

Traducción de la documentación de Laravel al español

Bienvenido a la traducción de la documentación de Laravel 6.0 al español.

Puedes encontrar la documentación oficial de Laravel en inglés en <https://laravel.com/docs>

La traducción de la documentación de Laravel 5.8 puede ser encontrada en <https://v5.documentacion-laravel.com/>

Nota importante sobre la traducción

Actualmente esta traducción de la documentación de Laravel se encuentra en período de prueba. Puedes realizar reportes sobre errores o detalles encontrados así como contribuir al progreso y mantenimiento de la misma en el repositorio oficial de la traducción.

Guía para contribuir

Si está enviando documentación para la **versión estable actual**, envíala a la rama correspondiente. Por ejemplo, la documentación para Laravel 6.0 se enviaría a la rama `6.0_press`. La documentación destinada a la próxima versión de Laravel debe enviarse a la rama `master`.

Sobre esta traducción

Para saber más sobre esta traducción puedes dirigirte a la sección de créditos.

Índice de la documentación

En este índice podrás encontrar la lista completa de todos los capítulos de la documentación, incluyendo la guía de actualización y las notas de lanzamiento.

TIP

Si apenas estás comenzando con Laravel, en [Styde.net](#) contamos con un [completo curso de Laravel 6 desde cero](#).

• Prólogo

- Notas de lanzamiento
- Guía de actualización
- Guía de contribuciones

• Primeros pasos

- Instalación
- Configuración
- Estructura de directorios
- Laravel Homestead
- Laravel Valet
- Despliegue

• Conceptos de arquitectura

- Ciclo de vida de la solicitud
- Contenedor de servicios
- Proveedores de servicios
- Facades
- Contratos

• Fundamentos

- Rutas
- Middleware

- Protección CSRF
- Controladores
- Solicitudes HTTP
- Respuestas HTTP
- Vistas
- Generación De URLs
- Sesión HTTP
- Validación
- Manejo de errores
- Registro

- **Frontend**

- Plantillas Blade
- Configuración regional
- JavaScript y estructuración de CSS
- Compilación de assets

- **Seguridad**

- Autenticación
- Autenticación de API
- Autorización
- Verificación de correos electrónico
- Cifrado
- Hashing
- Restablecimiento de contraseñas

- **Profundizando**

- Consola Artisan
- Broadcasting
- Caché
- Colecciones
- Eventos

- Almacenamiento de archivos
- Helpers
- Correos electrónicos
- Notificaciones
- Desarrollo de paquetes
- Colas de trabajo
- Programación de tareas

- **Bases de datos**

- Bases de datos: Primeros pasos
- Base de datos: Constructor de consultas (query builder)
- Base de datos: Paginación
- Base de datos: Migraciones
- Base de datos: Seeding
- Redis

- **ORM Eloquent**

- Eloquent: Primeros pasos
- Eloquent: Relaciones
- Eloquent: Colecciones
- Eloquent: Mutators
- Eloquent: Recursos API
- Eloquent: Serialización

- **Pruebas**

- Pruebas: Primeros pasos
- Pruebas HTTP
- Pruebas de consola
- Laravel Dusk
- Pruebas de base de datos
- Mocking

• Paquetes oficiales

- [Cashier](#)
- [Dusk](#)
- [Envoy](#)
- [Horizon](#)
- [Passport](#)
- [Scout](#)
- [Socialite](#)
- [Telescope](#)

Créditos

Sobre esta traducción

Esta traducción al español de la documentación oficial de Laravel es un esfuerzo hecho posible y llevado a cabo por el equipo de [Style.net](#) así como de un grupo de colaboradores cuyas contribuciones han ayudado a la culminación de este proyecto:

- [Duilio Palacios](#)
- [Pastor Ramos](#)
- [Clemir Rondón](#)
- [David Palacios](#)
- [Dimitri Acosta](#)
- [Carlos Fernandes](#)
- [Simon Montoya](#)
- [Jerson Moreno](#)

Puedes contribuir a mantener esta traducción actualizada o compartir cualquier duda o sugerencia en el repositorio de GitHub de la documentación [.](#)

Glosario

A continuación podrás ver una lista de términos con los que probablemente te encontrarás a lo largo de la documentación y su significado.

Callback

Un callback es una función que se pasa como parámetro de otras funciones y que se ejecuta dentro de éstas.

Closure

Un Closure es una función anónima. Los Closures a menudo son usadas como funciones callback y pueden ser usadas como parámetros en una función.

Mocking

Los mocks son objetos que simulan el comportamiento de objetos reales.

El mocking se usa mayoritariamente en las pruebas unitarias. Un objeto que va a ser probado puede que dependa de objetos más complejos para su funcionamiento. Para aislar el comportamiento del objeto es probable que quieras reemplazar estos objetos complejos por mocks que simulan el comportamiento de los objetos reales. Esto es útil si los objetos reales son difíciles de incorporar en pruebas unitarias.

Ciclo de vida

El ciclo de vida es el periodo de tiempo durante el cual un proceso o servicio se ejecuta.

Controlador

Un controlador es la pieza encargada de comunicarse con el modelo de tu base de datos (enviar y recibir datos) así como presentar dichos datos a la vista.

Evento

Un evento es una parte del código que es llamada sólo cuando el usuario interactua con ella.

Promesa

Una promesa representa la eventual ejecución exitosa (o fallo) de una operación asíncrona y su valor resultante.

Work queue

Una work queue es una lista de tareas a llevar a cabo.

Colección

Una colección representa un grupo de objetos, normalmente relacionados con la base de datos.

Eloquent

Eloquent es un ORM, es decir, una capa que permite realizar consultas a la base de datos de una forma más sencilla y sin necesidad de escribir SQL de forma obligatoria.

Instalación

- [Instalación](#)
 - [Requisitos del servidor](#)

- Instalar Laravel
- Configuración
- Configuración del servidor web
 - Configuración del directorio
 - URLs amigables

Instalación

TIP

¿Te gustaría un curso en video para profundizar tu aprendizaje? En Styde cuentas con un [completo curso de Laravel](#) totalmente gratuito que incluye más de 40 lecciones.

Requisitos del servidor

El framework Laravel tiene algunos requisitos del sistema. Todos estos requisitos son cubiertos por la máquina virtual [Laravel Homestead](#), así que es altamente recomendable que uses Homestead como tu entorno local de desarrollo de Laravel.

Sin embargo, si no estás utilizando Homestead, deberás asegurarte de que tu servidor cumpla con los siguientes requisitos:

- PHP >= 7.2.0
- BCMath PHP Extension
- Ctype PHP Extension
- JSON PHP Extension
- Mbstring PHP Extension
- Extensión OpenSSL de PHP
- Extensión PDO de PHP
- Extensión Tokenizer de PHP
- Extensión XML de PHP

Nota

La instalación de Laravel en un subdirectorio no está soportada.

Instalar Laravel

Laravel utiliza [Composer](#) para administrar sus dependencias. Por lo que, antes de utilizar Laravel, deberás asegurarte de tener Composer instalado en tu sistema.

TIP

En la lección [instalación de Composer y Laravel](#) del curso gratuito [Laravel desde cero](#) de Styde puedes ver el proceso de instalación paso a paso.

Mediante el instalador de Laravel

Primero, descarga el instalador de Laravel usando Composer:

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

php

Asegurate de colocar el directorio `vendor/bin` en tu `$PATH` para que el ejecutable de Laravel pueda ser localizado en tu sistema. Este directorio existe en diferentes ubicaciones según el sistema operativo que estés utilizando; sin embargo, algunas de las ubicaciones más comunes son las siguientes:

- macOS: `$HOME/.composer/vendor/bin`
- Distribuciones GNU / Linux: `$HOME/.config/composer/vendor/bin` o `$HOME/.composer/vendor/bin`
- Windows: `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin`

También puedes encontrar el ruta global de instalación de composer ejecutando `composer global about` y observando la primera línea.

Una vez instalado, el comando `laravel new` creará una nueva instalación de Laravel en el directorio que especifiques. Por ejemplo, `laravel new blog` creará un directorio `blog` que contendrá una nueva instalación de Laravel con todas las dependencias de Laravel ya instaladas:

```
laravel new blog
```

php

Mediante composer create-project

Alternativamente, también puedes instalar Laravel ejecutando el comando de Composer `create-project` en tu terminal:

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

php

Servidor de desarrollo local

Si tienes instalado PHP de manera local y te gustaría utilizar el servidor de desarrollo incorporado en PHP para servir tu aplicación, puedes usar el comando de Artisan `serve`. Este comando iniciará un servidor de desarrollo en `http://localhost:8000`:

```
php artisan serve
```

php

Otras opciones de desarrollo local más robustas están disponibles mediante [Homestead](#) y [Valet](#).

Configuración

Directorio público

Después de haber instalado Laravel, deberás configurar el documento raíz de tu servidor web para que sea el directorio `public`. El archivo `index.php` en este directorio funciona como controlador frontal (front controller) para todas las peticiones HTTP que entran a tu aplicación.

Archivos de configuración

Todos los archivos de configuración para el framework Laravel están almacenados en el directorio `config`. Cada opción está documentada, así que siéntete libre de revisar estos archivos y familiarizarte con las opciones disponibles para ti.

Permisos para directorios

Después de haber instalado Laravel, necesitarás configurar algunos permisos. Los directorios dentro de `storage` y `bootstrap/cache` deberán tener permiso de escritura para tu servidor web o Laravel no va a funcionar. Si estás utilizando la máquina virtual [Homestead](#), estos permisos ya están establecidos.

Clave de la aplicación

Lo siguiente que debes hacer después de instalar Laravel es establecer la clave de tu aplicación a una cadena aleatoria. Si instalas Laravel mediante Composer o el instalador de Laravel, esta clave ya ha sido establecida por el comando `php artisan key:generate`.

Típicamente, esta cadena debe tener una longitud de 32 caracteres. La clave puede ser establecida en el archivo de entorno `.env`. Si no has copiado el archivo `.env.example` a un nuevo archivo llamado `.env`, deberás hacerlo ahora. **Si la clave de la aplicación no está establecida, las sesiones de usuario y otros datos encriptados no serán seguros!**

Configuración adicional

Laravel casi no necesita de configuración adicional. ¡Eres libre de empezar a desarrollar! Sin embargo, puede que quieras revisar el archivo `config/app.php` y su documentación. Contiene varias opciones como `timezone` y `locale` que es posible que deseas ajustar en tu aplicación.

También puede que quieras configurar algunos componentes adicionales de Laravel, como:

- Cache
- Base de Datos
- Sesiones

Configuración del servidor web

Configuración del directorio

Laravel debe ser siempre servido desde la raíz del "directorio web" configurado en tu servidor web. No debes intentar servir una aplicación de Laravel desde un subdirectorio del "directorio web". Intentar hacer eso podría exponer archivos sensibles que se encuentran dentro de tu aplicación.

URLs amigables

Apache

Laravel incluye un archivo `public/.htaccess` que es utilizado para proporcionar URLs sin el controlador frontal `index.php` en la ruta. Antes de servir tu aplicación de Laravel con Apache, asegúrate de habilitar el módulo `mod_rewrite` para que tu archivo `.htaccess` funcione correctamente.

Si el archivo `.htaccess` que viene con Laravel no funciona con tu instalación de Apache, prueba esta alternativa:

```
Options +FollowSymLinks -Indexes
RewriteEngine On

RewriteCond %{HTTP:Authorization} .
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Nginx

Si estás utilizando Nginx, la siguiente directiva en la configuración de tu sitio va a dirigir todas las peticiones al controlador frontal `index.php`:

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

Cuando uses [Homestead](#) o [Valet](#), las URLs amigables serán configuradas automáticamente.

Configuración

- [Introducción](#)
- [Configuración del entorno](#)
 - [Tipos de variables de entorno](#)
 - [Recuperar la configuración del entorno](#)

- Determinando el entorno actual
- Ocultar variables de entorno a páginas de depuración
- Acceder a valores de configuración
- Almacenamiento en caché de la configuración
- Modo de mantenimiento

Introducción

Todos los archivos de configuración para el framework Laravel están almacenados en el directorio `config`. Cada opción está documentada, así que no dudes en consultar los archivos y familiarizarte con las opciones disponibles para ti.

Configuración del entorno

A menudo es útil tener diferentes valores de configuración basados en el entorno en el que se ejecuta la aplicación. Por ejemplo, es posible que deseas utilizar un servidor de caché localmente diferente al servidor que usas en producción.

Para hacer esto sencillo, Laravel utiliza la librería de PHP `DotEnv` por Vance Lucas. En una nueva instalación de Laravel, el directorio raíz de tu aplicación contendrá un archivo `.env.example`. Si instalas Laravel por medio de Composer, este archivo será renombrado automáticamente a `.env`. De lo contrario, deberás renombrar el archivo manualmente.

Tu archivo `.env` deberá omitirse en el sistema de control de versiones de tu aplicación, ya que cada desarrollador / servidor que usa tu aplicación puede requerir una configuración de entorno diferente. Además, esto sería un riesgo de seguridad en caso de que un intruso obtenga acceso al repositorio de control de versiones de tu aplicación.

Si estás desarrollando con un equipo, es posible que deseas continuar incluyendo el archivo `.env.example` en tu aplicación. Al poner valores de ejemplo (placeholder) en el archivo de configuración `.env.example`, otros desarrolladores en tu equipo podrán ver claramente cuáles variables de entorno se necesitan para ejecutar tu aplicación. También puedes crear un archivo `.env.testing`. Este archivo sobrescribirá el archivo `.env` al ejecutar pruebas con PHPUnit o al ejecutar comandos de Artisan con la opción `--env=testing`.

TIP

Cualquier variable en tu archivo `.env` puede ser anulada por variables de entorno externas tales como variables de entorno de nivel de servidor o de nivel de sistema.

Tipos de variables de entorno

Todas las variables en tus archivos `.env` se traducen como cadenas, así que algunos valores reservados han sido creados para permitirte retornar un rango más amplio de tipos desde la función `env()`:

Valor en <code>.env</code>	Valor en <code>env()</code>
<code>true</code>	(booleano) true
<code>(true)</code>	(booleano) true
<code>false</code>	(booleano) false
<code>(false)</code>	(booleano) false
<code>empty</code>	(cadena) ”
<code>(empty)</code>	(cadena) ”
<code>null</code>	(null) null
<code>(null)</code>	(null) null

Si necesitas definir una variable de entorno con un valor que contiene espacios, puedes hacerlo encerrando el valor en comillas dobles.

```
APP_NAME="My Application"
```

php

Recuperar la configuración del entorno

Todas las variables listadas en este archivo van a ser cargadas en la variable super-global de PHP `$_ENV` cuando tu aplicación reciba una solicitud. Sin embargo, puedes utilizar el helper `env` para recuperar valores de estas variables en tus archivos de configuración. De hecho, si revisas los archivos de configuración de Laravel, podrás notar que varias de estas opciones ya están utilizando este helper:

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

php

El segundo valor pasado a la función `env` es el "valor predeterminado". Este valor será utilizado si no se encuentra una variable de entorno existente para la clave proporcionada.

Determinando el entorno actual

El entorno actual de la aplicación es determinado por medio de la variable `APP_ENV` desde tu archivo `.env`. Puedes acceder a este valor por medio del método `environment` del facade `App`:

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

php

También puedes pasar argumentos al método `environment` para verificar si el entorno coincide con un valor determinado. El método va a retornar `true` si el entorno coincide con cualquiera de los valores dados:

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

php

TIP

La detección del entorno actual de la aplicación puede ser anulada por una variable de entorno `APP_ENV` a nivel del servidor. Esto puede ser útil cuando necesites compartir la misma aplicación para diferentes configuraciones de entorno, para que puedas configurar un host determinado para que coincida con un entorno determinado en las configuraciones de tu servidor.

Ocultar variables de entorno a páginas de depuración

Cuando una excepción no es capturada y la variable de entorno `APP_DEBUG` es igual a `true`, la página de depuración mostrará todas las variables de entorno y sus contenidos. En algunos casos vas a

querer ocultar ciertas variables. Puedes hacer esto actualizando la opción `debug_blacklist` en tu archivo de configuración `config/app.php`.

Algunas variables están disponibles tanto en las variables de entorno y en los datos del servidor / petición. Por lo tanto, puede que necesites ocultarlos tanto para `$_ENV` como `$_SERVER`:

```
return [php

    // ...

    'debug_blacklist' => [
        '_ENV' => [
            'APP_KEY',
            'DB_PASSWORD',
        ],
        '_SERVER' => [
            'APP_KEY',
            'DB_PASSWORD',
        ],
        '_POST' => [
            'password',
        ],
    ],
];
```

Acceder a valores de configuración

Puedes acceder fácilmente a tus valores de configuración utilizando la función helper global `config` desde cualquier lugar de tu aplicación. Se puede acceder a los valores de configuración usando la sintaxis de "punto", que incluye el nombre del archivo y la opción a la que deseas acceder. También puedes especificar un valor predeterminado que se devolverá si la opción de configuración no existe:

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

Para establecer valores de configuración en tiempo de ejecución, pasa un arreglo al helper `config`:

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

php

Almacenamiento en caché de la configuración

Para dar a tu aplicación un aumento de velocidad, debes almacenar en caché todos tus archivos de configuración en un solo archivo usando el comando de Artisan `config:cache`. Esto combinará todas las opciones de configuración para tu aplicación en un solo archivo que será cargado rápidamente por el framework.

Usualmente deberías ejecutar el comando `php artisan config:cache` como parte de tu rutina de despliegue a producción. El comando no se debe ejecutar durante el desarrollo local ya que las opciones de configuración con frecuencia deberán cambiarse durante el desarrollo de tu aplicación.

Nota

Si ejecutas el comando `config:cache` durante el proceso de despliegue, debes asegurarte de llamar solo a la función `env` desde tus archivos de configuración. Una vez que la configuración se ha almacenado en caché, el archivo `.env` no será cargado y todas las llamadas a la función `env` retornarán `null`.

Modo de mantenimiento

Cuando tu aplicación se encuentre en modo de mantenimiento, se mostrará una vista personalizada para todas las solicitudes en tu aplicación. Esto facilita la "desactivación" de tu aplicación mientras se está actualizando o cuando se realiza mantenimiento. Se incluye una verificación de modo de mantenimiento en la pila de middleware predeterminada para tu aplicación. Si la aplicación está en modo de mantenimiento, una excepción `MaintenanceModeException` será lanzada con un código de estado 503.

Para habilitar el modo de mantenimiento, ejecuta el comando de Artisan `down`:

```
php artisan down
```

php

También puedes proporcionar las opciones `message` y `retry` al comando `down`. El valor de `message` se puede usar para mostrar o registrar un mensaje personalizado, mientras que el valor de

`retry` se establecerá como el valor de cabecera HTTP `Retry-After` :

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

php

Incluso en modo de mantenimiento, se les puede permitir acceder a la aplicación a direcciones IP o redes específicas usando la opción `allow` del comando:

```
php artisan down --allow=127.0.0.1 --allow=192.168.0.0/16
```

php

Para deshabilitar el modo de mantenimiento, usa el comando `up` :

```
php artisan up
```

php

TIP

Puedes personalizar la plantilla predeterminada del modo de mantenimiento al definir tu propia plantilla en `resources/views/errors/503.blade.php`.

Modo de mantenimiento y colas

Mientras tu aplicación esté en modo de mantenimiento, no se manejarán `trabajos en cola`. Los trabajos continuarán siendo manejados de forma normal una vez que la aplicación esté fuera del modo de mantenimiento.

Alternativas al modo de mantenimiento

Como el modo de mantenimiento requiere que tu aplicación tenga varios segundos de tiempo de inactividad, considera alternativas como [Envoyer](#) para lograr hacer deploy de Laravel sin tiempo de inactividad.

Estructura de Directorios

- Introducción
- Directorio Raíz
 - Directorio app
 - Directorio bootstrap
 - Directorio config
 - Directorio database
 - Directorio public
 - Directorio resources
 - Directorio routes
 - Directorio storage
 - Directorio tests
 - Directorio vendor
- Directorio App
 - Directorio Broadcasting
 - Directorio Console
 - Directorio Events
 - Directorio Exceptions
 - Directorio Http
 - Directorio Jobs
 - Directorio Listeners
 - Directorio Mail
 - Directorio Notifications
 - Directorio Policies
 - Directorio Providers
 - Directorio Rules

Introducción

La estructura por defecto de aplicación de Laravel está pensada para proporcionar un buen punto de inicio tanto para grandes y pequeñas aplicaciones. Pero, eres libre de organizar tu aplicación como quieras. Laravel no impone casi ninguna restricción sobre donde una clase es ubicada - siempre y cuando Composer pueda cargar automáticamente la clase.

¿Dónde está el directorio de modelos?

Al comenzar con Laravel, muchos desarrolladores son confundidos por la falta de un directorio `models`. Sin embargo, la falta de dicho directorio es intencional. Encontramos la palabra "models" ambigua dado que significa muchas cosas diferentes para muchas personas. Algunos desarrolladores se refieren al "modelo" de una aplicación como la totalidad de toda su lógica de negocio, mientras que otros se refieren a los "modelos" como clases que interactúan con una base de datos relacional.

Por esta razón, elegimos ubicar los modelos de Eloquent por defecto en el directorio `app` y permitir al desarrollar ubicarlos en algún otro sitio si así lo eligen.

Directorio Raíz

Directorio App

El directorio `app` contiene el código principal de tu aplicación. Exploraremos este directorio con más detalle pronto; sin embargo, casi todas las clases en tu aplicación estarán en este directorio.

Directorio Bootstrap

El directorio `bootstrap` contiene el archivo `app.php` que maqueta el framework. Este directorio también almacena un directorio `cache` que contiene archivos generados por el framework para optimización de rendimiento como los archivos de cache de rutas y servicios.

Directorio Config

El directorio `config`, como el nombre implica, contiene todos los archivos de configuración de tu aplicación. Es una buena idea leer todos estos archivos y familiarizarte con todas las opciones disponibles para ti.

Directorio Database

El directorio `database` contiene las migraciones de tu base de datos, model factories y seeders. Si lo deseas, puedes también usar este directorio para almacenar una base de datos SQLite.

Directorio Public

El directorio `public` contiene el archivo `index.php`, el cual es el punto de acceso para todas las solicitudes llegan a tu aplicación y configura la autocarga. Este directorio también almacena tus assets, tales como imágenes, JavaScript y CSS.

Directorio Resources

El directorio `resources` contiene tus vistas así como también tus assets sin compilar tales como LESS, Sass o JavaScript. Este directorio también almacena todos tus archivos de idioma.

Directorio Routes

El directorio `routes` contiene todas las definiciones de rutas para tu aplicación. Por defecto, algunos archivos de rutas son incluidos con Laravel: `web.php`, `api.php`, `console.php` y `channels.php`.

El archivo `web.php` contiene rutas que `RouteServiceProvider` coloca en el grupo de middleware `web`, que proporciona estado de sesión, protección CSRF y encriptación de cookies. Si tu aplicación no ofrece una API sin estado, todas tus rutas probablemente serán definidas en el archivo `web.php`.

El archivo `api.php` contiene rutas que `RouteServiceProvider` coloca en el grupo de middleware `api`, que proporcionan limitación de velocidad. Estas rutas están pensadas para no tener estado, así que las solicitudes que llegan a la aplicación a través de estas rutas están pensadas para ser autenticadas mediante tokens y no tendrán acceso al estado de sesión.

El archivo `console.php` es donde puedes definir todos los comandos basados en Closures de tu aplicación. Cada Closure está enlazado a una instancia de comando permitiendo una forma simple de interactuar con los métodos de entrada y salida de cada comando. Aunque este archivo no define ninguna ruta HTTP, sí define puntos de entrada en consola (rutas) a tu aplicación.

El archivo `channels.php` es donde puedes registrar todos los canales de transmisión de eventos que tu aplicación soporta.

Directorio Storage

El directorio `storage` contiene tus plantillas compiladas de Blade, sesiones basadas en archivos, archivos de caches y otros archivos generados por el framework. Este directorio está segregado en los directorios `app`, `framework` y `logs`. El directorio `app` puede ser usado para almacenar cualquier archivo generado por tu aplicación. El directorio `framework` es usado para almacenar archivos generados por el framework y cache. Finalmente, el directorio `logs` contiene los archivos de registros de tu aplicación.

El directorio `storage/app/public` puede ser usado para almacenar archivos generados por el usuario, tales como imágenes de perfil, que deberían ser accesibles públicamente. Debes crear un enlace

simbólico en `public/storage` que apunte a este directorio. Puedes crear el enlace usando el comando `php artisan storage:link`.

El Directorio Tests

El directorio `tests` contiene tus pruebas automatizadas. Una prueba de ejemplo de [PHPUnit](#) es proporcionada. Cada clase de prueba debe estar precedida por la palabra `Test`. Puedes ejecutar tus pruebas usando los comandos `phpunit` o `php vendor/bin/phpunit`.

Directorio Vendor

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

Directorio App

La mayoría de tu aplicación está almacenada el directorio `app`. Por defecto, este directorio está regido por el nombre de espacio `App` y es cargado automáticamente por Composer usando el [estándar de autocarga PSR-4](#).

El directorio `app` contiene una variedad de directorios adicionales tales como `Console`, `Http` y `Providers`. Piensa en los directorios `Console` y `Http` como si proporcionaran una API al núcleo de tu aplicación, pero no contienen lógica de la aplicación como tal. En otras palabras, son dos formas de emitir comandos a tu aplicación. El directorio `Console` contiene todos tus comandos de Artisan, mientras que el directorio `Http` contiene tus controladores, middleware y solicitudes.

Una variedad de otros directorios serán generados dentro del directorio `app` cuando uses los comandos `make` de Artisan para generar clases. Así que, por ejemplo, el directorio `app/Jobs` no existirá hasta que ejenes el comando de Artisan `make:job` para generar una clase job.

TIP

Muchas de las clases en el directorio `app` pueden ser generadas por Artisan mediante comandos. Para ver los comandos disponibles, ejecuta el comando `php artisan list make` en tu terminal.

Directorio Broadcasting

El directorio `Broadcasting` contiene todas las clases de broadcast de tu aplicación. Estas clases son generadas usando el comando `make:channel`. Este directorio no existe por defecto, pero será creado para ti cuando crees tu primer canal. Para aprender más sobre canales, revisa la documentación sobre [broadcasting de eventos](#).

El Directorio Console

El directorio `Console` contiene todos los comandos personalizados de Artisan para tu aplicación. Estos comandos pueden ser generados usando el comando `make:command`. Este directorio también almacena el kernel de tu consola, que es donde tus comandos personalizados de Artisan son registrados y tus [tareas programadas](#) son definidas.

Directorio Events

Este directorio no existe por defecto, pero será creado para ti por los comandos de Artisan `event:generate` y `make:event`. El directorio `Events` almacena [clases de eventos](#). Los eventos pueden ser usados para alertar a otras partes de tu aplicación que una acción dada ha ocurrido, proporcionando una gran cantidad de flexibilidad y desacoplamiento.

Directorio Exceptions

El directorio `Exceptions` contiene el manejador de excepciones de tu aplicación y también es un buen lugar para colocar cualquier excepción lanzada por tu aplicación. Si te gustaría personalizar cómo las excepciones son mostradas o renderizadas, debes modificar la clase `Handler` en este directorio.

Directorio Http

El directorio `Http` contiene tus controladores, middleware y form requests. Casi toda la lógica para manejar las solicitudes que llegan a tu aplicación serán colocadas en este directorio.

Directorio Jobs

Este directorio no existe por defecto, pero será creado para ti si ejecutas el comando de Artisan `make:job`. El directorio `Jobs` almacena las [colas de trabajos](#) para tu aplicación. Los trabajos pueden ser encolados por tu aplicación o ejecutados sincrónicamente dentro del ciclo de vida actual de la solicitud. Los trabajos que son ejecutados sincrónicamente durante la solicitud actual son algunas veces referidos como "comandos" dado que son una implementación del [patrón de comandos](#).

Directorio Listeners

Este directorio no existe por defecto, pero será creado para ti si ejecutas los comandos de Artisan `event:generate` o `make:listener`. El directorio `Listeners` contiene las clases que manejan tus `eventos`. Los listeners de eventos reciben una instancia de evento y realizan la lógica en respuesta al evento siendo ejecutado. Por ejemplo, un evento `UserRegistered` puede ser manejado por un listener `SendWelcomeEmail`.

Directorio Mail

Este directorio no existe por defecto, pero será creado para ti si ejecutas el comando de artisan `make:mail`. El directorio `Mail` contiene todas tus clases que representan correos electrónicos enviados por tu aplicación. Los objetos de correo te permiten encapsular toda la lógica para construir un correo en una única y sencilla clase y que puede ser enviado usando el método `Mail::send`.

Directorio Notifications

Este directorio no existe por defecto, pero será creado para ti si ejecutas el comando de Artisan `make:notification`. El directorio `Notifications` contiene todas las notificaciones "transaccionales" que son enviadas por tu aplicación, tales como notificaciones sencillas sobre eventos que ocurren dentro de tu aplicación. Las características de notificaciones de Laravel abstraen el envío de notificaciones sobre una variedad de drivers como email, Slack, SMS o almacenados en la base de datos.

Directorio Policies

Este directorio no existe por defecto, pero será creado para ti si ejecutas el comando de Artisan `make:policy`. El directorio `Policies` contiene las clases de las políticas de autorización de tu aplicación. Las políticas son usadas para determinar si un usuario puede realizar una acción dada contra un recurso. Para más información, revisa la [documentación sobre autorización](#).

Directorio Providers

El directorio `Providers` contiene todos los [proveedores de servicios](#) para tu aplicación. Los proveedores de servicios maquetan tu aplicación al enlazar servicios en el contenedor de servicios, registrando eventos o realizando cualquier otra tarea para preparar tu aplicación para solicitudes entrantes.

En una aplicación de Laravel nueva, este directorio ya contendrá algunos proveedores. Eres libre de agregar tus propios proveedores a este directorio según sea necesario.

Directorio Rules

Este directorio no existe por defecto, pero será creado para ti si ejecutas el comando de Artisan `make:rule`. El directorio `Rules` contiene los objetos para las reglas de validación personalizadas de tu aplicación. Las reglas son usadas para encapsular lógica de validación complicada en un simple objeto. Para más información, revisa la [documentación sobre validación](#).

Laravel Homestead

- [Introducción](#)
- [Instalación y configuración](#)
 - [Primeros pasos](#)
 - [Configurar Homestead](#)
 - [Iniciar el box de Vagrant](#)
 - [Instalación por proyecto](#)
 - [Instalando características opcionales](#)
 - [Alias](#)
- [Uso diario](#)
 - [Acceder a homestead globalmente](#)
 - [Conexión vía SSH](#)
 - [Conectar a base de datos](#)
 - [Respaldos de base de datos](#)
 - [Instantáneas de la base de datos](#)
 - [Agregar sitios adicionales](#)
 - [Variables de entorno](#)
 - [Configurar tareas programadas](#)
 - [Configurar mailhog](#)
 - [Configurar minio](#)
 - [Puertos](#)
 - [Compartir tu entorno](#)
 - [Múltiples versiones PHP](#)

- Servidores web
- Correo electrónico
- Depuración y perfilado
 - Depuración de solicitudes web con Xdebug
 - Depuración de aplicaciones CLI
 - Perfilado de aplicaciones con Blackfire
- Interfaces de red
- Extender Homestead
- Actualizar Homestead
- Configuraciones específicas de proveedor
 - VirtualBox

Introducción

Laravel se ha esforzado en hacer que toda la experiencia del desarrollo de PHP sea placentera, incluyendo el entorno de desarrollo local. [Vagrant](#) provee una manera simple y elegante de administrar y provisionar máquinas virtuales.

Laravel Homestead es el box de Vagrant pre-empaquetado oficial que brinda un maravilloso entorno de desarrollo sin la necesidad de que tengas que instalar PHP, un servior web, ni ningún otro servidor de software en tu máquina local. ¡Basta de preocuparte por estropear tu sistema operativo! Los boxes de Vagrant son completamente desecharables. Si algo sale mal, simplemente puedes destruir y volver a crear el box en cuestión de minutos.

Homestead puede ejecutarse en sistemas Windows, Mac y Linux e incluye el Nginx, PHP, MySQL, PostgreSQL, Redis, Memcached, Node y todas las demás herramientas que necesitas para desarrollar aplicaciones de Laravel sorprendentes.

Nota

Si estás utilizando Windows, puede que necesites habilitar la virtualización por hardware (VT-x). Usualmente puede habilitarse en el BIOS. Si estás utilizando Hyper-V en un sistema UEFI puedes que requieras también deshabilitar Hyper-V para poder acceder a VT-x.

Software incluido

- Ubuntu 18.04

- Git
- PHP 7.4
- PHP 7.3
- PHP 7.2
- PHP 7.1
- PHP 7.0
- PHP 5.6
- Nginx
- MySQL
- Sqlite3
- PostgreSQL
- Composer
- Node (Con Yarn, Bower, Grunt, y Gulp)
- Redis
- Memcached
- Beanstalkd
- Mailhog
- ngrok
- wp-cli
- Minio

Software opcional

- Apache
- Blackfire
- Cassandra
- Chronograf
- CouchDB
- Crystal & Lucky Framework
- Docker
- Elasticsearch
- Gearman
- Go
- Grafana
- InfluxDB
- MariaDB

- MinIO
- MongoDB
- MySQL 8
- Neo4j
- Oh My Zsh
- Ruby & Rails
- Open Resty
- PM2
- Python
- RabbitMQ
- Solr
- Webdriver & Laravel Dusk Utilities
- Zend Z-Ray

Instalación y configuración

Primeros pasos

Antes de iniciar tu entorno de Homestead, debes instalar [VirtualBox 6.x](#), [VMware](#), [Parallels](#) o [Hyper-V](#) además de [Vagrant](#). Todos estos paquetes de software cuentan con un instalador fácil de usar para todos los sistemas operativos populares.

Para utilizar el proveedor de VMWare, necesitarás comprar tanto VMWare Fusion / Workstation y el [plugin de Vagrant para VMWare](#). A pesar de que esto no es gratuito, VMWare ofrece un mayor desempeño en velocidad al compartir directorios.

Para utilizar el proveedor de Parallels, debes instalar el [plugin de Vagrant para Parallels](#). Es totalmente gratuito.

Debido a las [limitaciones de Vagrant](#), el proveedor de Hyper-V ignora todas las configuraciones de red.

Instalar el Box de Vagrant para Homestead

Una vez que estén instalados VirtualBox / VMWare y Vagrant, deberás añadir el box `laravel/homestead` a tu instalación de Vagrant ejecutando el siguiente comando en la terminal. Esto tomará algunos minutos para descargar el box, dependiendo de tu velocidad de internet:

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

php

Si el comando falla, asegúrate de que tu instalación de Vagrant esté actualizada.

Instalar Homestead

Puedes instalar Homestead clonando el repositorio en tu máquina host. Considera clonar el repositorio en una carpeta `Homestead` dentro de tu directorio "home", ya que el box de Homestead actuará como host para todos tus proyectos de Laravel:

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

php

Debes hacer checkout a alguna versión etiquetada de Homestead ya que la rama `master` no siempre es estable. Puedes encontrar la versión estable más reciente en la [Página de Releases de GitHub](#). De forma alternativa puedes revisar el branch `release` el cual siempre es actualizado con la última versión estable.

```
cd ~/Homestead
```

php

```
// Checkout the stable "release" branch  
git checkout release
```

Una vez que hayas clonado el repositorio, ejecuta el comando `bash init.sh` desde el directorio Homestead para crear el archivo de configuración `Homestead.yaml`. El archivo `Homestead.yaml` estará situado en el directorio Homestead:

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

php

Configurar Homestead

Especificando tu proveedor

La clave `provider` en tu archivo `Homestead.yaml` indica cuál proveedor de Vagrant será utilizado: `virtualbox`, `vmware_fusion`, `vmware_workstation`, `parallels` o `hyperv`. Puedes

especificar en esta opción el proveedor de tu preferencia.

```
provider: virtualbox
```

php

Configurar directorios compartidos

La propiedad `folders` del archivo `Homestead.yaml` lista todos los directorios que deseas compartir con tu entorno de Homestead. A medida que los archivos dentro de estos directorios cambien, estos se mantendrán sincronizados con tu máquina local y el entorno de Homestead. Puedes configurar tantos directorios compartidos como sean necesarios:

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

php

Nota

Los usuarios de Windows no deben usar la sintaxis de ruta `~/` y en su lugar deberían usar la ruta completa al proyecto como `C:\Users\user\Code\project1`.

Siempre debes mapear proyectos individuales en su propio directorio en lugar de mapear tu directorio `~/code` entero. Cuando mapeas un directorio la maquina virtual debe mantener un seguimiento de todos los I/O que suceden en *cada* archivo en el directorio incluso si no estás trabajando en ese proyecto. Esto puede llevar a que sucedan problemas de rendimiento si tienes un gran número de archivos en un directorio. Virtualbox es más susceptible a esto que otros proveedores, lo que significa que es muy importante dividir tus directorios en proyectos individuales al usar el proveedor Virtualbox.

```
folders:  
  - map: ~/code/project1  
    to: /home/vagrant/project1  
  
  - map: ~/code/project2  
    to: /home/vagrant/project2
```

php

Nota

Nunca debes montar `.` (el directorio actual) al usar Homestead. Esto causa que Vagrant no mapee el directorio actual a `/vagrant` y romperá características adicionales además de causar resultados inesperados al momento del aprovisionamiento.

Para habilitar NFS[↗], solo necesitarás agregar un simple flag en la configuración de tu directorio sincronizado:

```
folders:  
  - map: ~/code/project1  
    to: /home/vagrant/project1  
    type: "nfs"
```

Nota

Cuando uses NFS en Windows, debes considerar instalar el plugin `vagrant-winnfsd`[↗]. Este plugin mantendrá correctamente el usuario / grupo para los archivos y directorios dentro del box de Homestead.

También puedes indicar cualquier opción soportada por los [Directarios Sincronizados](#)[↗] de Vagrant, listándolos bajo la clave `options` :

```
folders:  
  - map: ~/code/project1  
    to: /home/vagrant/project1  
    type: "rsync"  
    options:  
      rsync__args: ["--verbose", "--archive", "--delete", "-zz"]  
      rsync__exclude: ["node_modules"]
```

Configurar sitios de Nginx

¿No estás familiarizado con Nginx? No hay problema. La propiedad `sites` te permitirá mapear un “dominio” a un directorio en tu entorno de Homestead de manera sencilla. Una configuración simple de un sitio está incluido en el archivo `Homestead.yaml`. Nuevamente, podrás añadir tantos sitios a tu entorno de Homestead como sea necesario. Homestead puede funcionar como un conveniente entorno virtualizado para cada proyecto de Laravel en el que estés trabajando:

```
sites:  
  - map: homestead.test  
    to: /home/vagrant/project1/public
```

php

Si cambias la propiedad `sites` apropiadamente después de haber provisionado el box de Homestead, deberás volver a ejecutar `vagrant reload --provision` para actualizar la configuración de Nginx en la máquina virtual.

Nota

Recuerda que aunque los scripts de Homestead son construidos tan idempotente como sea posible, si estás experimentando problemas al momento del aprovisionamiento destruye la máquina usando `vagrant destroy && vagrant up` para reconstruir desde un estado correcto conocido.

El archivo hosts

Debes agregar los "dominios" para tus sitios de Nginx en el archivo `hosts` en tu máquina. El archivo `hosts` va a redirigir las peticiones de los sitios Homestead hacia tu máquina Homestead. En Mac y Linux, este archivo está ubicado en `/etc/hosts`. En Windows, este archivo está ubicado en `C:\Windows\System32\drivers\etc\hosts`. Las líneas que agregues a este archivo deberán verse de la siguiente forma:

```
192.168.10.10 homestead.test
```

php

Debes asegurarte de que la IP indicada sea la misma que está en el archivo `Homestead.yaml`. Una vez que hayas añadido el dominio a tu archivo `hosts` y hayas iniciado el box de Vagrant podrás acceder al sitio desde el navegador web:

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

php

Iniciando el box de Vagrant

Una vez que hayas editado el archivo `Homestead.yaml` a tu gusto, ejecuta el comando `vagrant up` desde tu directorio Homestead. Vagrant va a iniciar la máquina virtual y a configurar

automáticamente tus directorios compartidos y sitios de Nginx.

Para destruir la máquina, debes utilizar el comando `vagrant destroy --force`.

Instalación por proyecto

En lugar de instalar Homestead globalmente y compartir el mismo box de Homestead para todos tus proyectos, también es posible configurar una instancia de Homestead para cada proyecto que necesites. Instalar Homestead por proyecto puede ser beneficioso si deseas crear un `Vagrantfile` en tu proyecto, permitiendo así a otras personas trabajar en el mismo proyecto ejecutando simplemente `vagrant up`.

Para instalar Homestead directamente en tu proyecto, debes hacerlo por medio de Composer:

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

php

Una vez que Homestead haya sido instalado, usa el comando `make` para generar el archivo `Vagrantfile` y `Homestead.yaml` en la raíz de tu proyecto. El comando `make` configurará automáticamente las directivas `sites` y `folders` en el archivo `Homestead.yaml`.

Mac / Linux:

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

php

Windows:

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

php

Después, ejecuta el comando `vagrant up` en tu terminal y podrás acceder a tu proyecto desde el navegador en `http://homestead.test`. Recuerda que aún vas a necesitar agregar una entrada para `homestead.test` en tu archivo `/etc/hosts` para el dominio de tu elección.

Instalando características opcionales

Software opcional es instalado usando la opción "features" en tu archivo de configuración de Homestead. La mayoría de características son habilitadas o deshabilitadas con un valor booleano, mientras que algunas características soportan múltiples opciones de configuración:

```
features:
  - blackfire:
      server_id: "server_id"
      server_token: "server_value"
      client_id: "client_id"
      client_token: "client_value"
  - cassandra: true
  - chronograf: true
  - couchdb: true
  - crystal: true
  - docker: true
  - elasticsearch:
      version: 7
  - gearman: true
  - golang: true
  - grafana: true
  - influxdb: true
  - mariadb: true
  - minio: true
  - mongodb: true
  - mysql8: true
  - neo4j: true
  - ohmyzsh: true
  - openresty: true
  - pm2: true
  - python: true
  - rabbitmq: true
  - solr: true
  - webdriver: true
```

MariaDB

Activar MariaDB eliminará MySQL e instalará MariaDB. MariDB funciona como reemplazo de MySQL así que aún serás capaz de usar el driver de base de datos `mysql` en la configuración de la base de datos de tu aplicación.

MongoDB

La instalación por defecto establecerá el nombre de usuario de base de datos a `homestead` y su contraseña como `secret`.

Elasticsearch

Puedes especificar una versión soportada de Elasticsearch, la cual puede ser una versión principal o un número de versión exacto (major.minor.patch). La instalación por defecto creará un cluster llamado 'homestead'. Nunca debes darle a Elasticsearch más de la mitad de la memoria del sistema operativo, así que asegurate de que tu maquina Homestead al menos tiene el doble de la cantidad asignada a Elasticsearch.

TIP

Echa un vistazo a la [documentación de Elasticsearch](#) para aprender a personalizar tu configuración.

Neo4j

[Neo4j](#) es un sistema de manejo de bases de datos gráfico. Para instalar Neo4j Community Edition, actualiza tu archivo `Homestead.yaml` con la siguiente opción de configuración:

```
neo4j: true
```

php

La instalación por defecto establecerá el nombre de usuario de base de datos como `homestead` y su contraseña como `secret`. Para acceder al navegador Neo4j, visita `http://homestead.test:7474` mediante tu navegador. Los puertos `7687` (Bolt), `7474` (HTTP), y `7473` (HTTPS) están listos para manejar peticiones desde el cliente Neo4j.

Aliases

Puedes añadir alias de Bash a tu máquina de Homestead modificando el archivo `aliases` desde tu directorio de Homestead:

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

php

Después de haber actualizado el archivo `aliases`, debes volver a provisionar la máquina de Homestead usando el comando `vagrant reload --provision`. Con esto te podrás asegurar de que tus nuevos alias estén disponibles en la máquina.

Uso diario

Acceder a Homestead globalmente

En ocasiones puede que requieras de iniciar Homestead con el comando `vagrant up` desde cualquier parte en tu sistema de archivos. Esto es posible en sistemas Mac / Linux al agregar una función Bash en tu Bash Profile. En Windows, esto puede lograrse al agregar un archivo "batch" en tu `PATH`. Estos scripts te permitirán ejecutar cualquier comando de Vagrant desde cualquier parte en tu sistema y automáticamente apuntarán el comando hacia tu instalación de Homestead:

Mac / Linux

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

php

Asegúrate de modificar la ruta `~/Homestead` en la función hacia la ubicación actual de tu instalación de Homestead. Una vez que hayas instalado la función, podrás ejecutar comandos como `homestead up` o `homestead ssh` desde cualquier parte en tu sistema de archivos.

Windows

Crea un archivo batch llamado `homestead.bat` en algua parte de tu equipo con los siguientes comandos:

```
@echo off  
  
set cwd=%cd%  
set homesteadVagrant=C:\Homestead  
  
cd /d %homesteadVagrant% && vagrant %*  
cd /d %cwd%  
  
set cwd=  
set homesteadVagrant=
```

php

Asegúrate de modificar la ruta de ejemplo `C:\Homestead` en el script por la ruta actual de tu instalación de Homestead. Después de crear el archivo, agrega la ubicación a tu `PATH`. Hecho esto podrás ejecutar comandos como `homestead up` o `homestead ssh` desde cualquier lado en tu sistema.

Conexión vía SSH

Puedes conectarte a tu máquina virtual por medio de SSH haciendo uso del comando `vagrant ssh` en la terminal desde tu directorio Homestead.

Pero, dado que probablemente requieras conectarte frecuentemente a tu máquina de Homestead, deberías considerar agregar la "función" descrita anteriormente en tu equipo host para poder conectarte de manera rápida a tu box de Homestead por medio de SSH.

Conectarse a la base de datos

Una base de datos `homestead` es configurada por defecto tanto para MySQL como para PostgreSQL. Para conectarte a tu base de datos de MySQL o de PostgreSQL desde el cliente de base de datos de tu equipo host, deberás conectarte hacia `127.0.0.1` en el puerto `33060` (MySQL) o `54320` (PostgreSQL). El nombre de usuario y contraseña para ambas bases de datos son `homestead` / `secret`.

Nota

Solo deberías utilizar estos puertos no estándares para conectarte a tus bases de datos desde tu equipo host. Deberás utilizar los puertos por defecto 3306 y 5432 en tu archivo de configuración para la base de datos de Laravel que se encuentra ejecutándose *dentro* de la máquina virtual.

Respaldos de base de datos

Homestead puede hacer respaldos de tu base de datos automáticamente cuando tu box de Vagrant es destruida. Para utilizar esta característica, debes estar usando Vagrant 2.1.0 o una versión superior. O, si estás usando una versión inferior, debes instalar el plugin `vagrant-triggers`. Para activar los respaldos de base de datos automáticos, agrega la siguiente línea a tu archivo `Homestead.yaml`:

```
backup: true
```

php

Una vez esté configurado, Homestead exportará tus bases de datos a los directorios `mysql_backup` y `postgres_backup` cuando se ejecute el comando `vagrant destroy`. Estos directorios pueden ser encontrados en la carpeta donde clonaste Homestead o en el root de tu proyecto si estás usando el método [instalación por proyecto](#).

Instantáneas de la base de datos

Homestead admite la congelación del estado de las bases de datos MySQL y MariaDB y se ramifica entre ellas con [Logical MySQL Manager](#). Por ejemplo, imagine trabajar en un sitio con una base de datos de varios gigabytes. Puede importar la base de datos y tomar una instantánea. Después de realizar un trabajo y crear un contenido de prueba localmente, puede restaurar rápidamente al estado original.

Por debajo, LMM usa la funcionalidad de instantáneas delgadas de LVM con soporte de copia en escritura. En la práctica, esto significa que el cambio de una sola fila en una tabla solo hará que los cambios que realice se escriban en el disco, ahorrando tiempo y espacio de disco significativos durante las restauraciones.

Como `lmm` interactúa con LVM, debe ejecutarse como root. Para ver todos los comandos disponibles, ejecute `sudo lmm` dentro de la caja de vagrant. Un flujo de trabajo común sería:

1. Importe una base de datos a la rama predeterminada de `master` `lmm`.
2. Guarde una instantánea de la base de datos sin cambios usando `sudo lmm branch prod-YYYY-MM-DD`.
3. Modifica la Base de Datos.
4. Ejecute `sudo lmm merge prod-YYYY-MM-DD` para deshacer todos los cambios.
5. Ejecute `sudo lmm delete <branch>` para eliminar todas las ramas que no se necesiten.

Agregar sitios adicionales

Una vez que tu entorno de Homestead haya sido provisionado y esté en ejecución, es probable que requieras agregar sitios adicionales de Nginx para tu aplicación de Laravel. Puedes ejecutar tantas instalaciones de Laravel como deseas, simplemente debes añadirlas a tu archivo `Homestead.yaml`.

```
sites:  
  - map: homestead.test  
    to: /home/vagrant/project1/public  
  - map: another.test  
    to: /home/vagrant/project2/public
```

php

Si vagrant no está manejando tu archivo "hosts" de manera automática, también deberás agregar los nuevos sitios a este archivo.

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

php

Una vez que el sitio ha sido agregado, ejecuta el comando `vagrant reload --provision` desde tu directorio de Homestead.

Tipos de sitios

Homestead soporta varios tipos de sitios permitiéndote ejecutar fácilmente proyectos que no estén basados en Laravel. Por ejemplo, puedes agregar fácilmente una aplicación de Symfony en Homestead utilizando el tipo de sitio `symfony2` :

```
sites:  
- map: symfony2.test  
  to: /home/vagrant/my-symfony-project/web  
  type: "symfony2"
```

php

Los tipos de sitios disponibles son: `apache` , `apigility` , `expressive` , `laravel` (por defecto), `proxy` , `silverstripe` , `statamic` , `symfony2` , `symfony4` y `zf` .

Parámetros de los Sitios

También puedes agregar valores adicionales de `fastcgi_param` en Nginx para tus sitios por medio de la directiva `params` en el sitio. Por ejemplo, agregar el parámetro `FOO` con el valor de `BAR` :

```
sites:  
- map: homestead.test  
  to: /home/vagrant/project1/public  
  params:  
    - key: FOO  
      value: BAR
```

php

Variables de entorno

Puedes especificar variables de entorno globales al agregarlas en tu archivo `Homestead.yaml` :

```
variables:  
  - key: APP_ENV  
    value: local  
  - key: FOO  
    value: bar
```

php

Después de actualizar el archivo `Homestead.yaml`, deberás volver a provisionar la máquina ejecutando el comando `vagrant reload --provision`. Esto actualizará la configuración de PHP-FPM para todas las versiones instaladas de PHP y también actualizará el entorno para el usuario `vagrant`.

Configurar tareas programadas

Laravel proporciona una manera conveniente de ejecutar [tareas programadas](#) al configurar las tareas por medio del comando de Artisan `schedule:run` para que se ejecute cada minuto. El comando `schedule:run` va a examinar las tareas programadas definidas en tu clase `App\Console\Kernel` para determinar cuáles tareas deben ser ejecutadas.

Si deseas que el comando `schedule:run` sea ejecutado en un sitio de Homestead, debes indicar la opción `schedule` como `true` cuando definas el sitio:

```
sites:  
  - map: homestead.test  
    to: /home/vagrant/project1/public  
    schedule: true
```

php

La tarea programada para este sitio estará definida en el directorio `/etc/cron.d` de tu máquina virtual.

Configuración de mailhog

Mailhog te permite capturar fácilmente el correo saliente y examinarlo sin que éste sea enviado hacia sus destinatarios. Para comenzar, actualiza tu archivo `.env` con la siguiente configuración:

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

php

```
MAIL_PASSWORD=null  
MAIL_ENCRYPTION=null
```

Una vez que Mailhog ha sido configurado, puedes acceder al dashboard de Mailhog en <http://localhost:8025>.

Configuración de minio

Minio es un servidor de almacenamiento de objetos de código libre con una API compatible con Amazon S3. Para instalar Minio, actualiza tu archivo `Homestead.yaml` con la siguiente opción de configuración:

```
minio: true
```

php

Por defecto, Minio está disponible en el puerto 9600. Puedes acceder al panel de control de Minio visitando <http://localhost:9600/>. La clave de acceso por defecto es `homestead`, mientras que la clave secreta por defecto es `secretkey`. Al acceder a Minio, siempre debes usar la región `us-east-1`.

Para usar Minio necesitarás ajustar la configuración de disco S3 en tu archivo

`config/filesystems.php` Necesitarás añadir la opción `use_path_style_endpoint` a la configuración del disco, así como cambiar la clave `url` a `endpoint`:

```
's3' => [  
    'driver' => 's3',  
    'key' => env('AWS_ACCESS_KEY_ID'),  
    'secret' => env('AWS_SECRET_ACCESS_KEY'),  
    'region' => env('AWS_DEFAULT_REGION'),  
    'bucket' => env('AWS_BUCKET'),  
    'endpoint' => env('AWS_URL'),  
    'use_path_style_endpoint' => true  
]
```

php

Por último, asegúrate de que tu archivo `.env` tenga las siguientes opciones:

```
AWS_ACCESS_KEY_ID=homestead  
AWS_SECRET_ACCESS_KEY=secretkey  
AWS_DEFAULT_REGION=us-east-1  
AWS_URL=http://localhost:9600
```

php

Para proveer buckets, agrega una directiva `buckets` a tu archivo de configuración Homestead:

```
buckets:  
  - name: your-bucket  
    policy: public  
  - name: your-private-bucket  
    policy: none
```

Los valores soportados por `policy` incluyen: `none`, `download`, `upload` y `public`.

Puertos

Por defecto, los siguientes puertos están redirigidos a tu entorno de Homestead:

- **SSH:** 2222 → Redirige a 22
- **ngrok UI:** 4040 → Redirige a 4040
- **HTTP:** 8000 → Redirige a 80
- **HTTPS:** 44300 → Redirige a 443
- **MySQL:** 33060 → Redirige a 3306
- **PostgreSQL:** 54320 → Redirige a 5432
- **MongoDB:** 27017 → Redirige a 27017
- **Mailhog:** 8025 → Redirige a 8025
- **Minio:** 9600 → Redirige a 9600

Redirigir Puertos Adicionales

Si lo deseas, puedes redirigir puertos adicionales a tu box de Vagrant, así como su protocolo:

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

Compartir tu entorno

En ocasiones, podrás requerir compartir lo que estás haciendo con algún compañero de trabajo o algún cliente. Vagrant tiene incorporado una manera de hacer esto por medio del comando `vagrant share`; sin embargo, esto no funcionará si se tienen configurados múltiples sitios en tu archivo

`Homestead.yaml`.

Para resolver este problema, Homestead incluye su propio comando `share`. Para utilizarlo, conectate por SSH a tu máquina virtual de Homestead con el comando `vagrant ssh` y ejecuta el comando `share homestead.test`. Esto va a compartir el sitio `homestead.test` especificado en el archivo de configuración `Homestead.yaml`. Puedes sustituir el nombre del sitio en lugar de utilizar `homestead.test`.

```
share homestead.test
```

php

Después de ejecutar este comando, podrás ver que aparece una ventana de Ngrok, la cual contiene el log de actividad y las URLs accesibles de manera pública para el sitio compartido. Si deseas especificar una región personalizada, un subdominio o el tiempo de ejecución de Ngrok puedes hacerlo desde el comando `share`:

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

php

Nota

Recuerda, Vagrant es inherentemente inseguro y estarás compartiendo tu máquina virtual en Internet cuando ejecutes el comando `share`.

Múltiples versiones de PHP

Homestead 6 introduce soporte para múltiples versiones de PHP en una misma máquina virtual. Puedes especificar qué versión de PHP deseas utilizar para un sitio en particular desde tu archivo

`Homestead.yaml`. Las versiones disponibles de PHP son "5.6", "7.0", "7.1", "7.2", "7.3" y "7.4" (por defecto):

```
sites:  
  - map: homestead.test  
    to: /home/vagrant/project1/public  
    php: "7.1"
```

php

Además, puedes utilizar cualquiera de las versiones soportadas de PHP desde el CLI:

```
php5.6 artisan list  
php7.0 artisan list  
php7.1 artisan list  
php7.2 artisan list  
php7.3 artisan list  
php7.4 artisan list
```

php

También puedes actualizar la versión por defecto de la línea de comandos ejecutando los siguientes comandos dentro de tu máquina virtual de Homestead:

```
php56  
php70  
php71  
php72  
php73  
php74
```

php

Servidores web

Homestead utiliza por defecto el servidor web Nginx. Sin embargo, también se puede instalar Apache si se especifica el tipo de sitio como `apache`. Ambos servidores pueden instalarse al mismo tiempo, pero no pueden *ejecutarse* al mismo tiempo. El comando `flip` está disponible en el shell para facilitar el proceso de cambiar entre servidores web. El comando `flip` automáticamente va a determinar cuál servidor web está en ejecución, después lo va a detener y por último va a iniciar el otro servidor. Para utilizar este comando, primero deberás conectarte a la máquina virtual de Homestead por medio de SSH y ejecutar el comando en la terminal:

```
flip
```

php

Correo electrónico

Homestead incluye el agente de transferencia de correo Postfix, que está escuchando por defecto en el puerto `1025`. Así que puedes indicarle a tu aplicación que use el controlador de correo `smtp` en el puerto `1025` de `localhost`. Entonces, todos los correos enviados serán manejados por Postfix y atrapados por Mailhog. Para ver tus correos enviados, abre en tu navegador <http://localhost:8025>.

Depuración y perfilado

Depuración de solicitudes web con Xdebug

Homestead incluye soporte para la depuración por pasos usando [Xdebug](#). Por ejemplo, puede cargar una página web desde un navegador y PHP se conectará nuevamente a su IDE para permitir la inspección y modificación del código en ejecución.

Para habilitar la depuración, ejecute los siguientes comandos dentro de la caja vagrant:

```
$ sudo phpenmod xdebug  
  
# Update this command to match your PHP version...  
$ sudo systemctl restart php7.3-fpm
```

Luego, siga las instrucciones de su IDE para habilitar la depuración. Finalmente, configure su navegador para activar Xdebug con una extensión o [bookmarklet](#).

Nota

Xdebug puede ralentizar significativamente a PHP. Para deshabilitar Xdebug, ejecute `sudo phpdismod xdebug` dentro de su caja de Vagrant y reinicie el servicio FPM nuevamente.

Depuración de aplicaciones CLI

Para depurar a aplicación PHP, use el alias de shell `xphp` dentro de la caja Vagrant:

```
$ xphp path/to/script
```

Auto iniciando Xdebug

Al depurar pruebas funcionales que realizan solicitudes al servidor web, es más fácil iniciar automáticamente la depuración en lugar de modificar las pruebas para pasar a través de un encabezado personalizado o una cookie para activar la depuración. Para forzar el inicio de Xdebug, modifique `/etc/php/7.x/Fpm/conf.d/20-xdebug.ini` dentro de su caja Vagrant y agregue la siguiente configuración:

```
; If Homestead.yaml contains a different subnet for the ip address, this address  
xdebug.remote_host = 192.168.10.1  
xdebug.remote_autostart = 1
```

php

Perfilar aplicaciones con Blackfire

Blackfire [☞](#) es un servicio SaaS para perfilar solicitudes Web, aplicaciones de línea de comandos y escribir aserciones de rendimiento. Ofrece una interfaz de usuario interactiva que muestra datos de perfiles en call-graphs y líneas de tiempo. Está construido para ser usado en desarrollo, staging y producción, sin sobrecarga para usuarios finales. Proporciona comprobaciones de rendimiento, calidad y seguridad del código y opciones de configuración para `php.ini`.

Blackfire Player [☞](#) es una aplicación de Web Crawling, Pruebas Web y Web Scraping que puede funcionar junto con Blackfire para perfilar escenarios.

```
features:  
  - blackfire:  
      server_id: "server_id"  
      server_token: "server_value"  
      client_id: "client_id"  
      client_token: "client_value"
```

php

Las credenciales de servidor y credenciales de clientes [requieren una cuenta de usuario](#) [☞](#). Blackfire ofrece varias opciones para perfilar una aplicación, incluyendo una herramienta de línea de comandos y extensión de navegador. Por favor [revisa la documentación de Blackfire para más detalles](#) [☞](#).

Perfilando el Rendimiento de PHP usando XHGui

XHGui [☞](#) es una interfaz de usuario para explorar el rendimiento de sus aplicaciones PHP. Para habilitar XHGui, agregue `xhgui: 'true'` a la configuración de su sitio:

```
sites:  
  -  
    map: your-site.test  
    to: /home/vagrant/your-site/public  
    type: "apache"  
    xhgui: 'true'
```

php

Si el sitio ya existe, asegúrese de ejecutar `vagrant provision` después de actualizar su configuración.

Para perfilar una solicitud web, agregue `xhgui = on` como parámetro de consulta a una solicitud. XHGui automáticamente adjuntará una cookie a la respuesta para que las solicitudes posteriores no necesiten el valor de la cadena de consulta. Puede ver los resultados de su perfil de aplicación navegando a `http://your-site.test/xhgui`.

Para perfilar una solicitud de CLI usando XHGui, prefije el comando con `XHGUI=on`:

```
XHGUI=on path/to/script
```

php

Los resultados del perfil CLI pueden ser vistos en la misma forma como los resultados del perfil web.

Tenga en cuenta que el acto de perfilar ralentiza la ejecución del script y los tiempos absolutos pueden ser el doble de las solicitudes del mundo real. Por lo tanto, siempre compare el porcentaje de las mejoras y no los números absolutos. Además, tenga en cuenta que el tiempo de ejecución (o "Tiempo de pared") incluye cualquier tiempo que se pase en pausa en un depurador.

Desde que los perfiles de rendimiento ocupan un espacio de disco significativo, se eliminan automáticamente después de unos días.

Interfaces de red

La propiedad `networks` del archivo `Homestead.yaml` configura las interfaces de red de tu entorno Homestead. Puedes configurar tantas interfaces como sea necesario:

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

php

Para habilitar una interfaz en `puente`, debes indicar la propiedad `bridge` y cambiar el tipo de red a `public_network`:

```
networks:  
  - type: "public_network"
```

php

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

Para habilitar **DHCP**, solo debes remover la opción `ip` de tu configuración:

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

php

Extender Homestead

Puedes extender Homestead usando el script `after.sh` en la raíz de tu directorio Homestead. Dentro de este archivo, puedes agregar cualquier comando shell que sea necesario para configurar y personalizar apropiadamente tu máquina virtual.

Al personalizar Homestead, Ubuntu puede preguntar si deseas conservar la configuración original de un paquete o sobreescribirla con un nuevo archivo de configuración. Para evitar esto, debes usar el siguiente comando al instalar paquetes para evitar sobreescribir cualquier configuración escrita previamente por Homestead:

```
sudo apt-get -y \  
  -o Dpkg::Options::="--force-confdef" \  
  -o Dpkg::Options::="--force-confold" \  
  install your-package
```

php

Personalizaciones de usuario

Al usar Homestead en un ambiente de equipo, puedes querer configurar Homestead para que se ajuste mejor a tu estilo de desarrollo personal. Puedes crear un archivo `user-customizations.sh` en la raíz de tu directorio Homestead (el mismo directorio que contiene tu `Homestead.yaml`). Dentro de este archivo, puedes hacer cualquier personalización que quieras; sin embargo, `user-customizations.sh` no debe ser versionado.

Actualizar Homestead

Antes de comenzar a actualizar Homestead asegurate de ejecutar `vagrant destroy` para eliminar tu maquina virtual actual. Puedes actualizar Homestead en algunos sencillos pasos. Primero, debes actualizar el box de Homestead utilizando el comando `vagrant box update` :

```
vagrant box update
```

php

Después, debes actualizar el código fuente de Homestead. Si clonaste el repositorio puedes ejecutar los siguientes comandos en la ubicación donde clonaste originalmente el repositorio:

```
git fetch
```

php

```
git pull origin release
```

Estos comandos traen el código más reciente de Homestead del repositorio de GitHub, recuperan las últimas etiquetas y luego revisan la última versión etiquetada. Puede encontrar la última versión de lanzamiento estable en la [página de lanzamientos de GitHub](#).

Si realizaste la instalación de Homestead en tu proyecto por medio del archivo `composer.json`, debes asegurarte de que tu archivo `composer.json` contenga la dependencia `"laravel/homestead": "^10"` y después debes actualizar dichas dependencias:

```
composer update
```

php

Finalmente, debes destruir y regenerar tu box de Homestead para utilizar la última instalación de Vagrant. Para lograr esto, ejecuta los siguientes comandos en tu directorio de Homestead:

```
vagrant destroy
```

php

```
vagrant up
```

Configuraciones específicas de proveedor

VirtualBox

`natdnshostresolver`

Por defecto, Homestead configura la opción `natdnshostresolver` como `on`. Esto permite a Homestead utilizar la configuración del DNS de tu sistema operativo. Si lo deseas, puedes sobrescribir este comportamiento, agregando la siguiente línea al archivo `Homestead.yaml`:

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

php

Enlaces simbólicos en Windows

Si los enlaces simbólicos no funcionan correctamente en equipos Windows, puede que requieras agregar el siguiente bloque a tu `Vagrantfile`:

```
config.vm.provider "virtualbox" do |v|  
  v.customize ["setextradata", :id, "VBoxInternal2/SharedFoldersEnableSymlinks  
end
```

php

Laravel Valet

- [Introducción](#)
 - [Valet o Homestead](#)
- [Instalación](#)
 - [Actualización](#)
- [Activar sitios](#)
 - [El comando "Park"](#)
 - [El comando "Link"](#)
 - [Asegurar sitios con TLS](#)
- [Compartir sitios](#)
- [Variables de entorno específicas del sitio](#)

- Drivers de Valet personalizados
 - Drivers locales
- Otros comandos de Valet
- Archivos y directorios de Valet

Introducción

Valet es un entorno de desarrollo de Laravel para Mac. No requiere de Vagrant ni de modificar el archivo de configuración `/etc/hosts`. Incluso permite compartir tus sitios a través de túneles locales. *Genial ¿Verdad?*

Laravel Valet configura tu Mac para que siempre inicie el servicio de [Nginx](#) en segundo plano al iniciar tu computadora. Después, [DnsMasq](#) actuará como servidor proxy, procesando todas las peticiones en el dominio `*.test` apuntando a los sitios instalados en tu computadora local.

En otras palabras, es un entorno de desarrollo de Laravel sorprendentemente rápido y solamente utiliza cerca de 7MB de RAM. Laravel Valet no está pensado para ser un reemplazo de Vagrant y Homestead, en su lugar presenta una alternativa flexible y rápida, lo cual es una buena opción para quienes tengan una cantidad limitada de RAM.

Por defecto, Valet brinda soporte para las siguientes tecnologías, pero no está limitado a sólo ellas:

- [Laravel](#)
- [Lumen](#)
- [Bedrock](#)
- [CakePHP 3](#)
- [Concrete5](#)
- [Contao](#)
- [Craft](#)
- [Drupal](#)
- [Jigsaw](#)
- [Joomla](#)
- [Katana](#)
- [Kirby](#)
- [Magento](#)
- [OctoberCMS](#)
- [Sculpin](#)

- [Slim](#)
- [Statamic](#)
- [Static HTML](#)
- [Symfony](#)
- [WordPress](#)
- [Zend](#)

Además, es posible extender Valet con tu propio [driver personalizado](#).

Valet o Homestead

Como sabrás, Laravel ofrece Homestead, otro entorno de desarrollo local de Laravel. Homestead y Valet difieren en cuanto a la audiencia a la que están pensados y su aproximación al desarrollo local.

Homestead ofrece toda una máquina virtual de Ubuntu con Nginx instalado y configurado. Homestead es una muy buena elección si deseas tener un entorno de desarrollo virtualizado de Linux o si te encuentras trabajando con Windows / Linux.

Por otro lado, Valet solamente es soportado por Mac y requiere que instales PHP y un servidor de base de datos directamente en tu equipo local. Esto puede lograrse fácilmente haciendo uso de [Homebrew](#) con comandos como `brew install php` y `brew install mysql`. Valet proporciona un entorno de desarrollo local bastante rápido haciendo un uso mínimo de consumo de recursos, lo cual es genial para desarrolladores que solamente requieran de PHP / MySQL y no necesiten de todo un entorno virtualizado de desarrollo.

Tanto Valet como Homestead son buenas elecciones para configurar tu entorno de desarrollo de Laravel. El que sea que vayas a elegir depende completamente de tu gusto personal o las necesidades de tu equipo.

Instalación

Valet requiere de macOS y Homebrew. Antes de comenzar, asegurate de que ningún otro programa como Apache o Nginx esté utilizando el puerto 80 de tu computadora.

- Instala o actualiza [Homebrew](#) a su última versión con `brew update`.
- Instala PHP 7.4 usando Homebrew con `brew install homebrew/php/php`.
- Instala [Composer](#).
- Instala Valet por medio de Composer con `composer global require laravel/valet`.
Asegúrate de que el directorio `~/.composer/vendor/bin` se encuentre en el "PATH" de tu

sistema.

- Ejecuta el comando `valet install`. Esto va a configurar e instalar Valet y DnsMasq y va a registrar el daemon de Valet para que se inicie junto con el sistema operativo.

Una vez que Valet haya sido instalado, trata de hacer ping a cualquier dominio `*.test` desde tu terminal usando un comando como `ping foobar.test`. Si Valet ha sido instalado correctamente deberás ver una respuesta de ese dominio en la dirección `127.0.0.1`.

Valet iniciará automáticamente su daemon cada vez que el sistema inicie. Por lo tanto, si Valet se instaló adecuadamente, no hay necesidad de volver a ejecutar el comando `valet start` o `valet install`.

Utilizar otro dominio

Por defecto, Valet actuará como servidor de tus proyectos usando el TLD `.test`. Si lo prefieres, puedes cambiar este dominio por otro de tu elección utilizando el comando `valet tld dominio`.

Por ejemplo, si deseas utilizar el dominio `.app` en lugar de `.test`, ejecuta desde la terminal el comando `valet tld app` y Valet ahora funcionará como servidor de tus proyectos pero ahora con el dominio `*.app`.

Base de datos

Si necesitas de una base de datos, puedes instalar MySQL ejecutando el comando `brew install mysql@5.7` desde la terminal. Una vez que haya sido instalado, necesitarás iniciar el servicio de manera manual con el comando `brew services start mysql@5.7`. Podrás conectarte a tu base de datos en `127.0.0.1` utilizando el usuario `root` sin ninguna contraseña.

Versiones de PHP

Valet te permite cambiar entre versiones de PHP usando el comando `valet use php@version`. Valet instalará la versión de PHP especificada mediante Brew si aún no está instalada:

```
valet use php@7.2
```

php

```
valet use php
```

Nota

Valet sólo ejecuta una versión de PHP a la vez, incluso si tienes múltiples versiones de PHP instaladas.

Resetear tu instalación

Si estás teniendo problemas para que tu instalación de Valet funcione apropiadamente, ejecutar el comando `composer global update` seguido de `valet install` reseteará tu instalación y puede solucionar una variedad de problemas. En algunas ocasiones podría ser necesario un "reinicio forzado" ejecutando `valet uninstall --force` seguido de `valet install`.

Actualización

Puedes actualizar tu instalación de Valet ejecutando el comando `composer global update` desde la terminal. Después de actualizar, es una buena práctica ejecutar el comando `valet install` para que valet pueda hacer actualizaciones adicionales en sus archivos de configuración en caso de ser necesario.

Activar sitios

Una vez que Valet haya sido instalado, estarás listo para activar sitios. Valet proporciona dos comandos para ayudarte a activar sitios de Laravel: `park` y `link`.

El comando `park`

- Crea un nuevo directorio en tu Mac ejecutando algo como lo siguiente en la terminal `mkdir ~/sites`. Después, `cd ~/Sites` y ejecuta `valet park`. Este comando va a registrar tu directorio actual como una ruta en la que Valet deberá buscar los sitios.
- Después, crea un nuevo sitio de laravel dentro de este directorio: `laravel new blog`.
- Abre tu navegador y dirígete a `http://blog.test`.

Y eso es todo. Ahora, cada proyecto de Laravel que crees dentro de tu directorio `~/Sites` será visible desde el navegador utilizando la convención `http://folder-name.test`.

El comando `link`

El comando `link` también puede ser utilizado para enlazar sitios de Laravel. Este comando es útil si deseas configurar un solo sitio en un directorio y no todos los sitios dentro de él.

- Para utilizar este comando, deberás dirigirte a uno de tus proyectos desde la terminal y ejecutar `valet link app-name`. Valet creará un enlace simbólico en `~/.config/valet/Sites` el cuál

apuntará hacia tu directorio actual.

- Despues de ejecutar el comando `link`, podrás acceder al sitio desde tu navegador en `http://app-name.test`.

Para ver un listado de todos los directorios enlazados, ejecuta el comando `valet links`. Para destruir algún enlace simbólico deberás utilizar el comando `valet unlink app-name`.

TIP

Puedes utilizar `valet link` para configurar el mismo proyecto para multiples (sub)dominios.

Para agregar un subdominio o un dominio diferente para tu proyecto ejecuta `valet link subdomain.app-name`.

Asegurar sitios con TLS

Por defecto, Valet mostrará los sitios a través de HTTP plano. Sin embargo, si deseas que esté encriptado con TLS para ser utilizado con HTTP/2, el comando `secure` está disponible. Por ejemplo, si tu sitio está funcionando con Valet en el dominio `laravel.test`, podrás ejecutar el siguiente comando para asegurarlo:

```
valet secure laravel
```

php

Para quitar esta seguridad al sitio y revertir los cambios de nuevo hacia HTTP plano, deberás utilizar el comando `unsecure`. Al igual que el comando `secure`, este comando acepta el nombre del host al que se desea quitar la encriptación TLS.

```
valet unsecure laravel
```

php

Compartir sitios

Valet incluso tiene un comando para compartir tus sitios locales con el mundo, proporcionando una forma fácil de probar tus sitios en dispositivos móviles o compatirlo con miembros de tu equipo y clientes. Una vez que Valet está instalado no es necesario software adicional.

Compartir sitios mediante Ngrok

Para compartir un sitio, deberás dirigirte hacia el directorio del sitio desde la terminal y ejecutar el comando `valet share`. Una URL accesible de manera pública será copiada a tu portapapeles y estará lista para que la pegues directamente en tu navegador.

Para detener la ejecución de `share` en tu sitio, presiona `Control + C` para cancelar el proceso.

TIP

Puedes pasar parametros adicionales al comando share, como `valet share --region=eu`.

Para más información, consulta la documentación de ngrok [↗](#).

Compartir sitios en tu red local

Por defecto, Valet restringe el tráfico entrante a la interfaz `127.0.0.1`. De esta forma tu equipo de desarrollo no está expuesto a riesgos de seguridad en Internet.

Si deseas permitir que otros dispositivos en tu red local accedan a los sitios de Valet en tu equipo mediante la IP del computador (por ejemplo: `192.168.1.10/app-name.test`), necesitarás editar manualmente el archivo de configuración de Nginx apropiado para dicho sitio para remover la restricción en la directiva `listen` eliminando el prefijo `127.0.0.1:` en la directa para los puertos 80 y 443.

Si no has ejecutado `valet secure` en el proyecto, puedes abrir el acceso de la red para todos los sitios sin HTTPS ejecutando el archivo `/usr/local/etc/nginx/valet/valet.conf`. Sin embargo, si estás sirviendo el sitio del proyecto mediante HTTPS (has ejecutado `valet secure` para el sitio) entonces deberás editar el archivo `~/.config/valet/Nginx/app-name.test`.

Una vez que has actualizado la configuración de Nginx, ejecuta el comando `valet restart` para aplicar los cambios en la configuración.

VARIABLES DE ENTORNO ESPECÍFICAS DEL SITIO

Algunas aplicaciones que utilizan otros frameworks pueden depender de las variables de entorno del servidor, pero no proporcionan una manera para que esas variables sean configuradas dentro de tu proyecto. Valet te permite configurar variables de entorno específicas del sitio agregando un archivo `.valet-env.php` dentro de la raíz de tu proyecto. Estas variables se agregarán al arreglo global `$_SERVER`:

```
<?php
```

php

```
// Set $_SERVER['key'] to "value" for the foo.test site...
return [
    'foo' => [
        'key' => 'value',
    ],
];
// Set $_SERVER['key'] to "value" for all sites...
return [
    '*' => [
        'key' => 'value',
    ],
];
```

Drivers de Valet personalizados

Puedes escribir tu propio "driver" de Valet para utilizar aplicaciones de PHP que se estén ejecutando en otro framework o en un CMS que no sea soportado de manera nativa por Valet. Cuando se hace la instalación de Valet, es creado un directorio `~/.config/valet/Drivers` que contiene un archivo `SampleValetDriver.php`. Este archivo contiene la implementación de un driver de muestra para demostrar cómo escribir un driver personalizado. Escribir un driver solo requiere que implementes tres métodos: `serves`, `isStaticFile`, y `frontControllerPath`.

Los tres métodos reciben los valores de `$sitePath`, `$siteName`, y `$uri` como argumentos. La variable `$sitePath` es la ruta completa del sitio que será configurado en tu equipo, algo como `/Users/Lisa/Sites/my-project`. La variable `$siteName` representa la porción "host" / "site-name" del dominio { `my-project` }. La variable `$uri` es la petición URI entrante (`/foo/bar`).

Una vez que hayas terminado con tu driver de valet personalizado, se deberá colocar en el directorio `~/.config/valet/Drivers` usando la convención `FrameworkValetDriver.php` para nombrarlo. Por ejemplo, si estás escribiendo un driver personalizado de valet para WordPress, tu archivo deberá ser `WordPressValetDriver.php`.

Echemos un vistazo a la implementación de ejemplo en cada uno de los métodos del driver personalizado de Valet.

El método `serves`

El método `serves` deberá retornar `true` si tu driver debe encargarse de las peticiones entrantes. De otra manera, este método deberá retornar `false`. Por lo tanto, dentro de este método deberás intentar determinar si el `$sitePath` dado contiene un proyecto del tipo que deseas configurar.

Por ejemplo, vamos a pretender que estamos escribiendo un `WordPressValetDriver`. Nuestro método `serves` podría verse mas o menos como esto:

```
/** php
 * 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` deberá determinar si la petición entrante para un archivo es estático, como puede ser una imagen o una hoja de estilo. Si el archivo es estático, el método deberá retornar la ruta absoluta del archivo en disco. Si la petición entrante no es para un archivo estático, el metodo deberá retornar `false`:

```
/** php
 * 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;  
}
```

Nota

El método `isStaticFile` solo será llamado si el método `serves` retorna `true` para las peticiones entrantes y la URL es diferente a `/`.

El método `frontControllerPath`

El método `frontControllerPath` deberá retornar la ruta absoluta del "front controller" de tu aplicación, que usualmente es el archivo "index.php" o su 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';  
}
```

php

Drivers locales

Si deseas definir un driver de Valet personalizado para una aplicación sencilla, deberás crear un archivo `LocalValetDriver.php` en el directorio raíz de tu aplicación. El driver personalizado deberá extender de la clase base `ValetDriver` o extender del driver de alguna aplicación existente, como puede ser `LaravelValetDriver`.

```
class LocalValetDriver extends LaravelValetDriver  
{  
    /**  
     * Determine if the driver serves the request.  
    */
```

php

```

/*
 * @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
<code>valet forget</code>	Ejecuta este comando desde el directorio donde ejecutaste el comando <code>park</code> para eliminarlo de la lista de directorios configurados.
<code>valet log</code>	Ver una lista de logs escritos por servicios de Valet.
<code>valet paths</code>	Ver una lista de directorios configurados.
<code>valet restart</code>	Reiniciar el daemon de Valet.

Comando	Descripción
<code>valet start</code>	Iniciar el daemon de Valet.
<code>valet stop</code>	Detener el daemon de Valet.
<code>valet trust</code>	Agrega archivos sudoers para Brew y Valet para permitir que los comandos de Valet se ejecuten sin solicitar contraseñas.
<code>valet uninstall</code>	Desinstalar Valet: muestra instrucciones para la desinstalación manual; o pasa el parametro <code>--force</code> para eliminar Valet directamente.

Archivos y directorios de Valet

Puedes encontrar útil la siguiente información sobre directorios y archivos al momento de solucionar problemas en tu entorno de Valet:

Archivo / Ruta	Descripción
<code>~/.config/valet/</code>	Contiene toda la configuración de Valet. Es recomendable tener un respaldo de este directorio.
<code>~/.config/valet/dnsmasq.d/</code>	Contiene la configuración de DNSMasq.
<code>~/.config/valet/Drivers/</code>	Contiene drivers personalizados de Valet.

Archivo / Ruta	Descripción
<code>~/.config/valet/Extensions/</code>	Contiene extensiones y comandos personalizados de Valet.
<code>~/.config/valet/Nginx/</code>	Contiene toda las configuraciones de Nginx generadas por Valet. Estos archivos son compilados de nuevo al ejecutar los comandos <code>install</code> , <code>secure</code> y <code>tld</code> .
<code>~/.config/valet/Sites/</code>	Contiene todos los enlaces simbólicos de proyectos enlazados.
<code>~/.config/valet/config.json</code>	Archivo de configuración principal de Valet.

Archivo / Ruta	Descripción
<code>~/.config/valet/valet.sock</code>	Socket PHP-FPM usado por la configuración de Nginx de Valet. Esto sólo existirá si PHP se está ejecutando de forma apropiada.
<code>~/.config/valet/Log/fpm-php.www.log</code>	Registro de usuario para errores de PHP.
<code>~/.config/valet/Log/nginx-error.log</code>	Registro de usuario para errores de Nginx.
<code>/usr/local/var/log/php-fpm.log</code>	Registro de sistema para errores de PHP-FPM.
<code>/usr/local/var/log/nginx</code>	Contiene registros de acceso y error de Nginx.

Archivo / Ruta	Descripción
<code>/usr/local/etc/php/X.X/conf.d</code>	Contiene archivos <code>*.ini</code> para varias configuraciones de PHP.
<code>/usr/local/etc/php/X.X/php-fpm.d/valet-fpm.conf</code>	Archivo de configuración de PHP-FPM.
<code>~/.composer/vendor/laravel/valet/cli/stubs/secure.valet.conf</code>	Configuración por defecto de Nginx usada para construir certificados de sitios.

Despliegue

- Introducción
- Configuración del servidor
 - Nginx
- Optimización
 - Optimizar autoloader

- Optimizar configuración local
- Optimizar carga de rutas
- Deploy en forge

Introducción

Una vez que estés listo para hacer deploy de tu aplicación de Laravel a producción, deberías considerar algunos aspectos importantes para hacer que tu aplicación se ejecute de la forma más eficientemente posible. En este documento, vamos a cubrir muy buenos puntos para hacer que tu aplicación de Laravel sea desplegada correctamente.

