GEOMETRY.
\$table->geometryCollection('positions');	Equivalente a una GEOMETRYCOLLECTION.
\$table->increments('id');	Equivalente a un INTEGER SIN SIGNO (llave primaria) Auto-Incremento.
\$table->integer('votes');	Equivalente a un INTEGER.
\$table->ipAddress('visitor');	Equivalente a una dirección IP.
\$table->json('options');	Equivalente a un JSON.
\$table->jsonb('options');	Equivalente a un JSONB.
\$table->lineString('positions');	Equivalente a un LINESTRING.
\$table->longText('description');	Equivalente a un LONGTEXT.
\$table->macAddress('device');	Equivalente a una dirección MAC.

Comando	Descripción
\$table->mediumIncrements('id');	Equivalente a un MEDIUMINT SIN SIGNO (llave primaria) Auto-Incremento.
\$table->mediumInteger('votes');	Equivalente a un MEDIUMINT.
\$table->mediumText('description');	Equivalente a un MEDIUMTEXT.
\$table->morphs('taggable');	Equivalente a agregar un <code>taggable_id</code> INTEGER SIN SIGNO y <code>taggable_type</code> VARCHAR.
\$table->multiLineString('positions');	Equivalente a un MULTILINESTRING.
\$table->multiPoint('positions');	Equivalente a un MULTIPOINT.
\$table->multiPolygon('positions');	Equivalente a un MULTIPOLYGON.
\$table->nullableMorphs('taggable');	Agrega versiones nulas de <code>morphs()</code> .
\$table->nullableTimestamps();	Agrega versiones nulas de <code>timestamps()</code> .
\$table->point('position');	Equivalente a un POINT.
\$table->polygon('positions');	Equivalente a un POLYGON.
\$table->rememberToken();	Equivalente a agregar <code>remember_token</code> VARCHAR(100) que admite nulos.
\$table->smallIncrements('id');	Equivalente a un SMALLINT SIN SIGNO (llave primaria) Auto-Incremento.
\$table->smallInteger('votes');	Equivalente a un SMALLINT.
\$table->softDeletes();	Equivalente a un TIMESTAMP con campos null <code>deleted_at</code> para soft deletes.
\$table->softDeletesTz();	Equivalente a un TIMESTAMP (con zona horaria) con campos null <code>deleted_at</code> para soft deletes.

Comando	Descripción
\$table->string('name', 100);	Equivalente a un VARCHAR con longitud opcional.
\$table->text('description');	Equivalente a un TEXT.
\$table->time('sunrise');	Equivalente a un TIME.
\$table->timeTz('sunrise');	Equivalente a un TIME (con zona horaria).
\$table->timestamp('added_on');	Equivalente a un TIMESTAMP.
\$table->timestampTz('added_on');	Equivalente a un TIMESTAMP (con zona horaria).
\$table->timestamps();	Equivalente a agregar columnas nullable para <code>created_at</code> y <code>updated_at</code> TIMESTAMP.
\$table->timestampsTz();	Equivalente a agregar columnas nullable para <code>created_at</code> y <code>updated_at</code> TIMESTAMP (con zona horaria).
\$table->tinyIncrements('id');	Equivalente a un TINYINT SIN SIGNO (llave primaria) Auto-Incremento.
\$table->tinyInteger('votes');	Equivalente a un TINYINT.
\$table->unsignedBigInteger('votes');	Equivalente a un BIGINT SIN SIGNO.
\$table->unsignedDecimal('amount', 8, 2);	Equivalente a un DECIMAL SIN SIGNO, con precisión (dígitos totales) y escala (dígitos decimales).
\$table->unsignedInteger('votes');	Equivalente a un INTEGER SIN SIGNO.
\$table->unsignedMediumInteger('votes');	Equivalente a un MEDIUMINT SIN SIGNO.
\$table->unsignedSmallInteger('votes');	Equivalente a un SMALLINT SIN SIGNO.
\$table->unsignedTinyInteger('votes');	Equivalente a un TINYINT SIN SIGNO.

Comando	Descripción
<code>\$table->uuid('id');</code>	Equivalente a un UUID.

Modificadores de columnas

Además de los tipos de columna enumerados anteriormente, hay varios "modificadores" de columna que puede usar al agregar una columna a una tabla de la base de datos. Por ejemplo, para hacer la columna "Nula" (nullable) puede utilizar el método `nullable` :

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

A continuación se muestra una lista de todos los modificadores de columna disponibles. Esta lista no incluye los [index modifiers](#):

Modificador	Descripción
<code>->after('column')</code>	Coloca la columna después "after" de otra columna (solo MySQL)
<code>->autoIncrement()</code>	Establece columnas INTEGER como auto-incremento (llave primaria)
<code>->charset('utf8')</code>	Especifica un conjunto de caracteres para la columna (solo MySQL)
<code>->collation('utf8_unicode_ci')</code>	Especifica un cotejo para la columna (solo MySQL/SQL Server)
<code>->comment('my comment')</code>	Agrega un comentario a una columna (solo MySQL)
<code>->default(\$value)</code>	Especifica un valor "por defecto" para la columna
<code>->first()</code>	Coloca la columna como la primera en la tabla (sólo MySQL)
<code>->nullable(\$value = true)</code>	Permite insertar valores nulos en la columna

Modificador	Descripción
<code>->storedAs(\$expression)</code>	Crear una columna generada virtual (solo MySQL)
<code>->unsigned()</code>	Establece columnas INTEGER como SIN SIGNO (solo MySQL)
<code>->useCurrent()</code>	Establece columnas TIMESTAMP para usar CURRENT_TIMESTAMP como valor predeterminado
<code>->virtualAs(\$expression)</code>	Crear una columna generada virtual (solo MySQL)

Modificar Columnas

Requisitos Previos

Antes de modificar una columna, asegúrate de añadir la dependencia `doctrine/dbal` en tu fichero `composer.json`. La librería "Doctrine DBAL" es usada para determinar el estado actual de la columna y crear las "consultas SQL" necesarias para hacer los ajustes específicos en la columna:

```
composer require doctrine/dbal
```

Actualizando los atributos de las columnas

El método `change` permite modificar algunos tipos de columnas existentes a uno nuevo o cambiar los atributos de las columnas. Por ejemplo, puedes incrementar el tamaño de una columna de tipo string. Para ver funcionando el método `change`, vamos a incrementar el tamaño de la columna `name` desde 25 hasta 50:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

También podríamos modificar una columna para que pueda contener nulos (nullable):

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->nullable()->change();
});
```

Los siguientes tipos no se pueden "cambiar": char, double, enum, mediumInteger, timestamp, tinyInteger, ipAddress, json, jsonb, macAddress, mediumIncrements, morphs, nullableMorphs, nullableTimestamps, softDeletes, timeTz, timestampTz, timestamps, timestampsTz, unsignedMediumInteger, unsignedTinyInteger, uuid.

Renombrado columnas

Para renombrar una columna, puedes usar el método `renameColumn` en el constructor de esquemas (Schema Builder). Antes de renombrar la columna, asegúrate de añadir la dependencia `doctrine/dbal` en tu fichero `composer.json`:

```
Schema::table('users', function (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

Renombrar cualquier columna en un tabla que tambien tenga una columna de tipo `enum` no esta soportado actualmente.

Borrar Columnas

Para borrar una columna, usa el método `dropColumn` en el Schema builder. Antes de borrar columnas desde una base de datos SQLite, debes agregar la dependencia `doctrine/dbal` en el archivo `composer.json` y ejecutar el comando `composer update` en la terminal para instalar la librería:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('votes');
});
```

Puede borrar varias columnas de una tabla pasando una matriz con los nombres de las columnas usando el método `dropColumn`:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

Eliminar o modificar múltiples columnas dentro de una sola migración mientras usa una base de datos SQLite no está soportado.

Índices

Crear Índices

El constructor de esquemas (Schema Builder) soporta varios tipos de índices. Primero, veamos un ejemplo que especifica que los valores de una columnas deberían ser únicos. Para crear el índice podemos, simplemente, encadenar el método `unique` en la definición de la columna:

```
$table->string('email')->unique();
```

Como alternativa, puedes definir el nombre de la columna después de definir el índice:

```
$table->unique('email');
```

Incluso puedes pasar un array de columnas al método "index" para crear un índice compuesto:

```
$table->index(['account_id', 'created_at']);
```

Laravel automáticamente generará un nombre razonable para el índice, pero puedes pasarlo un segundo argumento al método para especificar el nombre:

```
$table->unique('email', 'unique_email');
```

Tipos de índices disponibles

Comando	Descripción
<code>\$table->primary('id');</code>	Agrega una llave primaria.
<code>\$table->primary(['id', 'parent_id']);</code>	Agrega llaves compuestas.
<code>\$table->unique('email');</code>	Agrega un índice único.
<code>\$table->index('state');</code>	Agrega un índice simple.
<code>\$table->spatialIndex('location');</code>	Agrega un índice espacial. (solo MySQL)

Longitud de índices y MySQL / MariaDB

Laravel usa el conjunto de caracteres `utf8mb4` por defecto, que incluye soporte para almacenar "emojis" en la base de datos. Si estás ejecutando una versión de MySQL anterior a la versión 5.7.7 o MariaDB anterior a la versión 10.2.2, necesitas configurar manualmente la longitud de cadena predeterminada generada por las migraciones para que MySQL cree índices para ellas. Puede configurar esto llamando al método `Schema::defaultStringLength` dentro del `AppServiceProvider` :

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Alternativamente, puedes habilitar la opción `innodb_large_prefix` para tu base de datos. Consulta la documentación de tu base de datos para obtener instrucciones sobre cómo habilitar correctamente esta opción.

Borrar Índices

Para borrar un índice, debe especificar el nombre del índice. Por defecto, Laravel asigna automáticamente un nombre lógico para los índices. Simplemente concatenando el nombre de la tabla, el nombre de la columna indexada y el tipo de el índice. Aquí tienes algunos ejemplos:

Comando	Descripción
<code>\$table->dropPrimary('users_id_primary');</code>	Elimina una clave primaria de la tabla "users".
<code>\$table->dropUnique('users_email_unique');</code>	Elimina un índice único de la tabla "users".
<code>\$table->dropIndex('geo_state_index');</code>	Elimina un índice básico de la tabla "geo".

Si pasas un array de columnas dentro de un método que borra índices, el nombre convencional del índice se generará basado en el nombre de la tabla, las columnas y el tipo de llave:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Restricciones de Llaves Foráneas

Laravel también provee soporte para la creación de restricciones de llaves foráneas, que son usadas para forzar la integridad referencial a nivel de la base de datos. Por ejemplo, definamos una columna `user_id` en la tabla `posts` que hace referencia a la columna `id` en la tabla `users` :

```
Schema::table('posts', function (Blueprint $table) {
    $table->integer('user_id')->unsigned();

    $table->foreign('user_id')->references('id')->on('users');
});
```

Puede especificar también la acción deseada para los atributos "on delete" (borrado) y "on update" (actualizado) de la restricción:

```
$table->foreign('user_id')
    ->references('id')->on('users')
    ->onDelete('cascade');
```

Para eliminar una llave foránea, debe utilizar el método `dropForeign`. Las restricciones de llave foránea usan la misma convención que los índices. Así, concatenaremos el nombre de la tabla y las columnas en la restricción y le colocaremos el sufijo "`_foreign`":

```
$table->dropForeign('posts_user_id_foreign');
```

O puedes pasar un array de valores que utilizará automáticamente el nombre convencional de la restricción cuando se esté borrando:

```
$table->dropForeign(['user_id']);
```

Puede habilitar o deshabilitar restricciones de llave foránea en sus migraciones utilizando los siguientes métodos:

```
Schema::enableForeignKeyConstraints();
```

```
Schema::disableForeignKeyConstraints();
```

Base de Datos: Poblado – *Seeding*

Introducción

Definir seeders

- # Utilizar factorías – *factories*

- # Llamar a seeders adicionales

Ejecutar seeders

Introducción

Laravel incluye un método simple para el poblado de la base de datos con datos de prueba utilizando clases "semilla" (seeds). Todas estas clases se almacenan en el directorio `database/seeds` . Las clases seed pueden tener cualquier nombre, pero normalmente deberían seguir una convención de nombres como `UsersTableSeeder` , etc. Por defecto, se incluye una clase `DatabaseSeeder` . Desde esta clase, se puede utilizar el método `call` para ejecutar otras clases de poblado, permitiendo controlar el orden.

Definir seeders

Para generar un seeder, ejecutar el [comando de Artisan](#) `make:seeder` . Todos los seeders que genera el framework se almacenarán en la carpeta `database/seeds` :

```
php artisan make:seeder UsersTableSeeder
```

Una clase seeder únicamente contiene el método `run` . Se llamará a este método cuando se ejecute el [comando de Artisan](#) `db:seed` . En el método `run` , se puede insertar información en la base de datos de cualquier modo. Se puede utilizar el [query builder](#) para insertar datos de forma manual o utilizar las [Eloquent model factories](#) (factorías de modelos Eloquent).

Como ejemplo, se va a modificar la clase por defecto `DatabaseSeeder` para añadir una declaración `insert` al método `run` :

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10). '@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

Utilizar factorías – *factories*

Por supuesto, especificar los atributos para cada modelo puede resultar tedioso. En su lugar, se pueden utilizar [factorías de modelos](#) (*model factories*) para generar grandes cantidades de registros en la base de datos. Primero revise la documentación sobre [factorías de modelos](#) para aprender a definir sus propias factorías. Una vez definidas, se puede utilizar la función *helper* `factory` para insertar registros en la base de datos.

Por ejemplo, para crear 50 usuarios y adjuntar una relación a cada usuario:

```

/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    factory(App\User::class, 50)->create()->each(function ($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
}

```

Llamar a seeders adicionales

En la clase `DatabaseSeeder`, se puede utilizar el método `call` para ejecutar otras clases de poblado. Utilizar el método `call` permite dividir el poblado de la base de datos en varios archivos por lo que las clases de poblado no se vuelven abrumadoramente grandes.

Simplemente pasar el nombre de la clase seeder a ejecutar:

```

/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    $this->call([
        UsersTableSeeder::class,
        PostsTableSeeder::class,
        CommentsTableSeeder::class,
    ]);
}

```

Ejecutar seeders

Una vez definidas las clases seeder, se puede utilizar el comando de Artisan `db:seed` para poblar la base de datos. Por defecto, el comando `db:seed` ejecuta la clase `DatabaseSeeder`, que se puede utilizar para llamar a las otras clases seed. Sin embargo, se puede utilizar la

opción `--class` para especificar una clase *seeder* concreta:

```
php artisan db:seed
```

```
php artisan db:seed --class=UsersTableSeeder
```

Se puede también poblar la base de datos utilizando el comando `migrate:refresh`, el cual hará un *rollback* (retroceso) y ejecutará todas las migraciones. Este comando resulta útil para reconstruir la base de datos completa:

```
php artisan migrate:refresh --seed
```

Redis

Introducción

Configuración

Predis

PhpRedis

Interactuando con Redis

Tuberías de comandos

Pub / Sub

Introducción

[Redis](#) es un sistema de almacenamiento avanzado clave-valor de código abierto.

Normalmente se refiere a él como un servidor de estructuras de datos puesto que las claves pueden contener [strings](#), [hashes](#), [listas](#), [sets](#), y [sets ordenados](#).

Antes de utilizar Redis con Laravel, es necesario instalar el paquete [predis/predis](#) a través de Composer:

```
composer require predis/predis
```

Alternativamente, se puede instalar la extensión de PHP [PhpRedis](#) vía PECL. La extensión es más compleja de instalar pero puede obtener mejor rendimiento para aplicaciones que tengan un uso intensivo de Redis.

Configuración

La configuración de Redis para una aplicación se encuentra en el archivo de configuración `config/database.php`. En este archivo, verá un array `redis` que contiene todos los servidores Redis utilizados por la aplicación:

```
'redis' => [  
  
  'client' => 'predis',  
  
  'default' => [  
    'host' => env('REDIS_HOST', 'localhost'),  
    'password' => env('REDIS_PASSWORD', null),  
    'port' => env('REDIS_PORT', 6379),  
    'database' => 0,  
  ],  
  
],
```

La configuración del servidor por defecto debería ser suficiente para el desarrollo. Sin embargo, se puede modificar este *array* dependiendo de su entorno. Cada servidor Redis definido en la configuración precisa de un nombre, *host* y puerto.

Configurar Clusters

Si su aplicación utiliza un cluster de servidores Redis, se debe definir en la clave `clusters` de la configuración de Redis:

```
'redis' => [  
  
  'client' => 'predis',  
  
  'clusters' => [  
    'default' => [  
      [  
        'host' => env('REDIS_HOST', 'localhost'),  
        'password' => env('REDIS_PASSWORD', null),  
        'port' => env('REDIS_PORT', 6379),  
        'database' => 0,  
      ],  
    ],  
  ],  
  
],
```

By default, clusters will perform client-side sharding across your nodes, allowing you to pool nodes and create a large amount of available RAM. However, note that client-side sharding does not handle failover; therefore, is primarily suited for cached data that is available from another primary data store. Si quiere utilizar los cluster de Redis nativos, debe especificarlo en la clave `options` del archivo de configuración de Redis:

```
'redis' => [
    'client' => 'predis',
    'options' => [
        'cluster' => 'redis',
    ],
    'clusters' => [
        // ...
    ],
],
```

Predis

Además de las opciones de configuración del servidor por defecto `host`, `port`, `database`, y `password`, Predis soporta [parámetros de conexión](#) adicionales que pueden definirse para cada uno de los servidores Redis. Para utilizar estas opciones de configuración adicionales, simplemente añádalas a la configuración de servidor de Redis en el archivo de configuración `config/database.php`:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_write_timeout' => 60,
],
```

PhpRedis

Si tiene la extensión de PhpRedis instalada via PECL, será necesario renombrar el alias `Redis` en el archivo de configuración `config/app.php`.

Para utilizar la extensión PhpRedis, debe cambiar la opción `client` de la configuración Redis a `phpredis`. Esta opción se encuentra en el archivo de configuración `config/database.php`:

```
'redis' => [
    'client' => 'phpredis',
    // Rest of Redis configuration...
],
```

Además de las opciones de configuración del servidor por defecto `host`, `port`, `database`, y `password`, PhpRedis soporta los siguientes parámetros de conexión: `persistent`, `prefix`, `read_timeout` and `timeout`. Se puede añadir cualquiera de estas opciones a la configuración del servidor Redis en el archivo de configuración `config/database.php`:

```
'default' => [
    'host' => env('REDIS_HOST', 'localhost'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => 0,
    'read_timeout' => 60,
],
```

Interactuar con Redis

Se puede interactuar con Redis llamado varios métodos de la `facade Redis`. La `facade Redis` soporta métodos dinámicos, por lo que se puede llamar a cualquier comando Redis en la `facade` y este se pasará directamente a Redis. En este ejemplo, se ejecutará el comando `GET` de Redis utilizando el método `get` de la `facade` Redis`:

```
$user]);
}
}
```

Por supuesto, como se mencionó anteriormente, se puede llamar cualquier comando Redis dentro de la facade `Redis`. Laravel usa los métodos mágicos para pasar los comandos al servidor Redis, así que simplemente basta con pasar los argumentos que espera el comando Redis:

```
Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

Como otra alternativa, se puede pasar el comando al servidor usando el método `command`, el mismo acepta el nombre del comando como primer argumento y un *array* de valores como segundo argumento:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

Utilizar varias conexiones de Redis

Se puede obtener una instancia de Redis llamando el método `Redis::connection`:

```
$redis = Redis::connection();
```

Esto retorna una instancia del servidor Redis predeterminado. También se puede pasar el nombre de una conexión o cluster al método `connection` para obtener un servidor o cluster específico tal y como se ha definido en su archivo de configuración Redis:

```
$redis = Redis::connection('my-connection');
```

Tuberías de comandos

Se deben utilizar tuberías cuando se desee enviar varios comandos al servidor en una misma operación. El método `pipeline` acepta un argumento: un `Closure` que recibe una instancia de Redis. Se pueden enviar todos sus comandos a esta instancia de Redis y se ejecutarán en una única operación:

```
Redis::pipeline(function ($pipe) {
    for ($i = 0; $i < 1000; $i++) {
        $pipe->set("key:$i", $i);
    }
});
```

Pub / Sub

Laravel provee una interfaz para los comandos de Redis `publish` y `subscribe`. Estos comandos de Reids permiten escuchar mensajes en un "canal" concreto. Se pueden publicar mensajes en el canal desde otra aplicación o incluso desde otro lenguaje de programación, permitiendo una comunicación sencilla entre aplicaciones y procesos.

Primero hay que configurar un *listener* de canal utilizando el método `subscribe`. Este método se ubicará en un [comando Artisan](#) puesto que ejecutar el método `subscribe` comienza un proceso largo:

```

<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Support\Facades\Redis;

class RedisSubscribe extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'redis:subscribe';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Subscribe to a Redis channel';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        Redis::subscribe(['test-channel'], function ($message) {
            echo $message;
        });
    }
}

```

Llegados a este punto, ya se pueden publicar mensajes en el canal utilizando el método

`publish :`

```
Route::get('publish', function () {
    // Route logic...

    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});
```

Subscripciones comodín

Utilizando el método `psubscribe`, se puede suscribir a un canal comodín, el cual puede ser útil para capturar todos los mensajes de todos los canales. El nombre de `$channel` se pasará como segundo argumento al `Closure` proporcionado como *callback*:

```
Redis::psubscribe(['*'], function ($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function ($message, $channel) {
    echo $message;
});
```

Eloquent ORM

Eloquent: Introducción

- # **Introduccion**
- # **Definir Modelos**
 - # Convenciones de Modelos Eloquent
- # **Retrieving Models**
 - # Colecciones
 - # Fragmentar Resultados
- # **Obtener Modelos / Aggregates**
 - # Obtener Aggregates
- # **Insertar & Actualizar Modelos**
 - # Inserciones
 - # Actualizaciones
 - # Asignación Masiva (Mass Assignment)
 - # Otros Metodos de Creación
- # **Eliminar Modelos**
 - # Soft Deleting (Borrado Lógico)
 - # Consultar Modelos Soft Deleted
- # **Query Scopes**
 - # Global Scopes
 - # Local Scopes
- # **Eventos**
 - # Observers

Introduccion

El ORM Eloquent que incluye Laravel provee una bonita y sencilla implementación de Active Record para trabajar con la base de datos. Cada tabla de la base de datos tiene un "Modelo" que se utiliza para interactuar con dicha tabla. Los modelos permiten consultar datos de las tablas, así como insertar nuevos elementos en ellas.

Antes de comenzar, hay que estar seguro de haber configurado la conexión a la base de datos en `config/database.php`. Para más información sobre como configurar la base de datos, consultar [la documentación](#).

Definir Modelos

Para comenzar, crearemos un modelo Eloquent. Los modelos se colocan normalmente en la raíz del directorio `app`, pero se pueden colocar en cualquier lugar que sea auto-cargado de acuerdo con el archivo `composer.json`. Todos los modelos Eloquent heredan de la clase `Illuminate\Database\Eloquent\Model`.

La forma más sencilla de crear un modelo es utilizando el [comando Artisan `make:model`](#):

```
php artisan make:model User
```

Para generar una [migración](#) cuando se genera el modelo, se puede utilizar la opción `--migration` o `-m`:

```
php artisan make:model User --migration
```

```
php artisan make:model User -m
```

Convenciones de Modelos Eloquent

Ahora, veamos un ejemplo de una clase modelo `Flight` (vuelo), el cual utilizaremos para obtener y almacenar información de la tabla `flights`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
```

Nombres de Tablas

Tener en cuenta que no se le ha indicado a Eloquent qué tabla utilizar para el modelo `Flight`. Se utilizará el "snake case" del plural del nombre de la clase como nombre de la tabla de no especificarse otro explícitamente. Por lo tanto, en este caso, Eloquent asumirá que le modelo

`Flight` almacena registros en la tabla `flights`. Se puede especificar otra tabla definiendo la propiedad `table` en el modelo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Claves Primarias

Eloquent también asumirá que cada tabla tiene una columna como clave primaria llamada `id`. Se puede definir la propiedad protegida `$primaryKey` para sobrescribir esta convención.

Además, Eloquent supone que la clave primaria es un valor entero creciente, lo que significa que, de manera predeterminada, la clave primaria se convertirá en `int` automáticamente. Si se desea utilizar una clave primaria no incremental o no numérica, se debe establecer la propiedad publica `$incrementing` en su modelo en `false`. Si la clave primaria no es un número entero, se debe establecer la propiedad protegida `$keyType` del modelo en `string`.

Timestamps

Por defecto, Eloquent espera que existan las columnas `created_at` y `updated_at` en las tablas. Para que Eloquent no gestione estas columnas automáticamente, establecer la propiedad `timestamps` del modelo a `false`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

Para personalizar el formato de los timestamps, establecer la propiedad `dateFormat` del modelo. Esta propiedad determina como los atributos de fechas son almacenados en la base de datos, así como su formato cuando el modelo es serializado a un array o JSON:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Si se necesita personalizar los nombres de las columnas utilizadas para almacenar los `timestamps`, se puede establecer las constantes `CREATED_AT` y `UPDATED_AT` en el modelo:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'last_update';
}
```

Conexión de Base de Datos

Por defecto, todos los modelos Eloquent utilizarán la conexión por defecto configurada en la aplicación. Para especificar otra conexión para un modelo, utilizar la propiedad `$connection`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

Retrieving Models

Una vez que se ha creado un modelo y [su tabla asociada en la base de datos](#), se puede comenzar a obtener datos de la base de datos. Cada modelo Eloquent es un potente [query builder](#) que permite consultar de forma fluida la tabla de la base de datos asociada al modelo. Por ejemplo:

```
<?php

use App\Flight;

$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

Añadir Constraints Adicionales

El método `all` de Eloquent retornará todos los resultados de la tabla del modelo. Puesto que cada modelo actúa como un [query builder](#), se pueden añadir constraints a las consultas, y por tanto utilizar el método `get` para obtener los resultados:

```
$flights = App\Flight::where('active', 1)
    ->orderBy('name', 'desc')
    ->take(10)
    ->get();
```

Nota: Puesto que los modelos Eloquent son query builders, se deben revisar todos los métodos disponibles en el query builder. Se puede utilizar cualquiera de estos métodos en Eloquent.

Colecciones

Para métodos de Eloquent como `all` y `get` que obtienen múltiples resultados, se retornará una instancia de `Illuminate\Database\Eloquent\Collection`. La clase `Collection` provee [una variedad de métodos útiles](#) para trabajar los resultados de Eloquent:

```
$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

Por supuesto, se puede recorrer esta colección como un array:

```
foreach ($flights as $flight) {  
    echo $flight->name;  
}
```

Fragmentar Resultados

Para procesar miles de registros Eloquent, utilizar el método `chunk`. El método `chunk` obtendrá "conjuntos" de modelos Eloquent alimentando con ellos un `Closure` para procesarlos. Utilizar el método `chunk` conservará memoria cuando se trabaja con grandes conjuntos de resultados:

```
Flight::chunk(200, function ($flights) {  
    foreach ($flights as $flight) {  
        //  
    }  
});
```

El primer argumento pasado el método es el número de registros que se desea recibir por "chunk". El Closure proporcionado como segundo parámetro se ejecutará por cada conjunto obtenido de la base de datos. Se ejecutará una consulta de base de datos para recuperar cada fragmento de registros pasados al *Closure*.

Usando Cursos

El método `cursor` permite iterar a través de los registros de la base de datos usando un cursor, que solo ejecutará una sola consulta. Al procesar grandes cantidades de datos, el método `cursor` se puede usar para reducir en gran medida el uso de la memoria:

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {  
    //  
}
```

Obtener Modelos / Aggregates

Por supuesto, además de obtener todos los registros de una tabla dada, se pueden obtener registros únicos utilizando `find` ó `first`. En lugar de devolver una colección de modelos, estos métodos retornan una instancia de un modelo único:

```
// Retrieve a model by its primary key...
$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();
```

También se puede llamar al método `find` con un array de claves primarias, que devolverá una colección de registros coincidentes:

```
$flights = App\Flight::find([1, 2, 3]);
```

Excepciones Not Found

A veces se puede necesitar lanzar excepciones si un modelo no se encuentra. Esto resulta particularmente útil en rutas y controladores. Los métodos `findOrFail` y `firstOrFail` recuperarán el primer resultado de la consulta; sin embargo, si no se encuentra ningún resultado, se arrojará una `Illuminate\Database\Eloquent\ModelNotFoundException`:

```
$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

Si no se detecta la excepción, se envía automáticamente al usuario una respuesta HTTP `404`. No es necesario escribir comprobaciones explícitas para devolver respuestas `404` cuando se utilizan estos métodos:

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

Obtener Aggregates

Por supuesto, también se pueden utilizar los [métodos aggregate](#) `count`, `sum`, `max` y otras que provee el [query builder](#). Estos métodos retornan el valor scalar apropiado en lugar de un modelo completo:

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

Insertar & Actualizar Modelos

Inserciones

Para crear un nuevo registro en la base de datos, simplemente hay que crear una nueva instancia del modelo, establecer los atributos y llamar al método `call`:

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
    /**
     * Create a new flight instance.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

En este ejemplo, simplemente se asigna el parámetro `name` de la solicitud HTTP entrante al atributo `name` de la instancia del modelo `App\Flight`. Cuando se ejecuta el método `save` se insertará un registro en la base de datos. Los timestamps `created_at` y `updated_at` se establecen automáticamente cuando se llama al método `save`, por lo que no es necesario establecerlos manualmente.

Actualizaciones

El método `save` puede también utilizarse para actualizar modelos que ya existen en la base de datos. Para actualizar un modelo, se debe recuperar, establecer los atributos a actualizar, y llamar al método `save`. De nuevo, el timestamp `updated_at` se actualizará automáticamente, por lo que no es necesario establecer su valor manualmente:

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

Mass Updates

Los updates se pueden ejecutar contra cualquier número de modelos que coincidan con una consulta concreta. En este ejemplo, todos los vuelos que están `active` y tienen una `destination` de `San Diego` se marcarán como `delayed` (retrasado):

```
App\Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

El método `update` espera un array de pares de columnas y valores representando las columnas que se deben actualizar.

Cuando se ejecuta una declaración de actualización masiva a través de Eloquent, los eventos del modelo `saved` y `updated` no se dispararán. This is because the models are never actually retrieved when issuing a mass update.

Asignación Masiva (Mass Assignment)

También se puede utilizar el método `create` para almacenar un modelo en una única línea. Desde el método se retornará la instancia del modelo insertado. Sin embargo, antes de ello, hay que especificar la propiedad `fillable` o `guarded` del modelo, pues todos los modelos Eloquent poseen protección contra la asignación en masa.

Una vulnerabilidad de asignación masiva tiene lugar cuando un usuario pasa un parámetro HTTP inesperado a través de la solicitud, y este parámetro cambia una columna de la base de datos que no se esperaba. Por ejemplo, un usuario malintencionado podría enviar un parámetro `is_admin` a través de una petición HTTP, el cual se marearía dentro del método `create` del modelo, permitiendo al usuario postularse como un administrador.

Así que, para empezar, hay que definir a qué atributos se les permite la asignación masiva. Esto se establece en la propiedad `fillable` del modelo. Por ejemplo, vamos a permitir la asignación masiva sobre el atributo `name` de nuestro modelo `Flight`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Una vez que se han marcado los atributos que permiten la asignación masiva, se puede utilizar el método `create` para insertar un nuevo registro en la base de datos. El método `create` retornará una instancia del modelo guardado:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

Si ya se tiene una instancia del modelo, se puede usar el método `fill` para completarla con un array de atributos:

```
$flight->fill(['name' => 'Flight 22']);
```

Guarding Attributes

Mientras que `$fillable` sirve como una "lista blanca" de atributos que pueden ser asignados masivamente, también se puede optar por `$guarded`. La propiedad `guarded` contiene un array de atributos que no pueden ser asignados de forma masiva. El resto de atributos que no se encuentren en el array si podrán. Por lo que, `$guarded` actúa como una "lista negra". Por supuesto, se debe establecer `$fillable` o `$guarded` - nunca ambos. En el ejemplo que sigue, todos los atributos **excepto** `price` contarán con asignación masiva:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

Si se desea que todos los atributos se puedan asignar en masa, se puede definir la propiedad `$guarded` como un array vacío:

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

Otros Metodos de Creación

`firstOrCreate / firstOrNew`

Hay otros dos métodos que se pueden utilizar para la creación de modelos asignando atributos masivamente: `firstOrCreate` y `firstOrNew`. El método `firstOrCreate` intentará encontrar un registro en la base de datos utilizando los pares de columna / valor proporcionados. Si el modelo no se puede encontrar en la base de datos, se insertará un registro con los atributos del primer parámetro, junto con los del segundo parámetro opcional.

El método `firstOrNew`, como `firstOrCreate` intentará encontrar un registro en la base de datos que concuerde con los atributos proporcionados. Sin embargo, si no se encuentra, retornará una nueva instancia del modelo. Tener en cuenta que el modelo retornado por `firstOrNew` no se ha almacenado todavía en la base de datos. Hay que llamar al método `save` de forma manual para persistirlo:

```
// Retrieve flight by name, or create it if it doesn't exist...
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

// Retrieve flight by name, or create it with the name and delayed attributes...
$flight = App\Flight::firstOrCreate(
    ['name' => 'Flight 10'], ['delayed' => 1]
);

// Retrieve by name, or instantiate...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

// Retrieve by name, or instantiate with the name and delayed attributes...
$flight = App\Flight::firstOrNew(
    ['name' => 'Flight 10'], ['delayed' => 1]
);
```

`updateOrCreate`

También se pueden encontrar situaciones en las que desee actualizar un modelo existente o crear un nuevo modelo, si no existe ninguno. Laravel proporciona un método `updateOrCreate` para hacer esto en un solo paso. Al igual que `firstOrCreate`, el método `updateOrCreate` persiste en el modelo, por lo que no es necesario llamar a `save()`:

```
// If there's a flight from Oakland to San Diego, set the price to $99.  
// If no matching model exists, create one.  
  
$flight = App\Flight::updateOrCreate(  
    ['departure' => 'Oakland', 'destination' => 'San Diego'],  
    ['price' => 99]  
);
```

Eliminar Modelos

Para eliminar un modelo, llamar al método `delete` de la instancia de este:

```
$flight = App\Flight::find(1);  
  
$flight->delete();
```

Eliminar un Modelo Existente por Clave

En el ejemplo anterior, estamos obteniendo el modelo de la base de datos antes de llamar al método `delete`. Sin embargo, si se conoce la clave primaria del modelo, se puede eliminar sin necesidad de obtenerlo. Para ello, ejecutar el método `destroy`:

```
App\Flight::destroy(1);  
  
App\Flight::destroy([1, 2, 3]);  
  
App\Flight::destroy(1, 2, 3);
```

Eliminar Modelos de una Consulta

Por supuesto, también se puede ejecutar una declaración de eliminación en un conjunto de modelos. En este ejemplo, se borrarán todos los vuelos marcados como inactivos. Al igual que las actualizaciones masivas, las eliminaciones masivas no activarán ningún evento de modelo para los modelos que se eliminan:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

Cuando se ejecuta una declaración de eliminación masiva a través de Eloquent, los eventos del modelo `deleting` y `deleted` no se dispararán. Esto se debe a que los modelos nunca se recuperan cuando se ejecuta la eliminación.

Soft Deleting (Borrado Lógico)

Además de eliminar registros de la base de datos, Eloquent también puede realizar el "borrado lógico" de modelos. Cuando un modelo se borra lógicamente, realmente no se elimina de la base de datos. En su lugar, se establece el atributo `deleted_at` del modelo y se almacena en la base de datos. Si un modelo posee un valor no-null en `deleted_at`, el modelo está soft-deleted. Para habilitar el soft deleting de un modelo, utilizar el trait `Illuminate\Database\Eloquent\SoftDeletes` sobre el modelo y añadir la columna `deleted_at` a la propiedad `$dates`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;

    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = ['deleted_at'];
}
```

Además se debe añadir la columna `deleted_at` a la tabla de la base de datos. El [schema builder](#) de Laravel posee un método para crear esta columna:

```
Schema::table('flights', function ($table) {
    $table->softDeletes();
});
```

Ahora, cuando se ejecute el método `delete` sobre el modelo, la columna `deleted_at` se establecerá con la fecha y hora actuales. Al consultar este modelo, los modelos soft-deleted se excluirán automáticamente de los resultados de la consulta.

Para determinar si una instancia de modelo ha sido soft-deleted, utilizar el método `trashed` :

```
if ($flight->trashed()) {  
    //  
}
```

Consultar Modelos Soft Deleted

Incluir Modelos Soft Deleted

Como se señaló anteriormente, los modelos soft-deleted se excluirán automáticamente de los resultados de consulta. Sin embargo, se puede forzar a incluir estos modelos en un conjunto de resultados utilizando el método `withTrashed` en la consulta:

```
$flights = App\Flight::withTrashed()  
    ->where('account_id', 1)  
    ->get();
```

El método `withTrashed` se puede utilizar también en una [relación](#):

```
$flight->history()->withTrashed()->get();
```

Obtener Solo Modelos Soft Deleted

El método `onlyTrashed` obtendrá **únicamente** modelos soft-deleted:

```
$flights = App\Flight::onlyTrashed()  
    ->where('airline_id', 1)  
    ->get();
```

Restaurar Modelos Soft Deleted

Para "deshacer" el borrado de un modelo soft-deleted a un estado activo, utilizar el método `restore` sobre la instancia del modelo:

```
$flight->restore();
```

También se puede utilizar `restore` sobre una consulta para restaurar múltiples modelos rápidamente. Nuevamente, al igual que las otras operaciones "masivas", esto no disparará ningún evento para los modelos que se restauran:

```
App\Flight::withTrashed()  
    ->where('airline_id', 1)  
    ->restore();
```

Al igual que le método `withTrashed`, `restore` se puede utilizar en [relaciones](#):

```
$flight->history()->restore();
```

Eliminar Modelos Permanentemente

Para realmente eliminar un modelo de la base de datos de forma permanente, utilizar el método `forceDelete` :

```
// Force deleting a single model instance...  
$flight->forceDelete();  
  
// Force deleting all related models...  
$flight->history()->forceDelete();
```

Query Scopes

Global Scopes

Los Global scopes permiten agregar restricciones a todas las consultas para un modelo determinado. La propia funcionalidad de [soft delete](#) de Laravel utiliza global scopes para extraer únicamente modelos "no eliminados" de la base de datos. Escribir propios global scopes puede proporcionar una manera conveniente y fácil de asegurarse de que cada consulta para un modelo determinado reciba ciertas restricciones.

Escribiendo Global Scopes

Escribir un global scope es bastante sencillo. Hay que definir una clase que implemente la interface `Illuminate\Database\Eloquent\Scope`. La interface requiere que se implemente un solo método: `apply`. En el método `apply` se pueden agregar `where` como restricciones a la consulta según sea necesario:

```
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope implements Scope
{
    /**
     * Apply the scope to a given Eloquent query builder.
     *
     * @param \Illuminate\Database\Eloquent\Builder $builder
     * @param \Illuminate\Database\Eloquent\Model $model
     * @return void
     */
    public function apply(Builder $builder, Model $model)
    {
        $builder->where('age', '>', 200);
    }
}
```

Si su ámbito global está agregando columnas a la cláusula de selección de la consulta, debe usar el método `addSelect` en lugar de `select`. Esto evitará el reemplazo involuntario de la query de selección existente de la consulta.

Aplicando Global Scopes

Para asignar un global scope a un modelo, se debe sobreescribir el método `boot` de un modelo y usar el método `addGlobalScope`:

```
<?php

namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope);
    }
}
```

Después de agregar el scope, una consulta a `User::all()` producirá el siguiente SQL:

```
select * from `users` where `age` > 200
```

Anonymous Global Scopes

Eloquent also allows you to define global scopes using Closures, which is particularly useful for simple scopes that do not warrant a separate class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('age', function (Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. The method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AgeScope::class)->get();
```

If you would like to remove several or even all of the global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();

// Remove some of the global scopes...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

Local Scopes

Los local scopes permiten definir un conjunto de constraints que se pueden reutilizar fácilmente a lo largo de la aplicación. Por ejemplo, puede que sea frecuente recuperar todos los usuarios considerados "popular". Para definir un scope, simplemente hay que prefijar un modelo Eloquent con `scope`.

Los scopes devolverán siempre una instancia del query buider:

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     *
     * @param \Illuminate\Database\Eloquent\Builder $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     *
     * @param \Illuminate\Database\Eloquent\Builder $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
        return $query->where('active', 1);
    }
}

```

Utilizar un Local Scope

Una el scope se ha definido, se puede llamar cuando se consulta el modelo. Sin embargo, no es necesario incluir el prefijo `scope` cuando se llama al método. Incluso se pueden encadenar llamadas a varios scopes, por ejemplo:

```
$users = App\User::popular()->active()->orderBy('created_at')->get();
```

Scopes Dinámicos

A veces es necesario que un scope acepte parámetros. Para ello, simplemente añadir los parámetros adicionales al scope. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     *
     * @param \Illuminate\Database\Eloquent\Builder $query
     * @param mixed $type
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

Ahora, ya se pueden pasar parámetros al scope:

```
$users = App\User::ofType('admin')->get();
```

Eventos

Eloquent models fire several events, allowing you to hook into the following points in a model's lifecycle: `retrieved` , `creating` , `created` , `updating` , `updated` , `saving` , `saved` , `deleting` , `deleted` , `restoring` , `restored` . Los eventos permiten ejecutar código de forma sencilla cada vez que una clase de un modelo específico se guarda o actualiza en la base de datos.

The `retrieved` event will fire when an existing model is retrieved from the database. When a new model is saved for the first time, the `creating` and `created` events will fire. Si el modelo ya existe en la base de datos y se llama al método `save`, se lanzarán los eventos `updating` / `updated`. Sin embargo, para ambos casos, se lanzarán los eventos `saving` / `saved`.

To get started, define a `$dispatchesEvents` property on your Eloquent model that maps various points of the Eloquent model's lifecycle to your own [event classes](#):

```
<?php

namespace App;

use App\Events\UserSaved;
use App\Events\UserDeleted;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observers classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the model as their only argument. Laravel does not include a default directory for observers, so you may create any directory you like to house your observer classes:

```
<?php

namespace App\Observers;

use App\User;

class UserObserver
{
    /**
     * Listen to the User created event.
     *
     * @param \App\User $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Listen to the User deleting event.
     *
     * @param \App\User $user
     * @return void
     */
    public function deleting(User $user)
    {
        //
    }
}
```

To register an observer, use the `observe` method on the model you wish to observe. You may register observers in the `boot` method of one of your service providers. In this example, we'll register the observer in the `AppServiceProvider` :

```
<?php

namespace App\Providers;

use App\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        User::observe(UserObserver::class);
    }

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Eloquent: Relaciones

Introducción

Definir relaciones

- # Uno a uno – *One to one*
- # Uno a muchos – *One to many*
- # Uno a muchos (inverso)
- # Muchos a muchos – *Many to many*
- # Has Many Through
- # Relaciones polimórficas
- # Relaciones polimórficas muchos a muchos

Consultas relacionadas

- # Métodos de relación vs. propiedades dinámicas
- # Consultar la existencia de relaciones
- # Consultar la inexistencia de relaciones
- # Contar modelos relacionados

Carga temprana – *Eager loading*

- # Restricciones de carga temprana – *Constraints*
- # Carga temprana pasiva – *Lazy eager loading*

Insertar & actualizar modelos relacionados

- # El método `save`
- # El método `create`
- # Relaciones "pertenece a" – *Belongs to*
- # Relaciones muchos a muchos – *Many to Many*

Touching Parent Timestamps

Introducción

Las tablas de las bases de datos se relacionan a menudo unas con otras. Por ejemplo, un artículo de un blog puede tener muchos comentarios, o un pedido podría estar relacionado con el usuario que lo solicitó. Eloquent facilita la gestión y el trabajo con estas relaciones fácilmente soportando varios tipos de relaciones diferentes:

- Uno a uno – *One to one*

- [Uno a muchos – One to many](#)
- [Muchos a muchos – Many to many](#)
- [Has Many Through](#)
- [Relaciones polimórficas](#)
- [Relaciones polimórficas muchos a muchos](#)

Definir relaciones

Las relaciones entre modelos Eloquent se definen como métodos en las propias clases. Dado que, como los propios modelos Eloquent, las relaciones también sirven como poderosos [query builders](#), la definición de relaciones como métodos proporciona potentes funciones de encadenamiento y consulta de métodos. Por ejemplo, se pueden encadenar restricciones adicionales en esta relación `posts` :

```
$user->posts()->where('active', 1)->get();
```

Pero, antes de sumergirnos demasiado en el uso de las relaciones, aprendamos a definir cada tipo.

Uno a uno – *One to one*

La relación uno-a-uno es una relación muy básica. Por ejemplo, un modelo `User` podría estar asociado con uno `Phone`. Para definir esta relación, hay que poner un método `phone` en el modelo `User`. El método `phone` debería llamar al método `hasOne` y retornar su resultado:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

El primer argumento pasado al método `hasOne` es el nombre del modelo relacionado. Una vez que la relación está definida, se deben recuperar los registros relacionados usando las propiedades dinámicas de Eloquent. Las propiedades dinámicas le permiten acceder a los métodos de la relación como si fueran propiedades definidas en el modelo:

```
$phone = User::find(1)->phone;
```

Eloquent determina la clave foránea de la relación basada en el nombre del modelo. En este caso, se asume que el modelo `Phone` tiene un clave foránea `user_id`. Si se desea sobrescribir esta convención, hay que pasar un segundo parámetro al método `hasOne` :

```
return $this->hasOne('App\Phone', 'foreign_key');
```

Además, Eloquent asume que la llave foránea debe tener un valor que coincida con la columna `id` (o la `$primaryKey` personalizada) columna del padre. En otras palabras, Eloquent buscará el valor de la columna `id` de los usuarios en la columna `user_id` de los registros `Phone`. Si desea que la relación utilice un valor distinto de `id`, puede pasar un tercer parámetro al método `hasOne` especificando la clave personalizada:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Definir la inversa de la relación

Por lo tanto, se puede acceder al modelo `Phone` desde `User`. Ahora se definirá una relación en el modelo `Phone` que permitirá acceder al `User` que posee el teléfono. Se puede definir la inversa de la relación `hasOne` utilizando el método `belongsTo`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

En el ejemplo anterior, Eloquent intentará emparejar el campo `user_id` del modelo `Phone` con un `id` del modelo `User`. Eloquent nombra la clave ajena por defecto con el nombre del método de la relación y el sufijo `_id`. Sin embargo, si la clave ajena del modelo `Phone` no es `user_id`, se puede pasar una diferente como segundo parámetro el método `belongsTo`:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key');
}
```

Si el modelo padre no utiliza `id` como clave primaria, o se desea hacer un join del modelo hijo a una columna diferente, se puede pasar un tercer parámetro al método `belongsTo` especificando la clave personalizada del modelo padre:

```
/**  
 * Get the user that owns the phone.  
 */  
  
public function user()  
{  
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');  
}
```

Modelos por defecto

La relación `belongsTo` permite definir un modelo por defecto para devolver si la relación es `null`. Este patrón se conoce normalmente como [patrón del objeto nulo – Null Object Pattern](#) y puede ayudar a eliminar comprobaciones condicionales en el código. En el siguiente ejemplo, la relación `user` devolverá un modelo `App\User` si no hay un `user` asociado al `post`:

```
/**  
 * Get the author of the post.  
 */  
  
public function user()  
{  
    return $this->belongsTo('App\User')->withDefault();  
}
```

Para poblar el modelo por defecto con atributos, se puede pasar un *array* o *Closure* al método `withDefault`:

```

/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault(function ($user) {
        $user->name = 'Guest Author';
    });
}

```

Uno a muchos – *One to many*

Una relación "uno-a-muchos" se usa para definir relaciones en las cuales un modelo único posee cualquier cantidad de otros modelos. Por ejemplo, un blog puede tener un número infinito de comentarios. Como en otras relaciones de Eloquent, las relaciones uno-a-muchos se definen colocando una función en el modelo Eloquent:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Hay que recordar que Eloquent determinará automáticamente la columna de la clave externa adecuada en el modelo `Comment`. Por convenio, Eloquent unirá el nombre del propio modelo y el sufijo `_id` mediante la convención "snake_case". Así, para este ejemplo, Eloquent asumirá que la clave ajena del método `Comment` es `post_id`.

Una vez que la relación ha sido definida, se puede acceder a la colección de comentarios accediendo a la propiedad `comments`. Recuerde, puesto que Eloquent provee de "propiedades dinámicas", se puede acceder a los métodos de la relación como si fueran propiedades del modelo:

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Por supuesto, puesto que todas las relaciones sirven como *query builders*, se pueden agregar agregar *constraints* adicionales a los comentarios que fueron obtenidos con la llamada al método `comments` y continuar la cadena de condiciones en la consulta:

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

Como en el método `hasOne`, se puede reemplazar las claves locales y foráneas pasando argumentos adicionales al método `hasMany`:

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

Uno a muchos (inverso)

Ahora que podemos acceder a todos los comentarios del `post`, vamos a definir la relación para permitir que un comentario acceda a su publicación padre. Para definir el inverso de una relación `hasMany`, definir una función de relación en el modelo hijo que llame al método `belongsTo`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Una vez que la relación se ha definido, se puede recuperar el modelo `Post` desde `Comment` accediendo a la "propiedad dinámica" `post`:

```
$comment = App\Comment::find(1);

echo $comment->post->title;
```

En el ejemplo anterior, Eloquent intentará emparejar el campo `user_id` del modelo `Phone` con un `id` del modelo `User`. Eloquent nombra la clave ajena por defecto con el nombre del método de la relación y el sufijo `_id`. Sin embargo, si la clave ajena del modelo `Phone` no es `user_id`, se puede pasar una diferente como segundo parámetro el método `belongsTo`:

```
/**  
 * Get the post that owns the comment.  
 */  
  
public function post()  
{  
    return $this->belongsTo('App\Post', 'foreign_key');  
}
```

Si el modelo padre no utiliza `id` como clave primaria, o se desea hacer un *join* del modelo hijo a una columna diferente, se puede pasar un tercer parámetro al método `belongsTo` especificando la clave personalizada del modelo padre:

```
/**  
 * Get the post that owns the comment.  
 */  
  
public function post()  
{  
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');  
}
```

Muchos a muchos – *Many to many*

Las relaciones muchos-a-muchos son un poco más complicadas que las `hasOne` o las `hasMany`. Un ejemplo de tal relación es un usuario que contiene varios roles, donde los roles son compartidos por otros usuarios. Por ejemplo, varios usuarios pueden tener el rol de "Admin". Para definir esta relación, se requieren tres tablas de la base de datos: `users`, `roles`, y `role_user`. La tabla `role_user` es derivada del orden alfabético de los nombres de los modelos relacionados y contiene las columnas `user_id` y `role_id`.

Las relaciones muchos-a-muchos se definen con un método que retorna el resultado del método `belongsToMany`. Por ejemplo, definir el método `roles` en el modelo `User`:

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}

```

Una vez definida la relación, se puede acceder a los roles del usuario usando la propiedad dinámica `roles` :

```

$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}

```

Por supuesto, como en las otras relaciones, se puede llamar al método `roles` para continuar encadenando restricciones a la consulta sobre la relación:

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

Como se ha mencionado anteriormente, para determinar el nombre de la tabla relacionada en un join, Eloquent unirá los nombres de los dos modelos relacionados en orden alfabético. Sin embargo, hay libertad para sobrescribir esta convención. Se puede hacer pasando un segundo parámetro al método `belongsToMany` :

```
return $this->belongsToMany('App\Role', 'role_user');
```

Además de poder personalizar el nombre de las tablas en un join, también se puedes personalizar los nombres de las columnas de las claves en la tabla añadiendo más argumentos al método `belongsToMany`. El tercer argumento es el nombre de la clave foránea en la cual se está definiendo la relación, mientras que el cuarto argumento es el nombre de la clave foránea del modelo al que se está haciendo el join:

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

Definiendo la Inversa de la Relación

Para definir la inversa de una relación de muchos a muchos, simplemente hay que poner otra llamada a `belongsToMany` en el modelo relacionado. Para continuar con el ejemplo de roles de usuario, se va a definir el método de `users` en el modelo `Role`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

Como se puede ver, la relación está definida exactamente de igual forma que su contraria `User`, con la excepción de que simplemente referencia al modelo `App\User`. Al reutilizar el método `belongsToMany`, todas las opciones para personalizar las claves están disponibles cuando se define la inversa de las relaciones muchos-a-muchos.

Recuperar Columnas de Tablas Intermedias

Como se ha visto, trabajar con relaciones muchos-a-muchos requiere la presencia de una tabla intermedia. Eloquent proporciona algunas formas muy útiles de interactuar con esta tabla. Por ejemplo, supongamos que un objeto `User` tiene muchos objetos `Role` con los que

se relaciona. Después de acceder a esta relación, se puede acceder a la tabla intermedia utilizando el atributo `pivot` en los modelos:

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Hay que tener en cuenta que el modelo obtenido `Role`, está asignado automáticamente a un atributo `pivot`. Este atributo contiene un modelo que representa la tabla intermedia y puede ser utilizado como cualquier otro modelo Eloquent.

Por defecto, sólo las claves del modelo estarán presentes en el objeto `pivot`. Si la tabla pivot contiene atributos adicionales, se deben especificar al definir la relación:

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

Si se desea que los campos `created_at` y `updated_at` se mantengan de forma automática, hay que utilizar el método `withTimestamps` al definir la relación:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Customizing The `pivot` Attribute Name

As noted earlier, attributes from the intermediate table may be accessed on models using the `pivot` attribute. However, you are free to customize the name of this attribute to better reflect its purpose within your application.

For example, if your application contains users that may subscribe to podcasts, you probably have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table accessor to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```
return $this->belongsToMany('App\Podcast')
    ->as('subscription')
    ->withTimestamps();
```

Once this is done, you may access the intermediate table data using the customized name:

```
$users = User::with('podcasts')->get();  
  
foreach ($users->flatMap->podcasts as $podcast) {  
    echo $podcast->subscription->created_at;  
}
```

Filtering Relationships Via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` using the `wherePivot` and `wherePivotIn` methods when defining the relationship:

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);  
  
return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your relationship, you may call the `using` method when defining the relationship. All custom models used to represent intermediate tables of relationships must extend the `Illuminate\Database\Eloquent\Relations\Pivot` class. For example, we may define a `Role` which uses a custom `UserRole` pivot model:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Role extends Model  
{  
    /**  
     * The users that belong to the role.  
     */  
    public function users()  
    {  
        return $this->belongsToMany('App\User')->using('App\UserRole');  
    }  
}
```

When defining the `UserRole` model, we will extend the `Pivot` class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Relations\Pivot;

class UserRole extends Pivot
{
    //
}
```

Has Many Through

The "has-many-through" relationship provides a convenient shortcut for accessing distant relations via an intermediate relation. Por ejemplo, un modelo `Country`, puede tener muchos modelos `Post` a través de un modelo intermedio `User`. En este ejemplo, se podría reunir fácilmente todos los mensajes de un blog para un país determinado. Echemos un vistazo a las tablas para definir esta relación:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Aunque `Post` no contiene una columna `country_id`, la relación `hasManyThrough` proporciona acceso a los mensajes de un país vía `$country->posts`. Para realizar esta consulta, Eloquent inspecciona `country_id` en la tabla intermedia `users`. Después de encontrar los IDs de usuarios coincidentes, serán usados para la consulta a la tabla `posts`.

Ahora que se ha examinado la estructura de la tabla para la relación, se va a definir sobre el modelo `Country` :

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{
    /**
     * Get all of the posts for the country.
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

El primer parámetro pasado al método `hasManyThrough` es el nombre del modelo final al que se desea acceder, mientras que el segundo parámetro es el nombre del modelo intermedio.

Los convenios típicos para claves de Eloquent serán usados para realizar las consultas de la relación. Si se desea personalizar las claves de la relación, se puede hacer por los parámetros tercero y cuarto al método `hasManyThrough`. El tercer parámetro es el nombre de la clave ajena del modelo intermedio. El cuarto parámetro corresponde con el nombre de la clave ajena del modelo final. El quinto argumento es la clave local, mientras que el sexto es la clave local del modelo intermedio:

```
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post',
            'App\User',
            'country_id', // Foreign key on users table...
            'user_id', // Foreign key on posts table...
            'id', // Local key on countries table...
            'id' // Local key on users table...
        );
    }
}
```

Relaciones polimórficas

Estructura de tablas

Las relaciones polimórficas permiten a un modelo pertenecer a más de un modelo en una sola asociación. Por ejemplo, imaginar usuarios que pueden "comentar" tanto *posts* como vídeos. Utilizando relaciones polimórficas, se puede utilizar una única tabla `comments` para ambos escenarios. Primero, examinemos la estructura de la tabla requerida para construir esta relación:

```
posts
  id - integer
  title - string
  body - text

videos
  id - integer
  title - string
  url - string

comments
  id - integer
  body - text
  commentable_id - integer
  commentable_type - string
```

Las dos columnas a destacar aquí son `commentable_id` y `commentable_type` de la tabla `comments`. La columna `commentable_id` contiene el ID del *post* o *video*, mientras que la columna `commentable_type` contiene el nombre de la clase del modelo propietario. La columna `commentable_type` es como el ORM determina el "tipo" de modelo propietario a retornar cuando se accede a la relación `commentable`.

Estructura del modelo

Examinemos que definiciones en el modelo son necesarias para construir esta relación:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get all of the owning commentable models.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Comment', 'commentable');
    }
}
```

Obtener relaciones polimórficas

Una vez definidas la tabla y los modelos de la base de datos, es posible acceder a las relaciones a través de sus modelos. Por ejemplo, para acceder a los comentarios de un `post`, se puede utilizar la propiedad dinámica `'comments'`:

```
$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}
```

Se puede obtener el propietario de una relación polimórfica desde el modelo polimórfico accediendo al nombre del método que ejecuta la llamada a `morphTo`. En este caso, es el método `commentable` en el modelo `Comment`. Ahora se puede acceder a ese método como propiedad dinámica:

```
$comment = App\Comment::find(1);

$commentable = $comment->commentable;
```

La relación `commentable` del modelo `Comment` retornará una instancia de `Post` o `Video`, dependiendo del tipo de modelo al que pertenezca.

Tipos polimórficos personalizados

Por defecto, Laravel utilizará el nombre completo de la clase para almacenar el tipo del modelo relacionado. Por ejemplo, dado el ejemplo anterior donde `Comment` puede pertenecer a un `Post` o `Video`, el valor por defecto de `commentable_type` será `App\Post` o `App\Video` respectivamente. Sin embargo, es posible desacoplar la base de datos de la estructura interna de la aplicación. En este caso, se puede definir un "mapa de morfismo" (*morph map*) para instruir a Eloquent en el uso de un nombre personalizado para cada modelo en lugar del nombre de la clase:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::morphMap([
    'posts' => 'App\Post',
    'videos' => 'App\Video',
]);
```

Se puede registrar el `morphMap` en la función `boot` del `AppServiceProvider` o en un `service provider` separado si se desea.