Configuración del servidor

Nginx

Si estás haciendo deploy de tu aplicación hacia un servidor que está ejecutando Nginx, puedes utilizar el siguiente archivo de configuración como punto de inicio para configurar tu servidor web. Principalmente, este archivo tendrá que ser personalizado dependiendo de la configuración de tu servidor. Si deseas asistencia en la administración de tu servidor, considera utilizar 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_pass unix:/var/run/php/php7.2-fpm.sock;
    fastcgi_index index.php;
    fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
    include fastcgi_params;
}

location ~ /\.well-known.* {
    deny all;
}
}
```

Optimización

Optimizar autoloader

Al hacer deploy a producción, debes asegurarte de optimizar el autoloader de Composer, para que éste pueda localizar rápidamente el archivo apropiado para cargar una clase dada:

```
composer install --optimize-autoloader --no-dev
```

php

TIP

Adicionalmente, para optimizar el autoloader, deberás asegurarte de incluir siempre el archivo `composer.lock` al controlador de versiones de tu proyecto. Las dependencias de tu proyecto se instalarán más rápido cuando exista el archivo `composer.lock`.

Optimizar configuración local

Al hacer deploy de tu aplicación a producción, deberás asegurarte de ejecutar el comando de Artisan `config:cache` durante el proceso de deploy:

```
php artisan config:cache
```

php

Este comando combinará todos los archivos de configuración de Laravel en un solo archivo en caché, lo que reduce en gran medida la cantidad de consultas que el framework debe hacer al sistema de archivos cuando carga tus valores de configuración.

Nota

Si ejecutas el comando `config:cache` durante el proceso de despliegue, debes asegurarte de que sólo estás llamando a la función `env` desde dentro de tus archivos de configuración. Una vez que la configuración ha sido agregada, el archivo `.env` no será cargado y todas las llamadas a la función `env` retornarán `null`.

Optimizar carga de rutas

Si estás construyendo una aplicación muy grande que contenga muchas rutas, deberías asegurarte de ejecutar el comando `route:cache` de Artisan durante el proceso de deploy.

```
php artisan route:cache
```

php

Este comando reduce todas tus rutas registradas en una única llamada al método dentro del archivo en cache, mejorando el rendimiento de registro de rutas cuando se tienen cientos de ellas.

Nota

Ya que esta característica utiliza la serialización de PHP, sólo se pueden almacenar en cache las rutas para las aplicaciones que estén basadas exclusivamente en controladores. PHP no es capaz de serealizar Closures.

Deploy en forge

Si no estás del todo listo para administrar la configuración de tu servidor o si no te sientes cómodo configurando los diferentes servicios necesarios para ejecutar aplicaciones robustas de Laravel, [Laravel Forge](#) es una excelente alternativa.

Laravel Forge puede crear servidores en varios proveedores de infraestructura como pueden ser DigitalOcean, Linode, AWS y más. Adicionalmente, Forge instala y administra todas las herramientas

necesarias para construir aplicaciones robustas de Laravel como Nginx, MySQL, Redis, Memcached, Beanstalk y más.

Ciclo de vida de la solicitud

- [Introducción](#)
- [Resumen del ciclo de vida](#)
- [Enfoque en los proveedores de servicios](#)

Introducción

Al usar cualquier herramienta en el "mundo real", te sientes más cómodo si entiendes como esa herramienta funciona. El desarrollo de aplicaciones no es diferente. Cuando entiendes cómo tus herramientas de desarrollo funcionan, te sientes más cómodo y seguro usándolas.

El objetivo de este documento es darte un buen resumen sobre cómo el framework Laravel funciona. Al conocer el framework mejor, todo lo demás se siente menos "mágico" y te sentirás más cómodo construyendo tus aplicaciones. Si no entiendes todos los términos de una sola vez, no te desesperes! Sólo trata de obtener una comprensión básica de lo que está sucediendo y tus conocimientos crecerán a medida que exploras otras secciones de la documentación.

Resumen del ciclo de vida

Lo primero

El punto de entrada para todas las solicitudes a una aplicación de Laravel es el archivo `public/index.php`. Todas las solicitudes son dirigidas a este archivo por la configuración de tu servidor web (Apache / Nginx). El archivo `index.php` no contiene mucho código. En su lugar, es un punto de partida para cargar el resto del framework.

El archivo `index.php` carga la definición de autocarga generada por Composer y luego retorna una instancia de la aplicación de Laravel desde el script `bootstrap/app.php`. La primera acción tomada por Laravel es crear una instancia de la aplicación / contenedor de servicios.

Kernel de HTTP / Consola

Luego, la solicitud entrante es enviada ya sea al kernel HTTP o al kernel de la consola, dependiendo del tipo de solicitud que está entrando en la aplicación. Estos dos kernels funcionan como la ubicación principal a través de la cual todas las solicitudes pasan. Por ahora, vamos a enfocarnos sólo en el kernel HTTP, que está ubicado en `app/Http/Kernel.php`.

El kernel HTTP extiende de la clase `Illuminate\Foundation\Http\Kernel`, que define un arreglo de `bootstrappers` que se ejecutarán antes de que la solicitud sea ejecutada. Estos maquetadores configuran el manejo de errores, registros, detectan en el entorno de la aplicación y realizan otras tareas que necesitan ser ejecutadas antes de que la solicitud sea manejada.

El kernel HTTP también define una lista de middleware HTTP que todas las solicitudes deben pasar antes de ser manejadas por la aplicación. Estos middleware manejan la lectura y escritura de la sesión HTTP, determinando si la aplicación está en modo de mantenimiento, verificando el token CSRF y más.

La firma del método para el método `handle` del kernel HTTP es bastante simple: recibe un `Request` y retorna un `Response`. Piensa en el Kernel como una caja negra grande que representa toda tu aplicación. Aliméntala con solicitudes HTTP y retornará respuestas HTTP.

Proveedores de servicios

Una de las acciones de maquetado más importantes del Kernel es cargar los proveedores de servicios de tu aplicación. Todos los proveedores de servicios de la aplicación son configurados en el arreglo `providers` del archivo de configuración `config/app.php`. Primero, el método `register` será llamado en todos los proveedores, luego, una vez que todos los proveedores sean registrados, el método `boot` será llamado.

Los proveedores de servicios son responsables de estructurar todos los distintos componentes del framework, como la base de datos, colas, validaciones y componentes de rutas. Dado que estructuran y configuran cada característica ofrecida por el framework, los proveedores de servicios son el aspecto más importante de todo el proceso de estructuración de Laravel.

Despachar la solicitud

Una vez que la aplicación ha sido estructurada y todos los proveedores de servicios han sido registrados, la solicitud o `Request` será manejada por el enrutador para su despacho. El enrutador enviará la solicitud a una ruta o controlador, así como ejecutará cualquier middleware específico de ruta.

Enfoque en los proveedores de servicios

Los proveedores de servicios son realmente la clave para estructurar una aplicación de Laravel. La instancia de la aplicación es creada, los proveedores de servicios son registrados y la solicitud es entregada a la aplicación ya estructurada. ¡Es realmente así de simple!

Tener un firme conocimiento sobre cómo una aplicación de Laravel es construida y estructurada mediante proveedores de servicios es muy útil. Los proveedores de servicios por defecto de tu aplicación están almacenados en el directorio `app/Providers`.

Por defecto, `AppServiceProvider` está casi vacío. Este proveedor es un buen lugar para agregar tu propia estructura de componentes y enlaces al contenedor de servicios de tu aplicación. Para aplicaciones grandes, puedes desear crear múltiples proveedores de servicios, cada uno que estuture componentes de una manera más granular.

Contenedor de servicios

- [Introducción](#)
- [Enlaces](#)
 - [Fundamentos de los enlaces](#)
 - [Enlazando interfaces a implementaciones](#)
 - [Enlaces contextuales](#)
 - [Etiquetado](#)
 - [Extendiendo enlaces](#)
- [Resolviendo](#)
 - [Método make](#)

- Inyección automática
- Eventos del contenedor
- PSR-11

Introducción

El contenedor de servicios de Laravel es una herramienta poderosa para administrar dependencias de clases y realizar inyección de dependencias. La inyección de dependencias es una frase bonita para básicamente decir: las dependencias de clases son "inyectadas" en la clase mediante el constructor o, en algunos casos, métodos "setter".

Echemos un vistazo a un ejemplo sencillo:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use App\Repositories\UserRepository;  
use App\User;  
  
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;  
    }  
}
```

php

```
/**
 * 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 retornar usuarios desde una fuente de datos. Así que, **inyectaremos** un servicio que es capaz de retornar los usuarios. En este contexto, nuestro `UserRepository` probablemente usa Eloquent para retornar la información de los usuarios desde la base de datos. Sin embargo, dado que el repositorio es inyectado, somos capaces de cambiarlo fácilmente con otra implementación. También somos capaces de "simular" o crear una implementación de ejemplo de `UserRepository` al probar nuestra aplicación.

Un conocimiento profundo del contenedor de servicios de Laravel es esencial para construir aplicaciones grandes y poderosas así como también contribuir al núcleo de Laravel.

Enlaces

Fundamentos de los enlaces

La mayoría de los enlaces de tu contenedor de servicios serán registrados dentro de proveedores de servicios, así que la mayoría de estos ejemplos muestra el uso del contenedor en ese contexto.

TIP

No hay necesidad de enlazar clases al contenedor si no dependen de ninguna interfaz. El contenedor no necesita ser instruido en cómo construir esos objetos, dado que puede resolver dichos objetos automáticamente usando reflejos.

Enlaces sencillos

Dentro de un proveedor de servicios, siempre tienes acceso al contenedor mediante la propiedad `$this->app`. Podemos registrar un enlace usando el método `bind`, pasando el nombre de la clase o interfaz que deseamos registrar junto con una `Closure` que retorna una instancia de la clase:

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

php

Observa que recibimos el contenedor como argumento. Podemos entonces usar el contenedor para resolver sub-dependencias del objeto que estamos construyendo.

Enlazando un singleton

El método `singleton` enlaza una clase o interfaz al contenedor que debería ser resuelto una sola vez. Una vez que el enlace de un singleton es resuelto, la misma instancia de objeto será retornada en llamadas siguientes al contenedor:

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

php

Enlazando instancias

También puedes enlazar una instancia de objeto existente al contenedor usando el método `instance`. La instancia dada siempre será retornada en llamadas siguientes al contenedor:

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

php

Enlazando valores primitivos

Algunas veces tendrás una clase que recibe algunas clases inyectadas, pero que también necesita un valor primitivo inyectado, como un entero. Puedes fácilmente usar enlaces contextuales para inyectar cualquier valor que tu clase pueda necesitar:

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

php

Enlazando interfaces a implementaciones

Una característica muy poderosa del contenedor de servicios es su habilidad para enlazar una interfaz a una implementación dada. Por ejemplo, vamos a suponer que tenemos una interfaz `EventPusher` y una implementación `RedisEventPusher`. Una vez que hemos programado nuestra implementación `RedisEventPusher` de esta interfaz, podemos registrarla con el contenedor de servicios de la siguiente manera:

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

php

Esta sentencia le dice al contenedor que debe injectar `RedisEventPusher` cuando una clase necesita una implementación de `EventPusher`. Ahora podemos determinar el tipo de la interfaz `EventPusher` en un constructor o cualquier otra ubicación donde las dependencias son inyectadas por el contenedor de servicios:

```
use App\Contracts\EventPusher;

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

php

Enlaces contextuales

Algunas veces tendrás dos clases que usan la misma interfaz, pero quieres injectar diferentes implementaciones en cada clase. Por ejemplo, dos controladores pueden depender de diferentes implementaciones del contrato `Illuminate\Contracts\Filesystem\Filesystem`. Laravel proporciona una simple y fluida interfaz para definir este comportamiento:

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

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

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

Etiquetado

Ocasionalmente, puedes necesitar resolver todo de una determinada "categoría" de enlaces. Por ejemplo, puede que estés construyendo un agregador de reportes que recibe un arreglo de diferentes implementaciones de la interfaz `Report`. Luego de registrar las implementaciones de `Report`, puedes asignarles una etiqueta usando el método `tag`:

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

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

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

Una vez que los servicios han sido etiquetados, puedes resolverlos fácilmente mediante el método

`tagged` :

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

php

Extendiendo enlaces

El método `extend` te permite modificar servicios resueltos. Por ejemplo, cuando un servicio es resuelto, puedes ejecutar código adicional para decorar o configurar el servicio. El método `extend` acepta un Closure, que debe retornar el servicio modificado como único argumento. La Closure recibe el servicio siendo resuelto y la instancia del contenedor:

```
$this->app->extend(Service::class, function ($service, $app) {  
    return new DecoratedService($service);  
});
```

php

Resolviendo

Método `make`

Puedes usar el método `make` para resolver una instancia de clase fuera del contenedor. El método `make` acepta el nombre de la clase o interfaz que deseas resolver:

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

php

Si estás en una ubicación de tu código que no tiene acceso a la variable `$app`, puedes usar el helper global `resolve` :

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

php

Si algunas de las dependencias de tu clase no son resueltas mediante el contenedor, puedes inyectarlas pasándolas como un arreglo asociativo al método `makewith` :

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

php

Inyección automática

Alternativamente, y de forma importante, puedes "determinar el tipo" de la dependencia en el constructor de una clase que es resuelta por el contenedor, incluyendo [controladores](#), [listeners de eventos](#), [colas](#), [middleware](#) y más. En la práctica, así es como la mayoría de tus objetos deben ser resueltos por el contenedor.

Por ejemplo, puedes determinar el tipo de un repositorio definido por tu aplicación en el constructor de un controlador. El repositorio será automáticamente resuelto e injectado 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  
     */
```

php

```
*/  
public function show($id)  
{  
    //  
}  
}
```

Eventos del contenedor

El contenedor de servicios ejecuta un evento cada vez que resuelve un objeto. Puedes escuchar a este evento usando 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 puedes ver, el objeto siendo resuelto será pasado a la función de retorno, permitiéndote establecer cualquier propiedad adicional en el objeto antes de que sea entregado a su consumidor.

PSR-11

El contenedor de servicios de Laravel implementa la interfaz [PSR-11](#). Por lo tanto, puedes determinar el tipo de la interfaz de contenedor PSR-11 para obtener una instancia del contenedor de Laravel:

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

Una excepción es mostrada si el identificador dado no puede ser resuelto. La excepción será una instancia de `Psr\Container\NotFoundExceptionInterface` si el identificador nunca fue enlazado.

Si el identificador fue enlazado pero ha sido incapaz de resolver, una instancia de `Psr\Container\ContainerExceptionInterface` será mostrada.

Proveedores de Servicios

- [Introducción](#)
- [Escribiendo proveedores de servicios](#)
 - [Método register](#)
 - [Método boot](#)
- [Registrando proveedores](#)
- [Proveedores diferidos](#)

Introducción

Los proveedores de servicios son la parte central de la maquetación de una aplicación Laravel. Tu propia aplicación, así como todos los servicios principales de Laravel son maquetados usando proveedores de servicios.

Pero, ¿qué queremos decir por "maquetación"? En general, nos referimos a **registrar** cosas, incluyendo registrar enlaces de contenedores de servicios, listeners de eventos, middleware e incluso rutas. Los proveedores de servicios son el lugar principal para configurar tu aplicación.

Si abres el archivo `config/app.php` incluido con Laravel, verás un arreglo `providers`. Estos son todos los proveedores de servicio que serán cargados para tu aplicación. Observa que muchos de éstos son proveedores "diferidos", lo que significa que no serán cargados en cada solicitud, sino sólo cuando los servicios que proporcionan sean necesarios.

En este resumen aprenderás a escribir tus propios proveedores de servicio y registrarlos en tu aplicación de Laravel.

Escribiendo proveedores de servicios

Todos los proveedores de servicios extienden de la clase `Illuminate\Support\ServiceProvider`.

La mayoría de los proveedores de servicio contienen un método `register` y `boot`. Dentro del método `register`, debes **enlazar cosas sólo al contenedor de servicios**. Nunca debes tratar de registrar ningún listener de eventos, rutas o cualquier otra pieza de funcionalidad dentro del método `register`.

La línea de comandos Artisan puede generar un nuevo proveedor mediante el comando

```
make:provider :
```

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

php

Método register

Como mencionamos anteriormente, dentro del método `register`, debes sólo enlazar cosas al contenedor de servicio. Nunca debes intentar registrar ningún listener de eventos, rutas o cualquier otra pieza de funcionalidad dentro del método `register`. De lo contrario, puedes accidentalmente usar un servicio que es proporcionado por un proveedor de servicio que no aún no se ha cargado.

Vamos a echar un vistazo a un proveedor de servicio básico. Dentro de cualquiera de los métodos de tu proveedor de servicios, siempre tienes acceso a la propiedad `$app` que proporciona acceso al contenedor de servicios:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\ServiceProvider;  
use Riak\Connection;  
  
class RiakServiceProvider extends ServiceProvider  
{  
    /**  
     * Register any application services.  
     *  
     * @return void  
     */
```

php

```
public function register()
{
    $this->app->singleton(Connection::class, function ($app) {
        return new Connection(config('riak'));
    });
}
```

Este proveedor de servicios sólo define un método `register` y usa dicho método para definir una implementación de `Riak\Connection` en el contenedor de servicios. Si no entiendes cómo el contenedor de servicios funciona, revisa su documentación.

Propiedades `bindings` y `singletons`

Si tu proveedor de servicios registra muchos bindings simples, puedes querer usar las propiedades `bindings` y `singletons` en lugar de manualmente registrar cada binding de contenedor. Cuando el proveedor de servicios es cargado por el framework, automáticamente comprobará dichas propiedades y registrará sus bindings:

```
<?php

namespace App\Providers;

use App\Contracts\DowntimeNotifier;
use App\Contracts\ServerProvider;
use App\Services\DigitalOceanServiceProvider;
use App\Services\PingdomDowntimeNotifier;
use App\Services\ServerToolsProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * All of the container bindings that should be registered.
     *
     * @var array
     */
    public $bindings = [
        ServerProvider::class => DigitalOceanServiceProvider::class,
    ];
}
```

```
/**
 * All of the container singletons that should be registered.
 *
 * @var array
 */
public $singletons = [
    DowntimeNotifier::class => PingdomDowntimeNotifier::class,
    ServerToolsProvider::class => ServerToolsProvider::class,
];
}
```

Método boot

Entonces, ¿qué sucede si necesitamos registrar un view composer dentro de nuestro proveedor de servicios? Esto debería ser hecho dentro del método `boot`. **Este método es llamado luego de que todos los demás proveedores de servicio sean registrados**, lo que quiere decir que tienes acceso a todos los demás proveedores de servicio que han sido registrados por el framework:

```
<?php
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 () {
            //
        });
    }
}
```

Inyección de dependencias en el método boot

Puedes escribir manualmente las dependencias para el método `boot` de tu proveedor de servicios. El contenedor de servicios inyectará automáticamente cualquier dependencia que necesites:

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

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

Registrando proveedores

Todos los proveedores de servicios son registrados en el archivo de configuración `config/app.php`. Este archivo contiene un arreglo `providers` donde puedes listar los nombres de clase de tus proveedores de servicios. Por defecto, una serie de proveedores de servicios principales de Laravel son listados en este arreglo. Estos proveedores maquetan los componentes principales de Laravel, como mailer, queue, cache entre otros.

Para registrar tu proveedor, agregalo al arreglo:

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

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

Proveedores diferidos

Si tu proveedor **sólo** está registrando enlaces en el contenedor de servicios, puedes elegir diferir su registro hasta que uno de los enlaces registrados sea necesario. Diferir la carga de dicho proveedor mejorará el rendimiento de tu aplicación, ya que no es cargado desde el sistema de archivos en cada solicitud.

Laravel compila y almacena una lista de todos los servicios suministrados por proveedores de servicios diferidos, junto con el nombre de clase de su proveedor de servicio. Luego, sólo cuando intentas resolver

uno de estos servicios Laravel carga el proveedor de servicio.

Para diferir la carga de un proveedor, implementa la interfaz

`\Illuminate\Contracts\Support\DeferrableProvider` y define un método `provides`. El método `provides` debe retornar los enlaces del contenedor de servicio registrados por el proveedor:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Contracts\Support\DeferrableProvider;  
use Illuminate\Support\ServiceProvider;  
use Riak\Connection;  
  
class RiakServiceProvider extends ServiceProvider implements DeferrableProvider  
{  
    /**  
     * 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
- Cuándo usar facades
 - Facades vs. inyección de dependencias
 - Facades vs. funciones helper
- Cómo funcionan las facades
- Facades en tiempo real
- Referencia de clases de facades

Introducción

Las Facades proveen una interfaz "estática" a las clases disponibles en el contenedor de servicios de la aplicación. Laravel viene con numerosas facades, las cuales brindan acceso a casi todas las características de Laravel. Las facades de Laravel sirven como "proxies estáticas" a las clases subyacentes en el contenedor de servicios, brindando el beneficio de una sintaxis tersa y expresiva, manteniendo mayor verificabilidad y flexibilidad que los métodos estáticos tradicionales.

Todas las facades de Laravel se definen en el namespace `Illuminate\Support\Facades`. Entonces, podemos fácilmente acceder a una facade de esta forma:

```
use Illuminate\Support\Facades\Cache;  
  
Route::get('/cache', function () {  
    return Cache::get('key');  
});
```

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

Cuándo usar facades

Las Facades tienen múltiples beneficios. Brindan una sintaxis tersa y memorizable que permite utilizar las características de Laravel sin tener que recordar nombres de clase largos que deben ser inyectados o configurados manualmente. Además, debido a su uso único de los métodos dinámicos PHP, son fáciles de probar.

Sin embargo, deben guardarse ciertas precauciones al hacer uso de facades. El peligro principal de las facades es la corrupción de alcance de clases. Como las facades son tan fáciles de usar y no requieren inyección, puede resultar fácil dejar que tus clases sigan creciendo y usar muchas facades en una sola clase. Usando inyección de dependencias, este potencial es mitigado por la retroalimentación visual que un constructor grande te da cuando tu clase está creciendo demasiado. Entonces, al usar facades, pon especial atención al tamaño de tu clase para que su alcance de responsabilidades permanezca limitado.

TIP

Cuando se construye un paquete de terceros que interactúa con Laravel, es mejor inyectar contratos de Laravel en vez de usar facades. Como los paquetes son construidos fuera de Laravel, no tendrás acceso a las funciones (helpers) de testing para facades de Laravel.

Facades vs. inyección de dependencias

Uno de los principales beneficios de la inyección de dependencias es la habilidad de intercambiar implementaciones de la clase inyectada. Esto es útil durante las pruebas debido a que puedes inyectar un mock o un stub y comprobar que esos métodos son llamados en el stub.

Típicamente, no sería posible imitar (mock) o sustituir (stub) un método de clase verdaderamente estático. Sin embargo, como las facades utilizan métodos dinámicos para hacer proxy de llamadas de método a objetos resueltos desde el contenedor de servicios, podemos de hecho probar las facades exactamente cómo 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');  
});
```

php

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

php

```
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 una variedad de funciones "helper", las cuales pueden realizar tareas comunes como generar vistas, disparar eventos, despachar trabajos, o mandar respuestas HTTP. Muchas de estas funciones helper realizan la misma función que su facade correspondiente. Por ejemplo, éstas llamadas facade y helper son equivalentes:

php

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

return view('profile');
```

No hay diferencia práctica en lo absoluto entre facades y funciones helper. Al usar funciones helper, aún se pueden probar como se probaría la facade correspondiente. Por ejemplo, dada la siguiente ruta:

php

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

Bajo la superficie, el helper `cache` llamará al método `get` en la clase subyacente a la facade `Cache`. Entonces, aún cuando estamos usando la función helper, podemos escribir la siguiente prueba para verificar que el método fue 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');
}
```

php

Cómo funcionan las facades

En una aplicación Laravel, una facade es una clase que provee acceso a un objeto desde el contenedor. La maquinaria que hace este trabajo está en la clase `Facade`. Las facades de Laravel y cualquier facade personalizada que crees, extenderá la clase base `Illuminate\Support\Facades\Facade`.

La clase base `Facade` hace uso del método mágico `__callStatic()` para aplazar las llamadas desde tu facade a un objeto resuelto desde el contenedor. En el ejemplo siguiente, se realiza una llamada al sistema de caché de Laravel. Al mirar este código, se puede 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.
     *
```

php

```

    * @param int $id
    * @return Response
    */
    public function showProfile($id)
    {
        $user = Cache::get('user:'.$id);

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

```

Nótese que cerca del inicio del archivo estamos "importando" la facade `Cache`. Esta facade sirve como proxy para acceder a la implementación subyacente de la interfaz

`Illuminate\Contracts\Cache\Factory`. Cualquier llamada que hagamos usando la facade será pasada a la instancia subyacente del servicio de caché de Laravel.

Si observamos la clase `Illuminate\Support\Facades\Cache` verás que no hay método estático

`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` extiende la clase `Facade` y define el método

`getFacadeAccessor()`. El trabajo de este método es devolver el nombre de un enlace de contenedor de servicios. Cuando un usuario referencia cualquier método estático en la facade `Cache`, Laravel resuelve el enlace `cache` desde el `contenedor de servicios` y ejecuta el método solicitado (en este caso, `get`) contra ese objeto.

Facades en tiempo real

Usando facades en tiempo real, puedes tratar cualquier clase en tu aplicación como si fuera una facade. Para ilustrar cómo esto puede ser utilizado, examinemos una alternativa. Por ejemplo, asumamos que

nuestro modelo `Podcast` tiene un método `publish`. Sin embargo, para publicar el podcast, necesitamos inyectar una instancia `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);  
    }  
}
```

php

Inyectar una implementación de publisher dentro del método nos permite probar fácilmente el método aislado porque podemos imitar (mock) el publisher inyectado. Sin embargo, requiere que pasemos una instancia publisher cada vez que llamamos al método `publish`. Usando facades en tiempo real, podemos mantener la misma verificabilidad sin que se requiera pasar explícitamente una instancia `Publisher`. Para generar una facade en tiempo real, se añade el prefijo `Facades` al namespace de la clase importada:

```
<?php  
  
namespace App;  
  
use Facades\App\Contracts\Publisher;  
use Illuminate\Database\Eloquent\Model;
```

php

```

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

        Publisher::publish($this);
    }
}

```

Cuando la facade en tiempo real es utilizada, la implementación publisher será resuelta en el contenedor de servicios usando la porción de la interfaz o nombre de clase que aparece después del prefijo

[Facades](#). Al probar, podemos usar las funciones helpers de testing para facades integradas en Laravel para imitar (mock) esta llamada de método:

```

<?php
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 de facades

A continuación encontrarás cada facade y su clase subyacente. Esta es una herramienta útil para explorar rápidamente dentro de la documentación API para cualquier raíz de facade dada. La llave `service container binding` también ha sido incluida donde aplica.

Facade	Class	Service Container Binding
App	Illuminate\Foundation\Application	app
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

Facade	Class	Service Container Binding
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	
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\LogManager	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

Facade	Class	Service Container Binding
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	
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	

Facade	Class	Service Container Binding
View	Illuminate\View\Factory	<code>view</code>
View (Instance)	Illuminate\View\View	

Contratos

- Introducción
 - Contratos vs. facades
- Cuando usar contratos
 - Bajo acoplamiento
 - Simplicidad
- Cómo usar contratos
- Referencia de contratos

Introducción

Los Contratos de Laravel son un conjunto de interfaces que definen los servicios principales proporcionados por el framework. Por ejemplo, un contrato [Illuminate\Contracts\Queue\Queue](#) define los métodos necesarios para las colas de trabajo, mientras que el contrato [Illuminate\Contracts\Mail\Mailer](#) define los métodos necesarios para el envío de correos electrónicos.

Cada contrato tiene una implementación correspondiente provista por el framework. Por ejemplo, laravel proporciona una implementación de cola con una variedad de conductores (drivers), y una implementación de envío de correo electrónico que funciona con [SwiftMailer](#).

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

Contratos vs. facades

Los facades de Laravel y las funciones de ayuda (helpers) proporcionan una forma sencilla de utilizar los servicios de Laravel sin necesidad de determinar el tipo y resolver contratos fuera del contenedor de servicios. En la mayoría de los casos, cada facade tiene un contrato equivalente.

A diferencia de las facades, que no necesitan que las requieras en el constructor de su clase, los contratos te permiten definir dependencias explícitas para tus clases. Algunos desarrolladores prefieren definir explícitamente sus dependencias de esta manera y, por lo tanto, prefieren usar contratos, mientras que otros desarrolladores disfrutan de la conveniencia de las facades.

TIP

La mayoría de las aplicaciones funcionarán bien sin importar si prefieres facades o contratos. Sin embargo, si estás construyendo un paquete, debes considerar seriamente el uso de contratos, ya que será más fáciles de probar en un contexto paquete.

Cuando usar contratos

Como se discutió en otro lugar, gran parte de la decisión de usar contratos o facades se reducirá a los gustos personales y los gustos de su equipo de desarrollo. Tanto los contratos como las facades se pueden utilizar para crear aplicaciones Laravel robustas y bien probadas. Mientras mantengas enfocadas las responsabilidades de tu clase, notarás muy pocas diferencias prácticas entre el uso de contratos y facades.

Sin embargo, todavía puede tener varias preguntas con respecto a los contratos. Por ejemplo, ¿por qué usar interfaces? ¿No es más complicado usar interfaces? Detallaremos las razones para utilizar interfaces en los siguientes encabezados: bajo acoplamiento y simplicidad.

Bajo acoplamiento

Primero, revisemos algunos códigos que están estrechamente acoplado a una implementación de caché. Considera 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 estrechamente acoplado a una implementación de caché determinada. Está estrechamente acoplado porque dependemos de una clase de caché concreta de un proveedor de paquetes. Si la API de ese paquete cambia, nuestro código también debe cambiar.

Del mismo modo, si queremos reemplazar nuestra tecnología de caché subyacente (Memcached) con otra tecnología (Redis), nuevamente tendremos que modificar nuestro repositorio. Nuestro repositorio no

debe tener tanto conocimiento sobre quién les proporciona los datos o cómo los proporcionan.

En lugar de este enfoque, podemos mejorar nuestro código dependiendo de una interfaz simple e independiente del proveedor:

```
<?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;  
    }  
}
```

php

Ahora el código no está acoplado a ningún proveedor específico, ni siquiera a Laravel. Dado que el paquete de contratos no contiene implementación ni dependencias, puede escribir fácilmente una implementación alternativa de cualquier contrato dado, lo que le permite reemplazar su implementación de caché sin modificar ninguno de los códigos que consumen caché.

Simplicidad

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

Además, cuando dependes de interfaces simples, tu código es más fácil de entender y mantener. En lugar de rastrear qué métodos están disponibles dentro de una clase grande y complicada, puedes hacer referencia a una interfaz sencilla y limpia.

Cómo usar contratos

Entonces, ¿Cómo se obtiene una implementación de un contrato? En realidad es bastante simple.

Muchos tipos de clases en Laravel se resuelven a través del [contenedor de servicio](#), incluyendo controladores, los escuchadores de eventos, middleware, trabajos de cola e incluso una Closure de ruta. Por lo tanto, para obtener una implementación de un contrato, puede simplemente "declarar el tipo" de la interfaz en el constructor de la clase que se está resolviendo.

Por ejemplo, veamos este listener de eventos:

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderWasPlaced;  
use App\User;  
use Illuminate\Contracts\Redis\Factory;  
  
class CacheOrderInformation  
{  
    /**  
     * The Redis factory implementation.  
     */  
    protected $redis;  
  
    /**  
     * Create a new event handler instance.  
     *  
     * @param Factory $redis  
     * @return void  
     */  
    public function __construct(Factory $redis)  
    {  
        $this->redis = $redis;  
    }  
}
```

php

```

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

```

Cuando se resuelve el escuchador de evento, el contenedor de servicios leerá las declaraciones de tipo en el constructor de la clase e injectará el valor apropiado. Para obtener más información sobre cómo registrar cosas en el contenedor de servicios, consulte [su documentación](#).

Referencia de contratos

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

Contrato	Referencias de la Facade
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	

Contrato	Referencias de la Facade
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
Illuminate\Contracts\Cache\Lock	
Illuminate\Contracts\Cache\LockProvider	
Illuminate\Contracts\Cache\Repository	Cache::driver()
Illuminate\Contracts\Cache\Store	
Illuminate\Contracts\Config\Repository	Config
Illuminate\Contracts\Console\Application	
Illuminate\Contracts\Console\Kernel	Artisan
Illuminate\Contracts\Container\Container	App
Illuminate\Contracts\Cookie\Factory	Cookie
Illuminate\Contracts\Cookie\QueueingFactory	Cookie::queue()
Illuminate\Contracts\Database\ModelIdentifier	

Contrato	Referencias de la Facade
Illuminate\Contracts\Debug\ExceptionHandler	
Illuminate\Contracts\Encryption\Encrypter	Crypt
Illuminate\Contracts\Events\Dispatcher	Event
Illuminate\Contracts\Filesystem\Cloud	Storage::cloud()
Illuminate\Contracts\Filesystem\Factory	Storage
Illuminate\Contracts\Filesystem\Filesystem	Storage::disk()
Illuminate\Contracts\Foundation\Application	App
Illuminate\Contracts\Hashing\Hasher	Hash
Illuminate\Contracts\Http\Kernel	
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

Contrato	Referencias de la Facade
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
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	

Contrato	Referencias de la Facade
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	
Illuminate\Contracts\View\Factory	View
Illuminate\Contracts\View\View	View::make()

Rutas

- Rutas básicas
 - Redireccionar rutas
 - Las rutas de vistas
- Los parámetros de rutas
 - Los parámetros requeridos
 - Los parámetros opcionales
 - Las restricciones de expresiones regulares

- Las rutas nombradas
- Los grupos de ruta
 - Los middleware
 - Los espacios de nombres
 - Enrutamiento de subdominios
 - Los prefijos de ruta
 - Los prefijos por nombre de ruta
- Enlazamiento de modelo de ruta (route model binding)
 - Enlazamiento implícito
 - Enlazamiento explícito
- Rutas Fallback
- Límite de rango
- La suplantación del método del formulario
- Accediendo la ruta actual

Rutas básicas

Las rutas de Laravel más básicas aceptan una URI y una `Closure`, proporcionando un método muy fácil y expresivo de definición de rutas:

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

Los archivos de ruta predeterminados

Todas las rutas de Laravel están definidas en tus archivos de ruta, los cuales están localizados en el directorio `routes`. Estos archivos son cargados automáticamente por el framework. El archivo `routes\web.php` define rutas que son para tu interface web. Estas rutas son asignadas al grupo de middleware `web`, el cual proporciona características como estado de sesión y protección CSRF. Las rutas en `routes\api.php` son independientes de estado y son asignadas al grupo de middleware `api`.

Para las principales aplicaciones, empezarás definiendo rutas en tu archivo `routes/web.php`. Las rutas definidas en `routes/web.php` pueden ser accedidas colocando la URL de la ruta definida en tu

navegador. Por ejemplo, puede acceder a la siguiente ruta al navegar a `http://your-app.dev/user` en tu navegador:

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

php

Las rutas definidas en el archivo `routes/api.php` son agrupadas dentro de un grupo de ruta por el `RouteServiceProvider`. Dentro de este grupo, el prefijo de URI `/api` es aplicado automáticamente de modo que no es necesario aplicarlo manualmente en todas las rutas en el archivo. Puedes modificar el prefijo y otras opciones de grupos de ruta al modificar tu clase `RouteServiceProvider`.

Métodos disponibles del enrutador

El enrutador permite que registres rutas que responden 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);
```

php

Algunas veces puede que necesites registrar una ruta que responda a verbos HTTP múltiples. Puedes hacerlo usando el método `match`. También, puedes incluso registrar una ruta que responda a todos los verbos HTTP usando el método `any`:

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

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

php

Protección CSRF

Cualquiera de los formularios HTML que apunten a rutas `POST`, `PUT`, or `DELETE` que sean definidas en el archivo de rutas `web` deberían incluir un campo de token CSRF. De otra manera, la solicitud será

rechazada. Puedes leer más sobre protección CSRF en la documentación de CSRF:

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

php

Redireccionar rutas

Si estás definiendo una ruta que redirecciona a otra URI, puedes usar el método `Route::redirect`.

Este método proporciona una forma abreviada conveniente de modo que no tengas que definir una ruta completa o de controlador para ejecutar una redirección básica:

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

php

Por defecto, `Route::redirect` retorna un código de estado `302`. Puedes personalizar el código de estado usando el tercer parámetro opcional:

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

php

Puedes usar el método `Route::permanentRedirect` para retornar un código de estado `301`:

```
Route::permanentRedirect('/here', '/there');
```

php

Rutas de vista

Si tu ruta necesita solamente devolver una vista, puedes usar el método `Route::view`. Igual que el método `redirect`, este método proporciona una forma abreviada básica de modo que no tengas que definir una ruta completa o de controlador. El método `view` acepta una URI como su primer argumento y un nombre de vista como su segundo argumento. Además, puedes proporcionar una arreglo de datos para pasar a la vista como un tercer argumento opcional:

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

php

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

Parámetros de ruta

Parámetros requeridos

Con frecuencia necesitarás capturar segmentos de la URI dentro de tu ruta. Por ejemplo, puedes necesitar capturar un ID de usuario de la URL. Puedes hacer eso al definir los parámetros de ruta:

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

php

Puedes definir tantos parámetros de ruta como requieras para tu ruta:

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

php

Los parámetros de ruta siempre son encerrados dentro de `{}`, deberían consistir de caracteres alfabéticos y no pueden contener un carácter `-`. En lugar de usar el carácter `-`, use el guión bajo (`_`). Los parámetros de ruta son inyectados dentro de las funciones de retorno de ruta / controlador en base a su orden - los nombres de los argumentos de la función de retorno / controlador no importan.

Parámetros opcionales

Ocasionalmente puede que necesites especificar un parámetro de ruta, pero que aparezca como un parámetro opcional de esa ruta. Puedes hacer eso al colocar un signo de interrogación `?` después del nombre del parámetro. Asegúrate de dar un valor por defecto a la variable correspondiente de la ruta.

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

php

Restricciones con expresiones regulares

Puedes restringir el formato de tus parámetros de ruta usando el método `where` en una instancia de ruta. El método `where` acepta el nombre del parámetro y una expresión regular que defina cómo el parámetro debería estar conformado:

```
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 prefieres que un parámetro de ruta siempre esté restringido por una expresión regular dada, puedes usar el método `pattern`. Deberías definir estos patrones en el método `boot` de tu `RouteServiceProvider`:

```
/**
 * Definir los enlaces de modelo de tus rutas, patrones, filtros, etc.
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');

    parent::boot();
}
```

Una vez que el patrón ha sido definido, es aplicado automáticamente a todas las rutas que usen ese nombre de parámetro:

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

php

Slashes codificados

El componente de rutas de Laravel permite todos los caracteres excepto `/`. Debes explícitamente permitir que `/` sea parte de tu placeholder usando una expresión regular de la condición `where`:

```
Route::get('search/{search}', function ($search) {  
    return $search;  
})->where('search', '.*');
```

php

Nota

Los slashes codificados sólo están soportados dentro del último segmento de la ruta.

Rutas nombradas

Las rutas nombradas permiten la generación de URLs o redirecciones para rutas específicas de una forma conveniente. Puedes especificar un nombre para una ruta al encadenar el método `name` en la definición de la ruta:

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

php

También puedes especificar los nombres de ruta para acciones de controlador:

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

php

Generación de URLs para las rutas nombradas

Una vez que has asignado un nombre a una ruta dada, puedes usar el nombre de la ruta cuando estás generando URLs o redireccionas por medio de la función `route` global:

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

// Generando Redirecciones...
return redirect()->route('profile');
```

Si la ruta nombrada posee parámetros, puedes pasar los parámetros como el segundo argumento de la función `route`. Los parámetros dados serán insertados automáticamente dentro de la URL en sus posiciones correctas:

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

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

Si pasas parámetros adicionales en el arreglo, estos pares clave / valor serán automáticamente agregados a la cadena de consulta generada de la URL:

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

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

// /user/1/profile?photos=yes
```

Inspeccionando la ruta actual

Si requieres determinar si la solicitud actual fue enrutada por una ruta nombrada dada, puedes usar el método `named` en una instancia de Ruta. Por ejemplo, puedes verificar el nombre de ruta actual desde el middleware de una ruta.

```
/**
 * Manejar una solicitud entrante.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Closure $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {
        //
    }

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

php

Los grupos de ruta

Los grupos de ruta permiten que tu compartas atributos de ruta, tales como los middleware o los espacios de nombres, a través de un número grande de rutas sin necesidad de definir esos atributos en cada ruta individual. Los atributos compartidos son especificados en un formato de arreglo como el primer parámetro al método `Route::group`.

Los grupos anidados intentan "fusionar" de forma inteligente los atributos al grupo de sus padres. Los middleware y condiciones `where` son mezcladas (merged) mientras que los nombres, nombres de espacio y prefijos son agregados (appended). Las delimitaciones de nombres de espacio y los slashes en los prefijos de URLs son automáticamente agregados cuando es apropiado.

Los middleware

Para asignar los middleware a todas las rutas dentro de un grupo, puedes usar el método `middleware` antes de la definición del grupo. Los middleware son ejecutados en base al orden en el cual son listados en el arreglo:

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

php

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

Los espacios de nombres

Otro uso común para los grupos de ruta es la asignación del mismo espacio de nombre de PHP a un grupo de controladores usando el método `namespace` :

```
Route::namespace('Admin')->group(function () {
    // Controladores dentro del espacio de nombre "App\Http\Controllers\Admin"
});
```

Recuerda que por defecto, el `RouteServiceProvider` incluye tus archivos de ruta dentro de un grupo de espacio de nombre, permitiéndote que registres rutas de controlador sin especificar el prefijo de espacio de nombre `App\Http\Controllers` completo. Así, puedes necesitar especificar solamente la porción del espacio de nombre que viene después del espacio de nombre `App\Http\Controllers` base.

El enrutamiento de subdominio

Los grupos de ruta también pueden ser usados para manejar enrutamiento de sub-dominio. Los Sub-dominios pueden ser asignados a parámetros de ruta justamente como URLs de ruta, permitiéndote que captures una porción del sub-dominio para uso en tu ruta o controlador. El sub-dominio puede ser especificado al ejecutar el método `domain` antes de definir el grupo.

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

Nota

Para asegurarte de que tus rutas de subdominios son accesibles, debes registrar rutas de subdominios antes de registrar rutas de dominio principal. Esto evitirá que las rutas principales sobrescriban rutas de subdominios que tienen la misma URI.

Prefijos de rutas

El método `prefix` puede ser usado para poner un prefijo a cada ruta en el grupo con una URI dada. Por ejemplo, puedes desear poner un prefijo a todas las URLs de ruta dentro del grupo con `admin` :

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

php

Los prefijos de nombre de ruta

El método `name` puede ser usado para poner prefijo a cada nombre de ruta en el grupo con una cadena dada. Por ejemplo, puedes desear poner prefijo a todos los nombres de ruta agrupados con `admin`. La cadena dada es prefijada al nombre de ruta exactamente cómo es especificada, así que nos aseguraremos de proporcionar el carácter de terminación `.` en el prefijo:

```
Route::name('admin.')->group(function () {
    Route::get('users', function () {
        // Nombre asignado de ruta "admin.users"...
    });
});
```

php

Enlazamiento de modelo de ruta (route model binding)

Cuando estamos inyectando un ID de modelo a una ruta o acción de controlador, usualmente consultarás para obtener el modelo que corresponde a esa ID. El enlazamiento de modelo de ruta de Laravel proporciona una forma conveniente de inyectar directamente las instancias del modelo en tus rutas. Por ejemplo, en lugar de inyectar un ID de usuario, puedes inyectar la instancia del modelo `User` completa que coincide con el ID dado.

Enlazamiento implícito

Laravel resuelve automáticamente los modelos de Eloquent en rutas o acciones de controlador cuyos nombres de variables declaradas coincidan con un nombre de segmento de ruta. Por ejemplo:

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

php

Debido a que la variable `$user` está declarada como el modelo de Eloquent `App\User` y el nombre de variable coincide con el segmento de URI `{user}`, Laravel inyectará automáticamente la instancia del modelo que tenga un ID coincidiendo con el valor correspondiente en la URI de la solicitud. Si una instancia del modelo que coincide no es encontrada en la base de datos, una respuesta HTTP 400 será generada automáticamente.

Personalizando el nombre de clave

Si prefieres que el enlazamiento del modelo use una columna de base de datos distinta del `id` cuando estás obteniendo una clase de modelo dada, puedes sobreescibir el método `getRouteKeyName` en el modelo de Eloquent:

```
/**
 * Obtener la clave de la ruta para el modelo.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'slug';
}
```

php

Enlazamiento explícito

Para registrar un enlazamiento explícito, usa el método `model` del enrutador para especificar la clase para un parámetro dado. Deberías definir tu enlazamiento del modelo explícito en el método `boot` de la clase `RouteServiceProvider`:

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

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

php

Seguido, define una ruta que contenga un parámetro `{user}` :

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

php

Debido a que hemos enlazado todos los parámetros de `{user}` al modelo `App\User`, una instancia `User` será inyectada dentro de la ruta. Así, por ejemplo, una solicitud a `profile/1` inyectará la instancia de la base de datos la cual tiene una ID de `1`.

Si una instancia de modelo que coincide no es encontrada en la base de datos, una respuesta HTTP 404 será generada automáticamente.

Personalizando la lógica de resolución

Si deseas usar tu propia lógica de resolución, puedes usar el método `Route::bind`. La `Closure` que pases al método `bind` recibirá el valor del segmento de URI y debería devolver la instancia de la clase que debería ser inyectada dentro de la ruta:

```
/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    parent::boot();

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

php

Como alternativa, puedes sobreescribir el método `resolveRouteBinding` en tu modelo Eloquent. Este método recibirá el valor del segmento URI y debe devolver la instancia de la clase que se debe injectar en la ruta:

```
/*
 * Retrieve the model for a bound value.
 *
 * @param mixed $value
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveRouteBinding($value)
{
    return $this->where('name', $value)->firstOrFail();
}
```

Rutas fallback

Usando el método `Route::fallback`, puedes definir una ruta que será ejecutada cuando ninguna otra ruta coincida con la petición entrante. Típicamente, las peticiones no gestionadas automáticamente mostrarán una página 404 a través del manejador de excepciones de tu aplicación. Sin embargo, ya que puedes definir la ruta `fallback` dentro de tu archivo `routes/web.php`, todo middleware en el grupo `web` aplicará a la ruta. Eres libre de añadir middleware adicionales a esta ruta de ser necesario:

```
Route::fallback(function () {
    //
});
```

Nota

La ruta alternativa siempre debe ser la última ruta registrada por tu aplicación.

Límite de rango

Laravel incluye un `middleware` para limitar el rango de acceso a rutas dentro de tu aplicación. Para empezar, asigna el middleware `throttle` a una ruta o grupo de rutas. El middleware `throttle`

acepta dos parámetros que determinan el máximo número de peticiones que pueden hacerse en un número de minutos dado. Por ejemplo, especifiquemos que un usuario autenticado puede acceder al siguiente grupo de rutas sesenta veces por minuto:

```
Route::middleware('auth:api', 'throttle:60,1')->group(function () {  
    Route::get('/user', function () {  
        //  
    });  
});
```

php

Limite de rango dinámico

Puedes especificar un máximo de peticiones dinámicas basado en un atributo del modelo `User` autenticado. Por ejemplo, si tu modelo `User` contiene un atributo `rate_limit`, puedes pasar el nombre del atributo al middleware `throttle` de modo que sea usado para calcular el conteo máximo de peticiones:

```
Route::middleware('auth:api', 'throttle:rate_limit,1')->group(function () {  
    Route::get('/user', function () {  
        //  
    });  
});
```

php

Límites de rango distintos para usuarios autenticados y no autenticados

Puedes especificar diferentes límites de rango para usuarios autenticados y no autenticados. Por ejemplo, puedes especificar un máximo de `10` peticiones por minuto para usuarios no autenticados y `60` para usuarios autenticados:

```
Route::middleware('throttle:10|60,1')->group(function () {  
    //  
});
```

php

También puedes combinar esta funcionalidad con límites de rango dinámicos. Por ejemplo, si tu usuario `Model` contiene un atributo `rate_limit`, puedes pasar el nombre del atributo al middleware `throttle` para que sea usado para calcular el número máximo de peticiones para usuarios autenticados:

```
Route::middleware('auth:api', 'throttle:10|rate_limit,1')->group(function () {  
    Route::get('/user', function () {  
        //  
    });  
});
```

Segmentos de límite de rango

Típicamente, probablemente especificarás un límite de rango para toda tu API. Sin embargo, tu aplicación puede requerir diferentes límites de rango para diferentes segmentos de tu API. si este es el caso, necesitarás pasar un nombre de segmento como tercer argumento del middleware `throttle` :

```
Route::middleware('auth:api')->group(function () {  
    Route::middleware('throttle:60,1,default')->group(function () {  
        Route::get('/servers', function () {  
            //  
        });  
    });  
    Route::middleware('throttle:60,1,deletes')->group(function () {  
        Route::delete('/servers/{id}', function () {  
            //  
        });  
    });  
});
```

La suplantación de método del formulario

Los formularios HTML no soportan acciones `PUT` , `PATCH` o `DELETE` . Así que, cuando estés definiendo rutas `PUT` , `PATCH` o `DELETE` que son llamadas desde un formulario HTML, necesitarás agregar un campo `_method` oculto para el formulario. El valor enviado con el campo `_method` será usado como el método de solicitud HTTP:

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

Puedes usar la directiva Blade `@method` para generar la entrada `_method`:

```
<form action="/foo/bar" method="POST">
    @method('PUT')
    @csrf
</form>
```

php

Accesando la ruta actual

Puedes usar los métodos `current`, `currentRouteName`, y `currentRouteAction` en la clase facade `Route` para accesar la información sobre el manejador de ruta de la solicitud entrante:

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

$name = Route::currentRouteName();

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

php

Consulta la documentación de la API sobre la [clase subyacente de la clase facade `Route`](#) y la [instancia de ruta](#) para revisar todos los métodos disponibles.

Middleware

- [Introducción](#)
- [Definiendo un Middleware](#)
- [Registrando un Middleware](#)
 - [Middleware Globales](#)
 - [Asignando un Middleware a una Ruta](#)
 - [Grupos de middleware](#)

- Clasificación de Middleware
- Parámetros en los Middleware
- Middleware Terminable

Introducción

Los Middleware proporcionan un mecanismo conveniente para filtrar consultas HTTP en toda tu aplicación. Por ejemplo, Laravel incluye un middleware que verifica si el usuario de tu aplicación está autenticado. Si el usuario no está autenticado, el middleware redireccionará al usuario a la pantalla de inicio de sesión. Sin embargo, si el usuario es autenticado, el middleware permitirá que la consulta proceda dentro de la aplicación.

Middleware adicionales pueden ser escritos para realizar una variedad de tareas además de autenticar. Un núcleo de un middleware podría ser responsable de agregar los encabezados apropiados para todas las respuestas que va dejando tu aplicación. Un middleware de registro podría registrar todas las consultas entrantes en tu aplicación.

Hay varios middleware incluidos en el framework Laravel, incluyendo middleware para autenticación y protección CSRF. Todos esos middleware están localizados en el directorio `app/Http\Middleware` .

Definiendo un Middleware

Para crear un nuevo middleware, usa el comando de Artisan: `make:middleware`

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

php

Este comando ubicará una nueva clase `CheckAge` dentro de tu directorio `app/Http\Middleware` .

En este middleware, nosotros solo permitiremos el acceso a la ruta si la `edad` suministrada es mayor que 200. De otra forma, redireccionaremos a los usuarios de vuelta a la URL `home` :

```
<?php  
  
namespace App\Http\Middleware;  
  
use Closure;  
  
class CheckAge
```

php

```
{  
    /**  
     * 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 puedes ver, si la `edad` dada es menor o igual a `200`, el middleware retornará una redirección HTTP al cliente; de otra forma, la solicitud pasará más adentro de la aplicación. Para pasar la solicitud más profundo dentro de la aplicación (permitiendo al middleware "pasar") llama al callback `$next` con el `$request`.

Es mejor visualizar el middleware como una serie de "capas" que deben pasar las solicitudes HTTP antes de que lleguen a tu aplicación. Cada capa puede examinar la solicitud e incluso rechazarla por completo.

TIP

Todos los middleware son resueltos a través del contenedor de servicio, de esta forma, puedes declarar el tipo de cualquier dependencia que necesites dentro del constructor del middleware.

Middleware Before y After

Que un middleware se ejecute antes o después de una solicitud depende del middleware en sí mismo. Por ejemplo, el siguiente middleware podría realizar alguna tarea **antes** que la solicitud sea manejada por la aplicación:

```
<?php  
  
namespace App\Http\Middleware;  
php
```

```
use Closure;

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

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

Sin embargo, este middleware podría realizar esta tarea **después** de que la solicitud sea manejada 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;
    }
}
```

php

Registrando un Middleware

Middleware Globales

Si tu quieres que un middleware corra durante cada solicitud HTTP a tu aplicación, lista la clase del middleware en la propiedad `$middleware` de tu clase `app/Http/Kernel.php`.

Asignando un Middleware a las Rutas

Si te gustaría asignar un middleware a rutas específicas, deberías primero asignar una clave al middleware en tu archivo `app/Http/Kernel.php`. Por defecto, la propiedad `$routeMiddleware` de esta clase contiene entradas para los middleware incluidos con Laravel. Para agregar uno propio, adjúntalo a esta lista y asignale una clave de tu elección:

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

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

Una vez el middleware ha sido definido en el núcleo HTTP, puedes usar el método `middleware` para asignar un middleware a una ruta:

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

Puedes además asignar multiples middleware a la ruta:

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

Cuando asignas middleware, puedes además pasar un nombre de clase plenamente calificado:

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

php

Grupos de Middleware

Algunas veces puedes querer agrupar varios middleware bajo una sola clave para hacerlos más fáciles de asignar a las rutas. Puedes hacer esto usando la propiedad `$middlewareGroups` de tu kernel HTTP.

Por defecto, Laravel viene con los grupos de middleware `web` y `api` que contienen middleware comunes que puedes aplicar a la UI de tu web y a las rutas de tu 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',  
    ],  
];
```

php

Los grupos de Middleware pueden ser asignados a las rutas y las acciones de los controladores usando la misma sintaxis como los middleware individuales. De nuevo, los grupos de middleware hacen más conveniente asignar muchos middleware a una ruta a la vez:

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

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

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

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

php

TIP

Por defecto, el grupo de middleware `web` es automáticamente aplicado a tu archivo `routes/web.php` por el `RouteServiceProvider`.

Clasificación de Middleware

Raramente, necesitarás que tu middleware se ejecute en un orden específico pero no tienes control sobre su orden cuando son asignados a una ruta. En este caso, puedes especificar la prioridad de tu middleware usando la propiedad `$middlewarePriority` de tu archivo `app/Http/Kernel.php`:

```
/**
 * The priority-sorted list of middleware.
 *
 * This forces non-global middleware to always be in the given order.
 *
 * @var array
 */
protected $middlewarePriority = [
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\Authenticate::class,
    \Illuminate\Session\Middleware\AuthenticateSession::class,
```

php

```
\Illuminate\Routing\Middleware\SubstituteBindings::class,  
\Illuminate\Auth\Middleware\Authorize::class,  
];
```

Parámetros en los middleware

Los middleware pueden además recibir parámetros adicionales. Por ejemplo, si tu aplicación necesita verificar que el usuario autenticado tiene un "rol" dado antes de ejecutar una acción dada, podrías crear un middleware `CheckRole` que reciba un nombre de rol como un argumento adicional.

Los parámetros adicionales en el middleware serán pasados al middleware después del 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);  
    }  
}
```

php

Los parámetros en los middleware pueden ser especificados al definir la ruta separando el nombre del middleware y los parámetros con `:`. Múltiples parámetros deben ser delimitados por comas:

```
Route::put('post/{id}', function ($id) {  
    //  
})->middleware('role:editor');
```

php

Middleware Terminable

Algunas veces un middleware puede necesitar hacer algún trabajo después de que la respuesta HTTP ha sido preparada. Si defines un método `terminate` en tu middleware y tu servidor web está usando FastCGI, el método `terminate` será llamado automáticamente después de que la respuesta sea enviada 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...  
    }  
}
```

php

El método `terminate` debería recibir tanto la consulta como la respuesta. Una vez has definido el middleware terminable, deberías agregarlo a la lista de rutas o como un middleware global en el archivo `app/Http/Kernel.php`.

Cuando llamas al método `terminate` en tu middleware, Laravel resolverá una instancia fresca del middleware del [contenedor de servicios](#). Si deseas utilizar la misma instancia middleware cuando los métodos `handle` y `terminate` sean llamados, registra el middleware con el contenedor usando el método `singleton` del contenedor. Típicamente esto debería ser realizado en el método `register` de tu `AppServiceProvider.php`:

Protección CSRF

- [Introducción](#)
- [Excluyendo URLs](#)
- [X-CSRF-Token](#)
- [X-XSRF-Token](#)

Introducción

Laravel hace que sea fácil proteger tu aplicación de ataques de tipo [cross-site request forgery](#) (CSRF). Los ataques de tipo CSRF son un tipo de explotación de vulnerabilidad malicioso por el cual comandos no autorizados son ejecutados en nombre de un usuario autenticado.

Laravel genera automáticamente un "token" CSRF para cada sesión de usuario activa manejada por la aplicación. Este token es usado para verificar que el usuario autenticado es quien en realidad está haciendo la petición a la aplicación.

En cualquier momento que definas un formulario HTML en tu aplicación, debes incluir un campo de token CSRF en el formulario con el propósito de que el middleware para protección CSRF pueda validar la solicitud. Puedes usar la directiva de Blade `@csrf` para generar el campo de token:

```
<form method="POST" action="/profile">  
    @csrf
```

php

```
...  
</form>
```

El middleware `VerifyCsrfToken`, el cual es incluido en el grupo de middleware `web`, verificará automáticamente que el token en el campo de la solicitud coincida con el almacenado en la sesión.

Tokens CSRF & JavaScript

Cuando se crean aplicaciones controladas por JavaScript, es conveniente hacer que tu biblioteca HTTP de JavaScript agregue el token CSRF a cada petición saliente. Por defecto, la librería HTTP Axios proporcionada en el archivo `resources/js/bootstrap.js` automáticamente envia un header `X-XSRF-TOKEN` usando el valor de la cookie encriptada `XSRF-TOKEN`. Si no estás usando esta librería, necesitarás configurar de forma manual este comportamiento en tus aplicaciones.

Excluyendo las URIs de la protección CSRF

Algunas veces puedes desear excluir un conjunto de URIs de la protección CSRF. Por ejemplo, si estás usando `Stripe` para procesar pagos y estás utilizando su sistema webhook, necesitarás excluir tu ruta de manejador webhook de Stripe de la protección CSRF ya que Stripe no sabrá que token CSRF enviar a sus rutas.

Típicamente, deberías colocar este tipo de rutas afuera del grupo de middleware `web` que el `RouteServiceProvider` aplica a todas las rutas en el archivo `routes/web.php`. Sin embargo, también puedes excluir las rutas al añadir sus URIs 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/*',
    'http://example.com/foo/bar',
    'http://example.com/foo/*',
];
}
```

TIP

El middleware CSRF está deshabilitado automáticamente al ejecutar pruebas.

X-CSRF-TOKEN

Además de comprobar el token CSRF como parámetro POST, el middleware `VerifyCsrfToken` también comprobará el encabezado de solicitud `X-CSRF-TOKEN`. Podrías, por ejemplo, almacenar el token en una etiqueta `meta` de HTML:

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

php

Entonces, una vez que has creado la etiqueta `meta`, puedes instruir una biblioteca como jQuery para añadir automáticamente el token a todos los encabezados de las peticiones. Esto proporciona protección CSRF fácil y conveniente para tus aplicaciones basadas en AJAX.

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});
```

php

X-XSRF-TOKEN

Laravel almacena el token CSRF actual en una cookie `XSRF-TOKEN` encriptada que es incluida con cada respuesta generada por el framework. Puedes usar el valor del cookie para establecer el encabezado de la solicitud `X-XSRF-TOKEN`.

Esta cookie primeramente es enviada por conveniencia ya que algunos frameworks JavaScript y librerías, como Angular y Axios colocan automáticamente su valor en el encabezado `X-XSRF-TOKEN` en las solicitudes de mismo origen.

TIP

Por defecto, el archivo `resources/js/bootstrap.js` incluye la librería HTTP Axios que enviará automáticamente esto por ti.

Controladores

- [Introducción](#)
- [Controladores básicos](#)
 - [Definiendo controladores](#)
 - [Controladores y espacios de nombres](#)
 - [Controladores de acción única](#)
- [Middleware de controlador](#)
- [Controladores de recursos](#)
 - [Rutas de recursos parciales](#)
 - [Recursos anidados](#)
 - [Nombrando rutas de recursos](#)
 - [Nombrando parámetros de rutas de recursos](#)
 - [Configuración regional para URLs de recursos](#)
 - [Complementando controladores de recursos](#)
- [Inyección de dependencias y controladores](#)
- [Caché de rutas](#)

Introducción

En lugar de definir toda la lógica de manejo de solicitud como Closure en archivos de ruta, puedes desear organizar este comportamiento usando clases Controller. Los controladores pueden agrupar la lógica de manejo de solicitud relacionada dentro de una sola clase. Los controladores son almacenados en el directorio `app/Http\Controllers`.

Controladores básicos

Definiendo controladores

A continuación se muestra un ejemplo de una clase de controlador básica. Nota que el controlador extiende la clase de controlador base incluida con Laravel. La clase base proporciona unos cuantos métodos de conveniencia tal como el método `middleware`, el cual puede ser usado para conectar un middleware a acciones de controlador:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use App\User;  
  
class UserController extends Controller  
{  
    /**  
     * Show the profile for the given user.  
     *  
     * @param int $id  
     * @return View  
     */  
    public function show($id)  
    {  
        return view('user.profile', ['user' => User::findOrFail($id)]);  
    }  
}
```

Puedes definir una ruta a esta acción de controlador de esta forma:

```
Route::get('user/{id}', 'UserController@show');
```

Ahora, cuando una solicitud coincide con la URI de la ruta especificada, se ejecutará el método `show` de la clase `UserController`. Los parámetros de ruta también se pasarán al método.

TIP

Los controladores no están **obligados** a extender de la clase base. Sin embargo, no tendrás acceso a características de conveniencia tales como los métodos `middleware`, `validate`, y `dispatch`.

Controladores y espacios de nombres

Es muy importante notar que no necesitamos especificar el espacio de nombre completo del controlador al momento de definir la ruta del controlador. Debido a que el `RouteServiceProvider` carga sus archivos de ruta dentro de un grupo de ruta que contiene el espacio de nombre, solamente necesitaremos la porción del nombre de la clase que viene después de la porción `App\Http\Controllers` del espacio de nombre.

Si eliges anidar tus controladores dentro del directorio `App\Http\Controllers`, usa el nombre de clase específico relativo al espacio de nombre raíz `App\Http\Controllers`. Así, si tu clase de controlador completa es `App\Http\Controllers\Photos\AdminController`, deberías registrar rutas al controlador de esta forma:

```
Route::get('foo', 'Photos\AdminController@method');
```

php

Controladores de acción única

Si prefieres definir un controlador que maneja solamente una acción única, debes colocar un único método `__invoke` en el controlador:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use App\User;  
  
class ShowProfile extends Controller  
{
```

php

```
/**
 * Show the profile for the given user.
 *
 * @param int $id
 * @return View
 */
public function __invoke($id)
{
    return view('user.profile', ['user' => User::findOrFail($id)]);
}
```

Al momento de registrar rutas para controladores de acción única, no necesitarás especificar un método:

```
Route::get('user/{id}', 'ShowProfile');
```

php

Puedes generar un controlador invocable usando la opción `--invokable` del comando Artisan `make:controller` :

```
php artisan make:controller ShowProfile --invokable
```

php

Middleware de controlador

Los [Middleware](#) pueden ser asignados a las rutas del controlador en tus archivos de ruta:

```
Route::get('profile', 'UserController@show')->middleware('auth');
```

php

Sin embargo, es más conveniente especificar los middleware dentro del constructor de tu controlador. Usando el método `middleware` del constructor de tu controlador, puedes asignar fácilmente los middleware a la acción del controlador. Incluso puedes restringir los middleware a sólo ciertos métodos en la clase del controlador:

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.

```

php

```
* @return void
*/
public function __construct()
{
    $this->middleware('auth');

    $this->middleware('log')->only('index');

    $this->middleware('subscribed')->except('store');
}
```

También los controladores permiten que registres los middleware usando una Closure. 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);
});
```

php

TIP

Puedes asignar los middleware a un subconjunto de acciones de controlador, esto puede indicar que tu controlador está creciendo demasiado. En lugar de esto, considera dividir tu controlador en varios controladores más pequeños.

Controladores de recursos

El enrutamiento de recurso de Laravel asigna las rutas típicas "CRUD" a un controlador con una sola línea de código. Por ejemplo, puedes desear crear un controlador que maneje todas las solicitudes HTTP para "photos" almacenadas por tu aplicación. Usando el comando Artisan `make:controller` , podemos crear fácilmente tal controlador:

```
php artisan make:controller PhotoController --resource
```

php

Este comando creará un controlador en `app/Http/Controllers/PhotoController.php`. El controlador contendrá un método para cada una de las operaciones de recursos disponibles.

Seguidamente, puedes registrar una ruta de recurso genérica al controlador:

```
Route::resource('photos', 'PhotoController');
```

php

Esta declaración de ruta única crea varias rutas para manejar una variedad de acciones del recurso. El controlador generado ya tendrá los métodos separados para cada una de las acciones, incluyendo comentarios que te informan de los verbos HTTP y URIs que manejan.

Puedes registrar muchos controladores de recursos a la vez pasando un arreglo al método

`resources` :

```
Route::resources([
    'photos' => 'PhotoController',
    'posts' => 'PostController'
]);
```

php

Acciones manejadas por el controlador de recursos

Tipo	URI	Acción	Nombre de la Ruta
GET	/photos	índice	photos.index
GET	/photos/create	crear	photos.create
POST	/photos	guardar	photos.store
GET	/photos/{photo}	mostrar	photos.show
GET	/photos/{photo}/edit	editar	photos.edit
PUT/PATCH	/photos/{photo}	actualizar	photos.update
DELETE	/photos/{photo}	eliminar	photos.destroy

Especificando el modelo del recurso

Si estás usando el enlace de modelo de ruta (route model binding) y deseas que los métodos del controlador de recursos declaren el tipo de una instancia de modelo, puedes usar la opción `--model` al momento de generar el controlador:

```
php artisan make:controller PhotoController --resource --model=Photo
```

php

Suplantar los métodos de formulario

Debido a que los formularios no pueden hacer solicitudes `PUT`, `PATCH`, o `DELETE`, necesitarás agregar un campo `_method` oculto para suplantar estos verbos HTTP. La directiva de Blade `@method` puede crear este campo para ti:

```
<form action="/foo/bar" method="POST">
    @method('PUT')
</form>
```

php

Rutas de recursos parciales

Al momento de declarar una ruta de recurso, puedes especificar un subconjunto de acciones que el controlador debería manejar en lugar de conjunto completo de acciones por defecto.

```
Route::resource('photos', 'PhotoController')->only([
    'index', 'show'
]);

Route::resource('photos', 'PhotoController')->except([
    'create', 'store', 'update', 'destroy'
]);
```

php

Rutas de recursos para APIs

Al momento de declarar rutas de recursos que serán consumidas por APIs, normalmente te gustará excluir rutas que presentan plantillas HTML tales como `create` y `edit`. Por conveniencia, puedes usar el método `apiResource` para excluir automáticamente éstas dos rutas:

```
Route::apiResource('photos', 'PhotoController');
```

Puedes registrar muchos controladores de recursos de API de una sola vez pasando un arreglo al método `apiResources` :

```
Route::apiResources([
```

```
    'photos' => 'PhotoController',
    'posts' => 'PostController'
```

```
]);
```

php

Para generar rápidamente un controlador de recursos API que no incluya los métodos `create` o `edit`, usa la opción `--api` cuando ejecutas el comando `make:controller` :

```
php artisan make:controller API/PhotoController --api
```

php

Recursos anidados

Algunas veces necesitarás definir rutas a un recurso "anidado". Por ejemplo, una imagen puede tener múltiples "comentarios" que podrían estar atados a ésta. Para "anidar" controladores de recursos, usa la notación de "punto" en la declaración de tu ruta:

```
Route::resource('photos.comments', 'PhotoCommentController');
```

php

Esta ruta registrará un recurso "anidado" al cual se puede acceder mediante URLs como la siguiente: `photos/{photos}/comments/{comments}`.

Anidación superficial

A menudo, no es completamente necesario tener tanto el ID del padre como del hijo dentro de una URI dado que el ID del hijo es un identificador único. Al usar identificadores únicos como claves primarias de auto incremento para identificar tus modelos en segmentos de una URI, puedes elegir usar "anidación superficial":

```
Route::resource('photos.comments', 'CommentController')->shallow();
```

php

La definición de ruta de arriba definirá las siguientes rutas:

Verb	URI	Action	Route Name
GET	/photos/{photo}/comments	index	photos.comments.index
GET	/photos/{photo}/comments/create	create	photos.comments.create
POST	/photos/{photo}/comments	store	photos.comments.store
GET	/comments/{comment}	show	comments.show
GET	/comments/{comment}/edit	edit	comments.edit
PUT/PATCH	/comments/{comment}	update	comments.update
DELETE	/comments/{comment}	destroy	comments.destroy

[◀](#) [▶](#)

Nombrando rutas de recursos

De forma predeterminada, todas las acciones de controlador de recursos tienen un nombre de ruta; sin embargo, puedes sobrescribir esos nombres al pasar un arreglo de nombres con tus opciones:

```
Route::resource('photos', 'PhotoController')->names([
    'create' => 'photos.build'
]);
```

php

Nombrando parámetros de rutas de recursos

De forma predeterminada, `Route::resource` creará los parámetros de ruta para tus rutas de recursos basado en la versión "singularizada" del nombre de recurso. Puedes sobrescribir fácilmente esto para cada recurso usando el método `parameters`. El arreglo pasado al método `parameters` debería ser un arreglo asociativo de nombres de recursos y nombres de parámetros:

```
Route::resource('users', 'AdminUserController')->parameters([
    'users' => 'admin_user'
]);
```

php

El ejemplo anterior genera las URLs siguientes para la ruta `show` del recurso:

```
/users/{admin_user}
```

php

Configuración regional para URLs de recursos

De forma predeterminada, `Route::resource` creará URLs de recursos usando verbos en Inglés. Si necesitas configurar los verbos de acción `create` y `edit` a un idioma, puedes usar el método `Route::resourceVerbs`. Esto puede ser hecho en el método `boot` de tu `AppServiceProvider`:

```
use Illuminate\Support\Facades\Route;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);
}
```

php

Una vez que los verbos han sido personalizados, un registro de ruta de recurso tal como

```
Route::resource('fotos', 'PhotoController')
```

producirá las siguientes URLs:

```
/fotos/crear
```

php

```
/fotos/{foto}/editar
```

Complementando controladores de recursos

Si necesitas agregar rutas adicionales para un controlador de recursos más allá del conjunto predeterminado de rutas de recursos, deberías definir esas rutas antes de que ejecutes

`Route::resource`; de otra forma, las rutas definidas por el método `resource` pueden tomar precedencia involuntariamente sobre tus rutas complementarias:

```
Route::get('photos/popular', 'PhotoController@method');
```

php

```
Route::resource('photos', 'PhotoController');
```

TIP

Recuerda mantener la lógica de tus controladores enfocada. Si te encuentras a ti mismo necesitando rutinariamente métodos fuera del conjunto típico de acciones de recurso, considera dividir tu controlador en dos controladores más pequeños.

Inyección de dependencias y controladores

Inyección al constructor

El [contenedor de servicio](#) de Laravel es usado para resolver todos los controladores de Laravel. Como resultado, estás habilitado para declarar el tipo de cualquier dependencia que tu controlador pueda necesitar en su constructor. Las dependencias declaradas serán automáticamente resueltas e inyectadas dentro de 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  
     */
```

php

```
public function __construct(UserRepository $users)
{
    $this->users = $users;
}
```

También puedes declarar el tipo de cualquier [Contrato de Laravel](#). Si el contenedor puede resolverlo, puedes declararlo. Dependiendo de tu aplicación, injectar tus dependencias dentro de tu controlador puede proporcionar mejor capacidad para pruebas.

Inyección de métodos

Adicional a la inyección al constructor, también puedes declarar el tipo de dependencias en los métodos de tu controlador. Un caso de uso común para la inyección de método está inyectando la instancia `Illuminate\Http\Request` dentro de tus métodos de 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 tu método de controlador también está esperando entrada de un parámetro de ruta, lista tus argumentos de ruta después de tus otras dependencias. Por ejemplo, si tu ruta es definida como esto:

```
Route::put('user/{id}', 'UserController@update');
```

php

Aún puedes declarar el tipo de la clase `Illuminate\Http\Request` y acceder a tu parámetro `id` al definir tu método de controlador de la siguiente manera:

```
<?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)  
    {  
        //  
    }  
}
```

php

Caché de rutas

Nota

Las rutas basadas en Closure no pueden ser cacheadas. Para usar caché de rutas, debes convertir cualquiera de las rutas Closure a clases de controlador.

Si tu aplicación está usando exclusivamente rutas basadas en controlador, deberías tomar ventaja de la caché de rutas de Laravel. Usar la cache de rutas reducirá drásticamente la cantidad de tiempo que toma registrar todas las rutas de tu aplicación. En algunos casos, incluso la rapidez de tu registro de rutas puede llegar a ser hasta 100 veces más rápida.

```
php artisan route:cache
```

php

Después de ejecutar este comando, tu archivo de rutas cacheado será cargado en cada solicitud.

Recuerda, si agregas cualquier ruta nueva necesitarás generar una caché de ruta nueva. Debido a esto, deberías ejecutar solamente el comando `route:cache` durante el despliegue o puesta en producción del proyecto.

Puedes usar el comando `route:clear` para limpiar la caché de ruta:

```
php artisan route:clear
```

php

Solicitudes HTTP

- [Accediendo a la solicitud](#)
 - [Ruta y método de la solicitud](#)
 - [Solicitudes PSR-7](#)
- [Recorte y normalización de entrada](#)
- [Obteniendo datos ingresados](#)
 - [Datos antiguos](#)
 - [Cookies](#)
- [Archivos](#)
 - [Obteniendo archivos cargados](#)
 - [Almacenando archivos cargados](#)
- [Configurando proxies de confianza](#)

Accediendo a la solicitud

Para obtener una instancia de la solicitud HTTP actual por medio de una inyección de dependencia, deberías poner la referencia de la clase `Illuminate\Http\Request` en tu método de controlador. La instancia de la solicitud entrante automáticamente será inyectada por el contenedor de servicio:

```
<?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 Y parámetros de rutas

Si tu método de controlador también está esperando la entrada de un parámetro de ruta deberías listar tus parámetros de ruta después de tus otras dependencias. Por ejemplo, si tu ruta es definida como sigue:

```
Route::put('user/{id}', 'UserController@update');
```

Todavía puedes poner la referencia de la clase `Illuminate\Http\Request` y acceder a tu parámetro de ruta `id` al definir tu método de controlador como sigue:

```
<?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 la solicitud a través de closures de rutas

También puedes poner la referencia de la clase `Illuminate\Http\Request` en una Closure de ruta. El contenedor de servicio automáticamente inyectará la solicitud entrante dentro de la Closure que es ejecutada:

```
use Illuminate\Http\Request;                                         php

Route::get('/', function (Request $request) {
    //
});
```

Ruta y método de la solicitud

La instancia `Illuminate\Http\Request` proporciona una variedad de métodos para examinar la solicitud HTTP para tu aplicación y extiende la clase

`Symfony\Component\HttpFoundation\Request`. Discutiremos algunos de los métodos más importantes a continuación.

Obteniendo la ruta de la solicitud

El método `path` devuelve la información de ruta de la solicitud. Así, si la solicitud entrante tiene como destino `http://domain.com/foo/bar`, el método `path` devolverá `foo/bar`:

```
$uri = $request->path();
```

php

El método `is` te permite verificar que la ruta de la solicitud entrante coincide con un patrón dado.

Puedes usar el carácter `*` para especificar que cualquier cadena puede coincidir al momento de utilizar este método:

```
if ($request->is('admin/*')) {  
    //  
}
```

php

Obteniendo la URL de la solicitud

Para obtener la URL completa de la solicitud entrante puedes usar los métodos `url` o `fullUrl`. El método `url` devolverá la URL sin la cadena de la consulta, mientras que el método `fullUrl` si la incluye:

```
// Without Query String...  
$url = $request->url();  
  
// With Query String...  
$url = $request->fullUrl();
```

php

Obteniendo el método de la solicitud

El método `method` devolverá el verbo HTTP de la solicitud. Puedes usar el método `isMethod` para verificar que el verbo HTTP coincide con una cadena dada:

```
$method = $request->method();  
  
if ($request->isMethod('post')) {  
    //  
}
```

php

Solicitudes PSR-7

El [estándar PSR-7](#) especifica interfaces para mensajes HTTP, incluyendo solicitudes y respuestas. Si prefieres obtener una instancia de una solicitud PSR-7 en lugar de una solicitud de Laravel, primero necesitarás instalar algunos paquetes de terceros. Laravel usa el componente *Symfony HTTP Message Bridge* para convertir solicitudes y respuestas típicas de Laravel en implementaciones compatibles con PSR-7:

```
composer require symfony/psr-http-message-bridge  
composer require nyholm/psr7
```

php

Una vez que has instalado estos paquetes, puedes obtener una solicitud PSR-7 al colocar la referencia de la interface de solicitud en tu Closure de ruta o método de controlador:

```
use Psr\Http\Message\ServerRequestInterface;  
  
Route::get('/', function (ServerRequestInterface $request) {  
    //  
});
```

php

TIP

Si devuelves una instancia de respuesta PSR-7 desde una ruta o controlador, automáticamente será convertida de vuelta a una instancia de respuesta de Laravel y será mostrada por el framework.

Recorte y normalización de entrada

De forma predeterminada, Laravel incluye los middleware `TrimStrings` y `ConvertEmptyStringsToNull` en la pila de middleware global de tu aplicación. Estos middleware son listados en la pila por la clase `App\Http\Kernel`. Estos middleware automáticamente recortarán todos los campos de cadena entrantes en la solicitud, así como convertirán cualquier campo de cadena vacío a `null`. Esto permite que no tengas que preocuparte sobre estos asuntos de normalización en tus rutas y controladores.

Si prefieres deshabilitar este comportamiento, puedes remover los dos middleware de tu pila de middleware de tu aplicación al eliminarlos de la propiedad `$middleware` de tu clase `App\Http\Kernel`.

Obteniendo datos ingresados

Obteniendo todos los datos ingresados

También puedes obtener todos los datos ingresados en forma de arreglo usando el método `all`:

```
$input = $request->all();
```

php

Obteniendo el valor de un campo

Usando unos pocos métodos básicos, puedes acceder a todos los datos ingresados por el usuario desde la instancia `Illuminate\Http\Request` sin preocuparte por cuál verbo HTTP fue usado por la solicitud. Sin importar el verbo HTTP, el método `input` puede ser usado para obtener la entrada de usuario:

```
$name = $request->input('name');
```

php

Puedes pasar un valor predeterminado como segundo argumento del método `input`. Este valor será devuelto si el valor de entrada solicitado no está presente en la solicitud:

```
$name = $request->input('name', 'Sally');
```

php

Al momento de trabajar con formularios que contienen arreglos de campos, usa notación de "punto" para acceder a estos arreglos:

```
$name = $request->input('products.0.name');
```

php

```
$names = $request->input('products.*.name');
```

Puedes llamar al método `input` sin ningún argumento para retornar todos los valores como arreglo asociativo:

```
$input = $request->input();
```

php

Obteniendo datos desde la cadena de consulta

Mientras el método `input` obtiene valores de la porción de datos de la solicitud completa (incluyendo la cadena de consulta), el método `query` solamente obtendrá valores de la cadena de consulta:

```
$name = $request->query('name');
```

php

Si los datos de los valores de la cadena de consulta solicitada no están presentes, el segundo argumento de este método será devuelto:

```
$name = $request->query('name', 'Helen');
```

php

Puedes ejecutar el método `query` sin ningún argumento con el propósito de obtener todos los valores de la cadena de consulta como un arreglo asociativo:

```
$query = $request->query();
```

php

Recuperando datos por medio de propiedades dinámicas

También puedes acceder a los datos ingresados por el usuario usando propiedades dinámicas en la instancia `Illuminate\Http\Request`. Por ejemplo, si uno de los formularios de tu aplicación contiene un campo `name`, puedes acceder al valor del campo de la siguiente forma:

```
$name = $request->name;
```

php

Al momento de usar propiedades dinámicas, Laravel primero buscará por el valor del parámetro en la porción de datos de la solicitud. Si no está presente, buscará el campo en los parámetros de ruta.

Obteniendo valores JSON

Al momento de enviar solicitudes JSON a tu aplicación, puedes acceder a los datos JSON por medio del método `input` al tiempo que el encabezado `Content-Type` de la solicitud sea establecido

apropiadamente a `application/json`. Incluso puedes usar sintaxis `."` para buscar adentro de los arreglos JSON:

```
$name = $request->input('user.name');
```

php

Retornando valores de campos booleanos

Al lidiar con elementos HTML como checkboxes, tu aplicación puede recibir valores *verdaderos o falsos* que son en realidad cadenas. Por ejemplo, "true" o "on". Por conveniencia, puedes usar el método `boolean` para retornar estos valores como booleanos. El método `boolean` retorna `true` para 1, "1", true, "true", "on" y "yes". Todos los demás valores retornarán `false`:

```
$archived = $request->boolean('archived');
```

php

Obteniendo una porción de los datos ingresados

Si necesitas obtener un subconjunto de los datos ingresados, puedes usar los métodos `only` y `except`. Ambos métodos aceptan un solo arreglo 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');
```

php

TIP

El método `only` devuelve todos los pares clave / valor que solicites; sin embargo, no devolverá pares clave / valor que no estén presentes en la solicitud.

Determinando si un Valor ingresado está presente

Deberías usar el método `has` para determinar si un valor está presente en la solicitud. El método `has` devuelve `true` si el valor está presente en la solicitud:

```
if ($request->has('name')) {  
    //  
}
```

php

Cuando es dado un arreglo, el método `has` determinará si todos los valores especificados están presentes:

```
if ($request->has(['name', 'email'])) {  
    //  
}
```

php

El método `hasAny` retorna verdadero si alguno de los valores especificados estan presentes:

```
if ($request->hasAny(['name', 'email'])) {  
    //  
}
```

php

Si prefieres determinar si un valor está presente en la solicitud y no esté vacío, puedes usar el método `filled` :

```
if ($request->filled('name')) {  
    //  
}
```

php

Para comprobar si la clave dada está ausente en la petición, puedes usar el método `missing` :

```
if ($request->missing('name')) {  
    //  
}
```

php

Entrada antigua

Laravel permite que mantengas los datos de una solicitud durante la próxima solicitud. Esta característica es útil particularmente para volver a llenar los formularios después de detectar errores de validación. Sin embargo, si estás usando las [características de validación](#) incluidas con Laravel, es poco probable que

necesites usar manualmente estos métodos, ya que algunas de las facilidades de validación integradas con Laravel las ejecutarán automáticamente.