Relaciones polimórficas muchos a muchos

Estructura de Tablas

Además de las relaciones polimórficas tradicionales, se pueden definir relaciones polimórficas "muchos-a-muchos" (many-to-many). Por ejemplo, un `Post` o `Video` de un post podrían compartir una relación polimórfica a un modelo `Tag`. Utilizar relaciones polimórficas muchos-a-muchos permite tener una única lista de tags que se comparten tanto con los blog posts como con los videos. Primero, esta sería la estructura de tablas:

```
posts
  id - integer
  name - string

videos
  id - integer
  name - string

tags
  id - integer
  name - string

taggables
  tag_id - integer
  taggable_id - integer
  taggable_type - string
```

Estructura del Modelo

A continuación, se definen las relaciones en el modelo. Los modelos `Post` y `Video` tendrán un método `tags` que llama al método `morphToMany` de la clase base Eloquent:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags()
    {
        return $this->morphToMany('App\Tag', 'taggable');
    }
}
```

Definiendo la Inversa de la Relación

Después, en el modelo `Tag`, se debe definir un método por cada modelo a relacionar. Por lo que, en este ejemplo, se definirá un método `posts` y otro `videos`:

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts()
    {
        return $this->morphedByMany('App\Post', 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany('App\Video', 'taggable');
    }
}

```

Recuperando La Relacion

Una vez definidas la tabla y los modelos de la base de datos, es posible acceder a las relaciones a través de sus modelos. Por ejemplo, para acceder a todas las etiquetas para un post sólo tiene que utilizar la `etiqueta` propiedad dinámica:

```

$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}

```

También puede recuperar el dueño de una relación polimórfica del modelo polimórfico mediante el acceso al nombre del método que realiza la llamada a `morphedByMany`. En nuestro caso, es decir, los `post` o `videos` métodos en la `Etiqueta` modelo. Por lo tanto, usted tendrá

acceso a los métodos como propiedades dinámicas:

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

Consultas Relacionadas

Since all types of Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing the relationship queries. Además, todos los tipos de relaciones Eloquent también sirven como [constructores de consultas](#), permitiendo continuar la cadena de restricciones sobre la consulta de la relación antes de finalizar la ejecución de SQL contra la base de datos.

Por ejemplo, imagine un sistema de blog en el cual un `Usuario` modelo tiene muchos `mensajes` modelos asociados:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

Puede consultar la relación de los `mensajes` y agregar restricciones adicionales a la relación, así como:

```
$user = App\User::find(1);

$user->posts()->where('active', 1)->get();
```

Se puede utilizar cualquiera de los métodos de [query builder](#) en la relación, por lo que debe asegurarse de explorar la documentación del generador de consultas para obtener información sobre todos los métodos disponibles.

Métodos de Relación Vs. Propiedades Dinámicas

Si no necesita agregar una restricción adicional a una consulta de relación Eloquent, puede simplemente acceder a la relación como si fuese una propiedad. Por ejemplo, al continuar usando nuestro ejemplo de modelos de `usuario` y `mensaje`, necesitaríamos acceso a los mensajes del usuario, así como:

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

Las propiedades dinámicas son "carga perezosa", significando que sólo cargarán sus datos de relación cuando actualmente se accede a ellos. Por esto, los desarrolladores normalmente usan [carga ansiosa](#) para precargar las relaciones que se sabe que serán accedidas después de cargar el modelo. La carga ansiosa provee una significante reducción en las consultas SQL que deben ser ejecutadas para cargar las relaciones de los modelos.

Consultar la Existencia de Relaciones

Cuando se acceden a los registros de un modelo, se pueden limitar los resultados basados en la existencia de una relación. Por ejemplo, imaginar que se desea obtener todos los posts que contengan al menos un comentario. Para ello, se pasaría el nombre de la relación al método `has` o a `orHas`:

```
// Retrieve all posts that have at least one comment...
$posts = App\Post::has('comments')->get();
```

Además se puede especificar un operador y un contador para personalizar la consulta:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Se pueden anidar estructuras `has` utilizando la notación de "puntos". Por ejemplo, se podrían obtener todos los posts que tienen al menos un comentario y un voto:

```
// Retrieve all posts that have at least one comment with votes...
$posts = Post::has('comments.votes')->get();
```

Si se necesita todavía más control, se pueden utilizar los métodos `whereHas` y `orWhereHas` para incluir condiciones "where" en las consultas `has`. Estos métodos permiten añadir constraints personalizadas a una relación, así como comprobar el contenido de un comentario:

```
// Retrieve all posts with at least one comment containing words like foo%
$post = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Consultar la inexistencia de relaciones

Cuando se acceden a los registros de un modelo, se pueden limitar los resultados basados en la inexistencia de una relación. Por ejemplo, imaginar que se desea obtener todos los posts que **no** contengan al menos un comentario. Para ello, se pasaría el nombre de la relación al método `doesntHave` o a `orDoesntHave`:

```
$posts = App\Post::doesntHave('comments')->get();
```

Si se necesita todavía más control, se pueden utilizar los métodos `whereDoesntHave` y `orWhereDoesntHave` para incluir condiciones "where" en las consultas `doesntHave`. Estos métodos permiten añadir constraints personalizadas a una relación, así como comprobar el contenido de un comentario:

```
$posts = Post::whereDoesntHave('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Contar modelos relacionados

If you want to count the number of results from a relationship without actually loading them you may use the `withCount` method, which will place a `{relation}_count` column on your resulting models. Por ejemplo:

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

You may add the "counts" for multiple relations as well as add constraints to the queries:

```
$posts = Post::withCount(['votes', 'comments' => function ($query) {
    $query->where('content', 'like', 'foo%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

You may also alias the relationship count result, allowing multiple counts on the same relationship:

```
$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function ($query) {
        $query->where('approved', false);
    }
])->get();

echo $posts[0]->comments_count;

echo $posts[0]->pending_comments_count;
```

Carga temprana

Cuando se acceden a las relaciones de Eloquent como propiedades, los datos de la relación se cargan de forma "diferida" (lazy loaded). Esto quiere decir que los datos de la relación no se cargan hasta que se accede a la propiedad. Sin embargo, Eloquent puede hacer "Carga Ambiciosa" de relaciones en el momento que se consulta al modelo padre. La carga ambiciosa alivia el problema de consulta N + 1. Para ilustrar el problema de consulta N + 1, considere un modelo `Libro` que está relacionado al modelo `Author` :

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

Ahora, vamos a recuperar todos los libros y sus autores:

```
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Este bucle ejecutará 1 consulta para recuperar todos los libros sobre la mesa, y luego otra consulta por cada libro para recuperar su autor. Por lo tanto, si tenemos 25 libros, este bucle correría 26 consultas: 1 por el libro original, y 25 consultas adicionales para recuperar el autor de cada libro.

Afortunadamente, podemos usar la carga ambiciosa para reducir esta operación a 2 consultas. Cuando se consulta, tu puedes especificar cuales relaciones deben cargarse también usando el método `with` :

```
$books = App\Book::with('author')->get();  
  
foreach ($books as $book) {  
    echo $book->author->name;  
}
```

Para esta operación, únicamente se ejecutarán dos consultas:

```
select * from books  
  
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Carga Ambiciosa De Relaciones Múltiples

A veces puede necesitar hacer carga ambiciosa de diferentes relaciones en una sola operación. Para ello, solo tiene que pasar argumentos adicionales al método `with`:

```
$books = App\Book::with(['author', 'publisher'])->get();
```

Carga temprana anidada

Para hacer Carga Ambiciosa de relaciones anidadas, debes usar la sintaxis "punto". Por Ejemplo, hagamos Carga Ambiciosa de todos los autores de los libros y de todos los contactos personales de los autores en una sentencia de Eloquent:

```
$books = App\Book::with('author.contacts')->get();
```

Eager Loading Specific Columns

You may not always need every column from the relationships you are retrieving. For this reason, Eloquent allows you to specify which columns of the relationship you would like to retrieve:

```
$users = App\Book::with('author:id,name')->get();
```

When using this feature, you should always include the `id` column in the list of columns you wish to retrieve.

Restringiendo la carga temprana

A veces puedes desear hacer Carga Ambiciosa a una relación, pero también especificar restricciones adicionales a la consulta para la hacer la Carga Ambiciosa. He aquí un ejemplo:

```
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `first`. Of course, you may call other [query builder](#) methods to further customize the eager loading operation:

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

Carga anticipada (Lazy Eager Loading)

A veces puede que tenga que hacer Carga Ambiciosa de una relación después de que el modelo padre ya se ha recuperado. Por ejemplo, esto puede ser útil si usted necesita decidir dinámicamente si debe cargar modelos relacionados:

```
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

If you need to set additional query constraints on the eager loading query, you may pass an array keyed by the relationships you wish to load. The array values should be [Closure](#) instances which receive the query instance:

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

To load a relationship only when it has not already been loaded, use the `loadMissing` method:

```
public function format(Book $book)
{
    $book->loadMissing('author');

    return [
        'name' => $book->name,
        'author' => $book->author->name
    ];
}
```

Inserting & Updating Related Models

El método Save

Eloquent provee métodos convenientes para la adición de nuevos modelos a las relaciones. Por ejemplo, quizás necesite insertar un nuevo `Comment` a un modelo `Post`. En lugar de configurar manualmente el atributo `post_id` en el `Comment`, puede insertar el `Comment` directamente desde el método `save` de la relación:

```
$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);
```

Nótese que no accedemos a los `comments` de la relación como una propiedad dinámica. En su lugar, llamamos al método `comments` para obtener una instancia de la relación. El método `save` agregará automáticamente el valor `post_id` apropiado al nuevo modelo `Comment`.

Si usted necesita grabar multiples modelos relacionados, puede usar el método `saveMany`

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

El método Create

Además de los métodos `save` y `saveMany`, usted puede utilizar también el método `create`, el cuál acepta una matriz de atributos, crea el modelo y lo inserta en la base de datos De nuevo, la diferencia entre `save` y `create` es que `save` acepta una instancia de un modelo completo de Eloquent mientras que `create` acepta una `matriz` de PHP:

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Before using the `create` method, be sure to review the documentation on attribute mass assignment.

You may use the `createMany` method to create multiple related models:

```
$post = App\Post::find(1);

$post->comments()->createMany([
    [
        'message' => 'A new comment.',
    ],
    [
        'message' => 'Another new comment.',
    ],
]);
```

Belongs To Relationships

Cuando actualizamos una relación `belongsTo`, se suele utilizar el método `associate`. Este método establecerá la clave foránea en el modelo hijo.

```
$account = App\Account::find(10);

$user->account()->associate($account);

$user->save();
```

When removing a `belongsTo` relationship, you may use the `dissociate` method. This method will set the relationship's foreign key to `null` :

```
$user->account()->dissociate();
```

```
$user->save();
```

Relaciones Muchos a Muchos

Adjuntar / Separar (Attaching / Detaching)

Eloquent also provides a few additional helper methods to make working with related models more convenient. Por ejemplo, imaginar que un usuario puede tener varios roles y un rol puede tener varios usuarios. Para adjuntar un rol a un usuario insertando un registro en la tabla intermedia que une los modelos, utilizar el método `attach` :

```
$user = App\User::find(1);
```

```
$user->roles()->attach($roleId);
```

Cuando se adjunta una relación a un modelo, se puede pasar además un array de datos adicional para insertarlo en la tabla intermedia:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

Por supuesto, a veces es necesario eliminar un rol de un usuario. Para eliminar un registro de una relación muchos-a-muchos, utilizar el método `detach`. El método `detach` eliminará el registro apropiado de la tabla intermedia; sin embargo, ambos modelos permanecerán en la base de datos:

```
// Detach a single role from the user...
```

```
$user->roles()->detach($roleId);
```

```
// Detach all roles from the user...
```

```
$user->roles()->detach();
```

Por comodidad, `attach` y `detach` aceptan además un array de IDs como entrada:

```
$user = App\User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires]
]);
```

Syncing Associations

Usted puede usar también el método `sync` para construir asociaciones muchos-a-muchos El método `sync` acepta una matriz de IDs para almacenar en la tabla intermedia (pivot) Cualquier ID que no esté contenido en la matriz será eliminado de la tabla intermedia So, after this operation is complete, only the IDs in the given array will exist in the intermediate table:

```
$user->roles()->sync([1, 2, 3]);
```

Usted puede pasar además valores adicionales con los IDs para la tabla intermedia:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

If you do not want to detach existing IDs, you may use the `syncWithoutDetaching` method:

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

Toggling Associations

The many-to-many relationship also provides a `toggle` method which "toggles" the attachment status of the given IDs. If the given ID is currently attached, it will be detached. Likewise, if it is currently detached, it will be attached:

```
$user->roles()->toggle([1, 2, 3]);
```

Saving Additional Data On A Pivot Table

Cuando trabajamos con una relación muchos-a-muchos, el método `Save` acepta , como segundo argumento, una matriz de attributes de la tabla anexa (tabla pivot)

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

Updating A Record On A Pivot Table

If you need to update an existing row in your pivot table, you may use `updateExistingPivot` method. This method accepts the pivot record foreign key and an array of attributes to update:

```
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

Touching Parent Timestamps

Cuando un modelo `belongsTo` (pertenece a) o `belongsToMany` (pertenece a muchos) otro modelo, tal como `Comment` pertenece a `Post`, puede ser útil actualizar los *timestamps* del padre cuando el hijo se actualiza. Por ejemplo, cuando un `Comment` se actualiza, se puede "tocar" (*touch*) el *timestamp* de `updated_at` del padre `Post`. Eloquent hace esto proceso muy sencillo. Simplemente añadir la propiedad `touches` conteniendo los nombres de las relaciones a tocar al modelo hijo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Ahora, al actualizar un `Comment`, el `Post` propietario actualizará también su columna `updated_at`, permitiendo saber cuando invalidar una caché del modelo `Post`:

```
$comment = App\Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```

Eloquent: Colecciones

- # Introducción
 - # Métodos disponibles
 - # Colecciones personalizadas
-

Introducción

Todos los conjuntos multi-resultado que devuelve Eloquent son instancias de [Illuminate\Database\Eloquent\Collection](#), incluyendo los resultados que se obtienen a través del método `get` o que se acceden a través de una relación. La colección Eloquent hereda de [Collection](#), por lo que hereda docenas de métodos que se utilizarán para trabajar de forma fluida con el *array* de modelos Eloquent subyacente.

Por supuesto, todas las colecciones sirven además como "iteradores", permitiendo iterar sobre ellas como si fueran *arrays* de PHP normales:

```
$users = App\User::where('active', 1)->get();  
  
foreach ($users as $user) {  
    echo $user->name;  
}
```

Sin embargo, las colecciones son mucho más potentes que los *arrays* y exponen una gran variedad de operaciones de mapeo / reducción que pueden ser encadenadas utilizando una interfaz muy intuitiva. Por ejemplo, para eliminar todos los modelos inactivos y obtener el nombre de cada usuario restante:

```

$users = App\User::where('active', 1)->get();

$names = $users->reject(function ($user) {
    return $user->active === false;
})
->map(function ($user) {
    return $user->name;
});

```

Mientras que la mayoría de los métodos de las colecciones Eloquent devuelven una nueva instancia de la colección, los métodos `pluck`, `keys`, `zip`, `collapse`, `flatten` y `flip` retornan una instancia de la colección base. Así mismo, si una operación `map` retorna una colección que no contiene ningún modelo Eloquent, será automáticamente convertida a una colección normal.

Métodos disponibles

La colección base

Todas las colecciones Eloquent heredan del objeto [colección de Laravel](#); por tanto, heredan todos los potentes métodos que provee esta clase base:

all	each	intersect
average	eachSpread	isEmpty
avg	every	isNotEmpty
chunk	except	keyBy
collapse	filter	keys
combine	first	last
concat	flatMap	map
contains	flatten	mapInto
containsStrict	flip	mapSpread
count	forget	mapToGroups
crossJoin	forPage	mapWithKeys
dd	get	max
diff	groupBy	median
diffKeys	has	merge
dump	implode	min

mode	reverse	transform
nth	search	union
only	shift	unique
pad	shuffle	uniqueStrict
partition	slice	unless
pipe	sort	values
pluck	sortBy	when
pop	sortByDesc	where
prepend	splice	whereStrict
pull	split	whereIn
push	sum	whereInStrict
put	take	whereNotIn
random	tap	whereNotInStrict
reduce	toArray	zip
reject	toJson	

Colecciones personalizadas

Para utilizar un objeto `Collection` personalizado con métodos propios, se puede reemplazar el método `newCollection` del modelo:

```
<?php

namespace App;

use App\CustomCollection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Create a new Eloquent Collection instance.
     *
     * @param  array  $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new CustomCollection($models);
    }
}
```

Una vez definido el método `newCollection`, se obtendrá una instancia de esta colección personalizada cada vez que Eloquent retorne una instancia `Collection` de ese modelo. Para utilizar una colección personalizada para cada modelo de la aplicación, debe sobrescribir el método `newCollection` en un modelo base que debe heredarse por el resto de modelos.

Eloquent: Mutators

Introducción

Accessors y mutators

Definiendo un *accessor*

Definiendo un *mutator*

Mutators de fechas

Casting de atributos

Casting Array & JSON

Introducción

Los *accessors* (accesores) y *mutators* (mutadores) permiten formatear los valores de los atributos de Eloquent cuando se accede a ellos o bien se establece o cambia su valor en una instancia. Por ejemplo, se puede usar [Laravel encrypter](#) para encriptar un valor mientras es almacenado en la base de datos, y luego automáticamente desencriptar el atributo cuando se acceder a él en un modelo de Eloquent.

Además de poder personalizar los *accessors* y *mutators*, Eloquent permite convertir campos de fechas a instancias de [Carbon](#) o incluso de [convertir campos texto a JSON](#).

Accessors y mutators

Definiendo un *accessor*

Para definir un *accessor*, hay que crear un método `getFooAttribute` en el modelo donde `Foo` es el nombre "studly" de la columna a la que se desea acceder (importante las mayúsculas y minúsculas). En este ejemplo, se define un accessor para el atributo `first_name`. Eloquent llamará automáticamente al método *accessor* cuando se intente obtener el valor del atributo `first_name`:

```

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the user's first name.
     *
     * @param  string  $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}

```

Como se puede ver, el valor original de la columna es pasado al *accessor*, permite manipularlo y devolverlo. Para acceder al valor del *accessor*, únicamente hay que acceder al atributo `first_name` en una instancia del modelo:

```

$user = App\User::find(1);

$firstName = $user->first_name;

```

Por supuesto, se pueden utilizar *accesors* para devolver valores computados de atributos existentes:

```

/**
 * Get the user's full name.
 *
 * @return string
 */
public function getFullNameAttribute()
{
    return "{$this->first_name} {$this->last_name}";
}

```

Definiendo un Mutator

Para definir un *mutator*, hay que crear un método `setFooAttribute` en el modelo donde `Foo` es el nombre "study" de la columna a la que se desea acceder (importante mayúsculas y minúsculas). Así que ahora se va a definir un *mutator* para el atributo `first_name`. Eloquent llamará automáticamente a este *mutator* cuando se intente alterar el valor del atributo `first_name` del modelo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Set the user's first name.
     *
     * @param string $value
     * @return void
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

El mutador recibirá el valor a modificar en el atributo, permitiendo la manipulación del mismo y guardarla en la propiedad interna `$attributes`. Así por ejemplo, si se intenta guardar `Sally` en el atributo `first_name`:

```
$user = App\User::find(1);

$user->first_name = 'Sally';
```

En este ejemplo, la función `setFirstNameAttribute` será llamada con el valor `Sally`. El *mutador* intentará aplicar la función `strtolower` al nombre y guardará su valor en el *array* interno `$attributes`.

Mutators de fechas

Por defecto, Eloquent convertirá las columnas `created_at` y `updated_at` en instancias de [Carbon](#), las cuales proporcionan una gran variedad de métodos útiles y heredan de la clase nativa de PHP `DateTime`. Se puede personalizar qué campos de fecha deben ser automáticamente mutados, e incluso completamente desactivar esta opción sobrescribiendo la propiedad `$dates` del modelo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be mutated to dates.
     *
     * @var array
     */
    protected $dates = [
        'created_at',
        'updated_at',
        'deleted_at'
    ];
}
```

Cuando una columna se considera una fecha, se puede configurar su valor a un *timestamp de Unix*, una cadena de fecha (`Y-m-d`), una cadena *date-time* y por supuesto una instancia `DateTime` / [Carbon](#). Las fechas serán automáticamente almacenadas correctamente en la base de datos:

```
$user = App\User::find(1);

$user->deleted_at = Carbon::now();

$user->save();
```

Como se indicó anteriormente, cuando se recuperan atributos que son listados en la propiedad `dates`, automáticamente serán convertidos a instancias de [Carbon](#), permitiendo utilizar cualquier método de Carbon sobre los atributos:

```
$user = App\User::find(1);

return $user->deleted_at->getTimestamp();
```

Formatos de fecha

Por defecto, los *timestamps* tiene el formato `'Y-m-d H:i:s'`. Si se necesita personalizar el formato del *timestamp*, hay que configurar la propiedad `$dateFormat` del modelo. Esta propiedad determina como los atributos de fechas son almacenados en la base de datos, así como su formato cuando el modelo es serializado a un array o JSON:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Casting de Atributos

La propiedad `$casts` del modelo proporciona un método adecuado para convertir atributos a tipos de datos comunes. La propiedad `$casts` debe contener un array donde la clave es el nombre del atributo a aplicar el *casting* y el valor el tipo de *casting* a realizar. Los tipos soportados para convertir son: `integer`, `real`, `float`, `double`, `string`, `boolean`, `object`, `array`, `collection`, `date`, `datetime`, y `timestamp`.

Por ejemplo, para convertir el atributo `is_admin`, el cual se almacena en la base de datos como un entero (`o 1`) a un valor booleano:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

Ahora el atributo `is_admin` siempre será convertido a tipo `boolean` cuando se acceda a él, incluso si el valor subyacente es almacenado en la base de datos como un entero:

```
$user = App\User::find(1);

if ($user->is_admin) {
    //
}
```

Casting Array & JSON

El tipo de conversión `array` es especialmente útil cuando se trabaja con columnas que están almacenadas como JSON serializados. Por ejemplo, si la base de datos tiene un campo de tipo `TEXT` o `JSON` que contiene un JSON serializado, añadiendo el *cast* `array` al atributo, automáticamente *deserializará* el atributo a un array de PHP cuando se acceda a él en un modelo Eloquent:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast to native types.
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

Una vez la conversión está definida, se puede tener acceso al atributo `options` y automáticamente será deserializado desde un JSON a un array de PHP. Cuando se establezca el valor del atributo `options`, el array dado será automáticamente serializado en JSON para su almacenamiento:

```
$user = App\User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

Eloquent: Recursos API

- # **Introducción**
- # **Generating Resources**
- # **Concept Overview**
- # **Writing Resources**
 - # Data Wrapping
 - # Pagination
 - # Conditional Attributes
 - # Conditional Relationships
 - # Adding Meta Data
- # **Resource Responses**

Introducción

Cuando se construye una API, se puede necesitar una capa de transformación que se ubica entre los modelos de Eloquent y las respuestas JSON que se devuelven a los usuarios de la aplicación. Las clases de recursos de Laravel permiten transformar los modelos y las colecciones en JSON.

Generar recursos

Para generar una clase recurso, se puede utilizar el comando de Artisan `make:resource` . Por defecto, los recursos se colocan en el directorio `app/Http/Resources` de la aplicación. Los recursos extienden de la clase `Illuminate\Http\Resources\Json\Resource` :

```
php artisan make:resource User
```

Recursos para Colecciones

Además de generar recursos que transforman modelos individuales, es posible generar recursos que son responsables de transformar colecciones de modelos. Esto le permite a la respuesta incluir enlaces y otra meta información que es relevante para la colección completa de un recurso en particular.

Para crear un recurso para colecciones, se debe usar la bandera `--collection` cuando se crea el recurso. O, simplemente incluir la palabra `Collection` en el nombre del recurso le indica a Laravel que debe crear un recurso para colecciones. Los recursos para colecciones extienden de la clase `Illuminate\Http\Resources\Json\ResourceCollection`:

```
php artisan make:resource Users --collection
```

```
php artisan make:resource UserCollection
```

Revisión de conceptos

Esta es una presentación de conceptos de alto nivel sobre los recursos y los recursos para colecciones. Se recomienda encarecidamente leer las otras secciones de esta documentación para ganar un conocimiento más profundo sobre la personalización y el poder que ofrecen los recursos.

Antes de entrar en detalle de las opciones disponibles cuando se crean recursos, demos una mirada de alto nivel a cómo se usan los recursos en Laravel. Una clase recurso representa un único modelo que necesita transformarse en una estructura JSON. Por ejemplo, esta es una simple clase de recurso de `User`:

```

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class User extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}

```

Cada clase recurso define un método `toArray` que retorna un *array* de atributos que deben ser convertidos a JSON cuando se envíe la respuesta. Tenga en cuenta que se pueden acceder a las propiedades del modelo directamente desde la variable `$this`. Esto es debido a que una clase de recurso automáticamente conectará las propiedades y los métodos del modelo que se inyecta para facilitar el acceso a los mismos. Una vez que se define el recurso, este puede retornarse desde una ruta o un controlador:

```

use App\User;
use App\Http\Resources\User as UserResource;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});

```

Colección de recursos

Si se está retornando una colección de recursos o una respuesta paginada, se puede usar el método `collection` cuando se crea la instancia del recurso en la ruta o controlador:

```
use App\User;  
use App\Http\Resources\User as UserResource;  
  
Route::get('/user', function () {  
    return UserResource::collection(User::all());  
});
```

Por supuesto, esto no permite añadir ninguna ningún metadato que pueda ser necesario retornarlo con la colección. Si se quiere personalizar la respuesta del recurso para colecciones, se puede crear un recurso dedicdo a representar la colección:

```
php artisan make:resource UserCollection
```

Una vez que se ha generado el recurso para colecciones, se puede definir fácilmente cualquier metadato que debería incluirse con la respuesta:

```

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}

```

Después de definir el recurso para colecciones, se puede retornar desde una ruta o un controlador:

```

use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
    return new UserCollection(User::all());
});

```

Escribir recursos

Si no se ha revisado la revisión de conceptos, se recomienda encarecidamente hacerlo antes de seguir adelante con la documentación.

Esencialmente, los recursos son simples. Sólo necesitan transformar un modelo dado en *array*. Así, cada recurso contiene un método `toArray` que traduce los atributos de dicho modelo a un *array* que es amigable para la API y que puede ser devuelto a los usuarios:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class User extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Una vez que se define el recurso, puede retornarse directamente desde una ruta o controlador:

```
use App\User;  
use App\Http\Resources\User as UserResource;  
  
Route::get('/user', function () {  
    return new UserResource(User::find(1));  
});
```

Relaciones

Si se quiere incluir un recurso relacionado en la respuesta, se puede simplemente agregarlo en el *array* que se retorna del método `toArray`. En este ejemplo se usa el método `collection` del recurso `Post` para agregar los posts del usuario en la respuesta del recurso:

```
/**  
 * Transform the resource into an array.  
 *  
 * @param \Illuminate\Http\Request  
 * @return array  
 */  
  
public function toArray($request)  
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'email' => $this->email,  
        'posts' => Post::collection($this->posts),  
        'created_at' => $this->created_at,  
        'updated_at' => $this->updated_at,  
    ];  
}
```

Si se quiere incluir una relación sólo cuando se haya cargado previamente, revise la documentación sobre relaciones condicionales.

Colección de recursos

Mientras los recursos traducen un modelo en un *array*, los recursos para colecciones traducen una colección de modelos a un *array*. No es absolutamente necesario definir una clase de recurso para colecciones por cada uno de los tipos de modelos existentes ya que todos los

recursos incluyen un método `collection` que genera un recurso para colecciones "ad-hoc" en el momento:

```
use App\User;  
use App\Http\Resources\User as UserResource;  
  
Route::get('/user', function () {  
    return UserResource::collection(User::all());  
});
```

Sin embargo, si se necesita personalizar los metadatos que se retornan con la colección, será necesario definir un recurso para colecciones:

```
<?php  
  
namespace App\Http\Resources;  
  
use Illuminate\Http\Resources\Json\ResourceCollection;  
  
class UserCollection extends ResourceCollection  
{  
    /**  
     * Transform the resource collection into an array.  
     *  
     * @param \Illuminate\Http\Request  
     * @return array  
     */  
    public function toArray($request)  
    {  
        return [  
            'data' => $this->collection,  
            'links' => [  
                'self' => 'link-value',  
            ],  
        ];  
    }  
}
```

Tal como los recursos singulares, los recursos para colecciones se pueden retornar desde las rutas o los controladores:

```
use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Encapsulamiento de datos

Por defecto, el recurso más externo se encapsula en una clave `data` cuando la respuesta se convierte a JSON. Así, por ejemplo, un recurso para colecciones se parece a lo siguiente:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ]
}
```

Si se quiere deshabilitar el encapsulamiento del recurso externo, se debe usar el método `withoutWrapping` en la clase base del recurso. Normalmente, se puede llamar a este método desde el `AppServiceProvider` o desde otro [service provider](#) que se cargue en cada petición a la aplicación:

```

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Http\Resources\Json\Resource;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Resource::withoutWrapping();
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}

```

nota El método `withoutWrapping` sólo afecta a la respuesta exterior y no removerá las claves `data` que se hayan agregado manualmente a los recursos para colecciones.

Encapsular recursos anidados

Se tiene total libertad para determinar cómo se encapsulan las relaciones entre los recursos. Si se quiere tener todos los recursos encapsulados dentro de una clave `data`, sin importar su anidado, se debe definir una clase de recurso para colecciones por cada recurso y retornar la colección dentro de una clave `data`.

Por supuesto, queda en duda si esto causará que el recurso exterior sea encapsulados en dos claves `data`. No hay de qué alarmarse, Laravel no permitirá que los recursos se encapsulen dos veces, así que no debe haber preocupación acerca del nivel de anidado de la colección que se está transformando:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return ['data' => $this->collection];
    }
}
```

Encapsular datos y paginación

Cuando se retorna una colección paginada en la respuesta, Laravel encapsulará los datos en una clave `data` aunque se haya llamado al método `withoutWrapping`. Esto se hace ya que las respuestas paginadas siempre contienen las claves `meta` y links con información acerca del estado del paginador:

```
{
  "data": [
    {
      "id": 1,
      "name": "Eladio Schroeder Sr.",
      "email": "therese28@example.com",
    },
    {
      "id": 2,
      "name": "Liliana Mayert",
      "email": "evandervort@example.com",
    }
  ],
  "links": {
    "first": "http://example.com/pagination?page=1",
    "last": "http://example.com/pagination?page=1",
    "prev": null,
    "next": null
  },
  "meta": {
    "current_page": 1,
    "from": 1,
    "last_page": 1,
    "path": "http://example.com/pagination",
    "per_page": 15,
    "to": 10,
    "total": 10
  }
}
```

Paginación

Siempre se puede pasar una instancia de *paginator* al método `collection` de un recurso o de un recurso para colecciones personalizado:

```
use App\User;
use App\Http\Resources\UserCollection;

Route::get('/users', function () {
    return new UserCollection(User::paginate());
});
```

Las respuestas paginadas siempre contienen las claves `meta` y `links` con información sobre el estado del *paginator*:

```
{
    "data": [
        {
            "id": 1,
            "name": "Eladio Schroeder Sr.",
            "email": "therese28@example.com",
        },
        {
            "id": 2,
            "name": "Liliana Mayert",
            "email": "evandervort@example.com",
        }
    ],
    "links":{
        "first": "http://example.com/pagination?page=1",
        "last": "http://example.com/pagination?page=1",
        "prev": null,
        "next": null
    },
    "meta":{
        "current_page": 1,
        "from": 1,
        "last_page": 1,
        "path": "http://example.com/pagination",
        "per_page": 15,
        "to": 10,
        "total": 10
    }
}
```

Atributos condicionales

En ocasiones es necesario incluir un atributo en un recurso para colecciones sólo si se cumple con una condición dada. Por ejemplo, se puede querer incluir un valor si el usuario actual es un "administrador". Laravel provee una variedad de métodos auxiliares que son de ayuda en estas situaciones. El método `when` se puede usar para agregar condicionalmente un atributo a una respuesta:

```
/**  
 * Transform the resource into an array.  
 *  
 * @param \Illuminate\Http\Request  
 * @return array  
 */  
  
public function toArray($request)  
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'email' => $this->email,  
        'secret' => $this->when($this->isAdmin(), 'secret-value'),  
        'created_at' => $this->created_at,  
        'updated_at' => $this->updated_at,  
    ];  
}
```

En este ejemplo, la clave `secret` solo será retornada en la respuesta final si el método `$this->isAdmin()` retorna `true`. Si el método retorna `false`, la clave `secret` se excluirá completamente de la respuesta antes de ser enviada al cliente. El método `when` permite definir los recursos expresivamente sin hacer uso de condicionales durante la construcción del `array`.

El método `when` también acepta un *Closure* como segundo argumento, permitiendo calcular el valor resultante únicamente si la condición dada es `true`:

```
'secret' => $this->when($this->isAdmin(), function () {  
    return 'secret-value';  
}),
```

Recuerde que las llamadas a los métodos en los recursos, se propagan hasta la instancia del modelo subyacente. Así, en este caso, el método `isAdmin` se propaga al modelo Eloquent que se dió originalmente al recurso.

Combinar atributos condicionales

En ocasiones, se tienen varios atributos que sólo deben incluirse en la respuesta del recurso si se cumple la misma condición. En ese caso, se puede usar el método `mergeWhen` para incluir los atributos en la respuesta únicamente cuando la condición dada es `true` :

```
/**  
 * Transform the resource into an array.  
 *  
 * @param \Illuminate\Http\Request  
 * @return array  
 */  
  
public function toArray($request)  
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'email' => $this->email,  
        $this->mergeWhen($this->isAdmin(), [  
            'first-secret' => 'value',  
            'second-secret' => 'value',  
        ]),  
        'created_at' => $this->created_at,  
        'updated_at' => $this->updated_at,  
    ];  
}
```

De nuevo, si la condición dada es `false`, estos atributos se eliminarán completamente de la respuesta antes de que sea enviada al cliente.

nota El método `mergeWhen` no debería usarse dentro de arrays que mezclan claves numéricas y de texto. Y más allá, no debería usarse en arrays que tienen claves numéricas que no están ordenadas secuencialmente.

Relaciones condicionales

Además de cargar atributos de forma condicional, se pueden incluir relaciones condicionales a las respuestas del recurso basadas en si la relación ya se ha cargado en el modelo. Esto le permite al controlador decidir cuáles relaciones deben cargarse en el modelo y el recurso puede fácilmente incluirlas sólo cuando estas han sido cargadas.

Finalmente, esto hace más fácil evitar los problemas de consultas "N+1" en los recursos. El método `whenLoaded` se puede usar para cargar condicionalmente una relación. Para evitar la carga innecesaria de relaciones, este método acepta el nombre de la relación en lugar de la relación propiamente:

```
/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => Post::collection($this->whenLoaded('posts')),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

En este ejemplo, si la relación no se ha cargado, la clave `posts` se eliminará completamente de la respuesta antes de enviarse al cliente.

Información *pivot* condicional

Sumado a incluir condicionalmente información de las relaciones en las respuesta de los recursos, se puede incluir condicionalmente datos de las tablas intermedias en las relaciones muchos-a-muchos usando el método `whenPivotLoaded`. El método `whenPivotLoaded` acepta el nombre de la tabla *pivot* como su primer argumento. El segundo argumento debería ser un *Closure* que defina el valor que debe retornarse si la información está disponible en el modelo:

```
/**  
 * Transform the resource into an array.  
 *  
 * @param \Illuminate\Http\Request  
 * @return array  
 */  
  
public function toArray($request)  
{  
    return [  
        'id' => $this->id,  
        'name' => $this->name,  
        'expires_at' => $this->whenPivotLoaded('role_users', function () {  
            return $this->pivot->expires_at;  
        }),  
    ];  
}
```

Agregar metadatos

Algunos estándares de API JSON requieren la adición de metadatos a las respuestas de los recursos y los recursos para colecciones. Frecuentemente esto incluye cosas como `links` al recurso o recursos relacionados, o metadatos acerca del recurso propiamente. Si se necesita retornar metadatos adicionales acerca de un recurso, simplemente deben incluirse en el método `toArray`. Por ejemplo, se puede incluir información de `link` cuando se transforma un recurso para colecciones:

```
/**  
 * Transform the resource into an array.  
 *  
 * @param \Illuminate\Http\Request  
 * @return array  
 */  
  
public function toArray($request)  
{  
    return [  
        'data' => $this->collection,  
        'links' => [  
            'self' => 'link-value',  
        ],  
    ];  
}
```

Cuando se retornan metadatos adicionales desde los recursos, no hay que preocuparse de reemplazar accidentalmente las claves `links` o `meta` que Laravel incluye automáticamente cuando se trata de respuestas paginadas. Cualquier clave `links` adicional simplemente se fusionará con los `links` que provienen del *paginator*.

Meta-datos al máximo nivel

En ocasiones, es posible que desee incluir ciertos metadatos con una respuesta de recursos si el recurso es un recurso externo el que se devuelve. Normalmente esto incluye meta-information sobre la respuesta como un todo. Para definir estos metadatos, agregue un método `with` a la clase recurso. Este método debe devolver un *array* de metadatos para incluir con la respuesta del recurso solo cuando el mismo es el recurso más externo que se está representando:

```

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return parent::toArray($request);
    }

    /**
     * Get additional data that should be returned with the resource array.
     *
     * @param \Illuminate\Http\Request $request
     * @return array
     */
    public function with($request)
    {
        return [
            'meta' => [
                'key' => 'value',
            ],
        ];
    }
}

```

Agregar metadatos al construir recursos

También se puede agregar datos de nivel superior al construir instancias de recursos en la ruta o controlador. El método `additional`, está disponible en todos los recursos, acepta un *array* de datos que se debe agregar a la respuesta del recurso:

```
return (new UserCollection(User::all()->load('roles'))  
    ->additional(['meta' => [  
        'key' => 'value',  
    ]]);
```

Respuestas de los Recursos

Como ya ha leído, los recursos pueden ser devueltos directamente desde las rutas y controladores:

```
use App\User;  
use App\Http\Resources\User as UserResource;  
  
Route::get('/user', function () {  
    return new UserResource(User::find(1));  
});
```

Sin embargo, a veces se puede necesitar personalizar la respuesta HTTP saliente antes de enviarla al cliente. Hay dos maneras de lograr esto. En primer lugar, se puede encadenar el método `response` al recurso. Este método devolverá una instancia de `Illuminate\Http\Response`, lo que permitirá un control total de los encabezados de la respuesta:

```
use App\User;  
use App\Http\Resources\User as UserResource;  
  
Route::get('/user', function () {  
    return (new UserResource(User::find(1))  
        ->response()  
        ->header('X-Value', 'True'));  
});
```

Alternativamente, se puede definir un método `withResponse` dentro del mismo recurso. Se llamará a este método cuando el recurso se devuelva como el recurso externo en una respuesta:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class User extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
        ];
    }

    /**
     * Customize the outgoing response for the resource.
     *
     * @param \Illuminate\Http\Request
     * @param \Illuminate\Http\Response
     * @return void
     */
    public function withResponse($request, $response)
    {
        $response->header('X-Value', 'True');
    }
}
```

Eloquent: Serialización

Introducción

Serializar modelos & colecciones

Serializar a arrays

Serializar a JSON

Ocultar atributos del JSON

Añadir valores al JSON

Serializar fechas

Introducción

Cuando se construyen APIs JSON, es común convertir modelos y sus relaciones en *arrays* o *JSON*. Eloquent incluye métodos para realizar estas conversiones, así como controlar qué atributos se incluyen en las serializaciones.

Serializar modelos & colecciones

Serializar a arrays

Para convertir un modelo y sus [relationships](#) en un *array*, debe usar el método `toArray`. Este método es recursivo, por lo que todos los atributos y relaciones (incluidas las relaciones de relaciones) se convertirán a *arrays*:

```
$user = App\User::with('roles')->first();

return $user->toArray();
```

Además también puede convertir modelos enteros de [colecciones](#) a *arrays*:

```
$users = App\User::all();  
  
return $users->toArray();
```

Serializando a JSON

Para convertir un modelo a JSON, debe simplemente utilizar el método `toJson`. Como `toArray`, el método `toJson` es recursivo, por lo que todos los atributos y relaciones se convertirán a JSON:

```
$user = App\User::find(1);  
  
return $user->toJson();
```

De forma alternativa, puede convertir un modelo o colección a una cadena, la cual llamará automáticamente al método `toJson` sobre el modelo de la colección:

```
$user = App\User::find(1);  
  
return (string) $user;
```

Puesto que los modelos y colecciones se convierten a JSON cuando se hace un casting de cadena, puede retornar objetos Eloquent directamente desde sus controladores o rutas de aplicación:

```
Route::get('users', function () {  
    return App\User::all();  
});
```

Ocultar atributos del JSON

Algunas veces podría desear limitar los atributos, tales como contraseñas, que están incluídas en su representación JSON o *array* del modelo. Para hacerlo, agregue una propiedad `$hidden` a su modelo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = ['password'];
}
```

Al ocultar relaciones, use el nombre del método de la relación.

Por otro lado, se puede definir la propiedad `visible` para definir una lista blanca de atributos que deben ser incluidos en sus representaciones JSON y *array* del modelo. Todos los demás atributos estarán ocultos cuando el modelo se convierta en un *array* o JSON:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be visible in arrays.
     *
     * @var array
     */
    protected $visible = ['first_name', 'last_name'];
}
```

Modificar la visibilidad de atributos temporalmente

Si desea hacer visibles algunos atributos normalmente ocultos en una instancia de modelo determinada, puede usar el método `makeVisible`. El método `makeVisible` devuelve la instancia del modelo para un conveniente encadenamiento de métodos:

```
return $user->makeVisible('attribute')->toArray();
```

Asimismo, si desea ocultar algunos atributos normalmente visibles en una instancia de modelo determinada, puede usar el método `makeHidden`.

```
return $user->makeHidden('attribute')->toArray();
```

Añadir valores al JSON

Ocasionalmente, al convertir modelos a un `array` o `JSON`, puede querer agregar atributos que no tienen una columna correspondiente en su base de datos. Para ello, hay que definir un [accessor](#) para el atributo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the administrator flag for the user.
     *
     * @return bool
     */
    public function getIsAdminAttribute()
    {
        return $this->attributes['admin'] == 'yes';
    }
}
```

Después de crear el *accessor*, añada el nombre del atributo a la propiedad `appends` del modelo. Note que los nombres de los atributos se mencionan normalmente en "snake case", aunque el accessor se define usando "camel case":

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The accessors to append to the model's array form.
     *
     * @var array
     */
    protected $appends = ['is_admin'];
}
```

Una vez que el atributo se ha añadido a la lista `appends`, se incluirá tanto en las representaciones del modelo *array* como del modelo JSON. Los atributos en el *array* `appends` respetarán además la configuración de `visible` y `hidden` del modelo.

Fecha de Serialización

Laravel extiende la biblioteca de fecha [Carbon](#) para proporcionar una personalización conveniente del formato de serialización JSON de Carbon. Para personalizar cómo se serializan todas las fechas de Carbon en su aplicación, use el método `Carbon::serializeUsing` method. El método `serializeUsing` acepta un *Closure* que devuelve una representación de la fecha en forma de cadena para la serialización JSON:

```
<?php

namespace App\Providers;

use Illuminate\Support\Carbon;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Carbon::serializeUsing(function ($carbon) {
            return $carbon->format('U');
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

Testing

Testing: Primeros pasos

- # **Introducción**
 - # **Entorno**
 - # **Creando & Ejecutando Tests**
-

Introducción

Laravel ha sido desarrollado teniendo en cuenta el testing. De hecho, el soporte para pruebas con PHPUnit se incluye "Out of the box" (de serie) y un archivo `phpunit.xml` ya está configurado para su aplicación. El framework también incluye métodos convenientes que le permiten probar sus aplicaciones de forma expresa.

De forma predeterminada, el directorio `tests` de su aplicación contiene dos directorios: `Feature` y `Unit`. "Unit test" son pruebas o "tests" que se enfocan en una porción muy pequeña y aislada de su código. De hecho, la mayoría de las pruebas unitarias probablemente se centran en un solo método. Las pruebas de funcionalidad o "feature tests" pueden probar una porción más grande de su código, incluyendo cómo varios objetos interactúan entre sí o incluso una solicitud HTTP completa a un "endpoint" JSON.

El archivo `ExampleTest.php` se incluye los directorios `Feature` y `Unit`. Después de instalar una nueva aplicación Laravel, simplemente ejecute `phpunit` en la línea de comandos para ejecutar sus pruebas.

Entorno

Cuando se ejecutan pruebas a través de `phpunit`, Laravel ajustará automáticamente el entorno de configuración a `testing` debido a las variables de entorno definidas en el fichero `phpunit.xml`. Laravel también configura automáticamente la sesión y la caché en el driver `array` mientras se realizan las pruebas, lo que significa que no se mantendrán los datos de la sesión o la caché durante la ejecución.

Puede definir libremente otros valores de configuración del entorno de prueba según sea necesario. Las variables de entorno `testing` pueden configurarse en el fichero `phpunit.xml`, pero asegúrese de borrar su caché de configuración usando el comando Artisan `config:clear` antes de ejecutar sus pruebas!

Creando & Ejecutando Tests

Para crear un nuevo test, utilizar el comando de Artisan `make:test` :

```
// Create a test in the Feature directory...
php artisan make:test UserTest

// Create a test in the Unit directory...
php artisan make:test UserTest --unit
```

Una vez generada la prueba, puede definir los métodos de prueba como lo haría normalmente con PHPUnit. Para ejecutar las pruebas, simplemente ejecute el comando `phpunit` desde su terminal:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

Si define su propio método `setUp` dentro de una clase de prueba, asegúrese de llamar `parent::setUp()`.

Pruebas HTTP

Introducción

Personalización de los Request Headers

Sesiones / Autenticación

Pruebas de los APIs JSON

Testeo de Subidas de Archivos

Verificaciones disponibles

Introducción

Laravel proporciona una API muy fluida para hacer peticiones HTTP a la aplicación y examinar la salida. Por ejemplo, echemos un vistazo a la prueba que se define a continuación:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');

        $response->assertStatus(200);
    }
}
```

El método `get` realiza una petición `GET` a la aplicación, mientras que el método `assertStatus` verifica que la respuesta devuelta debe tener el código de estado HTTP dado. Además de esta simple verificación, Laravel también contiene una variedad de verificaciones (asserts) para inspeccionar los encabezados de respuesta, contenido, estructura JSON y más.

Personalización de los Request Headers

Se puede usar el método `withHeader` para personalizar los encabezados de la solicitud antes de enviarla a la aplicación. Esto permite agregar los encabezados personalizados que desee a la solicitud:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->withHeaders([
            'X-Header' => 'Value',
        ])->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

Sesiones / Autenticación

Laravel proporciona varios ayudantes para trabajar con la sesión durante el testing HTTP. En primer lugar, se puede establecer los datos de la sesión en un array determinado utilizando el método `withSession`. Esto es útil para cargar la sesión con datos antes de emitir una solicitud a la aplicación:

```

<?php

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $response = $this->withSession(['foo' => 'bar'])
            ->get('/');
    }
}

```

Como es de esperar, uno de los usos más comunes para el uso de la sesión es la de mantener el estado de un usuario autenticado. El helper `actingAs` provee de una manera simple la posibilidad de autenticar a un usuario dado. Por ejemplo, se puede usar un [model factory](#) para generar y autenticar a un usuario:

```

<?php

use App\User;

class ExampleTest extends TestCase
{
    public function testApplication()
    {
        $user = factory(User::class)->create();

        $response = $this->actingAs($user)
            ->withSession(['foo' => 'bar'])
            ->get('/');
    }
}

```

También se puede especificar qué *guard* se debe usar para autenticar al usuario pasando el nombre del mismo como segundo argumento del método `actingAs`:

```
$this->actingAs($user, 'api')
```

Pruebas de los APIs JSON

Laravel también provee algunos helpers para testear API JSON y sus respuestas. Por ejemplo, los métodos `json`, `get`, `post`, `put`, `patch`, y `delete` se pueden usar para emitir solicitudes con varios verbos HTTP. También se pueden pasar fácilmente datos y encabezados a estos métodos. Para comenzar, se tiene una prueba para hacer una solicitud `POST` a `/user` y validar que se devuelvan los datos esperados:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

El método `assertJson` convierte la respuesta a un array y utiliza `PHPUnit::assertArraySubset` para verificar que el array dado existe dentro de la respuesta JSON que fue devuelta por la aplicación. Entonces, si hay otras propiedades en la respuesta JSON, esta prueba aún pasará mientras el fragmento dado esté presente.

Verificación de una coincidencia JSON exacta

Si desea verificar que el array dado coincide de manera **exacta** para el JSON devuelto, debe usar el método `assertExactJson`:

```
<?php

class ExampleTest extends TestCase
{
    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->json('POST', '/user', ['name' => 'Sally']);

        $response
            ->assertStatus(200)
            ->assertExactJson([
                'created' => true,
            ]);
    }
}
```

Pruebas de Subida de Archivos

La clase `Illuminate\Http\UploadedFile` provee un método `fake` que puede usarse para generar archivos vacíos o imágenes para probar. Esto, combinado con el método `fake` de la facade `Storage` simplifica enormemente las pruebas de subidas de archivos. Por ejemplo, puedes combinar esas dos características para probar fácilmente un formulario de subida de un avatar:

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // Assert the file was stored...
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}

```

Personalización de Archivo Falso

Cuando creas archivos usando el método `fake`, puedes especificar el ancho, la altura y el tamaño de la imagen para probar lo mejor posible tus reglas de validación:

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

Además de crear imágenes, puedes crear archivos de cualquier otro formato usando el método `create`:

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

Verificaciones Disponibles

Laravel provee gran variedad de métodos de verificación para tus pruebas [PHPUnit](#). Se puede acceder a dichas verificaciones en la respuesta devuelta desde los métodos de prueba `json`, `get`, `post`, `put` y `delete`:

Método	Descripción
<pre>\$response->assertSuccessful();</pre>	Verifica que la respuesta tiene un código de estado exitoso.
<pre>\$response->assertStatus(\$code);</pre>	Verifica que la respuesta tiene el código.
<pre>\$response->assertRedirect(\$uri);</pre>	Verifica que la respuesta es una redirección a un URI determinado.
<pre>\$response->assertHeader(\$headerName, \$value = null);</pre>	Verifica que la cabecera (header) es presente en la respuesta.
<pre>\$response->assertCookie(\$cookieName, \$value = null);</pre>	Verifica de la respuesta contiene la cookie dada.

Método	Descripción
<pre>\$response->assertPlainCookie(\$cookieName, \$value = null);</pre>	Verifica que la respuesta contiene la cookie dada (sin cifrar).
<pre>\$response->assertCookieExpired(\$cookieName);</pre>	Verifica de la respuesta contiene la cookie dada la misma espirada.
<pre>\$response->assertCookieMissing(\$cookieName);</pre>	Verifica de la respuesta no contiene la cookie dada.
<pre>\$response->assertSessionHas(\$key, \$value = null);</pre>	Verifica que la sesión contiene la información suministrada.
<pre>\$response->assertSessionHasErrors(array \$keys, \$format = null, \$errorBag = 'default');</pre>	Verifica que la sesión contiene un error para el campo dado.
<pre>\$response->assertSessionMissing(\$key);</pre>	Verifica que la sesión no contiene la información dada.

Método	Descripción
<code>\$response->assertJson(array \$data);</code>	Verifica que la respuesta contiene los datos en JSON suministrados.
<code>\$response->assertJsonFragment(array \$data);</code>	Verifica que la respuesta contiene un fragmento de los datos en JSON suministrados.
<code>\$response->assertJsonMissing(array \$data);</code>	Verifica que la respuesta no contiene un fragmento de los datos en JSON suministrados.
<code>\$response->assertExactJson(array \$data);</code>	Verifica que la respuesta coincide exactamente con los datos JSON dados.
<code>\$response->assertJsonStructure(array \$structure);</code>	Verifica que la respuesta tiene la estructura JSON dada.

Método	Descripción
<code>\$response->assertViewIs(\$value);</code>	Verifica que la vista dada devuelta por la ruta.
<code>\$response->assertViewHas(\$key, \$value = null);</code>	Verifica que la respuesta con una vista se le envió dato.
<code>\$response->assertViewHasAll(array \$data);</code>	Verifica que la vista de la respuesta tiene una lista de datos dada.
<code>\$response->assertViewMissing(\$key);</code>	Verifica que la vista que retorna la respuesta no falta una parte de los datos.
<code>\$response->assertSee(\$value);</code>	Verifica que la cadena dada está contenida dentro del texto de la respuesta.
<code>\$response->assertDontSee(\$value);</code>	Verifica que la cadena dada no se encuentra dentro del texto de la respuesta.

Método	Descripción
<pre>\$response->assertSeeText(\$value);</pre>	Verifica que la cadena dada está contenida dentro del texto de la respuesta.
<pre>\$response->assertDontSeeText(\$value);</pre>	Verifica que la cadena dada no está dentro del texto de la respuesta.

Tests En El Navegador (Laravel Dusk)

Introducción

Instalación

- # Usar Otros Navegadores

Comenzando

- # Generar Tests

- # Ejecutar Pruebas

- # Manejo Del Ambiente

- # Crear Navegadores

- # Autenticación

Interacción Con elementos

- # Selectores Dusk

- # Hacer Clic en Enlaces

- # Texto, Valores & Atributos

- # Usar Formularios

- # Adjuntar Archivos

- # Usar El Teclado

- # Usar el Mouse

- # Mantener el Ámbito De Selectores

- # Esperar Por Elementos

- # Making Vue Assertions

Available Assertions

Páginas

- # Generar Páginas

- # Configurar Páginas

- # Navegar Hacia Páginas

- # Shorthand Selectors

- # Métodos de Página

Componentes

- # Generar Componentes

- # Usar Componentes

Integración Contínua

- # Travis CI

- # CircleCI

- # Codeship

Introducción

Laravel Dusk provee una API de automatización y pruebas con el navegador que es fácil de usar. Por defecto, Dusk no requiere que se instale JDK o Selenium en la máquina. En su lugar, Dusk usa una instalación independiente de [ChromeDriver](#). Sin embargo, se tiene la libertad de usar cualquier otro driver compatible con Selenium que se deseé.

Instalación

Para comenzar, se debe incluir en el proyecto la dependencia de Composer `laravel/dusk` :

```
composer require --dev laravel/dusk
```

Una vez que Dusk está instalado, se debe registrar el service provider `Laravel\Browser\BrowserServiceProvider`. Típicamente, esto se realizará automáticamente vía el registro automático de service provider de Laravel.

Si se registra manualmente el service provider de Dusk, **nunca** debería hacerlo en el servidor del ambiente de producción, ya que esto podría permitir a usuarios arbitrarios que se autentiquen contra la aplicación.

Después de instalar el paquete Dusk, ejecute el comando de Artisan `dusk:install` :

```
php artisan dusk:install
```

Se creará un directorio `Browser` dentro del directorio `tests` y contendrá una prueba de ejemplo. Seguidamente, configure la variable de ambiente `APP_URL` en el archivo `.env`. Este valor debe coincidir con la URL que se usa para acceder a la aplicación desde un navegador.

Para ejecutar las pruebas, se usa el comando de Artisan `dusk`. El comando `dusk` acepta cualquier argumento que acepte el comando `phpunit` :

```
php artisan dusk
```

Usar Otros Navegadores

Por defecto, Dusk usa Google Chrome y una instalación independiente de [ChromeDriver](#) para ejecutar las pruebas en el navegador. Sin embargo, se puede iniciar un servidor propio de Selenium y ejecutar los test en cualquier navegador que se desee.

Para comenzar, se abre el archivo `tests/DuskTestCase.php`, el cual es el caso base de tests para la aplicación. Dentro de este archivo, se puede eliminar la llamada al método `startChromeDriver`. Esto hará que Dusk no ejecute automáticamente el ChromeDriver:

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 * @return void  
 */  
  
public static function prepare()  
{  
    // static::startChromeDriver();  
}
```

Seguidamente, se puede modificar el método `driver` para conectarse a la URL y el puerto que se quiera. Adicionalmente, se pueden modificar las "capacidades deseadas" que se deberían pasar al WebDriver:

```
/**  
 * Create the RemoteWebDriver instance.  
 *  
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver  
 */  
  
protected function driver()  
{  
    return RemoteWebDriver::create(  
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()  
    );  
}
```

Comenzando

Generar Pruebas

Para generar una prueba de Dusk, se usa el comando de Artisan `dusk:make` . La prueba que se genera se coloca en el directorio `test/Browser` :

```
php artisan dusk:make LoginTest
```

Ejecutar Pruebas

Para ejecutar las pruebas de browser, se usa el comando de Artisan `dusk` :

```
php artisan dusk
```

El comando `dusk` acepta cualquier argumento que normalmente aceptaría el runner de PHPUnit, permitiendo ejecutar únicamente los test de un cierto `group`, etc:

```
php artisan dusk --group=foo
```

Comenzar Manualmente ChromeDriver

Por defecto, Dusk automáticamente intentará iniciar el ChromeDriver. Si no funciona para un sistema particular, se puede iniciar manualmente el ChromeDriver antes de ejecutar el comando `dusk` . Si se elige iniciar el ChromeDriver manualmente, se debe comentar la siguiente línea del archivo `tests/DuskTestCase.php` :

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 * @return void  
 */  
  
public static function prepare()  
{  
    // static::startChromeDriver();  
}
```

Adicionalmente, si el ChromeDriver se inicia en un puerto distinto al 9515, se deberá modificar el método `driver` de la misma clase:

```
/**  
 * Create the RemoteWebDriver instance.  
 *  
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver  
 */  
  
protected function driver()  
{  
    return RemoteWebDriver::create(  
        'http://localhost:9515', DesiredCapabilities::chrome()  
    );  
}
```

Manejo Del Ambiente

Para forzar a Dusk a usar su propio archivo de ambiente cuando se ejecutan las pruebas, se crea un archivo `.env.dusk.{environment}` en la raíz del proyecto. Por ejemplo, si se va a iniciar el comando `dusk` desde un ambiente `local`, se debería crear un archivo `.env.dusk.local`.

Cuando se ejecutan las pruebas, Dusk hará un respaldo del archivo `.env` y renombrará el ambiente de Dusk a `.env`. Una vez que se hayan completado, el archivo `.env` se restaurará.

Crear Navegadores

Para comenzar, escribamos una prueba que verifique que se puede iniciar sesión en la aplicación. Después de generar una prueba, se puede modificar para navegar a la página de inicio de sesión, ingresar algunas credenciales y hacer clic en el botón "Login". Para crear una instancia del navegador, se llama al método `browse`:

```

<?php

namespace Tests\Browser;

use App\User;
use Tests\DuskTestCase;
use Laravel\Dusk\Chrome;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;

    /**
     * A basic browser test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $user = factory(User::class)->create([
            'email' => 'taylor@laravel.com',
        ]);

        $this->browse(function ($browser) use ($user) {
            $browser->visit('/login')
                ->type('email', $user->email)
                ->type('password', 'secret')
                ->press('Login')
                ->assertPathIs('/home');
        });
    }
}

```

Como se puede ver en el ejemplo anterior, el método `browse` acepta un callback. Dusk pasa automáticamente una instancia del navegador a este callback y es el objeto principal que se usa para interactuar con la aplicación.

Este test se puede usar para probar la pantalla de inicio de sesión que genera el comando de Artisan `make:auth`.

Crear Múltiples Navegadores

En algunas ocasiones puede que se necesiten múltiples navegadores para realizar apropiadamente una prueba. Por ejemplo, se pueden necesitar múltiples navegadores para probar una pantalla de chat que interactue websockets. Para crear múltiples navegadores, simplemente se "pide" por más de uno en la firma del callback que se le da al método

`browser :`

```
$this->browse(function ($first, $second) {
    $first->loginAs(User::find(1))
        ->visit('/home')
        ->waitForText('Message');

    $second->loginAs(User::find(2))
        ->visit('/home')
        ->waitForText('Message')
        ->type('message', 'Hey Taylor')
        ->press('Send');

    $first->waitForText('Hey Taylor')
        ->assertSee('Jeffrey Way');
});
```

Redimensionar Ventanas de Navegador

Se puede usar el método `resize` para ajustar el tamaño de la ventana del navegador:

```
$browser->resize(1920, 1080);
```

El método `maximize` se puede usar para maximizar la ventana del navegador:

```
$browser->maximize();
```

Autenticación

Frecuentemente, se van a probar páginas que requieren autenticación. Se puede usar el método de Dusk `loginAs` para evitar interactuar con la pantalla de inicio de sesión durante cada test. El método `loginAs` acepta un ID de usuario o una instancia del modelo user:

```
$this->browse(function ($first, $second) {  
    $first->loginAs(User::find(1))  
        ->visit('/home');  
});
```

Después de usar el método `loginAs`, la sesión del usuario se mantendrá para todos los test dentro del archivo.

Interacción Con elementos

Selectores Dusk

Elegir buenos selectores CSS para interactuar con los elementos es una de las partes más difíciles cuando se escriben las pruebas con Dusk. Con el tiempo, los cambios en el frontend pueden causar que selectores CSS como los siguientes rompan los tests:

```
// HTML...  
  