Enviando datos a la sesión

El método `flash` en la clase `Illuminate\Http\Request` enviará los datos ingresados a la `sesión` para que así estén disponibles durante la próxima solicitud realizada por el usuario:

```
$request->flash();
```

php

También puedes usar los métodos `flashOnly` y `flashExcept` para enviar un subconjunto de datos de la solicitud a la sesión. Estos métodos son útiles para mantener información sensible tales como contraseñas fuera de la sesión:

```
$request->flashOnly(['username', 'email']);  
  
$request->flashExcept('password');
```

php

Enviando datos y redirigir

Ya que con frecuencia querrás enviar datos a la sesión y luego redirigir a la página anterior puedes encadenar datos a una redirección usando el método `withInput` :

```
return redirect('form')->withInput();  
  
return redirect('form')->withInput(  
    $request->except('password')  
)
```

php

Obteniendo datos antiguos

Para obtener los datos de la sesión anterior, usa el método `old` en la instancia `Request`. El método `old` extraerá los datos de la solicitud y `sesión` anterior:

```
$username = $request->old('username');
```

php

Laravel también proporciona un helper global `old`. Si estás mostrando datos antiguos dentro de una [plantilla Blade](#), es más conveniente usar el helper `old`. Si no existen datos antiguos para el campo dado, será devuelto `null`:

```
<input type="text" name="username" value="{{ old('username') }}">
```

php

Cookies

Obteniendo cookies de las solicitudes

Todos las cookies creados por el framework Laravel son encriptadas y firmadas con un código de autenticación, significa que serán consideradas no válidas si han sido cambiados por el cliente. Para obtener el valor de una cookie de la solicitud, usa el método `cookie` en una instancia de `Illuminate\Http\Request`:

```
$value = $request->cookie('name');
```

php

Alternativamente, puedes usar la clase facade `Cookie` para acceder a los valores de las cookies:

```
use Illuminate\Support\Facades\Cookie;  
  
$value = Cookie::get('name');
```

php

Adjuntando cookies a las respuestas

Puedes adjuntar una cookie a una instancia saliente de `Illuminate\Http\Response` usando el método `cookie`. Debes pasar el nombre, valor y el número de minutos en los cuales dicha cookie debería ser válida:

```
return response('Hello World')->cookie(  
    'name', 'value', $minutes  
)
```

php

El método `cookie` también acepta unos cuantos argumentos los cuales son usados con menos frecuencia. Generalmente, estos argumentos tienen el mismo propósito y significan lo mismo que los argumentos que deberían ser dados al método `setcookie` nativo de PHP:

```
return response('Hello World')->cookie(  
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly  
)
```

php

Alternativamente, puedes usar la clase facade `Cookie` para "encolar" cookies para adjuntar a la respuesta saliente de tu aplicación. El método `queue` acepta una instancia `Cookie` o los argumentos necesarios para crear una instancia `Cookie`. Estas cookies serán adjuntadas a la respuesta saliente antes de que sea enviada al navegador:

```
Cookie::queue(Cookie::make('name', 'value', $minutes));  
  
Cookie::queue('name', 'value', $minutes);
```

php

Generando instancias cookie

Si prefieres generar una instancia `Symfony\Component\HttpFoundation\Cookie` que pueda ser dada a una instancia de respuesta en un momento posterior, puedes usar el helper global `cookie`. Este cookie no será enviado de regreso al cliente a menos que sea adjuntado a una instancia de respuesta:

```
$cookie = cookie('name', 'value', $minutes);  
  
return response('Hello World')->cookie($cookie);
```

php

Archivos

Obteniendo archivos cargados

Puedes acceder los archivos cargados de 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`, la cual extiende la clase `SplFileInfo` de PHP y proporciona una variedad de métodos para interactuar con el archivo:

```
$file = $request->file('photo');  
  
$file = $request->photo;
```

php

Puedes determinar si un archivo está presente en la solicitud usando el método `hasFile` :

```
if ($request->hasFile('photo')) {  
    //  
}
```

php

Validando cargas exitosas

Además de verificar si el archivo está presente, puedes verificar que no ocurrieron problemas cargando el archivo por medio del método `isValid` :

```
if ($request->file('photo')->isValid()) {  
    //  
}
```

php

Rutas y extensiones de archivo

La clase `UploadedFile` también contiene métodos para acceder a la ruta completa del archivo y su extensión. El método `extension` intentará adivinar la extensión del archivo en base a su contenido. Esta extensión puede ser diferente de la extensión que fue suministrada por el cliente:

```
$path = $request->photo->path();  
  
$extension = $request->photo->extension();
```

php

Otros métodos de archivo

Hay una variedad de otros métodos disponibles en instancias `UploadedFile`. Revisa la [documentación de la API para la clase](#) para más información concerniente a estos métodos.

Almacenando archivos cargados

Para almacenar un archivo cargado, típicamente usarás uno de tus [sistemas de archivos](#) configurados. La clase `UploadedFile` tiene un método `store` el cual moverá un archivo cargado a uno de tus discos, el cual puede ser una ubicación de tu sistema de archivo local o incluso una ubicación de almacenamiento en la nube como Amazon S3.

El método `store` acepta la ruta donde el archivo debería ser almacenado relativa al directorio raíz configurado del sistema de archivo. Esta ruta no debería contener un nombre de archivo, ya que un ID único será generado automáticamente para servir como el nombre del archivo.

El método `store` acepta un segundo argumento opcional para el nombre del disco que debería ser usado para almacenar el archivo. El método devolverá la ruta relativa del archivo al directorio raíz del disco:

```
$path = $request->photo->store('images');  
  
$path = $request->photo->store('images', 's3');
```

php

Si no quieres que un nombre de archivo sea generado automáticamente, puedes usar el método `storeAs`, el cual acepta la ruta, el nombre de archivo y el nombre del disco como sus argumentos:

```
$path = $request->photo->storeAs('images', 'filename.jpg');  
  
$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

php

Configurando proxies de confianza

Al momento de administrar tus aplicaciones detrás de un balanceador de carga que finaliza los certificados TLS / SSL, puedes notar que algunas veces tu aplicación no genera enlaces HTTPS. Típicamente esto es debido a que el tráfico de tu aplicación está siendo dirigido desde tu balanceador de carga por el puerto 80 y no sabe que debería generar enlaces seguros.

Para solucionar esto, puedes usar el middleware `App\Http\Middleware\TrustProxies` que es incluido en tu aplicación de Laravel, el cual permite que rápidamente personalices los balanceadores de carga o proxies que deberían ser de confianza en tu aplicación. Tus proxies de confianza deberían ser listados como un arreglo en la propiedad `$proxies` de este middleware. Además de la configuración de los proxies de confianza, puedes configurar los encabezados que están siendo enviados por tu proxy con información sobre la solicitud original:

```
<?php  
  
namespace App\Http\Middleware;
```

php

```
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 headers that should be used to detect proxies.
     *
     * @var string
     */
    protected $headers = Request::HEADER_X_FORWARDED_ALL;
}
```

TIP

Si estás usando Balanceo de Carga Elástico AWS, tu valor `$headers` debe ser

`Request::HEADER_X_FORWARDED_AWS_ELB`. Para más información de las constantes que pueden ser usadas en la propiedad `$headers`, revisa la documentación de Symfony sobre [proxies de confianza](#).

Confiar en todos los proxies

Si estás usando Amazon AWS u otro proveedor de balanceador de carga de la "nube", no puedes saber las direcciones IP de tus平衡adores reales. En este caso, puedes usar `**` para confiar en todos los proxies:

```
/**
 * The trusted proxies for this application.
 *
 * @var array
```

php

```
 */
protected $proxies = '***';
```

Respuestas HTTP

- Creando respuestas
 - Adjuntando encabezados a las respuestas
 - Adjuntando cookies a las respuestas
 - Cookies & Encriptación
- Redirecciones
 - Redireccionando a rutas nombradas
 - Redireccionando a acciones de controlador
 - Redireccionando a dominios externos
 - Redireccionando con los datos de una sesión movida rápidamente
- Otros tipos de respuestas
 - Respuestas de vista
 - Respuestas JSON
 - Descargas de archivo
 - Respuestas de archivo
- Macros de respuesta

Creando respuestas

Cadenas & Arreglos

Todas las rutas y controladores deberían devolver una respuesta para ser enviada de regreso al navegador del usuario. Laravel proporciona diferentes formas de devolver respuestas. La respuesta más básica es devolver una cadena desde una ruta o controlador. El framework convertirá la cadena en una respuesta HTTP completa:

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

php

Además de devolver cadenas desde tus rutas y controladores, también puedes devolver arreglos. El framework convertirá automáticamente el arreglo en una respuesta JSON:

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

php

TIP

¿Sabías que también puedes devolver colecciones de Eloquent desde tus rutas o controladores? Estas serán convertidas automáticamente a JSON. ¡Inténtalo!

Objetos de respuesta

Típicamente, no sólo estarás devolviendo cadenas básicas o arreglos desde tus acciones de ruta. Además, estarás devolviendo instancias `Illuminate\Http\Response` completas o [vistas](#).

Devolver una instancia `Response` completa te permite personalizar el código de estado y los encabezados HTTP de la respuesta. Una instancia `Response` hereda desde la clase `Symfony\Component\HttpFoundation\Response`, la cual proporciona una variedad de métodos para construir respuestas HTTP:

```
Route::get('home', function () {
    return response('Hello World', 200)
        ->header('Content-Type', 'text/plain');
});
```

php

Adjuntando encabezados a las respuestas

Ten en cuenta que la mayoría de los métodos de respuestas son encadenables, permitiendo la construcción fluida de instancias de respuesta. Por ejemplo, puedes usar el método `header` para agregar una serie de encabezados para la respuesta antes de enviarla de regreso al usuario:

```
return response($content)
    ->header('Content-Type', $type)
    ->header('X-Header-One', 'Header Value')
    ->header('X-Header-Two', 'Header Value');
```

php

O, puedes usar el método `withHeaders` para especificar un arreglo de encabezados para que sean agregados a la respuesta:

```
return response($content)
    ->withHeaders([
        'Content-Type' => $type,
        'X-Header-One' => 'Header Value',
        'X-Header-Two' => 'Header Value',
    ]);
```

php

Middleware para control de cache

Laravel incluye un middleware `cache.headers`, el cual puede ser usado para rápidamente establecer el encabezado `Cache-control` para un grupo de rutas. Si `etag` está especificado en la lista de directivas, un hash MD5 del contenido de la respuesta será automáticamente establecido como identificador del ETag:

```
Route::middleware('cache.headers:public;max_age=2628000;etag')->group(function (
    Route::get('privacy', function () {
        // ...
    });
    Route::get('terms', function () {
        // ...
    });
));
```

Adjuntando cookies a las respuestas

El método `cookie` en las instancias de respuesta permite que adjunes fácilmente cookies a la respuesta. Por ejemplo, puedes usar el método `cookie` para generar una cookie y adjuntarla fluidamente a la instancia de respuesta, de la siguiente manera:

```
return response($content)
    ->header('Content-Type', $type)
    ->cookie('name', 'value', $minutes);
```

php

El método `cookie` también acepta unos cuantos argumentos los cuales son usados con menos frecuencia. Generalmente, estos argumentos tienen el mismo propósito y significado que los argumentos que serán dados al método nativo de PHP `setcookie`:

```
->cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

php

Alternativamente, puedes usar la clase facade `Cookie` para agregar cookies a la cola y adjuntarlas a la respuesta saliente de tu aplicación. El método `queue` acepta una instancia `Cookie` o los argumentos que se necesitan para crear una instancia `Cookie`. Estas cookies serán adjuntadas a la respuesta saliente antes de que sea enviada al navegador:

```
Cookie::queue(Cookie::make('name', 'value', $minutes));
Cookie::queue('name', 'value', $minutes);
```

php

Cookies & Encriptación

De forma predeterminada, todos los cookies generados por Laravel son encriptados y firmados de modo que no puedan ser modificados o leídos por el cliente. Si prefieres deshabilitar la encriptación para un subconjunto de cookies generados por tu aplicación, puedes usar la propiedad `$except` del middleware `App\Http\Middleware\EncryptCookies`, el cual es localizado en el directorio `app/Http/Middleware`:

```
/**
 * The names of the cookies that should not be encrypted.
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

php

Redirecciones

Las respuestas redireccionadas son instancias de la clase `Illuminate\Http\RedirectResponse` y contienen los encabezados apropiados que se necesitan para redireccionar al usuario a otra URL. Hay varias formas de generar una instancia `RedirectResponse`. El método más simple es usar el helper global `redirect`:

```
Route::get('dashboard', function () {
    return redirect('home/dashboard');
});
```

php

Algunas veces podrás querer redireccionar al usuario a su página previa, tal como cuando un formulario enviado no es válido. Puedes hacer eso usando la función helper global `back`. Ya que esta característica utiliza la `sesión`, asegurate de que la ruta llamando a la función `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();
});
```

php

Redireccionando a rutas nombradas

Cuando ejecutas el helper `redirect` sin parámetros, una instancia de `Illuminate\Routing\Redirector` es devuelta, permitiendo que ejecutes cualquier método en la instancia `Redirector`. Por ejemplo, para generar una `RedirectResponse` para una ruta nombrada, puedes usar el método `route`:

```
return redirect()->route('login');
```

php

Si tu ruta tiene parámetros, puedes pasarlos como segundo argumento del método `route`:

```
// For a route with the following URI: profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

php

Rellenando parámetros a través de modelos de Eloquent

Si estás redireccionando a una ruta con un parámetro "ID" que está siendo rellenado desde un modelo Eloquent, puedes pasar el modelo como tal. El ID será extraído automáticamente:

```
// For a route with the following URI: profile/{id} php

return redirect()->route('profile', [$user]);
```

Si prefieres personalizar el valor que es colocado en el parámetro de la ruta, deberías sobrescribir el método `getRouteKey` en tu modelo Eloquent:

```
/** php
 * Get the value of the model's route key.
 *
 * @return mixed
 */
public function getRouteKey()
{
    return $this->slug;
}
```

Redireccionando a acciones de controlador

También puedes generar redirecciones a [acciones de controlador](#). Para hacer eso, pasa el controlador y nombre de acción al método `action`. Recuerda, no necesitas especificar el espacio de nombres completo del controlador ya que el `RouteServiceProvider` de Laravel establecerá el espacio de nombres del controlador base:

```
return redirect()->action('HomeController@index'); php
```

Si tu ruta de controlador requiere parámetros, puedes pasarlos como segundo argumento del método `action`:

```
return redirect()->action(
    'UserController@profile', ['id' => 1] php
```

```
);
```

Redireccionando a dominios externos

Algunas veces puedes necesitar redireccionar a un dominio fuera de tu aplicación. Puedes hacer eso ejecutando el método `away`, el cual crea una instancia de `RedirectResponse` sin alguna codificación, validación o verificación de URL adicional:

```
return redirect()->away('https://www.google.com');
```

php

Redireccionando con datos de sesión

El redireccionamiento a una nueva URL y el envío de los datos de la sesión son hechos usualmente al mismo tiempo. Típicamente, esto es hecho después de ejecutar una acción exitosamente cuando mueves rápidamente un mensaje de éxito de la sesión. Por conveniencia, puedes crear una instancia `RedirectResponse` y mover rápidamente los datos de la sesión en un solo encadenamiento de método fluido:

```
Route::post('user/profile', function () {
    // Update the user's profile...

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

php

Después de que el usuario es redireccionado, puedes mostrar el mensaje enviado desde la sesión. Por ejemplo, usando la sintaxis de Blade:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

php

Otros tipos de respuesta

El helper `response` puede ser usado para generar otros tipos de instancias de respuesta. Cuando el helper `response` es ejecutado sin argumentos, una implementación del [contrato Illuminate\Contracts\Routing\ResponseFactory](#) es devuelta. Este contrato proporciona varios métodos útiles para generar respuestas.

Respuestas de vista

Si necesitas control sobre el estado y encabezados de la respuesta pero también necesitas devolver una [vista](#) como el contenido de la respuesta, deberías usar el método `view` :

```
return response()
    ->view('hello', $data, 200)
    ->header('Content-Type', $type);
```

php

Ciertamente, si no necesitas pasar un código de estado HTTP o encabezados personalizados, deberías usar la función helper global `view` .

Respuestas JSON

El método `json` establecerá automáticamente el encabezado `Content-Type` a `application/json` , al igual que convertirá el arreglo dado a JSON usando la función de PHP `json_encode` :

```
return response()->json([
    'name' => 'Abigail',
    'state' => 'CA'
]);
```

php

Si prefieres crear una respuesta JSONP, puedes usar el método `json` en combinación con el método `withCallback` :

```
return response()
    ->json(['name' => 'Abigail', 'state' => 'CA'])
    ->withCallback($request->input('callback'));
```

php

Descargas de archivo

El método `download` puede ser usado para generar una respuesta que fuerza al navegador del usuario a descargar el archivo a una ruta dada. El método `download` acepta un nombre de archivo como segundo argumento del método, el cual determinará el nombre del archivo que es visto por el usuario que esté descargando el archivo. Finalmente, puedes pasar un arreglo de encabezados HTTP como tercer argumento del método:

```
return response()->download($pathToFile);  
  
return response()->download($pathToFile, $name, $headers);  
  
return response()->download($pathToFile)->deleteFileAfterSend();
```

php

Nota

Symfony HttpFoundation, la cual administra las descargas de archivo, requiere que el archivo que esté siendo descargado tenga un nombre de archivo ASCII.

Descargas en streaming

Algunas veces puedes querer convertir la cadena de respuesta de una operación dada a una respuesta descargable sin tener que escribir los contenidos de la operación al disco. Puedes usar el método `streamDownload` en este escenario. Este método acepta un callback, un nombre de archivo y un arreglo opcional de encabezados como argumentos:

```
return response()->streamDownload(function () {  
    echo GitHub::api('repo')  
        ->contents()  
        ->readme('laravel', 'laravel')['contents'];  
}, 'laravel-readme.md');
```

php

Respuestas de archivo

El método `file` puede ser usado para mostrar un archivo, tal como una imagen o PDF, directamente en el navegador del usuario en lugar de iniciar una descarga. Este método acepta la ruta del archivo como su primer argumento y un arreglo de encabezados como segundo argumento:

```
return response()->file($pathToFile);

return response()->file($pathToFile, $headers);
```

php

Macros de respuesta

Si prefieres definir una respuesta personalizada que puedas volver a usar en múltiples rutas y controladores, puedes usar el método `macro` de la clase facade `Response`. Por ejemplo, desde un método `boot` del proveedor de servicio

```
<?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));
        });
    }
}
```

php

La función `macro` acepta un nombre como su primer argumento y una Closure como segundo. La Closure de la macro será ejecutada al momento de ejecutar el nombre de la macro desde una implementación `ResponseFactory` o el helper `response`:

```
return response()->caps('foo');
```

php

Vistas

- Creando vistas
- Pasando datos a las vistas
 - Compartiendo datos con todas las vistas
- View Composers

Creando vistas

TIP

Para buscar más información sobre ¿Cómo escribir plantillas de Blade? Revisa la documentación de Blade completa para comenzar.

Las vistas contienen el HTML servido por tu aplicación y separa la lógica de tu controlador/aplicación de la lógica de presentación. Las vistas son almacenadas en el directorio `resources/views`. Una vista sencilla podría lucir de esta forma:

```
<!-- View stored in resources/views/greeting.blade.php -->          php

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Ya que esta vista es almacenada en `resources/views/greeting.blade.php`, podemos devolverla usando el helper global `view`, de la siguiente forma:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

php

Como puedes ver, el primer argumento pasado al helper `view` corresponde al nombre del archivo de la vista en el directorio `resources/views`. El segundo argumento es un arreglo de datos que debería estar disponible para la vista. En este caso, estamos pasando la variable `name`, la cual es mostrada en la vista usando la [sintaxis de Blade](#).

Las vistas también pueden estar anidadas dentro de sub-directorios del directorio `resources/views`. La notación de "Punto" puede ser usada para referenciar vistas anidadas. Por ejemplo, si tu vista está almacenada en `resources/views/admin/profile.blade.php`, puedes hacer referencia a esta de la siguiente forma:

```
return view('admin.profile', $data);
```

php

Determinando si una vista existe

Si necesitas determinar si una vista existe, puedes usar la clase facade `View`. El método `exists` devolverá `true` si la vista existe:

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    //
}
```

php

Creando la primera vista disponible

Usando el método `first`, puedes crear la primera vista que existe en un arreglo de vistas dado. Esto es útil si tu aplicación o paquete permite que las vistas sean personalizadas o sobrescritas:

```
return view()->first(['custom.admin', 'admin'], $data);
```

php

También puedes ejecutar este método por medio de la clase facade `View`:

```
use Illuminate\Support\Facades\View;  
  
return View::first(['custom.admin', 'admin'], $data);
```

php

Pasando datos a las vistas

Como viste en los ejemplos previos, puedes pasar un arreglo de datos a las vistas:

```
return view('greetings', ['name' => 'Victoria']);
```

php

Al momento de pasar información de esta manera, los datos deberían ser un arreglo con pares clave / valor. Dentro de tu vista, entonces puedes acceder a cada valor usando su clave correspondiente, tal como `<?php echo $key; ?>`. Como una alternativa a pasar un arreglo completo de datos a la función helper `view`, puedes usar el método `with` para agregar partes individuales de datos a la vista:

```
return view('greeting')->with('name', 'Victoria');
```

php

Compartiendo datos con todas las vistas

Ocasionalmente, puedes necesitar compartir una pieza de datos con todas las vistas que son renderizadas por tu aplicación. Puedes hacer eso usando el método `share` de la clase facade `View`. Típicamente, deberías colocar las ejecuciones a `share` dentro del método `boot` de un proveedor de servicio. Eres libre de agregarlos al `AppServiceProvider` o generar un proveedor de servicio diferente para alojarlos:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\View;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Register any application services.
```

php

```
* @return void
*/
public function register()
{
    //
}

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    View::share('key', 'value');
}
}
```

View Composers

Los view composers son funciones de retorno o métodos de clase que son ejecutados cuando una vista es renderizada. Si tienes datos que quieres que estén enlazados a una vista cada vez que la vista es renderizada, un view composer puede ayudarte a organizar esa lógica dentro de una sola ubicación.

Para este ejemplo, vamos a registrar los View Composers dentro de un [proveedor de servicio](#). Usaremos la clase facade `View` para acceder a la implementación de contrato

`Illuminate\Contracts\View\Factory` subyacente. Recuerda, Laravel no incluye un directorio predeterminado para los View Composers. Eres libre de organizarlos del modo que deseas. Por ejemplo, podrías crear un directorio `app/Http/View/Composers` :

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ViewServiceProvider extends ServiceProvider
{
```

```

    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        View::composer(
            'profile', 'App\Http\View\Composers\ProfileComposer'
        );

        // Using Closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }
}

```

Nota

Recuerda, si creas un nuevo proveedor de servicio para contener tus registros de View Composers, necesitarás agregar el proveedor de servicio al arreglo `providers` en el archivo de configuración `config/app.php`.

Ahora que hemos registrado el Composer, el método `ProfileComposer@compose` será ejecutado cada vez que la vista `profile` esté siendo renderizada. Así que, vamos a definir la clase composer:

```

<?php
namespace App\Http\View\Composers;

```

```
use App\Repositories\UserRepository;
use Illuminate\View\View;

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, el método `compose` del Composer es ejecutado con la instancia `Illuminate\View\View`. Puedes usar el método `with` para enlazar datos a la vista.

TIP

Todos los View Composers son resueltos por medio del [contenedor de servicio](#), de modo que puedes colocar la referencia a cualquiera de las dependencias que necesites dentro de un constructor del Composer.

Adjuntando un composer a múltiples vistas

Puedes adjuntar un View Composer a múltiples vistas de una vez al pasar un arreglo de vistas como primer argumento del método `composer` :

```
View::composer(  
    ['profile', 'dashboard'],  
    'App\Http\View\Composers\MyViewComposer'  
)
```

php

El método `composer` también acepta el carácter `*` como un comodín, permitiendo que adjunes un Composer a todas las vistas:

```
View::composer('*', function ($view) {  
    //  
});
```

php

View Creators

View Creators (creadores de vistas) son muy similares a los View Composers; sin embargo, son ejecutados inmediatamente después de que la vista sea instanciada en lugar de esperar hasta que la vista sea renderizada. Para registrar un View Creator, usa el método `creator` :

```
View::creator('profile', 'App\Http\View\Creators\ProfileCreator');
```

php

Generación de URLs

- Introducción
- Fundamentos
 - Generando URLs básicas
 - Accediendo la URL actual
- URLs para rutas nombradas
 - URLs firmadas
- URLs para acciones de controlador
- Valores predeterminados

Introducción

Laravel proporciona varios helpers para asistirte en la generación de URLs para tu aplicación. Éstos son útiles principalmente al momento de construir enlaces en tus plantillas y respuestas de API, o al momento de generar respuestas redireccionadas a otra parte de tu aplicación.

Fundamentos

Generando URLs básicas

El helper `url` puede ser usado para generar URLs arbitrarias en tu aplicación. La URL generada utilizará automáticamente el esquema (HTTP o HTTPS) y el host de la solicitud actual:

```
$post = App\Post::find(1);  
  
echo url("/posts/{$post->id}");  
  
// http://example.com/posts/1
```

Accediendo la URL actual

Si ninguna ruta es proporcionada al helper `url`, una instancia

`Illuminate\Routing\UrlGenerator` es devuelta, permitiéndote que accedas información sobre la URL actual:

```
// Obtener la URL actual sin la cadena de consulta...
echo url()->current();

// Obtener la URL actual incluyendo la cadena de consulta...
echo url()->full();

// Obtener la URL completa de la solicitud anterior...
echo url()->previous();
```

php

Cada uno de estos métodos también puede ser accedido por medio del facade `URL`:

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

php

URLs para rutas nombradas

El helper `route` puede ser usado para generar URLs para rutas nombradas. Las rutas nombradas permiten generar URLs sin estar acopladas a la URL real definida en la ruta. Por lo tanto, si la URL de la ruta cambia, no es necesario realizar cambios en las llamadas a la función `route`. Por ejemplo, imagina que tu aplicación contiene una ruta definida de la siguiente forma:

```
Route::get('/post/{post}', function () {
    //
})->name('post.show');
```

php

Para generar una URL a esta ruta, puedes usar el helper `route` así:

```
echo route('post.show', ['post' => 1]);

// http://example.com/post/1
```

php

Con frecuencia estarás generando URLs usando la clave primaria de modelos de Eloquent. Por esta razón, puedes pasar modelos de Eloquent como valores de parámetros. El helper `route` extraerá automáticamente la clave primaria del modelo:

```
echo route('post.show', ['post' => $post]);
```

php

El helper `route` también se puede usar para generar URL para rutas con múltiples parámetros:

```
Route::get('/post/{post}/comment/{comment}', function () {
    //
})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]);

// http://example.com/post/1/comment/3
```

php

URLs firmadas

Laravel te permite crear fácilmente URLs "firmadas" para rutas nombradas. Estas URLs tienen un hash de "firma" añadido a la cadena de solicitud que le permite a Laravel verificar que la URL no haya sido modificada desde que fue creada. Las URLs firmadas son especialmente útiles para rutas que están disponibles públicamente pero necesitan una capa de protección contra la manipulación de URLs.

Por ejemplo, puedes usar URLs firmadas para implementar un enlace público de "anular suscripción" que es enviado por correo electrónico a tus clientes. Para crear una URL firmada para una ruta nombrada, usa el método `signedRoute` del facade `URL`:

```
use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);
```

php

Si te gustaría generar una ruta firmada temporal que expira, puedes usar el método `temporarySignedRoute`:

```
use Illuminate\Support\Facades\URL;
```

php

```
return URL::temporarySignedRoute(
    'unsubscribe', now()->addMinutes(30), ['user' => 1]
);
```

Validando solicitudes a rutas firmadas

Para verificar que una solicitud entrante tiene una firma válida, debes llamar al método `hasValidSignature` en el `Request` entrante:

```
use Illuminate\Http\Request;                                         php

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (! $request->hasValidSignature()) {
        abort(401);
    }

    // ...
})->name('unsubscribe');
```

Alternativamente, puedes asignar el middleware

`Illuminate\Routing\Middleware\ValidateSignature` a la ruta. Si aún no está presente, debes asignar una clave a este middleware en el arreglo `routeMiddleware` de tu kernel HTTP:

```
/**                                                 php
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
];
```

Una vez que has registrado el middleware en tu kernel, puedes adjuntarlo a una ruta. Si la solicitud entrante no tiene una firma válida, el middleware automáticamente retornará una respuesta de error

`403`:

```
Route::post('/unsubscribe/{user}', function (Request $request) {  
    // ...  
})->name('unsubscribe')->middleware('signed');
```

php

URLs para acciones de controlador

La función `action` genera una URL para la acción de controlador dada. No necesitarás pasar el espacio de nombre completo del controlador. En lugar de eso, pasa el nombre de clase del controlador relativo al espacio de nombre `App\Http\Controllers`:

```
$url = action('HomeController@index');
```

php

También puedes hacer referencia a acciones con una sintaxis de arreglo “callable”:

```
use App\Http\Controllers\HomeController;  
  
$url = action([HomeController::class, 'index']);
```

php

Si el método del controlador acepta parámetros de ruta, puedes pasarlos como segundo argumento de la función:

```
$url = action('UserController@profile', ['id' => 1]);
```

php

Valores predeterminados

Para algunas aplicaciones, puedes querer especificar valores predeterminados para toda la solicitud en los parámetros de ciertas URL. Por ejemplo, imagina que muchas de tus rutas definen un parámetro

`{locale}` :

```
Route::get('/{locale}/posts', function () {  
    //  
})->name('post.index');
```

php

Es complicado pasar siempre el parámetro `locale` cada vez que ejecutas el helper `route`. Así, puedes usar el método `URL::defaults` para definir un valor predeterminado para este parámetro que siempre será aplicado durante la solicitud actual. Puedes querer ejecutar este método desde un middleware de ruta de modo que tengas acceso a la solicitud 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 predeterminado para el parámetro `locale` ha sido establecido, ya no estás obligado a pasar su valor al momento de generar URLs por medio del helper `route`.

Sesión HTTP

- [Introducción](#)
 - [Configuración](#)
 - [Prerequisitos de manejador](#)
- [Usando la sesión](#)

- Obteniendo datos
- Almacenando datos
- Datos instantáneos
- Eliminando datos
- Regenerando el ID de la sesión
- Agregando manejadores de sesión personalizada
 - Implementando el manejador
 - Registrando el manejador

Introducción

Ya que las aplicaciones manejadas por HTTP son sin estado, las sesiones proporcionan una forma de almacenar información sobre el usuario a través de múltiples solicitudes. Laravel viene con una variedad de backends de sesión que son accedidos a través de una expresiva API unificada. El soporte para los backends populares tales como [Memcached](#), [Redis](#) y bases de datos es incluido de forma predeterminada.

Configuración

El archivo de configuración de sesión es almacenado en `config/session.php`. Asegúrate de revisar las opciones disponibles para ti en este archivo. De forma predeterminada, Laravel está configurado para usar el manejador de sesión `cookie`, el cual trabajará bien con muchas aplicaciones.

La opción de configuración `driver` de la sesión define donde los datos de la sesión serán almacenados para cada solicitud. Laravel viene con varios excelentes manejadores de forma predeterminada.

- `file` - las sesiones son almacenadas en `storage/framework/sessions`.
- `cookie` - las sesiones son almacenadas en cookies encriptados seguros.
- `database` - las sesiones son almacenadas en una base de datos relacional.
- `memcached` / `redis` - las sesiones son almacenadas en rápidos almacenes basados en cache.
- `array` - las sesiones son almacenadas en un arreglo de PHP y no serán guardadas de forma permanente.

TIP

El driver array es usado durante las pruebas y previene que los datos almacenados en la sesión sean guardados de forma permanente.

Prerequisitos del driver

Base de datos

Al momento de usar el driver de sesión `database`, necesitarás crear una tabla para contener los elementos de la sesión. A continuación se muestra una declaración de `Schema` de ejemplo para la 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');  
});
```

php

Puedes usar el comando Artisan `session:table` para generar esta migración:

```
php artisan session:table  
  
php artisan migrate
```

php

Redis

Antes de usar sesiones de Redis con Laravel, necesitarás instalar ya sea la extensión PhpRedis de PHP mediante PECL o instalar el paquete `predis/predis` (~1.0) mediante Composer. Para más información sobre cómo configurar Redis, [consulta su página en la documentación](#).

Tip

En el archivo de configuración `session`, la opción `connection` puede ser usada para especificar que conexión de redis es usada por la sesión.

Usando la sesión

Obteniendo datos

Hay dos formas principales de trabajar con datos de sesión en Laravel: el helper global `session` y por medio de una instancia `Request`. Primero, vamos a echar un vistazo al acceder a la sesión por medio de una instancia `Request`, la cual puede ser referenciada en un método de controlador. Recuerda, las dependencias de métodos de controlador son inyectadas automáticamente por medio del [contenedor de servicio](#) de Laravel:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use Illuminate\Http\Request;  
  
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');  
  
        //  
    }  
}
```

php

Cuando obtienes un elemento de la sesión, también puedes pasar un valor predeterminado como segundo argumento del método `get`. Este valor predeterminado será devuelto si la clave especificada no existe en la sesión. Si pasas una `Closure` como el valor predeterminado del método `get` y la clave solicitada no existe, la `Closure` será ejecutada y su resultado devuelto:

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

php

Helper global de sesión

También puedes usar la función global de PHP `session` para obtener y almacenar datos en la sesión. Cuando el helper `session` es ejecutado con un solo argumento de cadena, devolverá el valor de esa clave de sesión. Cuando el helper es ejecutado con una arreglo de pares clave / valor, esos valores serán almacenados 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']);
});
```

php

TIP

Hay una pequeña diferencia práctica entre usar la sesión por medio de una instancia de solicitud HTTP contra usar el helper global `session`. Ambos métodos pueden ser [probados](#) por medio del método `assertSessionHas`, el cual está disponible en todos tus casos de prueba.

Obteniendo todos los datos de sesión

Si prefieres obtener todos los datos en la sesión, puedes usar el método `all` :

```
$data = $request->session()->all();
```

php

Determinando si un elemento existe en la sesión

Para determinar si un elemento está presente en la sesión, puedes usar el método `has`. El método `has` devuelve `true` si el valor está presente y no es `null`:

```
if ($request->session()->has('users')) {  
    //  
}
```

php

Para determinar si un elemento está presente en la sesión, incluso si su valor es `null`, puedes usar el método `exists`. El método `exists` devuelve `true` si el valor está presente:

```
if ($request->session()->exists('users')) {  
    //  
}
```

php

Almacenando datos

Para almacenar datos en la sesión, típicamente usarás el método `put` o el helper `session`:

```
// Via a request instance...  
$request->session()->put('key', 'value');  
  
// Via the global helper...  
session(['key' => 'value']);
```

php

Agregar a valores del arreglo de sesión

El método `push` puede ser usado para agregar un nuevo valor a un valor de sesión que está en un arreglo. Por ejemplo, si la clave `user.teams` contiene un arreglo de nombres de equipo, puedes agregar un nuevo valor al arreglo de la siguiente forma:

```
$request->session()->push('user.teams', 'developers');
```

php

Obteniendo y eliminando un elemento

El método `pull` obtendrá y eliminará un elemento de la sesión en una sola instrucción:

```
$value = $request->session()->pull('key', 'default');
```

php

Datos instantáneos

Algunas veces puedes querer almacenar varios elementos en la sesión para la próxima solicitud. Puedes hacer eso usando el método `flash`. Los datos almacenados en la sesión usando este método estarán disponibles de forma inmediata así como también en la solicitud HTTP posterior. Luego de la petición HTTP posterior, los datos instantáneos serán eliminados. Los datos instantáneos son principalmente útiles para mensajes de estado con vida corta:

```
$request->session()->flash('status', 'Task was successful!');
```

php

Si necesitas mantener tus datos instantáneos alrededor para varias solicitudes, puedes usar el método `reflash`, el cuál mantendrá todos los datos instantáneos para una solicitud adicional. Si solamente necesitas mantener datos instantáneos específicos, puedes usar el método `keep`:

```
$request->session()->reflash();  
  
$request->session()->keep(['username', 'email']);
```

php

Eliminando datos

El método `forget` removerá una porción de datos de la sesión. Si prefieres remover todos los datos de la sesión, puedes usar el método `flush`:

```
// Forget a single key...  
$request->session()->forget('key');  
  
// Forget multiple keys...  
$request->session()->forget(['key1', 'key2']);  
  
$request->session()->flush();
```

php

Regenerando el ID de la sesión

Regenerar el ID de la sesión es hecho frecuentemente con el propósito de prevenir que usuarios maliciosos exploten un ataque de [fijación de sesión](#) en tu aplicación.

Laravel regenera automáticamente el ID de la sesión durante la autenticación si estás usando el [LoginController](#) integrado; sin embargo, si necesitas regenerar manualmente el ID de la sesión, puedes usar el método [regenerate](#).

```
$request->session()->regenerate();
```

php

Agregando manejadores de sesión personalizados

Implementando el manejador

Tu manejador de sesión personalizado debería implementar la interface [SessionHandlerInterface](#). Esta interface contiene justo unos cuantos métodos que necesitamos implementar. Una implementación MongoDB truncada luce de forma similar a lo siguiente:

```
<?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) {}  
}
```

php

TIP

Laravel no viene con un directorio para contener tus extensiones. Eres libre de colocarlos en cualquier parte que quieras. En este ejemplo, hemos creado un directorio [Extensions](#) para alojar el manejador [MongoSessionHandler](#).

Ya que el propósito de estos métodos no es entendible rápidamente y sin dificultad, vamos a cubrir rápidamente lo que cada uno de estos métodos hace:

- El método `open` típicamente sería usado en sistemas de almacenamiento de sesión basada en archivo. Ya que Laravel viene con un manejador de sesión `file`, casi nunca necesitarás poner cualquier cosa en este método. Puedes dejarlo como un stub vacío. Es una característica de diseño de interface pobre (lo que discutiremos más tarde) que PHP nos obligue a implementar este método.
- El método `close`, como el método `open`, también puede ser descartado. Para la mayoría de los drivers, no es necesario.
- El método `read` debería devolver la versión de cadena de la sesión de datos asociada con la `$sessionId` dada. No hay necesidad de hacer alguna serialización u otra codificación al momento de obtener o almacenar los datos de la sesión en tu manejador, ya que Laravel ejecutará la serialización por ti.
- El método `write` debería escribir la cadena `$data` asociada dada con la `$sessionId` para algún sistema de almacenamiento persistente, tal como MongoDB, Dynamo, etc. Otra vez, no deberías ejecutar alguna serialización - Laravel ya ha manejado esto por ti.
- El método `destroy` debería remover los datos asociados con la `$sessionId` desde el almacenamiento persistente.
- El método `gc` debería destruir todos los datos de la sesión que son más viejos que el `$lifetime` dado, el cual es una marca de tiempo UNIX. Para los sistemas que se auto-expiran como Memcached y Redis, este método puede ser dejado vacío.

Registrando el manejador

Una vez que tu manejador ha sido implementado, estás listo para registrarlo en el framework. Para agregar manejadores adicionales para el backend de sesión de Laravel, puedes usar el método `extend` del método en la `Session facade`. Deberías ejecutar el método `extend` desde el método `boot` de un `proveedor de servicio`. Puedes hacer esto desde el existente `AppServiceProvider` o crear un nuevo proveedor de servicios completo:

```
<?php  
  
namespace App\Providers;  
  
use App\Extensions\MongoSessionHandler;  
use Illuminate\Support\Facades\Session;  
use Illuminate\Support\ServiceProvider;
```

```
class SessionServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return implementation of SessionHandlerInterface...
            return new MongoSessionHandler();
        });
    }
}
```

Una vez que el manejador de la sesión ha sido registrado, puedes usar el manejador `mongo` en tu archivo de configuración `config/session.php`.

Validación

- [Introducción](#)
- [Inicio rápido de validación](#)

- Definiendo las rutas
- Creando el controlador
- Escribiendo la lógica de validación
- Mostrando los errores de validación
- Una observación sobre los campos opcionales
- Validación de solicitudes de formulario
 - Creando solicitudes de formulario
 - Autorizando solicitudes de formulario
 - Personalizando los mensajes de error
 - Personalizando los atributos de validación
 - Preparar datos para validación
- Creando validadores manualmente
 - Redirección automática
 - Paquetes de errores con nombres
 - Hook de validación posterior
- Trabajando con los mensajes de error
 - Personalizar los mensajes de error
- Reglas de validación disponibles
- Agregando reglas condicionalmente
- Validando arreglos
- Personalizar las reglas de validación
 - Usando objetos de regla
 - Usando closures
 - Usando extensiones
 - Extensiones implícitas

Introducción

Laravel proporciona varios enfoques diferentes para validar los datos entrantes de tu aplicación. De forma predeterminada, la clase base del controlador de Laravel usa una característica `ValidatesRequests` la cual proporciona un método conveniente para validar la solicitud HTTP entrante con una variedad de poderosas reglas de validación.

Inicio rápido de validación

Para aprender sobre las poderosas características de validación de Laravel, vamos a observar un ejemplo completo validando un formulario y mostrando los mensajes de error devueltos al usuario.

Definiendo las rutas

Primero, vamos a asumir que tenemos las rutas siguientes definidas en nuestro archivo

`routes/web.php` :

```
Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@store');
```

php

La ruta `GET` mostrará un formulario al usuario para crear un nuevo post de blog, mientras que la ruta `POST` guardará el nuevo post de blog en la base de datos.

Creando el controlador

Luego, vamos a observar un simple controlador que maneja estas rutas. Dejaremos el método `store` vacío por ahora:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     *
```

php

```
* 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

Ahora estamos listos para completar nuestro método `store` con la lógica para validar el nuevo post de blog. Para hacer esto, usaremos el método `validate` proporcionado por el objeto `Illuminate\Http\Request`. Si las reglas de validación pasan, tu código continuará su ejecución normalmente; sin embargo, si la validación falla, se arrojará una excepción y la respuesta de error apropiada será devuelta automáticamente al usuario. En el caso de una solicitud HTTP tradicional, se generará una respuesta de redirección, mientras una respuesta JSON será enviada para las solicitudes AJAX.

Para lograr una mejor comprensión del método `validate`, regresemos al 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 puedes ver, pasamos las reglas de validación deseadas dentro del método `validate`. Otra vez, si la validación falla, se generará la respuesta apropiada. Si la validación pasa, nuestro controlador continuará la ejecución normalmente.

Deteniendo en la primera falla de validación

Algunas veces puede que desees detener la ejecución de las reglas de validación sobre un atributo después de la primera falla de validación. Para hacer eso, asigna la regla `bail` al atributo:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

php

En este ejemplo, si la regla `unique` del atributo `title` falla, la regla `max` no será verificada. Las reglas serán validadas en el orden que sean asignadas.

Una obsevación sobre los atributos anidados

Si tu solicitud HTTP contiene parámetros "anidados", puedes especificarlos en tus reglas de validación usando la sintaxis de "punto":

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

php

Mostrando los errores de validación

¿Qué sucede si los parámetros de solicitud entrantes no pasan las reglas de validación dados? Cómo mencionamos anteriormente, Laravel redirigirá al usuario de regreso a su ubicación previa. En adición, todos los errores de validación serán automáticamente movidos instantáneamente a la sesión.

De nuevo, observa que no tuvimos que enlazar explícitamente los mensajes de error con la vista en nuestra ruta `GET`. Esto es porque Laravel revisará los errores en la sesión de datos y los enlazará automáticamente a la vista si están disponibles. La variable `$errors` será una instancia de

`Illuminate\Support\MessageBag`. Para mayor información sobre cómo trabajar con este objeto, revisa su documentación.

TIP

La variable `$errors` es enlazada a la vista por el middleware `Illuminate\View\Middleware\ShareErrorsFromSession`, el cual es proporcionado por el grupo de middleware `web`. **Cuando este middleware se aplique una variable `$errors` siempre estará disponible en tus vistas**, permitiendo que asumas convenientemente que la variable `$errors` está definida siempre y puede ser usada con seguridad.

Así, en nuestro ejemplo, el usuario será redirigido al método `create` de nuestro controlador cuando la validación falle, permitiéndonos que muestre los mensajes de error en la vista:

```
<!-- /resources/views/post/create.blade.php --> 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 -->
```

Directiva `@error`

También puedes usar la directiva `@error` de `Blade` para rápidamente comprobar si los mensajes de error de validación existen para un atributo dado. Dentro de una directiva `@error`, puedes mostrar la variable `$message` para mostrar el mensaje de error:

```
<!-- /resources/views/post/create.blade.php --> php

<label for="title">Post Title</label>
```

```
<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Una observación sobre los campos opcionales

De forma predeterminada, Laravel incluye los middleware `TrimStrings` y

`ConvertEmptyStringsToNull` en la pila global de middleware de tu aplicación. Estos middleware son listados en la pila por la clase `App\Http\Kernel`. Debido a esto, con frecuencia necesitarás marcar tus campos "opcionales" de solicitud como `nullable` si no quieres que el validador considere los valores `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 que sea o `null` o una representación de fecha válida. Si el modificador `nullable` no es agregado a la definición de la regla, el validador consideraría el `null` como una fecha no válida.

Solicitudes AJAX y validación

En este ejemplo, usamos un formulario tradicional para enviar datos a la aplicación. Sin embargo, muchas aplicaciones usan solicitudes AJAX. Al momento de usar el método `validate` durante una solicitud AJAX, Laravel no generará una respuesta de redirección. En su lugar, Laravel genera una respuesta JSON conteniendo todos los errores de validación. Esta respuesta JSON será enviada con un código de estado HTTP 422.

Validación de solicitud de formulario

Creando solicitudes de formulario (form request)

Para escenarios de validación más complejos, puede que desees crear una "solicitud de formulario (form request)". Las Form Request son clases de solicitud personalizadas que contienen la lógica de validación.

Para crear una clase de Form Request, usa el comando de CLI de Artisan `make:request` :

```
php artisan make:request StoreBlogPost
```

php

La clase generada será colocada en el directorio `app/Http/Requests`. Si este directorio no existe, será creado cuando ejecutes el comando `make:request`. Agreguemos unas cuantas 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',  
    ];  
}
```

php

TIP

Puedes declarar el tipo de cualquier dependencia que necesites dentro de la firma del método `rules`. Se resolverán automáticamente a través del [contenedor de servicio](#) de Laravel.

Así que, ¿Cómo son evaluadas las reglas de validación? Todo lo que necesitas hacer es poner la referencia de la solicitud en tu método de controlador. La Form Request entrante es validada antes de que el método de controlador sea ejecutado, significa que no necesitas complicar tu controlador con ninguna lógica de validación:

```
/**  
 * Store the incoming blog post.  
 *  
 * @param StoreBlogPost $request  
 * @return Response  
 */  
public function store(StoreBlogPost $request)
```

php

```
{  
    // The incoming request is valid...  
  
    // Retrieve the validated input data...  
    $validated = $request->validated();  
}
```

Si la validación falla, una respuesta de redirección será generada para enviar al usuario de vuelta a su ubicación previa. Los errores también serán movidos instantáneamente a la sesión de modo que estén disponibles para mostrarlos. Si la solicitud fuese una solicitud AJAX, una respuesta HTTP con un código de estado 422 será devuelta al usuario incluyendo una representación JSON de los errores de validación.

Agregando hooks posteriores a solicitudes de formularios

Si prefieres agregar un hook "posterior" a una Form Request, puedes usar el método `withValidator`. Este método recibe el validador completamente construido, permitiendo que ejecutes cualquiera de sus métodos antes de que las reglas de validación sean evaluadas realmente:

```
/* *  
 * 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 file');  
        }  
    });  
}
```

Autorizando solicitudes de formularios

La clase Form Request también contiene un método `authorize`. Dentro de este método, puedes verificar si el usuario autenticado realmente tiene la autoridad para actualizar un recurso dado. Por ejemplo, puedes determinar si a un usuario le pertenece el comentario del blog que está intentando actualizar

php

```
/**  
 * 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 todas las form request extienden de la clase solicitud base (Request) de Laravel, podemos usar el método `user` para acceder al usuario actualmente autenticado. También observa la llamada al método `route` en el ejemplo anterior. Este método te otorga acceso a los parámetros de URI definidos en la ruta que es ejecutada, tal como el parámetro `{comment}` en el ejemplo de abajo:

php

```
Route::post('comment/{comment}');
```

Si el método `authorize` devuelve `false`, una respuesta HTTP con un código de estado 403 será devuelta automáticamente y tu método de controlador no se ejecutará.

Si planeas tener la lógica de autorización en otra parte de tu aplicación, devuelve `true` desde el método `authorize`:

php

```
/**  
 * Determine if the user is authorized to make this request.  
 *  
 * @return bool  
 */  
public function authorize()  
{  
    return true;  
}
```

TIP

Puedes declarar el tipo de cualquier dependencia que necesites dentro de la firma del método `authorize`. Se resolverán automáticamente a través de Laravel [contenedor de servicio](#).

Personalizando los mensajes de error

Puedes personalizar los mensajes de error usados por la solicitud de formulario al sobrescribir el método `messages`. Este método debería devolver un arreglo de atributos / pares de regla y sus correspondientes mensajes de error:

```
/** php
 * 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',
    ];
}
```

Personalizando los atributos de validación

Si desea que la parte `:attribute` de su mensaje de validación se reemplace con un nombre de atributo personalizado, puede especificar los nombres personalizados sobrescribiendo el método `attributes`. Este método debería devolver un arreglo de pares de atributo / nombre:

```
/** php
 * Get custom attributes for validator errors.
 *
 * @return array
 */
public function attributes()
{
    return [
        'email' => 'email address',
    ];
}
```

Preparar datos para validación

Si necesitas limpiar datos de la petición antes de aplicar tus reglas de validación, puedes usar el método `prepareForValidation` :

```
use Illuminate\Support\Str;  
  
/**  
 * Prepare the data for validation.  
 *  
 * @return void  
 */  
protected function prepareForValidation()  
{  
    $this->merge([  
        'slug' => Str::slug($this->slug),  
    ]);  
}
```

php

Creando validadores manualmente

Si no quieres usar el método `messages` en la solicitud, puedes crear una instancia de validador manualmente usando la clase `facade Validator`. El método `make` en la clase facade genera una nueva instancia del validador:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Validator;  
  
class PostController extends Controller  
{  
    /**  
     * Store a new blog post.  
     *  
     * @param Request $request  
    }
```

php

```

    * @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 bajo validación. El segundo argumento son las reglas de validación que deberían ser aplicadas a los datos.

Después de verificar si la validación de solicitud falló, puedes usar el método `withErrors` para mover instantáneamente los mensajes de error a la sesión. Al momento de usar este método, la variable `$errors` será compartida automáticamente con tus vistas después de la redirección, permitiendo que los muestres de vuelta al usuario. El método `withErrors` acepta un validador, un `MessageBag`, o un `array` de PHP.

Redirección automática

Si prefieres crear manualmente una instancia del validador pero aún tomar ventaja de la redirección automática ofrecida por el método `validate` de la solicitud, puedes ejecutar el método `validate` en una instancia de validador existente. Si la validación falla, el usuario automáticamente será redirigido o, en el caso de una solicitud AJAX, le será devuelta una respuesta JSON:

```

Validator::make($request->all(), [
  'title' => 'required|unique:posts|max:255',
  'body' => 'required',
])->validate();

```

php

Paquetes de errores con nombres

Si tienes múltiples formularios en una sola página, puede que desees nombrar el `MessageBag` de errores, permitiendo que obtengas los mensajes de error para un formulario específico. Pasa un nombre como segundo argumento a `withErrors` :

```
return redirect('register')
    ->withErrors($validator, 'login');
```

php

Entonces puedes acceder la instancia de `MessageBag` nombrada de la variable `$errors` :

```
{{ $errors->login->first('email') }}
```

php

Hook de validación posterior

El validador también permite que adjunes funciones de retorno para que sean ejecutadas después que se complete la validación. Esto permite que ejecutes fácilmente validación adicional e incluso agregar más mensajes de error a la colección de mensajes. Para empezar, usa el método `after` en una instancia de validador:

```
$validator = Validator::make(...);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add('field', 'Something is wrong with this file');
    }
});

if ($validator->fails()) {
    //
}
```

php

Trabajando con los mensajes de error

Después de ejecutar el método `errors` en una instancia `Validator`, recibirás una instancia `Illuminate\Support\MessageBag`, la cual tiene una variedad de métodos convenientes para

trabajar con los mensajes de error. La variable `$errors` que se hace disponible automáticamente para todas las vistas también es una instancia de la clase `MessageBag`.

Obteniendo el primer mensaje de error para un campo

Para obtener el primer mensaje de error para un campo dado, usa el método `first`:

```
$errors = $validator->errors();  
  
echo $errors->first('email');
```

php

Obteniendo todos los mensajes de error para un campo

Si necesitas obtener un arreglo de todos los mensajes para un campo dado, usa el método `get`:

```
foreach ($errors->get('email') as $message) {  
    //  
}
```

php

Si estás validando un campo de formulario de arreglo, puedes obtener todos los mensajes para cada uno de los elementos de arreglo usando el carácter `*`:

```
foreach ($errors->get('attachments.*') as $message) {  
    //  
}
```

php

Obteniendo todos los mensajes de error para todos los campos

Para obtener un arreglo de todos los mensajes para todos los campos, usa el método `all`:

```
foreach ($errors->all() as $message) {  
    //  
}
```

php

Determinando si existen mensajes para un campo

El método `has` puede ser usado para determinar si existe algún mensaje de error para un campo dado:

```
if ($errors->has('email')) {  
    //  
}  
php
```

Mensajes de error personalizados

Si es necesario, puedes usar mensajes de error personalizados en vez de los predeterminados. Hay varias formas para especificar mensajes personalizados. Primero, puedes pasar los mensajes personalizados como tercer argumento al método `Validator::make`:

```
$messages = [  
    'required' => 'The :attribute field is required.',  
];  
  
$validator = Validator::make($input, $rules, $messages);  
php
```

En este ejemplo, el marcador `:attribute` será reemplazado por el nombre real del campo bajo validación. También puedes utilizar otros marcadores en 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',  
];  
php
```

Especificando un mensaje personalizado para un atributo dado

Algunas veces puedes querer especificar un mensaje de error personalizado sólo para un campo específico. Puedes hacer eso usando notación de "punto". Especifica el nombre del atributo al principio, seguido por la regla:

```
$messages = [  
    'email.required' => 'We need to know your e-mail address!',  
];  
php
```

Especificando mensajes personalizados en archivos por idiomas

En muchos casos, probablemente especificarás tus mensajes personalizados en un archivo de idioma en lugar de pasarlo directamente al `Validator`. Para hacer eso, agrega tus mensajes al arreglo `custom` en el archivo de idioma `resources/lang/xx/validation.php`:

```
'custom' => [
    'email' => [
        'required' => 'We need to know your e-mail address!',
    ],
],
```

php

Especificando los atributos personalizados en archivos de idiomas

Si prefieres que la porción `:attribute` de tu mensaje de validación sea reemplazada con un nombre de atributo personalizado, puedes especificar el nombre personalizado en el arreglo `attributes` de tu archivo de idioma `resources/lang/xx/validation.php`:

```
'attributes' => [
    'email' => 'email address',
],
```

php

Especificando los valores personalizados en archivos de idiomas

A veces es posible que necesites que la parte `:value` de tu mensaje de validación sea reemplazada por una representación personalizada del valor. Por ejemplo, considera la siguiente regla que especifica que se requiere un número de tarjeta de crédito si el `payment_type` tiene un valor de `cc`:

```
$request->validate([
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

php

Si esta regla de validación falla, producirá el siguiente mensaje de error:

```
The credit card number field is required when payment type is cc.
```

php

En lugar de mostrar `cc` como el valor del tipo de pago, puedes especificar una representación de valor personalizada en tu archivo de idioma `validation` definiendo un arreglo `values` :

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```

php

Ahora, si la regla de validación falla, producirá el siguiente mensaje:

The credit card number field is required when payment type is credit card.

php

Reglas de validación disponibles

Debajo hay una lista con todas las reglas de validación disponibles y su función:

Accepted	Digits Between	Less Than Or Equal
Active URL	Dimensions (Image Files)	Max
After (Date)	Distinct	MIME Types
After Or Equal (Date)	E-Mail	MIME Type By File Extension
Alpha	Ends With	Min
Alpha Dash	Exclude If	Not In
Alpha Numeric	Exclude Unless	Not Regex
Array	Exists (Database)	Nullable
Bail	File	Numeric
Before (Date)	Filled	Password
Before Or Equal (Date)	Greater Than	Present
Between	Greater Than Or Equal	Regular Expression
Boolean	Image (File)	Required
Confirmed	In	Required If
Date	In Array	Required Unless
Date Equals	Integer	Required With
Date Format	IP Address	Required With All
Different	JSON	Required Without
Digits	Less Than	Required Without All

Same	String	URL
Size	Timezone	UUID
Starts With	Unique (Database)	

accepted

El campo bajo validación debe ser *yes*, *on*, *1*, o *true*. Esto es útil para validar la aceptación de "Términos de Servicio", por ejemplo.

active_url

El campo bajo validación debe tener un registro A o AAAA válido de acuerdo a la función de PHP `dns_get_record`. El hostname de la URL proporcionada es extraído usando la función de PHP `parse_url` antes de ser pasado a `dns_get_record`.

after:date

El campo bajo validación debe ser un valor después de una fecha dada. Las fechas serán pasadas a la función de PHP `strtotime`:

```
'start_date' => 'required|date|after:tomorrow'
```

php

En lugar de pasar una cadena de fecha para que sea evaluada por `strtotime`, puedes especificar otro campo para comparar con la fecha:

```
'finish_date' => 'required|date|after:start_date'
```

php

after_or_equal:date

El campo bajo validación debe ser un valor después o igual a la fecha dada. Para mayor información, observa la regla [after](#).

alpha

El campo bajo validación debe estar compuesto completamente por caracteres alfabéticos.

alpha_dash

El campo bajo validación puede tener caracteres alfanuméricos, al igual que guiones cortos y guiones largos.

alpha_num

El campo bajo validación debe estar compuesto completamente por caracteres alfanuméricos.

array

El campo bajo validación debe ser un `array` de PHP.

bail

Detiene la ejecución de las reglas de validación después del primer error de validación.

before:date

El campo bajo validación debe ser un valor que preceda la fecha dada. Las fechas serán pasadas a la función PHP `strtotime`. Además, como la regla `after` el nombre de otro campo bajo validación puede suministrarse como el valor de `fecha`.

before_or_equal:date

Este campo bajo validación debe ser un valor que preceda o igual a la fecha dada. Las fechas serán pasadas a la función de PHP `strtotime`. Además, como la regla `after` el nombre de otro campo bajo validación puede suministrarse como el valor de `fecha`.

between:min,max

El campo bajo validación debe tener un tamaño entre el `min` y `max` dados. Las cadenas, los números, los arreglos y los archivos se evalúan de la misma manera que la regla `size`.

boolean

El campo bajo validación debe poder ser convertido como un booleano. Las entradas aceptadas son `true`, `false`, `1`, `0`, `"1"`, y `"0"`.

confirmed

El campo bajo validación debe tener un campo que coincida con `foo_confirmation`. Por ejemplo, si el campo bajo validación es `password`, un campo `password_confirmation` que coincida debe estar presente en la entrada.

date

El campo bajo validación debe ser una fecha válida, no relativa, de acuerdo a la función de PHP `strtotime`.

date_equals:date

El campo bajo validación debe ser igual a la fecha dada. Las fechas serán pasadas en la función `strtotime` de PHP.

date_format:format

El campo bajo validación debe coincidir con el *format* dado. Deberías usar `date` o `date_format` al momento de validar un campo, no ambos. Esta regla de validación es compatible con todos los formatos compatibles con la clase [DateTime](#) de PHP.

different:field

El campo bajo validación debe tener un valor distinto de *field*.

digits:value

El campo bajo validación debe ser *numeric* y debe tener una longitud exacta de *value*.

digits_between:min,max

El campo bajo validación debe ser *número* y tener una longitud entre los valores de *min* y *max* dados.

dimensions

El archivo bajo validación debe ser una imagen que cumpla con las restricciones de dimensión como las especificadas por los parámetros de la regla:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

php

Las restricciones disponibles son: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.

Una restricción *ratio* debería ser representada como el ancho dividido por la altura. Esto puede ser especificado o por una instrucción como `3/2` o en decimal como `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

php

Dado que esta regla requiere varios argumentos, puedes usar el método `Rule::dimensions` para construir con fluidez la regla:

```
use Illuminate\Validation\Rule;  
  
Validator::make($data, [  
    'avatar' => [  
        'required',  
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),  
    ],  
]);
```

php

distinct

Al momento de trabajar con arreglos, el campo bajo validación no debe tener ningún valor duplicado.

```
'foo.*.id' => 'distinct'
```

php

email

El campo bajo validación debe estar formateado como una dirección de correo electrónico. Esta validación hace uso del paquete [egulias/email-validator](#) para validar la dirección de correo electrónico. Por defecto, el validador `RFCValidation` es aplicado, pero también puedes aplicar otros estilos de validación:

```
'email' => 'email:rfc,dns'
```

php

El ejemplo de arriba aplicará las validaciones `RFCValidation` y `DNSCheckValidation`. Aquí está una lista de los estilos de validacion que puedes aplicar:

- `rfc` : `RFCValidation`
- `strict` : `NoRFCWarningsValidation`
- `dns` : `DNSCheckValidation`
- `spoof` : `SpoofCheckValidation`
- `filter` : `FilterEmailValidation`

El validador `filter`, que hace uso de la función `filter_var` de PHP, se entrega con Laravel y es un comportamiento anterior a Laravel 5.8. Los validadores `dns` y `spoof` requieren la extensión `intl` de PHP.

ends_with:foo,bar,...

El campo bajo validación debe terminar con alguno de los valores dados.

exclude_if:anotherfield,value

El campo bajo validación será excluido de los datos de la petición retornados por los métodos `validate` y `validated` si el campo *anotherfield* es igual a *value*.

exclude_unless:anotherfield,value

El campo bajo validación será excluido de los datos de la petición retornados por los métodos `validate` y `validated` a menos que *anotherfield* sea igual a *value*.

exists:table,column

El campo bajo validación debe existir en una tabla de base de datos dada.

Uso Básico de la Regla Exists

```
'state' => 'exists:states'
```

php

Si la opción `column` no está especificada, se usará el nombre del campo.

Especificando un Nombre de Columna Personalizado

```
'state' => 'exists:states,abbreviation'
```

php

Ocasionalmente, puedes necesitar especificar una conexión de base de datos para que sea usada por la consulta de `exists`. Puedes acompañar esto al anteponer al nombre de la conexión el nombre de la tabla usando sintaxis de "punto":

```
'email' => 'exists:connection.staff,email'
```

php

En lugar de especificar el nombre de la tabla directamente, puedes especificar el modelo de Eloquent que debe ser usado para determinar el nombre de la tabla:

```
'user_id' => 'exists:App\User,id'
```

php

Si prefieres personalizar la consulta ejecutada por la regla de validación, puedes usar la clase [Rule](#) para definir con fluidez la regla. En este ejemplo, también especificaremos las reglas de validación como un arreglo en vez de usar el carácter `|` para delimitarlas.

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            $query->where('account_id', 1);
        }),
    ],
]);
```

php

file

El campo bajo validación debe ser un archivo que sea cargado exitosamente.

filled

El campo bajo validación no debe estar vacío cuando esté presente.

gt:*field*

El campo bajo validación debe ser mayor que el *field* dado. Los dos campos deben ser del mismo tipo. Las cadenas, los números, los arreglos y los archivos se evalúan utilizando las mismas convenciones que la regla [size](#).

gte:*field*

El campo bajo validación debe ser mayor o igual que el *field* dado. Los dos campos deben ser del mismo tipo. Las cadenas, los números, los arreglos y los archivos se evalúan utilizando las mismas convenciones que la regla [size](#).

image

El archivo bajo validación debe ser una imagen (jpeg, png, bmp, gif, svg o webp)

in:*foo,bar,...*

El archivo bajo validación debe estar incluido en la lista dada de valores. Debido a que esta regla requiere con frecuencia que hagas `implode` a un arreglo, el método `Rule::in` puede ser usado para construir fluidamente la regla:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

php

in_array:*anotherfield**

El campo bajo validación debe existir en los valores de *anotherfield*.

integer

El campo bajo validación debe ser un entero.

Nota

Esta regla de validación no verifica que el campo sea del tipo de variable "entero", sólo que el campo sea una cadena o valor número que contenga un entero.

ip

El campo bajo validación debe ser una dirección IP.

ipv4

El campo bajo validación debe ser una dirección IPv4.

ipv6

El campo bajo validación debe ser una dirección IPv6.

json

El campo bajo validación debe ser una cadena JSON válida.

lt:*field*

El campo bajo validación debe ser menor que el *field* dado. Los dos campos deben ser del mismo tipo. Las cadenas, los números, los arreglos y los archivos se evalúan utilizando las mismas convenciones que la regla `size`.

lte:*field*

El campo bajo validación debe ser menor o igual que el *field* dado. Los dos campos deben ser del mismo tipo. Las cadenas, los números, los arreglos y los archivos se evalúan utilizando las mismas convenciones que la regla `size`.

max:*value*

El campo bajo validación debe ser menor que o igual a un *valor* máximo. Las cadenas, los números, los arreglos y los archivos son evaluados de la misma forma como la regla `size`.

mimetypes:*text/plain*,...

El archivo bajo validación debe coincidir con uno de los tipos MIME dados:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

php

Para determinar el tipo MIME del archivo cargado, el contenido del archivo será leído y el framework intentará suponer el tipo MIME, el cual puede ser distinto del tipo MIME proporcionado por el cliente.

mimes:*foo,bar*,...

El archivo bajo validación debe tener un tipo MIME correspondiente a uno con las extensiones listadas.

Uso Básico de la Regla MIME

```
'photo' => 'mimes:image/jpeg,image/bmp,image/png'
```

php

Incluso aunque solamente necesites especificar las extensiones, en realidad esta regla valida contra el tipo MIME del archivo mediante la lectura de los contenidos del archivo y adivinando su tipo MIME.

Una lista completa de tipos MIME y sus correspondientes extensiones pueden ser encontrados en la siguiente ubicación: <https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

min:value

El campo bajo validación deben tener un *valor* mínimo. Las cadenas, los números, los arreglos y los archivos son evaluados en la misma forma como la regla `size`.

not_in:foo,bar,...

El campo bajo validación no debe estar incluido en la lista dada de valores. El método `Rule::notIn` puede ser usado para construir fluidamente la regla:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

php

not_regex:pattern

El campo bajo validación no debe coincidir con la expresión regular dada.

Internamente, esta regla usa la función PHP `preg_match`. El patrón especificado debe obedecer el mismo formato requerido por `preg_match` y, por lo tanto, también incluir delimitadores válidos. Por ejemplo: `'email' => 'not_regex:/^.+$/i'`.

Nota: Al usar los patrones `regex` / `not_regex`, puede ser necesario especificar reglas en un arreglo en lugar de usar delimitadores de tubería, especialmente si la expresión regular contiene un carácter barra `|`.

nullable

El campo bajo validación puede ser `null`. Esto es particularmente útil al momento de validar tipos primitivos tales como cadenas y enteros que pueden contener valores `null`.

numeric

El campo bajo validación debe ser numérico.

password

El campo bajo validación debe coincidir con la contraseña del usuario autenticado. Puedes especificar una protección de autenticación utilizando el primer parámetro de la regla:

```
'password' => 'password:api'
```

php

present

El campo bajo validación debe estar presente en los datos de entrada pero puede estar vacío.

regex:pattern

El campo bajo validación debe coincidir con la expresión regular dada.

Internamente, esta regla usa la función PHP `preg_match`. El patrón especificado debe obedecer el mismo formato requerido por `preg_match` y, por lo tanto, también incluir delimitadores válidos. Por ejemplo: `'email' => 'regex:/^.[+@].+$/i'`.

Nota: Al usar los patrones `regex` / `not_regex`, puede ser necesario especificar reglas en un arreglo en lugar de usar delimitadores de tubería, especialmente si la expresión regular contiene un carácter barra `|`.

required

El campo bajo validación debe estar presente entre los datos entrada y no vacío. Un campo es considerado "vacío" si algunas de las siguientes condiciones es cierta:

- El valor es `null`.
- El valor es una cadena vacía.
- El valor es un arreglo vacío o un objeto `Countable` vacío.
- El valor es un archivo cargado sin ruta.

required_if:anotherfield,value,...

El campo bajo validación debe estar presente y no vacío si el campo `anotherfield` es igual a cualquier `valor`.

Si deseas construir una condición más compleja para la regla `required_if`, puedes usar el método `Rule::requiredIf`. Este método acepta un valor booleano o un Closure. Cuando se pasa un Closure, éste debe devolver `true` o `false` para indicar si el campo bajo validación es obligatorio:

```
use Illuminate\Validation\Rule;  
  
Validator::make($request->all(), [  
    'role_id' => Rule::requiredIf($request->user()->is_admin),  
]);  
  
Validator::make($request->all(), [  
    'role_id' => Rule::requiredIf(function () use ($request) {  
        return $request->user()->is_admin;  
    }),  
]);
```

php

required_unless:anotherfield,value,...

El campo bajo validación debe estar presente y no vacío a menos que el campo *anotherfield* sea igual a cualquier *valor*.

required_with:foo,bar,...

El campo bajo validación debe estar presente y no vacío *solo* cuando cualquiera de los otros campos especificados están presentes.

required_with_all:foo,bar,...

El campo bajo validación debe estar presente y no vacío *solo* cuando todos los otros campos especificados están presentes.

required_without:foo,bar,...

El campo bajo validación debe estar presente y no vacío *solo* cuando cualquiera de los otros campos especificados no están presentes.

required_without_all:foo,bar,...

El campo bajo validación debe estar presente y no vacío *solo* cuando todos los demás campos especificados no están presentes.

same:field

El campo *field* dado debe coincidir con el campo bajo validación.

size:value

El campo bajo validación debe tener un tamaño que coincida con el *valor* dado. Para datos de cadena, el *valor* corresponde al número de caracteres. Para datos numéricos, el *valor* corresponde a un valor entero dado. Para un arreglo, el valor *size* corresponde con el número de elementos del arreglo. Para archivos, el valor de *size* corresponde al tamaño del archivo en kilobytes.

starts_with:foo,bar,...

El campo bajo validación debe comenzar con uno de los valores dados.

string

El campo bajo validación debe ser una cadena. Si prefieres permitir que el campo también sea `null`, deberías asignar la regla `nullable` al campo.

timezone

El campo bajo validación debe ser un identificador de zona horaria válida de acuerdo con la función de PHP `timezone_identifiers_list`

unique:table,column,except,idColumn

El campo bajo validación debe ser único en una tabla de base de datos dada. Si la opción `column` no es especificada, el nombre del campo será usado.

Especificando un nombre de tabla o columna personalizado:

En lugar de especificar el nombre de la tabla directamente, puedes especificar el modelo de Eloquent que debe ser usado para determinar el nombre de la tabla:

```
'email' => 'unique:App\User,email_address'
```

php

La opción `column` puede ser usada para especificar la columna correspondiente al campo en la base de datos. Si la opción `column` no es especificada, el nombre del campo será usado.

```
'email' => 'unique:users,email_address'
```

php

Conexión de base de datos personalizada

Ocasionalmente, puedes necesitar establecer una conexión personalizada para las consultas de bases de datos hechas por el validador. Como has visto anteriormente, al establecer `unique:users` como una regla de validación usará la conexión de base de datos predeterminada en la consulta de base de datos. Para sobrescribir esto, especifica la conexión y el nombre de la tabla usando la sintaxis de "punto":

```
'email' => 'unique:connection.users,email_address'
```

php

Forzando una regla unique para ignorar un ID dado:

Algunas veces, puedes desear ignorar un ID dado durante la verificación de unicidad. Por ejemplo, considera una pantalla "update profile" que incluya el nombre del usuario, dirección de correo electrónico, y ubicación. Posiblemente, querrás verificar que la dirección de correo electrónico es única. Sin embargo, si el usuario solamente cambia el campo nombre y no el campo con el correo electrónico, no quieres que un error de validación sea lanzado porque el usuario ya es el propietario de la dirección de correo electrónico.

Para instruir al validador para que ignore el ID del usuario, usaremos la clase `Rule` para definir fluidamente la regla. En este ejemplo, también especificaremos las reglas de validación como un arreglo en lugar de usar el carácter `|` para delimitar las reglas:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

php

Nota

Nunca debes pasar ningún input de la solicitud controlado por cualquier usuario en el método `ignore`. En su lugar, sólo debes pasar un ID único generado por el sistema, como un ID

autoincremental o UUID de una instancia de modelo Eloquent. De lo contrario, tu aplicación será vulnerable a un ataque de inyección SQL.

En lugar de pasar el valor de la clave del modelo al método `ignore`, puedes pasar la instancia completa del modelo. Laravel automáticamente extraerá la clave del modelo:

```
Rule::unique('users')->ignore($user)
```

php

Si tu tabla usa un nombre de columna de clave primaria en vez de `id`, puedes especificar el nombre de la columna al momento de llamar al método `ignore`:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

php

Por defecto, la regla `única` verificará la unicidad de la columna que coincide con el nombre del atributo que se valida. Sin embargo, puede pasar un nombre de columna diferente como segundo argumento al método `unique`:

```
Rule::unique('users', 'email_address')->ignore($user->id),
```

php

Agregando cláusulas `where` adicionales:

También puedes especificar restricciones de consultas al personalizar la consulta usando el método `where`. Por ejemplo, agreguemos una restricción que verifique que el `account_id` es `1`:

```
'email' => Rule::unique('users')->where(function ($query) {
    return $query->where('account_id', 1);
})
```

php

url

El campo bajo validación debe ser una URL válida.

uuid

El campo bajo validación debe ser un identificador único universal (UUID) RFC 4122 (versión 1, 3, 4 o 5) válido.

Agregando reglas condicionalmente

Validando sólo cuando un campo esté presente

En algunas situaciones, puedes desear ejecutar la verificación contra un campo **sólo** si ese campo está presente en el arreglo de campos. Para conseguir esto rápidamente, agrega la regla `sometimes` en tu lista:

```
$v = Validator::make($data, [  
    'email' => 'sometimes|required|email',  
]);
```

php

En el ejemplo anterior, el campo `email` solamente será validado si está presente en el arreglo

`$data`.

TIP

Si estás intentando validar un campo que siempre deba estar presente pero puede estar vacío, revisa [esta nota sobre campos opcionales](#)

Validación condicional compleja

Algunas veces puedes desear agregar reglas de validación basadas en lógica condicional más compleja. Por ejemplo, puedes desear solicitar un campo dado solamente si otro campo tiene un valor mayor que 100. O, puedes necesitar que dos campos tengan un valor dado solamente cuando otro campo esté presente. Agregar estas reglas de validación no tiene que ser un dolor. Primero, crea una instancia

`Validator` con tus *reglas estáticas* que nunca cambian:

```
$v = Validator::make($data, [  
    'email' => 'required|email',  
    'games' => 'required|numeric',  
]);
```

php

Asumamos que nuestra aplicación web es sobre coleccionistas de juegos. Si un coleccionista de juego se registra con nuestra aplicación y posee más de 100 juegos, queremos que explique porqué posee tantos juegos. Por ejemplo, quizá administre una tienda de reventa de juegos, o puede ser que solo disfrute

coleccionar. Para agregar este requerimiento condicionalmente, podemos usar el método `sometimes` en la instancia `Validator`:

```
$v->sometimes('reason', 'required|max:500', function ($input) {  
    return $input->games >= 100;  
});
```

php

El primer argumento pasado al método `sometimes` es el nombre del campo que estamos validando condicionalmente. El segundo argumento son las reglas que queremos agregar. Si la `Closure` pasada como tercer argumento devuelve `true`, las reglas serán agregadas. Este método hace que sea muy fácil construir validaciones condicionales complejas. Incluso puedes agregar validaciones condicionales para varios campos de una sola vez:

```
$v->sometimes(['reason', 'cost'], 'required', function ($input) {  
    return $input->games >= 100;  
});
```

php

TIP

El parámetro `$input` pasado a tu `Closure` será una instancia de `Illuminate\Support\Fluent` y puede ser usado para acceder a tus campos y archivos.