<button>Login</button>  
  
// Test...  
  
$browser->click('.login-page .container div > button');
```

Los selectores de Dusk permiten enfocarse en escribir test efectivos en lugar de recordar selectores de CSS. Para definir un selector, se agrega un atributo `dusk` al elemento HTML. Luego, se pone el prefijo `@` en el selector para manipular el elemento adjunto dentro las pruebas de Dusk:

```
// HTML...  
  
<button dusk="login-button">Login</button>  
  
// Test...  
  
$browser->click('@login-button');
```

Hacer Clic en Enlaces

Para hacer clic en un enlace, se puede usar el método `clickLink` en la instancia del navegador. El método `clickLink` hará clic en el enlace que muestre el texto dado:

```
$browser->clickLink($linkText);
```

Este método interactúa con jQuery. Si jQuery no está disponible, Dusk lo inyectará automáticamente a la página para que se encuentre disponible durante la duración de la prueba.

Texto, Valores & Atributos

Recuperar & Establecer Valores

Dusk provee varios métodos para interactuar con el texto que se está mostrando, valores y atributos de los elementos de la página. Por ejemplo, para obtener el "value" de un elemento que coincide con un selector específico, se usa el método `value` :

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

Recuperar Texto

El método `text` se puede usar para recuperar el texto que muestra un elemento que coincide con el selector dado:

```
$text = $browser->text('selector');
```

Recuperar Atributos

Finalmente, el método `attribute` se puede usar para recuperar un atributo del elemento que coincide con el selector dado:

```
$attribute = $browser->attribute('selector', 'value');
```

Usar Formularios

Transcribir Valores

Dusk provee una variedad de métodos para interactuar con formularios y elementos de captura de datos. Primero, veamos un ejemplo de transcribir texto en un campo:

```
$browser->type('email', 'taylor@laravel.com');
```

Nótese que, aunque el método acepta uno si es necesario, no se requiere pasar un selector CSS al método `type`. Si no se provee un selector CSS, Dusk buscará un elemento con el atributo `name` dado. Finalmente, Dusk intentará encontrar un `textarea` con el atributo `name` dado.

Para incluir texto en un campo sin borrar su contenido, se puede usar el método `append`:

```
$browser->type('tags', 'foo')
->append('tags', ', bar, baz');
```

Se puede limpiar el valor de un input usando el método `clear`:

```
$browser->clear('email');
```

Menús Desplegables

Para seleccionar un valor en un menú desplegable, se puede usar el método `select`. Como el método `type`, el método `select` no requiere el uso de un selector CSS. Cuando se pasa un valor al método `select`, se debe enviar el valor subyacente de la opción y no el texto que se muestra:

```
$browser->select('size', 'Large');
```

Se puede seleccionar un valor aleatorio omitiendo el segundo parámetro:

```
$browser->select('size');
```

Casillas De Selección

Para "chequear" una casilla de selección, se puede usar el método `check`. Como en muchos métodos relacionados con el ingreso de datos, no se requiere un selector CSS completo. Si no se puede encontrar una coincidencia exacta con el selector, Dusk buscará una casilla de verificación que coincida con el atributo `name`:

```
$browser->check('terms');

$browser->uncheck('terms');
```

Botones de Radio

Para "seleccionar" una opción de un botón de radio, se puede usar el método `radio`. Como muchos otros métodos de ingreso de datos, no se requiere un selector CSS completo. Si no se puede encontrar una coincidencia exacta con el selector, Dusk buscará un botón de radio que coincida con los atributos `name` y `value`:

```
$browser->radio('version', 'php7');
```

Adjuntar Archivos

El método `attach` se puede usar para adjuntar un archivo a un elemento `file`. Como en muchos métodos relacionados con el ingreso de datos, no se requiere un selector CSS completo. Si no se puede encontrar una coincidencia exacta con el selector, Dusk buscará un campo de archivo que coincida con el atributo `name`:

```
$browser->attach('photo', __DIR__. '/photos/me.png');
```

Usar El Teclado

El método `keys` permite proveer secuencias de entrada más complejas de las que normalmente se permiten con el método `type`. Por ejemplo, se pueden presionar teclas modificadoras mientras se ingresan valores. En este ejemplo, la tecla `shift` se mantiene presionada mientras se ingresa `taylor` en el elemento que coincide con el selector dado. Después de escribir `taylor`, se escribe `otwell` sin ninguna tecla modificadora:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

Se puede incluso enviar una "hot key" al selector primario que contiene la aplicación:

```
$browser->keys('.app', ['{command}', 'j']);
```

Todas las teclas modificadoras estan rodeadas por caracteres {}, y coinciden con las constantes definidas en la clase `Facebook\WebDriver\WebDriverKeys`, la cual se puede encontrar en Github.

Usar el Mouse

Hacer Clic En Elementos

El método `click` se puede usar para hacer "clic" en el elemento que coincide con el selector dado:

```
$browser->click('.selector');
```

Mouseover

El método `mouseover` se puede usar para cuando se necesita mover el mouse sobre el elemento que coincide con el selector:

```
$browser->mouseover('.selector');
```

Arrastrar & Soltar

El método `drag` se puede usar para arrastrar un elelmento que coincide con el selector dado hacia otro elemento:

```
$browser->drag('.from-selector', '.to-selector');
```

O, se puede arrastrar un elemento en una ciertas dirección:

```
$browser->dragLeft('.selector', 10);
$browser->dragRight('.selector', 10);
$browser->dragUp('.selector', 10);
$browser->dragDown('.selector', 10);
```

Mantener el Ámbito De Selectores

En ocasiones, se puede querer ejecutar varias operaciones mientras se mantiene el alcance de dichas operaciones dentro de un selector dado. Por ejemplo, Se puede querer verificar que un determinado texto exista sólo dentro de una tabla y hacer clic en un botón dentro de esa misma tabla. Se puede usar el método `with` para lograr eso. Todas las operaciones que se ejecuten dentro de la función anónima que se le da al método `with` se mantendrán dentro del ámbito del selector original:

```
$browser->with('.table', function ($table) {  
    $table->assertSee('Hello World')  
    ->clickLink('Delete');  
});
```

Esperar Por Elementos

Cuando se prueban aplicaciones que usan JavaScript de manera extensiva, frecuentemente se hace necesario "esperar" que ciertos elementos o datos estén disponibles antes de proseguir con la prueba. Dusk hace que esto sea muy sencillo. Usando una variedad de métodos, se puede esperar hasta que un elemento se haga visible en la página o inclusive esperar hasta que una expresión de JavaScript se evalúe como `true`.

Esperar

Si es necesario pausar la prueba por un determinado número de milisegundos, se usa el método `pause`:

```
$browser->pause(1000);
```

Esperar Por Elementos

El método `wait` se puede usar para pausar la ejecución de una prueba hasta que el elemento que coincide con el selector CSS dado se muestre en la pantalla. Por defecto, esto hará que el test se detenga hasta por un máximo de cinco segundos antes de lanzar una excepción. Si es necesario, se puede pasar un período de espera personalizado como segundo parámetro al método:

```
// Wait a maximum of five seconds for the selector...
$browser->waitFor('.selector');

// Wait a maximum of one second for the selector...
$browser->waitFor('.selector', 1);
```

También se puede esperar hasta que el elemento seleccionado desaparezca de la página:

```
$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);
```

Scoping Selectors When Available

En ocasiones, se quiere esperar por un selector dado e interactuar con el elemento que coincide con el selector. Por ejemplo, se puede querer esperar a que una ventana modal esté disponible y luego presionar el botón "OK" dentro de la modal. El método `whenAvailable` se puede usar en ese caso. Todas las operaciones de elementos que se realicen dentro de la función anónima dada tendrán el ámbito del selector original:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

Esperar Por Texto

El método `waitForText` se puede usar para esperar hasta que el texto dado se muestre en la página:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');

// Wait a maximum of one second for the text...
$browser->waitForText('Hello World', 1);
```

Esperar Por Enlaces

El método `waitForLink` se puede usar para esperar hasta que el texto de enlace dado se muestre en la página:

```
// Wait a maximum of five seconds for the link...
$browser->waitForLink('Create');

// Wait a maximum of one second for the link...
$browser->waitForLink('Create', 1);
```

Esperar Por La Dirección De La Página

Cuando se hace una revisión de una ruta tal como `$browser->assertPathIs('/home')`, esta puede fallar si `window.location.pathname` se está actualizando asíncronamente. Se puede usar el método `waitForLocation` para esperar a que la dirección tenga un valor dado:

```
$browser->waitForLocation('/secret');
```

Esperar Por La Recarga De La Página

Si es necesario hacer verificaciones después de que la página se haya recargado, se usa el método `waitForReload`:

```
$browser->click('.some-action')
->waitForReload()
->assertSee('something');
```

Esperar Por Expresiones JavaScript

Algunas veces se quiere pausar la ejecución de una prueba hasta que una determinada expresión JavaScript se evalúe como `true`. Esto se puede lograr fácilmente usando el método `waitForUntil`. Cuando se pasa una expresión a este método, no es necesario incluir la palabra reservada `return` ni terminar con un punto y coma:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitUntil('App.dataLoaded');

$browser->waitUntil('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitUntil('App.data.servers.length > 0', 1);
```

Esperar Con Una Función Anónima

Muchos de los métodos "wait" en Dusk se basan en el método subyacente `waitUsing`. Se puede usar este método directamente para esperar a que una determinada función anónima retorne `true`. El método `waitUsing` acepta el máximo número de segundos a esperar, el intervalo en el cual la función va a ser evaluada, la función anónima, y un mensaje opcional en caso de fallo:

```
$browser->waitUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

Hacer Verificaciones Vue

Dusk permite hacer verificaciones sobre el estado de los componentes de datos de [Vue](#). Por ejemplo, imaginando que la aplicación contiene el siguiente componente de Vue:

```

// HTML...

<profile dusk="profile-component"></profile>

// Component Definition...

Vue.component('profile', {
  template: '<div>{{ user.name }}</div>',

  data: function () {
    return {
      user: {
        name: 'Taylor'
      }
    };
  }
});

```

Se puede verificar el estado del componente de Vue de la siguiente forma:

```

/**
 * A basic Vue test example.
 *
 * @return void
 */
public function testVue()
{
  $this->browse(function (Browser $browser) {
    $browser->visit('/')
      ->assertVue('user.name', 'Taylor', '@profile-component');
  });
}

```

Verificaciones Disponibles

Dusk provee una variedad de verificaciones que se pueden hacer contra la aplicación. Todas las verificaciones disponibles están documentadas en la siguiente tabla:

Assertion	Descripción
\$browser->assertTitle(\$title)	Verificar que el título de la página coincida con el texto dado.
\$browser->assertTitleContains(\$title)	Verificar que el título de la página contenga el texto dado.
\$browser->assertPathBeginsWith(\$path)	Verificar que la ruta URL actual comience con la ruta dada.
\$browser->assertPathIs('/home')	Verificar que la ruta actual coincida con la ruta dada.
\$browser->assertPathIsNot('/home')	Verificar que la ruta actual no coincide con la ruta dada.
\$browser->assertRouteIs(\$name, \$parameters)	Verificar que la URL actual coincida con la URL de la ruta con nombre dada.
\$browser->assertQueryStringHas(\$name, \$value)	Assert the given query string parameter is present and has a given value.
\$browser->assertQueryStringMissing(\$name)	Assert the given query string parameter is missing.
\$browser->assertHasQueryStringParameter(\$name)	Assert that the given query string parameter is present.
\$browser->assertHasCookie(\$name)	Verificar que la cookie dada está presente.
\$browser->assertCookieMissing(\$name)	Verificar que la cookie dada no está presente.
\$browser->assertCookieValue(\$name, \$value)	Verificar que una cookie tiene un valor determinado.

Assertion	Descripción
\$browser->assertPlainCookieValue(\$name, \$value)	Verificar que una cookie no encriptada tiene un valor determinado.
\$browser->assertSee(\$text)	Verificar que el texto dado está presente en la página.
\$browser->assertDontSee(\$text)	Verificar que el texto dado no está presente en la página.
\$browser->assertSeeIn(\$selector, \$text)	Verificar que el texto dado está presente dentro de un selector.
\$browser->assertDontSeeIn(\$selector, \$text)	Verificar que el texto dado no está presente dentro del selector.
\$browser->assertSourceHas(\$code)	Verificar que determinado código fuente está presente en la página.
\$browser->assertSourceMissing(\$code)	Verificar que determinado código fuente no está presente en la página.
\$browser->assertSeeLink(\$linkText)	Verificar que el enlace dado está presente en la página.
\$browser->assertDontSeeLink(\$linkText)	Verificar que el enlace dado no está presente en la página.
\$browser->assertInputValue(\$field, \$value)	Verificar que la casilla de texto dada tiene un valor determinado.
\$browser->assertInputValueIsNot(\$field, \$value)	Verificar que la casilla de texto no tiene el valor dado.

Assertion	Descripción
\$browser->assertChecked(\$field)	Verificar que la casilla de verificación dada está marcada.
\$browser->assertNotChecked(\$field)	Verificar que la casilla de verificación dada no está marcada.
\$browser->assertRadioSelected(\$field, \$value)	Verificar que el botón de radio dado está seleccionado.
\$browser->assertRadioNotSelected(\$field, \$value)	Verificar que el botón de radio dado no está seleccionado.
\$browser->assertSelected(\$field, \$value)	Verificar que determinado menú desplegable dado tiene seleccionado el valor dado.
\$browser->assertNotSelected(\$field, \$value)	Verificar que determinado menú desplegable no tiene seleccionado el valor dado.
\$browser->assertSelectHasOptions(\$field, \$values)	Verificar que los valores en la matriz dada están disponibles para ser seleccionados.
\$browser->assertSelectMissingOptions(\$field, \$values)	Verificar que los valores en la matriz dada no están disponibles para ser seleccionados.
\$browser->assertSelectHasOption(\$field, \$value)	Verificar que el valor dado está disponible para ser seleccionado en el campo dado.
\$browser->assertValue(\$selector, \$value)	Verificar que el elemento que coincide con el selector dado tiene el valor dado.

Assertion	Descripción
\$browser->assertVisible(\$selector)	Verificar que el elemento que coincide con el selector es visible.
\$browser->assertMissing(\$selector)	Verificar que el elemento que coincide con el selector no es visible.
\$browser->assertDialogOpened(\$message)	Verificar que se ha abierto una ventana de diálogo JavaScript con el mensaje dado.
\$browser->assertVue(\$property, \$value, \$component)	Verificar que la propiedad data de un determinado componente Vue coincide con el valor dado.
\$browser->assertVueIsNot(\$property, \$value, \$component)	Verificar que la propiedad data de un componente Vue determinado no coincide con el valor dado.

Páginas

En ocasiones, la prueba requiere que se hagan en secuencia varias acciones complicadas. Esto puede hacer que las pruebas sean difíciles de leer y entender. Las Páginas permiten definir acciones expresivas que luego pueden ser realizadas en una página determinada usando un solo método. Las Páginas también permiten definir atajos hacia selectores comunes en la aplicación o en una sola página.

Generar Páginas

Para generar un objeto de página, se usa el comando de Artisan `dusk:page` . Todos estos objetos se colocarán en el directorio `tests/Browser/Pages` :

```
php artisan dusk:page Login
```

Configurar Páginas

Por defecto, las páginas tienen tres métodos: `url`, `assert`, y `elements`. Discutiremos ahora los métodos `url` y `assert`. El método `elements` se discute con [más detalle abajo](#).

El Método `url`

El método `url` debería retornar la ruta a la URL que representa la página. Dusk usa esta URL cuando navega la página en el browser:

```
/**  
 * Get the URL for the page.  
 *  
 * @return string  
 */  
  
public function url()  
{  
    return '/login';  
}
```

El Método `assert`

El método `assert` puede tomar cualquier verificación necesaria para verificar que el navegador está realmente en la página dada. Completar este método no es necesario, sin embargo, se tiene la libertad de hacer estas verificaciones si se lo desea. Estas verificaciones se ejecutan automáticamente cuando se navega en la página:

```
/**  
 * Assert that the browser is on the page.  
 *  
 * @return void  
 */  
  
public function assert(Browser $browser)  
{  
    $browser->assertPathIs($this->url());  
}
```

Navegar Hacia Páginas

Una vez que la página ha sido configurada, se puede navegar usando el método `visit`:

```
use Tests\Browser\Pages\Login;
```

```
$browser->visit(new Login);
```

En ocasiones uno ya se encuentra en una página dada y se necesita "cargar" los selectores y métodos en el contexto actual de la prueba. Esto es común cuando se presiona un botón y se es redirigido a una página determinada sin navegar explícitamente hacia ella. En esta situación, se puede usar el método `on` para cargar la página:

```
use Tests\Browser\Pages\CreatePlaylist;
```

```
$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

Shorthand Selectors

The `elements` method of pages allows you to define quick, easy-to-remember shortcuts for any CSS selector on your page. For example, let's define a shortcut for the "email" input field of the application's login page:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

Now, you may use this shorthand selector anywhere you would use a full CSS selector:

```
$browser->type('@email', 'taylor@laravel.com');
```

Global Shorthand Selectors

After installing Dusk, a base `Page` class will be placed in your `tests/Browser/Pages` directory. This class contains a `siteElements` method which may be used to define global shorthand selectors that should be available on every page throughout your application:

```
/**  
 * Get the global element shortcuts for the site.  
 *  
 * @return array  
 */  
  
public static function siteElements()  
{  
    return [  
        '@element' => '#selector',  
    ];  
}
```

Métodos de Página

In addition to the default methods defined on pages, you may define additional methods which may be used throughout your tests. For example, let's imagine we are building a music management application. A common action for one page of the application might be to create a playlist. Instead of re-writing the logic to create a playlist in each test, you may define a `createPlaylist` method on a page class:

```
<?php

namespace Tests\Browser\Pages;

use Laravel\Dusk\Browser;

class Dashboard extends Page
{
    // Other page methods...

    /**
     * Create a new playlist.
     *
     * @param \Laravel\Dusk\Browser $browser
     * @param string $name
     * @return void
     */
    public function createPlaylist(Browser $browser, $name)
    {
        $browser->type('name', $name)
            ->check('share')
            ->press('Create Playlist');
    }
}
```

Once the method has been defined, you may use it within any test that utilizes the page. The browser instance will automatically be passed to the page method:

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```

Componentes

Components are similar to Dusk’s “page objects”, but are intended for pieces of UI and functionality that are re-used throughout your application, such as a navigation bar or notification window. As such, components are not bound to specific URLs.

Generar Componentes

To generate a component, use the `dusk:component` Artisan command. New components are placed in the `test/Browser/Components` directory:

```
php artisan dusk:component DatePicker
```

As shown above, a "date picker" is an example of a component that might exist throughout your application on a variety of pages. It can become cumbersome to manually write the browser automation logic to select a date in dozens of tests throughout your test suite. Instead, we can define a Dusk component to represent the date picker, allowing us to encapsulate that logic within the component:

```
<?php

namespace Tests\Browser\Components;

use Laravel\Dusk\Browser;
use Laravel\Dusk\Component as BaseComponent;

class DatePicker extends BaseComponent
{
    /**
     * Get the root selector for the component.
     *
     * @return string
     */
    public function selector()
    {
        return '.date-picker';
    }

    /**
     * Assert that the browser page contains the component.
     *
     * @param  Browser  $browser
     * @return void
     */
    public function assert(Browser $browser)
    {
        $browser->assertVisible($this->selector());
    }

    /**
     * Get the element shortcuts for the component.
     *
     * @return array
     */
    public function elements()
    {
        return [
            '@date-field' => 'input.datepicker-input',
            '@month-list' => 'div > div.datepicker-months',
            '@day-list' => 'div > div.datepicker-days',
        ];
    }
}
```

```
}

/**
 * Select the given date.
 *
 * @param \Laravel\Dusk\Browser $browser
 * @param int $month
 * @param int $year
 * @return void
 */

public function selectDate($browser, $month, $year)
{
    $browser->click('@date-field')
        ->within('@month-list', function ($browser) use ($month) {
            $browser->click($month);
        })
        ->within('@day-list', function ($browser) use ($day) {
            $browser->click($day);
        });
}
}
```

Usar Componentes

Once the component has been defined, we can easily select a date within the date picker from any test. And, if the logic necessary to select a date changes, we only need to update the component:

```

<?php

namespace Tests\Browser;

use Tests\DuskTestCase;
use Laravel\Dusk\Browser;
use Tests\Browser\Components\DatePicker;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class ExampleTest extends DuskTestCase
{
    /**
     * A basic component test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->browse(function (Browser $browser) {
            $browser->visit('/')
                ->within(new DatePicker, function ($browser) {
                    $browser->selectDate(1, 2018);
                })
                ->assertSee('January');
        });
    }
}

```

Integración Contínua

Travis CI

To run your Dusk tests on Travis CI, we will need to use the "sudo-enabled" Ubuntu 14.04 (Trusty) environment. Since Travis CI is not a graphical environment, we will need to take some extra steps in order to launch a Chrome browser. In addition, we will use

`php artisan serve` to launch PHP's built-in web server:

```
sudo: required
dist: trusty

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222 http://localhost:8000
  - php artisan serve &

script:
  - php artisan dusk
```

CircleCI

CircleCI 1.0

If you are using CircleCI 1.0 to run your Dusk tests, you may use this configuration file as a starting point. Like TravisCI, we will use the `php artisan serve` command to launch PHP's built-in web server:

```
dependencies:
  pre:
    - curl -L -o google-chrome.deb https://dl.google.com/linux/direct/google-chrome-stable
    - sudo dpkg -i google-chrome.deb
    - sudo sed -i 's|HERE/chrome|"|HERE/chrome\\" --disable-setuid-sandbox\|g' /opt/google/chrome/google-chrome
    - rm google-chrome.deb

  test:
    pre:
      - "./vendor/laravel/dusk/bin/chromedriver-linux":
        background: true
      - cp .env.testing .env
      - "php artisan serve":
        background: true

    override:
      - php artisan dusk
```

CircleCI 2.0

If you are using CircleCI 2.0 to run your Dusk tests, you may add these steps to your build:

```
version: 2
jobs:
  build:
    steps:

      - run: sudo apt-get install -y libssqlite3-dev
      - run: cp .env.testing .env
      - run: composer install -n --ignore-platform-reqs
      - run: npm install
      - run: npm run production
      - run: vendor/bin/phpunit

      - run:
          name: Start Chrome Driver
          command: ./vendor/laravel/dusk/bin/chromedriver-linux
          background: true

      - run:
          name: Run Laravel Server
          command: php artisan serve
          background: true

      - run:
          name: Run Laravel Dusk Tests
          command: php artisan dusk
```

Codeship

To run Dusk tests on [Codeship](#), add the following commands to your Codeship project. Of course, these commands are simply a starting point and you are free to add additional commands as needed:

```
phpenv local 7.1
cp .env.testing .env
composer install --no-interaction
nohup bash -c "./vendor/laravel/dusk/bin/chromedriver-linux 2>&1 &" 
nohup bash -c "php artisan serve 2>&1 & && sleep 5
php artisan dusk
```


Pruebas de Base de Datos

- # Introducción
- # Generating Factories
- # Restablecer la base de datos después de cada test
- # Writing Factories
 - # Factory States
- # Using Factories
 - # Creación de modelos
 - # Persistencia de modelos
 - # Relaciones
- # Verificaciones disponibles

Introducción

Laravel proporciona una variedad de herramientas útiles para hacer más fácil el *testing* de aplicaciones basadas en bases de datos. En primer lugar se puede usar el helper `assertDatabaseHas` para confirmar que los datos existentes en la base de datos coinciden con un conjunto determinado de criterios. Por ejemplo, si se desea verificar que hay un registro en la tabla `users` con el valor `email` de `sally@example.com`, se puede hacer lo siguiente:

```
public function testDatabase()  
{  
    // Make call to application...  
  
    $this->assertDatabaseHas('users', [  
        'email' => 'sally@example.com'  
    ]);  
}
```

También se puede usar el helper `assertDatabaseMissing` para verificar que los datos no existen en la base de datos.

Desde luego, el método `assertDatabaseHas` y otros *helpers* como el se usan a conveniencia. Se es libre de usar cualquiera de los métodos de verificación integrados de PHPUnit para complementar las pruebas.

Generating Factories

Para crear una *factory*, se usa el [comando de Artisan `make:factory`](#):

```
php artisan make:factory PostFactory
```

La nueva *factory* será puesta in su directorio `database/factories`.

La opción `--model` puede ser usada para indicar el nombre del modelo creado por la *factory*. Esta opción se pre-llenará al archivo de *factory* generado con el modelo dado:

```
php artisan make:factory PostFactory --model=Post
```

Restablecimiento de la Base de Datos tras Testing

A menudo es útil restablecer su base de datos después de cada prueba para que los datos de una prueba anterior no interfieran con las pruebas posteriores. El trait `RefreshDatabase` toma el enfoque más óptimo para migrar su base de datos de prueba dependiendo de si está usando una base de datos in-memory o una base de datos tradicional. Simplemente use el trait en su clase de prueba y todo se manejará por usted:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->get('/');

        // ...
    }
}
```

Escritura de Fábricas

Al realizar la prueba, podría necesitar insertar unos pocos registros en su base de datos antes de ejecutar su prueba. En lugar de especificar manualmente el valor de cada columna cuando crea estos datos de prueba, Laravel le permite definir un set de configuración de atributos para cada uno de sus [Eloquent models](#) usando *factories* modelo. Para empezar, chequée el archivo `database/factories/UserFactory.php` en su aplicación. Listo para su uso, este archivo contiene una definición de *factory*:

```
use Faker\Generator as Faker;

(factory->define(App\User::class, function (Faker $faker) {
    static $password;

    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => $password ?: $password = bcrypt('secret'),
        'remember_token' => str_random(10),
    ];
});
```

Dentro del Closure, que sirve como definición de *factory*, puede devolver los valores de prueba predeterminados de todos los atributos en el modelo. El Closure recibirá una instancia de la librería PHP [Faker](#), la cual le permite generar a conveniencia varios tipos de datos al azar para probar.

También puede crear archivos de *factory* adicionales por cada modelo para una mejor organización. Por ejemplo, podría crear archivos `UserFactory.php` y `CommentFactory.php` dentro de su directorio `database/factories`. Todos los archivos dentro del directorio `factories` se cargarán automáticamente por Laravel.