Validando arreglos

Validar arreglos basados en campos de entrada de formulario no tiene que ser un dolor. Puedes usar "notación punto" para validar atributos dentro de un arreglo. Por ejemplo, si la solicitud entrante contiene un campo `photos[profile]`, puedes validarla como sigue:

```
$validator = Validator::make($request->all(), [  
    'photos.profile' => 'required|image',  
]);
```

php

También puedes validar cada elemento de un arreglo. Por ejemplo, para validar que cada dirección electrónica en un campo de entrada de arreglo sea único, puedes hacer lo siguiente:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);

```

php

De igual forma, puedes usar el carácter `*` al momento de especificar tus mensajes de validación en tus archivos de idiomas, haciendo que sea muy fácil usar un único mensaje de validación para campos basados en arreglos:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique e-mail address',
    ],
],
```

php

Reglas de validación personalizadas

Usando objetos de reglas

Laravel proporciona una variedad de reglas de validación útiles; sin embargo, puedes desear especificar algunas propias. Un método para registrar reglas de validación personalizadas es usar objetos de regla. Para generar un nuevo objeto de regla, puedes usar el comando Artisan `make:rule`. Usemos este comando para generar una regla que verifique que una cadena esté en mayúscula. Laravel colocará la nueva regla en el directorio `app/Rules`:

```
php artisan make:rule Uppercase
```

php

Una vez que la regla haya sido creada, estaremos listos para definir su comportamiento. Un objeto de regla contiene dos métodos: `passes` and `message`. El método `passes` recibe el nombre y valor de atributo, y debería devolver `true` o `false` dependiendo de si el valor de atributo es válido o no. El método `message` debería devolver el mensaje de error de validación que debería ser usado cuando la validación falle:

```
<?php
```

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, puedes ejecutar el helper `trans` de tu método `message` si prefieres devolver un mensaje de error de tus archivos de traducción:

```
php
/**
 * Get the validation error message.
 *
 * @return string
 */
public function message()
{
    return trans('validation.uppercase');
}
```

Una vez que la regla haya sido definida, puedes adjuntarla a un validador al pasar una instancia del objeto de regla con tus otras reglas de validación:

```
use App\Rules\Uppercase;                                         php

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

Usando closures

Si solo necesitas la funcionalidad de una regla personalizada una vez a lo largo de tu aplicación, puedes usar un Closure en lugar de un objeto de regla. El Closure recibe el nombre del atributo, el valor del atributo y una retorno de llamada (callback) `$fail` que se debe llamar si falla la validación:

```
$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function ($attribute, $value, $fail) {
            if ($value === 'foo') {
                $fail("$attribute.' is invalid.");
            }
        },
    ],
]);
```

Usando extensiones

Otro método para registrar reglas de validación personalizadas es usar el método `extend` en la clase `facade Validator`. Usemos este método dentro de un `proveedor de servicio` para registrar una regla de validación personalizada:

```
<?php                                         php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Validator;
```

```
class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Validator::extend('foo', function ($attribute, $value, $parameters, $validator) {
            return $value == 'foo';
        });
    }
}
```

La Closure del validador personalizada recibe cuatro argumentos: el nombre del atributo `$attribute` que está siendo validado, el valor `$value` del atributo, un arreglo de `$parameters` pasado a la regla, y la instancia `Validator`.

También puedes pasar una clase y método al método `extend` en vez de una Closure:

```
Validator::extend('foo', 'FooValidator@validate');
```

php

Definiendo el mensaje de error

También necesitarás definir un mensaje de error para tu regla personalizada. Puedes hacer eso o usando un arreglo de mensajes personalizados en línea o agregando una entrada en el archivo de idioma de validación. Este mensaje debería ser colocado en el primer nivel del arreglo, no dentro del arreglo `custom`, el cual es solamente para mensajes de error específico de atributos:

```
"foo" => "Your input was invalid!",  
  
"accepted" => "The :attribute must be accepted.",  
  
// The rest of the validation error messages...
```

php

Al momento de crear una regla de validación personalizada, algunas veces puedes necesitar definir reemplazos de marcadores personalizados para los mensajes de error. Puedes hacer eso creando un Validador personalizado como se describió anteriormente, entonces hacer una ejecución del método `replacer` en la clase facade `Validator`. Puedes hacer esto dentro del método `boot` de un proveedor de servicio:

```
/**  
 * Bootstrap any application services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    Validator::extend(...);  
  
    Validator::replacer('foo', function ($message, $attribute, $rule, $parameters)  
    {  
        return str_replace(...);  
    });  
}
```

php

Extensiones implícitas

De forma predeterminada, cuando un atributo que está siendo validado no está presente o contiene un valor vacío como es definido por la regla `required`, las reglas de validación normal, incluyendo las extensiones personalizadas, no son ejecutadas. Por ejemplo, la regla `unique` no será ejecutada contra un valor `null`:

```
$rules = ['name' => 'unique:users,name'];  
  
$input = ['name' => ''];  
  
Validator::make($input, $rules)->passes(); // true
```

php

Para que una regla se ejecute incluso cuando un atributo esté vacío, la regla debe implicar que el atributo sea obligatorio. Para crear tal extensión "implícita", usa el método `Validator::extendImplicit()`:

```
Validator::extendImplicit('foo', function ($attribute, $value, $parameters, $val  
    return $value == 'foo';  
});
```

Nota

Una extensión "implícita" solamente *implica* que el atributo es obligatorio. Si esto realmente invalida un atributo vacío o faltante depende de ti.

Reglas de objetos implícitas

Si te gustaría que una regla de objeto se ejecute cuando un atributo está vacío, debes implementar la interfaz `Illuminate\Contracts\Validation\ImplicitRule`. Esta interfaz funciona como una "interfaz marcador" para el validador; por lo tanto, no contiene ningún método que necesites implementar.

Manejo de Errores

- Introducción
- Configuración
- Manejador de excepciones
 - Método report
 - Método render
 - Excepciones renderizables y reportables

- Excepciones HTTP
 - Páginas de error HTTP personalizadas

Introducción

Cuando comienzas un nuevo proyecto de Laravel, el manejo de excepciones y errores ya está configurado para ti. La clase `App\Exceptions\Handler` es donde todas las excepciones disparadas por tu aplicación son registradas y después renderizadas de vuelta al usuario. Revisaremos más profundamente dentro de esta clase a través de esta documentación.

Configuración

La opción `debug` en tu archivo de configuración `config/app.php` determina cuanta información sobre un error se muestra realmente al usuario. Por defecto, esta opción es establecida para respetar el valor de la variable de entorno `APP_DEBUG`, la cual es almacenada en tu archivo `.env`.

Para desarrollo local, deberías establecer la variable de entorno a `true`. En tu entorno de producción, este valor debería estar siempre `false`. Si el valor es establecido a `true` en producción, te arriesgas a exponer valores de configuración sensativos a los usuarios finales de tu aplicación.

Manejador de excepciones

Método report

Todas las excepciones son manejadas por la clase `App\Exceptions\Handler`. Esta clase contiene dos métodos: `report` y `render`. Examinaremos cada uno de estos métodos en detalle. El método `report` se usa para registrar excepciones o enviarlas a un servicio externo como [Flare](#), [Bugsnag](#) o [Sentry](#). De forma predeterminada, el método `report` pasa la excepción a la clase base donde la excepción es registrada. Sin embargo, eres libre de registrar excepciones en la forma que deseas.

Por ejemplo, si necesitas reportar distintos tipos de excepciones en diferentes formas, puedes usar el operador de comparación `instanceof` de PHP:

```
/**  
 * Report or log an exception.  
 *  
 * This is a great spot to send exceptions to Flare, Sentry, Bugsnag, etc.  
 */
```

php

```
/*
 * @param \Exception $exception
 * @return void
 */
public function report(Exception $exception)
{
    if ($exception instanceof CustomException) {
        //
    }

    parent::report($exception);
}
```

TIP

En lugar de hacer uso de muchos `instanceof` en tu método `report`, considera usar excepciones reportables

Contexto de log global

De estar disponible, Laravel automáticamente agrega el ID del usuario actual al mensaje de log de cada excepción como datos contextuales. Puedes definir tus propios datos contextuales sobrescribiendo el método `context` de la clase `App\Exceptions\Handler` de tu aplicación. Esta información será incluida en cada mensaje de log de excepción escrito por tu aplicación:

```
/**
 * Get the default context variables for logging.
 *
 * @return array
 */
protected function context()
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

php

Helper `report`

Algunas veces puede que necesites reportar una excepción pero continuar manejando la solicitud actual. La función helper `report` permite que reportes rápidamente una excepción usando el método `report` de tu manejador de excepción sin renderizar una página de error:

```
public function isValid($value) php
{
    try {
        // Validate the value...
    } catch (Exception $e) {
        report($e);

        return false;
    }
}
```

Ignorando excepciones por tipo

La propiedad `$dontReport` del manejador de excepción contiene un arreglo de tipos de excepción que no serán registrados. Por ejemplo, excepciones que resulten de errores 404, al igual que otros varios tipos de errores, no son escritos a tus archivos de log. Puedes agregar otros tipos de excepción a este arreglo cuando lo necesites:

```
/**
 * 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,
];
```

Método render

El método `render` es responsable de convertir una excepción dada en una respuesta HTTP que debería ser devuelta al navegador. De forma predeterminada, la excepción es pasada a la clase base la

cual genera una respuesta para ti. Sin embargo, eres libre de revisar el tipo de excepción o devolver tu propia respuesta personalizada:

```
/** php
 * Render an exception into an HTTP response.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Exception $exception
 * @return \Illuminate\Http\Response
 */
public function render(Request $request, Exception $exception)
{
    if ($exception instanceof CustomException) {
        return response()->view('errors.custom', [], 500);
    }

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

Excepciones renderizables y reportables

En lugar de hacer verificaciones por tipo de excepciones en los métodos `report` y `render` del manejador de excepción, puedes definir métodos `report` y `render` directamente en tu excepción personalizada. Cuando estos métodos existen, serán ejecutados automáticamente por el framework:

```
<?php 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 $request
 * @return \Illuminate\Http\Response
 */
public function render($request)
{
    return response(...);
}
```

TIP

Puedes declarar el tipo de cualquier dependencia requerida en el método `report` y el [contenedor de servicios](#) las injectará automáticamente en el método.

Excepciones HTTP

Algunas excepciones describen códigos de error HTTP del servidor. Por ejemplo, esto puede ser un error "página no encontrada" (404), un "error no autorizado" (401) o incluso un error 500 generado por el desarrollador. Con el propósito de generar tal respuesta desde cualquier lugar en tu aplicación, puedes usar el helper `abort` :

```
abort(404);
```

php

El helper `abort` provocará inmediatamente una excepción la cual será renderizada por el manejador de excepción. Opcionalmente, puedes proporcionar el texto de la respuesta:

```
abort(403, 'Unauthorized action.');
```

php

Páginas de error HTTP personalizadas

Laravel hace fácil mostrar páginas de error personalizadas para varios códigos de estado HTTP. Por ejemplo, si deseas personalizar la página de error para los códigos de estado HTTP 404, crea una vista

`resources/views/errors/404.blade.php`. Este archivo será servido en todos los errores 404 generados por tu aplicación. La vista dentro de este directorio debería ser nombrada para coincidir con el código de estado HTTP que les corresponde. La instancia `HttpException` provocada por la función `abort` será pasada a la vista como una variable `$exception`:

```
<h2>{{ $exception->getMessage() }}</h2>
```

php

Puedes publicar las plantillas de página de error de Laravel usando el comando de Artisan `vender:publish`. Una vez que las plantillas han sido publicadas, puedes personalizarlas de la forma que quieras:

```
php artisan vendor:publish --tag=laravel-errors
```

php

Registro (Logging)

- [Introducción](#)
- [Configuración](#)
 - [Construyendo stacks de registros](#)
- [Escribiendo mensajes de registro](#)
 - [Escribiendo a canales específicos](#)
- [Configuración avanzada del canal Monolog](#)
 - [Personalizando Monolog para canales](#)
 - [Creando canales de manejador para Monolog](#)
 - [Creando canales mediante factories](#)

Introducción

Para ayudarte a aprender más acerca de lo que está sucediendo dentro de tu aplicación, Laravel proporciona un robusto servicio de registro que te permite registrar mensajes en archivos, en el registro de errores del sistema e incluso en Slack para notificar a todo tu equipo.

De forma interna, Laravel usa la biblioteca [Monolog](#), que proporciona soporte para una variedad de poderosos manejadores de registros. Laravel hace que sea pan comido configurar dichos manejadores, permitiéndote mezclar y juntarlos para personalizar el manejo de registros en tu aplicación.

Configuración

Toda la configuración para el sistema de registros de tu aplicación se encuentra en el archivo de configuración `config/logging.php`. Este archivo te permite configurar los canales de registros de tu aplicación, así que asegurarte de revisar cada uno de los canales disponibles y sus opciones.

Revisaremos algunas opciones comunes a continuación.

Por defecto, Laravel usara el canal `stack` al registrar mensajes. El canal `stack` es usado para agregar múltiples canales de registros en un solo canal. Para más información sobre construir stacks, revisa la [documentación debajo](#).

Configurando el nombre del canal

Por defecto, Monolog es instanciado con un "nombre de canal" que concuerda con el entorno actual, como `production` o `local`. Para cambiar este valor, agrega una opción `name` a la configuración de tu canal:

```
'stack' => [
    'driver' => 'stack',
    'name' => 'channel-name',
    'channels' => ['single', 'slack'],
],
```

Drivers de canales disponibles

Nombre	Descripción
<code>stack</code>	Wrapper para facilitar la creación de canales "multi-canales"

Nombre	Descripción
single	Canal de registro de un sólo archivo o ubicación (<code>streamHandler</code>)
daily	Driver de Monolog basado en <code>RotatingFileHandler</code> que rota diariamente
slack	Driver de Monolog basado en <code>SlackWebhookHandler</code>
papertrail	Driver de Monolog basado en <code>SyslogUdpHandler</code>
syslog	Driver de Monolog basado en <code>SyslogHandler</code>
errorlog	Driver de Monolog basado en <code>ErrorLogHandler</code>
monolog	Driver factory de Monolog que puede usar cualquier manejador de Monolog soportado
custom	Driver que llama a un factory especificado para crear un canal

TIP

Comprueba la documentación en [personalización avanzada de canales](#) para aprender más sobre `monolog` y drivers `personalizados`.

Configuración de los canales single y daily

Los canales `single` y `daily` tienen tres opciones de configuración opcionales: `bubble` , `permission` y `locking` .

Nombre	Descripción	Default
<code>bubble</code>	Indica si los mensajes deberían llegar a otros canales después de ser manejados	<code>true</code>
<code>permission</code>	Los permisos del archivo de registro	<code>0644</code>
<code>locking</code>	Intenta bloquear el archivo de registro antes de escribirlo	<code>false</code>

Configurando el canal de papertrail

El canal `papertrail` requiere de las opciones de configuración `url` y `port`. Puedes obtener estos valores desde [Papertrail](#).

Configurando el canal de Slack

El canal `slack` requiere una opción de configuración `url`. Esta URL debe coincidir con una URL de un [webhook entrante](#) que has configurado para tu equipo de Slack. Por defecto, Slack sólo recibirá registros en el nivel `critical` y superior; sin embargo, puedes ajustar esto en tu archivo de configuración `logging`.

Construyendo stacks de registros

Como mencionamos anteriormente, el driver `stack` permite que combines múltiples canales en un sólo canal de registro. Para ilustrar cómo usar stacks de registros, vamos a echar un vistazo a un ejemplo de configuración que podrías ver en una aplicación en producción:

```
php
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],
    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],
    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
        'level' => 'critical',
    ],
],
```

Vamos a examinar esta configuración. Primero, observa que nuestro canal `stack` agrega dos canales más mediante su opción `channels`: `syslog` y `slack`. Entonces, al registrar mensajes, ambos canales tendrán la oportunidad de registrar el mensaje.

Niveles de registro

Observa la opción de configuración `level` presente en la configuración de los canales `syslog` y `slack` en el ejemplo superior. Esta opción determina el "nivel" mínimo que un mensaje debe tener para poder ser registrado por el canal. Monolog, que hace funcionar los servicios de registros de Laravel, ofrece todos los niveles de registro definidos en la [especificación RFC 5424](#): `emergency`, `alert`, `critical`, `error`, `warning`, `notice`, `info` y `debug`.

Así que, imagina que registramos un mensaje usando el método `debug`:

```
Log::debug('An informational message.');
```

php

Dada nuestra configuración, el canal `syslog` escribirá el mensaje al registro del sistema; sin embargo, dado que el mensaje de error no es `critical` o superior, no será enviado a Slack. Sin embargo, si registramos un mensaje `emergency`, será enviado tanto al registro del sistema como a Slack dado que el nivel `emergency` está por encima de nuestro umbral mínimo para ambos canales:

```
Log::emergency('The system is down!');
```

php

Escribiendo mensajes de error

Puedes escribir información a los registros usando el `facade Log`. Como mencionamos anteriormente, el registrador proporciona los ocho niveles de registro definidos en la [especificación 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);
```

php

Así que, podrías llamar a cualquiera de esos métodos para registrar un mensaje para el nivel correspondiente. Por defecto, el mensaje será escrito al canal de registro por defecto tal y como está

configurado en tu archivo de configuración `config/logging.php`:

```
<?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)]);  
    }  
}
```

Información contextual

Un arreglo de datos contextuales puede ser pasado a los métodos de registro. Estos datos contextuales serán formateados y mostrados con el mensaje registrado:

```
Log::info('User failed to login.', ['id' => $user->id]);
```

Escribiendo a canales específicos

Algunas veces podrías querer registrar un mensaje a un canal aparte del canal por defecto de tu aplicación. Podrías usar el método `channel` en el facade `Log` para retornar y registrar a cualquier canal definido en tu archivo de configuración:

```
Log::channel('slack')->info('Something happened!');
```

php

Si quisieras crear un stack de registro a la carta consistiendo de múltiples canales, puedes usar el método `stack`:

```
Log::stack(['single', 'slack'])->info('Something happened!');
```

php

Personalización avanzada de canales de Monolog

Personalizando Monolog para canales

Algunas veces puede que necesites un control total sobre la forma en la que Monolog es configurado para un canal existente. Por ejemplo, podrías querer configurar una implementación personalizada para `FormatterInterface` de Monolog para los manejadores de un canal dado.

Para comenzar, define un arreglo `tap` en la configuración del canal. El arreglo `tap` debe contener una lista de clases que deben tener una oportunidad de personalizar (o hacerle "tap") a la instancia de Monolog luego de que es creada:

```
'single' => [
    'driver' => 'single',
    'tap' => [App\Logging\CustomizeFormatter::class],
    'path' => storage_path('logs/laravel.log'),
    'level' => 'debug',
],
```

php

Una vez que has configurado la opción `tap` en tu canal, estás listo para definir la clase que personalizará tu instancia de Monolog. Esta clase sólo necesita un método: `__invoke`, que recibe una instancia `Illuminate\Log\Logger`. La instancia `Illuminate\Log\Logger` redirige todas las llamadas de métodos a la instancia base de Monolog:

```
<?php

namespace App\Logging;

class CustomizeFormatter
```

php

```
{  
    /**  
     * Customize the given logger instance.  
     *  
     * @param \Illuminate\Log\Logger $logger  
     * @return void  
     */  
    public function __invoke($logger)  
    {  
        foreach ($logger->getHandlers() as $handler) {  
            $handler->setFormatter(...);  
        }  
    }  
}
```

TIP

Todas tus clases "tap" son resultas por el [contenedor de servicios](#), así que cualquier dependencia del constructor que requieran será inyectada automáticamente.

Creando canales para manejadores de Monolog

Monolog tiene una variedad de [manejadores disponibles](#). En algunos casos, el tipo de registro que quieras crear es simplemente un driver de Monolog con una instancia de un handler en específico. Estos canales pueden ser creados usando el driver `monolog`.

Al usar el driver `monolog`, la opción de configuración `handler` es usada para especificar que handler será instanciado. Opcionalmente, cualquier parámetros del constructor que el handler necesite puede ser especificado usando la opción de configuración `with`:

```
'logentries' => [  
    'driver' => 'monolog',  
    'handler' => Monolog\Handler\SyslogUdpHandler::class,  
    'with' => [  
        'host' => 'my.logentries.internal.datahubhost.company.com',  
        'port' => '10000',  
    ],  
],
```

Formateadores de Monolog

Al usar el driver `monolog`, `LineFormatter` de Monolog será usado como formateador por defecto. Sin embargo, puedes personalizar el tipo de formateador pasado al manejador usando las opciones de configuración `formatter` y `formatter_with`:

```
'browser' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\BrowserConsoleHandler::class,
    'formatter' => Monolog\Formatter\HtmlFormatter::class,
    'formatter_with' => [
        'dateFormat' => 'Y-m-d',
    ],
],
```

php

Si estás usando un manejador de Monolog que es capaz de proveer su propio formateador, puedes establecer el valor de la opción de configuración `formatter` a `default`:

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
```

php

Creando canales mediante factories

Si quieres definir un canal personalizado completo en el que tienes control total sobre la instanciación y configuración de Monolog, puedes especificar un driver personalizado en tu archivo de configuración `config/logging.php`. Tu configuración debe incluir una opción `via` que apunte a la clase factory que será invocada para crear la instancia de Monolog:

```
'channels' => [
    'custom' => [
        'driver' => 'custom',
        'via' => App\Logging\CreateCustomLogger::class,
    ],
],
```

php

Una vez que has configurado el canal personalizado, estás listo para definir la clase que creará tu instancia de Monolog. Esta clase sólo necesita un método: `__invoke`, el cual debe retornar una instancia de Monolog:

```
<?php  
  
namespace App\Logging;  
  
use Monolog\Logger;  
  
class CreateCustomLogger  
{  
    /**  
     * Create a custom Monolog instance.  
     *  
     * @param array $config  
     * @return \Monolog\Logger  
     */  
    public function __invoke(array $config)  
    {  
        return new Logger(...);  
    }  
}
```

Plantillas Blade

- Introducción
- Herencia de plantillas
 - Definir un layout
 - Extender un layout
- Componentes y slots

- Mostrando datos
 - Frameworks de Blade y JavaScript
- Estructuras de control
 - Sentencias if
 - Sentencias switch
 - Bucles
 - La variable loop
 - Comentarios
 - PHP
- Formularios
 - Campo CSRF
 - Campo method
 - Errores de validación
- Incluyendo sub-vistas
 - Renderizar vistas para colecciones
- Stacks
- Inyección de servicios
- Extendiendo Blade
 - Sentencias if personalizadas

Introducción

Blade es un motor de plantillas simple y a la vez poderoso proporcionado por Laravel. A diferencia de otros motores de plantillas populares de PHP, Blade no te impide utilizar código PHP plano en sus vistas. De hecho, todas las vistas de Blade son compiladas en código PHP plano y almacenadas en caché hasta que sean modificadas, lo que significa que Blade no añade sobrecarga a tu aplicación. Los archivos de las vistas de Blade tienen la extensión `.blade.php` y son usualmente almacenados en el directorio

`resources/views`.

TIP

En [Styde.net](#) contamos con una [completa lección sobre Blade](#) totalmente gratuita.

Herencia de plantillas

Definir un layout

Dos de los principales beneficios de usar Blade son *la herencia de plantillas y secciones*. Para empezar, veamos un ejemplo simple. Primero, vamos a examinar una página de layout "master". Ya que la mayoría de las aplicaciones web mantienen el mismo layout general a través de varias páginas, es conveniente definir este layout como una sola vista de Blade:

```
<!-- Almacenado en resources/views/layouts/app.blade.php --> 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 puedes ver, este archivo contiene el marcado típico de HTML. Sin embargo, toma nota de las directivas `@section` y `@yield`. La directiva `@section`, como su nombre lo indica, define una sección de contenido, mientras que la directiva `@yield` es utilizada para mostrar el contenido en una sección determinada.

Ahora que hemos definido un layout para nuestra aplicación, vamos a definir una página hija que herede el layout.

Extender un layout

Al definir una vista hija, utiliza la directiva de Blade `@extends` para indicar el layout que deberá "heredarse" en la vista hija. Las vistas que extiendan un layout de Blade pueden inyectar contenido en la sección del layout usando la directiva `@section`. Recuerda, como vimos en el ejemplo anterior, los contenidos de estas secciones se mostrarán en el layout usando `@yield`:

```
<!-- Almacenado en resources/views/child.blade.php -->
```

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 adjuntar (en lugar de sobrescribir) contenido al sidebar del layout. La directiva `@@parent` será reemplazada por el contenido del layout cuando la vista sea renderizada.

TIP

Contrario al ejemplo anterior, esta sección `sidebar` termina con `@endsection` en lugar de `@show`. La directiva `@endsection` sólo definirá una sección mientras que `@show` definirá y **automáticamente creará un yield** de la sección.

La directiva `@yield` también acepta un valor por defecto como segundo parametro. Este valor será renderizado si la sección siendo generada es `undefined`:

```
@yield('content', View::make('view.name'))
```

php

Las vistas de Blade se pueden retornar desde las rutas usando el helper global `view`:

```
Route::get('blade', function () {
    return view('child');
});
```

php

TIP

En [Style.net](#) contamos con una lección sobre layouts con Blade totalmente gratuita.

Componentes y slots

Los componentes y slots proporcionan beneficios similares a secciones y layouts; sin embargo, algunos encontrarán el modelo mental de componentes y slots más fácil de comprender. Primero, imaginemos un componente "alert" reutilizable que queremos que se reutilice en toda nuestra aplicación:

```
<!-- /resources/views/alert.blade.php --> php

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

La variable `{{ $slot }}` tendrá el contenido que deseamos injectar en el componente. Ahora, para construir el componente, podemos usar la directiva de Blade `@component` :

```
@component('alert') php
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Para instruir a Laravel para que cargue la primera vista que existe desde un arreglo de posibles vistas para el componente, puedes usar la directiva `componentFirst` :

```
@componentfirst(['custom.alert', 'alert']) php
    <strong>Whoops!</strong> Something went wrong!
@endcomponentfirst
```

A veces es útil definir múltiples slots para un componente. Vamos a modificar nuestro componente alerta para permitir la inyección de un "título". Los slots nombrados podrán despegarse al hacer "echo" a la variable que coincide con sus nombres:

```
<!-- /resources/views/alert.blade.php --> php
```

```
<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>

    {{ $slot }}
</div>
```

Ahora, podemos injectar contenido en el slot nombrado usando la directiva `@slot`. Cualquier contenido que no esté en la directiva `@slot` será pasado al componente en la variable `$slot`:

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

php

Pasando información adicional a los componentes

En ocasiones puedes necesitar pasar información adicional al componente. Por esta razón, puedes pasar un arreglo de información como segundo argumento a la directiva `@component`. Toda la información se hará disponible para la plantilla del componente como variables:

```
@component('alert', ['foo' => 'bar'])
    ...
@endcomponent
```

php

Agregando alias a componentes

Si tus componentes de Blade están almacenados en un subdirectorio, puedes querer agregarles un alias para tener un acceso más fácil. Por ejemplo, imagina un componente de Blade que está almacenado en `resources/views/components/alert.blade.php`. Puedes usar el método `component` para agregar un alias al componente de `components.alert` a `alert`. Típicamente, esto debe ser realizado en el método `boot` de tu `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

Blade::component('components.alert', 'alert');
```

php

Una vez que el alias ha sido agregado al componente, puedes renderizarlo usando una directiva:

```
@alert(['type' => 'danger'])  
    You are not allowed to access this resource!  
@endalert
```

php

Puedes omitir los parametros del componente si este no tiene slots adicionales:

```
@alert  
    You are not allowed to access this resource!  
@endalert
```

php

Mostrando datos

Puedes mostrar datos pasados a tu vista de Blade al envolver la variable entre llaves. Por ejemplo, dada la siguiente ruta:

```
Route::get('greeting', function () {  
    return view('welcome', ['name' => 'Samantha']);  
});
```

php

Puedes mostrar el contenido de la variable `name` de la siguiente manera:

```
Hello, {{ $name }}.
```

php

No estás limitado a mostrar sólo el contenido de las variables pasadas a la vista. También puedes hacer echo al resultado de cualquier función de PHP. De hecho, puedes poner cualquier código PHP que deseas dentro de la declaración echo de Blade:

```
The current UNIX timestamp is {{ time() }}.
```

php

TIP

Las declaraciones de Blade `{{ }}` son enviadas automáticamente mediante la función `htmlspecialchars` de PHP para prevenir ataques XSS.

Mostrar datos no escapados

De manera predeterminada, las declaraciones `{{ }}` de Blade son enviadas mediante la función `htmlspecialchars` de PHP para prevenir ataques XSS. Si no deseas que tu información sea escapada, puedes utilizar la siguiente sintaxis:

```
Hello, {!! $name !!}.  
php
```

Nota

Se muy cuidadoso cuando muestres contenido que sea suministrado por los usuarios de tu aplicación. Usa siempre las sentencias escapadas, ya que éstas previenen ataques XSS cuando se muestran datos suministrados por los usuarios.

Renderizar JSON

En ocasiones puedes pasar un arreglo a tu vista con la intención de renderizarla como JSON para inicializar una variable JavaScript. Por ejemplo:

```
<script>  
    var app = <?php echo json_encode($array); ?>;  
</script>  
php
```

Sin embargo, en lugar de llamar manualmente a `json_encode`, puedes usar la directiva de Blade `@json`. La directiva `@json` acepta los mismos argumentos que la función `json_encode` de PHP:

```
<script>  
    var app = @json($array);  
    var app = @json($array, JSON_PRETTY_PRINT);  
</script>  
php
```

Nota

Sólo debes usar la directiva `@json` para renderizar variables existentes como JSON. Las plantillas Blade están basadas en expresiones regulares e intentar pasar una expresión compleja a la directiva podría causar fallos inesperados.

La directiva `@json` es también útil para trabajar con componentes de Vue o atributos `data-*`:

```
<example-component :some-prop="@json($array)"></example-component>
```

php

Nota

El uso de `@json` en atributos de elementos requiere que esté rodeado por comillas simples.

Codificación de entidades HTML

Por defecto, Blade (y el helper `e` de Laravel) codificarán doblemente las entidades HTML. Si te gustaría deshabilitar la codificación doble, llama al método `Blade::withoutDoubleEncoding` desde el método `boot` de tu `AppServiceProvider`:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\Blade;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Bootstrap any application services.  
     *  
     * @return void  
     */  
    public function boot()  
    {  
        Blade::withoutDoubleEncoding();  
    }  
}
```

Frameworks de Blade Y JavaScript

Dado que muchos frameworks de JavaScript también usan llaves para indicar que una expresión dada debe mostrarse en el navegador, puedes utilizar el símbolo `@` para informar al motor de renderizado de Blade que una expresión debe permanecer intacta. Por ejemplo:

```
<h1>Laravel</h1>  
  
Hello, @{{ name }}.
```

php

En este ejemplo, el símbolo `@` será removido por Blade; sin embargo, la expresión `@{{ name }}` permanecerá intacta por el motor de Blade, lo que permitirá que pueda ser procesada por tu framework de JavaScript.

La directiva `@verbatim`

Si estás mostrando variables de JavaScript en una gran parte de tu plantilla, puedes ajustar el HTML en la directiva `@verbatim` para que no tengas que poner un prefijo en cada instrucción echo de Blade con un símbolo `@`:

```
@verbatim  
    <div class="container">  
        Hello, {{ name }}.  
    </div>  
@endverbatim
```

php

Estructuras de control

Además de la herencia de plantillas y la visualización de datos, Blade también proporciona accesos directos y convenientes para las estructuras de control comunes de PHP, tales como sentencias condicionales y bucles. Estos accesos directos proporcionan una manera muy limpia y concisa de trabajar con estructuras de control de PHP, al tiempo que permanecen familiares para sus contrapartes de PHP.

Sentencias if

Puede construir sentencias `if` usando las directivas `@if`, `@elseif`, `@else` y `@endif`. Estas directivas funcionan idénticamente a sus contrapartes 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
```

php

Por conveniencia, Blade también proporciona una directiva `@unless` :

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

php

Además de las directivas condicionales previamente mencionadas, las directivas `@isset` y `@empty` pueden ser usadas como accesos directos convenientes para sus respectivas funciones PHP:

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

php

Directivas de autenticación

Las directivas `@auth` y `@guest` pueden ser utilizadas para determinar rápidamente si el usuario actual está autenticado o si es un invitado:

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

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

php

Si es necesario, puede especificar el [guard de autenticación](#) que debe verificarse al usar las directivas

```
@auth y @guest :
```

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

php

Directivas de sección

Puede verificar si una sección tiene contenido usando la directiva `@hasSection` :

```
@hasSection('navigation')
<div class="pull-right">
    @yield('navigation')
</div>

<div class="clearfix"></div>
@endif
```

php

Sentencias switch

Las sentencias switch pueden ser construidas usando las directivas `@switch` , `@case` , `@break` ,

```
@default y @endswitch :
```

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
```

php

```
Default case...
```

```
@endswitch
```

Bucles

Además de las sentencias condicionales, Blade proporciona directivas simples para trabajar con estructuras en bucle de PHP. De nuevo, cada una de estas directivas funciona idénticamente a sus contrapartes PHP:

```
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
```

TIP

Cuando estés dentro del bucle, puedes usar la [variable loop](#) para obtener información valiosa acerca del bucle, como puede ser saber si estás en la primera o última iteración a través del bucle.

Al usar bucles puede finalizarlo u omitir la iteración actual:

```
php  
@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 en una línea:

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

php

La variable loop

Al realizar un ciclo, una variable `$loop` estará disponible dentro del ciclo. Esta variable proporciona acceso a un poco de información útil, como el índice del ciclo actual y si es la primera o la última iteración del ciclo:

```
@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
```

php

Si estás en un bucle anidado, puedes acceder a la variable `$loop` del bucle padre 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
```

php

La variable `$loop` también contiene una variedad de otras propiedades útiles:

Propiedad	Descripción
<code>\$loop->index</code>	El índice de la iteración del ciclo actual (comienza en 0).
<code>\$loop->iteration</code>	Iteración del ciclo actual (comienza en 1).
<code>\$loop->remaining</code>	Iteraciones restantes en el ciclo.
<code>\$loop->count</code>	La cantidad total de elementos en el arreglo que se itera.
<code>\$loop->first</code>	Si esta es la primera iteración a través del ciclo.
<code>\$loop->last</code>	Si esta es la última iteración a través del ciclo.
<code>\$loop->even</code>	Si esta es una iteración par a través del ciclo.
<code>\$loop->odd</code>	Si esta es una iteración impar a través del ciclo.
<code>\$loop->depth</code>	El nivel de anidamiento del bucle actual.
<code>\$loop->parent</code>	Cuando está en un bucle anidado, la variable de bucle del parente.

Comentarios

Blade también le permite definir comentarios en sus vistas. Sin embargo, a diferencia de los comentarios HTML, los comentarios de Blade no son incluidos en el HTML returned por la aplicación:

```
{{-- This comment will not be present in the rendered HTML --}}
```

php

PHP

En algunas situaciones, es útil insertar código PHP en sus vistas. Puedes usar la directiva de Blade `@php` para ejecutar un bloque de PHP plano en tu plantilla:

```
@php  
//  
@endphp
```

php

TIP

A pesar que Blade proporciona esta función, usarla con frecuencia puede ser una señal de que tienes demasiada lógica incrustada dentro de tu plantilla.

Formularios

Campo CSRF

Cada vez que defines un formulario HTML en tu aplicación, debes incluir un campo de token CSRF oculto en el formulario para que [el middleware de protección CSRF](#) pueda validar la solicitud. Puedes usar la directiva `@csrf` de Blade para generar el campo de token:

```
<form method="POST" action="/profile">  
    @csrf  
  
    ...  
</form>
```

php

Campo method

Dado que los formularios HTML no pueden hacer solicitudes `PUT`, `PATCH` o `DELETE` necesitarás agregar un campo `_method` oculto para suplantar estos verbos HTTP. La directiva `@method` de Blade puede crear este campo por ti:

```
<form action="/foo/bar" method="POST">  
    @method('PUT')
```

php

```
    ...  
  </form>
```

Errores de validación

La directiva `@error` puede ser usada para comprobar rápidamente si existen [mensajes de error de validación](#) para un atributo dado. Para una directiva `@error`, puedes imprimir la variable `$message` para mostrar el mensaje de error:

```
<!-- /resources/views/post/create.blade.php -->  
  <label for="title">Post Title</label>  
  <input id="title" type="text" class="@error('title') is-invalid @enderror">  
  @error('title')  
    <div class="alert alert-danger">{{ $message }}</div>  
  @enderror
```

php

Puedes pasar [el nombre de un error específico](#) como segundo parametro de la directiva `@error` para retornar los mensajes de error en páginas que contienen múltiples formularios:

```
<!-- /resources/views/auth.blade.php -->  
  
  <label for="email">Email address</label>  
  
  <input id="email" type="email" class="@error('email', 'login') is-invalid @enderror">  
  @error('email', 'login')  
    <div class="alert alert-danger">{{ $message }}</div>  
  @enderror
```

php

Incluyendo sub-vistas

La directiva `@include` de Blade te permite incluir una vista de Blade desde otra vista. Todas las variables que estén disponibles en la vista padre estarán disponibles para la vista incluida:

```
<div>  
  @include('shared.errors')
```

php

```
<form>
    <!-- Form Contents -->
</form>
</div>
```

Aunque la vista incluida heredará todos los datos disponibles en la vista principal, también puedes pasar un arreglo de datos adicionales a la vista incluida:

```
@include('view.name', ['some' => 'data'])
```

php

Si utilizas `@include` en una vista que no existe, Laravel lanzará un error. Si deseas incluir una vista que puede o no estar presente, deberás utilizar la directiva `@includeIf` :

```
@includeIf('view.name', ['some' => 'data'])
```

php

Si deseas incluir una vista si una condición booleana dada evalua a `true`, puedes utilizar la directiva `@includeWhen` :

```
@includeWhen($boolean, 'view.name', ['some' => 'data'])
```

php

Si te gustaría incluir una vista si una expresión booleana evalua a `false`, puedes usar la directiva `@includeUnless` :

```
@includeUnless($boolean, 'view.name', ['some' => 'data'])
```

php

Para incluir la primera vista que exista para un arreglo dado de vistas, puedes usar la directiva `@includeFirst` :

```
@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])
```

php

Nota

Debes evitar usar las constantes `__DIR__` y `__FILE__` en tus vistas de Blade, ya que se referirán a la ubicación de la vista compilada en caché.

Alias de includes

Si tus includes de Blade están almacenados en un subdirectorío, puedes desear crear un alias de ellos para un acceso más fácil. Por ejemplo, imagina un include de Blade que está almacenado en `resources/views/includes/input.blade.php` con el siguiente contenido:

```
<input type="{{ $type ?? 'text' }}>
```

php

Puedes usar el método `include` para crear un alias al include de `includes.input` a `input`. Normalmente, esto debería hacerse en el método `boot` de tu `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;  
  
Blade::include('includes.input', 'input');
```

php

Una vez que el include tiene un alias asignado, puedes renderizalo usando el nombre del alias como una directiva de Blade:

```
@input(['type' => 'email'])
```

php

Renderizar vistas para colecciones

Puedes combinar bucles e incluirlos en una sola línea con la directiva `@each` de Blade:

```
@each('view.name', $jobs, 'job')
```

php

El primer argumento es la vista parcial a renderizar por cada elemento en el arreglo o colección. El segundo argumento es el arreglo o colección que desea iterar, mientras que el tercer argumento es el nombre de la variable que será asignada a la iteración actual dentro de la vista. Así que, por ejemplo, si está iterando en un arreglo `jobs`, típicamente querrá tener acceso a cada trabajo como una variable `job` en su vista parcial. La llave de la iteración actual estará disponible como la variable `key` en su vista parcial.

También puede pasar un cuarto argumento a la directiva `@each`. Este argumento determina la vista que se va a renderizar si el arreglo dado está vacío.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

php

Nota

Las vistas renderizadas via `@each` no heredan las variables de la vista padre. Si la vista hija requiere de estas variables, deberá usar `@foreach` y `@include` en su lugar.

Pilas

Blade te permite agregar a pilas nombradas que pueden ser renderizados en otra parte de otra vista o layout. Esto es particularmente útil para especificar cualquier librería JavaScript requerida por las vistas hijas:

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

php

Puede agregar una pila tantas veces como lo necesite. Para renderizar el contenido completo de la pila, pasa el nombre de la pila a la directiva `@stack` :

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

php

Si te gustaría agregar contenido al inicio de una pila, debes usar la directiva `@prepend` :

```
@push('scripts')
    This will be second...
@endpush

// Luego...
```

php

```
@prepend('scripts')
    This will be first...
@endprepend
```

Inyección de servicios

La directiva `@inject` puede ser utilizada para recuperar un servicio del [contenedor de servicios](#) de Laravel. El primer argumento pasado a `@inject` es el nombre de la variable en la que se colocará 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>
```

php

Extendiendo Blade

Blade le permite definir sus propias directivas personalizadas utilizando el método `directive`. Cuando el compilador Blade encuentra la directiva personalizada, llamará al callback con la expresión que contiene la directiva.

El siguiente ejemplo crea una directiva `@datetime($var)` que le da formato a la variable `$var`, la cual es una instancia de `DateTime`:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
```

php

```
* @return void
*/
public function register()
{
    Blade::directive('datetime', function ($expression) {
        return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
    });
    //
}

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    //
    Blade::directive('datetime', function ($expression) {
        return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
    });
}
}
```

Como podrás ver, vamos a encadenar el método `format` en cualquier expresión que se pase a la directiva. Entonces, en este ejemplo, el PHP final generado por esta directiva será:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

php

Nota

Después de actualizar la lógica de la directiva de Blade, vas a necesitar eliminar todas las vistas de Blade guardadas en caché. Las vistas de Blade en caché pueden ser eliminadas con el comando de Artisan `view:clear`.

Sentencias if personalizadas

Programar una directiva personalizada algunas veces es más complejo de lo necesario al definir sentencias condicionales simples personalizadas. Por esa razón, Blade proporciona un método

`Blade::if` que le permitirá rápidamente definir directivas condicionales utilizando Closures. Por ejemplo, vamos a definir una condicional personalizada que verifica el entorno actual de la aplicación. Podemos hacer esto en el método `boot` de `AppServiceProvider` :

```
use Illuminate\Support\Facades\Blade; php

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

Una vez que el condicional personalizado haya sido definido, podremos usarlo fácilmente en nuestros templates:

```
@env('local') php
    // 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

@unlessenv('production')
    // The application is not in the production environment...
@endenv
```

Configuración Regional

- Introducción
 - Configurando la configuración regional
- Definiendo cadenas de traducciones
 - Usando claves cortas
 - Usando cadenas de traducciones como claves
- Retornando cadenas de traducciones
 - Reemplazando parametros en cadenas de traducciones
 - Pluralización
- Sobrescribiendo archivos del paquete de idioma

Introducción

Las características de configuración regional de Laravel proporcionan una forma conveniente de retornar cadenas en varios idiomas, permitiéndote soportar fácilmente múltiples idiomas en tu aplicación. Las cadenas de idiomas son almacenadas 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 retornan un arreglo de cadenas con sus claves. Por ejemplo:

```
<?php
return [
  'welcome' => 'Welcome to our application'
];
```

Nota

Para idiomas que difieren por territorio, debes nombrar los directorios de idiomas según la ISO 15897. Por ejemplo, "en_GB" debe ser usado para inglés británico en lugar de "en-gb".

Configurando la configuración regional

El idioma por defecto para tu aplicación se almacena en el archivo de configuración `config/app.php`. Puedes modificar este valor en base a las necesidades de tu aplicación. También puedes cambiar el idioma activo en tiempo de ejecución usando el método `setLocale` en el facade `App`:

```
Route::get('welcome/{locale}', function ($locale) {  
    App::setLocale($locale);  
  
    //  
});
```

php

Puedes configurar un "idioma alternativo", que será usado cuando el idioma activo no contiene una determinada cadena de traducción. Al igual que el idioma por defecto, el idioma alternativo también es configurado en el archivo de configuración `config/app.php`:

```
'fallback_locale' => 'en',
```

php

Determinando la configuración regional actual

Puedes usar los métodos `getLocale` y `isLocale` en el facade `App` para determinar la configuración regional actual o comprobar si la configuración tiene un valor dado:

```
$locale = App::getLocale();  
  
if (App::isLocale('en')) {  
    //  
}
```

php

Definiendo cadenas de traducciones

Usando claves cortas

Típicamente, las cadenas de traducciones son almacenadas en archivos dentro del directorio `resources/lang`. Dentro de este directorio debería haber un directorio para cada idioma soportado por la aplicación:

```
/resources                                php
  /lang
    /en
      messages.php
    /es
      messages.php
```

Todos los archivos de idioma retornan un arreglo de cadenas con sus claves. Por ejemplo:

```
<?php                                         php

// resources/lang/en/messages.php

return [
    'welcome' => 'Welcome to our application'
];
```

Usando cadenas de traducciones como claves

Para aplicaciones con grandes necesidades de traducción, definir cada cadena con una "clave corta" puede volverse confuso rápidamente al hacer referencia a estas en tus vistas. Por este motivo, Laravel también proporciona soporte para definir cadenas de traducciones usando la traducción "por defecto" de la cadena como clave.

Archivos de traducción que usan cadenas de traducción como claves son almacenados como archivos JSON en el directorio `resources/lang`. Por ejemplo, si tu aplicación tiene una traducción en español, debes crear un archivo `resources/lang/es.json`:

```
{                                              php
  "I love programming.": "Me encanta programar."
}
```

Retornando cadenas de traducciones

Puedes retornar líneas desde archivos de idioma usando la función helper `__`. La función `__` acepta el archivo y la clave de la cadena de traducción como primer argumento. Por ejemplo, vamos a retornar la cadena de traducción de `welcome` desde el archivo de idioma `resources/lang/messages.php`:

```
echo __('messages.welcome');  
  
echo __('I love programming.');
```

Si estás usando el [motor de plantillas Blade](#), puedes usar la sintaxis `@` para imprimir la cadena de traducción o usar la directiva `@lang`:

```
{{ __('messages.welcome') }}  
  
@lang('messages.welcome')
```

Si la cadena de traducción especificada no existe, la función `__` retornará la clave de la cadena de traducción. Así que, usando el ejemplo superior, la función `__` retornaría `messages.welcome` si la cadena de traducción no existe.

Nota

La directiva `@lang` no escapa ningún resultado. Eres **totalmente responsable** de escapar la salida al usar esta directiva.

Reemplazando parametros en cadenas de traducciones

Si lo deseas, puedes definir placeholders en tus cadenas de traducción. Todos los placeholders son precedidos por `:`. Por ejemplo, puedes definir un mensaje de bienvenida con un nombre como placeholder:

```
'welcome' => 'Welcome, :name',
```

Para reemplazar los placeholders al retornar una cadena de traducción, pasa un arreglo de reemplazos como segundo argumento de la función `__`:

```
echo __('messages.welcome', ['name' => 'dayle']);
```

php

Si tu placeholder contiene sólo letras mayúsculas o sólo tiene su primera letra en mayúscula, el valor traducido será escrito en mayúsculas de forma correcta:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE  
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

php

Pluralización

La pluralización es un problema complejo, ya que diferentes idiomas tienen una variedad de reglas complejas de pluralización. Usando el símbolo `|`, puedes distinguir entre las formas singulares y plurales de una cadena:

```
'apples' => 'There is one apple|There are many apples',
```

php

Puedes incluso crear reglas de pluralización más complejas que especifican cadenas de traducción para múltiples rangos de números:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

php

Luego de definir una cadena de traducción que tiene opciones de pluralización, puedes usar la función `trans_choice` para retornar la línea de un "conteo" dado. En este ejemplo, dado que el conteo es mayor que uno, la forma plural de la cadena de traducción es retornada:

```
echo trans_choice('messages.apples', 10);
```

php

También puedes definir atributos de placeholder en cadenas de pluralización. Estos placeholders pueden ser reemplazados pasando un arreglo como tercer argumento a la función `trans_choice`:

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',  
echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

php

Si te gustaría mostrar el valor entero que fue pasado a la función `trans_choice`, puedes también usar el placeholder `:count`:

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are :count',
```

php

Sobrescribiendo archivos del paquete de idioma

Algunos paquetes pueden venir con sus propios archivos de idioma. En lugar de cambiar los archivos principales del paquete para modificar esas líneas, puedes sobrescribirlas colocando archivos en el directorio `resources/lang/vendor/{package}/{locale}`.

Así que, por ejemplo, si necesitas sobrescribir las cadenas de traducción en inglés en `messages.php` para un paquete llamado `skyrim/hearthfire`, debes colocar un archivo de idioma en:

`resources/lang/vendor/hearthfire/en/messages.php`. Dentro de este archivo, debes sólo definir las cadenas de traducción que deseas sobrescribir. Cualquier cadena de traducción que no sobrescribas será cargada desde los archivos de idioma originales del paquete.

JavaScript y Estructuración de CSS

- [Introducción](#)
- [Escribiendo CSS](#)
- [Escribiendo JavaScript](#)
 - [Escribiendo componentes de Vue](#)
 - [Usando React](#)
- [Agregando Presets](#)

Introducción

Mientras Laravel no dicta la pauta sobre que pre-procesadores de JavaScript o CSS usar, si proporciona un punto de inicio básico usando [Bootstrap](#), [React](#) y / o [Vue](#) que será de utilidad para muchas aplicaciones. De forma predeterminada, Laravel usa [NPM](#) para instalar ambos paquetes de frontend.

La estructura de Boostrap y Vue proporcionada por Laravel se encuentra en el paquete de Composer [laravel/ui](#) , que se puede instalar usando Composer:

```
composer require laravel/ui --dev
```

Una vez que se haya instalado el paquete [laravel/ui](#) , puedes instalar la estructura del frontend usando el comando [ui](#) de artisan:

```
// Generando estructura básica...
php artisan ui bootstrap
php artisan ui vue
php artisan ui react

// Generando estructura del login y registro...
php artisan ui bootstrap --auth
php artisan ui vue --auth
php artisan ui react --auth
```

CSS

Laravel Mix proporciona una clara y expresiva API sobre compilación de Sass o Less, las cuales son extensiones de CSS plano que agregan variables, mixins y otras poderosas características que hacen el trabajo con CSS mucho más divertido. En este documento, discutiremos brevemente la compilación CSS en general; sin embargo, deberías consultar la documentación de Laravel Mix completa para mayor información sobre compilación de Sass o Less.

JavaScript

Laravel no requiere que uses un framework o biblioteca de JavaScript específica para construir tus aplicaciones. De hecho, no tienes que usar JavaScript en lo absoluto. Sin embargo, Laravel sí incluye algunas de las estructuras básicas para hacer más fácil los primeros pasos para escribir JavaScript moderno usando el framework [Vue](#). Vue proporciona una API expresiva para construir aplicaciones de

JavaScript robustas usando componentes. Como con CSS, podemos usar Laravel Mix para compilar fácilmente componentes de JavaScript en un único archivo de JavaScript para los eventos del navegador.

Removiendo la estructura del frontend

Si prefieres remover la estructura del frontend de tu aplicación, puedes usar el comando Artisan `preset`. Este comando, cuando se combina con la opción `none`, eliminará la maquetación de Bootstrap y Vue de tu aplicación, dejando solamente un archivo Sass en blanco y unas cuántas bibliotecas de utilidades de JavaScript comunes.

```
php artisan preset none
```

Escribiendo CSS

Después de instalar el paquete de Composer `laravel/ui` y [generada la estructura del frontend](#), el archivo `package.json` de Laravel incluye el paquete `bootstrap` que te ayuda a empezar a hacer un prototipo del frontend de tu aplicación usando Bootstrap. Sin embargo, siéntete libre de agregar o eliminar los paquetes del archivo `package.json` como sea necesario para tu aplicación. No es obligatorio que uses el framework Bootstrap para construir tu aplicación de Laravel - se proporciona un buen punto de inicio para aquellos que elijan usarlo.

Antes de compilar tu CSS, instala las dependencias de frontend de tu proyecto usando el [gestor de paquetes para Node \(NPM\)](#):

```
npm install
```

Una vez que las dependencias hayan sido instaladas usando `npm install`, puedes compilar tus archivos Sass a CSS plano usando Laravel Mix. El comando `npm run dev` procesará las instrucciones en tu archivo `webpack.mix.js`. Típicamente, tu CSS compilado estará ubicado en el directorio `public/css`:

```
npm run dev
```

El archivo `webpack.mix.js` incluido de forma predeterminada con Laravel compilará el archivo Sass `resources/sass/app.scss`. Este archivo `app.scss` importa un archivo de variables Sass y carga

Bootstrap, el cual proporciona un buen punto de comienzo para la mayoría de las aplicaciones. Siéntete libre de personalizar el archivo `app.scss` en la forma que deseas o incluso usar un pre-procesador completamente diferente configurando Laravel Mix.

Escribiendo JavaScript

Todas las dependencias de JavaScript requeridas por tu aplicación pueden ser encontradas en el archivo `package.json` en el directorio principal del proyecto. Este archivo es similar a un archivo `composer.json` excepto que éste especifica las dependencias de JavaScript en lugar de las dependencias de PHP. Puedes instalar estas dependencias usando el [gestor de paquetes de Node \(NPM\)](#):

```
npm install
```

TIP

De forma predeterminada, el archivo `package.json` de Laravel incluye unos cuantos paquetes tales como `vue` y `axios` para ayudarte a empezar a construir tu aplicación de JavaScript. Siéntete libre de agregar o eliminar del archivo `package.json` según sea necesario para tu aplicación.

Una vez que los paquetes sean instalados, puedes usar el comando `npm run dev` para [compilar tus recursos](#). Webpack es un empaquetador de módulos para aplicaciones modernas en JavaScript. Cuando ejecutes el comando `npm run dev`, Webpack ejecutará las instrucciones en tu archivo `webpack.mix.js`:

```
npm run dev
```

De forma predeterminada, el archivo de `webpack.mix.js` de Laravel compila tu archivo Sass y él de `resources/js/app.js`. Dentro de el archivo `app.js` puedes registrar tus componentes de Vue o, si prefieres un framework distinto, configurar tu propia aplicación de JavaScript. Tu JavaScript compilado será colocado típicamente en el directorio `public/js`.

TIP

El archivo `app.js` cargará el archivo `resources/js/bootstrap.js` el cual estructura y configura Vue, Axios, jQuery, y todas las demás dependencias de JavaScript. Si tienes dependencias adicionales de JavaScript que configurar, puedes hacerlo en este archivo.

Escribiendo componentes de Vue

Al usar el paquete `laravel/ui` para la estructura de tu frontend, se colocará un componente de Vue `ExampleComponent.vue` ubicado en el directorio `resources/js/components`. El archivo `ExampleComponent.vue` es un ejemplo de un [componente Vue de archivo único](#) el cual define su plantilla HTML y JavaScript en el mismo archivo. Los componentes de archivo único proporcionan un enfoque muy conveniente para construir aplicaciones manejadas por JavaScript. El componente de ejemplo es registrado en tu archivo `app.js`:

```
Vue.component(  
    'example-component',  
    require('./components/ExampleComponent.vue').default  
);
```

Para usar el componente en tu aplicación, puedes colocarlo en una de tus plantillas HTML. Por ejemplo, después de ejecutar el comando Artisan `php artisan ui vue --auth` para maquetar las pantallas de registro y autenticación de tu aplicación, podrías colocar el componente en la plantilla de Blade `home.blade.php`:

```
@extends('layouts.app')  
  
@section('content')  
    <example-component></example-component>  
@endsection
```

TIP

Recuerda, deberías ejecutar el comando `npm run dev` cada vez que cambies un componente de Vue. O, puedes ejecutar el comando `npm run watch` para monitorear y recompilar automáticamente tus componentes cada vez que sean modificados.

Si estás interesado en aprender más sobre escribir componentes de Vue, deberías leer la [Documentación de Vue](#), la cual proporciona un minucioso resumen fácil de leer del framework Vue.

TIP

En [Styde.net](#) contamos con un [completo curso sobre Vue.js](#) que cubre todo los aspectos del framework.

Usando React

Si prefieres usar React para construir tu aplicación de JavaScript, Laravel hace que sea una tarea fácil la de intercambiar la estructura de Vue con la de React:

```
composer require laravel/ui --dev

php artisan ui react

// Generando la estructura de login y registro
php artisan ui react --auth
```

Este único comando removerá la estructuración de Vue y la reemplazará con la de React, incluyendo un componente de ejemplo.

Agregando presets

Los presets son "macroable", lo que te permite agregar métodos adicionales a la clase `UiCommand` en tiempo de ejecución. Por ejemplo, el siguiente código agrega un método `nextjs` a la clase `UiCommand`. Por lo general debes declarar los macros de un preset en un [proveedor de servicios](#):

```
use Laravel\Ui\UiCommand;                                         php

UiCommand::macro('nextjs', function (UiCommand $command) {
    // Estructura de tu frontend...
});
```

Luego, puedes llamar al nuevo preset a través del comando `ui`:

```
php artisan ui nextjs
```

Compilación De Assets (Laravel Mix)

- Introducción
- Instalación y configuración
- Ejecutando Mix
- Trabajando con hojas de estilos
 - Less
 - Sass
 - Stylus
 - PostCSS
 - CSS plano
 - Procesamiento de URLs
 - Mapeo de fuente
- Trabajando con JavaScript
 - Extracción de paquetes de terceros
 - React
 - Vanilla JS
 - Configuración de webpack personalizada
- Copiando archivos y directorios
- Versionando / Destrucción de caché
- Recarga de Browsersync
- Variables de entorno
- Notificaciones

Introducción

Laravel Mix proporciona una API fluida para definir pasos de compilación de Webpack para tu aplicación de Laravel usando múltiples preprocesadores de CSS y JavaScript. A través de encadenamiento de cadenas simples, puedes definir fluidamente tus pipelines de assets. Por ejemplo:

```
mix.js('resources/js/app.js', 'public/js')
    .sass('resources/sass/app.scss', 'public/css');
```

php

Si alguna vez has estado confundido o agobiado al comenzar con Webpack y la compilación de assets, amarás Laravel Mix. Sin embargo, no estás obligado a usarlo durante el desarrollo de tu aplicación. Eres libre de usar cualquier pipeline de assets que deseas o incluso ninguno.

Instalación y configuración

Instalando Node

Antes de ejecutar Mix, debes asegurar de que Node.js y NPM están instalados en tu máquina.

```
node -v
npm -v
```

php

Por defecto, Laravel Homestead incluye todo lo que necesitas; sin embargo, si no estás usando Vagrant, entonces puedes fácilmente instalar la última versión de Node y NPM usando instaladores sencillos desde [su página de descargas](#).

Laravel Mix

El único paso restante es instalar Laravel Mix. Dentro de una instalación nueva de Laravel, encontrarás un archivo `package.json` en la raíz de tu estructura de directorios. El archivo por defecto `package.json` incluye todo lo que necesitas para comenzar. Piensa en éste como tu archivo `composer.json`, excepto que define dependencias de Node en lugar de PHP. Puedes instalar las dependencias a las cuales haces referencia ejecutando:

```
npm install
```

php

Ejecutando Mix

Mix es una capa de configuración basado en [Webpack](#), así que para ejecutar tus tareas de Mix sólo necesitas ejecutar uno de los scripts de NPM incluidos en el archivo `package.json` por defecto de Laravel:

```
// Run all Mix tasks...
npm run dev

// Run all Mix tasks and minify output...
npm run production
```

Observando cambios en los assets

El comando `npm run watch` continuará ejecutándose en tu terminal y observando todos los archivos relevantes por cambios. Webpack entonces automáticamente recompilará tus assets cuando detecte un cambio:

```
npm run watch
```

Puedes encontrar que en algunos entornos Webpack no está actualizando los cambios en tus archivos. Si éste es el caso en tu sistema, considera usar el comando `watch-poll`:

```
npm run watch-poll
```

Trabajando con hojas de estilos

El archivo `webpack.mix.js` es el punto de entrada para toda la compilación de assets. Piensa en éste como un wrapper de configuración liviano alrededor de Webpack. Las tareas de Mix pueden ser encadenadas para definir exactamente cómo tus assets deben ser compilados.

Less

El método `less` puede ser usado para compilar [Less](#) a CSS. Vamos a compilar nuestro archivo primario `app.less` a `public/css/app.css`.

```
mix.less('resources/less/app.less', 'public/css');
```

Múltiples llamadas al método `less` pueden ser usadas para compilar múltiples archivos:

```
mix.less('resources/less/app.less', 'public/css')
    .less('resources/less/admin.less', 'public/css');
```

php

Si deseas personalizar el nombre del archivo CSS compilado, puedes pasar una ruta de archivo completa como segundo argumento al método `less`:

```
mix.less('resources/less/app.less', 'public/stylesheets/styles.css');
```

php

Si necesitas sobrescribir [opciones subyacentes de Less](#), puedes pasar un objeto como tercer argumento a `mix.less()`:

```
mix.less('resources/less/app.less', 'public/css', {
    strictMath: true
});
```

php

Sass

El método `sass` te permite compilar [Sass](#) a CSS. Puedes usar el método de la siguiente manera:

```
mix.sass('resources/sass/app.scss', 'public/css');
```

php

De nuevo, como el método `less`, puedes compilar múltiples archivos de CSS a sus archivos de CSS respectivos e incluso personalizar el directorio de salida del CSS resultante:

```
mix.sass('resources/sass/app.sass', 'public/css')
    .sass('resources/sass/admin.sass', 'public/css/admin');
```

php

Opciones de Node-Sass pueden ser proporcionadas como tercer argumento:

```
mix.sass('resources/sass/app.sass', 'public/css', {
    precision: 5
```

php

```
});
```

Stylus

Similar a Less y Sass, el método `stylus` te permite compilar `Stylus` a CSS:

```
mix.stylus('resources/stylus/app.styl', 'public/css');
```

php

También puedes instalar plugins de Stylus adicionales, como `Rupture`. Primero, instala el plugin en cuestión mediante NPM (`npm install rupture`) y luego requiérelo en tu llamada a

```
mix.stylus() :
```

```
mix.stylus('resources/stylus/app.styl', 'public/css', {
    use: [
        require('rupture')()
    ]
});
```

php

PostCSS

`PostCSS`, una herramienta poderosa para transformar tu CSS, es incluido con Laravel Mix. Por defecto, Mix toma ventaja del popular plugin `Autoprefixer` para automáticamente aplicar todos los prefijos necesarios de CSS3. Sin embargo, eres libre de agregar plugins adicionales que sean apropiados para tu aplicación. Primero, instala el plugin deseado a través de NPM y luego haz referencia a éste en tu archivo

```
webpack.mix.js :
```

```
mix.sass('resources/sass/app.scss', 'public/css')
    .options({
        postCss: [
            require('postcss-css-variables')()
        ]
    });

```

php

CSS plano

Si simplemente te gustaría concatenar algunas hojas de CSS planas a un sólo archivo, puedes usar el método `styles`.

```
mix.styles([
    'public/css/vendor/normalize.css',
    'public/css/vendor/videojs.css'
], 'public/css/all.css');
```

Procesamiento de URLs

Debido a que Laravel Mix está construido en base a Webpack, es importante entender algunos conceptos de Webpack. Para compilación de CSS, Webpack reescribirá y optimizará cualquier llamada a `url()` dentro de tus hojas de estilos. Aunque esto inicialmente puede sonar extraño, es una pieza increíblemente poderosa de funcionalidad. Imagina que queremos compilar Sass que incluye una URL relativa a una imagen:

```
.example {
    background: url('../images/example.png');
}
```

:: note Las rutas absolutas para cualquier `url()` serán excluidas de la reescritura de URLs. Por ejemplo, `url('/images/thing.png')` o `url('http://example.com/images/thing.png')` no serán modificados.

Por defecto, Laravel Mix y Webpack encontrarán `example.png`, lo copiarán a tu directorio `public/images` y luego reescribirán el `url()` dentro de tu hoja de estilos generada. Como tal, tu archivo CSS compilado será:

```
.example {
    background: url(/images/example.png?d41d8cd98f00b204e9800998ecf8427e);
```

Tan útil como esta característica puede ser, es posible que tu estructura de directorios existente ya está configurada en una forma que quieras. Si este es el caso, puedes deshabilitar la reescritura de `url()` de la siguiente forma:

```
mix.sass('resources/app/app.scss', 'public/css')
    .options({
        processCssUrls: false
    });

```

php

Con esta adición a tu archivo `webpack.mix.js`, Mix ya no igualará cualquier `url()` o asset copiado a tu directorio público. En otras palabras, el CSS compilado se verá igual a como originalmente lo escribiste:

```
.example {
    background: url("../images/thing.png");
}
```

php

Mapeo de fuente

Aunque deshabilitado por defecto, el mapeo de fuentes puede ser activado llamando al método `mix.sourceMaps()` en tu archivo `webpack.mix.js`. Aunque viene con un costo de compilación/rendimiento, esto proporcionará información adicional de depuración a las herramientas de desarrollo de tu navegador al usar assets compilados.

```
mix.js('resources/js/app.js', 'public/js')
    .sourceMaps();
```

php

Webpack ofrece una variedad [de estilos de mapeo](#). Por defecto, el estilo de mapeo de Mix está establecido a `eval-source-map`, el cual proporciona un tiempo de recompilación corto. Si quieres cambiar el estilo de mapeo, puedes hacerlo usando el método `sourceMaps`:

```
let productionSourceMaps = false;

mix.js('resources/js/app.js', 'public/js')
    .sourceMaps(productionSourceMaps, 'source-map');
```

php

Trabajando con JavaScript

Mix proporciona múltiples características para ayudarte a trabajar con archivos de JavaScript, como compilar ECMAScript 2015, agrupación de módulos, minificación y concatenar archivos de JavaScript planos. Aún mejor, todos esto funciona fácilmente, sin requerir ningún tipo de configuración personalizada:

```
mix.js('resources/js/app.js', 'public/js');
```

php

Con esta única línea de código, puedes ahora tomar ventaja de:

- Sintaxis de ES2015.
- Modulos
- Compilación de archivos `.vue`.
- Minificación para entornos de producción.

Extracción de paquetes de terceros

Un potencial aspecto negativo de agrupar todo el JavaScript específico de la aplicación con tus paquetes de terceros es que hace que el almacenamiento en caché a largo plazo sea más difícil. Por ejemplo, una sola actualización al código de tu aplicación forazará el navegador a recargar todas tus paquetes de terceros incluso si no han cambiado.

Si pretendes hacer actualizaciones frecuentes del JavaScript de tu aplicación, deberías considerar extraer todos tus paquetes de terceros a su propio archivo. De esta forma, un cambio en el código de tu aplicación no afectará el almacenamiento en caché de tu archivo grande `vendor.js`. El método `extract` de Mix hace que esto sea muy fácil:

```
mix.js('resources/js/app.js', 'public/js')
    .extract(['vue'])
```

php

El método `extract` acepta un arreglo de todos los paquetes o módulos que deseas extraer a un archivo `vendor.js`. Usando el código de arriba como ejemplo, Mix generará los siguientes archivos:

- `public/js/manifest.js` : *The Webpack manifest runtime*
- `public/js/vendor.js` : *Your vendor libraries*
- `public/js/app.js` : *Your application code*

Para evitar errores de JavaScript, asegurate de cargar estos archivos en el orden adecuado:

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

php

React

Mix puede automáticamente instalar los plugins de Babel necesarios para el soporte de React. Para comenzar, reemplaza tu llamado a `mix.js()` por `mix.react()` :

```
mix.react('resources/js/app.jsx', 'public/js');
```

php

En segundo plano, Mix descargará e incluirá el plugin de Babel `babel-preset-react` apropiado.

Vanilla JS

Similar a combinar hojas de estilos con `mix.styles()`, puedes también combinar y minificar cualquier número de archivos JavaScript con el método `scripts()` :

```
mix.scripts([
    'public/js/admin.js',
    'public/js/dashboard.js'
], 'public/js/all.js');
```

php

Esta opción es particularmente útil para proyectos antiguos donde no necesitas compilación de Webpack para tu JavaScript.

TIP

Una ligera variación de `mix.scripts()` es `mix.babel()`. Su firma de método es identica a `scripts`; sin embargo, el archivo concatenado recibirá compilación de Babel, que traduce cualquier código ES2015 a JavaScript plano que todos los navegadores entenderán.

Configuración personalizada de webpack

Detrás de cámaras, Laravel Mix hace referencia a un archivo preconfigurado `webpack.config.js` para ayudarte a comenzar tan rápido como sea posible. Ocasionalmente, puedes necesitar modificar

este archivo de forma manual. Podrías tener un loader o plugin especial al que necesitas hacer referencia o quizás prefieres usar Stylus en lugar de Sass. En esos casos, tienes dos opciones:

Fusionar configuración personalizada

Mix proporciona un método `webpackConfig` útil que te permite fusionar cualquier configuración pequeña de Webpack. Esta es una opción particularmente atractiva, ya que no requiere que copies y mantengas tu propia copia del archivo `webpack.config.js`. El método `webpackConfig` acepta un objeto, que debe contener cualquier [configuración específica de Webpack](#) que deseas aplicar.

```
mix.webpackConfig({
  resolve: {
    modules: [
      path.resolve(__dirname, 'vendor/laravel/spark/resources/js')
    ]
  }
});
```

Archivos de configuración personalizados

Si te gustaría personalizar completamente tu configuración de Webpack, copia el archivo

`node_modules/laravel-mix/setup/webpack.config.js` al directorio principal de tu proyecto.

Luego, apunta todas las referencias a `--config` en tu archivo `package.json` al nuevo archivo de configuración copiado. Si deseas elegir esta forma de personalización, cualquier actualización futura al archivo `webpack.config.js` debe ser manualmente agregada a tu archivo personalizado.

Copiendo Archivos & Directorios

El método `copy` puede ser usado para copiar archivos y directorios a nuevas ubicaciones. Esto puede ser útil cuando un asset en particular dentro de tu directorio `node_modules` necesita ser reubicado a tu directorio `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, debes usar el método `copyDirectory` en su lugar:

```
mix.copyDirectory('resources/img', 'public/img');
```

php

Versionando / Destrucción de caché

Muchos desarrolladores prefijan sus assets compilados con una marca de tiempo o token único para forzar a los navegadores a cargar los assets nuevos en lugar de servir copias antiguas del código. Mix puede hacer esto por ti usando el método `version`.

El método `version` automáticamente agrega un hash único a los nombres de archivos de todos los archivos compilados, permitiendo una destrucción de caché más conveniente:

```
mix.js('resources/js/app.js', 'public/js')
    .version();
```

php

Luego de generar el archivo versionado, no sabrás el nombre exacto del archivo. Así que, debes usar la función global de Laravel `mix` dentro de tus `vistas` para cargar los assets apropiados. La función `mix` determinará automáticamente el nombre actual del archivo:

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}>
```

php

Dado que los archivos versionados son usualmente necesarios durante el desarrollo, puedes instruir al proceso de versionamiento para que sólo se ejecute durante `npm run production`:

```
mix.js('resources/js/app.js', 'public/js');

if (mix.inProduction()) {
    mix.version();
}
```

php

URLs base personalizadas

Si tus assets compilados de mix son desplegados a un CDN separado de tu aplicación, necesitarás cambiar la URL base generada por la función `mix`. Puedes hacer esto agregando una opción de configuración `mix_url` a tu archivo de configuración `config/app.php`:

```
'mix_url' => env('MIX_ASSET_URL', null)
```

php

Luego de configurar la URL de Mix, la función `mix` agregará la URL configurada al generar URLs a assets:

```
https://cdn.example.com/js/app.js?id=1964becbdd96414518cd
```

php

Recarga con Browsersync

[BrowserSync](#) puede monitorear automáticamente los cambios en tus archivos e injectar tus cambios al navegador sin requerir un refrescamiento manual. Puedes activar el soporte llamando al método

```
mix.browserSync();
```

```
mix.browserSync('my-domain.test');

// Or...

// https://browsersync.io/docs/options
mix.browserSync({
  proxy: 'my-domain.test'
});
```

php

Puedes pasar una cadena (proxy) u objeto (configuraciones de BrowserSync) a este método. Luego, inicia el servidor de desarrollo de Webpack usando el comando `npm run watch`. Ahora, cuando modifiques un script o archivo de PHP, observa mientras el navegador instantáneamente recarga la página para reflejar tus cambios.

Variables de entorno

Puedes injectar variables de entorno a Mix prefijando una clave en tu archivo `.env` con `MIX_`:

```
MIX_SENTRY_DSN_PUBLIC=http://example.com
```

php

Luego de que la variable ha sido definida en tu archivo `.env`, puedes acceder mediante el objeto `process.env`. Si el valor cambia mientras estás ejecutando una tarea `watch`, necesitarás reiniciar la

tarea:

```
process.env.MIX_SENTRY_DSN_PUBLIC
```

php

Notificaciones

Cuando esté disponible, Mix automáticamente mostrará notificaciones del sistema operativo para cada paquete. Esto te dará feedback instantáneo, sobre si la compilación ha sido exitosa o no. Sin embargo, pueden haber casos en los que preferirás deshabilitar estas notificaciones. Uno de esos casos puede ser ejecutar Mix en tu servidor de producción. Las notificaciones pueden ser deshabilitadas mediante el método `disableNotifications`.

```
mix.disableNotifications();
```

php

Autenticación

- [Introducción](#)
 - [Consideraciones de la base de datos](#)
- [Inicio rápido de autenticación](#)
 - [Enrutamiento](#)
 - [Vistas](#)
 - [Autenticando](#)
 - [Retornando el usuario autenticado](#)
 - [Proteger Rutas](#)
 - [Confirmación de contraseña](#)
 - [Regulación De Inicio De Sesión](#)
- [Autenticar usuarios manualmente](#)

- Recordar usuarios
- Otros métodos de autenticación
- Autenticación HTTP básica
 - Autenticación HTTP básica sin estado
- Cerrar sesión
 - Invalidar sesiones en otros dispositivos
- Autenticar con redes sociales
- Agregar guards personalizados
 - Guards de closures de peticiones
- Agregar proveedores de usuarios personalizados
 - La interfaz UserProvider
 - La interfaz Authenticatable
- Eventos

Introducción

TIP

¿Quieres comenzar rápido? Instala el paquete de Composer `laravel/ui` y ejecuta `php artisan ui vue --auth` en una nueva aplicación de Laravel. Luego de migrar tu base de datos, dirígete en tu navegador a `http://tu-app.test/register` o cualquier otra URL asignada a tu aplicación. ¡Estos dos comandos se encargarán de generar todo el sistema de autenticación!

Laravel hace la implementación de la autenticación algo muy sencillo. De hecho, casi todo se configura para ti por defecto. El archivo de configuración de la autenticación está localizado en `config/auth.php`, el cual contiene varias opciones bien documentadas para ajustar el comportamiento de los servicios de autenticación.

En esencia, las características de la autenticación de Laravel están compuestas de "guards" (guardias) y "providers" (proveedores). Los Guards definen cómo los usuarios son autenticados para cada petición. Por ejemplo, Laravel contiene un guard `session` el cual mantiene el estado utilizando el almacenamiento de sesión y las cookies.

Los proveedores definen cómo se retornan los usuarios de tu almacenamiento persistente. Laravel cuenta con soporte para recuperar los usuarios utilizando Eloquent y el constructor de consultas de la

base de datos. Sin embargo, eres libre de definir los proveedores adicionales que requiera tu aplicación.

¡No te preocunes si esto suena confuso por el momento! Muchas aplicaciones nunca necesitarán modificar la configuración predeterminada de la autenticación.

Consideraciones de la base de datos

De manera predeterminada, Laravel incluye un [Modelo de Eloquent](#) `App\User` en tu directorio `app`. Este modelo puede ser utilizado por el controlador de autenticación predeterminado de Eloquent. Si tu aplicación no utiliza Eloquent, deberás utilizar el controlador de autenticación `database` el cual utiliza el constructor de consultas (query builder) de Laravel.

Al crear el esquema de la base de datos para el modelo `App\User`, asegúrate de que la columna `password` sea de al menos 60 caracteres de longitud. Mantener una longitud de columna de cadena predeterminada a 255 caracteres sería una buena opción.

Además, debes verificar que tu tabla `users` (o equivalente) contenga un campo nulo de tipo cadena llamado `remember_token` de 100 caracteres. Esta columna se usará para almacenar un token para los usuarios que seleccionen la opción "remember me" (recuérdame) cuando inicien sesión en tu aplicación.

Inicio rápido de autenticación

Laravel viene con varios controladores de autenticación preconstruidos, los cuales están localizados en el nombre de espacio `App\Http\Controllers\Auth`. `RegisterController` maneja el registro de usuarios nuevos, `LoginController` maneja la autenticación, `ForgotPasswordController` maneja el envío de correos electrónicos para restablecer la contraseña y el `ResetPasswordController` contiene la lógica para reiniciar contraseñas. Cada uno de estos controladores utiliza un trait para incluir los métodos necesarios. En la mayoría de los casos no tendrás que modificar estos controladores en lo absoluto.

Enrutamiento

El paquete de Laravel `laravel/ui` proporciona una manera rápida de generar todas las rutas y vistas que necesitas para la autenticación con unos simples comando:

```
composer require laravel/ui --dev  
php artisan ui vue --auth
```

php

Este comando debe ser utilizado en aplicaciones nuevas e instalará vistas de diseño, registro e inicio de sesión, así como todas las rutas necesarias para la autenticación. También será generado un `HomeController` que se encargará de manejar las peticiones posteriores al login, como mostrar el dashboard de la aplicación.

TIP

Si tu aplicación no necesita registro, puedes desactivarlo eliminando el recién creado `RegisterController` y modificando tu declaración de ruta: `Auth::routes(['register' => false]);`.

Crear aplicaciones incluyendo autenticación

Si estás creando una nueva aplicación y te gustaría incluir autenticación, puedes usar la directiva `--auth` al crear tu aplicación. Este comando creará una nueva aplicación con toda la estructura de autenticación compilada e instalada:

```
laravel new blog --auth
```

php

Vistas

Como se mencionó en la sección anterior, el comando `php artisan ui vue --auth` del paquete `laravel/ui` creará todas las vistas que se necesiten para la autenticación y las colocará en el directorio `resources/views/auth`.

El comando `ui` también creará el directorio `resources/views/layouts`, el cual contendrá la plantilla base para tu aplicación. Todas estas vistas usan el framework de CSS Bootstrap, pero eres libre de modificarlo en base a tus preferencias.

Autenticación

Ahora que ya tienes tus rutas y vistas configuradas para los controladores de autenticación incluidos con el framework, iestás listo para registrar y autenticar usuarios nuevos para tu aplicación! Puedes acceder a tu aplicación en el navegador ya que los controladores de autenticación contienen la lógica (a través de traits) para autenticar usuarios existentes y almacenar usuarios nuevos en la base de datos.

Personalizar rutas

Cuando un usuario se ha autenticado exitosamente, será redirigido a la URI `/home`. Puedes personalizar la ruta de redirección post-autenticación usando la constante `HOME` definida en tu `RouteServiceProvider`:

```
public const HOME = '/home';
```

php

Personalizar usuario

Por defecto, Laravel utiliza el campo `email` para la autenticación. Si deseas modificar esto, puedes definir un método `username` en `LoginController`:

```
public function username()
{
    return 'username';
}
```

php

Personalizar guard

También puedes personalizar el "guard" que es utilizado para autenticar y registrar usuarios. Para empezar, define un método `guard` en `LoginController`, `RegisterController` y `ResetPasswordController`. Este método debe devolver una instancia de guard:

```
use Illuminate\Support\Facades\Auth;

protected function guard()
{
    return Auth::guard('guard-name');
}
```

php

Validación / Personalizar almacenamiento

Para modificar los campos del formulario que son requeridos cuando se registren usuarios nuevos en tu aplicación, o para personalizar cómo los nuevos usuarios son almacenados en tu base de datos, puedes modificar la clase `RegisterController`. Esta clase es responsable de validar y crear usuarios nuevos en tu aplicación.

El método `validator` de `RegisterController` contiene las reglas de validación para los usuarios nuevos de tu aplicación. Eres libre de modificar este método según te convenga.

El método `create` de `RegisterController` es responsable de crear registros nuevos de `App\User` en tu base de datos usando el [ORM Eloquent](#). Eres libre de modificar este método de acuerdo a las necesidades de tu base de datos.

Retornando el usuario autenticado

Puedes acceder al usuario autenticado por medio del 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();
```

php

Alternativamente, una vez que el usuario haya sido autenticado, puedes acceder al usuario autenticado mediante una instancia de `Illuminate\Http\Request`. Recuerda que las clases a las cuales se le declaren el tipo serán inyectadas automáticamente en los métodos de tu controlador:

```
<?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...
```

php

```
    }  
}
```

Determinar si el usuario actual está autenticado

Para determinar si el usuario actual está loggeado en tu aplicación, puedes usar el método `check` del facade `Auth`, el cual devolverá `true` si el usuario está autenticado:

```
use Illuminate\Support\Facades\Auth;  
  
if (Auth::check()) {  
    // The user is logged in...  
}
```

php

TIP

Aún cuando es posible determinar si un usuario está autenticado utilizando el método `check`, típicamente deberás usar un middleware para verificar que el usuario está autenticado antes de permitir al usuario acceder a ciertas rutas / controladores. Para aprender más acerca de esto, echa un vistazo a la documentación para [proteger rutas](#).

Proteger rutas

Puedes utilizar [middleware de rutas](#) para permitir acceder a ciertas rutas a los usuarios autenticados. Laravel incluye un middleware `auth`, el cual está definido en

`Illuminate\Auth\Middleware\Authenticate`. Ya que este middleware está registrado en tu kernel HTTP, todo lo que necesitas hacer es adjuntar el middleware a la definición de la ruta:

```
Route::get('profile', function () {  
    // Only authenticated users may enter...  
})->middleware('auth');
```

php

Si estás utilizando [controladores](#), puedes hacer una llamada al método `middleware` desde el constructor de tu controlador en lugar de adjuntarlo a la definición de la ruta:

```
public function __construct()
{
    $this->middleware('auth');
}
```

php

Redireccionar usuarios no autenticados

Cuando el middleware `auth` detecta un usuario no autorizado, redirigirá al usuario a la ruta nombrada `login`. Puedes modificar este comportamiento actualizando la función `redirectTo` en tu archivo `app/Http/Middleware/Authenticate.php`:

```
/**
 * Get the path the user should be redirected to.
 *
 * @param \Illuminate\Http\Request $request
 * @return string
 */
protected function redirectTo($request)
{
    return route('login');
}
```

php

Especificar un guard

Cuando adjuntas el middleware `auth` a una ruta, también puedes especificar cuál guard deberá ser utilizado para autenticar al usuario. El guard especificado deberá corresponder a una de las llaves en el arreglo `guards` del archivo de configuración `auth.php`:

```
public function __construct()
{
    $this->middleware('auth:api');
}
```

php

Confirmación de contraseña

Algunas veces, puedes querer requerir al usuario la confirmación de su contraseña antes de acceder a alguna área específica de tu aplicación. Por ejemplo, puedes requerir esto antes de que el usuario modifique cualquier configuración de facturación dentro de la aplicación.

Para lograr esto, Laravel proporciona un middleware `password.confirm`. Adjuntar el middleware `password.confirm` a una ruta redireccionará a los usuarios a una pantalla donde necesitan confirmar su contraseña antes de continuar:

```
Route::get('/settings/security', function () {  
    // Users must confirm their password before continuing...  
})->middleware(['auth', 'password.confirm']);
```

php

Luego de que el usuario ha confirmado con éxito su contraseña, este será redirigido a la ruta a la que originalmente intentaron acceder. Por defecto, luego de confirmar su contraseña, el usuario no tendrá que confirmar su contraseña de nuevo por tres horas. Eres libre de personalizar la longitud de tiempo antes de que el usuario deba volver a confirmar su contraseña usando la opción de configuración `auth.password_timeout`.

Regulación De Inicio De Sesión

Si estás utilizando la clase `LoginController` incorporada en Laravel, el trait `Illuminate\Foundation\Auth\ThrottlesLogins` se encontrará incluido en tu controlador. De manera predeterminada, el usuario no será capaz de iniciar sesión durante un minuto si falla al proveer las credenciales correctas después de varios intentos. El regulador (o throttle) es único para el nombre de usuario / dirección de correo electrónico del usuario y su dirección IP.

Autenticar usuarios manualmente

Nota que no estás obligado a utilizar los controladores de autenticación incluidos en Laravel. Si deseas eliminar estos controladores, tendrás que encargarte de administrar la autenticación de usuarios utilizando las clases de autenticación de Laravel directamente. No te preocupes, ies algo sencillo!

Vamos a acceder a los servicios de autenticación de Laravel por medio del `Facade`, así que hay que asegurarnos de importar el facade `Auth` al inicio de la clase. Después, veamos el método `attempt`:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Auth;
```

php

```
class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @param \Illuminate\Http\Request $request
     *
     * @return Response
     */
    public function authenticate(Request $request)
    {
        $credentials = $request->only('email', 'password');

        if (Auth::attempt($credentials)) {
            // Authentication passed...
            return redirect()->intended('dashboard');
        }
    }
}
```

El método `attempt` acepta un arreglo de pares llave / valor como primer argumento. Los valores en el arreglo serán utilizados para encontrar el usuario en la tabla de tu base de datos. Así que, en el ejemplo anterior, el usuario se obtiene por el valor de la columna `email`. Si se encuentra el usuario, la contraseña encriptada obtenida de la base de datos será comparada con el valor `password` pasado al método en el arreglo. No debes encriptar la contraseña especificada para el valor `password`, ya que el framework automáticamente va a encriptarlo antes de compararlo con la contraseña almacenada en la base de datos. Si dos contraseñas encriptadas coinciden, se iniciará una sesión autenticada para el usuario.

El método `attempt` va a devolver `true` si la autenticación fue exitosa. De otra forma, devolverá `false`.

El método `intended` del redireccionador va a redirigir al usuario a la URL que intentaba acceder antes de ser interceptado por el middleware de autenticación. Una URI de fallback puede ser proporcionada al método en caso de que el destino solicitado no esté disponible.

Especificar condiciones adicionales

Si lo deseas, puedes agregar condiciones extras a la consulta de autenticación además del correo electrónico del usuario y su contraseña. Por ejemplo, podemos verificar que un usuario esté marcado como "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1]))  
    // The user is active, not suspended, and exists.  
}
```

Nota

En estos ejemplos, `email` no es una opción requerida, solamente es utilizado como ejemplo. Debes utilizar cualquier columna que corresponda a "username" en tu base de datos.

Acceso a instancias específicas de guard

Puedes especificar qué instancia de guard deseas usar utilizando el método `guard` en el facade `Auth`. Esto te permitirá administrar la autenticación para partes separadas de tu aplicación utilizando modelos autenticables o tablas de usuarios independientes.

El nombre del guard pasado al método `guard` deberá corresponder a uno de los guards configurados en tu archivo de configuración `auth.php`:

```
if (Auth::guard('admin')->attempt($credentials)) {  
    //  
}
```

Cerrar sesión

Para desconectar usuarios de tu aplicación, debes utilizar el método `logout` del facade `Auth`. Esto va a borrar la información de autenticación en la sesión del usuario:

```
Auth::logout();
```

Laravel también proporciona métodos para desconectar a un usuario de la aplicación únicamente en su dispositivo actual, o para desconectar a un usuario de la aplicación en otros dispositivos:

```
Auth::logoutCurrentDevice();
```

```
Auth::logoutOtherDevices();
```

Nota

Antes de usar el método `logoutOtherDevices`, asegúrate de que el middleware `Illuminate\Session\Middleware\AuthenticateSession::class` está presente y activo en el grupo middleware `web` de tu kernel HTTP.

Recordar usuarios

Si desea proporcionar la funcionalidad de "recordarme" en tu aplicación, puedes pasar un valor booleano como segundo argumento al método `attempt`, que mantendrá al usuario autenticado indefinidamente, o hasta que cierre su sesión manualmente. Tu tabla `users` deberá incluir una columna de tipo string llamada `remember_token`, que será utilizada para almacenar el token de "recordarme".