Estados de Fábrica

Los estados le permiten definir modificaciones discretas que pueden ser aplicadas a sus *factories* modelo en cualquier combinación. Por ejemplo, su modelo `User` puede tener un estado `delinquent` que modifica uno de sus valores de atributo predeterminado. Puede definir sus transformaciones de estado usando el método `state`. Para estados simples, puede pasar un array de modificaciones de atributos:

```
$factory->state(App\User::class, 'delinquent', [
    'account_status' => 'delinquent',
]);
```

Si su estado requiere cálculo o una instancia `$faker`, puede usar un Closure para calcular las modificaciones de atributo del estado:

```
$factory->state(App\User::class, 'address', function ($faker) {
    return [
        'address' => $faker->address,
    ];
});
```

Uso de Fábricas

Creación de Modelos

Una vez que ha definido sus *factories*, puede usar la función global `factory` en sus pruebas o archivos poblados para generar instancias modelo. Entonces, echemos un vistazo a algunos ejemplos de creación de modelos. Primero, usaremos el método `make` para crear modelos pero sin guardarlos en la base de datos:

```
public function testDatabase()
{
    $user = factory(App\User::class)->make();

    // Use model in tests...
}
```

También puede crear una Colección de algunos modelos o crear modelos de un tipo dado:

```
// Create three App\User instances...
$users = factory(App\User::class, 3)->make();
```

Aplicación de Estados

Puede también aplicar cualquiera de sus `states` a sus modelos. Si desea aplicar múltiples trasformaciones de estado a los modelos, deberá especificar el nombre de cada estado que desee aplicar:

```
$users = factory(App\User::class, 5)->states('delinquent')->make();

$users = factory(App\User::class, 5)->states('premium', 'delinquent')->make();
```

Reemplazo de Atributos

Si desea reemplazar alguno de los valores por defecto de sus modelos, puede pasar un array de valores al método `make`. Solo los valores especificados serán reemplazados mientras que el resto de los valores permanecen establecidos en sus valores predeterminados según lo especificado por la *factory*:

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

Persistencia de Modelos

El método `create` no solo crea las instancias modelo sino que también los guarda en su base de datos usando el método de Eloquent `save`:

```
public function testDatabase()
{
    // Create a single App\User instance...
    $user = factory(App\User::class)->create();

    // Create three App\User instances...
    $users = factory(App\User::class, 3)->create();

    // Use model in tests...
}
```

Puede reemplazar atributos sobre el modelo pasando un array al método `create`:

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

Relaciones

En este ejemplo, adjuntaremos una relación a unos modelos creados. Al usar el método `create` para crear múltiples modelos, una [collection instance](#) de Eloquent es devuelta, permitiéndole usar cualquiera de las funciones convenientes proporcionadas por la colección, tal y como `each`:

```
$users = factory(App\User::class, 3)
    ->create()
    ->each(function ($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
});
```

Relaciones & y Atributos de Closures

Puede también enfocar las relaciones a modelos usando atributos Closure en sus definiciones de *factory*. Por ejemplo, si desea crear una nueva instancia de `User` al crear un `Post`, puede hacer lo siguiente:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        }
    ];
});
```

Estos Closures también reciben el array de atributo evaluado de la *factory* que los define:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => function () {
            return factory(App\User::class)->create()->id;
        },
        'user_type' => function (array $post) {
            return App\User::find($post['user_id'])->type;
        }
    ];
});
```

Verificaciones Disponibles

Laravel proporciona varias afirmaciones de base de datos de sus pruebas [PHPUnit](#):

Método	Descripción
<code>\$this->assertDatabaseHas(\$table, array \$data);</code>	Asegúrate que una tabla en la base de datos contiene los datos dados.
<code>\$this->assertDatabaseMissing(\$table, array \$data);</code>	Asegúrate que una tabla en la base de datos no contiene datos dados.
<code>\$this->assertSoftDeleted(\$table, array \$data);</code>	Asegúrate de que el registro dado haya sido borrado en forma suave.

Mocking

- # [Introducción](#)
 - # [Fake Bus](#)
 - # [Fake Event](#)
 - # [Fake Mail](#)
 - # [Notificaciones](#)
 - # [Fake Queue](#)
 - # [Fake Storage](#)
 - # [Facades](#)
-

Introducción

Al probar las aplicaciones de Laravel, es posible que desee "simular" ciertos aspectos de su aplicación para que no se ejecuten realmente durante una prueba determinada. Por ejemplo, cuando se prueba un controlador que distribuye un evento, es posible que desee simular los detectores de eventos para que no se ejecuten realmente durante la prueba. Esto le permite probar únicamente la respuesta HTTP del controlador sin preocuparse por la ejecución de los detectores de eventos, ya que los detectores de eventos se pueden probar en su propio caso de prueba.

Laravel proporciona ayudantes para simular eventos, trabajos, y facades listos para usarse. Estos ayudantes proporcionan principalmente una capa conveniente sobre Mockery para que no tenga que realizar manualmente llamadas de método de Mockery complicadas. Por supuesto, es libre de usar [Mockery](#) o la PHPUnit para crear sus propios simuladores o espías.

Fake Bus

Como alternativa a la simulación, puede usar el método de facade `Bus fake` para evitar que se envíen trabajos. Cuando usa fakes, las afirmaciones se realizan después de que se ejecuta el código bajo prueba:

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Bus;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Bus::fake();

        // Perform order shipping...

        Bus::assertDispatched(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was not dispatched...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}

```

Fake Event

Como alternativa a la simulación, puede usar el método de facade `fake Bus` para evitar que se ejecuten los detectores de eventos. A continuación, puede afirmar que los eventos fueron enviados e incluso inspeccionar los datos que recibieron. Cuando usa fakes, las afirmaciones se realizan después de que se ejecuta el código bajo prueba:

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Events\OrderShipped;
use App\Events\OrderFailedToShip;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    /**
     * Test order shipping.
     */
    public function testOrderShipping()
    {
        Event::fake();

        // Perform order shipping...

        Event::assertDispatched(OrderShipped::class, function ($e) use ($order) {
            return $e->order->id === $order->id;
        });

        Event::assertNotDispatched(OrderFailedToShip::class);
    }
}

```

Fake Mail

Puede usar el método de facade `fake Mail` para evitar que se envíe el correo. A continuación, puede afirmar que se enviaron [mailables](#) a los usuarios e incluso inspeccionar los datos que ellos recibieron. Cuando usa fakes, las afirmaciones se realizan después de que se ejecuta el código bajo prueba:

```

<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Mail::fake();

        // Perform order shipping...

        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {
            return $mail->order->id === $order->id;
        });

        // Assert a message was sent to the given users...
        Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {
            return $mail->hasTo($user->email) &&
                $mail->hasCc('...') &&
                $mail->hasBcc('...');
        });

        // Assert a mailable was sent twice...
        Mail::assertSent(OrderShipped::class, 2);

        // Assert a mailable was not sent...
        Mail::assertNotSent(AnotherMailable::class);
    }
}

```

Si está haciendo queue en formato mailable para la entrega en segundo plano, debe usar el método `assertQueued` en vez de `assertSent`:

```
Mail::assertQueued(...);  
Mail::assertNotQueued(...);
```

Notificación Fake

Debe usar el método de facade `Notification fake` para evitar que se envíen las notificaciones. A continuación, puede afirmar que se enviaron las `notifications` a los usuarios e incluso inspeccionar los datos que ellos recibieron. Cuando usa fakes, las afirmaciones se realizan después de que se ejecuta el código bajo prueba:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Notification::fake();

        // Perform order shipping...

        Notification::assertSentTo(
            $user,
            OrderShipped::class,
            function ($notification, $channels) use ($order) {
                return $notification->order->id === $order->id;
            }
        );

        // Assert a notification was sent to the given users...
        Notification::assertSentTo(
            [$user], OrderShipped::class
        );

        // Assert a notification was not sent...
        Notification::assertNotSentTo(
            [$user], AnotherNotification::class
        );
    }
}
```

Fake Queue

Como alternativa a la simulación, puede usar el método de facade `fake` `Queue` para evitar que los trabajos se mantengan en cola. Puede entonces afirmar que los trabajos se enviaron a la cola e incluso los datos que recibieron. Cuando usa fakes, las afirmaciones se realizan después de que se ejecuta el código bajo prueba:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Jobs\ShipOrder;
use Illuminate\Support\Facades\Queue;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Queue::fake();

        // Perform order shipping...

        Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {
            return $job->order->id === $order->id;
        });

        // Assert a job was pushed to a given queue...
        Queue::assertPushedOn('queue-name', ShipOrder::class);

        // Assert a job was pushed twice...
        Queue::assertPushed(ShipOrder::class, 2);

        // Assert a job was not pushed...
        Queue::assertNotPushed(AnotherJob::class);
    }
}
```

Fake Storage

El método de facade `fake Storage` le permite generar fácilmente un disco fake que, combinado con las utilidades de generación de archivos de la clase `UploadedFile`, simplifica enormemente la prueba de cargas de archivos. Por ejemplo:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Http\UploadedFile;
use Illuminate\Support\Facades\Storage;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $response = $this->json('POST', '/avatar', [
            'avatar' => UploadedFile::fake()->image('avatar.jpg')
        ]);

        // Assert the file was stored...
        Storage::disk('avatars')->assertExists('avatar.jpg');

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

Por defecto, el método `fake` eliminará todos los archivos en su directorio temporal. Si desea conservar estos archivos, puede usar a cambio el método "persistentFake".

Facades

A diferencia de las llamadas de métodos estáticos tradicionales, [facades](#) puede ser simulado. Esto le proporciona una gran ventaja por sobre los métodos estáticos tradicionales y le ofrece la misma capacidad de prueba que tendría si estuviera usando la inyección de dependencia. Al realizar la prueba, puede con frecuencia querer simular una llamada a un facade Laravel en uno de sus controladores. Por ejemplo, considere la acción del siguiente controlador:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

Podemos simular la llamada al facade `Cache` usando el método `shouldReceive`, el cual devolverá una instancia de una simulación [Mockery](#). Desde que facades en realidad se resuelve y maneja por el [service container](#) Laravel, tienen mucha más capacidad de prueba que una clase estática típica. Por ejemplo, simulemos nuestra llamada al método de facade `Cache get`:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Support\Facades\Cache;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class UserControllerTest extends TestCase
{
    public function testGetIndex()
    {
        Cache::shouldReceive('get')
            ->once()
            ->with('key')
            ->andReturn('value');

        $response = $this->get('/users');

        // ...
    }
}
```

No debería simular el facade `Request`. En cambio, pase la entrada que desee a los métodos de ayuda de HTTP, como `get` y `post` al ejecutar su prueba. Del mismo modo, en lugar de simular el facade `Config`, simplemente llame al método `Config::set` en sus pruebas.

Paquetes oficiales

Laravel Cashier

Introducción

Configuración

- # Stripe
- # Braintree
- # Configuración de divisas

Suscripciones

- # Crear suscripciones
- # Comprobar el estado de una suscripción
- # Cambiar planes
- # Cantidad de la suscripción
- # Impuestos de la suscripción
- # Cancelar suscripciones
- # Reactivar suscripciones
- # Actualizar tarjetas de crédito

Periodos de prueba de suscripciones

- # Con tarjeta de crédito
- # Sin tarjeta de crédito

Gestionar Stripe Webhooks

- # Definir gestores de eventos para webhooks
- # Suscripciones fallidas

Gestionar Braintree Webhooks

- # Definir gestores de eventos para webhooks
- # Suscripciones fallidas

Cargos únicos

Facturas

- # Generar facturas en PDF

Introducción

Laravel Cashier provee una interfaz fluida y expresiva para gestionar los servicios de cobro por suscripción de [Stripe](#) y [Braintree](#). Gestiona prácticamente todo lo referente al código de facturación de suscripciones que tanto se teme. Además de la gestión de suscripciones

básicas, Cashier maneja cupones, cambios de suscripciones, "cuantías", períodos de gracia de cancelación e incluso genera facturas PDF.

Si únicamente se está aplicando cargos "por compra" y no se ofrece un servicio de suscripción recurrente, no se debería usar Cashier. En su lugar, utilizar los SDKs de Stripe y Braintree directamente.

Configuración

Stripe

Composer

Primero, añadir el paquete de Cashier para Stripe a las dependencias:

```
composer require "laravel/cashier": "~7.0"
```

Migraciones de BD

Antes de utilizar Cashier, será necesario [preparar la base de datos](#). Se añadirán algunas columnas a la tabla `users` y se creará una nueva tabla `subscriptions` para almacenar todo lo relativo a las suscripciones de los clientes:

```

Schema::table('users', function ($table) {
    $table->string('stripe_id')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('stripe_id');
    $table->string('stripe_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});


```

Una vez que se han creado las migraciones, ejecutar el comando de Artisan `migrate`.

Modelo facturable - *billable*

A continuación, añadir el *trait* `Billable` a la definición del modelo. Este *trait* incluye varios métodos que permitirán realizar las tareas de facturación más comunes, como crear suscripciones, aplicar cupones y actualizar la información de una tarjeta de crédito:

```

use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}

```

Claves API

Finalmente hay que configurar las claves API de Stripe en el archivo de configuración `services.php`. Se pueden obtener desde el panel de control de Stripe:

```
'stripe' => [
    'model'  => App\User::class,
    'key'    => env('STRIPE_KEY'),
    'secret' => env('STRIPE_SECRET'),
],
```

Braintree

Consideraciones

Para muchas operaciones, las implementaciones de Cashier para Braintree y Stripe son las mismas. Ambos servicios proveen facturación por suscripción con tarjeta de crédito, pero Braintree soporta además pagos via PayPal. Además, Braintree no cuenta con algunas de las características de Stripe. Es importante tener esto en cuenta a la hora de decidir si utilizar Stripe o Braintree:

- Braintree soporta PayPal, Stripe no.
- Braintree no soporta los métodos `increment` y `decrement` en las suscripciones. Es una limitación de Braintree, no de Cashier.
- Braintree no soporta descuentos basados en porcentajes. De nuevo una limitación de Braintree, no de Cashier.

Composer

Primero, añadir el paquete de Cashier para Braintree a las dependencias:

```
composer require "laravel/cashier-braintree": "~2.0"
```

Service Provider

A continuación, registrar el archivo `Laravel\Cashier\CashierServiceProvider` como service provider en el archivo de configuración `config/app.php`:

```
Laravel\Cashier\CashierServiceProvider::class
```

Cupones de crédito

Antes de utilizar Cashier con Braintree, es necesario definir un descuento `plan-credit` en el panel de control de Braintree. Este descuento se utilizará para prorratear las suscripciones que cambian desde la suscripción anual a la mensual o viceversa.

La cantidad configurada en el panel de control de Braintree puede ser cualquier valor, ya que Cashier reemplazará la cantidad definida con la cantidad configurada cada vez que se aplique el cupón. Este cupón es necesario ya que Braintree no soporta de forma nativa el prorratoeo de suscripciones entre frecuencias de suscripción.

Migraciones de BD

Antes de usar Cashier, será necesario [preparar la base de datos](#). Se añadirán algunas columnas a la tabla `users` y se creará una nueva tabla `subscriptions` para almacenar todo lo relativo a las suscripciones de los clientes:

```
Schema::table('users', function ($table) {
    $table->string('braintree_id')->nullable();
    $table->string('paypal_email')->nullable();
    $table->string('card_brand')->nullable();
    $table->string('card_last_four')->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function ($table) {
    $table->increments('id');
    $table->integer('user_id');
    $table->string('name');
    $table->string('braintree_id');
    $table->string('braintree_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

Una vez que se han creado las migraciones, simplemente ejecutar el comando de Artisan `migrate`.

Modelo facturable - `billable`

A continuación, añadir el trait `Billable` a la definición del modelo:

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

Claves API

A continuación, debe configurar las siguientes opciones en el archivo `services.php` :

```
'braintree' => [
    'model' => App\User::class,
    'environment' => env('BRAINTREE_ENV'),
    'merchant_id' => env('BRAINTREE_MERCHANT_ID'),
    'public_key' => env('BRAINTREE_PUBLIC_KEY'),
    'private_key' => env('BRAINTREE_PRIVATE_KEY'),
],
```

Ahora hay que añadir las siguientes llamadas al SDK de Braintree al método `boot` del `AppServiceProvider` :

```
\Braintree\Configuration::environment(config('services.braintree.environment'));
\Braintree\Configuration::merchantId(config('services.braintree.merchant_id'));
\Braintree\Configuration::publicKey(config('services.braintree.public_key'));
\Braintree\Configuration::privateKey(config('services.braintree.private_key'));
```

Configuración de divisas

La moneda por defecto de Cashier son dólares Estadounidenses (USD). Se puede cambiar la moneda por defecto llamando al método `Cashier::useCurrency` desde el método `boot` de uno de los *service providers*. El método `useCurrency` acepta dos cadenas como parámetros: la moneda y su símbolo:

```
use Laravel\Cashier\Cashier;

Cashier::useCurrency('eur', '€');
```

Suscripciones

Crear suscripciones

Para crear una suscripción, es necesaria una instancia de un modelo *Billable*, que normalmente será una instancia de `App\User`. Una vez que se ha recuperado la instancia del modelo, se puede utilizar el método `newSubscription` para crear la suscripción del modelo:

```
$user = User::find(1);

$user->newSubscription('main', 'premium')->create($stripeToken);
```

El primer argumento pasado al método `newSubscription` corresponde al nombre de la suscripción. Si la aplicación únicamente ofrece una sola suscripción, se puede llamar a este método `main` o `primary`. El segundo argumento es el plan específico de Stripe/Braintree al que se está suscribiendo el usuario. Este valor debe corresponder con el identificador del plan en Stripe o Braintree.

El método `create`, el cual acepta el token de una tarjeta de crédito/fondo de Stripe, comenzará la suscripción así como actualizará la base de datos con el ID del cliente y otra información relevante de facturación.

Datos de usuario adicionales

Para especificar información adicional sobre el cliente, se puede pasar como segundo argumento al método `create`:

```
$user->newSubscription('main', 'monthly')->create($stripeToken, [
    'email' => $email,
]);
```

Para saber más sobre los campos adicionales que soporta Stripe o Braintree, comprobar la [documentación sobre creación de clientes](#) de Stripe o la sección correspondiente de [Braintree](#).

Cupones

Para aplicar cupones al crear suscripciones, se puede utilizar el método `withCoupon`:

```
$user->newSubscription('main', 'monthly')
    ->withCoupon('code')
    ->create($stripeToken);
```

Comprobar el estado de una suscripción

Una vez que un usuario se ha suscrito a la aplicación, se puede comprobar el estado de esta suscripción de varias formas. En primer lugar, el método `subscribed` retornará `true` si el usuario cuenta con una suscripción activa, incluso si el usuario se encuentra en el periodo de prueba:

```
if ($user->subscribed('main')) {
    //
}
```

El método `subscribed` también es un gran candidato para participar en los [middleware de ruta](#), permitiendo filtrar el acceso a las rutas y controladores basadas en el estado de las suscripciones de los usuarios:

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed('main')) {
        // This user is not a paying customer...
        return redirect('billing');
    }

    return $next($request);
}
```

Para determinar si un usuario se encuentra todavía en el periodo de prueba, utilizar el método `onTrial`. Este método puede resultar útil para mostrar un mensaje de advertencia al usuario de que se encuentra aún en este periodo:

```
if ($user->subscription('main')->onTrial()) {
    //
}
```

El método `subscribedToPlan` se puede utilizar para determinar si el usuario está suscrito a un plan concreto de Stripe/Braintree basado en el ID del plan. En este ejemplo, se determinará si la suscripción `main` del usuario está activamente suscrita al plan `monthly` :

```
if ($user->subscribedToPlan('monthly', 'main')) {  
    //  
}
```

Estado de suscripción cancelada

Para saber si un usuario fue alguna vez un usuario suscrito, pero ha cancelado su suscripción, existe el método `cancelled` :

```
if ($user->subscription('main')->cancelled()) {  
    //  
}
```

Se puede comprobar si el usuario ha cancelado la suscripción, pero se encuentra todavía en su "periodo de gracia" hasta que la suscripción caduque completamente. Por ejemplo, si un usuario cancela la suscripción el 5 de marzo y estaba prevista a caducar el 10 de marzo, el usuario se encuentra en el "periodo de gracia" hasta el 10 de marzo. Tenga en cuenta que el método `subscribed` todavía retornará `true` en este momento:

```
if ($user->subscription('main')->onGracePeriod()) {  
    //  
}
```

Cambiar planes

Después de que un usuario se suscriba a la aplicación, puede ocasionalmente requerir un cambio a otro plan de suscripción nuevo. Para cambiar un usuario a otra suscripción, pasar el identificador del plan al método `swap` :

```
$user = App\UserIdentity::find(1);  
  
$user->subscription('main')->swap('provider-plan-id');
```

Si el usuario está en periodo de prueba, se mantendrá. Además, si existe una "cuantía" para la suscripción, esa cantidad se mantendrá.

Para cambiar un plan y además cancelar el periodo de prueba en el que se encuentre el usuario, se puede usar el método `skipTrial` :

```
$user->subscription('main')
    ->skipTrial()
    ->swap('provider-plan-id');
```

Cuantía de la suscripción

Las cantidades de suscripción solo están soportadas en la versión de Stripe de Cashier. Braintree no tiene una característica que corresponda con "cantidad" como Stripe.

A veces las suscripciones dependen de la "cuantía". Por ejemplo, la aplicación puede cargar \$10 al mes **por usuario** en una cuenta. Para incrementar o disminuir la cuantía de la suscripción, utilizar los métodos `incrementQuantity` y `decrementQuantity` :

```
$user = User::find(1);

$user->subscription('main')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();

// Subtract five to the subscription's current quantity...
$user->subscription('main')->decrementQuantity(5);
```

Alternativamente, se puede establecer una cuantía específica utilizando el método `updateQuantity` :

```
$user->subscription('main')->updateQuantity(10);
```

El método `noProrate` se puede utilizar para actualizar la cuantía de la suscripción sin prorratear los cargos:

```
$user->subscription('main')->noProrate()->updateQuantity(10);
```

Para más información sobre las cuantías de suscripción, consultar la [documentación de Stripe](#).

Impuestos de la suscripción

Para definir el porcentaje de impuestos que paga un usuario en una suscripción, utilizar el método `taxPercentage` del modelo *billable* y retornar un valor numérico entre 0 y 100 con no mas de 2 decimales.

```
public function taxPercentage() {
    return 20;
}
```

El método `taxPercentage` permite sumar impuestos en una base modelo-a-modo, la cual puede ser útil para usuarios que abarcan varios países y reglas de impuestos.

El método `taxPercentage` aplica únicamente a los cargos por suscripción. Si se utiliza Cashier para "pagos únicos", será necesario especificar los impuestos en el momento del cobro.

Cancelar suscripciones

Para cancelar una suscripción, simplemente utilizar el método `cancel` sobre la suscripción del usuario:

```
$user->subscription('main')->cancel();
```

Cuando se cancela una suscripción, Cashier establece el valor de `ends_at` de forma automática en la base de datos. Esta columna se utiliza para saber cuando el método `subscribed` debe devolver `false`. Por ejemplo, si un usuario cancela una suscripción el 1 de marzo, pero la suscripción no estaba programada para terminar hasta el 5 de marzo, el método `subscribed` continuaría retornando `true` hasta el 5 de marzo.

Se puede saber si un usuario ha cancelado su suscripción pero aun está en el "periodo de gracia" utilizando el método `onGracePeriod`:

```
if ($user->subscription('main')->onGracePeriod()) {  
    //  
}
```

Para cancelar una suscripción inmediatamente, llamar al método `cancelNow` en la suscripción del usuario:

```
$user->subscription('main')->cancelNow();
```

Reactivar suscripciones

Si un usuario ha cancelado su suscripción y desea reanudarla, utilizar el método `resume`. El usuario **debe** estar en su *periodo de gracia* para reanudar una suscripción:

```
$user->subscription('main')->resume();
```

Si el usuario cancela una suscripción y luego la reanuda antes de que haya caducado completamente, no será facturado inmediatamente. En cambio, la suscripción será simplemente reactivada y será facturado en su ciclo de facturación original.

Actualizar tarjetas de crédito

El método `updateCard` se puede usar para actualizar la información de la tarjeta de crédito. Este método acepta un *token* de Stripe y asignará la nueva tarjeta como fuente de facturación por defecto:

```
$user->updateCard($stripeToken);
```

Periodos de prueba de suscripciones

Con tarjeta de crédito

Si desea ofrecer períodos de prueba para sus clientes mientras que consigue su información de pago, debe utilizar el método `trialDays` al crear las suscripciones:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')
    ->trialDays(10)
    ->create($stripeToken);
```

Este método establecerá la fecha de finalización del periodo de prueba de prueba en el registro de la suscripción de la base de datos, así como configurará Stripe/Braintree de no facturar al cliente hasta pasada esa fecha.

Si la suscripción del cliente no se cancela antes del final del periodo de prueba, se les facturará tan pronto como esta expire, por lo que asegúrese de notificar a sus usuarios que su periodo de prueba ha terminado.

Se puede determinar si un usuario está en periodo de prueba con el método `onTrial` de la instancia del usuario o el método `onTrial` de la instancia de la suscripción. Los dos ejemplos a continuación son idénticos:

```
if ($user->onTrial('main')) {
    //
}

if ($user->subscription('main')->onTrial()) {
    //
}
```

Sin tarjeta de crédito

Si se desea ofrecer periodos de prueba sin almacenar previamente el método de pago del usuario, simplemente establecer el valor de la columna `trial_ends_at` en el registro del usuario a la fecha de finalización deseada. Esto se hace normalmente en el proceso de registro:

```
$user = User::create([
    // Populate other user properties...
    'trial_ends_at' => Carbon::now()->addDays(10),
]);
```

Asegúrese de añadir un date mutator para `trial_ends_at` en la definición del modelo.

Cashier se refiere a este tipo de periodo de prueba como "periodo de prueba genérico", puesto que no está asociado a ninguna suscripción. El método `onTrial` de la instancia `User` retornará `true` si la fecha actual no es posterior a `trial_ends_at`:

```
if ($user->onTrial()) {  
    // User is within their trial period...  
}
```

También se puede utilizar el método `onGenericTrial` si se desea conocer específicamente que el usuario está en el periodo de prueba "genérico" y no ha creado una suscripción todavía:

```
if ($user->onGenericTrial()) {  
    // User is within their "generic" trial period...  
}
```

Una vez que se está listo para crear una suscripción para el usuario, se puede utilizar el método `newSubscription` normalmente:

```
$user = User::find(1);  
  
$user->newSubscription('main', 'monthly')->create($stripeToken);
```

Gestionar Stripe Webhooks

Ambos, Stripe y Braintree pueden notificar a la aplicación de una gran variedad de eventos a través de `webhooks`. Para gestionar los *Stripe webhooks*, definir una ruta que apunte al `webhook controller` de Cashier. Este controlador gestionará todas las peticiones entrantes de `webhooks` y las lanzará al método del controlador apropiado:

```
Route::post(  
    'stripe/webhook',  
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'  
>);
```

Una vez que se ha registrado la ruta, asegúrese de configurar la *webhook URL* en la configuración del panel de control de Stripe.

Por defecto, este controlador gestionará automáticamente la cancelación de suscripciones que tienen demasiados cargos fallidos (tal y como se defina en la configuración de Stripe); sin embargo, como pronto se descubrirá, se puede extender este controlador para gestionar cualquier evento *webhook* que sea necesario.

Webhooks & protección CSRF

Puesto que los *webhooks* de Stripe necesitan sortear la [protección CSRF](#) de Laravel, asegúrese de añadir la URI como una excepción en el *middleware* `VerifyCsrfToken` o listar la ruta fuera del grupo de *middleware* `web` :

```
protected $except = [
    'stripe/*',
];
```

Definir gestores de eventos para *webhooks*

Cashier gestiona automáticamente la cancelación de suscripciones cuando hay cargos fallidos, pero si hay algún *webhook* de Stripe adicional que se desee gestionar, simplemente hay que extender el *WebhookController*. Los nombres de los métodos deben corresponder a la convención de Stripe, específicamente, los métodos deben contener el prefijo `handle` y utilizar la nomenclatura "camel case" del *Webhook* de Stripe a gestionar. Por ejemplo, para gestionar el *webhook* `invoice.payment_succeeded`, se debe añadir el método `handleInvoicePaymentSucceeded` al controlador:

```
<?php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Stripe webhook.
     *
     * @param array $payload
     * @return Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }
}
```

Suscripciones fallidas

¿Qué ocurre si expira la tarjeta de un cliente? Sin problema - Cashier incluye un *WebhookController* que cancela la suscripción del cliente. Lo único que hay que hacer es apuntar una ruta al controlador:

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

¡Eso es todo! Los cobros fallidos serán capturados y gestionados por el controlador. El controlador cancelará la suscripción del cliente cuando Stripe determine que la suscripción ha fallado (normalmente tras tres intentos de cobro fallidos).