```
if (Auth::attempt(['email' => $email, 'password' => $password], $remember)) {  
    // The user is being remembered...  
}
```

TIP

Si estás utilizando el `LoginController` integrado en tu instalación de Laravel, la lógica apropiada para "recordar" usuarios ya se encontrará implementada por los traits utilizados por el controlador.

Si estás "recordando" usuarios, puedes utilizar el método `viaRemember` para determinar si el usuario se ha autenticado utilizando la cookie "recordarme":

```
if (Auth::viaRemember()) {  
    //  
}
```

Otros métodos de autenticación

Autenticar una instancia de usuario

Si necesitas registrar una instancia de usuario existente en tu aplicación, puedes llamar al método `login` con la instancia de usuario. El objeto proporcionado deberá ser una implementación de la interfaz `Illuminate\Contracts\Auth\Authenticatable`. El modelo `App\User` incluido en Laravel ya implementa esta interfaz:

```
Auth::login($user);
```

php

```
// Login and "remember" the given user...
Auth::login($user, true);
```

Puedes especificar la instancia de guard que deseas utilizar:

```
Auth::guard('admin')->login($user);
```

php

Autenticar un usuario por ID

Para autenticar un usuario en tu aplicación por su ID, debes usar el método `loginUsingId`. Este método acepta la clave primaria del usuario que deseas autenticar:

```
Auth::loginUsingId(1);
```

php

```
// Login and "remember" the given user...
Auth::loginUsingId(1, true);
```

Autenticar un usuario una vez

Puedes utilizar el método `once` para autenticar un usuario en tu aplicación para una única solicitud. No se utilizarán sesiones o cookies, lo que significa que este método puede ser bastante útil al construir una API sin estado:

```
if (Auth::once($credentials)) {
    //
}
```

php

Autenticación HTTP básica

La autenticación HTTP básica [🔗](#) proporciona una manera rápida de autenticar usuarios en tu aplicación sin configurar una página de "login" dedicada. Para iniciar, adjunta el middleware `auth.basic` a tu ruta. El middleware `auth.basic` está incluido en el framework de Laravel, por lo que no hay necesidad de definirlo:

```
Route::get('profile', function () {  
    // Only authenticated users may enter...  
})->middleware('auth.basic');
```

php

Una vez que el middleware haya sido adjuntado a la ruta, se preguntará automáticamente por las credenciales al acceder a la ruta desde tu navegador. Por defecto, el middleware `auth.basic` va a usar la columna `email` en el registro del usuario como "nombre de usuario".

Una nota sobre FastCGI

Si estás usando PHP FastCGI, la Autentincación Básica HTTP podría no funcionar correctamente por defecto. Las siguientes líneas deberán ser agregadas a tu archivo `.htaccess` :

```
RewriteCond %{HTTP:Authorization} ^(.+)$  
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

php

Autenticación HTTP básica sin estado

También puedes utilizar la Autenticación HTTP Básica sin establecer una cookie de identificación en la sesión, esto es particularmente útil para la autenticación API. Para hacer esto define un middleware que llame al método `onceBasic`. Si el método no devuelve ninguna respuesta, la petición puede pasarse a la aplicación:

```
<?php  
  
namespace App\Http\Middleware;  
  
use Illuminate\Support\Facades\Auth;  
  
class AuthenticateOnceWithBasicAuth  
{  
    /**  
     * Handle an incoming request.  
    */
```

php

```
* @param \Illuminate\Http\Request $request
* @param \Closure $next
* @return mixed
*/
public function handle($request, $next)
{
    return Auth::onceBasic() ?: $next($request);
}

}
```

A continuación [registra el middleware de ruta](#) y adjúntalo a la ruta:

```
Route::get('api/user', function () {
    // Only authenticated users may enter...
})->middleware('auth.basic.once');
```

php

Logging Out

Para cerrar manualmente la sesión de un usuario en tu aplicación, puedes usar el método `logout` en el facade `Auth`. Esto limpiará la información de autenticación en la sesión del usuario:

```
use Illuminate\Support\Facades\Auth;

Auth::logout();
```

php

Invalidando sesiones en otros dispositivos

Laravel también proporciona un mecanismo para invalidar y "sacar" las sesiones de un usuario que están activas en otros dispositivos sin invalidar la sesión en el dispositivo actual. Esta característica es típicamente utilizada cuando un usuario está cambiando o actualizando su contraseña y te gustaría invalidar sesiones en otros dispositivos mientras mantienes el dispositivo actual autenticado.

Antes de comenzar, debes asegurarte de que el middleware

`Illuminate\Session\Middleware\AuthenticateSession` está presente y no está comentado en tu clase `app/Http/Kernel.php` del grupo de middleware `web` :

```
'web' => [
    // ...
    \Illuminate\Session\Middleware\AuthenticateSession::class,
    // ...
]
```

php

Luego, puedes usar el método `logoutOtherDevices` en el facade `Auth`. Este método requiere que el usuario proporcione su contraseña actual, que tu aplicación debe aceptar mediante un campo de formulario:

```
use Illuminate\Support\Facades\Auth;

Auth::logoutOtherDevices($password);
```

php

Cuando el método `logoutOtherDevices` es invocado, las otras sesiones del usuario serán invalidadas completamente, lo que quiere decir que serán "sacadas" de todos los guards en los que previamente estaban autenticadas.

Nota

Al usar el middleware `AuthenticateSession` en combinación con un nombre de ruta personalizado para la ruta `login`, debes sobreescibir el método `unauthenticated` en el manejador de excepciones de tu aplicación para redirigir apropiadamente a los usuarios a tu página de inicio de sesión.

Agregar guards personalizados

Puedes definir tu propio guard de autenticación utilizando el método `extend` en el facade `Auth`. Debes colocar la llamada a este método `extend` en el [proveedor de servicios](#). Ya que Laravel cuenta con un `AuthServiceProvider`, puedes colocar el código en ese proveedor:

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
```

php

```
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Auth;

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(Auth::createUserProvider($config['provider']));
        });
    }
}
```

Como puedes ver en el ejemplo anterior, el callback pasado al método `extend` deberá retornar una implementación de `Illuminate\Contracts\Auth\Guard`. Esta interfaz contiene algunos métodos que tendrás que implementar para definir un guard personalizado. Una vez que tu guard personalizado haya sido definido, podrás utilizar este guard en la configuración `guards` de tu archivo de configuración `auth.php`:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

Guards de closures de peticiones

La forma más sencilla de implementar un sistema de autenticación basado en peticiones HTTP es usando el método `Auth::viaRequest`. Este método te permite definir rápidamente tu proceso de

autenticación usando sólo un Closure.

Para comenzar, llama al método `Auth::viaRequest` dentro del método `boot` de tu `AuthServiceProvider`. El método `viaRequest` acepta el nombre de un driver de autenticación como su primer argumento. Este nombre puede ser cualquier cadena que describa tu guard personalizado. El segundo argumento pasado al método debe ser un Closure que reciba la petición HTTP entrante y retorne una instancia de usuario o, si la autenticación falla, `null` :

```
use App\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Register any application authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Auth::viaRequest('custom-token', function ($request) {
        return User::where('token', $request->token)->first();
    });
}
```

Una vez que tu driver de autenticación personalizado ha sido definido, úsallo como un driver dentro de la configuración de `guards` de tu archivo de configuración `auth.php` :

```
'guards' => [
    'api' => [
        'driver' => 'custom-token',
    ],
],
```

Agregar proveedores de usuario personalizados

Si no estás utilizando una base de datos relacional tradicional para almacenar a tus usuarios, deberás extender Laravel con tu propio proveedor de autenticación. Usaremos el método `provider` en el

facade `Auth` para definir un proveedor de usuarios personalizado:

```
<?php  
  
namespace App\Providers;  
  
use App\Extensions\RiakUserProvider;  
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;  
use Illuminate\Support\Facades\Auth;  
  
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'));  
        });  
    }  
}
```

Después de haber registrado el proveedor utilizando el método `provider`, puedes cambiar al nuevo proveedor de usuarios en tu archivo de configuración `auth.php`. Primero, define un `provider` que utilice tu nuevo controlador:

```
'providers' => [  
    'users' => [  
        'driver' => 'riak',  
    ],  
],
```

Finalmente, puedes utilizar este proveedor en tu configuración de `guards`:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

php

La interfaz UserProvider

Las implementaciones `Illuminate\Contracts\Auth\UserProvider` son responsables solamente de obtener una implementación de `Illuminate\Contracts\Auth\Authenticatable` desde un sistema de almacenamiento persistente, como MySQL, Riak, etc. Estas dos interfaces permiten a los mecanismos de autenticación de Laravel continuar funcionando independientemente de cómo esté almacenada la información del usuario o qué tipo de clase es utilizado para representarlo.

Echemos un vistazo a la interfaz `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 $credential
}
```

php

La función `retrieveById` generalmente recibe una clave que representa al usuario, como un ID auto-incrementable de una base de datos MySQL. La implementación `Authenticatable` que coincide con el ID deberá ser recuperado y retornado por el método.

La función `retireveByToken` recupera un usuario por su `$identifier` único y su `$token` "recordar datos", almacenados en el campo `remember_token`. Como en el método anterior, la implementación `Authenticatable` deberá ser retornado.

El método `updateRememberToken` actualiza el campo `$user` y `remember_token` con el nuevo `$token`. Un nuevo token es asignado en un inicio de sesión con "recordar datos" o cuando el usuario cierre su sesión.

El método `retrieveByCredentials` recupera el arreglo de credenciales pasadas al método `Auth::attempt` cuando intentaloguearse a la aplicación. El método "consulta" el almacenamiento persistente en busca de las credenciales que coincidan con las del usuario. Típicamente, este método va a ejecutar una consulta con una condición "where" en `$credentials['username']`. El método deberá retornar una implementación de `Authenticatable`. **Este método no debe intentar realizar ninguna validación o autenticación por contraseña.**

El método `validateCredentials` deberá comparar el `$user` proporcionado con sus `$credentials` para autenticar el usuario. Por ejemplo, este método puede utilizar `Hash::check` para comparar los valores de `$user->getAuthPassword()` al valor de `$credentials['password']`. Este método deberá retornar `true` o `false` indicando si la contraseña es válida o no.

La interfaz Authenticatable

Ahora que hemos explorado cada uno de los métodos en `UserProvider`, vamos a echar un vistazo a la interfaz `Authenticatable`. Recuerda, el proveedor deberá retornar implementaciones de esta interfaz desde los métodos `retrieveById`, `retrieveByToken` 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();  
}
```

php

Esta interfaz es simple. El método `getAuthIdentifierName` debe retornar el nombre del campo "clave primaria" del usuario y el método `getAuthIdentifier` deberá retornar la "clave primaria" del

usuario. En un backend MySQL, nuevamente, esto deberá ser la clave auto-incrementable. El método `getAuthPassword` deberá retornar la contraseña encriptada del usuario. Esta interfaz permite que el sistema de autenticación funcione con cualquier clase de usuario, independientemente de qué capa de abstracción o qué ORM se está utilizando. Por defecto, Laravel incluye una clase `User` en el directorio `app` que implementa esta interfaz, por lo que puedes consultar esta clase para obtener un ejemplo de implementación.

Eventos

Laravel genera una variedad de `eventos` durante el proceso de autenticación. Puedes adjuntar listeners a estos eventos en tu `EventServiceProvider`:

```
php
/**
 * 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 de API

- Introducción
- Configuración
 - Preparando la base de datos
- Generando tokens
 - Hashing tokens
- Protegiendo rutas
- Pasando tokens en peticiones

Introducción

Por defecto, Laravel viene con una sencilla solución para autenticación de API mediante tokens aleatorios asignados a cada usuario de tu aplicación. En tu archivo de configuración

`config/auth.php`, un guard `api` ya está definido y utiliza un driver `token`. Este driver es responsable de inspeccionar el token de la API en la petición entrante y verificar que coincida con el token asignado al usuario en la base de datos.

Nota: Aunque Laravel viene con un sencillo guard de autenticación basado en token, te recomendamos considerar usar Laravel Passport para aplicaciones robustas en producción que

ofrecen autenticación de API.

Configuración

Preparando la base de datos

Antes de usar el driver `token`, necesitarás crear una migración que agrega una columna `api_token` a tu tabla `users`:

```
Schema::table('users', function ($table) {  
    $table->string('api_token', 80)->after('password')  
        ->unique()  
        ->nullable()  
        ->default(null);  
});
```

Una vez que la migración ha sido creada, ejecuta el comando de Artisan `migrate`.

TIP

Si eliges usar un nombre de columna diferente, asegurate de actualizar la opción `storage_key` de la configuración de tu API en el archivo `config/auth.php`.

Generando tokens

Una vez que la columna `api_token` ha sido agregada a tu tabla `users`, estás listo para asignar tokens de API aleatorios a cada usuario que se registra en tu aplicación. Debes asignar dichos tokens cuando un modelo `User` es creado para el usuario durante el registro. Al usar el `scaffolding de autenticación` proporcionado por el paquete de composer `laravel/ui`, esto puede ser hecho en el método `create` de `RegisterController`:

```
use Illuminate\Support\Facades\Hash;  
use Illuminate\Support\Str;  
  
/**  
 * Create a new user instance after a valid registration.  
 */
```

```
* @param array $data
* @return \App\User
*/
protected function create(array $data)
{
    return User::forceCreate([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => Hash::make($data['password']),
        'api_token' => Str::random(80),
    ]);
}
```

Hashing tokens

En los ejemplos de arriba, los tokens de API son almacenados en tu base de datos como texto plano. Si te gustaría agregar un hash a tus tokens de API usando hashing SHA-256, puedes establecer la opción `hash` de la configuración del guard de tu `api` a `true`. El guard `api` está definido en tu archivo de configuración `config/auth.php`:

```
'api' => [
    'driver' => 'token',
    'provider' => 'users',
    'hash' => true,
],
```

php

Generando tokens con hash

Al usar tokens de API con hash, no debes generar tus tokens de API durante el registro del usuario. En su lugar, necesitarás implementar tu propia página de administración de tokens de API dentro de tu aplicación. Esta página debe permitir a los usuarios inicializar y refrescar sus token de API. Cuando un usuario realiza una petición para inicializar o refrescar su token, debes almacenar una copia con hash del token en la base de datos y retornar una copia de texto plano del token a la vista / frontend del cliente para ser mostrado una sola vez.

Por ejemplo, un método de controlador que inicializa / refresca el token para un usuario dado y retorna el texto plano del token como una respuesta JSON pudiera verse de la siguiente manera:

```
<?php
```

php

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Str;

class ApiTokenController extends Controller
{
    /**
     * Update the authenticated user's API token.
     *
     * @param \Illuminate\Http\Request $request
     * @return array
     */
    public function update(Request $request)
    {
        $token = Str::random(80);

        $request->user()->forceFill([
            'api_token' => hash('sha256', $token),
        ])->save();

        return ['token' => $token];
    }
}
```

TIP

Dado que los tokens de la API en el ejemplo superior tienen suficiente entropía, es impráctico crear "tablas arcoíris" que buscar el valor original del token con hash. Por lo tanto, métodos de hashing lentos como `bcrypt` son innecesarios.

Protegiendo rutas

Laravel incluye un [guard de autenticación](#) que validará automáticamente tokens de API en peticiones entrantes. Sólo necesitas especificar el middleware `auth:api` en cualquier ruta que requiera un token de acceso válido:

```
use Illuminate\Http\Request;                                         php

Route::middleware('auth:api')->get('/user', function (Request $request) {
    return $request->user();
});
```

Pasando tokens En peticiones

Hay muchas formas de pasar el token de la API a tu aplicación. Discutiremos cada una de esas formas mientras usamos el paquete HTTP Guzzle para demostrar su uso. Puedes elegir cualquiera de estas formas dependiendo de las necesidades de tu aplicación.

Query string

Los usuarios de tu API pueden especificar su token como un valor de cadena de consulta `api_token` :

```
$response = $client->request('GET', '/api/user?api_token='.$token);                                         php
```

Request payload

Los usuarios de tu API pueden incluir su token de API en los parametros del formulario de la petición como `api_token` :

```
$response = $client->request('POST', '/api/user', [
    'headers' => [
        'Accept' => 'application/json',
    ],
    'form_params' => [
        'api_token' => $token,
    ],
]);
```

Bearer token

Los usuarios de tu API pueden proporcionar su token de API como un token `Bearer` en el encabezado `Authorization` de la petición:

```
$response = $client->request('POST', '/api/user', [
    'headers' => [
        'Authorization' => 'Bearer '.$token,
        'Accept' => 'application/json',
    ],
]);
```

php

Autorización

- Introducción
- Gates
 - Escribiendo gates
 - Autorizando acciones
 - Respuestas de gates
 - Interceptando comprobaciones de gates
- Creando políticas
 - Generando políticas
 - Registrando políticas
- Escribiendo políticas
 - Métodos de política
 - Respuestas de políticas
 - Métodos sin modelos
 - Usuarios invitados
 - Filtros de política
- Autorizando acciones usando políticas
 - Vía el modelo de usuario
 - Vía middleware
 - Vía helpers del controlador
 - Vía plantillas de blade

- Proporcionando contexto adicional

Introducción

Además de proveer servicios de autenticación por defecto, Laravel además provee una forma simple de autorizar acciones del usuario contra un recurso dado. Como con la autenticación, el enfoque de Laravel para la autorización es simple, y hay dos maneras principales de autorizar acciones: **gates** y **policies** (puertas y políticas).

Piensa en los gates y políticas como rutas y controladores. Los Gates proveen una manera simple, basada en funciones anónimas, para definir las reglas de autorización; mientras que las políticas, como los controladores, agrupan la lógica para un modelo o recurso en específico. Vamos a explorar los gates primero y luego las políticas.

No necesitas elegir entre el uso exclusivo de gates o de políticas cuando construyas una aplicación. Lo más probable es que la mayoría de las aplicaciones contengan una mezcla de gates y de políticas ¡Y eso está completamente bien! Los gates son más aplicables a acciones que no estén relacionadas a ningún modelo o recurso, como por ejemplo ver un tablero en el panel de administración. Por otro lado, las políticas deberán ser usadas cuando deseas autorizar una acción para un modelo o recurso en particular.

Gates

Escribiendo gates

Los gates son funciones anónimas (Closures) que determinan si un usuario está autorizado para ejecutar una acción dada y típicamente son definidos en la clase `App\Providers\AuthServiceProvider` usando el facade `Gate`. Los gates siempre reciben la instancia del usuario conectado como el primer argumento y pueden, opcionalmente, recibir argumentos adicionales que sean relevantes, como por ejemplo un modelo de Eloquent:

```
/*
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();
}
```

php

```
Gate::define('edit-settings', function ($user) {
    return $user->isAdmin;
});

Gate::define('update-post', function ($user, $post) {
    return $user->id === $post->user_id;
});

}
```

Los gates además pueden ser definidos escribiendo la clase y método a llamar como una cadena de texto `Class@method`, como cuando definimos controladores en las rutas:

```
/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', 'App\Policies\PostPolicy@update');
}
```

Gates de recursos

También puedes definir las habilidades de múltiples gates a la vez usando el método `resource`:

```
Gate::resource('posts', 'App\Policies\PostPolicy');
```

Esto es idéntico a definir los siguientes Gates uno por uno:

```
Gate::define('posts.view', 'App\Policies\PostPolicy@view');
Gate::define('posts.create', 'App\Policies\PostPolicy@create');
Gate::define('posts.update', 'App\Policies\PostPolicy@update');
Gate::define('posts.delete', 'App\Policies\PostPolicy@delete');
```

Por defecto, las habilidades `view`, `create`, `update`, y `delete` serán definidas. Además puedes sobrescribir las habilidades por defecto pasando un arreglo como tercer argumento al método `resource`. Las llaves del arreglo definen los nombres de las habilidades mientras que los valores definen los nombres de los métodos. Por ejemplo, el siguiente código creará dos nuevas definiciones de Gate - `posts.image` y `posts.photo`:

```
Gate::resource('posts', 'PostPolicy', [
    'image' => 'updateImage',
    'photo' => 'updatePhoto',
]);
```

php

Autorizando acciones

Para autorizar una acción usando gates, deberías usar los métodos `allows` o `denies`. Nota que no necesitas pasar el usuario autenticado cuando llames a estos métodos. Laravel se ocupará de esto por ti de forma automática:

```
if (Gate::allows('edit-settings')) {
    // The current user can edit settings
}

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...
}
```

php

Si quisieras determinar si un usuario en particular está autorizado para ejecutar una acción, puedes llamar al método `forUser` del facade `Gate`:

```
if (Gate::forUser($user)->allows('update-post', $post)) {
    // The user can update the post...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
```

php

```
// The user can't update the post...
}
```

Puedes autorizar múltiples acciones a la vez con los métodos `any` o `none`:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user cannot update or delete the post
}
```

php

Autorizando o mostrando excepciones

Si te gustaría intentar autorizar una acción y automáticamente mostrar un

`Illuminate\Auth\Access\AuthorizationException` si el usuario no está habilitado para realizar la acción dada, puedes usar el método `Gate::authorize`. Las instancias de `AuthorizationException` son convertidas automáticamente a una respuesta HTTP `403`:

```
Gate::authorize('update-post', $post);

// The action is authorized...
```

php

Proporcionando contexto adicional

Los métodos gate para autorizar habilidades (`allows`, `denies`, `check`, `any`, `none`, `authorize`, `can`, `cannot`) y las [directivas de Blade](#) para autorización (`@can`, `@cannot`, `@canany`) pueden recibir un arreglo como segundo argumento. Dichos elementos del arreglo son pasados como parametros al gate, y pueden ser usados como contexto adicional al tomar decisiones de autorización:

```
Gate::define('create-post', function ($user, $category, $extraFlag) {
    return $category->group > 3 && $extraFlag === true;
});

if (Gate::check('create-post', [$category, $extraFlag])) {
```

php

```
// The user can create the post...
}
```

Respuestas de gates

Hasta el momento, sólo hemos examinado gates que retornan simples valores booleanos. Sin embargo, algunas veces podrías querer retornar una respuesta más detallada, incluyendo un mensaje de error.

Para hacer eso, puedes retornar un `Illuminate\Auth\Access\Response` desde tu gate:

```
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function ($user) {
    return $user->isAdmin
        ? Response::allow()
        : Response::deny('You must be a super administrator.');
});
```

Al retornar una respuesta de autorización desde tu gate, el método `Gate::allows` aún retornará un valor booleano simple; sin embargo, puedes usar el método `Gate::inspect` para obtener la respuesta de autorización completa retornada por el gate:

```
$response = Gate::inspect('edit-settings', $post);

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

Por supuesto, al usar el método `Gate::authorize` para mostrar una `AuthorizationException` si la acción no está autorizada, el mensaje de error proporcionado por la respuesta de autorización será propagada a la respuesta HTTP:

```
Gate::authorize('edit-settings', $post);

// The action is authorized...
```

Interceptando comprobaciones de gates

Algunas veces, puedes querer otorgar todas las habilidades a un usuario en específico. Puedes usar el método `before` para definir un callback que es ejecutado antes de todas las demás comprobaciones de autorización:

```
Gate::before(function ($user, $ability) {  
    if ($user->isSuperAdmin()) {  
        return true;  
    }  
});
```

php

Si el callback `before` retorna un resultado que no es null dicho resultado será considerado el resultado de la comprobación.

Puedes usar el método `after` para definir un callback que será ejecutado luego de todas las demás comprobaciones de autorización:

```
Gate::after(function ($user, $ability, $result, $arguments) {  
    if ($user->isSuperAdmin()) {  
        return true;  
    }  
});
```

php

Similar a la comprobación `before`, si el callback `after` retorna un resultado que no sea null dicho resultado será considerado el resultado de la comprobación.

Creando políticas

Generando políticas

Los políticas son clases que organizan la lógica de autorización para un modelo o recurso en particular.

Por ejemplo, si tu aplicación es un blog, puedes tener un modelo `Post` con su correspondiente `PostPolicy` para autorizar acciones de usuario como crear o actualizar posts.

Puedes generar una política usando el comando de Artisan `make:policy`. La política generada será ubicada en el directorio `app/Policies`. Si el directorio no existe en tu aplicación, Laravel lo creará por

tj:

```
php artisan make:policy PostPolicy
```

php

El comando `make:policy` genera una clase de política vacía. Si quieres generar una clase con los métodos de política para un "CRUD" básico ya incluidos en la clase, puedes especificar la opción `--model` al ejecutar el comando:

```
php artisan make:policy PostPolicy --model=Post
```

php

TIP

Todas las políticas son resueltas a través del contenedor de servicios de Laravel, lo que te permite especificar las dependencias necesarias en el constructor de la política y estas serán automáticamente inyectadas.

Registrando políticas

Una vez que la política exista, ésta necesita ser registrada. La clase `AuthServiceProvider` incluida con las aplicaciones de Laravel contiene una propiedad `policies` que mapea tus modelos de Eloquent a sus políticas correspondientes. Registrar una política le indicará a Laravel qué política utilizar para autorizar acciones contra un modelo dado:

```
<?php  
  
namespace App\Providers;  
  
use App\Policies\PostPolicy;  
use App\Post;  
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;  
use Illuminate\Support\Facades\Gate;  
  
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();

    //
}

}
```

Política de auto-descubrimiento

En lugar de registrar manualmente políticas de modelos, Laravel puede auto-descubrir políticas siempre y cuando el modelo y la política sigan la convención de nombre estándar de Laravel. Específicamente, las políticas deben estar en un directorio `Policies` dentro del directorio que contiene los modelos. Así que, por ejemplo, los modelos pueden ser ubicados en el directorio `app` mientras que las políticas pueden tener un sufijo. Así que, un modelo `User` corresponderá a una clase `UserPolicy`.

Si te gustaría proporcionar tu propia lógica para descubrir políticas, puedes registrar un callback personalizado usando el método `Gate::guessPolicyNamesUsing`. Típicamente, este método debe ser llamado desde el método `boot` del `AuthServiceProvider` de tu aplicación:

```
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function ($modelClass) {
    // return policy class name...
});
```

Nota

Cualquier política que está explicitamente mapeada en tu `AuthServiceProvider` tendrá precedencia sobre cualquier posible política auto-descubierta.

Escribiendo políticas

Métodos de política

Una vez que la política haya sido registrada, puedes agregar métodos para cada acción a autorizar. Por ejemplo, vamos a definir un método `update` en nuestro `PostPolicy` para determinar si un `User` dado puede actualizar una instancia de un `Post`.

El método `update` recibirá una instancia de `User` y de `Post` como sus argumentos y debería retornar `true` o `false` indicando si el usuario está autorizado para actualizar el `Post` o no. En el siguiente ejemplo, vamos a verificar si el `id` del usuario concuerda con el atributo `user_id` del post:

```
<?php  
  
namespace App\Policies;  
  
use App\Post;  
use App\User;  
  
class PostPolicy  
{  
    /**  
     * Determine if the given post can be updated by the user.  
     *  
     * @param \App\User $user  
     * @param \App\Post $post  
     * @return Response $response  
     */  
    public function update(User $user, Post $post)  
    {  
        return $user->id === $post->user_id;  
    }  
}
```

php

Puedes continuar definiendo métodos adicionales en la política como sea necesario para las diferentes acciones que este autorice. Por ejemplo, puedes definir métodos `view` o `delete` para autorizar

varias acciones de `Post`, pero recuerda que eres libre de darle los nombres que quieras a los métodos de la política.

TIP

Si usas la opción `--model` cuando generes tu política con el comando de Artisan, éste contendrá métodos para las acciones `viewAny`, `view`, `create`, `update`, `delete`, `restore` y `forceDelete`.

Respuestas de políticas

Hasta el momento, sólo hemos examinado métodos de políticas que retornan simples valores booleanos. Sin embargo, algunas veces puedes querer retornar una respuesta más detallada, incluyendo un mensaje de error. Para hacer eso, puedes retornar un `Illuminate\Auth\Access\Response` desde el método de tu política:

```
use Illuminate\Auth\Access\Response;                                         php

/**
 * 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
        ? Response::allow()
        : Response::deny('You do not own this post.');
}
```

Al retornar una respuesta de autorización desde tu política, el método `Gate::allows` aún retornará un booleano simple; sin embargo, puedes usar el método `Gate::inspect` para obtener la respuesta de autorización completa retornada por el gate:

```
$response = Gate::inspect('update', $post);                                         php
```

```
if ($response->allowed()) {  
    // The action is authorized...  
} else {  
    echo $response->message();  
}
```

Por supuesto, al usar el método `Gate::authorize` para mostrar un `AuthorizationException` si la acción no está autorizada, el mensaje de error proporcionado por la respuesta de autorización será propagado a la respuesta HTTP:

```
Gate::authorize('update', $post);  
  
// The action is authorized...
```

php

Métodos sin modelos

Algunos métodos de políticas solo reciben el usuario autenticado y no una instancia del modelo que autorizan. Esta situación es común cuando autorizamos acciones `create`. Por ejemplo, si estás creando un blog, puedes querer revisar si un usuario está autorizado para crear nuevos posts o no.

Cuando definas métodos de política que no recibirán una instancia de otro modelo, así como el método `create`, debes definir el método con el usuario como único parámetro:

```
/**  
 * Determine if the given user can create posts.  
 *  
 * @param \App\User $user  
 * @return bool  
 */  
public function create(User $user)  
{  
    //  
}
```

php

Usuarios invitados

Por defecto, todos los gates y políticas automáticamente retornan `false` si la petición HTTP entrante no fue iniciada por un usuario autenticado. Sin embargo, puedes permitir que estas comprobaciones de

autorización sean pasadas a tus gates y políticas con una declaración de tipo "opcional" o suministrando un valor por defecto `null` para la definición del argumento de usuario:

```
<?php  
  
namespace App\Policies;  
  
use App\Post;  
use App\User;  
  
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 $post)  
    {  
        return optional($user)->id === $post->user_id;  
    }  
}
```

php

Filtros de política

Es posible que quieras autorizar todas las acciones para algunos usuarios en un política dada. Para lograr esto, define un método `before` en la política. El método `before` será ejecutado antes de los otros métodos en la política, dándote la oportunidad de autorizar la acción antes que el método destinado de la política sea llamado. Esta característica es comúnmente usada para otorgar autorización a los administradores de la aplicación para que ejecuten cualquier acción:

```
public function before($user, $ability)  
{  
    if ($user->isSuperAdmin()) {  
        return true;  
    }  
}
```

php

Si quisieras denegar todas las autorizaciones para un usuario deberías retornar `false` en el método `before`. Si retornas `null`, la decisión de autorización recaerá sobre el método de la política.

Nota

El método `before` de una clase política no será llamado si la clase no contiene un método con un nombre que concuerde con el nombre de la habilidad siendo revisada.

Autorizando acciones usando políticas

Vía el modelo user

El modelo `User` que se incluye por defecto en tu aplicación de Laravel trae dos métodos para autorizar acciones: `can` y `cant` (puede y no puede). El método `can` acepta el nombre de la acción que deseas autorizar y el modelo relevante. Por ejemplo, vamos a determinar si un usuario está autorizado para actualizar un `Post` dado:

```
if ($user->can('update', $post)) {  
    //  
}
```

Si una [política está registrado](#) para el modelo dado, el método `can` automáticamente llamará a la política apropiada y retornará un resultado booleano. Si no se ha registrado una política para el modelo dado, el método `can` intentará llamar al Gate basado en Closures que coincida con la acción dada.

Acciones que no requieren modelos

Recuerda, algunas acciones como `create` pueden no requerir de la instancia de un modelo. En estas situaciones, puedes pasar el nombre de una clase al método `can`. El nombre de la clase ser usado para determinar cuál política usar cuando se autorice la acción:

```
use App\Post;  
  
if ($user->can('create', Post::class)) {  
    // Executes the "create" method on the relevant policy...  
}
```

Vía middleware

Laravel incluye un middleware que puede autorizar acciones antes de que la petición entrante alcance tus rutas o controladores. Por defecto, el middleware `Illuminate\Auth\Middleware\Authorize` es asignado a la llave `can` de tu clase `App\Http\Kernel`. Vamos a explorar un ejemplo usando el middleware `can` para autorizar que un usuario pueda actualizar un post de un blog:

```
use App\Post;                                         php

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

En este ejemplo, estamos pasando al middleware `can` dos argumentos, el primero es el nombre de la acción que deseamos autorizar y el segundo es el parámetro de la ruta que deseamos pasar al método de la política. En este caso, como estamos usando [implicit model binding](#), un modelo `Post` será pasado al método de la política. Si el usuario no está autorizado a ejecutar la acción dada, el middleware generará una respuesta HTTP con el código de estatus `403`.

Acciones que no requieren modelos

Como mencionamos antes, algunas acciones como `create` pueden no requerir de una instancia de un modelo. En estas situaciones, puedes pasar el nombre de la clase al middleware. El nombre de la clase será usado para determinar cuál política usar para autorizar la acción:

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Post');
```

Vía helpers de controladores

Además de proveer métodos útiles en el modelo `User`, Laravel también provee un método muy útil llamado `authorize` en cualquier controlador que extienda la clase base

`App\Http\Controllers\Controller`. Como el método `can`, este método acepta el nombre de la acción que quieras autorizar y el modelo relevante. Si la acción no es autorizada, el método `authorize` arrojará una excepción de tipo

`\Illuminate\Auth\Access\AuthorizationException`, la cual será convertida por el manejador de excepciones por defecto de Laravel en una respuesta HTTP con un código `403` :

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use App\Post;  
use Illuminate\Http\Request;  
  
class PostController extends Controller  
{  
    /**  
     * Update the given blog post.  
     *  
     * @param Request $request  
     * @param Post $post  
     * @return Response  
     * @throws \Illuminate\Auth\Access\AuthorizationException  
     */  
    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 hemos discutido previamente, algunas acciones como `create` pueden no requerir una instancia de un modelo. En estas situaciones, deberías pasar el nombre de la clase al método `authorize`. El nombre de la clase determinará la política a usar para autorizar la acción:

```
/**  
 * Create a new blog post.  
 *  
 * @param Request $request  
 * @return Response  
 * @throws \Illuminate\Auth\Access\AuthorizationException
```

```
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}
```

Autorizando controladores de recursos

Si estás utilizando [controladores de recursos](#), puedes hacer uso del método `authorizeResource` en el constructor del controlador. Este método agregará las definiciones de middleware `can` apropiadas a los métodos del controlador de recursos.

El método `authorizeResource` acepta el nombre de clase del modelo como primer argumento y el nombre del parámetro de ruta / petición que contendrá el ID del modelo como segundo argumento:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    public function __construct()
    {
        $this->authorizeResource(Post::class, 'post');
    }
}
```

php

Los siguientes métodos de controlador serán mapeados con su método de política respectivo:

Método de controlador	Método de política
index	viewAny
show	view

Método de controlador	Método de política
create	create
store	create
edit	update
update	update
destroy	delete

TIP

Puedes usar el comando `make:policy` con la opción `--model` para generar rápidamente una clase de política para un modelo dado: `php artisan make:policy PostPolicy --model=Post`.

Vía plantillas de blade

Cuando esribas plantillas de Blade, puedes querer mostrar una porción de la página solo si el usuario está autorizado para ejecutar una acción determinada. Por ejemplo, puedes querer mostrar un formulario para actualizar un post solo si el usuario puede actualizar el post. En situaciones así, puedes usar las directivas `@can` y `@cannot`:

```
php
@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 accesos directos convenientes para no tener que escribir sentencias `@if` y `@unless`. Las sentencias `@can` y `@cannot` de arriba son equivalentes a las siguientes sentencias, respectivamente:

```
@if (Auth::user()->can('update', $post))
```

```
    <!-- The Current User Can Update The Post -->
```

```
@endif
```

php

```
@unless (Auth::user()->can('update', $post))
```

```
    <!-- The Current User Can't Update The Post -->
```

```
@endunless
```

También puedes determinar si un usuario tiene habilidad de autorización desde una lista de habilidades dadas. Para lograr esto, usa la directiva `@canany`:

```
@canany(['update', 'view', 'delete'], $post)
```

```
    // The current user can update, view, or delete the post
```

```
@elsecanany(['create'], \App\Post::class)
```

```
    // The current user can create a post
```

```
@endcanany
```

php

Acciones que no requieren modelos

Así como otros métodos de autorización, puedes pasar el nombre de una clase a las directivas `@can` y `@cannot` si la acción no requiere una instancia de un modelo:

```
@can('create', App\Post::class)
```

```
    <!-- The Current User Can Create Posts -->
```

```
@endcan
```

php

```
@cannot('create', App\Post::class)
```

```
    <!-- The Current User Can't Create Posts -->
```

```
@endcannot
```

Proporcionando contexto adicional

Al autorizar acciones usando políticas, puedes pasar un arreglo como segundo argumento a las diferentes funciones y helpers de autorización. El primer elemento en el arreglo será usado para

determinar que política debe ser invocada, mientras que el resto de los elementos del arreglo son pasados como parametros al método de la política y puede ser usado como contexto adicional al tomar decisiones de autorización. Por ejemplo, considera la siguiente definición de método `PostPolicy` que contiene un parametro adicional `$category` :

```
/*
 * Determine if the given post can be updated by the user.
 *
 * @param \App\User $user
 * @param \App\Post $post
 * @param int $category
 * @return bool
 */
public function update(User $user, Post $post, int $category)
{
    return $user->id === $post->user_id &&
           $category > 3;
}
```

php

Al intentar determinar si el usuario autenticado puede actualizar un post dado, podemos invocar este método de política de la siguiente manera:

```
/*
 * Update the given blog post.
 *
 * @param Request $request
 * @param Post $post
 * @return Response
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post)
{
    $this->authorize('update', [$post, $request->input('category')]);

    // The current user can update the blog post...
}
```

php

Verificación de Correo Electrónico

- Introducción
- Consideraciones de la base De datos
- Rutas
 - Protegiendo rutas
- Vistas
- Luego de verificar correos electrónicos
- Eventos

Introducción

Muchas aplicaciones web requieren que los usuarios verifiquen sus correos electrónicos usando la aplicación. En lugar de forzarte a volver a implementar esto en cada aplicación, Laravel proporciona métodos convenientes para enviar y verificar solicitudes de verificación de correos electrónicos.

Preparación del modelo

Para comenzar, verifica que tu modelo `App\User` implementa la interfaz

`Illuminate\Contracts\Auth\MustVerifyEmail` :

```
<?php  
  
namespace App;  
  
use Illuminate\Contracts\Auth\MustVerifyEmail;  
use Illuminate\Foundation\Auth\User as Authenticatable;  
use Illuminate\Notifications\Notifiable;  
  
class User extends Authenticatable implements MustVerifyEmail  
{  
    use Notifiable;
```

```
// ...  
}
```

Consideraciones de la base de datos

Columna de verificación de correo electrónico

Luego, tu tabla `user` debe contener una columna `email_verified_at` para almacenar la fecha y la hora en la que la dirección de correo electrónico fue verificada. Por defecto, la migración de la tabla `user` incluida con el framework Laravel ya incluye esta columna. Así que, lo único que necesitas es ejecutar la migración de la base de datos:

```
php artisan migrate
```

php

Rutas

Laravel incluye la clase `Auth\VerificationController` que contiene la lógica necesaria para enviar enlaces de verificación y verificar correos electrónicos. Para registrar las rutas necesarias para este controlador, pasa la opción `verify` al método `Auth::routes` :

```
Auth::routes(['verify' => true]);
```

php

Protegiendo rutas

El middleware de rutas puede ser usado para permitir que sólo usuarios autorizados puedan acceder a una ruta dada. Laravel viene con un middleware `verified`, que está definido en `Illuminate\Auth\Middleware\EnsureEmailIsVerified`. Dado que este middleware ya está registrado en el kernel HTTP de tu aplicación, lo único que necesitas hacer es adjuntar el middleware a una definición de ruta:

```
Route::get('profile', function () {  
    // Only verified users may enter...  
})->middleware('verified');
```

php

Vistas

Para generar todas las vistas necesarias para la verificación de correo electrónico, puedes usar el paquete `laravel/ui` de Composer:

```
composer require laravel/ui --dev  
php artisan ui vue --auth
```

php

La vista de verificación de correo electrónico es colocada en

`resources/views/auth/verify.blade.php`. Eres libre de personalizar esta vista según sea necesario para tu aplicación.

Luego de verificar correos electrónicos

Luego de que una dirección de correo electrónico es verificada, el usuario será redirigido automáticamente a `/home`. Puedes personalizar la ubicación de redirección post-verificación definiendo un método `redirectTo` o propiedad en `VerificationController`:

```
protected $redirectTo = '/dashboard';
```

php

Eventos

Laravel despacha eventos durante el proceso de verificación de correo electrónico. Puedes agregar listeners a estos eventos en tu `EventServiceProvider`:

```
/**  
 * The event listener mappings for the application.  
 *  
 * @var array  
 */  
protected $listen = [  
    'Illuminate\Auth\Events\Verified' => [  
        'App\Listeners\LogVerifiedUser',  
    ],  
];
```

php

Cifrado

- [Introducción](#)
- [Configuración](#)
- [Usando el cifrador](#)

Introducción

El cifrado de Laravel utiliza OpenSSL para proporcionar el cifrado AES-256 y AES-128. Se recomienda encarecidamente usar las funciones de cifrado incorporadas de Laravel y no intente desplegar tus algoritmos de cifrado "de cosecha propia". Todos los valores cifrados de Laravel son firmados utilizando un código de autenticación de mensaje (MAC) para que su valor subyacente no pueda modificarse una vez cifrado.

Configuración

Antes de usar el cifrado de Laravel, debes establecer la opción `key` en tu archivo de configuración `config/app.php`. Deberías usar el comando `php artisan key: generate` para generar esta clave, ya que este comando de Artisan usará el generador de bytes aleatorios seguros de PHP para construir tu clave. Si este valor no se establece correctamente, todos los valores cifrados por Laravel serán inseguros.

Usando el cifrador

Cifrar un valor

Puedes cifrar un valor usando el helper o función de ayuda `encrypt`. Todos los valores cifrados se cifran utilizando OpenSSL y el cifrado `AES-256-CBC`. Además, todos los valores cifrados están

firmados con un código de autenticación de mensaje (MAC) para detectar cualquier modificación en 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();  
    }  
}
```

Cifrado sin serialización

Los valores cifrados se pasan a través de una serialización durante el proceso de cifrado, lo que permite el cifrado de objetos y matrices. De este modo, los clientes que no son PHP y reciben valores cifrados tendrán que des-serializar los datos. Si deseas cifrar y descifrar valores sin serialización, puede usar los métodos `encryptString` y `decryptString` de la facade `Crypt`:

```
use Illuminate\Support\Facades\Crypt;  
  
$encrypted = Crypt::encryptString('Hello world.');
```

```
$decrypted = Crypt::decryptString($encrypted);
```

Descifrando un valor

Puedes descifrar los valores usando el helper o función de ayuda `decrypt`. Si el valor no se puede descifrar correctamente, como cuando el MAC no es válido, se lanzará una

`Illuminate\Contracts\Encryption\DecryptException` :

```
use Illuminate\Contracts\Encryption\DecryptException;          php

try {
    $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
    //
}
```

Hashing

- [Introducción](#)
- [Configuración](#)
- [Uso básico](#)

Introducción

El facade `Hash` de Laravel proporciona hashing seguro de Bcrypt y Argon2 para almacenar contraseñas de usuarios. Si estás usando las clases integradas `LoginController` y `RegisterController` que están incluidas con tu aplicación de Laravel usarán Bcrypt para registro y autenticación de forma predeterminada.

TIP

Bcrypt es una buena opción para el hashing de contraseñas dado que su "factor de trabajo" es ajustable, lo que quiere decir que el tiempo que toma generar un hash puede ser aumentado a medida que la capacidad de hardware incrementa.

Configuración

El driver de hashing por defecto para tu aplicación está configurado en el archivo de configuración `config/hashing.php`. Actualmente hay tres drivers soportados: [Bcrypt](#) y [Argon2](#) (variantes Argon2i y Argon2id).

Nota

El driver Argon2i requiere PHP 7.2.0 o superior y el driver Argon2id requiere PHP 7.3.0 o superior.

Uso básico

Puedes hacer hash a una contraseña llamando al método `make` en el 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...
```

php

```
$request->user()->fill([
    'password' => Hash::make($request->newPassword)
])->save();
}

}
```

Ajustando el factor de trabajo de Bcrypt

Si estás usando el algoritmo Bcrypt, el método `make` te permite administrar el factor de trabajo del algoritmo usando la opción `rounds`; sin embargo, el valor por defecto es aceptable para la mayoría de las aplicaciones:

```
$hashed = Hash::make('password', [
    'rounds' => 12
]);
```

Ajustando el factor de trabajo de Argon2

Si estás usando el algoritmo de Argon2, el método `make` te permite administrar la carga de trabajo del algoritmo usando las opciones `memory`, `time` y `threads`; sin embargo, los valores por defecto son aceptables para la mayoría de las aplicaciones:

```
$hashed = Hash::make('password', [
    'memory' => 1024,
    'time' => 2,
    'threads' => 2,
]);
```

TIP

Para mayor información de estas opciones, revisa la [documentación oficial de PHP](#).

Verificando una contraseña contra un hash

El método `check` te permite verificar que una cadena de texto plano dada corresponde a un hash dado. Sin embargo, si estás usando el [LoginController](#) incluido con Laravel, probablemente no necesitarás usar esto directamente, ya que este controlador automáticamente llama a este método:

```
if (Hash::check('plain-text', $hashedPassword)) {  
    // Las contraseñas coinciden...  
}
```

php

Comprobando si una contraseña necesita un nuevo hash

La función `needsRehash` te permite determinar si el factor de trabajo usado por el hasher ha cambiado desde que el hash fue agregado a la contraseña:

```
if (Hash::needsRehash($hashed)) {  
    $hashed = Hash::make('plain-text');  
}
```

php

Restablecimiento de contraseñas

- Introducción
- Consideraciones de la base de datos
- Enrutamiento
- Vistas
- Después de restablecer contraseñas
- Personalización

Introducción

TIP

¿Quieres comenzar rápido? Instala el paquete de composer `laravel/ui` y ejecuta `php artisan ui vue --auth` en una aplicación de Laravel nueva. Luego de migrar tu base de

datos, navega hasta `http://your-app.test/register` o cualquier otra URL asignada a tu aplicación. Este simple comando se encargará de maquetar todo tu sistema de autenticación, incluyendo el restablecimiento de contraseñas!

La mayoría de las aplicaciones web proporciona una forma para que los usuarios restablecen sus contraseñas olvidadas. En lugar de forzarte a reimplementar esto en cada aplicación, Laravel proporciona métodos convenientes para enviar recordatorios de contraseñas y realizar restablecimientos de contraseñas.

TIP

Antes de usar las características de restablecimiento de contraseñas de Laravel, tu usuario debe usar el trait `Illuminate\Notifications\Notifiable`.

Consideraciones de la base de datos

Para comenzar, verifica que tu modelo `App\User` implementa la interfaz `Illuminate\Contracts\Auth\CanResetPassword`. El modelo `App\User` incluido con el framework ya implementa esta interfaz y usa el trait `Illuminate\Auth\Passwords\CanResetPassword` para incluir los métodos necesarios para implementar la interfaz.

Generando la migración para la tabla de tokens de restablecimiento

Luego, una tabla debe ser creada para almacenar los tokens de restablecimiento de contraseña. La migración para esta tabla está incluida con Laravel por defecto y se encuentra en el directorio `database/migrations`. Así que, todo lo que necesitas hacer es ejecutar tus migraciones de la base de datos:

```
php artisan migrate
```

php

Enrutamiento

Laravel incluye las clases `Auth\ForgotPasswordController` y `Auth\ResetPasswordController` que contienen la lógica necesaria para enviar enlaces de restablecimiento de contraseña y restablece

contraseñas de usuarios mediante correo electrónico. Todas las rutas necesarias para realizar restablecimiento de contraseñas pueden ser generadas usando el paquete de Composer: `laravel/ui`

```
composer require laravel/ui --dev  
php artisan ui vue --auth
```

php

Vistas

Para generar todas las vistas necesarias para el restablecimiento de contraseñas, puedes usar el paquete de Composer: `laravel/ui`

```
composer require laravel/ui --dev  
php artisan ui vue --auth
```

php

Estas vistas están ubicadas en `resources/views/auth/passwords`. Eres libre de personalizarlas según sea necesario para tu aplicación.

Luego de resetear contraseñas

Una vez que has definido las rutas y vistas para restablecer las contraseñas de tus usuarios, puedes acceder a la ruta en tu navegador en `/password/reset`. El `ForgotPasswordController` incluido con el framework ya incluye la lógica para enviar los correos con el enlace de restablecimiento, mientras que `ResetPasswordController` incluye la lógica para restablecer las contraseñas de los usuarios.

Luego de que una contraseña es restablecida, la sesión del usuario será automáticamente iniciada y será redirigido a `/home`. Puedes personalizar la ubicación de redirección definiendo una propiedad `redirectTo` en `ResetPasswordController`:

```
protected $redirectTo = '/dashboard';
```

php

Nota

Por defecto, los tokens para restablecer contraseñas expiran luego de una hora. Puedes cambiar esto mediante la opción de restablecimiento de contraseñas `expire` en tu archivo `config/auth.php`.

Personalización

Personalización de los guards de autenticación

En tu archivo de configuración `auth.php`, puedes configurar múltiples "guards", que podrán ser usados para definir el comportamiento de autenticación para múltiples tablas de usuarios. Puedes personalizar el controlador `ResetPasswordController` incluido para usar el guard de tu preferencia sobrescribiendo el método `guard` en el controlador. Este método debe retornar una instancia guard:

```
use Illuminate\Support\Facades\Auth;                                         php

/**
 * Get the guard to be used during password reset.
 *
 * @return \Illuminate\Contracts\Auth\StatefulGuard
 */

protected function guard()
{
    return Auth::guard('guard-name');
}
```

Personalización del broker de contraseña

En tu archivo de configuración `auth.php`, puedes configurar múltiples "brokers" de contraseñas, que pueden ser usados para restablecer contraseñas en múltiples tablas de usuarios. Puedes personalizar los controladores `ForgotPasswordController` y `ResetPasswordController` incluidos para usar el broker de tu elección sobrescribiendo el método `broker`:

```
use Illuminate\Support\Facades\Password;                                     php

/**
 * Get the broker to be used during password reset.
 *
```

```
* @return PasswordBroker
*/
protected function broker()
{
    return Password::broker('name');
}
```

Personalización del correo de reseteo

Puedes fácilmente modificar la clase de la notificación usada para enviar el enlace de restablecimiento de contraseña al usuario. Para comenzar, sobrescribe el método `sendPasswordResetNotification` en tu modelo `User`. Dentro de este método, puedes enviar la notificación usando cualquier clase que selecciones. El `$token` de restablecimiento de contraseña es el primer argumento recibido por el método:

```
/**
 * Send the password reset notification.
 *
 * @param string $token
 * @return void
 */
public function sendPasswordResetNotification($token)
{
    $this->notify(new ResetPasswordNotification($token));
}
```

php

Consola artisan

- Introducción
 - Tinker (REPL)
- Escritura de comandos

- Generación de comandos
- Estructura de un comando
- Comandos de función anónima (closure)
- Definición de expectativas de entrada
 - Argumentos
 - Opciones
 - Arreglos como entradas
 - Descripciones de entrada
- Entrada/Salida de comandos
 - Recuperación de entradas
 - Solicitud de entradas
 - Escritura de salida
- Registro de comandos
- Ejecución de comandos de forma programática
 - Llamando comandos desde otros comandos

Introducción

Artisan es la interfaz de línea de comando incluida con Laravel. Provee un número de comandos útiles que pueden ayudarte mientras construyes tu aplicación. Para ver una lista de todos los comandos Artisan disponibles, puedes usar el comando `list` :

```
php artisan list
```

php

También cada comando incluye una "ayuda" en pantalla, la cual muestra y describe los argumentos y opciones disponibles. Para ver una pantalla de ayuda, coloca `help` antes del nombre del comando:

```
php artisan help migrate
```

php

Tinker REPL

Laravel Tinker es un poderoso REPL para el framework Laravel, desarrollado usando el paquete [PsySH](#).

Instalación

Todas las aplicaciones de Laravel incluyen Tinker por defecto. Sin embargo, de ser necesario puedes instalarlo manualmente usando Composer:

```
composer require laravel/tinker
```

php

Uso

Tinker te permite interactuar con toda tu aplicación de Laravel en la línea de comandos, incluyendo el ORM Eloquent, trabajos, eventos y más. Para ingresar al entorno de Tinker, ejecuta el comando de Artisan `Tinker`:

```
php artisan tinker
```

php

Puedes publicar el archivo de configuración de Tinker usando el comando `vendor:publish`:

```
php artisan vendor:publish --provider="Laravel\\Tinker\\TinkerServiceProvider"
```

php

Nota

La función helper `dispatch` y el método `dispatch` en la clase `Dispatchable` dependen de garbage collection para colocar el trabajo en la cola. Por lo tanto, al usar tinker, debes usar `Bus::dispatch` o `Queue::push` para despachar trabajos.

Lista blanca de comandos

Tinker utiliza una lista blanca para determinar qué comandos de Artisan pueden ejecutarse dentro de su shell. Por defecto, puedes ejecutar los comandos `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `optimize` y `up`. Si deseas hacer una lista blanca de más comandos, puedes agregarlos al arreglo `command` en tu archivo de configuración `tinker.php`:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

php

Lista negra de alias

Por lo general, Tinker automáticamente asigna alias a las clases según las necesites en Tinker. Sin embargo, es posible que deseas que nunca se agreguen alias a algunas clases. Puedes lograr esto listando las clases en el arreglo `dont_alias` de tu archivo de configuración `tinker.php`:

```
'dont_alias' => [
    App\User::class,
],
```

php

Escritura de comandos

Además de los comandos proporcionados por Artisan, también puedes crear tus propios comandos personalizados. Los comandos son típicamente almacenados en el directorio

`app/Console/Commands`; sin embargo, eres libre de escoger tu propia ubicación de almacenamiento, siempre y cuando tus comandos puedan ser cargados por Composer.

Generación de comandos

Para crear un nuevo comando, usa el comando Artisan `make:command`. Este comando creará una nueva clase de comando en el directorio `app/Console/Commands`. No te preocupes si este directorio no existe en tu aplicación, pues éste será creado la primera vez que ejecutes el comando Artisan `make:command`. El comando generado incluirá el conjunto de propiedades y métodos por defecto que están presentes en todos los comandos:

```
php artisan make:command SendEmails
```

php

Estructura de un comando

Después de generar tu comando, debes llenar las propiedades `signature` y `description` de la clase, las cuales serán usadas cuando se muestra tu comando en la pantalla `list`. El método `handle` será llamado cuando tu comando es ejecutado. Puedes colocar tu lógica del comando en este método.

TIP

Para una mayor reutilización del código, es una buena práctica mantener ligeros tus comandos de consola y dejar que se remitan a los servicios de aplicaciones para llevar a cabo sus tareas. En el siguiente ejemplo, toma en cuenta que inyectamos una clase de servicio para hacer el "trabajo pesado" de enviar los correos electrónicos.

Echemos un vistazo a un comando de ejemplo. Observa que podemos inyectar cualquier dependencia que necesitemos en el método `handle()` del comando. El [contenedor de servicios](#) de Laravel automáticamente inyectará todas las dependencias cuyos tipos (interfaces y/o clases) estén asignados en los parámetros del constructor (type-hinting):

```
<?php  
  
namespace App\Console\Commands;  
  
use App\DripEmailer;  
use App\User;  
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';  
  
    /**  
     * Create a new command instance.  
     *  
     * @return void  
     */  
    public function __construct()  
    {
```

```

        parent::__construct();
    }

    /**
     * Execute the console command.
     *
     * @param \App\DripEmailer $drip
     * @return mixed
     */
    public function handle(DripEmailer $drip)
    {
        $drip->send(User::find($this->argument('user')));
    }
}

```

Comandos usando una función anónima (closure)

Los comandos basados en Closure proporcionan una alternativa para definir comandos de consola como clases. De la misma manera que los Closures de rutas son una alternativa para los controladores, piensa en los Closures de comandos como una alternativa a las clases de comandos. Dentro del método `commands` de tu archivo `app/Console/Kernel.php`, Laravel carga el archivo `routes/console.php`:

```

/**
 * Register the Closure based commands for the application.
 *
 * @return void
 */
protected function commands()
{
    require base_path('routes/console.php');
}

```

Aunque este archivo no define rutas HTTP, define los puntos de entrada (rutas) a tu aplicación basados en consola. Dentro de este archivo, puedes definir todas sus rutas basadas en Closure usando el método `Artisan::command`. El método `command` acepta dos argumentos: la [firma del comando](#) y un Closure, el cual recibe los argumentos y opciones de los comandos:

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
});
```

php

El Closure está vinculado a la instancia del comando subyacente, así tienes acceso completo a todos los métodos helper a los que normalmente podrías acceder en una clase de comando completa.

Determinación de tipos (type-hinting) de dependencias

Además de recibir los argumentos y opciones de tu comando, en los Closures de comandos puedes también determinar los tipos de las dependencias adicionales que te gustaría resolver del [contenedor de servicios](#):

```
use App\DripEmailer;
use App\User;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

php

Descripciones de un closure de comando

Al definir un comando basado en Closure, puedes usar el método `describe` para agregar una descripción al comando. Esta descripción será mostrada cuando ejecutes los comandos `php artisan list` o `php artisan help`:

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

php

Definición de expectativas de entrada

Al escribir comandos de consola, es común recopilar información del usuario a través de argumentos u opciones. Laravel hace que sea muy conveniente definir la entrada que esperas del usuario usando la propiedad `signature` en tus comandos. La propiedad `signature` te permite definir el nombre, los argumentos y las opciones para el comando en una sintaxis tipo ruta simple y expresiva.

Argumentos

Todos los argumentos y opciones suministrados por el usuario están envueltos en llaves. En el siguiente ejemplo, el comando define un argumento **obligatorio** `user` :

```
/** php
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user}';
```

También puedes hacer que los argumentos sean opcionales y definir valores predeterminados para los argumentos:

```
// Optional argument...
email:send {user?}

// Optional argument with default value...
email:send {user=foo}
```

Opciones

Las opciones, como los argumentos, son otra forma de entrada de usuario. Las opciones son prefijadas por dos guiones (`--`) cuando se especifican en la línea de comando. Hay dos tipos de opciones: aquellas que reciben un valor y las que no. Las opciones que no reciban un valor se comportarán como un "interruptor" booleano. Echemos un vistazo a un ejemplo de este tipo de opción:

```
/** php
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue}';
```

En este ejemplo, la opción `--queue` puede ser especificada cuando ejecutas el comando Artisan. Si la opción `--queue` es pasada, su valor será `true`. En caso contrario, el valor será `false` :

```
php artisan email:send 1 --queue
```

php

Opciones con valores

Vamos a ver una opción que espera un valor. Si el usuario debe especificar un valor para una opción, agrega como sufijo el signo `=` al nombre de la opción:

```
/**  
 * The name and signature of the console command.  
 *  
 * @var string  
 */  
protected $signature = 'email:send {user} {--queue=}';
```

php

En este ejemplo, el usuario puede pasar un valor para la opción, de esta manera:

```
php artisan email:send 1 --queue=default
```

php

Puedes asignar valores por defecto a las opciones especificando el valor predeterminado después del nombre de la opción. Si ningún valor es pasado por el usuario, el valor por defecto será usado:

```
email:send {user} {--queue=default}
```

php

Atajos de opciones

Para asignar un atajo cuando defines una opción, puedes especificarlo antes del nombre de la opción y usar un delimitador `|` para separar el atajo del nombre completo de la opción:

```
email:send {user} {--Q|queue}
```

php

Arreglos como entradas

Si deseas definir argumentos u opciones para esperar entradas de arreglos, puedes usar el carácter `*`. Primero, vamos a ver un ejemplo que especifica un arreglo como argumento:

```
email:send {user*}
```

php

Al llamar a este método, los argumentos `user` pueden pasarse en orden a la línea de comando. Por ejemplo, el siguiente comando establecerá el valor de `user` como `['foo', 'bar']`:

```
php artisan email:send foo bar
```

php

Al definir una opción que espera un arreglo como entrada, cada valor de la opción pasado al comando debería ser prefijado con el nombre de la opción:

```
email:send {user} {--id=*}
```

php

```
php artisan email:send --id=1 --id=2
```

Descripciones de entrada

Puedes asignar descripciones para los argumentos y opciones de entrada separando el parámetro de la opción usando dos puntos `:`. Si necesitas un poco más de espacio para definir tu comando, no dudes en extender la definición a través de múltiples 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}';
```

php

Entrada y salida (E/S) de comandos

Recuperación de entrada

Cuando tu comando es ejecutado, obviamente necesitarás acceder a los valores de los argumentos y opciones aceptados por tu comando. Para ello, puedes usar los métodos `argument` y `option`:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

php

Si necesitas recuperar todos los argumentos como un arreglo, llama al método `arguments` :

```
$arguments = $this->arguments();
```

php

Las opciones pueden ser recuperadas con tanta facilidad como argumentos utilizando el método `option`. Para recuperar todas las opciones como un arreglo, llama al método `options` :

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options...
$options = $this->options();
```

php

Si el argumento o la opción no existe, será retornado `null`.

Solicitud de entrada

Además de mostrar salidas, puedes también pedir al usuario que proporcione información durante la ejecución del comando. El método `ask` le indicará al usuario la pregunta dada, aceptará su entrada, y luego devolverá la entrada del usuario a tu comando:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
```

php

```
public function handle()
{
    $name = $this->ask('What is your name?');
}
```

El método `secret` es similar a `ask`, pero la entrada del usuario no será visible para ellos cuando la escriban en la consola. Este método es útil cuando se solicita información confidencial tal como una contraseña:

```
$password = $this->secret('What is the password?');
```

php

Pedir confirmación

Si necesitas pedirle al usuario una simple confirmación, puedes usar el método `confirm`. Por defecto, este método devolverá `false`. Sin embargo, si el usuario ingresa `y` o `yes` en respuesta a la solicitud, el método devolverá `true`.

```
if ($this->confirm('Do you wish to continue?')) {
    //
}
```

php

Autocompletado

El método `anticipate` puede ser usado para proporcionar autocompletado para posibles opciones. El usuario aún puede elegir cualquier respuesta, independientemente de las sugerencias de autocompletado:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

php

Alternativamente, puedes pasar una Closure como segundo argumento del método `anticipate`. La Closure será llamada cada vez que el usuario ingrese un carácter. La Closure debe aceptar una cadena como parámetro que contenga el texto ingresado por el usuario y retornar un arreglo de opciones de auto-completado:

```
$name = $this->anticipate('What is your name?', function ($input) {
    // Return auto-completion options...
});
```

php

```
});
```

Preguntas de selección múltiple

Si necesitas darle al usuario un conjunto de opciones predefinadas, puedes usar el método `choice`.

Puedes establecer el índice del valor predeterminado del arreglo que se devolverá si no se escoge ninguna opción:

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $defaultIndex);
```

Adicionalmente, el método `choice` acepta un cuarto y quinto argumento opcional para determinar el maximo número de intentos para seleccionar una respuesta valida y si múltiples selecciones están permitidas:

```
$name = $this->choice(  
    'What is your name?',  
    ['Taylor', 'Dayle'],  
    $defaultIndex,  
    $maxAttempts = null,  
    $allowMultipleSelections = false  
);
```

Escritura de salida

Para enviar datos de salida a la consola, usa los métodos `line`, `info`, `comment`, `question` y `error`. Cada uno de estos métodos usará colores ANSI apropiados para su propósito. Por ejemplo, vamos a mostrar alguna información general al usuario. Normalmente, el método `info` se mostrará en la consola como texto verde:

```
/**  
 * Execute the console command.  
 *  
 * @return mixed  
 */  
public function handle()  
{
```

```
$this->info('Display this on the screen');  
}
```

Para mostrar un mensaje de error, usa el método `error`. El texto del mensaje de error es típicamente mostrado en rojo:

```
$this->error('Something went wrong!');
```

php

Si desea mostrar la salida de consola sin color, usa el método `line`:

```
$this->line('Display this on the screen');
```

php

Diseños de tabla

El método `table` hace que sea fácil formatear correctamente varias filas / columnas de datos. Simplemente pasa los encabezados y filas al método. El ancho y la altura se calcularán dinámicamente en función de los datos dados:

```
$headers = ['Name', 'Email'];  
  
$users = App\User::all(['name', 'email'])->toArray();  
  
$this->table($headers, $users);
```

php

Barras de progreso

Para tareas de larga ejecución, podría ser útil mostrar un indicador de progreso. Usando el objeto de salida, podemos iniciar, avanzar y detener la Barra de Progreso. Primero, define el número total de pasos por los que el proceso pasará. Luego, avanza la barra de progreso después de procesar cada elemento:

```
$users = App\User::all();  
  
$bar = $this->output->createProgressBar(count($users));  
  
$bar->start();  
  
foreach ($users as $user) {
```

php

```
$this->performTask($user);

$bar->advance();
}

$bar->finish();
```

Para opciones más avanzadas, verifica la [documentación del componente Progress Bar de Symfony](#).

Registro de comandos

Debido a la llamada al método `load` en el método `commands` del kernel de tu consola, todos los comandos dentro del directorio `app/Console/Commands` se registrarán automáticamente con Artisan. De hecho, puedes realizar llamadas adicionales al método `load` para escanear otros directorios en busca de comandos Artisan:

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

    // ...
}
```

También puedes registrar comandos manualmente agregando su nombre de clase en la propiedad `$commands` de tu archivo `app/Console/Kernel.php`. Cuando Artisan arranca, todos los comandos listados en esta propiedad serán resueltos por el [contenedor de servicios](#) y serán registrados con Artisan:

```
protected $commands = [
    Commands\SendEmails::class
];
```

Ejecución de comandos de forma programática

En ocasiones, es posible que deseas ejecutar un comando de Artisan fuera de la interfaz de línea de comandos (CLI). Por ejemplo, puedes desear ejecutar un comando Artisan de una ruta o controlador. Puedes usar el método `call` en el facade `Artisan` para lograr esto. El método `call` acepta el nombre o la clase del comando como primer argumento y un arreglo de parámetros del comando como segundo argumento. El código de salida será devuelto:

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});

});
```

php

Alternativamente, puedes pasar todo el comando de Artisan para el metodo `call` como una cadena:

```
Artisan::call('email:send 1 --queue=default');
```

php

Usando el método `queue` en el facade `Artisan`, puedes incluso poner en cola comandos Artisan para ser procesados en segundo plano por tus [queue workers](#). Antes de usar este método, asegurate que tengas configurado tu cola y se esté ejecutando un oyente de cola (queue listener):

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});

});
```

php

También puedes especificar la conexión o cola a la que debes enviar el comando Artisan:

```
Artisan::queue('email:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

php

Pasando valores de tipo arreglo

Si tu comando define una opción que acepta un arreglo, puedes pasar un arreglo de valores a la opción:

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--id' => [5, 13]
    ]);
});
```

php

Pasando valores booleanos

Si necesitas especificar el valor de una opción que no acepta valores de tipo cadena, tal como la opción `--force` en el comando `migrate:refresh`, debes pasar `true` o `false`:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

php

Llamando comandos desde otros comandos

Algunas veces puedes desear llamar otros comandos desde un comando Artisan existente. Puedes hacerlo usando el método `call`. Este método acepta el nombre del comando y un arreglo de parámetros del comando:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
}
```

php

Si deseas llamar a otro comando de consola y eliminar toda su salida, puedes usar el método

`callSilent` . Este método tiene la misma firma que el método `call` :

```
$this->callSilent('email:send', [  
    'user' => 1, '--queue' => 'default'  
]);
```

php

Broadcasting

- Introducción
 - Configuración
 - Prerrequisitos del driver
- Descripción general
 - Usando una aplicación de ejemplo
- Definiendo eventos de transmisión
 - Nombre de la transmisión
 - Datos de la transmisión
 - Cola de la transmisión
 - Condiciones de la transmisión
- Autorizando canales
 - Definiendo rutas de autorización
 - Definiendo callbacks de autorización
 - Definiendo clases de canales
- Transmitiendo eventos
 - Sólo a otros
- Recibiendo transmisiones
 - Instalando Laravel Echo
 - Escuchando eventos

- Dejando un canal
- Nombres de espacio
- Canales de presencia
 - Autorizando canales de presencia
 - Uniéndose a canales de presencia
 - Transmitiendo a canales de presencia
- Eventos del cliente
- Notificaciones

Introducción

En muchas aplicaciones web modernas, los WebSockets son usados para implementar interfaces de usuarios actualizadas en tiempo real. Cuando algún dato es actualizado en el servidor, un mensaje es típicamente enviado a través de una conexión WebSocket para ser manejado por el cliente. Esto proporciona una alternativa más robusta y eficiente para monitorear continuamente tu aplicación en busca de cambios.

Para asistirte en la construcción de ese tipo de aplicaciones, Laravel hace fácil "emitir" tus eventos a través de una conexión WebSocket. Emitir tus eventos te permite compartir los mismos nombres de eventos entre tu código del lado del servidor y tu aplicación JavaScript del lado de cliente.

TIP

Antes de sumergirnos en la emisión de eventos, asegurate de haber leído toda la documentación de Laravel sobre eventos y listeners.

Configuración

Toda la configuración de transmisión de eventos de tu aplicación está almacenada en el archivo de configuración `config/broadcasting.php`. Laravel soporta múltiples drivers de transmisión: [canales de Pusher](#), [Redis](#) y un driver `log` para desarrollo local y depuración. Adicionalmente, un driver `null` es incluido, que te permite deshabilitar totalmente las emisiones. Un ejemplo de configuración para cada uno de los drivers está incluido en el archivo de configuración `config/broadcasting.php`.

Proveedor de servicios Broadcast

Antes de transmitir cualquier evento, necesitarás primero registrar

`App\Providers\BroadcastServiceProvider`. En aplicaciones de Laravel nuevas, sólo necesitas descomentar este proveedor en el arreglo `providers` de tu archivo de configuración `config/app.php`. Este proveedor te permitirá registrar las rutas de autorización del broadcast y los callbacks.

Token CSRF

Laravel Echo necesitará acceso al token CSRF de la sesión actual. Debes verificar que el elemento HTML `head` de tu aplicación define una etiqueta `meta` que contiene el token CSRF:

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

html

Prerrequisitos del driver

Canales de Pusher

Si estás transmitiendo tus eventos mediante [canales de Pusher](#), debes instalar el SDK de PHP para canales de Pusher mediante el administrador de paquetes Composer:

```
composer require pusher/pusher-php-server "~4.0"
```

sh

Luego, debes configurar tus credenciales del canal en el archivo de configuración

`config/broadcasting.php`. Un ejemplo de configuración de canal está incluido en este archivo, permitiéndote especificar rápidamente la clave del canal, contraseña y ID de la aplicación. La configuración de `pusher` del archivo `config/broadcasting.php` también te permite especificar `options` adicionales que son soportadas por canales, como el cluster:

```
'options' => [
    'cluster' => 'eu',
    'useTLS' => true
],
```

php

Al usar canales y [Laravel Echo](#), debes especificar `pusher` como tu transmisor deseado al instanciar la instancia de Echo en tu archivo `resources/js/bootstrap.js`:

```
import Echo from "laravel-echo";  
  
window.Pusher = require('pusher-js');  
  
window.Echo = new Echo({  
    broadcaster: 'pusher',  
    key: 'your-pusher-channels-key'  
});
```

php

Redis

Si estás usando el transmisor de Redis, debes instalar ya sea la extensión de PHP phppredis mediante PECL o instalar la librería Predis mediante Composer:

```
composer require predis/predis
```

php

El transmisor de Redis transmitirá mensajes usando las característica pub / sub de Redis; sin embargo, necesitarás unir esto con un servidor de WebSocket que puede recibir mensajes desde Redis y emitirlos a tus canales de WebSocket.

Cuando el transmisor de Redis publica un evento, éste será publicado en los nombres de canales especificados en el evento y la carga será una cadena codificada de JSON que contiene el nombre del evento, una carga `data` y el usuario que genero el ID de socket del evento (si aplica).

Socket.IO

Si vas a unir el transmisor de Redis con un servidor Socket.IO, necesitarás incluir la librería de Socket.IO en tu aplicación. Puedes instalarla mediante el gestor de paquetes NPM:

```
npm install --save socket.io-client
```

php

Luego, necesitarás instanciar Echo con el conector `socket.io` y un `host`.

```
import Echo from "laravel-echo"  
  
window.io = require('socket.io-client');
```

php

```
window.Echo = new Echo({
    broadcaster: 'socket.io',
    host: window.location.hostname + ':6001'
});
```

Finalmente, necesitarás ejecutar un servidor de Socket.IO compatible. Laravel no incluye la implementación de un servidor Socket.IO; sin embargo, un servidor de Socket.IO de la comunidad es actualmente mantenido en el repositorio de GitHub [tlaverdure/laravel-echo-server](#).

Prerrequisitos de la cola

Antes de transmitir eventos, también necesitarás configurar y ejecutar un [listener de colas](#). Toda la transmisión de eventos es realizada mediante trabajos en cola para que el tiempo de respuesta de tu aplicación no se vea necesariamente afectado.

Descripción general

La transmisión de eventos de Laravel te permite transmitir tus eventos del lado del servidor de Laravel a tu aplicación JavaScript del lado del cliente usando un enfoque basado en drivers a los WebSockets.

Actualmente, Laravel viene con drivers de [canales de Pusher](#) y Redis. Los eventos pueden ser fácilmente consumidos en el lado del cliente usando el paquete de JavaScript [Laravel Echo](#).

Los eventos son transmitidos mediante "canales", que pueden ser definidos como públicos o privados. Cualquier visitante en tu aplicación puede suscribirse a una canal público sin necesidad de autenticación o autorización; sin embargo, para poder suscribirse a canales privados, un usuario debe estar autenticado y autorizado para escuchar en dicho canal.

Usando una aplicación de ejemplo

Antes de sumergirnos en cada componente de la transmisión de eventos, vamos a ver un resumen usando una tienda virtual como ejemplo. No discutiremos los detalles sobre configurar [canales de Pusher](#) o [Laravel Echo](#) dado que éstos será discutido a detalle en otras secciones de esta documentación.

En nuestra aplicación, vamos a asumir que tenemos una página que permite a los usuarios ver el estado de envío de sus ordenes. Vamos también a asumir que un evento `ShippingStatusUpdated` es ejecutado cuando un estado de envío es procesado por la aplicación:

```
event(new ShippingStatusUpdated($update));
```

php

Interfaz `ShouldBroadcast`

Cuando un usuario está viendo una de sus órdenes, no queremos que tengan que refrescar la página para ver las actualizaciones del estado. En su lugar, queremos transmitir las actualizaciones a la aplicación a medida que son creadas. Así que, necesitamos marcar el evento

`ShippingStatusUpdated` con la interfaz `ShouldBroadcast`. Esto instruirá a Laravel para que transmita el evento cuando es ejecutado:

```
<?php  
  
namespace App\Events;  
  
use Illuminate\\Broadcasting\\Channel;  
use Illuminate\\Broadcasting\\InteractsWithSockets;  
use Illuminate\\Broadcasting\\PresenceChannel;  
use Illuminate\\Broadcasting\\PrivateChannel;  
use Illuminate\\Contracts\\Broadcasting\\ShouldBroadcast;  
use Illuminate\\Queue\\SerializesModels;  
  
class ShippingStatusUpdated implements ShouldBroadcast  
{  
    /**  
     * Information about the shipping status update.  
     *  
     * @var string  
     */  
    public $update;  
}
```

php

La interfaz `ShouldBroadcast` requiere que nuestro evento defina un método `broadcastOn`. Este método es responsable de retornar los canales en los que el evento debería transmitir. Un stub vacío para este método está definido en las clases de eventos generadas, así que sólo necesitamos rellenar sus detalles. Sólo queremos que el creador de la orden sea capaz de ver las actualizaciones de estado, así que transmitiremos el evento en un canal privado que está enlazado a la orden:

```
/*
 * Get the channels the event should broadcast on.
 *
 * @return \Illuminate\Broadcasting\PrivateChannel
 */
public function broadcastOn()
{
    return new PrivateChannel('order.'.$this->update->order_id);
}
```

php

Autorizando canales

Recuerda, los usuarios deben ser autorizados para escuchar en canales privados. Podemos definir las reglas de autorización de nuestro canal en el archivo `routes/channels.php`. En este ejemplo, necesitamos verificar que cualquier usuario intentando escuchar en el canal privado `order.1` es realmente el creador de la orden:

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {
    return $user->id === Order::findOrNew($orderId)->user_id;
});
```

php

El método `channel` acepta dos argumentos: el nombre del canal y un callback que retorna `true` o `false` indicando si el usuario está autorizado para escuchar en el canal.

Todos los callbacks de autorización reciben al usuario actualmente autenticado como primer argumento y cualquier parámetro adicional como siguientes argumentos. En este ejemplo, estamos usando el placeholder `{orderId}` para indicar que la porción "ID" del nombre del canal es un wildcard.

Escuchar transmisiones de eventos

Luego, todo lo que queda es escuchar el evento en nuestra aplicación de JavaScript. Podemos hacer esto usando Laravel Echo. Primero, usaremos el método `private` para suscribirnos a un canal privado. Luego, podemos usar el método `listen` para escuchar el evento `ShippingStatusUpdated`. Por defecto, todas las propiedades públicas del evento serán incluidas en el evento de transmisión:

```
Echo.private(`order.${orderId}`)
    .listen('ShippingStatusUpdated', (e) => {
```

php

```
    console.log(e.update);
});
```

Definiendo la transmisión de eventos

Para informar a Laravel que un evento dado debería ser transmitido, implementa la interfaz

`Illuminate\Contracts\Broadcasting\ShouldBroadcast` en la clase del evento. Esta interfaz ya está importada en todas las clases de eventos generadas por el framework para que así puedas agregarla fácilmente a tus eventos.

La interfaz `ShouldBroadcast` requiere que implementes un sólo método: `broadcastOn`. El método `broadcastOn` debería retornar un canal o un arreglo de canales en los que el evento debería transmitirse. Los canales deben ser instancias de `Channel`, `PrivateChannel` o `PresenceChannel`. Las instancias de `Channel` representan canales públicos a los que cualquier usuario puede suscribirse mientras que `PrivateChannels` y `PresenceChannels` representan canales privados que requieren [autorización](#):

```
<?php  
  
namespace App\Events;  
  
use App\User;  
use Illuminate\Broadcasting\Channel;  
use Illuminate\Broadcasting\InteractsWithSockets;  
use Illuminate\Broadcasting\PresenceChannel;  
use Illuminate\Broadcasting\PrivateChannel;  
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;  
use Illuminate\Queue\SerializesModels;  
  
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);
}
```

Luego, sólo necesitas [ejecutar el evento](#) como normalmente lo harías. Una vez que el evento ha sido ejecutado, [un trabajo en cola](#) transmitirá automáticamente el evento a través de tu driver de transmisión especificado.

Nombre de la transmisión

Por defecto, Laravel transmitirá el evento usando el nombre de clase del evento. Sin embargo, puedes personalizar el nombre de la transmisión definiendo un método `broadcastAs` en el evento:

```
/*
 * The event's broadcast name.
 *
 * @return string
 */
public function broadcastAs()
{
    return 'server.created';
}
```

Si personalizas el nombre de la transmisión usando el método `broadcastAs`, debes asegurarte de registrar tu listener prefijándolo con un carácter `.`. Esto instruirá a Echo a no agregar el nombre de espacio de la aplicación al evento:

```
.listen('.server.created', function (e) {  
    ...  
});
```

php

Datos de la transmisión

Cuando un evento es transmitido, todas sus propiedades `public` son automáticamente serializadas y transmitidas como carga del evento, permitiéndote acceder a cualquiera de sus datos públicos desde tu aplicación de JavaScript. Así que, por ejemplo, si tu evento tiene una sola propiedad pública `$user` que contiene un modelo de Eloquent, la carga de transmisión del evento sería:

```
{  
    "user": {  
        "id": 1,  
        "name": "Patrick Stewart"  
        ...  
    }  
}
```

php

Sin embargo, si deseas tener mayor control sobre la carga transmitida, puedes agregar un método `broadcastWith` a tu evento. Este método debería retornar el arreglo de datos que deseas transmitir como la carga del evento:

```
/**  
 * Get the data to broadcast.  
 *  
 * @return array  
 */  
public function broadcastWith()  
{  
    return ['id' => $this->user->id];  
}
```

php

Cola de transmisión

Por defecto, cada evento transmitido es colocado en la cola por defecto para la conexión de cola por defecto especificada en tu archivo de configuración `queue.php`. Puedes personalizar la cola usada por

el transmisor definiendo una propiedad `broadcastQueue` en la clase de tu evento. Esta propiedad debería especificar el nombre de la cola que deseas usar al transmitir:

```
/**  
 * The name of the queue on which to place the event.  
 *  
 * @var string  
 */  
public $broadcastQueue = 'your-queue-name';
```

php

Si quieres transmitir tu evento usando la cola `sync` en lugar del driver de cola por defecto, puedes implementar la interfaz `ShouldBroadcastNow` en lugar de `ShouldBroadcast`:

```
<?php  
  
use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;  
  
class ShippingStatusUpdated implements ShouldBroadcastNow  
{  
    //  
}
```

php

Condiciones de la transmisión

Algunas veces quieres transmitir tu evento sólo si una condición dada es verdadera. Puedes definir estas condiciones agregando un método `broadcastWhen` a la clase de tu evento:

```
/**  
 * Determine if this event should broadcast.  
 *  
 * @return bool  
 */  
public function broadcastWhen()  
{  
    return $this->value > 100;  
}
```

php

Autorizando canales

Los canales privados requieren que autorices que el usuario actualmente autenticado puede escuchar en el canal privado. Esto es logrado haciendo una solicitud HTTP a tu aplicación de Laravel con el nombre del canal y permitiendo a tu aplicación de terminar si el usuario puede escuchar en dicho canal. Al usar [Laravel Echo](#), la solicitud HTTP para autorizar suscripciones a canales privados será realizada automáticamente; sin embargo, si necesitas definir las rutas necesarias para responder a estas solicitudes.

Definiendo rutas de autorización

Afortunadamente, Laravel hace que sea fácil definir las rutas para responder a las solicitudes de autorización de canales. En el `BroadcastServiceProvider` incluido con tu aplicación de Laravel, verás una llamada al método `Broadcast::routes`. Este método registrará la ruta `/broadcasting/auth` para manejar las solicitudes de autorización:

```
Broadcast::routes();
```

php

El método `Broadcast::routes` automáticamente coloca sus rutas dentro del grupo de middleware `web`; sin embargo, puedes pasar un arreglo de atributos de ruta al método si te gustaría personalizar los atributos asignados:

```
Broadcast::routes($attributes);
```

php

Personalizando endpoints de autorización

Por defecto, Echo usará el endpoint `/broadcasting/auth` para autorizar acceso a canales. Sin embargo, puedes especificar tus propios endpoints de autorización pasando la opción de configuración `authEndpoint` a tu instancia de Echo:

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key',
    authEndpoint: '/custom/endpoint/auth'
});
```

php

Definiendo callbacks de autorización

Luego, necesitamos definir la lógica que realizará la autorización del canal. Esto es hecho en el archivo `routes/channels.php` que es incluido con tu aplicación. En este archivo, puedes usar el método `Broadcast::channel` para registrar callbacks de autorización de canales:

```
Broadcast::channel('order.{orderId}', function ($user, $orderId) {  
    return $user->id === Order::findOrNew($orderId)->user_id;  
});
```

php

El método `channel` acepta dos argumentos: el nombre del canal y un callback que retorna `true` o `false` indicando si el usuario está autorizado para escuchar el canal.

Todos los callbacks de autorización reciben al usuario actualmente autenticado como primer argumento y cualquier parametro wildcard como sus argumentos siguientes. En este ejemplo, estamos usando el placeholder `{orderId}` para indicar la porción "ID" del nombre del canal es un wildcard.

Enlace de modelos del callback de autorización

Igual que las rutas HTTP, las rutas de los canales pueden tomar ventaja de [modelo de enlace de rutas](#) de forma implícita y explícita. Por ejemplo, en lugar de recibir la cadena o ID numérico de la orden, puedes solicitar una instancia del modelo `Order`:

```
use App\Order;  
  
Broadcast::channel('order.{order}', function ($user, Order $order) {  
    return $user->id === $order->user_id;  
});
```

php

Autenticación del callback de autorización

Los canales privados y de presencia autentican al usuario actual a través de la protección de autenticación por defecto de la aplicación. Si el usuario no está autenticado, la autorización del canal es automáticamente negada y el callback de autorización nunca se ejecuta. Sin embargo, puedes asignar múltiples protecciones personalizadas que deben autenticar la solicitud entrante si es necesario:

```
Broadcast::channel('channel', function () {  
    // ...  
}, ['guards' => ['web', 'admin']]));
```

php

Definiendo clases de canales

Si tu aplicación está consumiendo muchos canales diferentes, tu archivo `routes/channels.php` podría volverse voluminoso. Así que, en lugar de usar Closures para autorizar canales, puedes usar clases de canales. Para generar una clase de canal, usa el comando de artisan `make:channel`. Este comando colocará una nueva clase de canal en el directorio `App/Broadcasting`.

```
php artisan make:channel OrderChannel
```

php

Luego, registra tu canal en tu archivo `routes/channels.php`:

```
use App\Broadcasting\OrderChannel;  
  
Broadcast::channel('order.{order}', OrderChannel::class);
```

php

Finalmente, puedes colocar la lógica de autorización para tu canal en el método `join` de la clase del canal. Este método `join` contendrá la misma lógica que típicamente habrías colocado en el Closure de tu canal de autorización. Puedes también tomar ventaja del modelo de enlace de canales:

```
<?php  
  
namespace App\Broadcasting;  
  
use App\Order;  
use App\User;  
  
class OrderChannel  
{  
    /**  
     * Create a new channel instance.  
     *  
     * @return void  
     */  
    public function __construct()  
    {  
        //  
    }  
  
    /**
```

php

```
* Authenticate the user's access to the channel.  
*  
* @param \App\User $user  
* @param \App\Order $order  
* @return array|bool  
*/  
public function join(User $user, Order $order)  
{  
    return $user->id === $order->user_id;  
}  
}
```

TIP

Como muchas otras clases en Laravel, las clases de canales automáticamente serán resueltas por el [contenedor de servicios](#). Así que, puedes declarar el tipo de cualquier dependencia requerida por tu canal en su constructor.

Transmitiendo eventos

Una vez que has definido un evento y lo has marcado con la interfaz `ShouldBroadcast`, sólo necesitas ejecutar el evento usando la función `event`. El despachador de eventos notará que el evento está marcado con la interfaz `ShouldBroadcast` y agrega el evento a la cola para transmisión:

```
event(new ShippingStatusUpdated($update));
```

php

Sólo a otros

Al construir una aplicación que usa la transmisión de eventos, puedes sustituir la función `event` por la función `broadcast`. Como la función `event`, la función `broadcast` despacha el evento a tus listeners del lado del servidor:

```
broadcast(new ShippingStatusUpdated($update));
```

php

Sin embargo, la función `broadcast` también expone el método `toOthers` que te permite excluir al usuario actual de los recipientes de la transmisión:

```
broadcast(new ShippingStatusUpdated($update))->toOthers();
```

php

Para entender mejor cuando es posible que quieras usar el método `toOthers`, vamos a imaginar una aplicación de lista de tareas donde un usuario puede crear una nueva tarea ingresando un nombre de tarea. Para crear una tarea, tu aplicación puede hacer una solicitud a un punto de salida `/task` que transmite la creación de la tarea y retorna una representación JSON de la nueva tarea. Cuando tu aplicación de JavaScript recibe la respuesta del punto de salida, podría directamente insertar la nueva tarea en su lista de tareas de la siguiente forma:

```
axios.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
 });
```

php

Sin embargo, recuerda que también transmitimos la creación de la tarea. Si tu aplicación de JavaScript está escuchando este evento para agregar tareas a la lista de tareas, tendrás tareas duplicadas en tu lista: una del punto de salida y una de la transmisión. Puedes resolver esto usando el método `toOthers` para instruir al transmisor para que no transmita el evento al usuario actual.

Nota

Tu evento debe usar el trait `Illuminate\Broadcasting\InteractsWithSockets` para poder llamar al método `toOthers`.

Configuración

Cuando incializas una instancia de Laravel Echo, un ID de socket es asignado a la conexión. Si estás usando [Vue](#) y [Axios](#), el ID del socket será agregado automáticamente a cada solicitud saliente como un header `X-Socket-ID`. Entonces, cuando llamas al método `toOthers`, Laravel extraerá el ID del socket desde el encabezado e instruirá al transmisor a no transmitir a ninguna conexión con dicho ID de socket.

Si no estás usando Vue y Axios, necesitarás configurar manualmente tu aplicación de JavaScript para enviar el encabezado `X-Socket-ID`. Puedes retornar el ID del socket usando el método `Echo.socketId`:

```
var socketId = Echo.socketId();
```

php

Recibiendo transmisiones

Instalando laravel echo

Laravel Echo es una librería de JavaScript que hace que sea fácil suscribirse a canales y escuchar transmisiones de eventos en Laravel. Puedes instalar Echo mediante el administrador de paquetes NPM. En este ejemplo, también instalaremos el paquete `pusher-js` dado que usaremos el transmisor de canales de Pusher:

```
npm install --save laravel-echo pusher-js
```

php

Una vez que Echo es instalado, estás listo para crear una instancia nueva de Echo en el JavaScript de tu aplicación. Un buen lugar para hacer esto es en la parte inferior del archivo

`resources/js/bootstrap.js` que es incluido con el framework Laravel:

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key'
});
```

php

Al crear una instancia de Echo que usa el conector `pusher`, puedes especificar un `cluster` así como si la conexión debería ser realizada mediante TLS (por defecto, cuando `forceTLS` es `false`, una conexión no-TLS será realizada si la página fue cargada mediante HTTP, o como fallback si la conexión TLS falla):

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key',
    cluster: 'eu',
    forceTLS: true
});
```

php

Usando una instancia de cliente existente

Si ya tienes una instancia de cliente de canales de Pusher o Socket.io que te gustaría que Echo usara, puedes pasarla a Echo mediante la opción de configuración `client`:

```
const client = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key',
    client: client
});
```

php

Escuchando eventos

Una vez que has instalado e instanciado Echo, estás listo para comenzar a escuchar transmisiones de eventos. Primero, usa el método `channel` para retornar una instancia de un canal, luego llama al método `listen` para escuchar a un evento especificado:

```
Echo.channel('orders')
    .listen('OrderShipped', (e) => {
        console.log(e.order.name);
   });
```

php

Si te gustaría escuchar eventos en un canal privado, usa el método `private` en su lugar. Puedes continuar encadenando llamadas al método `listen` para escuchar múltiples eventos en un sólo canal:

```
Echo.private('orders')
    .listen(...)
    .listen(...)
    .listen(...);
```

php

Abandonando un canal

Para abandonar un canal, puedes llamar al método `leaveChannel` en tu instancia de Echo:

```
Echo.leaveChannel('orders');
```

php

Si te gustaría abandonar un canal y también sus canales privados y presenciales asociados, puedes usar el método `leave` :

```
Echo.leave('orders');
```

php

Nombres de espacio

Puedes haber notado en los ejemplos superiores que no especificamos un nombre de espacio completo para las clases del evento. Esto es debido a que Echo automáticamente asumirá que los eventos están ubicados en el nombre de espacio `App\Events`. Sin embargo, puedes configurar el nombre de espacio principal cuando instancias Echo pasando una opción de configuración `namespace` :

```
window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-pusher-channels-key',
    namespace: 'App.Other.Namespace'
});
```

php

Alternativamente, puedes prefijar las clases del evento con un `.` al suscribirte a estos usando Echo. Esto te permitirá siempre especificar el nombre de clase completamente calificado:

```
Echo.channel('orders')
    .listen('.Namespace\\Event\\class', (e) => {
        //
    });

```

php

Canales de presencia

Los Canales de Presencia son construidos sobre la seguridad de los canales privados mientras que exponen la característica adicional de saber quien está suscrito al canal. Esto hace que sea fácil construir características de aplicación poderosas y colaborativas como notificar a usuarios cuando otro usuario está viendo la misma página.

Autorizando canales de presencia

Todos los canales de Presencia son también canales privados; por lo tanto, los usuarios deben estar autorizados para accederlos. Sin embargo, al definir callbacks de autorización para canales de presencia, no retornarás `true` si el usuario está autorizado para unirse al canal. En su lugar, debes retornar un arreglo de datos sobre el usuario.

Los datos retornados por el callback de autorización estarán disponibles para los listeners de eventos de canales de presencia en tu aplicación de JavaScript. Si el usuario no está autorizado para unirse al canal de presencia, debes retornar `false` o `null`:

```
Broadcast::channel('chat.{roomId}', function ($user, $roomId) {  
    if ($user->canJoinRoom($roomId)) {  
        return ['id' => $user->id, 'name' => $user->name];  
    }  
});
```

php

Uniéndose a canales de presencia

Para unirse a un canal de presencia, puedes usar el método `join` de Echo. El método `join` retornará una implementación de `PresenceChannel` que, junto con la exposición del método `listen`, te permite suscribirte a los eventos `here`, `joining` y `leaving`.

```
Echo.join(`chat.${roomId}`)  
    .here((users) => {  
        //  
    })  
    .joining((user) => {  
        console.log(user.name);  
    })  
    .leaving((user) => {  
        console.log(user.name);  
    });
```

php

El callback `here` será ejecutado inmediatamente una vez que el canal se haya unido con éxito y recibirá un arreglo que contiene la información del usuario para todos los demás usuarios actualmente subscriptos al canal. El método `joining` será ejecutado cuando un nuevo usuario se une a un canal, mientras que el método `leaving` será ejecutado cuando un usuario abandona el canal.

Transmitiendo a canales de presencia

Los canales de Presencia pueden recibir eventos igual que los canales públicos y privados. Usando el ejemplo de una sala de chat, podemos querer transmitir eventos `NewMessage` al canal de presencia de la sala. Para hacer eso, retornaremos una instancia de `PresenceChannel` desde el método `broadcastOn` del evento:

```
/*
 * Get the channels the event should broadcast on.
 *
 * @return Channel|array
 */
public function broadcastOn()
{
    return new PresenceChannel('room.'.$this->message->room_id);
}
```

php

Como los eventos públicos o privados, los canales de presencia pueden ser transmitidos usando la función `broadcast`. Como con otros eventos, puedes usar el método `toOthers` para excluir al usuario actual de recibir las transmisiones:

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

php

Puedes escuchar el evento `join` mediante el método `listen` de Echo:

```
Echo.join(`chat.${roomId}`)
    .here(...)
    .joining(...)
    .leaving(...)
    .listen('NewMessage', (e) => {
        //
    });
});
```

php

Eventos del cliente

TIP

Al usar [canales de Pusher](#), debes habilitar la opción "Client Events" en la sección "App Settings" del [dashboard de tu aplicación](#) para enviar eventos del cliente.

Algunas veces puedes querer transmitir un evento a otros clientes conectados sin tocar tu aplicación en lo absoluto. Esto puede ser particularmente útil para cosas como "escribir" notificaciones, donde quieres advertir a los usuarios de tu aplicación que otro usuario está escribiendo un mensaje en una pantalla dada.

Para transmitir eventos del cliente, puedes usar el método `whisper` de Echo:

```
Echo.private('chat')
  .whisper('typing', {
    name: this.user.name
  });
php
```

Para escuchar eventos del cliente, puedes usar el método `listenForWhisper`:

```
Echo.private('chat')
  .listenForWhisper('typing', (e) => {
    console.log(e.name);
  });
php
```

Notificaciones

Al juntar transmisión de eventos con [notificaciones](#), tu aplicación de JavaScript puede recibir nuevas notificaciones mientras ocurren sin necesidad de refrescar la página. Primero, asegurate de leer la documentación sobre el uso [del canal de transmisión de notificaciones](#).

Una vez que has configurado una notificación para usar el canal de transmisión, puedes escuchar a los eventos de la transmisión usando el método `notification` de Echo. Recuerda, el nombre del canal debe ser igual al nombre de la clase de la entidad recibiendo la notificaciones:

```
Echo.private(`App.User.${userId}`)
  .notification((notification) => {
    console.log(notification.type);
  });
php
```

En este ejemplo, todas las notificaciones enviadas a instancias de `App\User` mediante el canal `broadcast` serán recibidas por el callback. Un callback de autorización de canal para el canal `App\User.{id}` es incluido en el `BroadcastServiceProvider` que viene con el framework Laravel por defecto.

Caché

- Configuración
 - Prerrequisitos del controlador
- Uso de caché
 - Obtener una instancia de caché
 - Recuperar elementos de caché
 - Almacenar elementos de caché
 - Eliminar elementos de caché
 - Cierres atómicos
 - El helper cache
- Etiquetas de caché
 - Almacenar elementos de caché etiquetados
 - Acceder a elementos de caché etiquetados
 - Eliminar elementos de caché etiquetados
- Agregar controladores de caché personalizados
 - Escribir el driver
 - Registrar el driver
- Eventos

Configuración

Laravel proporciona una API expresiva y unificada para varios backends de almacenamiento de caché. La configuración de caché está ubicada en `config/cache.php`. En este archivo puedes indicar el

controlador de caché que deseas utilizar por defecto en toda tu aplicación. Por defecto, Laravel es compatible con los almacenamientos en caché más populares, tales como [Memcached](#) y [Redis](#).

El archivo de configuración de caché contiene otras opciones adicionales, las cuales están documentadas dentro del mismo archivo, por lo que deberás asegurarte de revisar estas opciones. Por defecto, Laravel está configurado para utilizar el controlador de caché `local`, que almacena los objetos de caché serializados en el sistema de archivos. Para aplicaciones más grandes, es recomendable que utilices un controlador más robusto como Memcached o Redis. Incluso puedes configurar múltiples configuraciones de caché para el mismo controlador.

Prerrequisitos del controlador

Base de datos

Cuando utilices el controlador de caché `database`, necesitarás configurar una tabla que contenga los elementos de caché. Puedes encontrar un `Schema` de ejemplo en la tabla inferior:

```
Schema::create('cache', function ($table) {  
    $table->string('key')->unique();  
    $table->text('value');  
    $table->integer('expiration');  
});
```

TIP

También puedes utilizar el comando `php artisan cache:table` para generar una migración con el esquema apropiado.

Memcached

Utilizar el controlador Memcached requiere que tengas instalado el [paquete de Memcached PECL](#).

Puedes listar todos tus servidores de Memcached en el archivo de configuración `config/cache.php`:

```
'memcached' => [  
    [  
        'host' => '127.0.0.1',  
        'port' => 11211,  
        'weight' => 100
```

```
 ],  
 ],
```

También puedes establecer la opción `host` a la ruta de un socket de UNIX. Si haces esto, la opción `port` se debe establecer a `0`:

```
'memcached' => [  
    [  
        'host' => '/var/run/memcached/memcached.sock',  
        'port' => 0,  
        'weight' => 100  
    ],  
],
```

php

Redis

Antes de comenzar a utilizar el caché con Redis en Laravel, deberás instalar ya sea la extensión de PHP `PhpRedis` mediante PECL o instalar el paquete `predis/predis` (~1.0) mediante Composer.

Para más información sobre cómo configurar Redis, consulta la [página de la documentación de Laravel](#).

Uso de caché

Obtener una instancia de caché

Las interfaces `Illuminate\Contracts\Cache\Factory` y `Illuminate\Contracts\Cache\Repository` proporcionan acceso a los servicios de caché de Laravel. La interfaz `Factory` proporciona acceso a todos los controladores de caché definidos para tu aplicación. La interfaz `Repository` típicamente es una implementación del controlador de caché predeterminado para tu aplicación según lo especificado en tu archivo de configuración de `cache`.

Sin embargo, también puedes usar el facade `Cache`, que es lo que usaremos a lo largo de esta documentación. El facade `Cache` proporciona acceso conveniente y directo a las implementaciones subyacentes de las interfaces de Laravel.

```
<?php
```

```
namespace App\Http\Controllers;
```

php

```
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 múltiples almacenamientos de caché

Usando el facade `Cache`, puedes acceder a varios almacenamientos de caché a través del método `store`. La llave que se pasa al método `store` debe corresponder a uno de los almacenamientos listados en el arreglo de configuración `stores` en tu archivo de configuración `cache`:

```
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

Recuperar elementos de caché

El método `get` en el facade `Cache` es utilizado para recuperar elementos desde la caché. Si el elemento no existe en caché, se va a regresar `null`. Si lo deseas, puedes pasar un segundo argumento al método `get` indicando el valor predeterminado que deseas retornar en caso de que el elemento no exista:

```
$value = Cache::get('key');

$value = Cache::get('key', 'default');
```

Incluso puedes pasar una `Closure` como valor predeterminado. El resultado del `closure` será devuelto si el elemento especificado no existe en caché. Pasar un Closure te permite diferir la recuperación de valores predeterminados de una base de datos a otro servicio externo:

```
$value = Cache::get('key', function () {
    return DB::table(...)->get();
});
```

php

Comprobar la existencia de un elemento

El método `has` se puede utilizar para determinar la existencia de un elemento en caché. Este método devolverá `false` si el valor es `null`:

```
if (Cache::has('key')) {
    //
}
```

php

Incrementando / Decrementando valores

Los métodos `increment` y `decrement` se pueden usar para ajustar el valor de los elementos enteros en caché. Ambos métodos aceptan un segundo parámetro opcional que indica la cantidad por la cual incrementar o disminuir el valor del elemento:

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

php

Recuperar y almacenar

En ocasiones, es posible que deseas recuperar un elemento de la memoria caché, pero también almacenar un valor predeterminado si el elemento no existe. Por ejemplo, puedes deseas recuperar todos los usuarios de la memoria caché o, si no existen, recuperarlos desde la base de datos y agregarlos a la caché. Puedes hacer esto utilizando el método `Cache::remember`:

```
$value = Cache::remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

php

```
});
```

Si el elemento no existe en la memoria caché, se ejecutará el `Closure` pasado al método `remember` y su resultado se colocará en caché.

Puedes utilizar el método `rememberForever` para recuperar un elemento del caché o almacenarlo para siempre:

```
$value = Cache::rememberForever('users', function () {  
    return DB::table('users')->get();  
});
```

php

Recuperar y eliminar

Si necesitas recuperar un elemento del caché y después eliminarlo, puedes utilizar el método `pull`. Al igual que el método `get`, se devolverá `null` si el elemento no existe en la memoria caché:

```
$value = Cache::pull('key');
```

php

Almacenar elementos en caché

Puedes utilizar el método `put` en el facade `Cache` para almacenar elementos en caché:

```
Cache::put('key', 'value', $seconds);
```

php

Si el tiempo de almacenamiento no es pasado al método `put`, el elemento será almacenado indefinidamente:

```
Cache::put('key', 'value');
```

php

En lugar de pasar el número de segundos como un entero, también puedes pasar una instancia de `DateTime` que represente el tiempo de expiración del elemento almacenado en caché:

```
Cache::put('key', 'value', now()->addMinutes(10));
```

php

Almacenar si no está presente

El método `add` solo agregará el elemento a caché si éste no existe todavía en la memoria caché. El método va a regresar `true` si el elemento realmente se agregó a la caché. De otra manera, el método va a regresar `false`:

```
Cache::add('key', 'value', $seconds);
```

Almacenar elementos para siempre

El método `forever` puede ser utilizado para almacenar un elemento en la memoria caché de manera permanente. Como estos elementos no caducan, se deben eliminar de la memoria caché manualmente utilizando el método `forget`:

```
Cache::forever('key', 'value');
```

php

TIP

Si utilizas el controlador de Memcached, los elementos almacenados "permanentemente" podrán ser eliminados una vez que la caché alcance su tamaño límite.

Eliminar elementos de la caché

Puedes eliminar elementos de caché utilizando el método `forget`:

```
Cache::forget('key');
```

php

También puedes eliminar elementos de caché especificando un TTL cero o negativo:

```
Cache::put('key', 'value', 0);
```

php

```
Cache::put('key', 'value', -5);
```

Puedes borrar todo el caché utilizando el método `flush`:

```
Cache::flush();
```

php

Nota

La limpieza de caché no respeta el prefijo del caché y borrará todas las entradas del caché.

Considera esto cuidadosamente cuando borres un caché que sea compartido por otras aplicaciones.

Bloqueos atómicos

Nota

Para usar esta característica, tu aplicación debe estar haciendo uso de los drivers de caché `memcached`, `dynamodb`, o `redis` como el driver de caché por defecto de tu aplicación. Adicionalmente, todos los servidores deben estar comunicándose con el mismo servidor de caché central.

Los bloqueos atómicos permiten la manipulación de bloqueos distribuidos sin que tengas que preocuparte sobre las condiciones de la carrera. Por ejemplo, [Laravel Forge](#) usa bloqueos atómicos para asegurarse de que sólo una tarea remota está siendo ejecutada en un servidor a la vez. Puedes crear y administrar bloqueos usando el método `Cache::lock`:

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

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Lock acquired for 10 seconds...

    $lock->release();
}
```

php

El método `get` también acepta una Closure. Luego de que la Closure sea ejecutada, Laravel automáticamente liberará el cierre:

```
Cache::lock('foo')->get(function () {
    // Lock acquired indefinitely and automatically released...
});
```

php

Si el bloqueo no está disponible en el momento en que lo solicitas, puedes instruir a Laravel para que espere un número determinado de segundos. Si el bloqueo no puede ser adquirido dentro del tiempo límite especificado, una excepción `Illuminate\Contracts\Cache\LockTimeoutException` será mostrada:

```
use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {
    $lock->block(5);

    // Lock acquired after waiting maximum of 5 seconds...
} catch (LockTimeoutException $e) {
    // Unable to acquire lock...
} finally {
    optional($lock)->release();
}

Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired after waiting maximum of 5 seconds...
});
```

php

Administrando bloqueos a través de procesos

Algunas veces, necesitarás adquirir un bloqueo en un proceso para liberarlo en otro proceso distinto más adelante. Por ejemplo, podemos solicitar un bloqueo durante la ejecución de un proceso que hace una solicitud web pero queremos liberarlo después que se ejecute un trabajo que es despachado donde se hizo la solicitud a una cola de trabajos. En un escenario como éste, necesitaríamos tomar la identificación del propietario del bloqueo (owner token) en el ámbito donde se produce el mismo y pasarlo al trabajo que va a la cola de trabajos de modo que pueda volver a instanciar el bloqueo usando ese identificador.

```
// Within Controller...
$podcast = Podcast::find($id);
```

php

```
$lock = Cache::lock('foo', 120);

if ($result = $lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}

// Within ProcessPodcast Job...
Cache::restoreLock('foo', $this->owner)->release();
```

Si prefieres liberar un bloqueo sin necesidad de indicar su propietario, puedes usar el método

`forceRelease` :

```
Cache::lock('foo')->forceRelease();
```

php

El helper cache

Además de usar el facade `Cache` o la interfaz de caché, también puedes usar la función global `cache` para recuperar y almacenar información a través del caché. Cuando se llama a la función `cache` con un solo argumento, devolverá el valor de la clave dada:

```
$value = cache('key');
```

php

Si proporcionas un arreglo de pares clave / valor y su tiempo de expiración a la función, almacenará los valores en caché durante la duración especificada:

```
cache(['key' => 'value'], $seconds);

cache(['key' => 'value'], now()->addMinutes(10));
```

php

Cuando la función `cache` es llamada sin ningún argumento, ésta retorna una instancia de la implementación `Illuminate\Contracts\Cache\Factory`, permitiéndote llamar a otros métodos de almacenamiento en caché:

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
```

php

```
});
```

TIP

Al realizar pruebas utilizando la función global `cache`, deberás usar el método `Cache::shouldReceive` como si estuvieras [probando un facade](#).

Cache tags

Nota

Las etiquetas de caché no son compatibles cuando usas los controladores de caché `file` o `database`. Además, cuando se utilicen múltiples etiquetas con cachés que son almacenados "permanentemente", el rendimiento será mejor si utilizas un controlador como `memcached`, el cual automáticamente purga los registros obsoletos.

Almacenar Elementos De Caché Etiquetados

Las etiquetas de caché te permiten etiquetar elementos relacionados en caché y después limpiar todos los valores almacenados en caché asignados a una etiqueta dada. Puedes acceder a un caché etiquetado al pasar un arreglo ordenado de nombres de etiquetas. Por ejemplo, vamos a acceder a un caché etiquetado y al valor `put` en el caché:

```
Cache::tags(['people', 'artists'])->put('John', $john, $seconds);
```

php

```
Cache::tags(['people', 'authors'])->put('Anne', $anne, $seconds);
```

Acceder a elementos de caché etiquetados

Para recuperar un elemento de caché etiquetado, pasa la misma lista ordenada de etiquetas al método `tags` y después haz un llamado al método `get` con la clave que deseas recuperar:

```
$john = Cache::tags(['people', 'artists'])->get('John');
```

php

```
$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

Eliminar elementos de caché etiquetados

Puedes borrar todos los elementos a los que se les asigna una etiqueta o lista de etiquetas. Por ejemplo, la siguiente sentencia eliminaría todos los cachés etiquetados tanto con `people`, `authors` o ambos. Por lo tanto, tanto `Anne` como `John` serán eliminados de caché:

```
Cache::tags(['people', 'authors'])->flush();
```

php

Por el contrario, la siguiente sentencia eliminará solamente los cachés con la etiqueta `authors`, por lo tanto se eliminará `Anne`, pero `John` no:

```
Cache::tags('authors')->flush();
```

php

Agregar controladores de caché personalizados

Escribir el controlador

Para crear el controlador de caché, primero se debe implementar la [interfaz](#)

`Illuminate\Contracts\Cache\Store`. Por lo tanto, una implementación de caché de MongoDB se vería de la siguiente manera:

```
<?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, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
```

php

```
    public function getPrefix() {}  
}
```

Solo necesitas implementar cada uno de estos métodos utilizando una conexión de MongoDB. Para tener un ejemplo de cómo implementar cada uno de estos métodos, puedes echar un vistazo a [Illuminate\Cache\MemcachedStore](#) en el código fuente del framework. Una vez que completes la implementación, puedes finalizar con el registro de tu controlador personalizado.

```
Cache::extend('mongo', function ($app) {  
    return Cache::repository(new MongoStore);  
});
```

php

TIP

Si te preguntas dónde puedes colocar el código de tu driver de caché personalizado, puedes crear un espacio de nombre [Extensions](#) en tu directorio [app](#). Sin embargo, ten en cuenta que Laravel no tiene una estructura de aplicación rígida y por tanto eres libre de organizar tu aplicación de acuerdo a tus preferencias.

Registrando el driver

Para registrar el controlador de caché personalizado con Laravel, debes utilizar el método [extend](#) en el facade [Cache](#). La llamada a [Cache::extend](#) puede hacerse en el método [boot](#) del [App\Providers\AppServiceProvider](#) predeterminado que contiene cada aplicación nueva de Laravel, o puedes crear tu propio proveedor de servicios para alojar la extensión - solo recuerda registrar el proveedor en el arreglo de proveedores en [config/app.php](#) :

```
<?php  
  
namespace App\Providers;  
  
use App\Extensions\MongoStore;  
use Illuminate\Support\Facades\Cache;  
use Illuminate\Support\ServiceProvider;  
  
class CacheServiceProvider extends ServiceProvider  
{
```

php

```

    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Cache::extend('mongo', function ($app) {
            return Cache::repository(new MongoStore());
        });
    }
}

```

El primer argumento pasado al método `extend` es el nombre del controlador. Esto corresponde a la opción `driver` en el archivo de configuración `config/cache.php`. El segundo argumento es un Closure que debe regresar una instancia de `Illuminate\Cache\Repository`. El Closure debe pasar una instancia de `$app`, que es una instancia del [contenedor de servicios](#).

Una vez que hayas registrado tu extensión, actualiza la opción `driver` en tu archivo de configuración `config/cache.php` con el nombre de tu extensión.

Eventos

Para ejecutar código en cada operación de caché, puedes escuchar los [eventos](#) activados por el caché. Normalmente, debes colocar estos listener de eventos dentro de tu `EventServiceProvider`:

```

    /**
     * The event listener mappings for the application.
     *
     * @var array
     */

```

php

```
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
 - Creando colecciones
 - Extendiendo colecciones
- Métodos disponibles
- Mensajes de orden superior
- Colecciones lazy
 - Introducción
 - Creando colecciones lazy
 - El contrato Enumerable
 - Métodos de colección lazy

Introducción

La clase `Illuminate\Support\Collection` provee una interfaz fluida y conveniente para trabajar con arreglos de datos. Por ejemplo, mira el siguiente código. Usaremos la función helper `collect` para crear una nueva instancia de `Collection` pasando un arreglo como parámetro, se ejecuta la función `strtoupper` en cada elemento y luego elimina todos los elementos vacíos:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {  
    return strtoupper($name);  
})  
->reject(function ($name) {  
    return empty($name);  
});
```

Como puedes ver, la clase `Collection` te permite encadenar sus métodos para realizar un mapeo fluido y reducir el arreglo subyacente. En general, las colecciones son inmutables, es decir, cada método de `Collection` retorna una nueva instancia de `Collection`.

Creando colecciones

Como se ha mencionado más arriba, el helper `collect` retorna una nueva instancia de `Illuminate\Support\Collection` para el arreglo dado. Entonces, crear una colección es tan simple como:

```
$collection = collect([1, 2, 3]);
```

TIP

Las respuestas de Eloquent siempre retornan una instancia de `Collection`.

Extendiendo colecciones

Las colecciones son "macroable", es decir, te permite agregar métodos adicionales a la clase `Collection` en tiempo de ejecución. Por ejemplo, el siguiente código agrega un método `toUpper` a la clase `Collection`:

```
use Illuminate\Support\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']
```

Por lo general, los macros para una colección se declaran en un [proveedor de servicios](#).

Métodos disponibles

Por el resto de esta documentación, discutiremos cada método disponible en la clase [Collection](#).

Recuerda, todos estos métodos pueden estar encadenados a la manipulación fluida del arreglo subyacente. Además, casi todos los métodos devuelven una nueva instancia de [Collection](#), lo que te permite conservar la copia original de la colección cuando sea necesario:

all
average
avg
chunk
collapse
collect
combine
concat
contains
containsStrict
count
countBy
crossJoin
dd

diff
diffAssoc
diffKeys
dump
duplicates
duplicatesStrict
each
eachSpread
every
except
filter
first
firstWhere
flatMap
flatten
flip
forget
forPage
get
groupBy
has
implode
intersect
intersectByKeys
isEmpty
isNotEmpty
join
keyBy
keys
last
macro
make
map
mapInto
mapSpread
mapToGroups

mapWithKeys
max
median
merge
mergeRecursive
min
mode
nth
only
pad
partition
pipe
pluck
pop
prepend
pull
push
put
random
reduce
reject
replace
replaceRecursive
reverse
search
shift
shuffle
skip
slice
some
sort
sortBy
sortByDesc
sortKeys
sortKeysDesc
splice

split
sum
take
tap
times
toArray
toJson
transform
union
unique
uniqueStrict
unless
unlessEmpty
unlessNotEmpty
unwrap
values
when
whenEmpty
whenNotEmpty
where
whereStrict
whereBetween
whereIn
whereInStrict
whereInstanceOf
whereNotBetween
whereNotIn
whereNotInStrict
wrap
zip

Lista de métodos

`all()`

El método `all` devuelve el arreglo subyacente representado por la colección:

```
collect([1, 2, 3])->all();
```

php

```
// [1, 2, 3]
```

average()

Alias del método `avg`.

avg()

El método `avg` retorna el **promedio** de una llave dada:

```
$average = collect([('foo' => 10), ['foo' => 10], ['foo' => 20], ['foo' => 40]])  
// 20  
  
$average = collect([1, 1, 2, 4])->avg();  
// 2
```

chunk()

El método `chunk` divide la colección en múltiples colecciones más pequeñas de un tamaño dado:

```
$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 las **vistas** cuando se trabaja con un sistema de grillas como el de **Bootstrap**. Imagina que tienes una colección de modelos **Eloquent** y quieres mostrar en una grilla lo siguiente:

```
@foreach ($products->chunk(3) as $chunk)  
  <div class="row">
```

php

```
@foreach ($chunk as $product)
    <div class="col-xs-4">{{ $product->name }}</div>
@endforeach
</div>
@endforeach
```

collapse()

El método `collapse` contrae una colección de arreglos en una sola colección 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]
```

php

combine()

El método `combine` combina las llaves de la colección con los valores de otro arreglo o colección:

```
$collection = collect(['name', 'age']);  
  
$combined = $collection->combine(['George', 29]);  
  
$combined->all();  
  
// ['name' => 'George', 'age' => 29]
```

php

collect() {#collection-method}

El método `collect` retorna una nueva instancia `Collection` con los elementos actualmente en la colección:

```
$collectionA = collect([1, 2, 3]);  
  
$collectionB = $collectionA->collect();
```

php

```
$collectionB->all();
```

```
// [1, 2, 3]
```

El método `collect` es principalmente útil para convertir [colecciones lazy](#) a instancias `Collection` estándares:

```
$lazyCollection = LazyCollection::make(function () {  
    yield 1;  
    yield 2;  
    yield 3;  
});  
  
$collection = $lazyCollection->collect();  
  
get_class($collection);  
  
// 'Illuminate\Support\Collection'  
  
$collection->all();  
  
// [1, 2, 3]
```

php

TIP

El método `collect` es especialmente útil cuando tienes una instancia `Enumerable` y necesitas una instancia de colección "no lazy". Dado que `collect()` es parte del contrato `Enumerable`, puedes usarlo para obtener una instancia `Collection`.

concat()

El método `concat` concatena un arreglo dado o valores de una colección al final de la colección:

```
$collection = collect(['John Doe']);  
  
$concatenated = $collection->concat(['Jane Doe'])->concat(['name' => 'Johnny Doe']);  
  
$concatenated->all();
```

php

```
// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

contains()

El método `contains` determina si la colección contiene un elemento dado:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);  
  
$collection->contains('Desk');  
  
// true  
  
$collection->contains('New York');  
  
// false
```

php

También puedes pasar la llave y el valor al método `contains`, que determinará si existe en la colección:

```
$collection = collect([  
    ['product' => 'Desk', 'price' => 200],  
    ['product' => 'Chair', 'price' => 100],  
]);  
  
$collection->contains('product', 'Bookcase');  
  
// false
```

php

Finalmente, también puedes pasar una función de retorno al método `contains` para realizar tu propia comprobación:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->contains(function ($value, $key) {  
    return $value > 5;  
});
```

php

```
// false
```

El método `contains` utiliza comparaciones "flexibles" (loose) al verificar valores de elementos, lo que significa que una cadena con un valor entero se considerará igual a un entero del mismo valor. Usa el método `containsStrict` si deseas una comparación "estricta".

containsStrict()

Este método funciona igual que el método `contains`; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

TIP

El comportamiento de este método es modificado al usar [colecciones de Eloquent](#).

count()

El método `count` devuelve la cantidad total de elementos en la colección:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->count();  
  
// 4
```

php

countBy()

El método `count By` cuenta las ocurrencias de valores en la colección. Por defecto, el método cuenta las ocurrencias de cada elemento:

```
$collection = collect([1, 2, 2, 2, 3]);  
  
$counted = $collection->countBy();  
  
$counted->all();  
  
// [1 => 1, 2 => 3, 3 => 1]
```

php

Sin embargo, puedes pasar una función de retorno (callback) al método `countBy` para contar todos los elementos por un valor personalizado:

```
$collection = collect(['alice@gmail.com', 'bob@yahoo.com', 'carlos@gmail.com']);  
  
$counted = $collection->countBy(function ($email) {  
    return substr(strrchr($email, "@"), 1);  
});  
  
$counted->all();  
  
// ['gmail.com' => 2, 'yahoo.com' => 1]
```

`crossJoin()`

El método `crossJoin` realiza un join cruzado entre los valores de la colección y los arreglos o colecciones dadas, devolviendo 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` muestra los elementos de la colección y finaliza la ejecución del script:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
Collection {
    #items: array:2 [
        0 => "John Doe"
        1 => "Jane Doe"
    ]
}
*/
*/
```

Si no quieres dejar de ejecutar el script, usa el método `dump`.

diff()

El método `diff` compara la colección con otra colección o una `arreglo` simple de PHP basado en sus valores. Este método devolverá los valores en la colección original que no están presentes en la colección dada:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();
```

```
// [1, 3, 5]
```

TIP

El comportamiento de este método es modificado al usar [colecciones de Eloquent](#).

diffAssoc()

El método `diffAssoc` compara la colección con otra colección o un `arreglo` simple de PHP basado en sus claves y valores. Este método devolverá los pares 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 `arreglo` de PHP simple en base a sus claves. Este método devolverá los pares 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` volca los elementos de la colección:

```
$collection = collect(['John Doe', 'Jane Doe']);  
  
$collection->dump();  
  
/*  
Collection {  
    #items: array:2 [  
        0 => "John Doe"  
        1 => "Jane Doe"  
    ]  
}  
*/
```

Si deseas detener la ejecución del script después de volcar la colección, use el método `dd`.

duplicates()

El método `duplicates` obtiene y retorna valores duplicados de la colección:

```
$collection = collect(['a', 'b', 'a', 'c', 'b']);  
php
```

```
$collection->duplicates();
```

```
// [2 => 'a', 4 => 'b']
```

If the collection contains arrays or objects, you can pass the key of the attributes that you wish to check for duplicate values:

```
$employees = collect([
    ['email' => 'abigail@example.com', 'position' => 'Developer'],
    ['email' => 'james@example.com', 'position' => 'Designer'],
    ['email' => 'victoria@example.com', 'position' => 'Developer'],
])

$employees->duplicates('position');

// [2 => 'Developer']
```

php

duplicatesStrict()

Este método tiene la misma firma que el método `duplicates`, sin embargo, todos los valores son comparados usando comparaciones "estrictas".

each()

El método `each` itera sobre los elementos de la colección y pasa cada elemento a una función de retorno (callback):

```
$collection->each(function ($item, $key) {
    //
});
```

php

Si deseas detener la iteración a través de los elementos, puedes devolver `false` en la función de retorno (callback):

```
$collection->each(function ($item, $key) {
    if /* some condition */) {
        return false;
    }
});
```

php

eachSpread()

El método `eachSpread` itera sobre los elementos de la colección, pasando cada valor de elemento anidado a la función de retorno (callback):

```
$collection = collect(['John Doe', 35], ['Jane Doe', 33]);  
  
$collection->eachSpread(function ($name, $age) {  
    //  
});
```

php

Puedes detener la iteración a través de los elementos al devolver `false` en la función de retorno (callback):

```
$collection->eachSpread(function ($name, $age) {  
    return false;  
});
```

php

every()

El método `every` se puede usar para verificar que todos los elementos de una colección pasen una comprobación dada a través de una función de retorno (callback):

```
collect([1, 2, 3, 4])->every(function ($value, $key) {  
    return $value > 2;  
});  
  
// false
```

php

Si la colección está vacía, `every` devolverá true:

```
$collection = collect([]);  
  
$collection->every(function ($value, $key) {  
    return $value > 2;  
});
```

php

```
// true
```

except()

El método `except` devuelve todos los elementos de la colección, excepto aquellos con las llaves especificadas:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);  
  
$filtered = $collection->except(['price', 'discount']);  
  
$filtered->all();  
  
// ['product_id' => 1]
```

Para hacer lo contrario a `except`, vea el método [only](#).

TIP

El comportamiento de este método es modificado al usar [colecciones de Eloquent](#).

filter()

El método `filter` filtra la colección usando una función de retorno (callback), manteniendo solo los elementos que pasan la comprobación dada:

```
$collection = collect([1, 2, 3, 4]);  
  
$filtered = $collection->filter(function ($value, $key) {  
    return $value > 2;  
});  
  
$filtered->all();  
  
// [3, 4]
```

Si no se proporciona una función de retorno, se eliminarán todos los elementos de la colección que son equivalentes a `false` :

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);  
  
$collection->filter()->all();  
  
// [1, 2, 3]
```

php

Para hacer lo contrario a `filter`, echa un vistazo al método `reject`.

first()

El método `first` devuelve el primer elemento de la colección que pasa la comprobación en una función de retorno (callback) dada:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {  
    return $value > 2;  
});  
  
// 3
```

php

También puedes llamar al método `first` sin argumentos para obtener el primer elemento de la colección. Si la colección está vacía, se devuelve `null` :

```
collect([1, 2, 3, 4])->first();  
  
// 1
```

php

firstWhere()

El método `firstWhere` devuelve el primer elemento de la colección con la clave y el valor proporcionado:

```
$collection = collect([  
    ['name' => 'Regena', 'age' => null],  
    ['name' => 'Linda', 'age' => 14],  
    ['name' => 'Diego', 'age' => 23],
```

php

```
[ 'name' => 'Linda', 'age' => 84],  
]);  
  
$collection->firstWhere('name', 'Linda');  
  
// [ 'name' => 'Linda', 'age' => 14]
```

También puedes llamar al método `firstWhere` con un operador:

```
$collection->firstWhere('age', '>=', 18);  
  
// [ 'name' => 'Diego', 'age' => 23]
```

Similar al método `where`, puedes pasar un argumento al método `firstWhere`. En este escenario, el método `firstWhere` retornará el primer elemento donde el valor de la clave dada es "verídico":

```
$collection->firstWhere('age');  
  
// [ 'name' => 'Linda', 'age' => 14]
```

flatMap()

El método `flatMap` itera a través de la colección y pasa cada valor a una función de retorno (callback). La función de retorno es libre de modificar el elemento y devolverlo, formando así una nueva colección de elementos modificados. Entonces, el arreglo se aplana a un solo nivel:

```
$collection = collect([  
    ['name' => 'Sally'],  
    ['school' => 'Arkansas'],  
    ['age' => 28]  
]);  
  
$flattened = $collection->flatMap(function ($values) {  
    return array_map('strtoupper', $values);  
});  

```

flatten()

El método `flatten` aplana una colección multidimensional en una de una sola dimensión:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]  
  
$flattened = $collection->flatten();  
  
$flattened->all();  
  
// ['taylor', 'php', 'javascript'];
```

Opcionalmente, puedes pasarle a la función 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, al llamar a `flatten` sin proporcionar la profundidad también se aplanan los arreglos anidados, lo que da como resultado `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Proporcionar una profundidad te permite restringir los niveles de arreglos anidados que se aplinarán.

flip()

El método `flip` intercambia las llaves de la colección con sus valores correspondientes:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$flipped = $collection->flip();  
  
$flipped->all();  
  
// ['taylor' => 'name', 'laravel' => 'framework']
```

php

forget()

El método `forget` elimina un elemento de la colección por su clave:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$collection->forget('name');  
  
$collection->all();  
  
// ['framework' => 'laravel']
```

php

Nota

A diferencia de la mayoría de métodos de una colección, `forget` no devuelve una nueva colección modificada; modifica la colección a la que se llama.

forPage()

El método `forPage` devuelve una nueva colección que contiene los elementos que estarían presentes en un número de página determinado. El método acepta el número de página como su primer argumento y la cantidad de elementos para mostrar por página como su segundo argumento:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);  
  
$chunk = $collection->forPage(2, 3);
```

php

```
$chunk->all();
```

```
// [4, 5, 6]
```

get()

El método `get` devuelve el elemento en una clave determinada. Si la clave no existe, se devuelve `null`:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$value = $collection->get('name');  
  
// taylor
```

php

Opcionalmente, puedes pasar un valor predeterminado como segundo argumento:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);  
  
$value = $collection->get('foo', 'default-value');  
  
// default-value
```

php

Incluso puedes pasar una función de retorno (callback) como el valor por defecto. El resultado de la función de retorno se devolverá si la clave especificada no existe:

```
$collection->get('email', function () {  
    return 'default-value';  
});  
  
// default-value
```

php

groupBy()

El método `groupBy` agrupa los elementos de la colección con una clave determinada:

```
$collection = collect([  
    ['account_id' => 'account-x10', 'product' => 'Chair'],
```

php

```

['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 clave, también puedes pasar una función de retorno (callback). La función de retorno debe devolver el valor de la clave por la que deseas 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'],
    ],
]
*/

```

Además de pasar una clave, también puedes pasar una función de retorno (callback). La función de retorno debe devolver el valor de la clave por la que deseas agrupar:

```
php
$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);
```
$result = $data->groupBy([
 'skill',
 function ($item) {
 return $item['roles'];
 },
], $preserveKeys = true);
```
/*
[
    1 => [
        'Role_1' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_2' => [
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
        ],
        'Role_3' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
        ],
    ],
    2 => [
        'Role_1' => [
            30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
        ],
        'Role_2' => [
            40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
        ],
    ],
];
*/

```

has()

El método `has` determina si existe una clave dada en la colección:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount' => 5]);  
  
$collection->has('product');  
  
// true  
  
$collection->has(['product', 'amount']);  
  
// true  
  
$collection->has(['amount', 'price']);  
  
// false
```

implode()

El método `implode` une a los elementos de una colección. Sus argumentos dependen del tipo de elemento en la colección. Si la colección contiene arreglos u objetos, debes pasar la clave de los atributos que deseas unir y la cadena que deseas colocar entre los valores:

```
$collection = collect([  
    ['account_id' => 1, 'product' => 'Desk'],  
    ['account_id' => 2, 'product' => 'Chair'],  
]);  
  
$collection->implode('product', ' ', ' ');  
  
// Desk, Chair
```

Si la colección contiene cadenas simples o valores numéricos, pasa el separador como único argumento para el método:

```
collect([1, 2, 3, 4, 5])->implode('-');  
  
// '1-2-3-4-5'
```

intersect()

El método `intersect` elimina cualquier valor de la colección original que no esté presente en el arreglo o colección dada. La colección resultante conservará las claves de la colección original:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);  
  
$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);  
  
$intersect->all();  
  
// [0 => 'Desk', 2 => 'Chair']
```

php

TIP

El comportamiento de este método es modificado al usar [colecciones de Eloquent](#).

intersectByKeys()

El método `intersectByKeys` elimina cualquier clave de la colección original que no esté presente en el arreglo o colección dada:

```
$collection = collect([  
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009  
]);  
  
$intersect = $collection->intersectByKeys([  
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011  
]);  
  
$intersect->all();  
  
// ['type' => 'screen', 'year' => 2009]
```

php

isEmpty()

El método `isEmpty` devuelve `true` si la colección está vacía; de lo contrario, se devuelve `false`:

```
collect([])->isEmpty();
```

php

```
// true
```

isNotEmpty()

El método `isNotEmpty` devuelve `true` si la colección no está vacía; de lo contrario, se devuelve `false`:

```
collect([])->isNotEmpty();
```

php

```
// false
```

join()

El método `join` une los valores de la colección con una cadena:

```
collect(['a', 'b', 'c'])->join(' ', ' '); // 'a, b, c'  
collect(['a', 'b', 'c'])->join(' ', ' ', ' and ') // 'a, b, and c'  
collect(['a', 'b'])->join(' ', ' and ') // 'a and b'  
collect(['a'])->join(' ', ' and ') // 'a'  
collect([])->join(' ', ' and ') // ''
```

php

keyBy()

El método `keyBy` agrupa una colección por claves indicando una clave como parámetro. Si varios elementos tienen la misma clave, solo el último 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();  
  
/*
```

php

```
[  
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],  
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],  
]  
*/
```

También puedes pasar una función de retorno (callback) al método. La función debe devolver el valor de la clave de la colección:

```
$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` devuelve 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` devuelve el último elemento de la colección que pasa una condición dentro de una función de retorno (callback):

```
collect([1, 2, 3, 4])->last(function ($value, $key) {  
    return $value < 3;  
});  
  
// 2
```

php

También puedes llamar al método `last` sin parámetros para obtener el último elemento de la colección. Si la colección está vacía, se devuelve `null` :

```
collect([1, 2, 3, 4])->last();  
  
// 4
```

php

macro()

El método estático `macro` te permite agregar métodos a la clase `Collection` en tiempo de ejecución. Consulta la documentación en [Extendiendo Colecciones](#) para mas información.

make()

El método estático `make` crea una nueva instancia de `Collection`. Más información en la sección de [Creando Colecciones](#).

map()

El método `map` itera a través de la colección y pasa cada valor a una función de retorno. La función de retorno es libre de modificar el elemento y devolverlo, 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();
```

php

```
// [2, 4, 6, 8, 10]
```

Nota

Como la mayoría de los otros métodos de colecciones, `map` devuelve una nueva instancia de colección; no modifica la colección a la que se llama. Si quieras transformar la colección original, usa el método `transform`.

mapInto()

El método `mapInto()` itera sobre la colección, creando una nueva instancia de la clase 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 sobre los elementos de la colección, pasando cada valor de elemento anidado a la función de retorno pasada como parámetro. La función de retorno es libre de modificar el

elemento y devolverlo, 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 = $chunks->mapSpread(function ($even, $odd) {  
    return $even + $odd;  
});  
  
$sequence->all();  
  
// [1, 5, 9, 13, 17]
```

mapToGroups()

El método `mapToGroups` agrupa los elementos de la colección por la función de retorno dada. La función de retorno debería devolver un arreglo asociativo que contenga una única 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 a través de la colección y pasa cada valor a la función de retorno dada. La función de retorno debe devolver un arreglo asociativo que contiene una única 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` devuelve el valor máximo de una clave determinada:

```
$max = collect([('foo' => 10], ['foo' => 20])->max('foo');  
// 20  
  
$max = collect([1, 2, 3, 4, 5])->max();  
// 5
```

php

median()

El método `median` devuelve el **valor medio** de una clave dada:

```
$median = collect([('foo' => 10], ['foo' => 10], ['foo' => 20], ['foo' => 40])->  
// 15  
  
$median = collect([1, 1, 2, 4])->median();  
// 1.5
```

php

merge()

El método `merge` combina el arreglo o colección dada con la colección original. Si una clave en los elementos dados coincide con una clave de la colección original, el valor de los elementos dados sobrescribirá el valor en 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]
```

php

Si las llaves de los elementos son numéricas, los valores se agregarán al final de la colección:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

php

mergeRecursive()

El método `mergeRecursive` une el arreglo o colección dada de forma recursiva con la colección original. Si una cadena en los elementos dados coincide con una cadena en la colección original, entonces los valores para dichas cadenas son unidos en un arreglo, y esto es hecho de forma recursiva:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['product_id' => 2, 'price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

php

min()

El método `min` devuelve el valor mínimo de una llave determinada:

```
$min = collect([[ 'foo' => 10], [ 'foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

php

mode()

El método `mode` devuelve el **valor moda** de una clave dada:

```
$mode = collect([('foo' => 10), ['foo' => 10], ['foo' => 20], ['foo' => 40])->mode();  
// [10]  
  
$mode = collect([1, 1, 2, 4])->mode();  
// [1]
```

nth()

El método `nth` crea una nueva colección que consiste en cada elemento n-ésimo:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);  
  
$collection->nth(4);  
  
// ['a', 'e']
```

Opcionalmente puedes pasar un desplazamiento como segundo argumento:

```
$collection->nth(4, 1);  
  
// ['b', 'f']
```

only()

El método `only` devuelve los elementos de la colección con las claves especificadas:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100, 'discount' => 10]);  
  
$filtered = $collection->only(['product_id', 'name']);  
  
$filtered->all();  
  
// ['product_id' => 1, 'name' => 'Desk']
```

Para hacer lo inverso a `only`, usa el método `except`.

TIP

El comportamiento de este método es modificado al usar [colecciones de Eloquent](#).

pad()

El método `pad` llenará el arreglo con el valor dado hasta que el arreglo alcance el tamaño especificado. Este método se comporta como la función [array_pad](#) de PHP.

Para llenar a la izquierda, debes especificar un tamaño negativo. No se realizará ningún relleno si el valor absoluto del tamaño dado es menor o igual que la longitud del arreglo:

```
$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 PHP `list` para separar los elementos que pasan una comprobación dada de aquellos que no lo hacen:

```
$collection = collect([1, 2, 3, 4, 5, 6]);  
  
list($underThree, $equalOrAboveThree) = $collection->partition(function ($i) {  
    return $i < 3;  
});  
  
$underThree->all();
```

```
// [1, 2]  
  
$equalOrAboveThree->all();  
  
// [3, 4, 5, 6]
```

pipe()

El método `pipe` pasa la colección a una función de retorno y devuelve el resultado:

```
$collection = collect([1, 2, 3]);  
  
$piped = $collection->pipe(function ($collection) {  
    return $collection->sum();  
});  
  
// 6
```

php

pluck()

El método `pluck` recupera todos los valores para una llave dada:

```
$collection = collect([  
    ['product_id' => 'prod-100', 'name' => 'Desk'],  
    ['product_id' => 'prod-200', 'name' => 'Chair'],  
]);  
  
$plucked = $collection->pluck('name');  
  
$plucked->all();  
  
// ['Desk', 'Chair']
```

php

También puedes especificar cómo deseas que se coloquen las llaves:

```
$plucked = $collection->pluck('name', 'product_id');  
  
$plucked->all();
```

php

```
// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

Si existen llaves duplicadas, el último elemento que coincida será insertado en la colección recuperada:

```
$collection = collect([
    ['brand' => 'Tesla', 'color' => 'red'],
    ['brand' => 'Pagani', 'color' => 'white'],
    ['brand' => 'Tesla', 'color' => 'black'],
    ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']
```

pop()

El método `pop` elimina y devuelve 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` agrega un elemento al comienzo de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);
```

```
$collection->all();  
  
// [0, 1, 2, 3, 4, 5]
```

También puedes pasar un segundo argumento para establecer la clave del elemento antepuesto:

```
$collection = collect(['one' => 1, 'two' => 2]);  
  
$collection->prepend(0, 'zero');  
  
$collection->all();  
  
// ['zero' => 0, 'one' => 1, 'two' => 2]
```

php

pull()

El método `pull` elimina y devuelve 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']
```

php

push()

El método `push` agrega un elemento al final de la colección:

```
$collection = collect([1, 2, 3, 4]);  
  
$collection->push(5);  
  
$collection->all();  
  
// [1, 2, 3, 4, 5]
```

php

put()

El método `put` establece la clave y el valor dado en la colección:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);  
  
$collection->put('price', 100);  
  
$collection->all();  
  
// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

php

random()

El método `random` devuelve un elemento aleatorio de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->random();  
  
// 4 - (retrieved randomly)
```

php

Opcionalmente, puedes pasar un número entero a `random` para especificar cuántos elementos deseas recuperar al azar. Siempre se devuelve una colección de valores cuando se pasa explícitamente la cantidad de valores que deseas recibir:

```
$random = $collection->random(3);  
  
$random->all();  
  
// [2, 4, 5] - (retrieved randomly)
```

php

Si la colección tiene un número menor de elementos al solicitado, el método arrojará un `InvalidArgumentException`.

reduce()

El método `reduce` reduce la colección a un solo valor, pasando el resultado de cada iteración a la iteración siguiente:

```
$collection = collect([1, 2, 3]);  
  
$total = $collection->reduce(function ($carry, $item) {  
    return $carry + $item;  
});  
  
// 6
```

php

El valor de `$carry` en la primera iteración es `null`; sin embargo, puedes especificar su valor inicial pasando un segundo argumento a reducir:

```
$collection->reduce(function ($carry, $item) {  
    return $carry + $item;  
}, 4);  
  
// 10
```

php

reject()

El método `reject` filtra la colección usando una función de retorno. La función de retorno debe devolver `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]
```

php

Para el inverso del método `reject`, ve el método `filter`.

replace()

El método `replace` se comporta de forma similar a `merge`; sin embargo, en adición a sobrescribir los elementos que coinciden con las cadenas, el método `replace` también sobrescribirá los elementos

en la colección que tienen claves numéricas coincidentes:

```
$collection = collect(['Taylor', 'Abigail', 'James']);  
  
$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);  
  
// ['Taylor', 'Victoria', 'James', 'Finn']
```

php

replaceRecursive()

Este método funciona como el método `replace`, pero se reflejará en arreglos y aplicará el mismo proceso de reemplazo a los valores internos:

```
$collection = collect(['Taylor', 'Abigail', ['James', 'Victoria', 'Finn']]);  
  
$replaced = $collection->replaceRecursive(['Charlie', 2 => [1 => 'King']]);  
  
$replaced->all();  
  
// ['Charlie', 'Abigail', ['James', 'King', 'Finn']]
```

php

reverse()

El método `reverse` invierte el orden de los elementos de la colección, conservando 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',  
]
```

php

```
]  
*/
```

search()

El método `search` busca en la colección el valor dado y devuelve su clave si se encuentra. Si el valor no se encuentra, se devuelve `false`.

```
$collection = collect([2, 4, 6, 8]);  
  
$collection->search(4);  
  
// 1
```

php

La búsqueda se realiza usando una comparación "flexible" (loose), lo que significa que una cadena con un valor entero se considerará igual a un número entero del mismo valor. Para usar una comparación "estricta", pasa `true` como segundo parámetro del método:

```
$collection->search('4', true);  
  
// false
```

php

Alternativamente, puedes pasar tu propia función de retorno para buscar el primer elemento que pase la validación:

```
$collection->search(function ($item, $key) {  
    return $item > 5;  
});  
  
// 2
```

php

shift()

El método `shift` elimina y devuelve el primer elemento de la colección:

```
$collection = collect([1, 2, 3, 4, 5]);
```

php

```
$collection->shift();  
  
// 1  
  
$collection->all();  
  
// [2, 3, 4, 5]
```

shuffle()

El método `shuffle` mezcla aleatoriamente los elementos en la colección:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$shuffled = $collection->shuffle();  
  
$shuffled->all();  
  
// [3, 2, 5, 1, 4] - (generated randomly)
```

php

skip() {#collection-method}

El método `skip` retorna una nueva colección, sin los primeros números de elementos dados:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);  
  
$collection = $collection->skip(4);  
  
$collection->all();  
  
// [5, 6, 7, 8, 9, 10]
```

php

slice()

El método `slice` devuelve una porción de la colección que comienza en el índice pasado como parámetro:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
```

php

```
$slice = $collection->slice(4);  
  
$slice->all();  
  
// [5, 6, 7, 8, 9, 10]
```

Si deseas limitar el tamaño de la porción devuelta, pase el tamaño deseado como segundo argumento para el método:

```
$slice = $collection->slice(4, 2);  
  
$slice->all();  
  
// [5, 6]
```

php

El segmento devuelto conservará las claves de forma predeterminada. Si no deseas conservar las claves originales, puedes usar el método `values` para volverlos a indexar.

some()

Alias para el método `contains`.

sort()

El método `sort` ordena la colección. La colección ordenada conserva las claves del arreglo original, por lo que en este ejemplo utilizaremos el método `values` para restablecer las claves de los índices numerados consecutivamente:

```
$collection = collect([5, 3, 1, 2, 4]);  
  
$sorted = $collection->sort();  
  
$sorted->values()->all();  
  
// [1, 2, 3, 4, 5]
```

php

Si tus necesidades de ordenamiento son más avanzadas, puedes pasar una función de retorno a `sort` con tu propio algoritmo. Consulta la documentación de PHP en [uasort](#), que es lo que llama el método `sort` de la colección.

TIP

Si necesitas ordenar una colección de matrices u objetos anidados, consulta los métodos

`sortBy` y `sortByDesc`.

sortBy()

El método `sortBy` ordena la colección con la clave dada. La colección ordenada conserva las claves del arreglo original, por lo que en este ejemplo utilizaremos el método `values` para restablecer las claves de los índices numerados consecutivamente:

```
$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],
]
*/
```

Puedes también pasar tu propia función de retorno 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 funcionalidad que el método `sortBy`, pero ordenará la colección en el orden opuesto.

sortKeys()

The `sortKeys` method ordena la colección por las llaves del arreglo asociativo subyacente:

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
[
    'first' => 'John',
    'id' => 22345,
    'last' => 'Doe',
]
*/
```

sortKeysDesc()

Este método tiene la misma funcionalidad que el método `sortKeys`, pero ordenará la colección en el orden opuesto.

splice()

El método `splice` elimina y devuelve una porción de elementos comenzando en el índice especificado:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2);  
  
$chunk->all();  
  
// [3, 4, 5]  
  
$collection->all();  
  
// [1, 2]
```

Puedes pasar un segundo parámetro para limitar el tamaño del fragmento resultante:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$chunk = $collection->splice(2, 1);  
  
$chunk->all();  
  
// [3]  
  
$collection->all();  
  
// [1, 2, 4, 5]
```

Además, puedes pasar un tercer parámetro que contenga los nuevos elementos para reemplazar los elementos 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 una colección en el número de grupos dado:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$groups = $collection->split(3);  
  
$groups->toArray();  
  
// [[1, 2], [3, 4], [5]]
```

php

sum()

El método `sum` devuelve la suma de todos los elementos de la colección:

```
collect([1, 2, 3, 4, 5])->sum();  
  
// 15
```

php

Si la colección contiene arreglos u objetos anidados, debes pasar una clave para determinar qué valores sumar:

```
$collection = collect([  
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],  
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],  
]);  
  
$collection->sum('pages');
```

php

```
// 1272
```

Además, puedes pasar una función de retorno para determinar qué valores de la colección 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
```

php

take()

El método `take` devuelve una nueva colección con el número especificado de elementos:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

php

También puedes pasar un número entero negativo para tomar la cantidad especificada de elementos del final de la colección:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

php

tap()

El método `tap` pasa la colección a la función de retorno dada, lo que te permite "aprovechar" la colección en un punto específico y hacer algo con los elementos sin afectar a la propia colección:

```
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function ($collection) {
        Log::debug('Values after sorting', $collection->values()->toArray());
    })
    ->shift();

// 1
```

php

times()

El método estático `times` crea una nueva colección invocando una función de retorno y la cantidad determinada de veces:

```
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collections->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

php

Este método puede ser útil cuando se combina con Factories para crear modelos [Eloquent](#):

```
$categories = Collection::times(3, function ($number) {
    return factory(Category::class)->create(['name' => "Category No. $number"]);
});

$categories->all();

/*
[
    ['id' => 1, 'name' => 'Category No. 1'],
    ['id' => 2, 'name' => 'Category No. 2'],
]
```

php

```
        ['id' => 3, 'name' => 'Category No. 3'],
    ]
*/
```

toArray()

El método `toArray` convierte la colección en un simple `array` de PHP. Si los valores de la colección son modelos [Eloquent](#), los modelos también se convertirán en arreglos:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);  
  
$collection->toArray();  
  
/*  
 [  
     ['name' => 'Desk', 'price' => 200],  
 ]  
*/
```

php

Nota

`toArray` también convierte todos los objetos anidados de la colección que son una instancia de `Arrayable` en un arreglo. En cambio, si deseas obtener el arreglo subyacente sin procesar, usa el método `all`.

toJson()

El método `toJson` convierte la colección en una cadena serializada JSON:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);  
  
$collection->toJson();  
  
// '{"name":"Desk", "price":200}'
```

php

transform()

El método `transform` itera sobre la colección y llama a la función de retorno dada con cada elemento de la colección. Los elementos en la colección serán reemplazados por los valores devueltos de la función de retorno:

```
$collection = collect([1, 2, 3, 4, 5]);  
  
$collection->transform(function ($item, $key) {  
    return $item * 2;  
});  
  
$collection->all();  
  
// [2, 4, 6, 8, 10]
```

Nota

A diferencia de la mayoría de otros métodos de las colecciones, `transform` modifica la colección en sí. Si deseas crear una nueva colección en su lugar, usa el método `map`.

union()

El método `union` agrega el arreglo dado a la colección. Si el arreglo contiene claves que ya están en la colección original, 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` devuelve todos los elementos únicos en la colección. La colección devuelta conserva las claves del arreglo original, por lo que en este ejemplo utilizaremos el método `values` para restablecer las llaves de los índices numerados consecutivamente:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);  
  
$unique = $collection->unique();  
  
$unique->values()->all();  
  
// [1, 2, 3, 4]
```

php

Al tratar con arreglos u objetos anidados, puedes especificar la clave utilizada para determinar la singularidad:

```
$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'],  
 * ]  
 */
```

php

También puedes pasar una función de retorno para determinar la singularidad del elemento:

```
$unique = $collection->unique(function ($item) {  
    return $item['brand'].$item['type'];  
});  
  
$unique->values()->all();  
  
/*  
 *  
 * [
```

php

```
[ '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 comparaciones "flexibles" (loose) al verificar valores de elementos, lo que significa que una cadena con un valor entero se considerará igual a un entero del mismo valor. Usa el método `uniqueStrict` para filtrar usando una comparación "estricta".

TIP

El comportamiento de este método es modificado al usar [colecciones de Eloquent](#).

`uniqueStrict()`

Este método tiene la misma funcionalidad que el método `unique`; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

`unless()`

El método `unless` ejecutará una función de retorno a menos que el primer argumento dado al método se evalúe como `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]
```

php

Para hacer lo inverso a `unless`, usa el método `when`.

unlessEmpty()

Alias para el método `whenNotEmpty`.

unlessNotEmpty()

Alias para el método `whenEmpty`.

unwrap()

El método estático `unwrap` devuelve los elementos subyacentes de la colección del valor dado cuando corresponda:

```
Collection::unwrap(collect('John Doe'));  
// ['John Doe']  
  
Collection::unwrap(['John Doe']);  
// ['John Doe']  
  
Collection::unwrap('John Doe');  
// 'John Doe'
```

values()

El método `values` devuelve una nueva colección con las claves restablecidas en 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 una función de retorno cuando el primer argumento dado al método se evalúa como `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]
```

Para hacer lo inverso a `when`, usa el método `unless`.

whenEmpty()

El método `whenEmpty` ejecutará la función de retorno dada cuando la colección esté vacía:

```
$collection = collect(['michael', 'tom']);  
  
$collection->whenEmpty(function ($collection) {  
    return $collection->push('adam');  
});  
  
$collection->all();  
  
// ['michael', 'tom']  
  
$collection = collect();
```

```
$collection->whenEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// ['adam']

$collection = collect(['michael', 'tom']);

$collection->whenEmpty(function ($collection) {
    return $collection->push('adam');
}, function($collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['michael', 'tom', 'taylor']
```

Para el inverso de `whenEmpty`, ve el método `whenNotEmpty`.

`whenNotEmpty()`

El método `whenNotEmpty` ejecutará la función de retorno dada cuando la colección no esté vacía:

```
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom', 'adam']

$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});
```

```
$collection->all();

// []

$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
}, function($collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

Para el inverso de `whenNotEmpty` , ve el método `whenEmpty` .

where()

El método `where` filtra la colección por clave y valor pasados como parámetros:

```
$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` usa comparaciones "flexibles" (loose) al verificar valores de elementos, lo que significa que una cadena con un valor entero se considerará igual a un entero del mismo valor. Usa el método `whereStrict` para hacer comparaciones "estrictas".

whereStrict()

Este método tiene la misma funcionalidad que el método `where`; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

whereBetween()

El método `whereBetween` filtra la colección dentro de un rango dado:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/*
[
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]
*/
```

whereIn()

El método `whereIn` filtra la colección por una clave / valor contenida dentro del arreglo dado:

```
$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` usa comparaciones "flexibles" (loose) al verificar valores de elementos, lo que significa que una cadena con un valor entero se considerará igual a un número entero del mismo valor. Usa el método `whereInStrict` para hacer comparaciones "estrictas".

whereInStrict()

Este método tiene la misma funcionalidad que el método `whereIn`; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

whereInstanceOf()

El método `whereInstanceOf` filtra la colección por un tipo de clase dado:

```
use App\User;  
use App\Post;  
  
$collection = collect([  
    new User,  
    new User,  
    new Post,  
]);  
  
$filtered = $collection->whereInstanceOf(User::class);  
  
$filtered->all();  
  
// [App\User, App\User]
```

whereNotBetween()

El método `whereNotBetween` filtra la colección fuera de un rango dado:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
[
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Pencil', 'price' => 30],
]
*/
```

whereNotIn()

El método `whereNotIn` filtra la colección por una clave / valor que no está contenida dentro del arreglo dado:

```
$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 comparaciones "flexibles" (loose) cuando se comprueban los valores de los elementos, lo que significa que una cadena con un valor entero se considerará igual a un número entero del mismo valor. Usa el método `whereNotInStrict` para hacer comparaciones "estrictas".

whereNotInStrict()

Este método tiene la misma funcionalidad que el método `whereNotIn`; sin embargo, todos los valores se comparan utilizando comparaciones "estrictas".

wrap()

El método estático `wrap` envuelve el valor dado en una colección cuando corresponda:

```
$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']
```

php

zip()

El método `zip` combina los valores del arreglo con los valores de la colección original en el índice correspondiente:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

php

Mensajes de orden superior

Las colecciones también brindan soporte para "mensajes de orden superior", que son atajos para realizar acciones comunes en las colecciones. Los métodos de recopilación que proporcionan mensajes de orden superior son: `average` , `avg` , `contains` , `each` , `every` , `filter` , `first` , `flatMap` , `groupBy` , `keyBy` , `map` , `max` , `min` , `partition` , `reject` , `some` , `sortBy` , `sortByDesc` , `sum` , and `unique` .

Se puede acceder a cada mensaje de orden superior como una propiedad dinámica en una instancia de colección. Por ejemplo, usemos el mensaje `each` de orden superior para llamar a un método en cada objeto dentro de una colección:

```
$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

php

Del mismo modo, podemos usar el mensaje de orden superior `sum` para reunir el número total de "votos" para una colección de usuarios:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

php

Colecciones lazy

Introducción

NOTA

Antes de aprender más sobre las colecciones Lazy de Laravel, toma algo de tiempo para familiarizarte con los generadores de PHP [»](#).

Para suplementar la ya poderosa clase `Collection`, la clase `LazyCollection` aprovecha los [generadores](#) de PHP para permitirte trabajar con datasets muy largos manteniendo un uso de memoria bajo.

Por ejemplo, imagina que tu aplicación necesita procesar un archivo log de múltiples gigabytes tomando ventaja de los métodos de colección de Laravel para parsear los registros. En lugar de leer el archivo completo en memoria una sola vez, las colecciones lazy pueden ser usadas para mantener sólo una pequeña parte del archivo en memoria en un momento dado:

```
use App\LogEntry;
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function ($lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

Imagina que necesitas iterar 10000 modelos de Eloquent. Al usar colecciones tradicionales de Laravel, los 10000 modelos deben ser cargados en memoria al mismo tiempo:

```
$users = App\User::all()->filter(function ($user) {
    return $user->id > 500;
});
```

Sin embargo, el método `cursor` del query builder retorna una instancia `LazyCollection`. Esto te permite ejecutar un sólo query en la base de datos así como también mantener sólo un modelo de

Eloquent cargado en memoria a la vez. En este ejemplo, el callback `filter` no es ejecutado hasta que realmente iteramos sobre cada usuario de forma individual, permitiendo una reducción drástica en el uso de memoria:

```
$users = App\User::cursor()->filter(function ($user) {  
    return $user->id > 500;  
});  
  
foreach ($users as $user) {  
    echo $user->id;  
}
```

php

Creando colecciones lazy

Para crear una instancia de colección lazy, debes pasar una función generadora de PHP al método `make` de la colección:

```
use Illuminate\Support\LazyCollection;  
  
LazyCollection::make(function () {  
    $handle = fopen('log.txt', 'r');  
  
    while (($line = fgets($handle)) !== false) {  
        yield $line;  
    }  
});
```

php

El contrato Enumerable

Casi todos los métodos disponibles en la clase `Collection` también están disponibles en la clase `LazyCollection`. Ambas clases implementan el contrato `Illuminate\Support\Enumerable`, el cual define los siguientes métodos:

all
average
avg
chunk
collapse

collect
combine
concat
contains
containsStrict
count
countBy
crossJoin
dd
diff
diffAssoc
diffKeys
dump
duplicates
duplicatesStrict
each
eachSpread
every
except
filter
first
firstWhere
flatMap
flatten
flip
forPage
get
groupBy
has
implode
intersect
intersectByKeys
isEmpty
isNotEmpty
join
keyBy

keys
last
macro
make
map
mapInto
mapSpread
mapToGroups
mapWithKeys
max
median
merge
mergeRecursive
min
mode
nth
only
pad
partition
pipe
pluck
random
reduce
reject
replace
replaceRecursive
reverse
search
shuffle
skip
slice
some
sort
sortBy
sortByDesc
sortKeys

sortKeysDesc
split
sum
take
tap
times
toArray
toJson
union
unique
uniqueStrict
unless
unlessEmpty
unlessNotEmpty
unwrap
values
when
whenEmpty
whenNotEmpty
where
whereStrict
whereBetween
whereIn
whereInStrict
whereInstanceOf
whereNotBetween
whereNotIn
whereNotInStrict
wrap
zip

NOTA

Los métodos que mutan la colección (como `shift` , `pop` , `prepend` etc.) *no* están disponibles en la clase `LazyCollection` .

Métodos de colección lazy

Además de los métodos definidos en el contrato `Enumerable`, la clase `LazyCollection` contiene los siguientes métodos:

`tapEach()` {#collection-method}

Mientras que el método `each` llama directamente al callback dado por cada elemento en la colección, el método `tapEach` sólo llama al callback dado cuando los elementos están siendo sacados de la lista uno por uno:

```
$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value) {  
    dump($value);  
});  
  
// Nothing has been dumped so far...  
  
$array = $lazyCollection->take(3)->all();  
  
// 1  
// 2  
// 3
```

`remember()` {#collection-method}

El método `remember` retorna una nueva colección lazy que recordará cualquier valor que ya haya sido enumerado y no lo retornará de nuevo cuando la colección sea enumerada otra vez:

```
$users = User::cursor()->remember();  
  
// No query has been executed yet...  
  
$users->take(5)->all();  
  
// The query has been executed and the first 5 users have been hydrated from the  
// database.  
  
$users->take(20)->all();  
  
// First 5 users come from the collection's cache... The rest are hydrated from
```

Eventos

- Introducción
- Registro de eventos y oyentes
 - Generación de eventos y oyentes
 - Registro manual de eventos
 - Descubrimiento de eventos
- Definiendo eventos
- Definiendo oyentes
- Oyentes de eventos en cola
 - Accediendo manualmente a la cola
 - Manejo de trabajos fallidos
- Despachando eventos
- Suscriptores de eventos
 - Escribiendo suscriptores de eventos
 - Registrando suscriptores de eventos

Introducción

Los eventos de Laravel proporcionan una implementación de observador simple, lo que permite suscribirse y escuchar diversos eventos que ocurren en tu aplicación. Las clases de eventos normalmente se almacenan en el directorio `app/Events`, mientras que tus oyentes se almacenan en `app/Listeners`. No te preocupes si no ves estos directorios en tu aplicación, ya que se crearán para ti cuando generes eventos y oyentes utilizando los comandos de consola Artisan.

Los eventos sirven como una excelente manera de desacoplar varios aspectos de tu aplicación, ya que un solo evento puede tener múltiples oyentes que no dependen entre sí. Por ejemplo, es posible que deseas enviar una notificación de Slack a tu usuario cada vez que se envíe un pedido. En lugar de acoplar

tu código de procesamiento de pedidos a tu código de notificación Slack, puedes generar un evento `OrderShipped`, que un oyente puede recibir y transformar en una notificación Slack.

Registro de eventos y oyentes

El `EventServiceProvider` incluido en tu aplicación Laravel proporciona un lugar conveniente para registrar todos los oyentes de eventos de tu aplicación. La propiedad `listen` contiene un arreglo de todos los eventos (claves) y sus oyentes (valores). Puedes agregar tantos eventos a este arreglo como lo requieras tu aplicación. Por ejemplo, agreguemos un evento `OrderShipped`:

```
/*
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\OrderShipped' => [
        'App\Listeners\SendShipmentNotification',
    ],
];
```

php

Generación de eventos y oyentes

Por supuesto, crear manualmente los archivos para cada evento y oyente es engorroso. En vez de eso, agrega oyentes y eventos a tu `EventServiceProvider` y usa el comando `event:generate`. Este comando generará cualquier evento u oyente que esté listado en tu `EventServiceProvider`. Los eventos y oyentes que ya existen quedarán intactos:

```
php artisan event:generate
```

php

Registro manual de eventos

Normalmente, los eventos deberían registrarse a través del arreglo `$listen` de `EventServiceProvider`; sin embargo, también puedes registrar manualmente eventos basados en Closure en el método `boot` de tu `EventServiceProvider`:

```
/**  
 * Register any other events for your application.  
 *  
 * @return void  
 */  
public function boot()  
{  
    parent::boot();  
  
    Event::listen('event.name', function ($foo, $bar) {  
        //  
    });  
}
```

php

Comodín de oyentes de un evento

Puedes incluso registrar oyentes usando el `*` como un parámetro comodín, lo que te permite capturar múltiples eventos en el mismo oyente. Los comodines de oyentes reciben el nombre del evento como su primer argumento y el arreglo de datos de eventos completo como su segundo argumento:

```
Event::listen('event.*', function ($eventName, array $data) {  
    //  
});
```

php

Descubrimiento de eventos

En vez de registrar eventos y oyentes (listeners) manualmente en el arreglo `$listen` del `EventServiceProvider`, puedes habilitar la detección automática de eventos. Cuando se habilita la detección de eventos, Laravel encontrará y registrará automáticamente tus eventos y oyentes escaneando el directorio `Listeners` de tu aplicación. Además, todos los eventos definidos explícitamente listados en el `EventServiceProvider` seguirán registrados.

Laravel encuentra los listeners de eventos mediante el escaneo de las clases listener usando reflexión. Cuando Laravel encuentra algún método de clase listener que empieza por `handle`, Laravel registrará dichos métodos como listeners de eventos para el evento que está escrito en la firma del método:

```
use App\Events\PodcastProcessed;
```

php

```
class SendPodcastProcessedNotification
{
    /**
     * Handle the given event.
     *
     * @param \App\Events\PodcastProcessed
     * @return void
     */
    public function handle(PodcastProcessed $event)
    {
        //
    }
}
```

La detección de eventos está deshabilitada de forma predeterminada, pero puedes habilitarla sobreescribiendo el método `shouldDiscoverEvents` del `EventServiceProvider` de tu aplicación:

```
/**
 * Determine if events and listeners should be automatically discovered.
 *
 * @return bool
 */
public function shouldDiscoverEvents()
{
    return true;
}
```

php

Por defecto, se escanearán todos los oyentes dentro del directorio `Listeners` de tu aplicación. Si deseas definir directorios adicionales para analizar, puedes sobreescibir el método

`discoverEventsWithin` en tu `EventServiceProvider`:

```
/**
 * Get the listener directories that should be used to discover events.
 *
 * @return array
 */
protected function discoverEventsWithin()
{
    return [

```

php

```
        $this->app->path('Listeners'),  
    ];  
}
```

En producción, es probable que no deseas que el framework analice todos tus oyentes en cada petición. Por lo tanto, durante tu proceso de despliegue, debes ejecutar el comando Artisan `event:cache` para almacenar en caché un manifiesto de todos los eventos y oyentes de tu aplicación. Este manifiesto será utilizado por el framework para acelerar el proceso de registro de eventos. El comando `event:clear` puede ser usado para destruir la caché.

TIP TIP

El comando `event:list` puede ser usado para mostrar una lista de todos los eventos y oyentes registrados por tu aplicación.

Definiendo eventos

Una clase de evento es un contenedor de datos que guarda la información relacionada con el evento. Por ejemplo, supongamos que nuestro evento `OrderShipped` (orden enviada) generado recibe un objeto **ORM Eloquent**:

```
<?php  
  
namespace App\Events;  
  
use App\Order;  
use Illuminate\Queue\SerializesModels;  
  
class OrderShipped  
{  
    use SerializesModels;  
  
    public $order;  
  
    /**  
     * Create a new event instance.  
     *  
     * @param \App\Order $order  
     * @return void  
    */
```

```
 */
public function __construct(Order $order)
{
    $this->order = $order;
}
}
```

Como puedes ver, esta clase de evento no contiene lógica. Es un contenedor para la instancia `Order` que se compró. El trait `SerializesModels` utilizado por el evento serializará con elegancia cualquier modelo Eloquent si el objeto del evento se serializa utilizando la función de PHP `serialize`.

Definiendo oyentes

A continuación, echemos un vistazo al oyente de nuestro evento de ejemplo. Los oyentes de eventos reciben la instancia de evento en su método `handle`. El comando `event:generate` importará automáticamente la clase de evento adecuada y declarará el tipo de evento en el método `handle`. Dentro del método `handle`, puedes realizar las acciones necesarias para responder al evento:

```
<?php  
php  
  
namespace App\Listeners;  
  
use App\Events\OrderShipped;  
  
class SendShipmentNotification  
{  
    /**  
     * Create the event listener.  
     *  
     * @return void  
     */  
    public function __construct()  
    {  
        //  
    }  
  
    /**  
     * Handle the event.  
     *  
     * @param \App\Events\OrderShipped $event  
     * @return void  
     */
```

```
 */
public function handle(OrderShipped $event)
{
    // Access the order using $event->order...
}
}
```

TIP TIP

Tus oyentes de eventos también pueden declarar el tipo de cualquier dependencia que necesiten de sus constructores. Todos los oyentes de eventos se resuelven a través [contenedor de servicio](#) de Laravel, por lo que las dependencias se inyectarán automáticamente.

Deteniendo la propagación de un evento

A veces, es posible que deseas detener la propagación de un evento a otros oyentes. Puede hacerlo devolviendo `false` desde el método `handle` de tu oyente.

Oyentes de Eventos de Cola

Los oyentes de colas pueden ser beneficiosos si tu oyente va a realizar una tarea lenta, como enviar un correo electrónico o realizar una solicitud HTTP. Antes de comenzar con oyentes de cola, asegúrate de [configurar su cola](#) e iniciar un oyente de colas en tu servidor o entorno de desarrollo local.

Para especificar que un oyente debe estar en cola, agrega la interfaz `ShouldQueue` a la clase de oyente. Los oyentes generados por el comando de Artisan `event:generate` ya tienen esta interfaz importada en el espacio de nombres actual, por lo que puedes usarla inmediatamente:

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderShipped;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class SendShipmentNotification implements ShouldQueue  
{  
    //  
}
```

php

¡Eso es! Ahora, cuando este oyente es llamado por un evento, el despachador de eventos lo colocará en cola automáticamente usando el [sistema de colas](#) de Laravel. Si no se lanzan excepciones cuando la cola ejecuta el oyente, el trabajo en cola se eliminará automáticamente una vez que haya terminado de procesarse.

Personalizando la conexión de la cola y el nombre de la cola

Si deseas personalizar la conexión de cola, el nombre de la cola o el tiempo de demora de la cola de un oyente de eventos, puedes definir las propiedades `$connection`, `$queue` o `$delay` en tu clase de oyente:

```
<?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';  
  
    /**  
     * The time (seconds) before the job should be processed.  
     *  
     * @var int  
     */  
    public $delay = 60;  
}
```

Cola condicional de listeners

Algunas veces, necesitarás determinar si un listener debe ser agregado a una cola en base a datos que sólo están disponibles en tiempo de ejecución. Para lograr esto, un método `shouldQueue` puede ser agregar a un listener para determinar si el listener debe ser agregado a una cola y ejecutado de forma sincronica:

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderPlaced;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class RewardGiftCard implements ShouldQueue  
{  
    /**  
     * Reward a gift card to the customer.  
     *  
     * @param \App\Events\OrderPlaced $event  
     * @return void  
     */  
    public function handle(OrderPlaced $event)  
    {  
        //  
    }  
  
    /**  
     * Determine whether the listener should be queued.  
     *  
     * @param \App\Events\OrderPlaced $event  
     * @return bool  
     */  
    public function shouldQueue(OrderPlaced $event)  
    {  
        return $event->order->subtotal >= 5000;  
    }  
}
```

Cola condicional de listeners

Algunas veces, puedes necesitar determinar si un listener debería agregarse a una cola en función de algún dato que sólo está disponible en tiempo de ejecución. Para lograr esto, el método `shouldQueue` puede ser añadido a un listener para determinar si el listener debe ser agregado a una cola y ejecutado de forma sincronizada:

```
<?php  
  
namespace App\Listeners;  
  
use App\Events\OrderPlaced;  
use Illuminate\Contracts\Queue\ShouldQueue;  
  
class RewardGiftCard implements ShouldQueue  
{  
    /**  
     * Reward a gift card to the customer.  
     *  
     * @param \App\Events\OrderPlaced $event  
     * @return void  
     */  
    public function handle(OrderPlaced $event)  
    {  
        //  
    }  
  
    /**  
     * Determine whether the listener should be queued.  
     *  
     * @param \App\Events\OrderPlaced $event  
     * @return bool  
     */  
    public function shouldQueue(OrderPlaced $event)  
    {  
        return $event->order->subtotal >= 5000;  
    }  
}
```

Accediendo manualmente a la cola

Si necesitas acceder manualmente a los métodos `delete` y `release` de la cola de trabajo subyacente del oyente, puedes hacerlo utilizando el trait `Illuminate\Queue\InteractsWithQueue`.