Gestionar Braintree Webhooks

Ambos, Stripe y Braintree pueden notificar a la aplicación de una gran variedad de eventos a través de *webhooks*. Para gestionar los *webhooks* de Braintree, defina una ruta que apunte al *webhook controller* de Cashier. Este controlador gestionará todas las peticiones entrantes de *webhooks* y las lanzará al método del controlador apropiado:

```
Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

Una vez que se ha registrado la ruta, asegúrese de configurar la URL del *webhook* en la configuración del panel de control de Braintree.

Por defecto, este controlador gestionará automáticamente la cancelación de suscripciones que tienen demasiados cargos fallidos (tal y como se defina en la configuración de Braintree); sin embargo, como pronto descubrirá, se puede extender este controlador para gestionar cualquier evento *webhook* que sea necesario.

Webhooks & protección CSRF

Puesto que los *webhooks* de Braintree necesitan sortear la [protección CSRF](#) de Laravel, asegúrese de añadir la URI como una excepción en el *middleware* `VerifyCsrfToken` o listar la ruta fuera del grupo de *middleware* `web` :

```
protected $except = [
    'braintree/*',
];
```

Definir gestores de eventos para webhooks

Cashier gestiona automáticamente la cancelación de suscripciones cuando hay cargos fallidos, pero si hay algún *webhook* de Stripe adicional que se desee gestionar, simplemente hay que extender el *WebhookController*. El nombre del método debe corresponderse con la convención de Cashier, los métodos deben contener el prefijo `handle` y en modo "camel case" el nombre del *webhook* de Braintree que desee gestionar. Por ejemplo, para gestionar el *webhook* `dispute_opened`, debe añadir el siguiente método al controlador

`handleDisputeOpened` :

```

<?php

namespace App\Http\Controllers;

use Braintree\WebhookNotification;
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle a Braintree webhook.
     *
     * @param  WebhookNotification  $webhook
     * @return Response
     */
    public function handleDisputeOpened(WebhookNotification $notification)
    {
        // Handle The Event
    }
}

```

Suscripciones fallidas

¿Qué pasa si se caduca una tarjeta de crédito? Cashier incluye un controlador de *Webhook* que cancelará automáticamente la suscripción. Simplemente hay que apuntar la ruta al controlador:

```

Route::post(
    'braintree/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);

```

¡Eso es todo! Los cobros fallidos serán capturados y gestionados por el controlador. El controlador cancelará la suscripción del cliente cuando Braintree determine que la suscripción ha fallado (normalmente tras tres intentos fallidos de cobro). No olvidar: será necesario configurar la URI del *webhook* en el panel de control de Braintree.

Cargos únicos

Cargo simple

Al utilizar Stripe, el método `charge` acepta la cantidad a cobrar en el **menor denominador de divisa utilizado por la aplicación**. Sin embargo, al utilizar Braintree, se debe pasar la cantidad exacta al método `charge`:

Para realizar un cargo "único" contra la tarjeta de crédito de un usuario suscrito, se puede utilizar el método `charge` de la instancia del modelo billable.

```
// Stripe Accepts Charges In Cents...
$user->charge(100);

// Braintree Accepts Charges In Dollars...
$user->charge(1);
```

El método `charge` acepta un *array* como segundo argumento, permitiendo pasar cualquier opción que se desee a la creación del cargo de Stripe/Braintree. Consultar la documentación de Stripe o Braintree para conocer las opciones disponibles en la creación de cargos:

```
$user->charge(100, [
    'custom_option' => $value,
]);
```

El método `charge` lanzará una excepción si el cargo falla. Si el cargo es satisfactorio, se retornará la respuesta Stripe/Braintree completa:

```
try {
    $response = $user->charge(100);
} catch (Exception $e) {
    //
}
```

Cargo con factura

A veces es necesario hacer un cargo único y además generar una factura para el cargo para poder ofrecerle el recibo en PDF al cliente. El método `invoiceFor` permite hacer justo eso. Por ejemplo, se va a generar una factura para un cliente de \$5.00 como "cargo único":

```
// Stripe Accepts Charges In Cents...
$user->invoiceFor('One Time Fee', 500);

// Braintree Accepts Charges In Dollars...
$user->invoiceFor('One Time Fee', 5);
```

La factura será cobrada inmediatamente contra la tarjeta de crédito del usuario. El método `invoiceFor` acepta además un *array* como tercer argumento, permitiendo pasar tantas opciones como sea necesario a la creación del cargo de Stripe/Braintree subyacente:

```
$user->invoiceFor('One Time Fee', 500, [
    'custom-option' => $value,
]);
```

El método `invoiceFor` creará una factura de Stripe que re-intentará si se producen fallos de facturación. Si no se desea que las facturas re-intenten cobrar los cargos fallidos, es necesario cerrarlos utilizando el API de Stripe tras el primer cargo fallido.

Facturas

Se puede obtener fácilmente un *array* de facturas del modelo billable utilizando el método `invoices` :

```
$invoices = $user->invoices();

// Include pending invoices in the results...
$invoices = $user->invoicesIncludingPending();
```

Cuando se listan las facturas de un cliente, se pueden utilizar los métodos *helper* para mostrar la información relevante a la facturación. Por ejemplo, se puede listar cada factura en una tabla, permitiendo al usuario descargar cualquiera de ellas:

```

<table>
    @foreach ($invoices as $invoice)
        <tr>

            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
            <td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>
        </tr>
    @endforeach
</table>

```

Generar facturas en PDF

Desde una ruta o controlador, utilizar el método `downloadInvoice` para generar la descarga de una factura en PDF. Este método automáticamente genera la respuesta HTTP apropiada para enviar la descarga al navegador:

```

use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor'  => 'Your Company',
        'product' => 'Your Product',
    ]);
});

```

Envoy Task Runner

Introducción

Instalación

Definir tareas

Configuración

Variables

Historias – *Stories*

Varios servidores

Ejecutar tareas

Confirmar la ejecución de tareas

Notificaciones

Slack

Introducción

[Laravel Envoy](#) proporciona una sintaxis minimalista para la definición de las tareas más comunes que se ejecutan en sus servidores remotos. Usando el estilo de sintaxis de Blade, puede configurar fácilmente tareas de despliegue (*deployment*), comandos de Artisan y mucho más. Actualmente, Envoy sólo es compatible con los sistemas operativos Mac y Linux.

Instalación

En primer lugar, instalar Envoy usando el comando `global` de Composer:

```
composer global require laravel/envoy
```

Puesto que las librerías globales de Composer pueden causar conflictos de versión de paquetes en algunas ocasiones, se puede considerar usar `cgr`, el cual sería un reemplazo de `composer global require`. Las instrucciones de instalación de la librería `cgr` las puede [encontrar en GitHub](#).

Asegúrese de situar el directorio `~/.composer/vendor/bin` en su PATH, permitiendo que se encuentre el ejecutable de `envoy` al ejecutar el comando `envoy` en su terminal.

Actualizar Envoy

También puede usar Composer para mantener su instalación de Envoy actualizada. Al ejecutar el comando `composer global update` se actualizarán todos sus paquetes Composer instalados globalmente:

```
composer global update
```

Definir tareas

Todas las tareas de Envoy deben definirse en un archivo llamado `Envoy.blade.php` ubicado en la raíz del proyecto. He aquí un ejemplo para empezar:

```
@servers(['web' => ['user@192.168.1.1']])

@task('foo', ['on' => 'web'])
    ls -la
@endtask
```

Como se puede observar, se define un *array* `@servers` al inicio del archivo, lo que le permite hacer referencia a estos servidores en la opción `on` de las declaraciones de tareas. Dentro de las declaraciones `@task`, se debe colocar el código Bash que se deberá ejecutar en el servidor cuando se ejecute la tarea.

Puede forzar un *script* para ejecutarlo localmente especificando la dirección IP del servidor como `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Configuración

A veces, puede que se tenga que ejecutar código PHP antes de ejecutar las tareas en Envoy. Se puede utilizar la directiva `@setup` para declarar variables y ejecutar PHP general antes de que se ejecuten cualquiera de sus otras tareas:

```
@setup
$now = new DateTime();

$environment = isset($env) ? $env : "testing";
@endsetup
```

Si se precisa de otros ficheros PHP antes de ejecutar la tarea, podrá utilizar la directiva `@include` al inicio de su fichero `Envoy.blade.php`:

```
@include('vendor/autoload.php')

@task('foo')
    # ...
@endtask
```

Variables

Si es necesario, puede pasar valores de opciones a las tareas de Envoy usando la línea de comandos:

```
envoy run deploy --branch=master
```

Podrá acceder a las opciones en sus tareas a través de la sintaxis "echo" de Blade. Desde luego, también podrá usar las declaraciones `if` y bucles de sus tareas. Por ejemplo, para verificar la presencia de una variable `$branch` antes de ejecutar el comando `git pull`:

```

@servers(['web' => '192.168.1.1'])

@task('deploy', ['on' => 'web'])
    cd site

    @if ($branch)
        git pull origin {{ $branch }}
    @endif

    php artisan migrate
@endtask

```

Historias – *Stories*

Las historias agrupan un conjunto de tareas con un nombre único y conveniente, lo que le permite agrupar tareas pequeñas enfocadas en grandes tareas. Por ejemplo, una *story* `deploy` puede ejecutar las tareas `git` y `composer` listando los nombres de las mismas en su definición:

```

@servers(['web' => '192.168.1.1'])

@story('deploy')
    git
    composer
@endstory

@task('git')
    git pull origin master
@endtask

@task('composer')
    composer install
@endtask

```

Una vez que la historia ha sido escrita, puede ejecutarse como una tarea normal:

```
envoy run deploy
```

Varios servidores

Envoy permite ejecutar tareas en varios servidores fácilmente. Para empezar, añada los servidores adicionales a la declaración `@servers`. Cada servidor debe tener un nombre único. Una vez que haya definido sus servidores adicionales, liste cada uno de los servidores en el *array* de tareas `on`:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2']])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Ejecución paralela

Por defecto, las tareas se ejecutarán sobre cada servidor en serie. En otras palabras, una tarea terminará de correr en el primer servidor antes de proceder a ejecutarla en el segundo. Si prefiere ejecutar la tarea en varios servidores en paralelo, agregue `parallel` a la declaración:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])

@task('deploy', ['on' => ['web-1', 'web-2'], 'parallel' => true])
    cd site
    git pull origin {{ $branch }}
    php artisan migrate
@endtask
```

Ejecutar tareas

Para ejecutar una tarea o historia definida en el fichero `Envoy.blade.php`, ejecute el comando de Envoy `run`, pasando el nombre de la tarea o historia a ejecutar. Envoy ejecutará la tarea y mostrará la salida de los servidores donde se está corriendo:

```
envoy run task
```

Confirmar la ejecución de tareas

Si desea que se le solicite confirmación antes de correr una tarea determinada en sus servidores, debe agregar la directiva `confirm` a su declaración de tarea. Esta opción es particularmente útil para operaciones destructivas:

```
@task('deploy', ['on' => 'web', 'confirm' => true])
  cd site
  git pull origin {{ $branch }}
  php artisan migrate
@endtask
```

Notificaciones

Slack

Envoy también admite el envío de notificaciones a [Slack](#) después de ejecutar cada tarea. La directiva `@slack` acepta un URL del *hook* de Slack y un nombre de canal. Puede obtener la URL del *webhook* mediante la creación de una integración "Incoming WebHooks" en su panel de control de Slack. Debe pasar la URL del *webhook* completo a la directiva `@slack` :

```
@finished
  @slack('webhook-url', '#bots')
@endfinished
```

Se puede proporcionar uno de los siguientes como el argumento de canal:

- Para enviar una notificación a un canal: `#canal`
- Para enviar una notificación a un usuario: `@usuario`

Laravel Horizon

Introducción

Instalación

Configuración

Autenticación del Dashboard

Ejecutar Horizon

Desplegar Horizon

Etiquetas

Notificaciones

Métricas

Introducción

Horizon proporciona un hermoso *dashboard* y una configuración controlada por código para sus colas de Redis gestionadas por Laravel. Horizon permite monitorear de forma sencilla las métricas clave de su sistema de colas (*queues*), tales como el rendimiento de trabajos (*jobs*), el tiempo de ejecución y las fallas del trabajos.

Toda la configuración se almacena en un único y sencillo archivo de configuración, permitiendo que permanezca en el lugar de control del código donde todo su equipo puede colaborar.

Instalación

Debido a su uso de señales de proceso asincrónicas, Horizon requiere PHP 7.1+.

Puede usar Composer para instalar su proyecto Laravel:

```
composer require laravel/horizon
```

Después de instalar Horizon, publique sus recursos (*assets*) usando el comando Artisan `vendor:publish` :

```
php artisan vendor:publish --provider="Laravel\Horizon\HorizonServiceProvider"
```

Configuración

Después de publicar los *assets* de Horizon, su fichero de configuración principal se ubicará en `config/horizon.php`. Este fichero de configuración permite configurar las opciones de sus *workers* y cada opción incluye una descripción de su propósito, por lo tanto, asegúrese de explorar este fichero completamente.

Opciones de balance

Horizon le permite elegir entre tres estrategias de balance: `simple`, `auto`, y `false`. La estrategia `simple`, la cual es la predeterminada, divide los trabajos entrantes uniformemente entre los procesos:

```
'balance' => 'simple',
```

La estrategia `auto` ajusta el número de procesos de trabajo por cola basada en la actual carga de trabajo de la cola. Por ejemplo, si su cola de `notifications` tiene 1,000 trabajos en espera mientras su cola `render` está vacía, Horizon distribuirá más trabajos a su cola de `notifications` hasta que esté vacía. Cuando la opción `balance` se establece a `false`, se utilizará el funcionamiento predeterminado de Laravel, el cual procesa colas en el orden que se estableció en la configuración.

Autenticación del Dashboard

Horizon expone un *dashboard* en `/horizon`. Por defecto, solo podrá acceder a este *dashboard* en el entorno `local`. Para definir una política de acceso más específica para el *dashboard*, deberá usar el método `Horizon::auth`. El método `auth` acepta un *Callback* la cual deberá retornar `true` o `false`, indicando si el usuario debería tener acceso al *dashboard* de Horizon:

```
Horizon::auth(function ($request) {
    // return true / false;
});
```

Ejecutar Horizon

Una vez que haya configurado sus *workers* en el fichero de configuración `config/horizon.php` , puede iniciar Horizon usando el comando Artisan `horizon` . Este único comando iniciará todos los *workers* configurados:

```
php artisan horizon
```

Puede pausar el proceso de Horizon e instruirlo para continuar procesando trabajos utilizando `horizon:pause` y los comandos Artisan `horizon:continue` :

```
php artisan horizon:pause
```

```
php artisan horizon:continue
```

Puede finalizar el proceso principal de Horizon utilizando el comando Artisan `horizon:terminate` . Todos los trabajos que Horizon esté ejecutando en ese momento se completarán y a continuación se detendrá su ejecución:

```
php artisan horizon:terminate
```

Desplegar Horizon

Si está desplegando Horizon en un servidor online, debe configurar un monitor de procesos para monitorear para monitorear el comando `php artisan horizon` y re-ejecutarlo si se detiene de forma inesperada. Al desplegar código nuevo a su servidor, tendrá que detener el proceso principal de Horizon para que pueda ser reiniciado por su monitor de procesos y recibir los cambios de código.

Puede finalizar el proceso principal de Horizon utilizando el comando Artisan `horizon:terminate` . Todos los trabajos que Horizon esté ejecutando en ese momento se completarán y a continuación se detendrá su ejecución:

```
php artisan horizon:terminate
```

Configuración de Supervisor

Si está utilizando el monitor de procesos *Supervisor* para gestionar su proceso `horizon` , el siguiente fichero de configuración deberá ser suficiente:

```
[program:horizon]
process_name=%(program_name)s
command=php /home/forge/app.com/artisan horizon
autostart=true
autorestart=true
user=forge
redirect_stderr=true
stdout_logfile=/home/forge/app.com/horizon.log
```

Si no se encuentra cómodo gestionando sus propios servidores, considere utilizar Laravel Forge. Forge provee servidores PHP 7+ con todo lo que necesita para ejecutar aplicaciones modernas y robustas de Laravel con Horizon.

Etiquetas

Horizon permite asignar "etiquetas" a trabajos, incluyendo *mailables*, *event broadcasting*, notificaciones y colas de *event listeners*. De hecho, Horizon etiquetará de forma inteligente y automática la mayoría de los trabajos dependiendo de los modelos Eloquent que estén relacionados con el trabajo. Por ejemplo, revise el siguiente *job*:

```
<?php

namespace App\Jobs;

use App\Video;
use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Foundation\Bus\Dispatchable;

class RenderVideo implements ShouldQueue
{
    use Dispatchable, InteractsWithQueue, Queueable, SerializesModels;

    /**
     * The video instance.
     *
     * @var \App\Video
     */
    public $video;

    /**
     * Create a new job instance.
     *
     * @param \App\Video $video
     * @return void
     */
    public function __construct(Video $video)
    {
        $this->video = $video;
    }

    /**
     * Execute the job.
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

```
    }  
}
```

Si este trabajo se encuentra en una cola con una instancia `App\Video`, por ejemplo, que tiene un `id` con valor `1`, automáticamente recibirá la etiqueta `App\Video:1`. Esto se debe a que Horizon examina las propiedades del trabajo buscando modelos Eloquent. Si se encuentran modelos Eloquent, Horizon inteligentemente etiquetará el trabajo utilizando el nombre de la clase y la clave principal del modelo:

```
$video = App\Video::find(1);  
  
App\Jobs\RenderVideo::dispatch($video);
```

Etiquetado manual

Si prefiere definir las etiquetas de forma manual, puede definir un método `tags` en la clase:

```
class RenderVideo implements ShouldQueue  
{  
    /**  
     * Get the tags that should be assigned to the job.  
     *  
     * @return array  
     */  
    public function tags()  
    {  
        return ['render', 'video:'.$this->video->id];  
    }  
}
```

Notificaciones

Note: Antes de utilizar las notificaciones, deberá agregar el paquete Composer `guzzlehttp/guzzle` a su proyecto. Cuando configure Horizon a enviar notificaciones SMS, también deberá revisar los requerimientos para el driver de notificación Nexmo.

Si prefiere recibir notificaciones cuando una de las colas tenga un tiempo de espera muy largo, se pueden utilizar los métodos `Horizon::routeMailNotificationsTo` , `Horizon::routeSlackNotificationsTo` y `Horizon::routeSmsNotificationsTo` . Podrá ejecutar estos métodos desde el `AppServiceProvider` de su aplicación:

```
Horizon::routeMailNotificationsTo('example@example.com');
Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');
Horizon::routeSmsNotificationsTo('15556667777');
```

Configurar umbrales de tiempo de espera de la notificación

Podrá configurar cuántos segundos se consideran una "larga espera" dentro del fichero de configuración `config/horizon.php` . La opción de configuración `waits` dentro de este fichero, permite controlar el umbral de espera por cada combinación conexión/cola:

```
'waits' => [
    'redis:default' => 60,
],
```

Métricas

Horizon incluye un *dashboard* de métricas el cual proporciona información sobre sus trabajos y tiempos de espera de las colas y rendimiento. Con el fin de añadir datos a este *dashboard*, deberá configurar el comando de Artisan `snapshot` para ejecutarse cada cinco minutos a través del `scheduler` de su aplicación:

```
/**
 * Define the application's command schedule.
 *
 * @param \Illuminate\Console\Scheduling\Schedule $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```


Autenticación API (Passport)

- # Introducción
- # Instalación
 - # Inicio rápido Frontend
 - # Despliegue de *Passport*
- # Configuración
 - # Duración de token
- # Emisión de tokens de acceso
 - # Gestión de clientes
 - # Solicitud de tokens
 - # Refrescando Tokens
- # Tokens de concesión de contraseña
 - # Crear un cliente de concesión de contraseña
 - # Solicitud de Tokens
 - # Solicitud de todos los ámbitos
- # Fichas de donaciones implícitas
- # Tokens de concesión de credenciales de cliente
- # Tokens de acceso personal
 - # Crear un cliente de acceso personal
 - # Administrar tokens de acceso personal
- # Protección de Rutas
 - # A través de *middleware*
 - # Pasando el token de acceso
- # Alcance Token
 - # Definiendo scopes
 - # Asignación de scopes a tokens
 - # Verificación de scopes
- # Consumir su API con JavaScript
- # Eventos
- # Testing

Introducción

Laravel ya facilita la autenticación a través de los formularios de inicio de sesión tradicionales, pero ¿qué pasa con las APIs? Las APIs generalmente usan tokens para autenticar usuarios y no mantienen el estado de la sesión entre las solicitudes. Laravel hace que la autenticación de API sea muy sencilla utilizando Laravel Passport, que proporciona una implementación completa del servidor OAuth2 para su aplicación en cuestión de minutos. Passport está construido sobre el servidor [League OAuth2](#) que es mantenido por Alex Bilbie.

Esta documentación asume que usted ya está familiarizado con OAuth2. Si usted no sabe nada acerca de OAuth2, considere familiarizarse con la terminología y características generales de OAuth2 antes de continuar.

Instalación

Para empezar, instale Passport a través del gestor de paquetes Composer:

```
composer require laravel/passport
```

El proveedor de servicio Passport registra su propio directorio de migración de base de datos con el framework, por lo que debe migrar su base de datos después de registrar el proveedor. Las migraciones de Passport crearán las tablas que su aplicación necesita para almacenar clientes y tokens de acceso:

```
php artisan migrate
```

Si no va a utilizar las migraciones predeterminadas de Passport, debe llamar al método `Passport::ignoreMigrations` en el método `register` de su `AppServiceProvider`. Puede exportar las migraciones predeterminadas utilizando `php artisan vendor:publish --tag=passport-migrations`.

A continuación, debe ejecutar el comando `passport:install`. Este comando creará las claves de cifrado necesarias para generar tokens de acceso seguro. Además, el comando creará clientes "de acceso personal" y "de concesión de contraseña" que se utilizarán para generar tokens de acceso:

```
php artisan passport:install
```

Después de ejecutar el comando, agregue el *trait* `Laravel\Passport\HasApiTokens` en su modelo `App\User`. Este *trait* proporcionará algunos *helpers* a su modelo que le permitirán inspeccionar el token y alcances del usuario autenticado:

```
<?php

namespace App;

use Laravel\Passport\HasApiTokens;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;
}
```

A continuación, debe llamar al método `Passport::routes` dentro del método `boot` de su `AuthServiceProvider`. Este método registrará las rutas necesarias para emitir tokens de acceso y revocará los tokens de acceso, clientes y tokens de acceso personal:

```

<?php

namespace App\Providers;

use Laravel\Passport\Passport;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Passport::routes();
    }
}

```

Por último, en el archivo de configuración `config/auth.php`, debe cambiar la opción `driver` de `api` a `passport`. Esto le indicará a su aplicación el uso de Passport's `TokenGuard` al autenticar las solicitudes de API entrantes:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
    'api' => [
        'driver' => 'passport',
        'provider' => 'users',
    ],
],
```

Inicio rápido Frontend

Para poder utilizar los componentes Passport de Vue, debe estar utilizando el framework JavaScript Vue. Estos componentes también utilizan el framework CSS Bootstrap. Sin embargo, incluso si no está utilizando estas herramientas, los componentes sirven como una referencia valiosa para su propia implementación frontend.

Passport se entrega con una API JSON que puede utilizar para permitir a sus usuarios crear clientes y tokens de acceso personal. Sin embargo, puede llevar mucho tiempo codificar un frontend para interactuar con estas APIs. Por lo tanto, Passport también incluye componentes [Vue](#) preconstruidos que puede utilizar como ejemplo de implementación o punto de partida para su propia implementación.

Para publicar los componentes Vue de Passport, utilice el comando Artisan `vendor:publish` :

```
php artisan vendor:publish --tag=passport-components
```

Los componentes publicados se colocarán en el directorio `resources/assets/js/components` . Una vez que los componentes se hayan publicado, debe registrarlos en su archivo `resources/assets/js/app.js` :

```
Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue')
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue')
);
```

Después de registrar los componentes, asegúrese de ejecutar `npm run dev` para recomilar sus recursos. Una vez que haya recompilado sus recursos, puede colocar los componentes en una de las plantillas de su aplicación para empezar a crear clientes y tokens de acceso personal:

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Despliegue de *Passport*

Al implementar Passport en sus servidores de producción por primera vez, es probable que tenga que ejecutar el comando `passport:keys`. Este comando genera las claves de encriptación que necesita Passport para generar el token de acceso. Las claves generadas no suelen mantenerse en el control de origen:

```
php artisan passport:keys
```

Configuración

Duración de token

De forma predeterminada, Passport emite tokens de acceso de larga duración que nunca necesitan actualizarse. Si desea configurar una vida útil más corta, puede utilizar los métodos `tokensExpireIn` y `refreshTokensExpireIn`. Estos métodos deben ser llamados desde el método `boot` de su `AuthServiceProvider`:

```
use Carbon\Carbon;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::tokensExpireIn(Carbon::now()->addDays(15));

    Passport::refreshTokensExpireIn(Carbon::now()->addDays(30));
}
```

Emisión de tokens de acceso

El uso de OAuth2 con códigos de autorización es como la mayoría de los desarrolladores que están familiarizados con OAuth2 lo usan. Cuando se utilizan códigos de autorización, una aplicación cliente redirigirá a un usuario a su servidor donde aprobará o denegará la solicitud de emisión de un token de acceso al cliente.

Gestión de clientes

En primer lugar, los desarrolladores que necesitan interactuar con la API de su aplicación necesitarán registrar su aplicación con la suya creando un "cliente". Normalmente, esto consiste en proporcionar el nombre de su aplicación y una URL a la que su aplicación puede redirigir después de que los usuarios aprueben su solicitud de autorización.

El comando `passport:client`

La forma más sencilla de crear un cliente es utilizando el comando Artisan `passport:client`. Este comando puede ser utilizado para crear sus propios clientes para probar su funcionalidad OAuth2. Cuando ejecute el comando `client`, Passport le pedirá más información sobre su cliente y le proporcionará un ID de cliente y un valor secreto:

```
php artisan passport:client
```

API JSON

Dado que sus usuarios no podrán utilizar el comando `client`, Passport proporciona una API JSON que puede utilizar para crear clientes. Esto le ahorra la molestia de tener que codificar manualmente los controladores para crear, actualizar y eliminar clientes.

Sin embargo, usted necesitará emparejar la API JSON de Passport con su propio frontend para proporcionar un panel de control para que sus usuarios administren sus clientes. A continuación, revisaremos todos los endpoints de API para administrar clientes. Para mayor comodidad, utilizaremos [Axios](#) para demostrar cómo realizar solicitudes HTTP en los puntos finales.

Si no desea implementar el frontend de administración de cliente completo usted mismo, puede utilizar el frontend quickstart para tener un frontend completamente funcional en cuestión de minutos.

GET `/oauth/clients`

Esta ruta devuelve todos los clientes para el usuario autenticado. Esto es principalmente útil para listar todos los clientes del usuario para que puedan editarlos o borrarlos:

```
axios.get('/oauth/clients')
  .then(response => {
    console.log(response.data);
 });
```

POST `/oauth/clients`

Esta ruta se utiliza para crear nuevos clientes. Requiere dos tipos de datos: el nombre del cliente `name` y una URL de redirección `redirect`. La URL de redirección `redirect` es donde el usuario será redirigido después de aprobar o rechazar una solicitud de autorización.

Cuando se crea un cliente, se emitirá un ID y un valor secreto. Estos valores se utilizarán cuando solicite tokens de acceso desde su aplicación. La ruta de creación del cliente devolverá la nueva instancia del cliente:

```
const data = {
  name: 'Client Name',
  redirect: 'http://example.com/callback'
};

axios.post('/oauth/clients', data)
  .then(response => {
    console.log(response.data);
  })
  .catch(response => {
    // List errors on response...
  });

```

PUT /oauth/clients/{client-id}

Esta ruta se utiliza para actualizar clientes. Requiere dos tipos de datos: el nombre del cliente `name` y una URL de redirección `redirect`. La URL de redirección `redirect` es donde el usuario será redirigido después de aprobar o rechazar una solicitud de autorización. La ruta devolverá la instancia de cliente actualizada:

```
const data = {
  name: 'New Client Name',
  redirect: 'http://example.com/callback'
};

axios.put('/oauth/clients/' + clientId, data)
  .then(response => {
    console.log(response.data);
  })
  .catch(response => {
    // List errors on response...
  });

```

DELETE /oauth/clients/{client-id}

Esta ruta se utiliza para eliminar clientes:

```
axios.delete('/oauth/clients/' + clientId)
  .then(response => {
    //
  });
}
```

Solicitud de tokens

Redireccionando para autenticación

Una vez que se ha creado un cliente, los desarrolladores pueden utilizar su ID de cliente y el valor secreto para solicitar un código de autorización y un token de acceso desde su aplicación. En primer lugar, la aplicación consumidora debe hacer una solicitud de redireccionamiento a la ruta `/oauth/authorize` de su aplicación así:

```
Route::get('/redirect', function () {
  $query = http_build_query([
    'client_id' => 'client-id',
    'redirect_uri' => 'http://example.com/callback',
    'response_type' => 'code',
    'scope' => '',
  ]);

  return redirect('http://your-app.com/oauth/authorize?' . $query);
});
```

Recuerde, la ruta `/oauth/authorize` ya está definida por el método `Pasaporte::route`. No es necesario definir manualmente esta ruta.

Aprobar la solicitud

Al recibir solicitudes de autorización, Passport mostrará automáticamente una plantilla al usuario para que éste pueda aprobar o denegar la solicitud de autorización. Si se aprueba la solicitud, será redirigido de vuelta al `redirect_uri` especificado por la aplicación consumidora. El `redirect_uri` debe coincidir con la URL de redirección especificada `redirect` de cuando se creó el cliente.

Si desea personalizar la pantalla de autorización, puede publicar las vistas de Passport utilizando el comando Artisan `vendor:publish`. Las vistas publicadas se colocarán en `resources/views/vendor/passport`:

```
php artisan vendor:publish --tag=passport-views
```

Conversión de códigos de autorización para acceder a los Tokens

Si el usuario autoriza la solicitud, se redireccionará a la aplicación consumidora. El consumidor debe entonces enviar una solicitud `POST` a la aplicación para solicitar un token de acceso. La solicitud debe incluir el código de autorización emitido por la aplicación. En este ejemplo, usaremos la librería Guzzle HTTP para hacer la petición `POST`:

```
Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://your-app.com/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => 'client-id',
            'client_secret' => 'client-secret',
            'redirect_uri' => 'http://example.com/callback',
            'code' => $request->code,
        ],
    ]);
    return json_decode((string) $response->getBody(), true);
});
```

Esta ruta `/oauth/token` devuelve una respuesta JSON que contiene los atributos `access_token`, `refresh_token`, y `expires_in`. El atributo `expires_in` contiene el número de segundos hasta que expira el token de acceso.

Como la ruta `/oauth/authorize`, la ruta `/oauth/token` se define por el método `Passport::routes`. No es necesario definir manualmente esta ruta.

Refrescando Tokens

Si su aplicación emite tokens de acceso de corta duración, los usuarios tendrán que refrescar sus tokens mediante el token de actualización que se les proporcionó cuando se emitió el token de acceso. En este ejemplo, usaremos la librería Guzzle HTTP para hacer la petición POST:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

La ruta `/oauth/token` devuelve una respuesta JSON que contiene los atributos `access_token`, `refresh_token`, y `expires_in`. El atributo `expires_in` contiene el número de segundos hasta que expira el token de acceso.

Tokens de concesión de contraseña

La concesión de la contraseña OAuth2 permite a sus otros clientes, como una aplicación móvil, obtener una clave de acceso utilizando una dirección de correo electrónico / nombre de usuario y contraseña. Esto le permite emitir tokens de acceso de forma segura a sus clientes de primera mano sin requerir que sus usuarios pasen por todo el flujo de redirecccionamiento del código de autorización de OAuth2.

Crear un cliente de concesión de contraseña

Antes de que su aplicación pueda emitir tokens a través de la concesión de contraseña, deberá crear un cliente apropiado para ello. Puede hacerlo utilizando el comando `passport:client` con la opción `--password`. Si ya ha ejecutado el comando `passport:install`, no necesita ejecutar este comando:

```
php artisan passport:client --password
```

Solicitud de tokens

Una vez que haya creado un cliente de concesión de contraseña, puede solicitar un token de acceso enviando una solicitud `POST` a la ruta `/oauth/token` con la dirección de correo electrónico y contraseña del usuario. Recuerde, esta ruta ya está registrada por el método `Passport::routes` para que no sea necesario definirla manualmente. Si la solicitud tiene éxito, recibirá un `access_token` y `refresh_token` en la respuesta JSON del servidor:

```
$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

Recuerde, los tokens de acceso son de larga duración por defecto. Sin embargo, usted es libre de configurar la vida útil máxima del token de acceso si es necesario.