Este trait se importa de forma predeterminada en los oyentes 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);  
        }  
    }  
}
```

Manejo de trabajos fallidos

A veces, tus oyentes de eventos en cola pueden fallar. Si el oyente en cola supera el número máximo de intentos según lo define tu trabajador de cola, se llamará al método `failed` en tu oyente. El método `failed` recibe la instancia del evento y la excepción que causó el error:

```
<?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 $exception)
    {
        //
    }
}
```

Despachando eventos

Para enviar un evento, puedes pasar una instancia del evento a la función de ayuda (helper) `event` . El helper enviará el evento a todos tus oyentes registrados. Dado que el helper `event` está disponible globalmente, puedes llamarlo desde cualquier lugar de tu 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));
    }
}
```

TIP TIP

Al realizar pruebas, puede ser útil afirmar que ciertos eventos se enviaron sin activar realmente a tus oyentes. Las [funciones de ayuda \(helpers\)](#) incluidas en Laravel hace que sea fácil de hacerlo.

Suscriptores de eventos

Escribiendo suscriptores de eventos

Los suscriptores de eventos son clases que pueden suscribirse a múltiples eventos dentro de la misma clase, lo que te permite definir varios manejadores de eventos dentro de una sola clase. Los suscriptores deben definir un método `subscribe`, al que se le pasará una instancia de despachador de eventos. Puedes llamar al método `listen` en el despachador dado para registrar los oyentes de eventos:

```
<?php
namespace App\Listeners;
```

```

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin($event) {}

    /**
     * Handle user logout events.
     */
    public function handleUserLogout($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@handleUserLogin'
        );

        $events->listen(
            'Illuminate\Auth\Events\Logout',
            'App\Listeners\UserEventSubscriber@handleUserLogout'
        );
    }
}

```

Registrando suscriptores de eventos

Después de escribir el suscriptor, estás listo para registrarlo con el despachador de eventos. Puede registrar suscriptores usando la propiedad `$subscribe` en el `EventServiceProvider`. Por ejemplo, vamos a agregar el `UserEventSubscriber` a la lista:

```

<?php
namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvi

```

```
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

- Introducción
- Configuración
 - Disco público
 - Driver local
 - Prerrequisitos del driver
 - Cache
- Obteniendo instancias del disco
- Retornando archivos

- Descargando archivos
- URLs de archivos
- Metadatos de archivos
- Almacenando archivos
 - Carga de archivos
 - Visibilidad de archivos
- Eliminando archivos
- Directorios
- Sistemas de archivos personalizados

Introducción

Laravel proporciona una poderosa abstracción del sistema de archivos gracias al genial paquete de PHP [Flysystem](#) de Frank de Jonge. La integración de Flysystem de Laravel proporciona drivers simples de usar para trabajar con sistemas de archivos locales, Amazon S3 y Rackspace Cloud Storage.

Configuración

La configuración del sistema de archivos está ubicada en `config/filesystems.php`. Dentro de este archivo puedes configurar todos tus "discos". Cada disco representa un driver de almacenamiento y una ubicación de almacenamiento en particular. Configuraciones de ejemplo para cada driver soportado están incluidas en el archivo de configuración. Así que, modifica la configuración para reflejar tus preferencias de almacenamiento y credenciales.

Puedes configurar tantos discos como quieras e incluso tener múltiples discos que usen el mismo driver.

El disco público

El disco `public` está pensado para archivos que serán públicamente accesibles. Por defecto, el disco `public` usa el driver `local` y almacena estos archivos en `storage/app/public`. Para hacerlos accesibles desde la web, debes crear un enlace simbólico desde `public/storage` a `storage/app/public`. Esta convención mantendrá tus archivos públicamente accesibles en un directorio que puede ser fácilmente compartido a través de despliegues al usar sistemas de despliegue sin tiempo de inactividad como [Envoyer](#).

Para crear un enlace simbólico, puedes usar el comando de Artisan `storage:link`:

```
php artisan storage:link
```

php

Una vez que un archivo ha sido guardado y el enlace simbólico ha sido creado, puedes crear una URL a los archivos usando el helper `asset` :

```
echo asset('storage/file.txt');
```

php

Driver local

Al usar el driver `local`, todas las operaciones sobre archivos son relativas al directorio `root` definido en tu archivo de configuración. Por defecto, este valor está establecido al directorio `storage/app`. Por lo tanto, el siguiente método almacenará un archivo en `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

php

Permisos

La visibilidad `public` se traduce a `0755` para directorios y `0644` para archivos. Puedes modificar el mapeo de permisos por defecto en tu archivo de configuración `filesystems`:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0664,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0775,
            'private' => 0700,
        ],
    ],
],
```

php

Prerrequisitos del driver

Paquetes de Composer

Antes de usar los drivers de SFTP, S3 o Rackspace, necesitarás instalar el paquete apropiado mediante Composer:

- SFTP: `league/flysystem-sftp ~1.0`
- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

Algo sumamente recomendable para mejorar el rendimiento es usar un adaptador de caché. Necesitarás un paquete adicional para esto:

- CachedAdapter: `league/flysystem-cached-adapter ~1.0`

Configuración del driver S3

La información de configuración del driver de S3 está ubicada en tu archivo de configuración

`config/filesystems.php`. Este archivo contiene un arreglo de configuración de ejemplo para un driver de S3. Eres libre de modificar este arreglo con tu propia configuración y credenciales de S3. Por conveniencia, estas variables de entorno coinciden con la convención de nombres usada por AWS CLI.

Configuración del driver FTP

Las integraciones de Flysystem de Laravel funcionan bien con FTP; sin embargo, una configuración de ejemplo no está incluida con el archivo de configuración por defecto del framework

`filesystems.php`. Si necesitas configurar un sistema de archivos FTP, puedes usar la siguiente configuración de ejemplo:

```
'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 SFTP

Las integraciones de Flysystem de Laravel funcionan bien con SFTP; sin embargo, una configuración de ejemplo no está incluída con el archivo de configuración por defecto del framework

[filesystems.php](#). Si necesitas configurar un sistema de archivos SFTP, puedes usar la siguiente configuración de ejemplo:

```
'sftp' => [
    'driver' => 'sftp',
    'host' => 'example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Settings for SSH key based authentication...
    // 'privateKey' => '/path/to/privateKey',
    // 'password' => 'encryption-password',

    // Optional SFTP Settings...
    // 'port' => 22,
    // 'root' => '',
    // 'timeout' => 30,
],
```

Configuración del driver Rackspace

Las integraciones de Flysystem de Laravel funcionan bien con Rackspace; sin embargo, una configuración de ejemplo no está incluida con el archivo de configuración por defecto del framework

[filesystems.php](#). Si necesitas configurar un sistema de archivos de Rackspace, puedes usar la siguiente configuración de ejemplo:

```
'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',  
],
```

Cache

Para habilitar la cache para un disco dado, puedes agregar una directiva `cache` a las opciones de configuración del disco. La opción `cache` debe ser un arreglo de opciones de cache que contiene un nombre de disco `disk`, el tiempo de expiración en segundos `expire`, y el prefijo `prefix` de la cache:

```
's3' => [  
    'driver' => 's3',  
  
    // Other Disk Options...  
  
    'cache' => [  
        'store' => 'memcached',  
        'expire' => 600,  
        'prefix' => 'cache-prefix',  
    ],  
],
```

Obteniendo instancias del disco

El facade `Storage` puede ser usado para interactuar con cualquier de tus discos configurados. Por ejemplo, puedes usar el método `put` en el facade para almacenar un avatar en el disco por defecto. Si llamas a métodos en el facade `Storage` sin primero llamar al método `disk`, la llamada al método será automáticamente pasada al disco por defecto:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::put('avatars/1', $fileContents);
```

Si tus aplicaciones interactúan con múltiples discos, puedes usar el método `disk` en el facade `Storage` para trabajar con archivos en un disco en particular:

```
Storage::disk('s3')->put('avatars/1', $fileContents);
```

php

Retornando archivos

El método `get` puede ser usado para retornar el contenido de un archivo. Las cadenas del archivo serán retornadas por el método. Recuerda, todas las rutas del archivo deben ser especificadas relativas a la ubicación "raíz" configurada por el disco:

```
$contents = Storage::get('file.jpg');
```

php

El método `exists` puede ser usado para determinar si un archivo existe en el disco:

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

php

El método `missing` puede ser usado para determinar si falta un archivo en el disco:

```
$missing = Storage::disk('s3')->missing('file.jpg');
```

php

Descargando archivos

El método `download` puede ser usado para generar una respuesta que obliga al navegador del usuario a descargar el archivo al directorio dado. El método `download` acepta un nombre de archivo como segundo argumento del método, que determinará el nombre del archivo que es visto por el usuario descargando el archivo. Finalmente, puedes pasar un arreglo de encabezados HTTP como tercer argumento al método:

```
return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

php

URLs de archivos

Puedes usar el método `url` para obtener la URL del archivo dado. Si estás usando el driver `local`, esto típicamente agregará `/storage` a la ruta dada y retornará una URL relativa al archivo. Si estás usando el driver `s3` o `rackspace`, será retornada la URL remota completamente habilitada:

```
use Illuminate\Support\Facades\Storage;  
  
$url = Storage::url('file.jpg');
```

Nota

Recuerda, si estás usando el driver `local`, todos los archivos que deberían ser públicamente accesibles deben ser colocados en el directorio `storage/app/public`. Además, debes crear un enlace simbólico a `public/storage` que apunte al directorio `storage/app/public`.

URLs temporales

Para archivos almacenados usando los drivers `s3` o `rackspace`, puedes crear una URL temporal a un archivo dado usando el método `temporaryUrl`. Este método acepta una ruta y una instancia `DateTime` que especifica cuando la URL debería expirar:

```
$url = Storage::temporaryUrl(  
    'file.jpg', now()->addMinutes(5)  
);
```

Si necesitas especificar [parámetros de petición de S3](#) adicionales, puedes pasar el arreglo de parámetros de petición como tercer argumento del método `temporaryUrl`:

```
$url = Storage::temporaryUrl(  
    'file.jpg',  
    now()->addMinutes(5),  
    ['ResponseContentType' => 'application/octet-stream']  
);
```

Personalización del host de URL local

Si te gustaría predefinir el host para archivos almacenados en un disco usando el driver `local`, puedes agregar una opción `url` al arreglo de configuración del disco:

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

Metadatos de archivos

Además de leer y agregar archivos, Laravel también puede proporcionar información sobre los archivos. Por ejemplo, el método `size` puede ser usado para obtener el tamaño del archivo en bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');
```

El método `lastModified` retorna la marca de tiempo de UNIX de la última vez en que el archivo fue modificado:

```
$time = Storage::lastModified('file.jpg');
```

Almacenando archivos

El método `put` puede ser usado para almacenar el contenido de archivos en un disco. Puedes también pasar un `recurso` de PHP al método `put`, que usará el soporte subyacente de stream de Flysystem. Recuerda, todas las rutas de archivos deben ser especificadas de forma relativa a la ubicación "raíz" configurada en el disco:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

Streaming automático

Si te gustaría que Laravel automáticamente haga streaming de un archivo dado a tu ubicación de almacenamiento, puedes usar los métodos `putFile` o `putFileAs`. Este método acepta una instancia de `Illuminate\Http\File` o `Illuminate\Http\UploadedFile` y automáticamente hará stream del 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 algunas cosas importantes a tener en cuenta sobre el método `putFile`. Observa que sólo especificamos un nombre de directorio, no un nombre de archivo. Por defecto, el método `putFile` generará un ID único que servirá como nombre del archivo. La extensión del archivo será determinada examinando el tipo MIME del archivo. La ruta al archivo será retornada por el método `putFile` para que puedes almacenar la ruta, incluyendo el nombre de archivo generado, en tu base de datos.

Los métodos `putFile` y `putFileAs` también aceptan un argumento para especificar la "visibilidad" del archivo almacenado. Esto es particularmente útil si estás almacenando el archivo en disco en la nube como S3 y te gustaría que el archivo sea públicamente accesible:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

Añadir al inicio o al final de un archivo

Los métodos `prepend` y `append` te permiten escribir al inicio o final de un archivo:

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

Copiando y moviendo archivos

El método `copy` puede ser usado para copiar un archivo existente a una nueva ubicación en el disco, mientras que el método `move` puede ser usado para renombrar o mover un archivo existente a una nueva ubicación:

```
Storage::copy('old/file.jpg', 'new/file.jpg');  
  
Storage::move('old/file.jpg', 'new/file.jpg');
```

php

Carga de archivos

En las aplicaciones web, una de los casos de uso más comunes para almacenar archivos es almacenar archivos cargados por los usuarios como imágenes de perfil, fotos y documentos. Laravel hace que sea muy fácil almacenar archivos cargados usando el método `store` en la instancia de un archivo cargado. Llama al método `store` con la ruta en la quieras almacenar el archivo:

```
<?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;  
    }  
}
```

php

Hay algunas cosas importantes a tener en cuenta sobre este ejemplo. Observa que sólo especificamos un nombre de directorio, no un nombre de archivo. Por defecto, el método `store` generará un ID único que servirá como nombre de archivo. La extensión del archivo será determinada examinando el tipo MIME del archivo. La ruta al archivo será retornada por el método `store` para que puedas guardar la ruta, incluyendo el nombre generado, en tu base de datos.

También puedes llamar al método `putFile` en el facade `Storage` para realizar la misma manipulación de archivo del ejemplo superior:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

php

Especificando un nombre de archivo

Si no te gustaría que un nombre de archivo sea automáticamente asignado a tu archivo almacenado, puedes usar el método `storeAs`, que recibe una ruta, el nombre del archivo y el disco (opcional) y sus argumentos:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

php

Puedes usar el método `putFileAs` en el facade `Storage`, que realizará las mismas manipulaciones de archivos del ejemplo de arriba:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

php

Nota

Caracteres unicode invalidos y que no pueden ser mostrados serán automáticamente eliminados de rutas de archivos. Por lo tanto, podrías querer sanitizar las rutas de tus archivos antes de pasárlas a los métodos de almacenamiento de archivos de Laravel. Las rutas de archivos son normalizadas usando el método `League\Flysystem\Util::normalizePath`.

Especificando un disco

Por defecto, este método usará tu disco predeterminado. Si te gustaría especificar otro disco, pasa el nombre del disco como segundo argumento al método `store` :

```
$path = $request->file('avatar')->store(  
    'avatars' . $request->user()->id, 's3'  
)
```

php

Otra información de archivos

Si te gustaría obtener el nombre original del archivo cargado, puedes hacer esto usando el método

`getClientOriginalName` :

```
$name = $request->file('avatar')->getClientOriginalName();
```

php

El método `extension` puede ser usado para obtener la extensión del archivo cargado:

```
$extension = $request->file('avatar')->extension();
```

php

Visibilidad de archivos

En la integración de Flysystem de Laravel, "visibilidad" es una abstracción de permisos de archivos a través de múltiples plataformas. Los archivos pueden ser declarados tanto `public` o `private`.

Cuando un archivo es declarado `public`, estás indicando que el archivo debería ser generalmente accesible por otros. Por ejemplo, al usar el driver de S3, puedes retornar URLs para archivos `public`.

Puedes establecer la visibilidad al establecer el archivo mediante el método `put` :

```
use Illuminate\Support\Facades\Storage;  
  
Storage::put('file.jpg', $contents, 'public');
```

php

Si el archivo ya ha sido almacenado, su visibilidad puede ser retornada y establecida mediante los métodos `getVisibility` y `setVisibility` :

```
$visibility = Storage::getVisibility('file.jpg');
```

php

```
Storage::setVisibility('file.jpg', 'public');
```

Eliminando archivos

El método `delete` acepta un solo nombre de archivo o un arreglo de archivos a eliminar del disco:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::delete('file.jpg');  
  
Storage::delete(['file.jpg', 'file2.jpg']);
```

php

Si es necesario, puedes especificar el disco en el que se debe eliminar el archivo:

```
use Illuminate\Support\Facades\Storage;  
  
Storage::disk('s3')->delete('folder_path/file_name.jpg');
```

php

Directarios

Obtener todos los archivos dentro de un directorio

El método `files` retorna un arreglo de todos los archivos en un directorio dado. Si te gustaría retornar una lista de todos los archivos dentro de un directorio dado incluyendo subdirectorios, puedes usar el método `allFiles` :

```
use Illuminate\Support\Facades\Storage;  
  
$files = Storage::files($directory);  
  
$files = Storage::allFiles($directory);
```

php

Obtener todos los directorios dentro de un directorio

El método `directories` retorna un arreglo de todos los directorios dentro de un directorio dado. Adicionalmente, puedes usar el método `allDirectories` para obtener una lista de todos los

directorios dentro de un directorio dado y todos sus subdirectorios:

```
$directories = Storage::directories($directory);  
  
// Recursive...  
$directories = Storage::allDirectories($directory);
```

php

Crear un directorio

El método `makeDirectory` creará el directorio dado, incluyendo cualquier subdirectorio necesario:

```
Storage::makeDirectory($directory);
```

php

Eliminar un directorio

Finalmente, el método `deleteDirectory` puede ser usado para eliminar un directorio y todos sus archivos:

```
Storage::deleteDirectory($directory);
```

php

Sistemas de archivos personalizados

La integración de Flysystem de Laravel proporciona drivers para múltiples "drivers"; sin embargo, Flysystem no está limitado a estos y tiene adaptadores para muchos otros sistemas de almacenamiento. Puedes crear un driver personalizado si quieres usar alguno de los adaptadores adicionales en tu aplicación de Laravel.

Para configurar el sistema de archivos personalizado necesitarás un adaptador de Flysystem. Vamos a agregar un adaptador de Dropbox mantenido por la comunidad a nuestro proyecto:

```
composer require spatie/flysystem-dropbox
```

php

Luego, debes crear un [proveedor de servicios](#) como `DropboxServiceProvider`. En el método `boot` del proveedor, puedes usar el método `extend` del facade `Storage` para definir el driver personalizado:

```
<?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
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function ($app, $config) {
            $client = new DropboxClient(
                $config['authorization_token']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }
}
```

El primer argumento del método `extend` es el nombre del driver y el segundo es una Closure que recibe las variables `$app` y `$config`. La Closure resolver debe retornar una instancia de

`League\Flysystem\Filesystem`. La variable `$config` contiene los valores definidos en `config/filesystems.php` para el disco especificado.

Luego, registra el proveedor de servicios en tu archivo de configuración `config/app.php`:

```
'providers' => [
    // ...
    App\Providers\DropboxServiceProvider::class,
];
```

Una vez que has creado y registrado el proveedor de servicios de la extensión, puedes usar el driver `dropbox` en tu archivo de configuración `config/filesystems.php`.

Helpers

- [Introducción](#)
- [Métodos disponibles](#)

Introducción

Laravel incluye una variedad de funciones "helpers" globales de PHP. Muchas de esas funciones son usadas por el mismo framework; sin embargo, eres libre de usarlas en tus aplicaciones si lo encuentras conveniente.

Métodos disponibles

Arreglos & Objetos

[Arr::add](#)

[Arr::collapse](#)

[Arr::divide](#)

Arr::dot	Arr::last	Arr::sortRecursive
Arr::except	Arr::only	Arr::where
Arr::first	Arr::pluck	Arr::wrap
Arr::flatten	Arr::prepend	data_fill
Arr::forget	Arr::pull	data_get
Arr::get	Arr::random	data_set
Arr::has	Arr::set	head
Arr::isAssoc	Arr::sort	last

Rutas

app_path	database_path	resource_path
base_path	mix	storage_path
config_path	public_path	

Cadenas

--	Str::finish	Str::slug
class_basename	Str::is	Str::snake
e	Str::isUuid	Str::start
preg_replace_array	Str::kebab	Str::startsWith
Str::after	Str::limit	Str::studly
Str::afterLast	Str::orderedUuid	Str::title
Str::before	Str::plural	Str::uuid
Str::beforeLast	Str::random	Str::words
Str::camel	Str::replaceArray	trans
Str::contains	Str::replaceFirst	trans_choice
Str::containsAll	Str::replaceLast	
Str::endsWith	Str::singular	

URLs

action	route	secure_url
asset	secure_asset	url

Variados

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	throw_if
cache	info	throw_unless
class_uses_recursive	logger	today
collect	method_field	trait_uses_recursive
config	now	transform
cookie	old	validator
csrf_field	optional	value
csrf_token	policy	view
dd	redirect	with

Listado de Métodos

Arreglos & Objetos

Arr::add() {#collection-method .first-collection-method}

La función `Arr::add` agrega una clave / valor dada a un arreglo si la clave no existe previamente en el arreglo o existe pero con un valor `null` :

```
use Illuminate\Support\Arr;                                         php

$array = Arr::add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]

$array = Arr::add(['name' => 'Desk', 'price' => null], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

Arr::collapse() {#collection-method}

La función `Arr::collapse` colapsa un arreglo de arreglos en un único arreglo:

```
use Illuminate\Support\Arr;  
  
$array = Arr::collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);  
  
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

php

Arr::divide() {#collection-method}

La función `Arr::divide` retorna dos arreglos, uno contiene las claves y el otro contiene los valores del arreglo dado:

```
use Illuminate\Support\Arr;  
  
[$keys, $values] = Arr::divide(['name' => 'Desk']);  
  
// $keys: ['name']  
  
// $values: ['Desk']
```

php

Arr::dot() {#collection-method}

La función `Arr::dot()` aplana un arreglo multidimensional en un arreglo de un sólo nivel que usa la notación de "punto" para indicar la profundidad:

```
use Illuminate\Support\Arr;  
  
$array = ['products' => ['desk' => ['price' => 100]]];  
  
$flattened = Arr::dot($array);  
  
// ['products.desk.price' => 100]
```

php

Arr::except() {#collection-method}

La función `Arr::except()` remueve los pares clave / valor de un arreglo:

```
use Illuminate\Support\Arr;  
  
$array = ['name' => 'Desk', 'price' => 100];  
  
$filtered = Arr::except($array, ['price']);  
  
// ['name' => 'Desk']
```

php

`Arr::first()` {#collection-method}

La función `Arr::first()` devuelve el primer elemento de un arreglo que cumpla la condición dada:

```
use Illuminate\Support\Arr;  
  
$array = [100, 200, 300];  
  
$first = Arr::first($array, function ($value, $key) {  
    return $value >= 150;  
});  
  
// 200
```

php

Un valor por defecto puede ser pasado como un tercer parámetro al método. Este valor será retornado si no hay un valor que cumpla la condición:

```
use Illuminate\Support\Arr;  
  
$first = Arr::first($array, $callback, $default);
```

php

`Arr::flatten()` {#collection-method}

La función `Arr::flatten` unifica un arreglo multidimensional en un arreglo de un solo nivel:

```
use Illuminate\Support\Arr;  
  
$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];
```

php

```
$flattened = Arr::flatten($array);

// ['Joe', 'PHP', 'Ruby']
```

Arr::forget() {#collection-method}

La función `Arr::forget` remueve un par clave / valor de un arreglo anidado usando la notación de "punto":

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::forget($array, 'products.desk');

// ['products' => []]
```

php

Arr::get() {#collection-method}

La función `Arr::get` recupera un valor de un arreglo anidado usando la notación de "punto":

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$price = Arr::get($array, 'products.desk.price');

// 100
```

php

La función `Arr::get` acepta un valor por defecto, el cual será devuelto si la clave especificada no es encontrada:

```
use Illuminate\Support\Arr;

$discount = Arr::get($array, 'products.desk.discount', 0);

// 0
```

php

Arr::has() {#collection-method}

La función `Arr::has` comprueba si un elemento o elementos dados existen en un arreglo usando la notación de "punto":

`Arr::isAssoc()` {#collection-method}

`Arr::isAssoc` retorna `true` si el arreglo dado es un arreglo asociativo. Un arreglo es considerado "asociativo" si no tiene claves nmericas secuenciales comenzando por cero:

`Arr::last()` {#collection-method}

La función `Arr:::last` retorna el último elemento de un arreglo que cumpla la condición dada:

```
use Illuminate\Support\Arr;  
  
$array = [100, 200, 300, 110];
```

```
$last = Arr::last($array, function ($value, $key) {
    return $value >= 150;
});

// 300
```

Un valor por defecto puede ser pasado como tercer argumento al método. Este valor será devuelto si ningún valor cumple la condición:

```
use Illuminate\Support\Arr;

$last = Arr::last($array, $callback, $default);
```

php

Arr::only() {#collection-method}

La función `Arr::only` retorna solo el par clave / valor especificado del arreglo dado:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$slice = Arr::only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

php

Arr::pluck() {#collection-method}

La función `Arr::pluck` recupera todos los valores para una clave dada de un arreglo:

```
use Illuminate\Support\Arr;

$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$names = Arr::pluck($array, 'developer.name');
```

php

```
// ['Taylor', 'Abigail']
```

Puedes además especificar como deseas que la lista resultante sea codificada:

```
use Illuminate\Support\Arr;  
  
$names = Arr::pluck($array, 'developer.name', 'developer.id');  
  
// [1 => 'Taylor', 2 => 'Abigail']
```

php

Arr::prepend() {#collection-method}

La función `Arr::prepend` colocará un elemento al comienzo de un arreglo:

```
use Illuminate\Support\Arr;  
  
$array = ['one', 'two', 'three', 'four'];  
  
$array = Arr::prepend($array, 'zero');  
  
// ['zero', 'one', 'two', 'three', 'four']
```

php

Si es necesario, puedes especificar la clave que debería ser usada por el valor:

```
use Illuminate\Support\Arr;  
  
$array = ['price' => 100];  
  
$array = Arr::prepend($array, 'Desk', 'name');  
  
// ['name' => 'Desk', 'price' => 100]
```

php

Arr::pull() {#collection-method}

La función `Arr::pull` retorna y remueve un par clave / valor de un arreglo:

```
use Illuminate\Support\Arr;  
  
$array = ['name' => 'Desk', 'price' => 100];  
  
$name = Arr::pull($array, 'name');  
  
// $name: Desk  
  
// $array: ['price' => 100]
```

php

Un valor por defecto puede ser pasado como tercer argumento del método. Este valor será devuelto si la clave no existe:

```
use Illuminate\Support\Arr;  
  
$value = Arr::pull($array, $key, $default);
```

php

Arr::random() {#collection-method}

La función `Arr::random` retorna un valor aleatorio de un arreglo:

```
use Illuminate\Support\Arr;  
  
$array = [1, 2, 3, 4, 5];  
  
$random = Arr::random($array);  
  
// 4 - (retrieved randomly)
```

php

Puedes además especificar el número de elementos a retornar como un segundo argumento opcional. Nota que proveer este argumento retornará un arreglo, incluso si solo deseas un elemento:

```
use Illuminate\Support\Arr;  
  
$items = Arr::random($array, 2);  
  
// [2, 5] - (retrieved randomly)
```

php

Arr::set() {#collection-method}

La función `Arr::set` establece un valor dentro de un arreglo anidado usando la notación de "punto":

```
use Illuminate\Support\Arr;  
  
$array = ['products' => ['desk' => ['price' => 100]]];  
  
Arr::set($array, 'products.desk.price', 200);  
  
// ['products' => ['desk' => ['price' => 200]]]
```

php

Arr::sort() {#collection-method}

La función `Arr::sort` clasifica un arreglo por sus valores:

```
use Illuminate\Support\Arr;  
  
$array = ['Desk', 'Table', 'Chair'];  
  
$sorted = Arr::sort($array);  
  
// ['Chair', 'Desk', 'Table']
```

php

Puedes además clasificar el arreglo por los resultados de la función de retorno dada:

```
use Illuminate\Support\Arr;  
  
$array = [  
    ['name' => 'Desk'],  
    ['name' => 'Table'],  
    ['name' => 'Chair'],  
];  
  
$sorted = array_values(Arr::sort($array, function ($value) {  
    return $value['name'];  
}));  
  
/*  
[
```

php

```
        ['name' => 'Chair'],
        ['name' => 'Desk'],
        ['name' => 'Table'],
    ]
*/
```

Arr::sortRecursive() {#collection-method}

La función `array_sort_recursive` clasifica recursivamente un arreglo usando la función `sort` para sub-arreglos numéricos y `ksort` para sub-arreglos asociativos:

```
use Illuminate\Support\Arr;                                         php

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
    ['one' => 1, 'two' => 2, 'three' => 3],
];

$sorted = Arr::sortRecursive($array);

/*
[
    ['JavaScript', 'PHP', 'Ruby'],
    ['one' => 1, 'three' => 3, 'two' => 2],
    ['Li', 'Roman', 'Taylor'],
]
*/
```

Arr::where() {#collection-method}

La función `Arr::where` filtra un arreglo usando la función de retorno dada:

```
use Illuminate\Support\Arr;                                         php

$array = [100, '200', 300, '400', 500];

$filtered = Arr::where($array, function ($value, $key) {
    return is_string($value);
});
```

```
// [1 => '200', 3 => '400']
```

Arr::wrap() {#collection-method}

La función `Arr::wrap` envuelve el valor dado en un arreglo. Si el valor dado ya es un arreglo este no será cambiado:

```
use Illuminate\Support\Arr;  
  
$string = 'Laravel';  
  
$array = Arr::wrap($string);  
  
// ['Laravel']
```

Si el valor dado es nulo, un arreglo vacío será devuelto:

```
use Illuminate\Support\Arr;  
  
$nothing = null;  
  
$array = Arr::wrap($nothing);  
  
// []
```

data_fill() {#collection-method}

La función `data_fill` establece un valor faltante dentro de un arreglo 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 además 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()` {#collection-method}

La función `data_get` recupera un valor de un arreglo anidado u objeto usando la notación de "punto":

```
$data = ['products' => ['desk' => ['price' => 100]]];

$price = data_get($data, 'products.desk.price');

// 100
```

La función `data_get` acepta además un valor por defecto, el cual será retornado si la clave especificada no es encontrada:

```
$discount = data_get($data, 'products.desk.discount', 0);

// 0
```

La función también acepta wildcards usando asteriscos, que pueden tener como objetivo cualquier clave del arreglo u objeto:

```
$data = [  
    'product-one' => ['name' => 'Desk 1', 'price' => 100],  
    'product-two' => ['name' => 'Desk 2', 'price' => 150],  
];  
  
data_get($data, '*.*.name');  
  
// ['Desk 1', 'Desk 2'];
```

php

`data_set()` {#collection-method}

La función `data_set` establece un valor dentro de un arreglo anidado u objeto usando la notación de "punto":

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_set($data, 'products.desk.price', 200);  
  
// ['products' => ['desk' => ['price' => 200]]]
```

php

Esta función además acepta comodines y establecerá 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],  
    ],  
];
```

php

```
        ['name' => 'Desk 2', 'price' => 200],  
    ],  
*/
```

Por defecto, cualquier valor existente es sobrescrito. Si deseas solo establecer un valor si no existe, puedes pasar `false` como cuarto argumento:

```
$data = ['products' => ['desk' => ['price' => 100]]];  
  
data_set($data, 'products.desk.price', 200, false);  
  
// ['products' => ['desk' => ['price' => 100]]]
```

php

head() {#collection-method}

La función `head` retorna el primer elemento en el arreglo dado:

```
$array = [100, 200, 300];  
  
$first = head($array);  
  
// 100
```

php

last() {#collection-method}

La función `last` retorna el último elemento en el arreglo dado:

```
$array = [100, 200, 300];  
  
$last = last($array);  
  
// 300
```

php

Rutas

app_path() {#collection-method}

La función `app_path` retorna la ruta completa al directorio `app`. Además puedes usar la función `app_path` para generar una ruta completa a un archivo relativo al directorio de la aplicación:

```
$path = app_path();  
  
$path = app_path('Http/Controllers/Controller.php');
```

`base_path()` {#collection-method}

La función `base_path` retorna la ruta completa a la raíz del proyecto. Además puedes usar la función `base_path` para generar una ruta completa a un archivo dado relativo al directorio raíz del proyecto:

```
$path = base_path();  
  
$path = base_path('vendor/bin');
```

`config_path()` {#collection-method}

La función `config_path` retorna la ruta completa al directorio `config`. Puedes además usar la función `config_path` para generar una ruta completa a un archivo dado dentro del directorio de configuración de la aplicación:

```
$path = config_path();  
  
$path = config_path('app.php');
```

`database_path()` {#collection-method}

La función `database_path` retorna la ruta completa al directorio `database`. Puedes además usar la función `database_path` para generar una ruta completa a un archivo dado dentro del directorio `database`:

```
$path = database_path();  
  
$path = database_path('factories/UserFactory.php');
```

`mix()` {#collection-method}

La función `mix` retorna la ruta al archivo versionado Mix:

```
$path = mix('css/app.css');
```

php

`public_path()` {#collection-method}

La función `public_path` retorna la ruta completa al directorio `public`. Puedes además usar la función `public_path` para generar una ruta completa a un archivo dado dentro del directorio `public`:

```
$path = public_path();  
  
$path = public_path('css/app.css');
```

php

`resource_path()` {#collection-method}

La función `resource_path` retorna la ruta completa al directorio `resources`. Puedes además usar la función `resource_path` para generar una ruta completa a un archivo dado dentro del directorio `resources`:

```
$path = resource_path();  
  
$path = resource_path('sass/app.scss');
```

php

`storage_path()` {#collection-method}

La función `storage_path` retorna la ruta completa al directorio `storage`. Puedes además usar la función `storage_path` para generar una ruta completa a un archivo dado dentro del directorio `storage`:

```
$path = storage_path();  
  
$path = storage_path('app/file.txt');
```

php

Cadenas

`__()` {#collection-method}

La función `__` traduce la cadena de traducción dada o clave de traducción dada usando tus archivos de localización:

```
echo __('Welcome to our application');  
echo __('messages.welcome');
```

php

Si la cadena o llave de traducción especificada no existe, la función `__` retornará el valor dado. Así, usando el ejemplo de arriba, la función `__` podría retornar `messages.welcome` si esa clave de traducción no existe.

`class_basename()` {#collection-method}

La función `class_basename` retorna el nombre de la clase dada con el espacio de nombre de la clase removido:

```
$class = class_basename('Foo\Bar\Baz');  
  
// Baz
```

php

`e()` {#collection-method}

La función `e` ejecuta la función de PHP `htmlspecialchars` con la opción `double_encode` establecida establecida a `true` por defecto:

```
echo e('<html>foo</html>');  
  
// &lt;html&ampgtfoo&lt;/html&ampgt
```

php

`preg_replace_array()` {#collection-method}

La función `preg_replace_array` reemplaza un patrón dado en la cadena secuencialmente usando un arreglo:

```
$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
```

php

`Str::after()` {#collection-method}

La función `Str::after` retorna todo después del valor dado en una cadena. La cadena entera será retornada si el valor no existe dentro de la cadena:

```
use Illuminate\Support\Str;  
  
$slice = Str::after('This is my name', 'This is');  
  
// ' my name'
```

php

`Str::afterLast()` {#collection-method}

El método `Str::afterLast` retorna todo luego de la última ocurrencia del valor dado en una cadena. La cadena entera será retornada si el valor no existe dentro de la cadena:

```
use Illuminate\Support\Str;  
  
$slice = Str::afterLast('App\Http\Controllers\Controller', '\\');  
  
// 'Controller'
```

php

`Str::before()` {#collection-method}

La función `Str::before` retorna todo antes del valor dado en una cadena:

```
use Illuminate\Support\Str;
```

php

```
$slice = Str::before('This is my name', 'my name');

// 'This is '
```

Str::beforeLast() {#collection-method}

El método `Str::beforeLast` retorna todo antes de la última ocurrencia del valor dado en una cadena:

```
use Illuminate\Support\Str;

$slice = Str::beforeLast('This is my name', 'is');

// 'This '
```

php

Str::camel() {#collection-method}

La función `Str::camel` convierte la cadena dada a `camelCase`:

```
use Illuminate\Support\Str;

$converted = Str::camel('foo_bar');

// fooBar
```

php

Str::contains() {#collection-method}

La función `Str::contains` determina si la cadena dada contiene el valor dado (sensible a mayúsculas y minúsculas):

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', 'my');

// true
```

php

Puedes además pasar un arreglo de valores para determinar si la cadena dada contiene cualquiera de los valores:

```
use Illuminate\Support\Str;  
  
$contains = Str::contains('This is my name', ['my', 'foo']);  
  
// true
```

php

Str::containsAll() {#collection-method}

El método `Str::containsAll` determina si la cadena dada contiene todos los valores del arreglo:

```
use Illuminate\Support\Str;  
  
$containsAll = Str::containsAll('This is my name', ['my', 'name']);  
  
// true
```

php

Str::endsWith() {#collection-method}

La función `Str::endsWith` determina si la cadena dada finaliza con el valor dado:

```
use Illuminate\Support\Str;  
  
$result = Str::endsWith('This is my name', 'name');  
  
// true
```

php

También puedes pasar un arreglo de valores para determinar si la cadena dada termina con alguno de los valores dados:

```
use Illuminate\Support\Str;  
  
$result = Str::endsWith('This is my name', ['name', 'foo']);  
  
// true
```

php

```
$result = Str::endsWith('This is my name', ['this', 'foo']);  
  
// false
```

Str::finish() {#collection-method}

La función `Str::finish` agrega una instancia individual del valor dado a una cadena si éste no finaliza con el valor:

```
use Illuminate\Support\Str;  
  
$adjusted = Str::finish('this/string', '/');  
  
// this/string/  
  
$adjusted = Str::finish('this/string/', '/');  
  
// this/string/
```

php

Str::is() {#collection-method}

La función `Str::is` determina si una cadena dada concuerda con un patrón dado. Asteriscos pueden ser usados para indicar comodines:

```
use Illuminate\Support\Str;  
  
$matches = Str::is('foo*', 'foobar');  
  
// true  
  
$matches = Str::is('baz*', 'foobar');  
  
// false
```

php

Str::isUuid() {#collection-method}

El método `Str::isUuid` determina si la cadena dada es un UUID valido:

```
use Illuminate\Support\Str;  
  
$isUuid = Str::isUuid('a0a2a2d2-0b87-4a18-83f2-2529882be2de');  
  
// true  
  
$isUuid = Str::isUuid('laravel');  
  
// false
```

php

Str::kebab() {#collection-method}

La función `Str::kebab` convierte la cadena dada a `kebab-case`:

```
use Illuminate\Support\Str;  
  
$converted = Str::kebab('fooBar');  
  
// foo-bar
```

php

Str::limit() {#collection-method}

La función `Str::limit` trunca la cadena dada en la longitud especificada:

```
use Illuminate\Support\Str;  
  
$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20);  
  
// The quick brown fox...
```

php

Puedes además pasar un tercer argumento para cambiar la cadena que será adjuntada al final:

```
use Illuminate\Support\Str;  
  
$truncated = Str::limit('The quick brown fox jumps over the lazy dog', 20, ' (...');  
  
// The quick brown fox (...)
```

php

`Str::orderedUuid` {#collection-method}

El método `Str::orderedUuid` genera una "primera marca de tiempo" UUID que puede ser eficientemente almacenada en una columna indexada de la base de datos:

```
use Illuminate\Support\Str;  
  
return (string) Str::orderedUuid();
```

php

`Str::plural()` {#collection-method}

La función `Str::plural` convierte una cadena de una única palabra a su forma plural. Esta función actualmente solo soporta el idioma inglés:

```
use Illuminate\Support\Str;  
  
$plural = Str::plural('car');  
  
// cars  
  
$plural = Str::plural('child');  
  
// children
```

php

Puedes además proporcionar un entero como segundo argumento a la función para recuperar la forma singular o plural de la cadena:

```
use Illuminate\Support\Str;  
  
$plural = Str::plural('child', 2);  
  
// children  
  
$plural = Str::plural('child', 1);  
  
// child
```

php

`Str::random()` {#collection-method}

La función `Str::random` genera una cadena aleatoria con la longitud especificada. Esta función usa la función PHP `random_bytes` :

```
use Illuminate\Support\Str;  
  
$random = Str::random(40);
```

php

`Str::replaceArray()` {#collection-method}

La función `Str::replaceArray` reemplaza un valor dado en la cadena secuencialmente usando un arreglo:

```
use Illuminate\Support\Str;  
  
$string = 'The event will take place between ? and ?';  
  
$replaced = Str::replaceArray('?', ['8:30', '9:00'], $string);  
  
// The event will take place between 8:30 and 9:00
```

php

`Str::replaceFirst()` {#collection-method}

La función `Str::replaceFirst` reemplaza la primera ocurrencia de un valor dado en una cadena:

```
use Illuminate\Support\Str;  
  
$replaced = Str::replaceFirst('the', 'a', 'the quick brown fox jumps over the la  
// a quick brown fox jumps over the lazy dog
```

php

`Str::replaceLast()` {#collection-method}

La función `Str::replaceLast` reemplaza la última ocurrencia de un valor dado en una cadena:

```
use Illuminate\Support\Str;  
  
$replaced = Str::replaceLast('the', 'a', 'the quick brown fox jumps over the laz
```

php

```
// the quick brown fox jumps over a lazy dog
```

Str::singular() {#collection-method}

La función `Str::singular` convierte una cadena a su forma singular. Esta función actualmente solo soporta el idioma inglés:

```
use Illuminate\Support\Str;  
  
$singular = Str::singular('cars');  
  
// car  
  
$singular = Str::singular('children');  
  
// child
```

php

Str::slug() {#collection-method}

La función `Str::slug` genera una URL amigable con la cadena dada:

```
use Illuminate\Support\Str;  
  
$slug = Str::slug('Laravel 5 Framework', '-');  
  
// laravel-5-framework
```

php

Str::snake() {#collection-method}

La función `Str::snake()` convierte la cadena dada a `snake_case`:

```
use Illuminate\Support\Str;  
  
$converted = Str::snake('fooBar');  
  
// foo_bar
```

php

`Str::start()` {#collection-method}

La función `Str::start` agrega una instancia individual del valor dado a una cadena si ésta no inicia con ese valor:

```
use Illuminate\Support\Str;                                         php

$adjusted = Str::start('this/string', '/');

// /this/string

$adjusted = Str::start('/this/string', '/');

// /this/string
```

`Str::startsWith()` {#collection-method}

La función `Str::startsWith` determina si la cadena dada comienza con el valor dado:

```
use Illuminate\Support\Str;                                         php

$result = Str::startsWith('This is my name', 'This');

// true
```

`Str::studly()` {#collection-method}

La función `Str::studly` convierte la cadena dada a `StudlyCase`:

```
use Illuminate\Support\Str;                                         php

$converted = Str::studly('foo_bar');

// FooBar
```

`Str::title()` {#collection-method}

La función `Str::title` convierte la cadena dada a `Title Case`:

```
use Illuminate\Support\Str;  
  
$converted = Str::title('a nice title uses the correct case');  
  
// A Nice Title Uses The Correct Case
```

php

Str::uuid() {#collection-method}

El método `Str::uuid` genera un UUID (versión 4):

```
use Illuminate\Support\Str;  
  
return (string) Str::uuid();
```

php

Str::words() {#collection-method}

El método `Str::words` limita el número de palabras en una cadena:

```
use Illuminate\Support\Str;  
  
return Str::words('Perfectly balanced, as all things should be.', 3, ' >>');  
  
// Perfectly balanced, as >>
```

php

trans() {#collection-method}

La función `trans` traduce la clave de traducción dada usando tus archivos de localización:

```
echo trans('messages.welcome');
```

php

Si la clave de traducción especificada no existe, la función `trans` retornará la clave dada. Así, usando el ejemplo de arriba, la función `trans` podría retornar `messages.welcome` si la clave de traducción no existe.

trans_choice() {#collection-method}

La función `trans_choice` traduce la clave de traducción dada con inflexión:

```
echo trans_choice('messages.notifications', $unreadCount);
```

php

Si la clave de traducción dada no existe, la función `trans_choice` retornará la clave dada. Así, usando el ejemplo de arriba, la función `trans_choice` podría retornar `messages.notifications` si la clave de traducción no existe.

URLs

`action() {#collection-method}`

La función `action` genera una URL para la acción del controlador dada. No necesitas pasar el espacio de nombre completo. En lugar de eso, pasa al controlador el nombre de clase relativo al espacio de nombre `App\Http\Controllers`:

```
$url = action('HomeController@index');  
  
$url = action([HomeController::class, 'index']);
```

php

Si el método acepta parámetros de ruta, puedes pasarlos como segundo argumento al método:

```
$url = action('UserController@profile', ['id' => 1]);
```

php

`asset() {#collection-method}`

La función `asset` genera una URL para un asset usando el esquema actual de la solicitud (HTTP o HTTPS):

```
$url = asset('img/photo.jpg');
```

php

Puedes configurar la URL host del asset estableciendo la variable `ASSET_URL` en tu archivo `.env`. Esto puede ser útil si alojas tus assets en un servicio externo como Amazon S3:

```
// ASSET_URL=http://example.com/assets  
  
$url = asset('img/photo.jpg'); // http://example.com/assets/img/photo.jpg
```

php

route() {#collection-method}

La función `route` genera una URL para el nombre de ruta dado:

```
$url = route('routeName');
```

php

Si la ruta acepta parámetros, puedes pasarlos como segundo argumento al método:

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

php

Por defecto, la función `route` genera una URL absoluta. Si deseas generar una URL relativa, puedes pasar `false` como tercer argumento:

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

php

secure_asset() {#collection-method}

La función `secure_asset` genera una URL para un asset usando HTTPS:

```
$url = secure_asset('img/photo.jpg');
```

php

secure_url() {#collection-method}

La función `secure_url` genera una URL HTTPS completa a la ruta dada:

```
$url = secure_url('user/profile');  
  
$url = secure_url('user/profile', [1]);
```

php

url() {#collection-method}

La función `url` genera una URL completa a la ruta dada:

```
$url = url('user/profile');  
  
$url = url('user/profile', [1]);
```

php

Si una ruta no es proporcionada, una instancia de `Illuminate\Routing\UrlGenerator` es retornada:

```
$current = url()->current();  
  
$full = url()->full();  
  
$previous = url()->previous();
```

php

Variados

`abort()` {#collection-method}

La función `abort` arroja una excepción HTTP que será renderizada por el manejador de excepciones:

```
abort(403);
```

php

Puedes además proporcionar el texto de respuesta de la excepción y las cabeceras de la respuesta personalizados:

```
abort(403, 'Unauthorized.', $headers);
```

php

`abort_if()` {#collection-method}

La función `abort_if` arroja una excepción HTTP si una expresión booleana dada es evaluada a `true`:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

php

Como el método `abort`, puedes proporcionar además el texto de respuesta para la excepción como tercer argumento y un arreglo de cabeceras de respuesta personalizadas como cuarto argumento.

`abortUnless()` {#collection-method}

La función `abortUnless` arroja una excepción HTTP si una expresión booleana dada es evaluada a `false`:

```
abortUnless(Auth::user()->isAdmin(), 403);
```

php

Como el método `abort`, puedes proporcionar además el texto de respuesta para la excepción como tercer argumento y un arreglo de cabeceras de respuesta personalizadas como cuarto argumento.

`app()` {#collection-method}

La función `app` retorna la instancia del contenedor de servicio:

```
$container = app();
```

php

Puedes pasar una clase o nombre de interfaz para resolverlo desde el contenedor:

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

php

`auth()` {#collection-method}

La función `auth` retorna una instancia del autenticador. Puedes usarla en vez del facade `Auth` por conveniencia:

```
$user = auth()->user();
```

php

Si es necesario, puedes especificar con cual instancia del guard podrías acceder:

```
$user = auth('admin')->user();
```

php

`back()` {#collection-method}

La función `back` genera una respuesta de redirección HTTP a la ubicación previa del usuario:

```
return back($status = 302, $headers = [], $fallback = false);  
  
return back();
```

php

`bcrypt()` {#collection-method}

La función `bcrypt` encripta el valor dado usando Bcrypt. Puedes usarlo como una alternativa al facade `Hash`:

```
$password = bcrypt('my-secret-password');
```

php

`blank()` {#collection-method}

La función `blank` retorna `true` si el valor dado es "vacío":

```
blank('');  
blank(' ');  
blank(null);  
blank(collect());  
  
// true  
  
blank(0);  
blank(true);  
blank(false);  
  
// false
```

php

Para lo inverso de `blank`, mira el método `filled`.

`broadcast()` {#collection-method}

La función `broadcast` emite el evento dado a sus listeners:

```
broadcast(new UserRegistered($user));
```

php

cache() {#collection-method}

La función `cache` puede ser usada para obtener un valor de la cache. Si la clave dada no existe en la cache, un valor opcional por defecto será retornado:

```
$value = cache('key');
```

php

```
$value = cache('key', 'default');
```

Puedes agregar elementos a la cache pasando un arreglo de pares clave / valor a la función. También debes pasar la cantidad de segundos o la duración que el valor almacenado en caché debe considerarse válido:

```
cache(['key' => 'value'], 300);
```

php

```
cache(['key' => 'value'], now()->addSeconds(10));
```

class_uses_recursive() {#collection-method}

La función `class_uses_recursive` retorna todos los traits usados por una clase, incluyendo traits por todas las clases padre:

```
$traits = class_uses_recursive(App\User::class);
```

php

collect() {#collection-method}

La función `collect` crea una instancia de colecciones del valor dado:

```
$collection = collect(['taylor', 'abigail']);
```

php

config() {#collection-method}

La función `config` obtiene el valor de una variable de configuración. Los valores de configuración pueden ser accesados usando la sintaxis de "punto", la cual incluye el nombre del archivo y la opción que deseas acceder. Un valor por defecto puede ser especificado y es retornado si la opción de configuración no existe:

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

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

php

Puedes establecer variables de configuración en tiempo de ejecución pasando un arreglo de pares clave / valor:

```
config(['app.debug' => true]);
```

php

`cookie()` {#collection-method}

La función `cookie` crea una nueva instancia de cookie:

```
$cookie = cookie('name', 'value', $minutes);
```

php

`csrf_field()` {#collection-method}

La función `csrf_field` genera un campo de entrada `hidden` que contiene el valor del token CSRF. Por ejemplo, usando la sintaxis de Blade:

```
{{ csrf_field() }}
```

php

`csrf_token()` {#collection-method}

La función `csrf_token` recupera el valor del actual token CSRF:

```
$token = csrf_token();
```

php

`dd()` {#collection-method}

La función `dd` desecha las variables dadas y finaliza la ejecución del script:

```
dd($value);
```

php

```
dd($value1, $value2, $value3, ...);
```

Si no quieres detener la ejecución de tu script, usa la función `dump` en su lugar.

`decrypt()` {#collection-method}

La función `decrypt` desencripta el valor dado usando el encriptador de Laravel:

```
$decrypted = decrypt($encrypted_value);
```

php

`dispatch()` {#collection-method}

La función `dispatch` empuja el trabajo dado sobre la cola de trabajos de Laravel:

```
dispatch(new App\Jobs\SendEmails);
```

php

`dispatch_now()` {#collection-method}

La función `dispatch_now` ejecuta el trabajo dado inmediatamente y retorna el valor de su método

`handle` :

```
$result = dispatch_now(new App\Jobs\SendEmails);
```

php

`dump()` {#collection-method}

La función `dump` desecha las variables dadas:

```
dump($value);

dump($value1, $value2, $value3, ...);
```

php

Si quieras parar de ejecutar el script después de desechar las variables, usa la función `dd` en su lugar.

`encrypt()` {#collection-method}

La función `encrypt` encripta el valor dado usando el encriptador de Laravel:

```
$encrypted = encrypt($unencrypted_value);
```

php

env() {#collection-method}

La función `env` recupera el valor de una variable de entorno o retorna un valor por defecto:

```
$env = env('APP_ENV');

// Returns 'production' if APP_ENV is not set...
$env = env('APP_ENV', 'production');
```

php

Nota

Si ejecutas el comando `config:cache` durante tu proceso de despliegue, deberías estar seguro de que eres el único llamando a la función `env` desde dentro de tus archivos de configuración. Una vez que la configuración está en caché, el archivo `.env` no será cargado y todas las llamadas a la función `.env` retornarán `null`.

event() {#collection-method}

La función `event` despacha el `evento` dado a sus listeners:

```
event(new UserRegistered($user));
```

php

factory() {#collection-method}

La función `factory` crea un constructor de model factories para una clase dada, nombre y cantidad. Este puede ser usado mientras `pruebas` o haces `seeding`:

```
$user = factory(App\User::class)->make();
```

php

filled() {#collection-method}

La función `filled` retorna el valor dado que no esté "vacío":

```
filled(0);  
filled(true);  
filled(false);  
  
// true  
  
filled('');  
filled('   ');  
filled(null);  
filled(collect());  
  
// false
```

php

Para el inverso de `filled`, mira el método `blank`.

`info()` {#collection-method}

La función `info` escribirá información al `log`:

```
info('Some helpful information!');
```

php

Un arreglo de datos contextuales puede además ser pasado a la función:

```
info('User login attempt failed.', ['id' => $user->id]);
```

php

`logger()` {#collection-method}

La función `logger` puede ser usada para escribir mensaje de nivel `debug` al `log`:

```
logger('Debug message');
```

php

Un arreglo de datos contextuales puede además ser pasado a la función:

```
logger('User has logged in.', ['id' => $user->id]);
```

php

Una instancia del `logger` será retornada si no hay un valor pasado a la función:

```
logger()->error('You are not allowed here.');
```

php

method_field() {#collection-method}

La función `method_field` genera un campo de entrada HTML `hidden` que contiene el valor falsificado del verbo de los formularios HTTP. Por ejemplo, usando la [sintaxis de Blade](#):

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

php

now() {#collection-method}

La función `now` crea una nueva instancia `Illuminate\Support\Carbon` con la hora actual:

```
$now = now();
```

php

old() {#collection-method}

La función `old` recupera un [viejo valor de entrada](#) flasheado en la sesión:

```
$value = old('value');

$value = old('value', 'default');
```

php

optional() {#collection-method}

La función `optional` acepta cualquier argumento y te permite acceder a propiedades o métodos de llamada en ese objeto. Si el objeto dado es `null`, las propiedades y métodos retornarán `null` en vez de causar un error:

```
return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}
```

php

La función `optional` también acepta un Closure como segundo argumento. El Closure será invocado si el valor proporcionado como primer argumento no es null:

```
return optional(User::find($id), function ($user) {
    return new DummyUser;
});
```

php

policy() {#collection-method}

El método `policy` recupera una instancia de la [política](#) para una clase dada:

```
$policy = policy(App\User::class);
```

php

redirect() {#collection-method}

La función `redirect` retorna una [respuesta de redirección HTTP](#) o retorna la instancia del redirector si no hay argumentos llamados:

```
return redirect($to = null, $status = 302, $headers = [], $secure = null);

return redirect('/home');

return redirect()->route('route.name');
```

php

report() {#collection-method}

La función `report` reportará una excepción usando el método `report` de tu [manejador de excepciones](#):

```
report($e);
```

php

request() {#collection-method}

La función `request` retorna la instancia de la [solicitud](#) actual u obtiene un elemento de entrada:

```
$request = request();
```

php

```
$value = request('key', $default);
```

rescue() {#collection-method}

La función `rescue` ejecuta la función de retorno dada y almacena en cache cualquier excepción que ocurra durante su ejecución. Todas las excepciones que son capturadas serán enviadas al método `report` de tu [manejador de excepciones](#); no obstante, la solicitud continuará procesando:

```
return rescue(function () {
    return $this->method();
});
```

php

También puedes pasar un segundo argumento a la función `rescue`. Este argumento será el valor por "defecto" que debería ser retornado si una excepción ocurre mientras se ejecuta la función de retorno:

```
return rescue(function () {
    return $this->method();
}, false);

return rescue(function () {
    return $this->method();
}, function () {
    return $this->failure();
});
```

php

resolve() {#collection-method}

La función `resolve` resuelve un nombre de clase o interfaz dado a su instancia usando el[contenedor de servicios](#):

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

php

response() {#collection-method}

La función `response` crea una instancia de [respuesta](#) u obtiene una instancia del factory de respuesta:

```
return response('Hello World', 200, $headers);  
  
return response()->json(['foo' => 'bar'], 200, $headers);
```

php

retry() {#collection-method}

La función `retry` intenta ejecutar la función de retorno dada hasta que el máximo número de intentos límite se cumple. Si la función de retorno no arroja una excepción, su valor de retorno será retornado. Si la función de retorno arroja una excepción, se volverá a intentar automáticamente. Si el máximo número de intentos es excedido, la excepción será arrojada:

```
return retry(5, function () {  
    // Attempt 5 times while resting 100ms in between attempts...  
}, 100);
```

php

session() {#collection-method}

La función `session` puede ser usada para obtener o establecer valores de `session`:

```
$value = session('key');
```

php

Puedes establecer valores pasando un arreglo de pares clave / valor a la función:

```
session(['chairs' => 7, 'instruments' => 3]);
```

php

La sesión almacenada será retornada si no se pasa un valor a la función:

```
$value = session()->get('key');  
  
session()->put('key', $value);
```

php

tap() {#collection-method}

La función `tap` acepta dos argumentos: un `$value` arbitrario y una función de retorno. El `$value` será pasado a la función de retorno y será retornado por la función `tap`. El valor de retorno de la

función de retorno es irrelevante:

```
$user = tap(User::first(), function ($user) {
    $user->name = 'taylor';

    $user->save();
});
```

php

Si no hay función de retorno para la función `tap`, puedes llamar cualquier método en el `$value` dado. El valor de retorno del método al que llama siempre será `$value`, sin importar lo que el método retorna en su definición. Por ejemplo, el método de Eloquent `update` típicamente retorna un entero. Sin embargo, podemos forzar que el método retorne el modelo en sí mismo encadenando el método `update` a través de la función `tap`:

```
$user = tap($user)->update([
    'name' => $name,
    'email' => $email,
]);
```

php

Para agregar un método `tap` a una clase, puedes agregar el trait `Illuminate\Support\Traits\Tappable` a la clase. El método `tap` de este trait acepta un Closure como único argumento. La instancia del objeto será pasada al Closure y luego retornada por el método `tap`:

```
return $user->tap(function ($user) {
    //
});
```

php

`throw_if()` {#collection-method}

La función `throw_if` arroja la excepción dada si una expresión booleana dada es evaluada a `true`:

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw_if(
    ! Auth::user()->isAdmin(),
    AuthorizationException::class,
```

php

```
'You are not allowed to access this page'  
);
```

throw_unless() {#collection-method}

La función `throw_unless` arroja la excepción dada si una expresión booleana dada es evaluada a `false`:

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);  
  
throw_unless(  
    Auth::user()->isAdmin(),  
    AuthorizationException::class,  
    'You are not allowed to access this page'  
);
```

php

today() {#collection-method}

La función `today` crea una nueva instancia de `\Illuminate\Support\Carbon` para la fecha actual:

```
$today = today();
```

php

trait_uses_recursive() {#collection-method}

La función `trait_uses_recursive` retorna todos los traits usados por un trait:

```
$traits = trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

php

transform() {#collection-method}

La función `transform` ejecuta una función de retorno en un valor dado si el valor no está en `vacio` y retorna el resultado de la función de retorno:

```
$callback = function ($value) {  
    return $value * 2;  
};  
  
$result = transform(5, $callback);
```

php

```
// 10
```

Un valor o `closure` puede ser pasado como el tercer parámetro al método. Este valor será retornado si el valor dado está vacío:

```
$result = transform(null, $callback, 'The value is blank');  
  
// The value is blank
```

php

`validator()` `{#collection-method}`

La función `validator` crea un nueva instancia del `validador` con los argumentos dados. Puedes usarlo en vez del facade `Validator` por conveniencia:

```
$validator = validator($data, $rules, $messages);
```

php

`value()` `{#collection-method}`

La función `value` retorna el valor dado. Sin embargo, si pasas un `Closure` a la función, el `Closure` será ejecutado y su resultado será devuelto:

```
$result = value(true);
```

php

```
// true
```

```
$result = value(function () {  
    return false;  
});
```

```
// false
```

`view()` `{#collection-method}`

La función `view` recupera una instancia de la `vista`:

```
return view('auth.login');
```

php

with() {#collection-method}

La función `with` retorna el valor dado. Si se le pasa un `Closure` como segundo argumento a la función, el `Closure` será ejecutado y su resultado será devuelto:

```
$callback = function ($value) {
    return (is_numeric($value)) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

php

Correos Electrónicos

- Introducción
 - Requisitos previos
- Generando mailables
- Escribiendo mailables
 - Configurando el envío
 - Configurando la vista
 - Datos en vistas

- Archivos adjuntos
- Archivos adjuntos en línea
- Personalizar el mensaje de swiftMailer
- Mailables en markdown
 - Generando mailables en markdown
 - Escribiendo mensajes en markdown
 - Personalizando los componentes
- Enviando correo
 - Colas de correos
- Renderizando mailables
 - Previsualizando mailables en el navegador
- Configuración regional de mailables
- Correos y desarrollo Local
- Eventos

Introducción

Laravel proporciona una API limpia y simple sobre la popular biblioteca [SwiftMailer](#) con drivers para SMTP, Mailgun, Postmark, Amazon SES y [sendmail](#), permitiéndote comenzar rápidamente a enviar correos a través de un servicio local o en la nube de tu elección.

Requisitos previos

Los drivers basados en una API como Mailgun y Postmark suelen ser más simples y rápidos que los servidores SMTP. Si es posible, deberías usar uno de estos drivers. Todos los drivers con API requieren la biblioteca Guzzle HTTP, que puede instalarse a través del gestor de paquetes Composer:

```
composer require guzzlehttp/guzzle
```

php

Driver Mailgun

Para usar el driver de Mailgun, primero instale Guzzle, luego configura la opción [driver](#) en tu archivo de configuración [config/mail.php](#) en [mailgun](#). Luego, verifica que tu archivo de configuración [config/services.php](#) contiene las siguientes opciones:

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

php

Si no estás usando la [región de Mailgun](#) "US", puedes definir el endpoint de tu región en el archivo de configuración `services`:

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
    'endpoint' => 'api.eu.mailgun.net',
],
```

php

Driver Postmark

Para usar el driver de Postmark, instala el transporte de SwiftMailer de Postmark mediante Composer:

```
composer require wildbit/swiftmailer-postmark
```

php

Luego, instala Guzzle y establece la opción `driver` en tu archivo de configuración `config/mail.php` a `postmark`. Finalmente, verifica que tu archivo de configuración `config/services.php` contiene las siguientes opciones:

```
'postmark' => [
    'token' => 'your-postmark-token',
],
```

php

Driver SES

Para usar el driver de Amazon SES, primero debes instalar Amazon AWS SDK para PHP. Puedes instalar esta biblioteca agregando la siguiente línea a la sección `require` del archivo `composer.json` y ejecutando el comando `composer update`:

```
"aws/aws-sdk-php": "~3.0"
```

php

A continuación, configura la opción `driver` en tu archivo de configuración `config/mail.php` en `ses` y verifica que tu 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
],
```

Si necesitas incluir [opciones adicionales](#) al ejecutar la petición `SendRawEmail` de SES, puedes definir un arreglo `options` dentro de tu configuración de `ses`:

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region', // e.g. us-east-1
    'options' => [
        'ConfigurationSetName' => 'MyConfigurationSet',
        'Tags' => [
            [
                'Name' => 'foo',
                'Value' => 'bar',
            ],
        ],
    ],
],
```

Generando mailables

En Laravel, cada tipo de correo electrónico enviado por su aplicación se representa como una clase "Mailable". Estas clases se almacenan en el directorio `app/Mail`. No te preocupes si no ves este directorio en tu aplicación, ya que se generará para ti cuando crees tu 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`. Dentro de este método, puedes llamar a varios métodos como `from`, `subject`, `view` y `attach` para configurar la presentación y entrega del correo electrónico.

Configurando el remitente

Usando el método `from`

Primero, exploremos la configuración del remitente para el correo electrónico. O, en otras palabras, para quién será el correo electrónico (`from`). Hay dos formas de configurar el remitente. En primer lugar, puede usar el método `from` dentro de su método `build` de la clase mailable:

```
/** php
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
        ->view('emails.orders.shipped');
}
```

Usando una dirección global con `from`

Sin embargo, si tu aplicación utiliza la misma dirección "from" para todos sus correos electrónicos, puede resultar engorroso llamar al método `from` en cada clase mailable que genere. En su lugar, puede especificar una dirección global "from" en su 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'], php
```

Adicionalmente, puedes definir una dirección global "reply_to" dentro de tu archivo de configuración `config/mail.php`:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

php

Configurando la vista

Dentro de un método 'build' de la clase Mailable, puedes usar el método `view` para especificar qué plantilla se debe usar al representar los contenidos del correo electrónico. Dado que cada correo electrónico generalmente usa una Plantilla Blade para representar sus contenidos, tienes toda la potencia y la comodidad del motor de plantillas Blade al construir el HTML de su correo electrónico:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */  
public function build()  
{  
    return $this->view('emails.orders.shipped');  
}
```

php

TIP

Es posible que desees crear un directorio `resources/views/emails` para albergar todas tus plantillas de correos electrónicos; sin embargo, puedes colocarlos donde quieras siempre y cuando este dentro del directorio `resources/views`.

Correos con texto plano

Si deseas definir una versión de texto sin formato en tu correo electrónico, puedes usar el método `text`. Al igual que el método `view`, el método `text` acepta un nombre de plantilla que se usará para representar el contenido del correo electrónico. Eres libre de definir una versión HTML y de texto sin formato del mensaje:

```
/**  
 * Build the message.  
 *  
 * @return $this  
 */
```

php

```
public function build()
{
    return $this->view('emails.orders.shipped')
        ->text('emails.orders.shipped_plain');
}
```

Datos en Vistas

A través de propiedades públicas

Por lo general, querrás pasar algunos datos a tu vista que puedes utilizar al representar el HTML del correo electrónico. Hay dos maneras en que puedes hacer que los datos estén disponibles para la vista. Primero, cualquier propiedad pública definida en tu clase Mailable se pondrá automáticamente a disposición de la vista. Entonces, por ejemplo, puedes pasar datos al constructor de tu clase Mailable y establecer esos datos a propiedades públicas definidas en la clase:

```
<?php 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 tu vista, por lo que puedes acceder a ella como si tuvieras acceso a cualquier otro dato en tus plantillas Blade:

```
<div>
    Price: {{ $order->price }}
</div>
```

php

A través del método `with` :

Si deseas personalizar el formato de los datos de tu correo electrónico antes de enviarlos a la plantilla, puedes pasar manualmente los datos a la vista mediante el método `with`. Por lo general, aún podrás pasar datos a través del constructor de la clase Mailable; sin embargo, debes establecer estos datos en propiedades `protected` o `private` para que los datos no estén automáticamente disponibles para la plantilla. Luego, al llamar al método `with`, se pase un arreglo de datos que deseas poner a disposición de la plantilla:

```
<?php

namespace App\Mail;

use App\Order;
use Illuminate\Bus\Queueable;
```

php

```

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 han pasado con el método `with`, estarán automáticamente disponibles en la vista, por lo que puedes acceder a ellos como lo harías con cualquier otro dato en las plantillas Blade:

```
<div>
    Price: {{ $orderPrice }}
</div>
```

php

Archivos adjuntos

Para agregar archivos adjuntos a un correo electrónico, podemos usar el método `attach` dentro del método `build` de la clase Mailable. El método `attach` acepta la ruta completa al archivo como su primer argumento:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file');
}
```

php

Al adjuntar archivos a un mensaje, también puedes especificar el nombre para mostrar y / o el tipo MIME pasando un `array` como segundo argumento al 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',
        ]);
}
```

php

Adjuntando archivos desde el disco

Si has almacenado un archivo en uno de tus [discos](#), puedes adjuntarlo al correo electrónico usando el método `attachFromStorage` :

```
/** php
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('email.orders.shipped')
        ->attachFromStorage('/path/to/file');
}
```

De ser necesario, puedes especificar el nombre del archivo adjunto y opciones adicionales usando el segundo y tercer argumento del método `attachFromStorage` :

```
/** php
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('email.orders.shipped')
        ->attachFromStorage('/path/to/file', 'name.pdf', [
            'mime' => 'application/pdf'
        ]);
}
```

El método `attachFromStorageDisk` puede ser usado si necesitas especificar un disco de almacenamiento diferente a tu disco por defecto:

```
/** php
 * Build the message.
 *
 * @return $this
 */
```

```
public function build()
{
    return $this->view('email.orders.shipped')
        ->attachFromStorageDisk('s3', '/path/to/file');
}
```

Archivos adjuntos desde la memoria

El método `attachData` se puede usar para adjuntar una cadena de bytes sin formato como un archivo adjunto. Por ejemplo, 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 arreglo 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',
        ]);
}
```

php

Archivos adjuntos en línea

La incrustación de imágenes en línea en sus correos electrónicos suele ser engorrosa; sin embargo, Laravel proporciona una forma conveniente de adjuntar imágenes a sus correos electrónicos y recuperar el CID apropiado. Para incrustar una imagen en línea, usa el método `embed` en la variable `$message` dentro de tu plantilla de correo electrónico. Laravel automáticamente hace que la variable `$message` esté disponible para todas tus plantillas de correo electrónico, por lo que no tienes que preocuparte por pasarlal manualmente:

```
<body>
    Here is an image:
```

php

```
  
</body>
```

Nota

La variable `$message` no está disponible en los mensajes ya que los mensajes de texto plano (plain-text) no utilizan archivos adjuntos en línea.

Incrustar datos adjuntos de la memoria

Si ya tienes una cadena de datos en la memoria que deseas incorporar a una plantilla de correo electrónico, puedes usar el método `embedData` en la variable `$message`:

```
<body>  
    Here is an image from raw data:  
  
      
</body>
```

Personalizar el mensaje de SwiftMailer

El método `withSwiftMessage` de la clase base `Mailable` te permite registrar una función anónima que se invocará con la instancia del mensaje de SwiftMailer sin procesar antes de enviar el mensaje. Esto te 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');  
    });  
}
```

Mailables en markdown

Los mensajes escritos con Markdown le permiten aprovechar las plantillas y los componentes precompilados de las notificaciones por correo en tus documentos. Dado que los mensajes se escriben en Markdown, Laravel puede generar plantillas HTML atractivas para los mensajes y generar automáticamente una contraparte de texto sin formato.

Generar mailables en markdown

Para generar una clase de Mailable con una plantilla para Markdown puedes usar la opción `--markdown` del comando `make:mail` :

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

php

Luego de generar la clase, dentro de su método `build` debes llamar llame al método `markdown` en lugar del método `view`. Los métodos `markdown` aceptan el nombre de la plantilla Markdown y un arreglo 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');  
}
```

php

Escribir mensajes en Markdown

Los correos con Markdown utilizan una combinación de componentes Blade y sintaxis Markdown que le permiten construir fácilmente mensajes de correo al mismo tiempo que aprovechas los componentes prefabricados de Laravel:

php

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Thanks, <br>
{{ config('app.name') }}
@endcomponent
```

TIP

No uses una sangría excesiva al escribir correos electrónicos de Markdown. Los analizadores de Markdown renderizarán contenido sangrado como bloques de código.

Componente button

El componente de botón representa un enlace de botón centrado. El componente acepta dos argumentos, una `url` y un `color` opcional. Los colores compatibles son `primary`, `success` y `error`. Puedes agregar los botones que deseas a un mensaje:

php

```
@component('mail::button', ['url' => $url, 'color' => 'success'])
View Order
@endcomponent
```

Componente panel

El componente del panel representa el bloque de texto dado en un panel que tiene un color de fondo ligeramente diferente que el resto del mensaje. Esto te permite llamar la atención sobre un bloque de texto dado:

php

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Componente table

El componente de tabla le permite transformar una tabla en Markdown a una tabla HTML. El componente acepta la tabla en Markdown como su contenido. La alineación de columna de tabla es compatible con la sintaxis de alineación de tabla de Markdown predeterminada:

```
@component('mail::table')  
| Laravel | Table | Example |  
| ----- | :-----: | -----: |  
| Col 2 is | Centered | $10 |  
| Col 3 is | Right-Aligned | $20 |  
@endcomponent
```

php

Personalizar los componentes

Puedes exportar todos los componentes de correo Markdown a su propia aplicación para personalización. Para exportar los componentes, use el comando `vendor:publish` y la opción del tag `laravel-mail` de esta forma:

```
php artisan vendor:publish --tag=laravel-mail
```

php

Este comando publicará los componentes de correo Markdown en el directorio `resources/views/vendor/mail`. El directorio `mail` contendrá un directorio `html` y `text`, cada uno con sus respectivas representaciones de cada componente disponible. Eres libre de personalizar estos componentes como deseas.

Personalizar el CSS

Después de exportar los componentes, el directorio `resources/views/vendor/mail/html/themes` contendrá un archivo `default.css`, puedes personalizar el CSS en este archivo y sus estilos se alinearán automáticamente en las representaciones HTML de sus mensajes de correo Markdown.

Si te gustaría construir un nuevo tema para los componentes de Markdown de Laravel, puedes colocar un archivo CSS dentro del directorio `html/themes`. Luego de nombrar y guardar tu archivo de CSS, actualiza la opción `theme` del archivo de configuración `mail` con el nuevo nombre de tu tema.

Para personalizar un tema para un mailable individual, debes establecer la propiedad `$theme` de la clase mailable a el nombre del tema que debería ser usado al enviar el mailable.

Enviar correo

Para enviar un mensajes debes usar el método `to` en el `facade` llamado `Mail`. El método `to` acepta una dirección de correo, una instancia de usuario o una colección de usuarios. Si pasas un objeto o una colección de objetos, el remitente utilizará automáticamente sus propiedades de "email" y "name" cuando configure los destinatarios del correo electrónico, por lo tanto, asegúrese de que estos atributos estén disponibles en sus objetos. Una vez que haya especificado sus destinatarios, puede pasar una instancia de su clase mailable al método `send` :

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use App\Mail\OrderShipped;  
use App\Order;  
use Illuminate\Http\Request;  
use Illuminate\Support\Facades\Mail;  
  
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));  
    }  
}
```

No estás limitado a especificar los destinatarios "a" al enviar un mensaje. Eres libre de configurar los destinatarios "a", "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));
```

php

Renderizar mailables

Algunas veces puedes querer capturar el contenido HTML de un mailable sin enviarlo. Para lograr esto, puedes llamar al método `render` del mailable. Este método retornará los contenidos evaluados del mailable como una cadena:

```
$invoice = App\Invoice::find(1);

return (new App\Mail\InvoicePaid($invoice))->render();
```

php

Previsualizar mailables en el navegador

Al diseñar una plantilla mailable, es conveniente previsualizar rápidamente el mailable renderizado en tu navegador como una plantilla de Blade corriente. Por esta razón, Laravel te permite retornar cualquier mailable directamente desde un Closure de ruta o un controlador. Cuando un mailable es retornado, será renderizado y mostrado en el navegador, permitiéndote previsualizar 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);
});
```

php

Correo en cola

Poniendo en cola un correo electrónico

Con el envío de correos electrónicos puede alargar drásticamente el tiempo de respuesta de su aplicación, muchos desarrolladores eligen poner correos electrónicos en cola para el envío en segundo plano. Laravel lo hace fácil usando su función incorporada [API de cola unificada](#). Para poner en cola un

correo, use el método `queue` en el facade `Mail` después de especificar los destinatarios del mensaje:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

php

Este método se encargará automáticamente de insertar un trabajo en la cola para que el mensaje se envíe en segundo plano. Necesitarás [configurar tus colas](#) antes de usar esta característica.

Cola de mensajes retrasada

Si deseas retrasar la entrega de un mensaje de correo electrónico en cola, puedes usar el método `later`. Como primer argumento, el método `later` acepta una instancia `DateTime` que indica cuándo se debe enviar el mensaje:

```
$when = now()->addMinutes(10);

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later($when, new OrderShipped($order));
```

php

Enviar a queues específicas

Como todas las clases mailable generadas usando el comando `make:mail` usan el trait `Illuminate\Bus\Queueable` puedes llamar a los métodos `onQueue` y `onConnection` en cualquier instancia de clase mailable, lo que te permite especificar la conexión y nombre de cola para el mensaje:

```
$message = (new OrderShipped($order))
    ->onConnection('sq')
    ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

php

En cola por defecto

Si tienes clases mailables que deseas que siempre se pongan en cola, puedes implementar la interfaz `ShouldQueue` en la clase. Ahora, incluso si llamas al método `send` cuando envíes correos el mailable se pondrá en cola ya que implementa la interfaz:

```
use Illuminate\Contracts\Queue\ShouldQueue; php

class OrderShipped extends Mailable implements ShouldQueue
{
    //
}
```

Configuración regional de mailables

Laravel te permite enviar mailables en una configuración regional diferente al del idioma actual, e incluso recordará dicha configuración si el correo es agregado a una cola.

Para lograr esto, el facade `Mail` ofrece un método `locale` para establecer el idioma deseado. La aplicación cambiará a dicho configuración regional cuando el mailable sea formateado y luego volverá a la configuración anterior cuando el formato es completado:

```
Mail::to($request->user())->locale('es')->send(
    new OrderShipped($order)
);
```

Configuración regional de usuarios

Algunas veces, las aplicaciones almacenan la configuración regional preferida de cada usuario. Al implementar la interfaz `HasLocalePreference` en uno o más de tus modelos, puedes instruir a Laravel a usar dicha configuración almacenada al enviar correos electrónicos:

```
use Illuminate\Contracts\Translation\HasLocalePreference; php

class User extends Model implements HasLocalePreference
{
```

```
 /**
 * Get the user's preferred locale.
 *
 * @return string
 */
public function preferredLocale()
{
    return $this->locale;
}
```

Una vez has implementado la interfaz, Laravel automáticamente usará la configuración regional preferida al enviar mailables y notificaciones al modelo. Por lo tanto, no hay necesidad de llamar al método `locale` al usar esta interfaz:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

php

Correos y desarrollo local

Al desarrollar una aplicación que envía correos electrónicos, probablemente no deseas enviar correos electrónicos a direcciones reales. Laravel proporciona varias formas de "desactivar" el envío real de correos electrónicos durante el desarrollo local.

Driver Log

En lugar de enviar sus correos electrónicos, el driver de correos `log` escribirá todos los mensajes de correo electrónico en tus archivos de logs para su inspección. Para obtener más información sobre cómo configurar su aplicación por entorno, revisa la [configuración en la documentación](#).

Destinatario universal

Otra solución proporcionada por Laravel es establecer un destinatario universal de todos los correos electrónicos enviados por el framework. De esta forma, todos los correos electrónicos generados por tu aplicación serán enviados a una dirección específica, en lugar de la dirección realmente especificada al enviar el mensaje. Esto se puede hacer a través de la opción `to` en tu archivo de configuración `config/mail.php`:

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

php

Mailtrap

Finalmente, puedes usar un servicio como [Mailtrap](#) y el driver `smtp` para enviar sus mensajes de correo electrónico a un buzón ‘ficticio’ donde puedes verlos en un verdadero cliente de correo electrónico. Este enfoque tiene el beneficio de permitirle 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 se ha enviado un mensaje. Recuerda, estos eventos se disparan cuando el correo *se envía*, no cuando se pone en cola. Puedes registrar un detector de eventos para este evento en tu `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',
    ],
];
```

php

Notificaciones

- Introducción
- Crear notificaciones
- Enviar notificaciones
 - Utilizar el atributo notifiable
 - Utilizar la facade notification
 - Especificar canales de entrega
 - Notificaciones en cola
 - Notificaciones bajo demanda
- Notificaciones por correo
 - Formato para mensajes por correo
 - Personalizar el remitente
 - Personalizar el destinatario
 - Personalizar el asunto
 - Personalizar las plantillas
 - Previsualizar notificaciones de correo
- Notificaciones por correo en markdown
 - Generar el mensaje
 - Escribir el mensaje
 - Personalizar los componentes
- Notificaciones de la base de datos
 - Prerrequisitos
 - Formato de notificaciones de base de datos
 - Acceder a las notificaciones
 - Marcar notificaciones como leídas
- Notificaciones de difusión
 - Prerrequisitos
 - Formato de notificaciones de difusión
 - Escuchar notificaciones
- Notificaciones por SMS
 - Prerrequisitos
 - Formato de Notificaciones por SMS

- Formato de Notificaciones por Shortcode
- Personalizar el número remitente
- Enrutar notificaciones por SMS
- Notificaciones por Slack
 - Prerrequisitos
 - Formato de notificaciones por Slack
 - Archivos adjuntos en Slack
 - Enrutar notificaciones por Slack
- Configuración regional de notificaciones
- Eventos de notificación
- Canales personalizados

Introducción

Además de soporte para enviar correos electrónicos, Laravel brinda soporte para el envío de notificaciones mediante una variedad de canales de entrega, incluyendo correo, SMS (a través de [Nexmo](#)) y [Slack](#). Las notificaciones pueden ser también almacenadas en una base de datos para que puedan ser mostradas en la interfaz de tu página web.

Generalmente, las notificaciones deben ser mensajes cortos e informativos que notifiquen a los usuarios que algo ocurrió en tu aplicación. Por ejemplo, si estás escribiendo una aplicación de facturación, podrías enviar una notificación de "Recibo de Pago" a tus usuarios mediante correo electrónico y por SMS.

Crear notificaciones

En Laravel, cada notificación está representada por una sola clase (generalmente almacenada en el directorio `app/Notifications`). No te preocupes si no ves este directorio en tu aplicación, será creada por ti cuando ejecutes el comando Artisan `make:notification` :

```
php artisan make:notification InvoicePaid
```

php

Este comando colocará una clase de notificación nueva en tu directorio `app/Notifications` . Cada clase de notificación contiene un método `via` y un número variable de métodos de construcción de mensaje (tales como `toMail` o `toDatabase`) que convierten la notificación en un mensaje optimizado para ese canal en particular.