Solicitud de todos los ámbitos

Cuando utilice la concesión de contraseña, puede que desee autorizar el token para todos los ámbitos soportados por su aplicación. Puede hacerlo solicitando el scope `*`. Si usted solicita el scope `*`, el método `can` en la instancia del token siempre regresará `true`. Este alcance sólo puede asignarse a un token que se emita utilizando la concesión de `password`:

```
$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'password',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'username' => 'taylor@laravel.com',
        'password' => 'my-password',
        'scope' => '*',
    ],
])
```

Implicit Grant Tokens

La concesión implícita es similar a la concesión del código de autorización; sin embargo, el token se devuelve al cliente sin cambiar un código de autorización. Esta concesión se utiliza más comúnmente para aplicaciones JavaScript o móviles en las que las credenciales del cliente no se pueden almacenar de forma segura. Para habilitar la concesión, se debe llamar al método `enableImplicitGrant` en su `AuthServiceProvider`:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::enableImplicitGrant();
}
```

Una vez que se ha habilitado una concesión, los desarrolladores pueden utilizar su ID de cliente para solicitar un token de acceso desde su aplicación. La aplicación consumidora debería hacer una petición de redireccionamiento a la ruta `/oauth/authorize` de su aplicación:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

Recuerde, la ruta `/oauth/authorize` ya está definida por el método `Passport::routes`. No es necesario definir manualmente esta ruta.

Tokens de concesión de credenciales de cliente

La concesión de credenciales de cliente es adecuada para la autenticación máquina a máquina. Por ejemplo, puede utilizar esta concesión en un trabajo programado que esté realizando tareas de mantenimiento a través de una API. Para usar este método primero necesita agregar un nuevo middleware a su `$routeMiddleware` en `app/Http/Kernel.php`:

```
use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'client' => CheckClientCredentials::class,
];
```

A continuación, conecte este middleware a una ruta:

```
Route::get('/user', function(Request $request) {
    ...
})->middleware('client');
```

Para recuperar un token, haga una solicitud al endpoint `oauth/token`:

```
$guzzle = new GuzzleHttp\Client;

$response = $guzzle->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'client_credentials',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => 'your-scope',
    ],
]);

return json_decode((string) $response->getBody(), true)['access_token'];
```

Tokens de acceso personal

A veces, sus usuarios pueden querer emitir tokens de acceso para ellos mismos sin pasar por el típico flujo de redireccionamiento de código de autorización. Permitir a los usuarios emitir tokens para sí mismos a través de la interfaz de usuario de su aplicación puede ser útil para permitir a los usuarios experimentar con su API o puede servir como un enfoque más simple para emitir tokens de acceso en general.

Los tokens de acceso personal son siempre duraderos. Su vida útil no se modifica cuando se utilizan los métodos `tokensExpireIn` o `refreshTokensExpireIn`.

Crear un cliente de acceso personal

Antes de que su aplicación pueda emitir tokens de acceso personal, deberá crear un cliente de acceso personal. Puede hacerlo utilizando el comando `passport:client` con la opción `--personal`. Si ya ha ejecutado el comando `passport:install`, no necesita ejecutar este comando:

```
php artisan passport:client --personal
```

Administrar tokens de acceso personal

Una vez que haya creado un cliente de acceso personal, puede emitir tokens para un usuario determinado utilizando el método `createToken` en la instancia de modelo de `User`. El método `createToken` acepta el nombre del token como primer argumento y una matriz opcional de [scopes](#) como segundo argumento:

```
$user = App\User::find(1);

// Creating a token without scopes...
$token = $user->createToken('Token Name')->accessToken;

// Creating a token with scopes...
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

API JSON

Passport también incluye una API JSON para la gestión de tokens de acceso personal. Usted puede asociar esto con su propio frontend para ofrecer a sus usuarios un panel de control para gestionar los tokens de acceso personal. A continuación, revisaremos todos los endpoints de la API para administrar los tokens de acceso personal. Para mayor comodidad, utilizaremos [Axios](#) para demostrar cómo realizar solicitudes HTTP en los puntos finales.

Si no desea implementar el frontend de acceso personal usted mismo, puede utilizar el quickstart del frontend para tener un frontend totalmente funcional en cuestión de minutos.

GET /oauth/scopes

Esta ruta devuelve todos los [alcances \(scopes\)](#) definidos para su aplicación. Puede utilizar esta ruta para listar los alcances que un usuario puede asignar a un token de acceso personal:

```
axios.get('/oauth/scopes')
  .then(response => {
    console.log(response.data);
 });
```

GET /oauth/personal-access-tokens

Esta ruta devuelve todos los tokens de acceso personal creados por el usuario autenticado. Esto es principalmente útil para listar todos los tokens del usuario para que puedan editarlos o borrarlos:

```
axios.get('/oauth/personal-access-tokens')
  .then(response => {
    console.log(response.data);
  });
});
```

POST /oauth/personal-access-tokens

Esta ruta crea nuevos tokens de acceso personal. Requiere dos piezas de datos: `name` y `scopes` que deben asignarse al token:

```
const data = {
  name: 'Token Name',
  scopes: []
};

axios.post('/oauth/personal-access-tokens', data)
  .then(response => {
    console.log(response.data.accessToken);
  })
  .catch(response => {
    // List errors on response...
  });
});
```

DELETE /oauth/personal-access-tokens/{token-id}

Esta ruta se puede utilizar para borrar los tokens de acceso personal:

```
axios.delete('/oauth/personal-access-tokens/' + tokenId);
```

Protección de Rutas

A través de *middleware*

Passport incluye un [authentication guard \(protector de autenticación\)](#) que validará los tokens de acceso en las solicitudes entrantes. Una vez que haya configurado el `api` guard para usar el controlador de `passport`, sólo necesita especificar el middleware `auth:api` en cualquier ruta que requiera un token de acceso válido:

```
Route::get('/user', function () {
    //
})->middleware('auth:api');
```

Pasando el token de acceso

Al llamar a rutas protegidas por Passport, los consumidores de la API de su aplicación deben especificar su token de acceso como `Bearer` token en el encabezado `Authorization` de su solicitud. Por ejemplo, cuando se utiliza la biblioteca Guzzle HTTP:

```
$response = $client->request('GET', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer ' . $accessToken,
    ],
]);
```

Alcance Token

Definiendo scopes

Los scopes (alcances) permiten a sus clientes API obtener un conjunto específico de permisos cuando solicitan autorización para acceder a una cuenta. Por ejemplo, si está creando una aplicación de comercio electrónico, no todos los consumidores de la API necesitarán poder realizar pedidos. En su lugar, puede permitir que los consumidores sólo soliciten autorización para acceder al estatus de envío de pedidos. En otras palabras, los alcances permiten a los usuarios de su aplicación limitar las acciones que una aplicación de terceros puede realizar en su nombre.

Puede definir los alcances de su API usando el método `Passport::tokensCan` en el método `boot` de su `AuthServiceProvider`. El método `tokensCan` acepta una matriz de nombres y descripciones de alcance. La descripción del alcance puede ser cualquier cosa que desee y se mostrará a los usuarios en la pantalla de autorización:

```
use Laravel\Passport\Passport;

Passport::tokensCan([
    'place-orders' => 'Place orders',
    'check-status' => 'Check order status',
]);
```

Asignación de scopes a tokens

Cuando solicita códigos de autorización

Cuando solicita un token de acceso utilizando la concesión de código de autorización, los consumidores deben especificar sus alcances deseados como parámetro de cadena de consulta de `scope`. El parámetro `scope` debe ser una lista de alcances delimitada por el espacio:

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => 'client-id',
        'redirect_uri' => 'http://example.com/callback',
        'response_type' => 'code',
        'scope' => 'place-orders check-status',
    ]);

    return redirect('http://your-app.com/oauth/authorize?'.$query);
});
```

Cuando se emitan Tokens de acceso personal

Si está emitiendo tokens de acceso personal utilizando el método `createToken` del modelo `User`, puede pasar la matriz de alcances deseados como segundo argumento al método:

```
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

Verificación de scopes

Passport incluye dos middleware que se pueden utilizar para verificar que una solicitud entrante se autentica con un token al que se le ha concedido un alcance dado. Para empezar, agregue el siguiente middleware a la propiedad `$routeMiddleware` de su archivo `app/Http/Kernel.php`:

```
'scopes' => \Laravel\Passport\Http\Middleware\CheckScopes::class,  
'scope' => \Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

Verifique todos los alcances

El middleware `scopes` puede ser asignado a una ruta para verificar que el token de acceso de la solicitud entrante tiene *todos* los alcances listados:

```
Route::get('/orders', function () {  
    // Access token has both "check-status" and "place-orders" scopes...  
})->middleware('scopes:check-status,place-orders');
```

Verifique cualquier alcance

El middleware `scope` puede asignarse a una ruta para verificar que el token de acceso de la solicitud entrante tenga *al menos uno* de los alcances listados:

```
Route::get('/orders', function () {  
    // Access token has either "check-status" or "place-orders" scope...  
})->middleware('scope:check-status,place-orders');
```

Comprobación de ámbitos en una instancia de un Token

Una vez que una solicitud autenticada de token de acceso ha entrado en su aplicación, puede comprobar si el token tiene un alcance determinado utilizando el método `tokenCan` en la instancia de `Usuario` autenticado:

```
use Illuminate\Http\Request;  
  
Route::get('/orders', function (Request $request) {  
    if ($request->user()->tokenCan('place-orders')) {  
        //  
    }  
});
```

Consumir su API con JavaScript

Al crear una API, puede ser extremadamente útil poder consumir su propia API desde su aplicación JavaScript. Este enfoque al desarrollo de API permite que su propia aplicación consuma la misma API que usted está compartiendo con el mundo. La misma API puede ser consumida por su aplicación web, aplicaciones móviles, aplicaciones de terceros y cualquier SDK que pueda publicar en varios administradores de paquetes.

Normalmente, si desea consumir su API desde su aplicación JavaScript, debería enviar manualmente un token de acceso a la aplicación y pasarlo con cada solicitud. Sin embargo, Passport incluye un middleware que puede manejar esto por usted. Todo lo que necesita hacer es añadir el middleware `CreateFreshApiToken` al grupo de middleware `web` :

```
'web' => [
    // Other middleware...
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
],
```

Este middleware Passport adjuntará una cookie `laravel_token` a sus respuestas salientes. Esta cookie contiene un JWT cifrado que Passport utilizará para autenticar las solicitudes API de su aplicación JavaScript. Ahora, puede realizar solicitudes a la API de su aplicación sin pasar explícitamente un token de acceso:

```
axios.get('/api/user')
  .then(response => {
    console.log(response.data);
 });
```

Cuando se utiliza este método de autenticación, el scaffolding JavaScript predeterminado de Laravel instruye a Axios para que siempre envíe en el encabezado el `X-CSRF-TOKEN` y el `X-Requested-With`. Sin embargo, debe asegurarse de incluir su token CSRF en un [meta tag HTML](#):

```
window.axios.defaults.headers.common = {
  'X-Requested-With': 'XMLHttpRequest',
};
```

Si está utilizando un framework JavaScript diferente, debe asegurarse de que está configurado para enviar en los encabezados el `X-CSRF-TOKEN` y `X-Requested-With` con cada solicitud saliente.

Eventos

Passport levanta eventos cuando se emiten tokens de acceso y tokens de actualización. Usted puede usar estos eventos para limpiar o revocar otros tokens de acceso en su base de datos. Puede adjuntar oyentes a estos eventos en el `EventServiceProvider` de su aplicación:

```
/**  
 * The event listener mappings for the application.  
 *  
 * @var array  
 */  
  
protected $listen = [  
    'Laravel\Passport\Events\AccessTokenCreated' => [  
        'App\Listeners\RevokeOldTokens',  
    ],  
  
    'Laravel\Passport\Events\RefreshTokenCreated' => [  
        'App\Listeners\PruneOldTokens',  
    ],  
];
```

Testing

El método `actingAs` de Passport se puede utilizar para especificar el usuario actualmente autenticado, así como sus alcances. El primer argumento dado al método `actingAs` es la instancia de usuario y el segundo es un array de alcances que debe ser otorgado al token del usuario:

```
public function testServerCreation()
{
    Passport::actingAs(
        factory(User::class)->create(),
        ['create-servers']
    );

    $response = $this->post('/api/create-server');

    $response->assertStatus(200);
}
```

Laravel Scout

Introducción

Instalación

- # Colas – *Queueing*
- # Prerequisitos del driver

Configuración

- # Configurar el indexado de modelos
- # Configurar datos para la búsqueda

Indexar

- # Importación por lotes
- # Agregar registros
- # Actualizar registros
- # Eliminar registros
- # Pausar el indexado

Buscar

- # Cláusulas *where*
- # Paginación

Motores propios

Introducción

Laravel Scout proporciona una solución sencilla, basada en *drivers* para añadir la búsqueda de texto completo a sus modelos [Eloquent](#). Usando los observadores de modelo, Scout automáticamente mantendrá sus índices de búsqueda sincronizados con sus registros Eloquent.

Actualmente, Scout trabaja con un *driver* [Algolia](#); sin embargo, es bastante sencillo extender Scout con sus propias implementaciones de búsqueda.

Instalación

Primero, instale Scout utilizando Composer:

```
composer require laravel/scout
```

Después de instalar Scout, debe publicar la configuración usando el comando Artisan `vendor:publish`. Este comando publicará el archivo de configuración `scout.php` en su directorio `config`:

```
php artisan vendor:publish --provider="Laravel\\Scout\\ScoutServiceProvider"
```

Finalmente, agregue el *trait* `Laravel\\Scout\\Searchable` al modelo en el que desea buscar. Este *trait* registrará un observador de modelo para mantenerlo sincronizado con sus *drivers* de búsqueda:

```
<?php

namespace App;

use Laravel\\Scout\\Searchable;
use Illuminate\\Database\\Eloquent\\Model;

class Post extends Model
{
    use Searchable;
}
```

Colas – Queueing

Aunque no es estrictamente necesario para usar Scout, considere seriamente configurar un queue driver antes de usar la librería. La ejecución de un *queue driver* le permitirá a Scout poner en cola a todas las operaciones que sincronicen la información de su modelo con sus índices de búsqueda, proporcionando tiempos de respuesta mucho mejores para la interfaz web de su aplicación.

Una vez que haya configurado un `queue driver`, ajuste el valor de la opción `queue` en su archivo de configuración `config/scout.php` a `true`:

```
'queue' => true,
```

Prerequisitos del driver

Algolia

Cuando utilice el *driver* Algolia, debe configurar sus credenciales Algolia `id` y `secret` en su fichero de configuración `config/scout.php`. Una vez que sus credenciales hayan sido configuradas, también necesitará instalar el SDK PHP de Algolia a través de Composer:

```
composer require algolia/algoliasearch-client-php
```

Configuración

Configurar el indexado de modelos

Cada modelo Eloquent se sincroniza con un determinado "index" de búsqueda, que contiene todos los registros en donde se puede buscar para ese modelo. En otras palabras, puede pensar en cada índice como una tabla MySQL. De forma predeterminada, cada modelo persistirá en un índice que normalmente coincide con el nombre de la "tabla" del modelo. Normalmente, esta es la forma plural del nombre del modelo; sin embargo, se es libre de personalizar el índice anulando el método `buscableAs` en el modelo:

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * Get the index name for the model.
     *
     * @return string
     */
    public function searchableAs()
    {
        return 'posts_index';
    }
}
```

Configurar datos para la búsqueda

Por defecto, la forma completa del método `toArray` de un modelo se persistirá en su índice de búsqueda. Si desea personalizar los datos que se sincronizan con el índice de búsqueda, puede sustituir el método `toSearchableArray` en el modelo:

```
<?php

namespace App;

use Laravel\Scout\Searchable;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // Customize array...

        return $array;
    }
}
```

Indexar

Importación por lotes

Si está instalando Scout en un proyecto existente, es posible que ya tenga registros de base de datos que necesite importar a su controlador de búsqueda. Scout proporciona un comando Artisan `import` que puede utilizar para importar todos los registros existentes en sus índices de búsqueda:

```
php artisan scout:import "App\Post"
```

Agregar registros

Una vez que haya agregado el trait `Laravel\Scout\Searchable` a un modelo, todo lo que necesita hacer es `save` en la instancia del modelo y automáticamente se agregará a su índice de búsqueda. Si usted ha configurado Scout para [use queues](#) esta operación será realizada en segundo plano por su *queue worker*:

```
$order = new App\Order;  
  
// ...  
  
$order->save();
```

Agregar vía query

Si desea agregar una colección de modelos a su índice de búsqueda mediante una consulta Eloquent, puede encadenar el método `searchable` a la consulta. El método `searchable` [fragmentara los resultados](#) de la consulta y añadirá los registros a su índice de búsqueda. Una vez más, si ha configurado Scout para usar colas, todos los "fragmentos" se añadirán en segundo plano:

```
// Adding via Eloquent query...  
App\Order::where('price', '>', 100)->searchable();  
  
// You may also add records via relationships...  
$user->orders()->searchable();  
  
// You may also add records via collections...  
$orders->searchable();
```

El método `searchable` puede considerarse una operación "upsert". En otras palabras, si el registro del modelo ya está en su índice, se actualizará. Si no existe en el índice de búsqueda, se añadirá.

Actualizar registros

Para actualizar un modelo *searchable*, sólo necesita actualizar las propiedades de la instancia del modelo y [guardarlo \(save\)](#) en su base de datos. Scout persistirá automáticamente los cambios en su índice de búsqueda:

```
$order = App\Order::find(1);

// Update the order...

$order->save();
```

También puede utilizar el método `searchable` en una consulta Eloquent para actualizar una colección de modelos. Si los modelos no existen en su índice de búsqueda, se crearán:

```
// Updating via Eloquent query...
App\Order::where('price', '>', 100)->searchable();

// You may also update via relationships...
$user->orders()->searchable();

// You may also update via collections...
$orders->searchable();
```

Eliminar registros

Para eliminar un registro de su índice, simplemente `elimine (delete)` el modelo de la base de datos. Esta forma de eliminación es incluso compatible con los modelos soft deleted:

```
$order = App\Order::find(1);

$order->delete();
```

Si no desea recuperar el modelo antes de borrar el registro, puede utilizar el método `unsearchable` en una instancia o colección de consulta Eloquent:

```
// Removing via Eloquent query...
App\Order::where('price', '>', 100)->unsearchable();

// You may also remove via relationships...
$user->orders()->unsearchable();

// You may also remove via collections...
$orders->unsearchable();
```

Pausar el indexado

A veces puede ser necesario realizar un lote de operaciones Eloquent en un modelo sin sincronizar los datos con el índice de búsqueda. Puede hacerlo utilizando el método `withoutSyncingToSearch`. Este método acepta una única llamada de retorno que se ejecutará inmediatamente. Cualquier operación del modelo que ocurra dentro de la llamada de retorno no se sincronizará con el índice del modelo:

```
App\Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

Buscar

Puede comenzar a buscar un modelo utilizando el método `search`. El método de búsqueda acepta una sola cadena que se utilizará para buscar en sus modelos. A continuación, debería encadenar el método `get` en la consulta de búsqueda para recuperar los modelos Eloquent que coinciden:

```
$orders = App\Order::search('Star Trek')->get();
```

Dado que las búsquedas Scout devuelven una colección de modelos Eloquent, incluso puede devolver los resultados directamente desde una ruta o controlador y se convertirán automáticamente a JSON:

```
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return App\Order::search($request->search)->get();
});
```

Si desea obtener los resultados brutos antes de convertirlos a modelos Eloquent, debe utilizar el método `raw`:

```
$orders = App\Order::search('Star Trek')->raw();
```

Las consultas de búsqueda normalmente se realizarán en el índice especificado por el método [searchableAs](#) del modelo. Sin embargo, puede utilizar el método [within](#) para especificar un índice personalizado:

```
$orders = App\Order::search('Star Trek')
->within('tv_shows_popularity_desc')
->get();
```

Cláusulas Where

Scout le permite añadir cláusulas simples "where" a sus consultas de búsqueda. Actualmente, estas cláusulas sólo admiten comprobaciones básicas de igualdad numérica, y son útiles principalmente para determinar el alcance de las consultas de búsqueda por parte de un *tenant ID*. Dado que una búsqueda por indice no es relacional, las cláusulas "where" más avanzadas no son actualmente compatibles:

```
$orders = App\Order::search('Star Trek')->where('user_id', 1)->get();
```

Paginación

Además de recuperar una colección de modelos, puede paginar los resultados de su búsqueda utilizando el método [paginate](#). Este método devolverá una instancia [Paginator](#) igual que si hubiera [paginado una consulta Eloquent tradicional](#):

```
$orders = App\Order::search('Star Trek')->paginate();
```

Puede especificar cuántos modelos desea recuperar por página pasando la cantidad como primer argumento al método [paginate](#):

```
$orders = App\Order::search('Star Trek')->paginate(15);
```

Una vez que se hayan recuperado los resultados, puede mostrarlos y colocar los enlaces de la página usando [Blade](#) como si hubiera paginado una consulta Eloquent tradicional:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

Motores propios

Escribiendo el motor

Si los buscadores Scout incorporados no se ajustan a sus necesidades, puede escribir su propio motor de búsqueda personalizado y registrarlo con Scout. Su motor debe extender la clase abstracta [Laravel\Scout\Engines\Engine](#). Esta clase abstracta contiene cinco métodos que su motor debe implementar:

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function map($results, $model);
```

Puede ser útil revisar las implementaciones de estos métodos en la clase [Laravel\Scout\Engines\AlgoliaEngine](#). Esta clase le proporcionará un buen punto de partida para aprender a implementar cada uno de estos métodos en su propio motor.

Registrando el motor

Una vez escrito el motor personalizado, puede registrarse con Scout usando el método `extend` del gestor del motor de Scout. Debe llamar al método `extend` desde el método `boot` de su [AppServiceProvider](#) o desde cualquier otro proveedor de servicios utilizado por su aplicación. Por ejemplo, si ha escrito un [MySqlSearchEngine](#), puede registrarlo así:

```
use Laravel\Scout\EngineManager;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

Una vez que su motor ha sido registrado, puede especificarlo como `driver` Scout predeterminado en su archivo de configuración `config/scout.php`:

```
'driver' => 'mysql',
```

Laravel Socialite

- # [Introducción](#)
 - # [Instalación](#)
 - # [Configuración](#)
 - # [Rutas](#)
 - # [Parámetros opcionales](#)
 - # [Ámbitos de acceso – Access scopes](#)
 - # [Autenticación sin estado](#)
 - # [Obtener los Datos del Usuario](#)
-

Introducción

Además de la típica autenticación basada en formularios, Laravel proporciona una simple y conveniente forma de autenticar con proveedores OAuth utilizando [Laravel Socialite](#). Socialite actualmente soporta autenticación con Facebook, Twitter, LinkedIn, Google, GitHub y Bitbucket.

Adaptadores para otras plataformas se pueden encontrar en Socialite Providers gestionado por la comunidad.

Instalación

Para comenzar con Socialite, utilizar Composer para añadir el paquete a las dependencias del proyecto:

```
composer require laravel/socialite
```

Configuración

Antes de usar Socialite, será necesario añadir los credenciales para los servicios OAuth que utilice la aplicación. Estos credenciales se deben colocar en el archivo de configuración `config/services`, y deben utilizar la clave `facebook`, `twitter`, `linkedin`, `google`, `github` o `bitbucket`, en función de los proveedores que requiera la aplicación. Por ejemplo:

```
'github' => [
    'client_id' => env('GITHUB_CLIENT_ID'),           // Your GitHub Client ID
    'client_secret' => env('GITHUB_CLIENT_SECRET'), // Your GitHub Client Secret
    'redirect' => 'http://your-callback-url',
],
```

Si la opción `redirect` contiene una ruta relativa, se resolverá automáticamente a una URL completa.

Rutas

A continuación, ¡ya podemos autenticar usuarios! Habrá que definir dos rutas: una para redireccionar al usuario al proveedor OAuth, y otra como respuesta después de la autenticación del proveedor. Se accederá a Socialite utilizando la facade `Socialite`:

```

<?php

namespace App\Http\Controllers\Auth;

use Socialite;

class LoginController extends Controller
{
    /**
     * Redirect the user to the GitHub authentication page.
     *
     * @return \Illuminate\Http\Response
     */
    public function redirectToProvider()
    {
        return Socialite::driver('github')->redirect();
    }

    /**
     * Obtain the user information from GitHub.
     *
     * @return \Illuminate\Http\Response
     */
    public function handleProviderCallback()
    {
        $user = Socialite::driver('github')->user();

        // $user->token;
    }
}

```

El método `redirectToProvider` se encarga de enviar al usuario al proveedor de OAuth, mientras que el método que `user` leerá la solicitud entrante y recuperará la información del usuario desde el proveedor.

Por supuesto, hay que definir las rutas a los métodos del controlador:

```

Route::get('login/github', 'Auth\LoginController@redirectToProvider');
Route::get('login/github/callback', 'Auth\LoginController@handleProviderCallback');

```

Parámetros opcionales

Algunos proveedores OAuth soportan parámetros opcionales en la solicitud de redirección. Para incluir cualquier parámetro en la petición, utilizar el método `with` con un array asociativo:

```
return Socialite::driver('google')
    ->with(['hd' => 'example.com'])
    ->redirect();
```

Al usar el método `with`, sea cuidadoso con no pasar ninguna palabra reservada como `state` o `response_type`.

Ámbitos de acceso – Access scopes

Antes de redireccionar al usuario, se pueden añadir "ámbitos" (o `scopes`) a la petición utilizando el método `scopes`. Este método juntará los `scopes` existentes con los que se provean nuevos:

```
return Socialite::driver('github')
    ->scopes(['read:user', 'public_repo'])
    ->redirect();
```

Se pueden reemplazar todos los `scopes` utilizando el método `setScopes`:

```
return Socialite::driver('github')
    ->setScopes(['read:user', 'public_repo'])
    ->redirect();
```

Autenticación sin estado

El método `stateless` se puede utilizar para desactivar la verificación de estado de sesión. Es útil cuando se añade la autenticación social a un API:

```
return Socialite::driver('google')->stateless()->user();
```

Obtener los datos del usuario

Una vez obtenida la instancia del usuario, se pueden acceder a más detalles sobre el usuario:

```
$user = Socialite::driver('github')->user();  
  
// OAuth Two Providers  
$token = $user->token;  
$refreshToken = $user->refreshToken; // not always provided  
$expiresIn = $user->expiresIn;  
  
// OAuth One Providers  
$token = $user->token;  
$tokenSecret = $user->tokenSecret;  
  
// All Providers  
$user->getId();  
$user->getNickname();  
$user->getName();  
$user->getEmail();  
$user->getAvatar();
```

Obtener detalles del usuario con un *token*

Si ya se tiene un *token* de acceso válido para un usuario, se pueden obtener sus detalles utilizando el método `userFromToken` :

```
$user = Socialite::driver('github')->userFromToken($token);
```