Enviar notificaciones

Usar el atributo notifiable

Las notificaciones pueden ser enviadas en dos formas: usando el método `notify` del atributo `Notifiable` o usando `Notification` facade. Primero, exploremos el uso del atributo:

```
<?php  
  
namespace App;  
  
use Illuminate\Notifications\Notifiable;  
use Illuminate\Foundation\Auth\User as Authenticatable;  
  
class User extends Authenticatable  
{  
    use Notifiable;  
}
```

php

Este atributo es utilizado por el modelo `App\User` por defecto y contiene un método que puede ser usado para enviar notificaciones: `notify`. El método `notify` espera recibir una instancia de notificación:

```
use App\Notifications\InvoicePaid;  
  
$user->notify(new InvoicePaid($invoice));
```

php

TIP

Recuerda que puedes usar el atributo `Illuminate\Notifications\Notifiable` en cualquiera de tus modelos. No estás limitado a incluirlo solamente en tu modelo `User`.

Usar la facade notification

Alternativamente, puedes enviar notificaciones mediante la facade `Notification`. Esto es útil principalmente cuando necesitas enviar una notificación a múltiples entidades notificables, como un

grupo de usuarios. Para enviar notificaciones usando la facade, pasa todas las entidades notificables y la instancia de notificación al método `send` :

```
Notification::send($users, new InvoicePaid($invoice));
```

php

Especificar canales de entrega

Cada clase de notificación tiene un método `via` que determina mediante cuáles canales será entregada la notificación. Las notificaciones pueden ser enviadas por los canales `mail`, `database`, `broadcast`, `nexmo`, y `slack`.

TIP

Si estás interesado en utilizar otros canales de entrega como Telegram o Pusher, revisa el sitio dirigido por la comunidad [Laravel Notification Channels](#).

El método `via` recibe una instancia `$notifiable` la cual será una instancia de la clase a la cual la notificación está siendo enviada. Puedes usar `$notifiable` para determinar mediante cuáles canales debería ser entregada la notificación:

```
/**
 * Get the notification's delivery channels.
 *
 * @param mixed $notifiable
 * @return array
 */
public function via($notifiable)
{
    return $notifiable->prefers_sms ? ['nexmo'] : ['mail', 'database'];
}
```

php

Notificaciones en cola

Nota

Antes de poner notificaciones en cola, se debe configurar una cola y activar un worker.

Enviar notificaciones puede tomar tiempo, especialmente si el canal necesita una API externa para llamar o entregar la notificación. Para acelerar el tiempo de respuesta de tu notificación, permite que sea puesta en cola añadiendo la interfaz `ShouldQueue` y el atributo `Queueable` a tu clase. La interfaz y el atributo son importados para todas las notificaciones generadas usando `make:notification`, así que puedes añadir de inmediato a tu 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 a tu notificación, puedes enviarla con normalidad. Laravel detectará `ShouldQueue` en la clase y automáticamente pondrá en cola la entrega de la notificación:

```
$user->notify(new InvoicePaid($invoice));
```

Si quisieras retrasar la entrega de la notificación, puedes encadenar el método `delay` al instanciar tu notificación:

```
$when = now()->addMinutes(10);  
  
$user->notify((new InvoicePaid($invoice))->delay($when));
```

Notificaciones bajo demanda

A veces puede que necesites enviar una notificación a alguien que no está almacenado como "usuario" de tu aplicación. Usando el método `Notification::route`, puedes especificar información de enrutamiento para una notificación ad-hoc antes de enviarla:

```
Notification::route('mail', 'taylor@example.com')
    ->route('nexmo', '55555555555')
    ->route('slack', 'https://hooks.slack.com/services/...')

    ->notify(new InvoicePaid($invoice));
```

php

Notificaciones por correo

Formato de mensajes por correo

Si una notificación tiene soporte para ser enviada por correo, se debe definir un método `toMail` en la clase de la notificación. Este método recibirá una entidad `$notifiable` y debe devolver una instancia `\Illuminate\Notifications\Messages\MailMessage`. Los mensajes por correo pueden contener líneas de texto, así como una "llamada a la acción". Observemos un método `toMail` de ejemplo:

```
/**
 * 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);

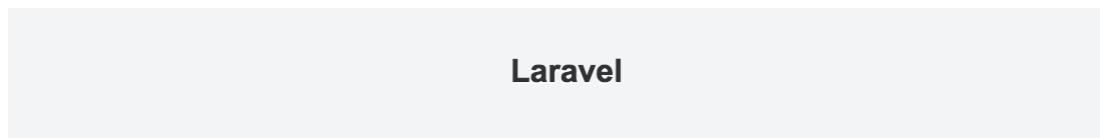
    return (new MailMessage)
        ->greeting('Hello!')
        ->line('One of your invoices has been paid!')
        ->action('View Invoice', $url)
        ->line('Thank you for using our application!');
}
```

php

TIP

Nota que se está usando el método `$this->invoice->id` en el método `toMail`. Puedes pasar cualquier dato que la notificación necesite para generar su mensaje dentro del constructor de la notificación.

En este ejemplo, registramos un saludo, una línea de texto, un llamado a la acción y luego otra línea de texto. Estos elementos proporcionados por el objeto `MailMessage` hacen que sea rápido y sencillo dar formato a pequeños correos transaccionales. El canal de correo entonces traducirá los componentes del mensaje en una plantilla HTML agradable y con capacidad de respuesta, justo con su contraparte de texto simple. He aquí un ejemplo de un correo 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>

A screenshot of an email message. The footer at the bottom contains the text "© 2016 Laravel. All rights reserved.".

TIP

Al enviar notificaciones por correo, asegúrate de establecer el valor `name` en tu archivo `config/app.php`. Este valor será usado en el encabezado y pie de los mensajes de notificación por correo.

Otras opciones de formato para notificaciones

En lugar de definir las "líneas" de texto en la clase `notification`, puedes usar el método `view` para especificar una plantilla personalizada que debe ser usada para renderizar el correo de notificación:

```
/** php
 * 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, puedes devolver un `objeto mailable` desde el método `toMail`:

```
use App\Mail\InvoicePaid as Mailable; php

/**
 * 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 acerca de errores, como un pago fallido. puedes indicar que un mensaje por correo se refiere a un error llamando al método `error` cuando se construye el mensaje. Al usar el método `error` en un mensaje por correo, el botón de llamado a la acción será rojo en vez de azul:

```
/** php
 * 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 remitente

Por defecto, el remitente del correo electrónico es definido en el archivo `config/mail.php`. Sin embargo, también puedes definir un remitente a través de una notificación específica:

```
/** php
 * Get the mail representation of the notification.
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->from('noreply@laravel.com', 'Laravel')
        ->line('...');

}
```

Personalizar el destinatario

Al enviar notificaciones mediante el canal `mail`, el sistema de notificaciones automáticamente buscará una propiedad `email` en tu entidad notificable. Puedes personalizar la dirección de correo electrónico usada para entregar la notificación definiendo el 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.  
     *  
     * @param \Illuminate\Notifications\Notification $notification  
     * @return array|string  
     */  
    public function routeNotificationForMail($notification)  
    {  
        // Return email address only...  
        return $this->email_address;  
  
        // Return name and email address...  
        return [$this->email_address => $this->name];  
    }  
}
```

php

Personalizar el asunto

Por defecto, el asunto del correo electrónico es el nombre de la clase de notificación formateada a "title case". Así que si tu clase de notificación se llama `InvoicePaid`, el asunto del correo será `Invoice Paid`. Si se prefiere especificar un asunto explícito para el mensaje, puedes llamar al método `subject` al construir el mensaje:

```
/**  
 * Get the mail representation of the notification.
```

php

```
* @param mixed $notifiable
* @return \Illuminate\Notifications\Messages\MailMessage
*/
public function toMail($notifiable)
{
    return (new MailMessage)
        ->subject('Notification Subject')
        ->line('...');
}
```

Personalizar las plantillas

Puedes modificar las plantillas HTML y de texto simple usadas por las notificaciones de correo publicando los recursos del paquete de notificación. Luego de ejecutar este comando, las plantillas de notificación de correo estarán ubicadas en el directorio `resources/views/vendor/notifications` :

```
php artisan vendor:publish --tag=laravel-notifications
```

php

Previsualizar notificaciones de correo

Al diseñar una plantilla de notificación de correo, es conveniente previsualizar rápidamente el mensaje de correo renderizado en tu navegador como una plantilla normal de Blade. Por esta razón, Laravel te permite retornar cualquier mensaje de correo generado por una notificación de correo directamente desde un Closure de ruta o un controlador. Cuando un `MailMessage` es retornado, este será renderizado y mostrado en el navegador, permitiéndote previsualizar rápidamente su diseño sin necesidad de enviarlo a un correo electrónico real:

```
Route::get('mail', function () {
    $invoice = App\Invoice::find(1);
    return (new App\Notifications\InvoicePaid($invoice))
        ->toMail($invoice->user);
});
```

php

Notificaciones por correo markdown

Las notificaciones por correo Markdown permiten tomar ventaja de las plantillas prefabricadas para notificaciones por correo, dando a su vez libertad para escribir mensajes más largos y personalizados. Como los mensajes están escritos en Markdown, Laravel puede renderizar plantillas HTML bellas y responsivas para los mensajes y a la vez generar automáticamente su contraparte en texto simple.

Generar el mensaje

Para generar una notificación con su plantilla Markdown correspondiente, puedes usar la opción `--markdown` del comando Artisan `make:notification` :

```
php artisan make:notification InvoicePaid --markdown=mail.invoice.paid
```

php

Como todas las otras notificaciones, aquellas que usan plantillas Markdown deben definir un método `toMail` en su clase de notificación. Sin embargo, en lugar de usar los modelos `line` y `action` para construir la notificación, se usa el método `markdown` para especificar el nombre de la plantilla Markdown a ser usada:

```
/*
 * 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]);
}
```

php

Escribir el mensaje

Las notificaciones por correo Markdown usan una combinación de componentes Blade y sintaxis Markdown que te permiten construir fácilmente notificaciones a la vez que se apalancan los componentes de notificación prefabricados por Laravel:

php

```
@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 renderiza un enlace a un botón centrado. El componente acepta dos argumentos, una `url` y un `color` opcional. Los colores disponibles son `blue`, `green`, y `red`. Puedes añadir tantos componentes de botón a una notificación como deseas:

php

```
@component('mail::button', ['url' => $url, 'color' => 'green'])
View Invoice
@endcomponent
```

Componente panel

El componente panel renderiza el bloque de texto dado en un panel que tiene un color de fondo ligeramente distinto al resto de la notificación. Esto permite poner énfasis en un determinado bloque de texto:

php

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Componente table

El componente table permite transformar una tabla Markdown en una tabla HTML. El componente acepta la tabla Markdown como contenido. La alineación de columnas es soportada usando la sintaxis de alineación de tablas de Markdown por defecto:

```
@component('mail::table')
| Laravel      | Table          | Example   |
| ----- | :-----: | -----: |
| Col 2 is    | Centered       | $10        |
| Col 3 is    | Right-Aligned | $20        |
@endcomponent
```

php

Personalizar los componentes

Puedes exportar todos los componentes de notificación de Markdown a tu propia aplicación para personalización. Para exportar los componentes, se usa el comando Artisan `vendor:publish` para publicar la etiqueta del asset `laravel-mail`:

```
php artisan vendor:publish --tag=laravel-mail
```

php

Este comando publicará los componentes de correo de Markdown al directorio `resources/views/vendor/mail`. El directorio `mail` contendrá los directorios `html` y `text`, cada uno contiene sus respectivas representaciones de cada componente disponible. Eres libre de personalizar estos componentes de acuerdo a su preferencia.

Personalizar CSS

Después de exportar los componentes, el directorio `resources/views/vendor/mail/html/themes` contendrá un archivo `default.css`. Puedes personalizar el CSS en este archivo y los estilos automáticamente se alinearán con las representaciones HTML de las notificaciones Markdown.

Si te gustaría construir un nuevo tema para los componentes Markdown de Laravel, puedes colocar un archivo CSS dentro del directorio `html/themes`. Luego de nombrar y guardar tus archivos de CSS, actualiza la opción `theme` del archivo de configuración `mail` para que coincida con el nombre de tu nuevo tema.

Para personalizar un tema para una notificación individual, puedes llamar al método `theme` al momento de construir el mensaje de la notificación. El método `theme` acepta el nombre del tema que debería ser usado al momento de enviar la notificación:

```
/** *
 * Get the mail representation of the notification.
```

php

```
* @param mixed $notifiable
* @return \Illuminate\Notifications\Messages\MailMessage
*/
public function toMail($notifiable)
{
    return (new MailMessage)
        ->theme('invoices')
        ->theme('invoice')
        ->subject('Invoice Paid')
        ->markdown('mail.invoice.paid', ['url' => $url]);
}
```

Notificaciones de base de datos

Prerrequisitos

El canal de notificaciones `database` guarda la información de notificación en una tabla de base de datos. Esta tabla contendrá información como el tipo de notificación así como datos JSON personalizados que describen la notificación.

Puedes buscar en la tabla para mostrar las notificaciones en la interfaz de usuario de la aplicación. Pero, antes de poder hacer esto, necesitarás crear una tabla de base de datos para almacenar las notificaciones. Puedes usar el comando `notifications:table` para generar una migración con el esquema de tabla apropiado:

```
php artisan notifications:table
php artisan migrate
```

php

Agregar formato a las notificaciones de la base de datos

Si una notificación posee soporte para ser almacenada en una tabla de base de datos, debes definir un método `toDatabase` o `toArray` en la clase de notificación. Este método recibirá una entidad `$notifiable` y debería devolver un arreglo PHP sencillo. El arreglo devuelto estará codificado como JSON y almacenado en la columna `data` de la tabla `notifications`. Observemos 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,
    ];
}
```

php

toDatabase Vs. toArray

El método `toArray` también es usado por el canal `broadcast` para determinar cuáles datos difundir al cliente JavaScript. Si prefieres tener dos representaciones de arreglos para los canales `database` y `broadcast`, debes definir un método `toDatabase` en lugar de `toArray`.

Acceder a las notificaciones

Una vez que las notificaciones se almacenan en la base de datos, necesitas una forma conveniente de acceder a ellas desde tus entidades notificables. El atributo

`Illuminate\Notifications\Notifiable`, el cual está incluido en el modelo de Laravel `App\User` por defecto, incluye una relación Eloquent `notifications` que devuelve las notificaciones para la entidad. Para conseguir las notificaciones, puedes acceder a este método como a cualquier otra relación Eloquent. Por defecto, las notificaciones serán clasificadas por el timestamp `created_at`:

```
$user = App\User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->type;
}
```

php

Si quieres recibir sólo las notificaciones "no leídas (unread)", puedes usar la relación `unreadNotifications`. Nuevamente, las notificaciones serán clasificadas por el timestamp

```
created_at :
```

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    echo $notification->type;
}
```

php

TIP

Para acceder a las notificaciones desde el cliente JavaScript, se debe definir un controlador de notificaciones para tu aplicación que devuelva las notificaciones para una entidad notificable, como el usuario actual. puedes entonces elaborar una petición HTTP al URI de ese controlador desde el cliente JavaScript.

Marcar notificaciones como leídas

Normalmente, querrás marcar una notificación como "leída (read)" cuando un usuario la ve. El atributo `Illuminate\Notifications\Notifiable` provee un método `markAsRead` el cual actualiza la columna `read_at` en el registro de base de datos de las notificaciones:

```
$user = App\User::find(1);

foreach ($user->unreadNotifications as $notification) {
    $notification->markAsRead();
}
```

php

Sin embargo, en lugar de hacer bucle a través de cada notificación, puedes usar el método

`markAsRead` directamente en un grupo de notificaciones:

```
$user->unreadNotifications->markAsRead();
```

php

Asimismo, puedes utilizar una consulta de actualización masiva para marcar todas las notificaciones como leídas sin necesidad de recuperarlas de la base de datos:

```
$user = App\User::find(1);
```

php

```
$user->unreadNotifications()->update(['read_at' => now()]);
```

Puedes hacer `delete` a las notificaciones para removerlas por completo de la tabla:

```
$user->notifications()->delete();
```

php

Notificaciones de difusión

Prerrequisitos

Antes de difundir notificaciones, debes configurar y familiarizarse con los servicios [broadcasting de eventos](#) de Laravel. La difusión de eventos brinda una forma de reaccionar a los eventos de Laravel disparados por el servidor, desde el cliente JavaScript.

Formato de notificaciones de difusión

El canal `broadcast` difunde notificaciones usando los servicios [broadcasting de eventos](#) de Laravel, permitiéndole al cliente JavaScript capturar notificaciones en tiempo real. Si una notificación posee soporte para difusión, debes definir un método `toBroadcast` en la clase de notificación. Este método recibirá una entidad `$notifiable` y debe devolver una instancia `BroadcastMessage`. Si el método `toBroadcast` no existe, el método `toArray` será usado para recopilar los datos que deberían ser transmitidos. Los datos devueltos estarán codificados como JSON y se difundirán al cliente JavaScript.

Observemos un ejemplo del método `toBroadcast`:

```
use Illuminate\Notifications\Messages\BroadcastMessage;
```

php

```
/**
 * Get the broadcastable representation of the notification.
 *
 * @param mixed $notifiable
 * @return BroadcastMessage
 */
public function toBroadcast($notifiable)
{
    return new BroadcastMessage([
        'id' => $notifiable->id,
        'type' => $notifiable->type,
        'body' => $notifiable->body,
        'url' => $notifiable->url,
        'image' => $notifiable->image,
        'name' => $notifiable->name,
        'created_at' => $notifiable->created_at,
        'updated_at' => $notifiable->updated_at,
    ]);
}
```

```
'invoice_id' => $this->invoice->id,  
'amount' => $this->invoice->amount,  
]);  
}
```

Configuración de la cola de difusión

Todas las notificaciones de difusión son puestas en cola para ser difundidas. Si prefieres configurar la conexión a la cola o el nombre de la cola usada para las operaciones de difusión, puedes usar los métodos `onConnection` y `onQueue` de `BroadcastMessage` :

```
return (new BroadcastMessage($data))  
    ->onConnection('sqS')  
    ->onQueue('broadcasts');
```

php

TIP

Adicional a los datos especificados, las notificaciones de difusión contendrán también un campo `type` que contiene el nombre de clase de la notificación.

Escuchar notificaciones

Las notificaciones se difundirán en un canal privado formateado utilizando la convención `{notifiable}.{id}`. Por lo tanto, si estás enviando una notificación a una instancia `App\User` con una ID de `1`, la notificación será difundida en el canal privado `App.User.1`. Al usar `Laravel Echo`, puedes fácilmente escuchar notificaciones en un canal utilizando el método helper `notification` :

```
Echo.private('App.User.' + userId)  
    .notification((notification) => {  
        console.log(notification.type);  
   });
```

php

Personalizar el canal de notificación

Si quieras personalizar los canales mediante los cuales una entidad notificable recibe sus notificaciones de difusión, puedes definir un método `receivesBroadcastNotificationsOn` en la entidad notificable:

```
<?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;  
    }  
}
```

php

Notificaciones por SMS

Prerrequisitos

El envío de notificaciones por SMS en Laravel trabaja con [Nexmo](#). Antes de poder enviar notificaciones mediante Nexmo, necesitas instalar el paquete Composer `laravel/nexmo-notification-channel`:

```
composer require laravel/nexmo-notification-channel
```

php

Esto también instalará el paquete [nexmo/laravel](#). Este paquete viene con [su propio archivo de configuración](#). Puedes usar las variables de entorno `NEXMO_KEY` y `NEXMO_SECRET` para establecer tus clave pública y privada de Nexmo.

Luego, necesitas agregar algunas opciones de configuración al archivo `config/services.php`. Puedes copiar el ejemplo de configuración siguiente para empezar:

```
'nexmo' => [
    'sms_from' => '15556666666',
],
```

php

La opción `sms_from` es el número de teléfono remitente de los mensajes SMS. Se debe generar un número de teléfono para la aplicación en el panel de control de Nexmo.

Formato de notificaciones de SMS

Si una notificación tiene soporte para ser enviada mediante SMS, debes definir un método `toNexmo` en la clase de notificación. Este método recibirá una entidad `$notifiable` y debe devolver una instancia `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');
}
```

php

Formato de Notificaciones de Shortcode

Laravel también admite el envío de notificaciones de código corto que son plantillas predefinidas en tu cuenta de Nexmo. Puedes indicar el tipo de notificación (`alert`, `2fa` o `marketing`), así como los valores personalizados que llenarán la plantilla:

```
/**
 * Get the Nexmo / Shortcode representation of the notification.
 *
 * @param mixed $notifiable
 * @return array
 */
public function toShortcode($notifiable)
{
```

php

```
        return [
            'type' => 'alert',
            'custom' => [
                'code' => 'ABC123',
            ],
        ];
    }
}
```

TIP

Tal como en [Enrutar notificaciones por SMS](#), deberías implementar el método `routeNotificationForshortcode` en tu modelo notifiable.

Contenido unicode

Si el mensaje SMS contiene caracteres Unicode, debes 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();
}
```

php

Personalizando el número remitente

Si deseas enviar algunas notificaciones desde un número telefónico diferente al especificado en el archivo `config/services.php`, puedes usar el método `from` en una instancia `NexmoMessage` :

```
/*
 * Get the Nexmo / SMS representation of the notification.
 *

```

php

```
* @param mixed $notifiable
* @return NexmoMessage
*/
public function toNexmo($notifiable)
{
    return (new NexmoMessage)
        ->content('Your SMS message content')
        ->from('15554443333');
}
```

Enrutar notificaciones por SMS

Para enrutar notificaciones de Nexmo al número de teléfono correcto, define un método

`routeNotificationForNexmo`

php

```
<?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.
     *
     * @param \Illuminate\Notifications\Notification $notification
     * @return string
     */
    public function routeNotificationForNexmo($notification)
    {
        return $this->phone_number;
    }
}
```

Notificaciones por Slack

Prerrequisitos

Antes de poder enviar notificaciones mediante Slack, debes instalar el paquete para el canal de notificación mediante Composer:

```
composer require laravel/slack-notification-channel
```

php

También necesitarás configurar una integración “[Incoming Webhook](#)” para tu equipo en Slack. Esta integración proveerá una URL utilizable para [enrutamiento de notificaciones de Slack](#).

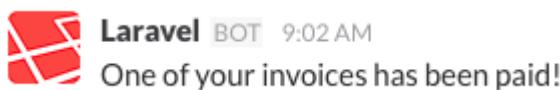
Formato de notificaciones por Slack

Si una notificación tiene soporte para ser enviada como mensaje por Slack, debes definir un método `toSlack` en la clase de notificación. El método recibirá una entidad `$notifiable` y debe devolver una instancia `Illuminate\Notifications\Messages\SlackMessage`. Los mensajes de Slack pueden contener texto así como un “archivo adjunto” que formatea texto adicional o un arreglo de campos. Observemos 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!');  
}
```

php

En este ejemplo estamos solamente enviando una línea de texto a Slack, la cual creará un mensaje que luce como éste:



Personalizar el remitente y destinatario

Puedes usar los métodos `from` y `to` para personalizar el remitente y el destinatario. El método `from` acepta un nombre de usuario y un identificador emoji, mientras que el método `to` acepta un canal y un 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');
}
```

php

También puedes utilizar una imagen como logo en vez 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/img/favicon/favicon.ico')
        ->content('This will display the Laravel logo next to the message');
}
```

php

Archivos adjuntos en Slack

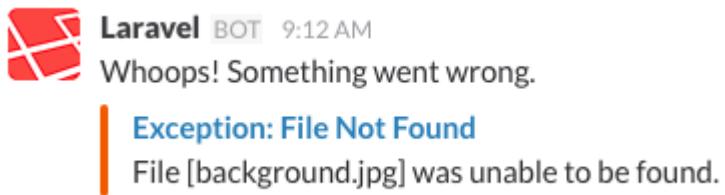
También puedes añadir "adjuntos" a los mensajes en Slack. Éstos brindan opciones de formato más amplias que mensajes de texto simple. En este ejemplo, enviaremos una notificación de error acerca de

una excepción que ocurrió en una aplicación, incluyendo un enlace para ver más detalles sobre la excepción:

```
/** php
 * 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 en Slack como el siguiente:



Los adjuntos te permitirán especificar un arreglo de datos que deben ser presentados al usuario. Los datos dados serán presentados en forma de tabla para su fácil lectura:

```
/** php
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */
public function toSlack($notifiable)
```

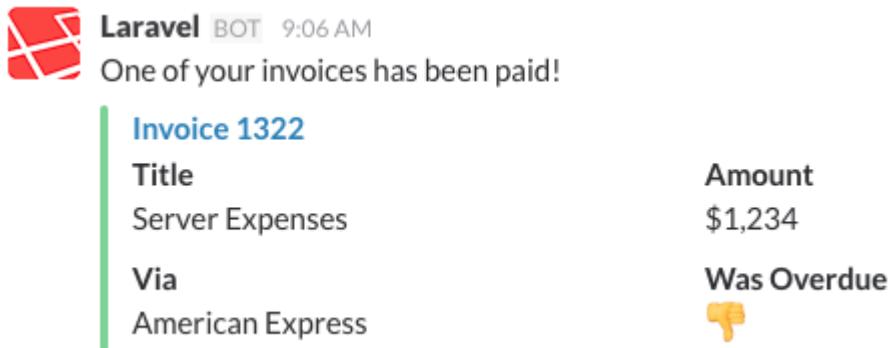
```

{
    $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 generará un mensaje en Slack como el siguiente:



Contenido adjunto en markdown

Si algunos de tus campos adjuntos contienen Markdown, puedes usar el método `markdown` para instruir a Slack procesar y mostrar los campos proporcionados como texto formateado en Markdown. Los valores aceptados por este método son: `pretext`, `text`, y/o `fields`. Para más información sobre formato de adjuntos de Slack, revisa la [documentación del API de Slack](#):

```

/**
 * Get the Slack representation of the notification.
 *
 * @param mixed $notifiable
 * @return SlackMessage
 */

```

php

```
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 notificaciones de Slack a la ubicación apropiada, debes definir un método

`routeNotificationForSlack` en tu entidad notificable. Esto debería devolver un webhook URL al cual debe ser entregada la notificación. Las Webhook URLs pueden ser generadas añadiendo un servicio "Incoming Webhook" a tu equipo 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.
     *
     * @param \Illuminate\Notifications\Notification $notification
     * @return string
     */
    public function routeNotificationForSlack($notification)
    {
        return 'https://hooks.slack.com/services/...';
    }
}
```

```
    }  
}
```

Configuración regional de notificaciones

Laravel te permite enviar notificaciones en una configuración regional distinta al idioma actual e incluso recordará esta configuración si la notificación está encolada.

Para lograr esto, la clase `Illuminate\Notifications\Notification` ofrece un método `locale` para establecer el idioma deseado. La aplicación cambiará a esta configuración cuando la notificación esté siendo formateada y luego se revertirá a la configuración regional previa cuando el formato esté completo:

```
$user->notify((new InvoicePaid($invoice))->locale('es'));
```

php

La configuración regional de múltiples entradas notificables también puede ser logradas mediante la facade `Notification`:

```
Notification::locale('es')->send($users, new InvoicePaid($invoice));
```

php

Configuración regional preferida por el usuario

A veces, las aplicaciones almacenan la configuración regional preferida de cada usuario. Al implementar la interfaz `HasLocalePreference` en tu modelo notificable, puedes instruir a Laravel que use esta configuración almacenada al enviar una notificación:

```
use Illuminate\Contracts\Translation\HasLocalePreference;
```

php

```
class User extends Model implements HasLocalePreference  
{  
    /**  
     * Get the user's preferred locale.  
     *  
     * @return string  
     */  
    public function preferredLocale()  
    {
```

```
        return $this->locale;
    }
}
```

Una vez esté implementada la interfaz, Laravel usará automáticamente la configuración regional preferida al enviar notificaciones y mailables al modelo. Por lo tanto, no es necesario llamar al método `locale` cuando usas esta interfaz:

```
$user->notify(new InvoicePaid($invoice));
```

php

Eventos de notificación

Cuando una notificación es enviada, el evento

`Illuminate\Notifications\Events\NotificationSent` es desencadenado por el sistema de notificación. Esto contiene la entidad "notificable" y la instancia de notificación en sí. Puedes registrar listeners para este evento en tu `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Notifications\Events\NotificationSent' => [
        'App\Listeners\LogNotification',
    ],
];
```

php

TIP

Luego de registrar listeners en tu `EventServiceProvider`, usa el comando Artisan `event:generate` para generar rápidamente clases de listeners.

Dentro de un listener de eventos, puedes acceder a las propiedades `notifiable`, `notification` y `channel` del evento para aprender más acerca de 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  
    // $event->response  
}
```

php

Canales personalizados

Laravel viene una gran cantidad de canales de notificación, pero puedes ser deseable escribir controladores propios para entregar notificaciones mediante otros canales. Laravel hace de esto algo sencillo. Para empezar, debes definir una clase que contenga un método `send`. El método debe recibir dos argumentos: un `$notifiable` y un `$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);  
    }  
}
```

php

```
        // Send notification to the $notifiable instance...
    }
}
```

Una vez que la clase de notificación ha sido definida, puedes devolver el nombre de la clase desde el método `via` de cualquier notificación:

```
<?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](#)
 - [Una nota sobre facades](#)
- [Descubrimiento de paquetes](#)
- [Proveedores de servicios](#)
- [Recursos](#)
 - [Configuración](#)
 - [Migraciones](#)
 - [Factories](#)
 - [Rutas](#)
 - [Traducciones](#)
 - [Vistas](#)
- [Comandos](#)
- [Archivos públicos](#)
- [Publicar grupos de archivos](#)

Introducción

Los paquetes son la forma principal de agregar funcionalidad a Laravel. Los paquetes pueden ser cualquier cosa, desde una estupenda manera de trabajar con fechas como [Carbon](#), o un framework completo de pruebas BDD como [Behat](#).

Hay diferentes tipos de paquetes. Algunos paquetes son independientes, lo que significa que funcionan con cualquier framework de PHP. Carbon y Behat son ejemplos de paquetes independientes. Cualquiera

de estos paquetes se puede usar con Laravel simplemente solicitándolos en el archivo

`composer.json`.

Por otro lado, otros paquetes están específicamente destinados para su uso con Laravel. Estos paquetes pueden tener rutas, controladores, vistas y configuraciones específicamente diseñadas para mejorar una aplicación Laravel. Esta guía cubre principalmente el desarrollo de aquellos paquetes que son específicos de Laravel.

Una nota sobre facades

Al escribir una aplicación Laravel, generalmente no importa si usas interfaces o facades ya que ambos brindan niveles esencialmente iguales de capacidad de pruebas. Sin embargo, al escribir paquetes, tu paquete normalmente no tendrá acceso a todos las funciones helpers de prueba de Laravel. Si deseas escribir pruebas para el paquete como si existiera dentro de una típica aplicación Laravel puedes usar el paquete [Orchestral Testbench](#).

Descubrimiento de paquetes

En el archivo de configuración `config/app.php` de una aplicación Laravel, la opción `providers` define una lista de proveedores de servicios que Laravel debe cargar. Cuando alguien instala tu paquete normalmente querrás que tu proveedor de servicio sea incluido en esta lista. En lugar de requerir que los usuarios agreguen manualmente su proveedor de servicios a la lista, puede definir el proveedor en la sección `extra` del archivo `composer.json` de tu paquete. Además de los proveedores de servicios, también puedes enumerar los facades que deseas registrar:

```
php
{
    "extra": {
        "laravel": {
            "providers": [
                "Barryvdh\\Debugbar\\ServiceProvider"
            ],
            "aliases": {
                "Debugbar": "Barryvdh\\Debugbar\\Facade"
            }
        }
    },
}
```

Una vez que tu paquete se haya configurado para su descubrimiento, Laravel registrará automáticamente sus proveedores de servicios y facades cuando esté instalado, creando una experiencia

de instalación conveniente para los usuarios de tu paquete.

Exclusión del descubrimiento de paquetes

Si eres el consumidor de un paquete y deseas deshabilitar el descubrimiento de paquetes para un paquete, puedes incluir el nombre del paquete en la sección `extra` del archivo `composer.json` de tu aplicación Laravel:

```
php
"extra": {
    "laravel": {
        "dont-discover": [
            "barryvdh/laravel-debugbar"
        ]
    }
},
```

Puede deshabilitar el descubrimiento de paquetes para todos los paquetes que usan el carácter `*` dentro de la directiva `dont-discover` de tu aplicación:

```
php
"extra": {
    "laravel": {
        "dont-discover": [
            "*"
        ]
    }
},
```

Proveedores de servicios

Los Proveedores de Servicios son la conexión entre tu paquete y Laravel. Un proveedor de servicios es responsable de enlazar cosas a Laravel con el Contenedor de Servicios e informar a Laravel dónde cargar los recursos del paquete como vistas y archivos de configuración y de configuración regional.

Un Proveedor de Servicios extiende de la clase `Illuminate\Support\ServiceProvider` y contiene dos métodos: `register` y `boot`. La clase base `ServiceProvider` está ubicada en `illuminate/support`, donde debemos especificar todas las dependencias de nuestro paquete. Para

obtener más información sobre la estructura y el propósito de los proveedores de servicios, visita su documentación.

Recursos

Configuración

Por lo general, deberás publicar el archivo de configuración de tu paquete en el propio directorio `config` de la aplicación. Esto permitirá a los usuarios anular fácilmente sus opciones de configuración predeterminadas. Para permitir que se publiquen sus archivos de configuración, debes llamar al método `publishes` desde el método `boot` de tu proveedor de servicios:

```
/** php
 * 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 tu paquete ejecutan el comando `vendor:publish` de Laravel, su archivo se copiará a la ubicación de publicación especificada. Una vez que se haya publicado su configuración, se podrá acceder a sus valores como cualquier otro archivo de configuración:

```
$value = config('courier.option');
```

Nota

No debes definir funciones anónimas en tus archivos de configuración ya que no se pueden serializar correctamente cuando los usuarios ejecutan el comando Artisan `config:cache`.

Configuración predeterminada del paquete

También puedes fusionar tu propio archivo de configuración de paquete con la copia publicada de la aplicación. Esto permitirá que los usuarios definan solo las opciones que realmente desean anular en la copia publicada de la configuración. Para fusionar las configuraciones, use el método

`mergeConfigFrom` dentro del método `register` de tu proveedor de servicios:

```
/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->mergeConfigFrom(
        __DIR__ . '/path/to/config/courier.php', 'courier'
    );
}
```

php

Nota

Este método solo combina el primer nivel de la matriz de configuración. Si los usuarios definen parcialmente una matriz de configuración multidimensional las opciones faltantes no se fusionarán.

Rutas

Si tu paquete contiene rutas, puede cargarlas usando el método `loadRoutesFrom`. Este método determinará automáticamente si las rutas de la aplicación se almacenan en caché y no cargarán el archivo de rutas si las rutas ya se han almacenado en caché:

```
/**
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadRoutesFrom(__DIR__ . '/routes.php');
}
```

php

Migraciones

Si tu paquete contiene [migraciones de base de datos](#), puedes usar el método `loadMigrationsFrom` para informarle a Laravel cómo cargarlas. El método `loadMigrationsFrom` acepta la ruta a las migraciones de tu paquete como su único argumento:

```
/** php
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadMigrationsFrom(__DIR__.'/path/to/migrations');
}
```

Una vez que se hayan registrado las migraciones de tu paquete, éstas se ejecutarán automáticamente cuando se utilice el comando `php artisan migrate`. Cabe destacar que no es necesario exportarlas al directorio principal de las migraciones en la aplicación.

Factories

Si tu paquete contiene [factories de bases de datos](#), puedes usar el método `loadFactoriesFrom` para informar a Laravel de cómo cargarlos. El método `loadFactoriesFrom` acepta la ruta al factory de tu paquete como único argumento:

```
/** php
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    $this->loadFactoriesFrom(__DIR__.'/path/to/factories');
}
```

Una vez que el factory de tu paquete ha sido registrado, puedes usarlos en tus aplicación:

```
factory(Package\Namespace\Model::class)->create();
```

php

Traducciones

Si tu paquete contiene [archivos de traducción](#) puedes usar el método `loadTranslationsFrom` para informarle a Laravel cómo cargarlos. Por ejemplo, si tu paquete se llama `courier`, debes agregar lo siguiente al método `boot` de tu proveedor de servicios:

```
/*
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadTranslationsFrom(__DIR__ . '/path/to/translations', 'courier');
}
```

Las traducciones de paquetes se refencian usando la convención de sintaxis `package::file.line`. Por lo tanto, puedes cargar la línea `welcome` del paquete `courier` del archivo `messages` de la siguiente manera:

```
echo trans('courier::messages.welcome');
```

Publicación de traducciones

Si deseas publicar las traducciones de tu paquete en el directorio `resources/lang/vendor` de la aplicación, puedes usar el método `publishes` del proveedor de servicios. El método `publishes` acepta un arreglo de rutas de paquetes y sus ubicaciones de publicación deseadas. Por ejemplo, para publicar los archivos de traducción para el paquete `courier`, puedes 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 tu paquete ejecutan el comando Artisan `vendor:publish` de Laravel, las traducciones de tu paquete se publicarán en la ubicación de publicación especificada.

Vistas

Para registrar las [vistas](#) de tu paquete con Laravel necesitas decirle a Laravel dónde están ubicadas.

Puedes hacerlo utilizando el método `loadViewsFrom` del proveedor de servicios. El método

`loadViewsFrom` acepta dos argumentos: la ruta a sus plantillas de vista y el nombre de tu paquete.

Por ejemplo, si el nombre de tu paquete es `courier`, debe agregar lo siguiente al método `boot` de tu proveedor de servicios:

```
/*
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->loadViewsFrom(__DIR__ . '/path/to/views', 'courier');
}
```

Las vistas de paquete se refieren usando la convención de sintaxis `package::view`. Entonces, una vez que tu ruta de vista se registra en un proveedor de servicios, puedes cargar la vista `admin` del paquete `courier` de la siguiente manera:

```
Route::get('admin', function () {
    return view('courier::admin');
});
```

Desactivar vistas del paquete

Cuando utilizas el método `loadViewsFrom`, Laravel en realidad registra dos ubicaciones para sus vistas: el directorio `resources/views/vendor` de la aplicación y el directorio que tu especificas. Entonces, usando el ejemplo `courier`, Laravel primero comprobará si el desarrollador ha proporcionado una versión personalizada de la vista en `resources/views/vendor/courier`. Entonces, si la vista no se ha personalizado, Laravel buscará en el directorio de las vistas del paquete que has colocado en el método `loadViewsFrom`. Esto facilita a los usuarios del paquete personalizar o anular las vistas de tu paquete.

Publicación de vistas

Si desea que tus vistas estén disponibles para su publicación en el directorio

`resources/views/vendor` de la aplicación, puedes usar el método `publishes` del proveedor de servicios. El método `publishes` acepta una matriz de rutas de vista de paquete 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 ejecutan el comando Artisan `vendor:publish` de Laravel, las vistas de su paquete se copiarán en la ubicación especificada.

Comandos

Para registrar los comandos Artisan de tu paquete con Laravel puedes usar el método `commands`. Este método espera un arreglo con los nombres de clases de comando. Una vez que los comandos han sido registrados, puedes ejecutarlos usando [Artisan CLI](#):

```
/***
 * Bootstrap the application services.
 *
 * @return void
 */
public function boot()
{
    if ($this->app->runningInConsole()) {
        $this->commands([
            FooCommand::class,
            BarCommand::class,
        ]);
    }
}
```

php

Archivos públicos

Tu paquete puede tener archivos como JavaScript, CSS e imágenes. Para publicar estos archivos en el directorio `public` de la aplicación debes usar el método `publishes` del proveedor de servicios. En este ejemplo, también agregaremos una etiqueta de grupo de archivos `public`, que se puede usar para publicar grupos de archivos relacionados:

```
/***
 * Perform post-registration booting of services.
 *
 * @return void
 */
public function boot()
{
    $this->publishes([
        __DIR__ . '/path/to/assets' => public_path('vendor/courier'),
    ], 'public');
}
```

php

Ahora, cuando los usuarios de tu paquete ejecuten el comando `vendor:publish` tus archivos se copiarán en la ubicación especificada. Como normalmente necesitarás sobrescribir los archivos cada vez que se actualice el paquete, puedes usar el indicador `--force` :

```
php artisan vendor:publish --tag=public --force
```

php

Publicar grupos de archivos

Es posible que deseas publicar grupos de archivos y recursos de paquetes por separado. Por ejemplo, es posible que deseas permitir que los usuarios publiquen los archivos de configuración de su paquete sin verse obligados a publicar los archivos de tu paquete. Puede hacer esto "etiquetándolos" cuando llames 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 proveedor de servicios de paquetes:

```
/**  
 * 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');  
}
```

php

Ahora tus usuarios pueden publicar estos grupos por separado al hacer referencia a su etiqueta cuando ejecuten el comando `vendor:publish`:

```
php artisan vendor:publish --tag=config
```

php

Colas De Trabajo

- Introducción
 - Conexiones vs. colas
 - Notas y requisitos previos de driver
- Creación de trabajos
 - Generación de clases de trabajos
 - Estructura de clases
 - Middleware job
- Despachar trabajos
 - Despacho postergado
 - Envío sincrónico
 - Encadenamiento de trabajos
 - Personalizar la Cola y conexión
 - Especificar intentos máximos de trabajos / valores de timeout
 - Límites de rango
 - Manejo de errores
- Closures de dolas
- Ejecutar el worker de cola
 - Prioridades en cola
 - Workers de cola y despliegue
 - Expiraciones de trabajo y tiempos de espera
- Configuración de Supervisor
- Manejo de trabajos fallidos
 - Remediando trabajos fallidos
 - Eventos de trabajos fallidos
 - Reintentando trabajos fallidos
 - Ignorando modelos faltantes
- Eventos de trabajo

Introducción

TIP

Laravel ahora ofrece Horizon, un hermoso tablero y sistema de configuración para las colas motorizadas por Redis. Entra en [Horizon documentation](#) para más información.

Las colas de Laravel brindan una API unificada a través de una variedad de backends de cola diferentes como Beanstalk, Amazon SQS, Redis, o incluso una base de datos relacional. Las colas permiten diferir el procesamiento de una tarea que consume tiempo, como enviar un correo electrónico, para un momento posterior. Diferir estas tareas acelera drásticamente las solicitudes web en tu aplicación.

La configuración de cola está almacenada en `config/queue.php`. En este archivo encontrarás configuraciones de conexión para cada driver de cola incluido en el framework, que comprende una base de datos, [Beanstalkd](#), [Amazon SQS](#), [Redis](#), y un controlador sincrónico que ejecutará trabajos inmediatamente (para uso local). Un driver de cola `null` también está incluido, que descarta los trabajos completados de la cola.

Conexiones vs. colas

Antes de empezar con las colas de Laravel, es importante entender la distinción entre "conexiones" y "colas". En tu archivo `config/queue.php`, hay una opción de configuración `connections`. Esta opción define una conexión particular a un servicio de backend como Amazon SQS, Beanstalk o Redis. Sin embargo, cualquier conexión de cola dada puede tener múltiples "colas" las cuales pueden ser considerarse como diferentes pilas de trabajos en espera.

Ten en cuenta que cada ejemplo de configuración de conexión en el archivo `queue` contiene un atributo `queue`. Ésta es la cola por defecto a la cual los trabajos serán despachados cuando son enviados a una conexión dada. En otras palabras, si despachas un trabajo sin definir explícitamente a cuál cola debe ser despachado, el trabajo será colocado en la cola 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');
```

php

Algunas aplicaciones quizá no necesiten nunca insertar trabajos en múltiples colas, prefiriendo en su lugar tener una cola simple. Sin embargo, empujar trabajos a múltiples colas puede ser especialmente útil para aplicaciones que deseen priorizar o segmentar el procesamiento de sus trabajos, puesto que el

worker de cola de Laravel permite especificar cuáles colas deben ser procesadas de acuerdo a su prioridad. Por ejemplo, si se insertan trabajos a una cola `high` se puede ejecutar un worker que les dé mayor prioridad de procesamiento:

```
php artisan queue:work --queue=high,default
```

php

Notas y requisitos previos del driver

Base de datos

Para utilizar el driver de cola `database`, necesitarás una tabla de base de datos para mantener los trabajos. Para generar una migración que crea esta tabla, ejecute el comando Artisan `queue:table`. Una vez creada la migración, puede migrar la base de datos mediante el comando `migrate`:

```
php artisan queue:table
```

php

```
php artisan migrate
```

Redis

Para usar el controlador de cola `redis`, debes configurar una conexión a una base de datos Redis en tu archivo `config/database.php`.

Redis Cluster

Si tu conexión de cola Redis usa un Redis Cluster, tus nombres de cola deben contener un **key hash tag** ↗. Esto es requerido para asegurar que todas las llaves Redis para una determinada cola sean colocadas en el mismo hash slot:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => '{default}',
    'retry_after' => 90,
],
```

php

Bloqueo

Al usar la cola Redis, se puede usar la opción de configuración `block_for` para especificar por cuánto tiempo debería esperar el controlador para que un trabajo esté disponible antes de repetirse a través del bucle del worker y volver a consultar la base de datos Redis.

Ajustar este valor en la carga de cola puede ser más eficiente que consultar continuamente la base de datos Redis buscando nuevos trabajos. Por ejemplo, puedes establecer el valor en `5` para indicar que el controlador debe bloquearse por cinco segundos mientras espera a que un trabajo esté disponible:

```
'redis' => [
    'driver' => 'redis',
    'connection' => 'default',
    'queue' => 'default',
    'retry_after' => 90,
    'block_for' => 5,
],
```

php

Nota

Establecer `block_for` a `0` causará que los trabajadores de una cola se bloquen de forma indefinida hasta que una tarea esté disponible. Esto también evitará que señales como `SIGTERM` sean manejadas hasta que la siguiente tarea sea procesada.

Requisitos previos para otros controladores

Las siguientes dependencias son necesarias para sus controladores respectivos:

- Amazon SQS: `aws/aws-sdk-php ~3.0`
- Beanstalkd: `pda/pheanstalk ~4.0`
- Redis: `predis/predis ~1.0` o la extensión de PHP `phpredis`

Creación de trabajos

Generación de clases de trabajos

Por defecto, todos los trabajos que se pueden poner en cola para la aplicación son almacenados en el directorio `app/Jobs`. Si `app/Jobs` no existe, será creado cuando se ejecute el comando Artisan `make:job`. Puedes generar un nuevo trabajo en cola utilizando la CLI Artisan:

```
php artisan make:job ProcessPodcast
```

php

La clase generada implementará la interfaz `Illuminate\Contracts\Queue\ShouldQueue`, indicando a Laravel que el trabajo debe ser insertado a la cola de forma asíncrona.

Estructura de clases

Las clases de trabajo son muy sencillas, normalmente contienen un único método `handle` que se llama cuando la cola procesa el trabajo. Para empezar, echemos un vistazo a una clase de trabajo de ejemplo. En este ejemplo, vamos a pretender que administraremos un servicio de publicación de podcasts y necesitamos procesar los archivos de podcast cargados 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;  
    }  
}
```

php

```
/**
 * Execute the job.
 *
 * @param AudioProcessor $processor
 * @return void
 */
public function handle(AudioProcessor $processor)
{
    // Process uploaded podcast...
}
```

En este ejemplo, ten en cuenta que hemos podido pasar un [modelo Eloquent](#) directamente hacia el constructor del trabajo en cola. Debido al trait `SerializesModels` que el trabajo está usando, los modelos Eloquent y sus relaciones cargadas serán serializados y deserializados correctamente cuando el trabajo se esté procesando. Si tu trabajo en cola acepta un modelo Eloquent en su constructor, sólo el identificador para el modelo será serializado en la cola. Cuando el trabajo se maneja realmente, el sistema de cola volverá a recuperar automáticamente la instancia del modelo completo y sus relaciones cargadas desde la base de datos. Todo es totalmente transparente a tu aplicación y previene inconvenientes que pueden surgir de serializar instancias Eloquent completas.

El método `handle` es llamado cuando el trabajo es procesado por la cola. Ten en cuenta que podemos declarar el tipo de dependencias en el método `handle` del trabajo. El [contenedor de servicios](#) de Laravel automáticamente inyecta estas dependencias.

Si te gustaría tomar control sobre cómo el contenedor inyecta dependencias en el método `handle`, puedes usar el método `bindMethod` del contenedor. El método `bindMethod` acepta una función de retorno (callback) que recibe el trabajo y el contenedor. Dentro del callback, eres libre de invocar al método `handle` de la forma que deseas. Típicamente, deberías llamar a este método desde un [proveedor de servicios](#):

```
use App\Jobs\ProcessPodcast;                                         php

$this->app->bindMethod(ProcessPodcast::class.'@handle', function ($job, $app) {
    return $job->handle($app->make(AudioProcessor::class));
});
```

Nota

Los datos binarios, como los contenidos de imagen, deben ser pasados a través de la función `base64_encode` antes de ser pasados a un trabajo en cola. De otra forma, el trabajo podría no serializarse correctamente a JSON cuando es colocado en la cola.

Manejando relaciones

Dado que las relaciones cargadas también son serializadas, la cadena serializada de la tarea puede volverse algo larga. Para evitar que las relaciones sean serializadas, puedes llamar al método `withoutRelations` en el modelo al momento de establecer el valor de una propiedad. Este método retornará una instancia del modelo sin cargar ninguna relación:

```
/** php
 * Create a new job instance.
 *
 * @param \App\Podcast $podcast
 * @return void
 */
public function __construct(Podcast $podcast)
{
    $this->podcast = $podcast->withoutRelations();
}
```

Middleware job

El middleware job te permite envolver lógica personalizada alrededor de la ejecución de trabajos en cola, reduciendo el boilerplate en los mismos. Por ejemplo, considera el siguiente método `handle` el cual depende de las características de limitación de Redis para permitir que se procese sólo un trabajo cada cinco segundos:

```
/** php
 * Execute the job.
 *
 * @return void
 */
public function handle()
{
```

```
Redis::throttle('key')->block(0)->allow(1)->every(5)->then(function () {
    info('Lock obtained...');

    // Handle job...
}, function () {
    // Could not obtain lock...

    return $this->release(5);
});

}
```

Aunque este código es válido, la estructura del método `handle` se vuelve ruidosa dado que está llena de lógica de limitación de Redis. Además, esta lógica de limitación debe ser duplicada para cualquier otro trabajo que queramos limitar.

En lugar de limitar en el método `handle`, podemos definir un middleware job que maneja el límite. Laravel no tiene una ubicación por defecto para el middleware job, así que eres bienvenido a colocar el middleware job en cualquier sitio de tu aplicación. En este ejemplo, colocaremos el middleware en un directorio `app/Jobs/Middleware` :

```
<?php                                         php

namespace App\Jobs\Middleware;

use Illuminate\Support\Facades\Redis;

class RateLimited
{
    /**
     * Process the queued job.
     *
     * @param mixed $job
     * @param callable $next
     * @return mixed
     */
    public function handle($job, $next)
    {
        Redis::throttle('key')
            ->block(0)->allow(1)->every(5)
            ->then(function () use ($job, $next) {
                // Lock obtained...
            });
    }
}
```

```
        $next($job);
    }, function () use ($job) {
        // Could not obtain lock...

        $job->release(5);
    });
}
}
```

Como puedes ver, al igual que [el middleware route](#), el middleware job recibe el trabajo siendo procesado y un callback que debe ser invocado para continuar procesando el trabajo.

Luego de crear el middleware job, este puede ser agregado a una tarea retornándolas desde el método `middleware` de la tarea. Este método no existe en tareas creadas con el comando de Artisan `make:job`, así que necesitarás agregarla a tu propia definición de clase de la tarea:

```
use App\Jobs\Middleware\RateLimited;                                         php

/**
 * Get the middleware the job should pass through.
 *
 * @return array
 */
public function middleware()
{
    return [new RateLimited];
}
```

Despachar trabajos

Una vez escrita la clase de trabajo, se puede despachar usando el método `dispatch` en el mismo trabajo. Los argumentos pasados a `dispatch` serán entregados al constructor de trabajos:

```
<?php                                                               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);
    }
}
```

Despacho postergado

Si quieras postergar la ejecución de un trabajo en cola, puedes utilizar el método `delay` al despachar un trabajo. Por ejemplo, especifiquemos que un trabajo no debería estar disponible para procesamiento hasta 10 minutos después que haya sido despachado:

```
<?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)  
        ->delay(now()->addMinutes(10));  
}  
}
```

Nota

El servicio de cola Amazon SQS tiene un tiempo máximo de retraso de 15 minutos.

Despacho sincrónico

Si deseas enviar un trabajo inmediatamente (sincrónicamente), puedes usar el método `dispatchNow`. Al usar este método, el trabajo no se pondrá en cola y se ejecutará inmediatamente dentro del proceso actual:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use App\Jobs\ProcessPodcast;  
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::dispatchNow($podcast);  
    }  
}
```

Encadenamiento de trabajos

El encadenamiento de trabajos te permite especificar una lista de trabajos en cola que deben ser ejecutados en secuencia. Si un trabajo en la secuencia falla, el resto no será ejecutado. Para ejecutar una cadena de trabajos en cola, puedes utilizar el método `withChain` en cualquier trabajo a enviar:

```
ProcessPodcast::withChain([
    new OptimizePodcast,
    new ReleasePodcast
])->dispatch();
```

php

Nota

La eliminación de trabajos mediante el método `$this->delete()` no impedirá que se procesen los trabajos encadenados. La cadena sólo dejará de ejecutarse si falla un trabajo en la cadena.

Cola y conexión en cadena

Si quieras especificar la cola y conexión por defecto que debe ser usada para los trabajos encadenados, se puede usar los métodos `allOnConnection` and `allOnQueue`. Estos métodos especifican la conexión y nombre de cola que debe ser usado a menos que el trabajo en cola sea asignado explícitamente a una diferente conexión / cola:

```
ProcessPodcast::withChain([
    new OptimizePodcast,
    new ReleasePodcast
])->dispatch()->allOnConnection('redis')->allOnQueue('podcasts');
```

php

Personalizar La Cola Y La Conexión

Despachar a una cola específica

Al insertar trabajos en diferentes colas, puedes "categorizar" los trabajos en cola e incluso priorizar cuántos workers son asignados a las distintas colas. Sin embargo, es preciso resaltar que esto no inserta

trabajos en diferentes "conexiones" de cola definidas en el archivo de configuración de colas, sino en colas específicas dentro de una sola conexión. Para especificar la cola, se usa el método `onQueue` al despachar un trabajo:

```
<?php 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');
    }
}
```

Despachar a una conexión específica

Si estás trabajando con múltiples conexiones de cola, puedes especificar en cuál conexión deseas insertar un trabajo. Para especificar la conexión, utiliza el método `onConnection` al despachar el trabajo:

```
<?php 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('sq
    }
}
```

Puedes encadenar los métodos `onConnection` y `onQueue` para especificar la conexión y cola de un trabajo:

```
ProcessPodcast::dispatch($podcast)
    ->onConnection('sq
    ->onQueue('processing');
```

php

Alternativamente, puedes especificar `connection` como una propiedad en la clase del trabajo:

```
<?php

namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The queue connection that should handle the job.
     *
     * @var string
     */
    public $connection = 'sq
}
```

php

Especificar intentos máximos de un trabajo y valores de tiempos de espera (timeout)

Número de intentos máximo

Una forma de especificar el número máximo de veces que un trabajo puede ser intentado es mediante la opción `--tries` en la línea de comandos Artisan:

```
php artisan queue:work --tries=3
```

php

Sin embargo, puedes tomar un camino más granular definiendo el número máximo de intentos dentro de la clase de trabajos. Si el número máximo de intentos está especificado en el trabajo, precederá sobre el valor provisto en la línea de comandos:

```
<?php  
  
namespace App\Jobs;  
  
class ProcessPodcast implements ShouldQueue  
{  
    /**  
     * The number of times the job may be attempted.  
     *  
     * @var int  
     */  
    public $tries = 5;  
}
```

php

Intentos basados en tiempo

Como alternativa a definir cuántas veces un trabajo puede ser intentado antes de que falle, puedes definir en qué momento el trabajo debería pasar a tiempo de espera (timeout). Esto permite intentar un trabajo cualquier cantidad de veces dentro de un marco de tiempo dado. Para definir el momento en el que un trabajo debería pasar a timeout, se agrega un método `retryUntil` en la clase de trabajos:

```
/**  
 * Determine the time at which the job should timeout.  
 *  
 * @return \DateTime
```

php

```
 */
public function retryUntil()
{
    return now() ->addSeconds(5);
}
```

TIP

También puedes definir un método `retryUntil` en los listeners de eventos en cola.

Tiempo de espera (timeout)

Nota

La característica `timeout` está optimizada para PHP 7.1+ y la extensión `pcntl`.

De igual modo, el número máximo de segundos para ejecutar un trabajo pueden ser especificados usando la opción `--timeout` en la línea de comandos Artisan:

```
php artisan queue:work --timeout=30
```

php

Sin embargo, es posible querer definir el número máximo de segundos para ejecutar un trabajo dentro de su clase. Si el timeout está especificado en el trabajo, prevalecerá sobre cualquier otro timeout especificado en la línea de comandos:

```
<?php
```

php

```
namespace App\Jobs;

class ProcessPodcast implements ShouldQueue
{
    /**
     * The number of seconds the job can run before timing out.
     *
     * @var int
     */
    public $timeout = 120;
}
```

Límite de rango

Nota

Esta característica requiere que la aplicación pueda interactuar con un Redis server.

Si tu aplicación interactúa con Redis, puedes regular los trabajos en cola por tiempo o concurrencia. Esta característica puede ser de ayuda cuando los trabajos en cola interactúan con APIs que también poseen límite de frecuencia.

Por ejemplo, usando el método `throttle`, puedes regular cierto tipo de trabajo para que se ejecute sólo diez veces por minuto. Si no se puede obtener un bloqueo, normalmente debes liberar el trabajo de vuelta a la cola para que pueda volver a intentarlo más tarde:

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

php

TIP

En el ejemplo anterior, `key` puede ser cualquier cadena que identifique únicamente el tipo de trabajo que se quiere limitar. Por ejemplo, puedes desear construir la key basada en el nombre de clase del trabajo y las IDS de los modelos Eloquent en los cuáles opera.

Nota

Liberar un trabajo limitado de vuelta a la cola seguirá incrementando el número total de `intentos` del trabajo.

Alternativamente, puedes 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 solo debe ser modificado por un trabajo a la vez. Por ejemplo, utilizando el método `funnel`, puedes limitar los trabajos de un tipo dado para que solo 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);
});
```

php

TIP

Al utilizar límite de frecuencias, el número de intentos que el trabajo necesitará para ejecutarse exitosamente puede ser difícil de determinar. Por lo tanto, es útil combinar límite de frecuencias con [intentos basados en el tiempo](#).

Manejo de errores

Si una excepción es lanzada mientras el trabajo está siendo procesado, el trabajo será automáticamente liberado a la cola para que pueda ser intentado de nuevo. El trabajo continuará siendo liberado hasta que haya sido intentado el número de veces máximo permitido por tu aplicación. El número máximo de intentos es definido por la opción `--tries` usada en el comando Artisan `queue:work`. De forma alterna, el número máximo de intentos puede ser definido en la clase de trabajos en sí. Más información acerca de ejecutar el worker de cola [se puede encontrar a continuación](#).

Closures de colas

En lugar de enviar una clase de trabajo a la cola, también puedes enviar una Closure. Esto es ideal para tareas rápidas y simples que deben ejecutarse fuera del ciclo de solicitud actual:

```
$podcast = App\Podcast::find(1);

dispatch(function () use ($podcast) {
    $podcast->publish();
});
```

php

Al enviar Closures a la cola, el contenido del código del Closure está firmado criptográficamente para que no se pueda modificar en tránsito.

Ejecutar el worker de cola

Laravel incluye un worker de cola que procesará trabajos nuevos a medida que éstos son insertados en la cola. Puedes ejecutar el worker usando el comando Artisan `queue:work`. Ten en cuenta que una vez iniciado `queue:work`, continuará ejecutándose hasta que sea detenido manualmente o hasta que la terminal sea cerrada:

```
php artisan queue:work
```

php

TIP

Para mantener el proceso `queue:work` ejecutado permanentemente en segundo plano, debes usar un monitor de procesos como [Supervisor](#) para asegurar que el worker de cola no deja de ejecutarse.

Recuerda, los workers en cola son procesos de larga duración y almacenan el estado de la aplicación iniciada en la memoria. Como resultado, no notarán cambios en la base de código después de que se han iniciado. Por lo tanto, durante el proceso de despliegue, asegúrate de [reiniciar los workers de cola](#). Además, recuerda que cualquier estado estático creado o modificado por tu aplicación no será automáticamente reseteado entre tareas.

Alternativamente, puedes ejecutar el comando `queue:listen`. Al usar el comando `queue:listen`, no tienes que reiniciar manualmente el worker cuando quieras recargar tu código actualizado o resetear el estado de la aplicación; sin embargo, este comando no es tan eficiente como `queue:work`:

```
php artisan queue:listen
```

php

Especificando la conexión y cola

También puedes especificar qué conexión de cola debe utilizar el worker. El nombre de conexión pasado al comando `work` debe corresponder a una de las conexiones definidas en tu archivo de configuración `config/queue.php`:

```
php artisan queue:work redis
```

php

Puedes personalizar tu worker de colas más allá al solo procesar colas particulares para una conexión dada. Por ejemplo, si todos tus correos electrónicos son procesados en una cola `emails` en tu cola de conexión `redis`, puedes emitir el siguiente comando para iniciar un worker que solo procesa dicha cola:

```
php artisan queue:work redis --queue=emails
```

php

Procesar un sólo trabajo

La opción `--once` puede ser usada para indicarle al worker procesar sólo un trabajo de la cola:

```
php artisan queue:work --once
```

php

Procesar todos los trabajos en cola y luego salir

La opción `--stop-when-empty` puede ser usada para indicarle al worker procesar todos los trabajos y luego salir elegantemente. Esta opción puede ser útil al trabajar colas de Laravel con un contenedor Docker si deseas desactivar el contenedor cuando la cola esté vacía:

```
php artisan queue:work --stop-when-empty
```

php

Consideraciones de recursos

Los Daemon de workers de cola no "reinician" el framework antes de procesar cada trabajo. Por lo tanto, debes liberar cualquier recurso pesado luego de que cada trabajo sea completado. Por ejemplo, si estás realizando manipulación de imágenes con la librería GD, debes liberar la memoria cuando se termine con `imagedestroy`.

Prioridades de cola

A veces puedes desear priorizar cómo se procesan las colas. Por ejemplo, en tu `config/queue.php` puedes establecer la `queue` predeterminada para tu conexión `redis` en `low`. Sin embargo, ocasionalmente puedes desear insertar un trabajo en una cola de prioridad `high` de esta forma:

```
dispatch((new Job)->onQueue('high'));
```

php

Para iniciar un worker que verifique que todos los trabajos en la cola `high` sean procesados antes de continuar con los trabajos en `low`, pasa una lista de nombres de colas delimitada por comas al comando `work`:

```
php artisan queue:work --queue=high,low
```

php

Workers de Cola y despliegue

Debido a que los workers de cola son procesos de vida útil larga, no detectarán cambios en el código sin ser reiniciados. Así que la forma más sencilla de poner en producción una aplicación utilizando workers de cola es reiniciando los workers durante el proceso de despliegue. Puedes con elegancia reiniciar todos los workers ejecutando el comando `queue:restart`:

```
php artisan queue:restart
```

php

Este comando indicará a todos los workers de cola que "mueran" luego de terminar el procesamiento de su trabajo actual para que ningún trabajo existente se pierda. Como los workers de cola morirán cuando se ejecute el comando `queue:restart`, un administrador de procesos debe estar en ejecución, como `Supervisor` para reiniciar automáticamente los workers de la cola.

TIP

La cola utiliza `caché` para almacenar señales de reinicio, por lo que debes verificar si un driver de caché está configurado debidamente en tu aplicación antes de utilizar esta característica.

Expiraciones De Trabajo Y Tiempos De Espera

Expiración de trabajos

En tu 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 está siendo procesado. Por ejemplo, si el valor de `retry_after` es establecido en `90`, el trabajo será liberado de nuevo a la cola si se ha estado procesando por 90

segundos sin haber sido eliminado. Generalmente, debes fijar el valor de `retry_after` al número máximo de segundos que le toma razonablemente a tus trabajos ser completamente procesados.

Nota

La única conexión de cola que no contiene un valor `retry_after` es Amazon SQS. SQS reintentará el trabajo basándose en el Default Visibility Timeout [🔗](#) que es administrado dentro de la consola de AWS.

Worker timeouts

El comando Artisan `queue:work` expone una opción `--timeout`. `--timeout` especifica qué tanto el proceso maestro de cola de Laravel esperará antes de detener un worker de cola hijo que está procesando un trabajo. A veces un proceso de cola hijo puede "congelarse" por varias razones, como una llamada HTTP externa que no responde. La opción `--timeout` remueve los procesos congelados que han excedido el tiempo límite especificado:

```
php artisan queue:work --timeout=60
```

php

La opción de configuración `retry_after` y la opción CLI `--timeout` son diferentes, pero trabajan juntas para asegurar que los trabajos no se pierdan y que los trabajos se procesen exitosamente sólo una vez.

Nota

El valor `--timeout` siempre debe ser al menos unos segundos menor que el valor de configuración `retry_after`. Esto asegurará que un worker procesando un trabajo determinado siempre sea detenido antes que el trabajo se reintente. Si la opción `--timeout` es mayor al valor de configuración `retry_after`, los trabajos podrían ser procesados dos veces.

Duración de descanso del worker

Cuando hay trabajos disponibles en cola, el worker seguirá procesando trabajos sin retraso entre ellos. Sin embargo, la opción `sleep` determina por cuánto tiempo "dormirá" el worker si no hay nuevos

trabajos disponibles. Mientras duerme, el worker no procesará trabajos nuevos - los trabajos serán procesados luego de que el worker despierte.

```
php artisan queue:work --sleep=3
```

php

Configuración De Supervisor

Instalar Supervisor

Supervisor es un monitor de procesos para el sistema operativo Linux y reiniciará automáticamente tu proceso `queue:work` si éste falla. Para instalar Supervisor en Ubuntu, se puede usar el siguiente comando:

```
sudo apt-get install supervisor
```

php

TIP

Si configurar Supervisor por ti mismo suena abrumador, considera usar [Laravel Forge](#), el cual instalará y configurará Supervisor automáticamente para tus proyectos en Laravel.

Configurar Supervisor

Los archivos de configuración de Supervisor están almacenados generalmente en el directorio `/etc/supervisor/conf.d`. Dentro de este directorio, puedes crear cualquier número de archivos de configuración que le instruyan a Supervisor cómo monitorear los procesos. Por ejemplo, creamos un archivo de configuración `laravel-worker.conf` que inicie y monitoree el 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
stopwaitsecs=3600
```

php

En este ejemplo, la directiva `numprocs` le indicará a Supervisor ejecutar ocho procesos `queue:work` y monitorearlos todos, reiniciándolos automáticamente si fallan. Debes cambiar la porción `queue:work` de la directiva `command` para reflejar la conexión de cola deseada.

Nota

Debes asegurarte de que el valor de `stopwaitsecs` es mayor que el número de segundos consumido por tu tarea de más larga duración. De otra forma, Supervisor podría detener la tarea antes de que se termine de procesar.

Iniciar Supervisor

Una vez que el archivo de configuración haya sido creado, puedes actualizar la configuración de Supervisor e iniciar los procesos usando los siguientes comandos:

```
sudo supervisorctl reread  
sudo supervisorctl update  
sudo supervisorctl start laravel-worker:*
```

php

Para más información acerca de Supervisor, consulta [la documentación de Supervisor](#).

Manejo de trabajos fallidos

Algunas veces los trabajos en cola fallarán. Esto no es problema, las cosas no siempre salen como esperamos! Laravel incluye una forma conveniente de especificar el número máximo de veces que un trabajo debe ser intentado. Luego que un trabajo haya excedido esta cantidad de intentos, será insertado en la tabla de base de datos `failed_jobs`. Para crear una migración para la tabla `failed_jobs` puedes usar el comando `queue:failed-table`:

```
php artisan queue:failed-table  
php artisan migrate
```

php

Entonces, al ejecutar el `worker de cola`, debes especificar el número máximo de intentos que un trabajo debe intentarse usando la opción `--tries` en el comando `queue:work`. Si no especificas un valor para `--tries` los trabajos se intentarán indefinidamente:

```
php artisan queue:work redis --tries=3
```

php

Adicionalmente, puedes especificar cuantos segundos debe esperar Laravel antes de volver a intentar un trabajo que ha fallado usando la opción `--delay`. Por defecto, un trabajo se vuelve a intentar inmediatamente:

```
php artisan queue:work redis --tries=3 --delay=3
```

php

Si te gustaría configurar la demora del trabajo fallido por cada trabajo, puedes hacerlo definiendo una propiedad `retryAfter` en tu clase de cola de trabajos:

```
/**  
 * The number of seconds to wait before retrying the job.  
 *  
 * @var int  
 */  
public $retryAfter = 3;
```

php

Limpiar después de un trabajo fallido

Se puede definir un método `failed` directamente en la clase de trabajo, permitiendo realizar una limpieza específica de trabajo cuando una falla ocurre. Esta es la ubicación perfecta para enviar una alerta a tus usuarios o revertir cualquier acción realizada por el trabajo. La `Exception` que causó la falla en el trabajo será pasada al método `failed`:

```
<?php  
  
namespace App\Jobs;  
  
use Exception;  
use App\Podcast;  
use App\AudioProcessor;  
use Illuminate\Bus\Queueable;
```

php

```
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...
    }
}
```

Nota

El método `failed` no será llamado si la tarea fue despachada usando el método `dispatchNow`.

Eventos de trabajo fallido

Si quieras registrar un evento para ser llamado cuando un trabajo falle, puedes usar el método

`Queue::failing`. Este evento representa una gran oportunidad para notificarle a tu equipo por correo electrónico o por [Slack](#). Por ejemplo, puedes adjuntar una respuesta a este evento desde el `AppServiceProvider` incluido en Laravel:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\Queue;  
use Illuminate\Queue\Events\JobFailed;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Register the service provider.  
     *  
     * @return void  
     */  
    public function register()  
    {  
        //  
    }  
  
    /**  
     * Bootstrap any application services.  
     *  
     * @return void  
     */  
    public function boot()  
    {  
        Queue::failing(function (JobFailed $event) {  
            // $event->connectionName  
        });  
    }  
}
```

```
// $event->job  
// $event->exception  
});  
}  
}
```

Reintentando trabajos fallidos

Para visualizar todos los trabajos fallidos insertados en la tabla de base de datos `failed_jobs` se puede usar el comando Artisan `queue:failed` :

```
php artisan queue:failed
```

php

El comando `queue:failed` listará la ID del trabajo, su conexión, cola y el tiempo en el cual falló. La ID del trabajo puede ser usada para reintentar el trabajo fallido. Por ejemplo, para reintentar un trabajo fallido con una ID `5`, ejecuta el siguiente comando:

```
php artisan queue:retry 5
```

php

Para reintentar todos tus trabajos fallidos, ejecuta el comando `queue:retry` y pasa `all` como ID:

```
php artisan queue:retry all
```

php

Si deseas borrar un trabajo fallido, puedes usar el comando `queue:forget` :

```
php artisan queue:forget 5
```

php

Para eliminar todos los trabajos fallidos, puedes usar el comando `queue:flush` :

```
php artisan queue:flush
```

php

Ignorando modelos faltantes

Cuando inyectas un modelo Eloquent en un trabajo, se serializa automáticamente antes de colocarlo en la cola y se restaura cuando se procesa el trabajo. Sin embargo, si el modelo ha sido eliminado mientras el trabajo estaba esperando a ser procesado por un trabajador, tu trabajo puede fallar con la excepción `ModelNotFoundException`.

Por conveniencia, puedes elegir eliminar automáticamente los trabajos con modelos faltantes configurando la propiedad `deleteWhenMissingModels` de tu trabajo en `true`:

```
/**  
 * Delete the job if its models no longer exist.  
 *  
 * @var bool  
 */  
public $deleteWhenMissingModels = true;
```

php

Eventos de trabajo

Usando los métodos `before` y `after` en la facade `Queue`, puedes especificar funciones de retorno (callbacks) para que sean ejecutadas antes o después de que un trabajo en cola sea procesado. Estas callbacks son una gran oportunidad para realizar registro adicional o incrementar estadísticas para un panel de control. Generalmente, debes llamar a estos métodos desde un [proveedor de servicios](#). Por ejemplo puedes usar `AppServiceProvider`, incluido en 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  
     */
```

php

```

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()
{
    //
}

```

Usando el método `looping` en la facade `Queue`, puedes especificar funciones de retorno (callbacks) que se ejecuten antes que el worker intente recuperar un trabajo de una cola. Por ejemplo, quizás necesites registrar una Closure para deshacer cualquier transacción abierta por un trabajo fallido anteriormente:

```

Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});

```

Programación de tareas

- Introducción
- Definición de programaciones
 - Programando comandos de Artisan
 - Programando trabajos en cola
 - Programando comandos de shell
 - Programando opciones de frecuencias
 - Zonas Horarias
 - Previniendo superposición de tareas
 - Ejecutando tareas en un servidor
 - Tareas en segundo plano
 - Modo de mantenimiento
- Resultado de la Tarea
- Hooks de tareas

Introducción

En el pasado, es posible que hayas generado una entrada Cron para cada tarea que necesitabas programar en su servidor. Sin embargo, esto puede convertirse rápidamente en un sufrimiento, dado que tu programación de tareas no está en el control de versiones y debes hacer SSH a tu servidor para agregar entradas Cron adicionales.

El programador de comandos de Laravel te permite definir tu programación de comandos de forma fluida y expresiva dentro de Laravel. Al usar el programador, una sola entrada Cron es necesaria en tu servidor. Tu programación de tareas es definida en el método `schedule` del archivo `app/Console/Kernel.php`. Para ayudarte a comenzar, un ejemplo sencillo está definido dentro del método.

Iniciando el programador

Al usar el programador, sólo necesitas agregar la siguiente entrada Cron a tu servidor. Si no sabes cómo agregar entradas Cron a tu servidor, considera usar un servicio como [Laravel Forge](#) que puede

administrar las entradas Cron por ti:

```
* * * * * php /path-to-your-project/artisan schedule:run >> /dev/null 2>&1
```

php

Este Cron llamará al programador de tareas de Laravel cada minuto. Cuando el comando `schedule:run` es ejecutado, Laravel evaluará tus tareas programadas y ejecutará las tareas pendientes.

Definición de programaciones

Puedes definir todas tus tareas programadas en el método `schedule` de la clase `App\Console\Kernel`. Para empezar, veamos un ejemplo de programación de una tarea. En este ejemplo, programaremos una `closure` que será llamada cada día a medianoche. Dentro de la `Closure` ejecutaremos un consulta a la base de datos para vaciar una tabla:

```
<?php
```

```
namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;
use Illuminate\Support\Facades\DB;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        //
    ];

    /**
     * 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();
}
```

Además de programar usando Closures, también puedes usar [objetos invocables](#). Los objetos invocables son clases PHP sencillas que contienen un método `__invoke`:

```
$schedule->call(new DeleteRecentUsers)->daily();
```

php

Programando comandos de artisan

Además de programador llamadas a Closures, también puedes programar comandos de Artisan y comandos del sistema operativo. Por ejemplo, puedes usar el método `command` para programar un comando de Artisan usando ya sea el nombre del comando o de la clase:

```
$schedule->command('emails:send Taylor --force')->daily();

$schedule->command(EmailsCommand::class, ['Taylor', '--force'])->daily();
```

php

Programando trabajos en colas

El método `job` puede ser usado para programar un trabajo en cola. Este método proporciona una forma conveniente de programar trabajos sin usar el método `call` para crear Closures de forma manual para agregar el trabajo a la cola:

```
$schedule->job(new Heartbeat)->everyFiveMinutes();

// Dispatch the job to the "heartbeats" queue...
$schedule->job(new Heartbeat, 'heartbeats')->everyFiveMinutes();
```

php

Programando comandos de shell

El método `exec` puede ser usado para emitir un comando al sistema operativo:

```
$schedule->exec('node /home/forge/script.js')->daily();
```

php

Programando opciones de frecuencias

Hay una variedad de programaciones que puedes asignar a tu tarea:

Método	Descripción
<code>->cron('* * * * *');</code>	Ejecuta la tarea en una programación Cron personalizada
<code>->everyMinute();</code>	Ejecuta la tarea cada minuto
<code>->everyFiveMinutes();</code>	Ejecuta la tarea cada cinco minutos
<code>->everyTenMinutes();</code>	Ejecuta la tarea cada diez minutos
<code>->everyFifteenMinutes();</code>	Ejecuta la tarea cada quince minutos
<code>->everyThirtyMinutes();</code>	Ejecuta la tarea cada treinta minutos
<code>->hourly();</code>	Ejecuta la tarea cada hora
<code>->hourlyAt(17);</code>	Ejecuta la tarea cada hora en el minuto 17
<code>->daily();</code>	Ejecuta la tarea cada día a la medianoche
<code>->dailyAt('13:00');</code>	Ejecuta la tarea cada día a las 13:00
<code>->twiceDaily(1, 13);</code>	Ejecuta la tarea cada día a las 1:00 y a las 13:00
<code>->weekly();</code>	Ejecuta la tarea cada domingo a las 00:00
<code>->weeklyOn(1, '8:00');</code>	Ejecuta a tarea cada semana los lunes a las 8:00
<code>->monthly();</code>	Ejecuta la tarea el primer día de cada mes a las 00:00

Método	Descripción
<code>->monthlyOn(4, '15:00');</code>	Ejecuta la tarea el 4 de cada mes a las 15:00
<code>->quarterly();</code>	Ejecuta la tarea el primer día de cada trimestre a las 00:00
<code>->yearly();</code>	Ejecuta la tarea el primer día de cada año a las 00:00
<code>->timezone('America/New_York');</code>	Establece la zona horaria

Estos métodos pueden ser combinados con restricciones adicionales para crear programaciones más ajustadas que sólo se ejecutan en determinados días de la semana. Por ejemplo, para programar un comando para que sea ejecutado los lunes:

```
// Ejecuta una vez por semana los martes a la 1 PM...
$schedule->call(function () {
    //
})->weekly()->mondays()->at('13:00');

// Ejecuta cada hora de 8 AM a 5 PM los días laborales...
$schedule->command('foo')
    ->weekdays()
    ->hourly() method
    ->timezone('America/Chicago')
    ->between('8:00', '17:00');
```

php

A continuación hay una lista de restricciones de programación adicionales:

Method	Description
<code>->weekdays();</code>	Limita la tarea a los días laborales
<code>->weekends();</code>	Limita la tarea a los fines de semana
<code>->sundays();</code>	Limita la tarea a los domingos
<code>->mondays();</code>	Limita la tarea a los lunes

Method	Description
<code>->tuesdays();</code>	Limita la tarea los martes
<code>->wednesdays();</code>	Limita la tarea a los miércoles
<code>->thursdays();</code>	Limita la tarea a los jueves
<code>->fridays();</code>	Limita la tarea a los viernes
<code>->saturdays();</code>	Limita la tarea a los sábados
<code>->between(\$start, \$end);</code>	Limita la tarea para ser ejecutado entre \$start y \$end
<code>->when(Closure);</code>	Limita la tarea dependiendo de una prueba de veracidad
<code>->environments(\$env);</code>	Limita la tarea a ambientes específicos

Restricciones de tiempo between

El método `between` puede ser usado para limitar la ejecución de una tarea dependiendo de la hora del día:

```
$schedule->command('reminders:send')
    ->hourly()
    ->between('7:00', '22:00');
```

php

De forma similar, el método `unlessBetween` puede ser usado para excluir la ejecución de una tarea por un periodo de tiempo:

```
$schedule->command('reminders:send')
    ->hourly()
    ->unlessBetween('23:00', '4:00');
```

php

Restricciones de veracidad

El método `when` puede ser usado para limitar la ejecución de una tarea en base al resultado de un test de veracidad dado. En otras palabras, si la `Closure` dada retorna `true`, la tarea será ejecutada

siempre y cuando ninguna otra restricción prevenga que la tarea de ser ejecutada:

```
$schedule->command('emails:send')->daily()->when(function () {  
    return true;  
});
```

php

El método `skip` puede ser visto como el inverso de `when`. Si el método `skip` retorna `true`, la tarea programada no será ejecutada:

```
$schedule->command('emails:send')->daily()->skip(function () {  
    return true;  
});
```

php

Al usar métodos `when` encadenados, el comando programado sólo será ejecutado si todas las condiciones `when` retornan `true`.

Restricciones de entorno

El método `environments` se puede utilizar para ejecutar tareas sólo en los entornos dados:

```
$schedule->command('emails:send')  
    ->daily()  
    ->environments(['staging', 'production']);
```

php

Zonas horarias

Usando el método `timezone`, puedes especificar que el tiempo de una tarea programada debe ser interpretada en una zona horaria dada:

```
$schedule->command('report:generate')  
    ->timezone('America/New_York')  
    ->at('02:00')
```

php

Si estás asignando la misma zona horaria a todas tus tareas programadas, puedes desear definir un método `scheduleTimezone` en tu archivo `app/Console/Kernel.php`. Este método debería retornar la zona horaria por defecto que debe ser asignada a todas las tareas programadas.

```
/**  
 * Get the timezone that should be used by default for scheduled events.  
 *  
 * @return \DateTimeZone|string|null  
 */  
protected function scheduleTimezone()  
{  
    return 'America/Chicago';  
}
```

php

Nota

Recuerda que algunas zonas horarias usan horario de verano. Cuando ocurren cambios por horario de verano, tu tarea programada puede ejecutarse dos veces o puede no ser ejecutada. Por esto, recomendamos evitar programación con zona horaria en la medida de lo posible.

Previniendo superposición de tareas

Por defecto, las tareas programadas serán ejecutadas incluso si la instancia anterior de la tarea todavía está en ejecución. Para evitar esto, puedes usar el método `withoutOverlapping` :

```
$schedule->command('emails:send')->withoutOverlapping();
```

php

En este ejemplo, el comando de Artisan `emails:send` será ejecutado cada minuto si ya no está siendo ejecutado. El método `withoutOverlapping` es especialmente útil si tienes tareas que varían drásticamente en su tiempo de ejecución, evitando que puedas predecir exactamente cuánto tiempo tomará una tarea.

Si es necesario, puedes especificar cuántos minutos deben pasar antes de que el bloqueo "sin superposición" expire. Por defecto, el bloqueo expirará luego de 24 horas:

```
$schedule->command('emails:send')->withoutOverlapping(10);
```

php

Ejecutando tareas en un servidor

Nota

Para utilizar esta característica, tu aplicación debe estar usando el controlador de caché `memcached` o `redis` como predeterminado. Además, todos los servidores deben comunicarse al mismo servidor central de caché.

Si tu aplicación está siendo ejecutada en múltiples servidores, puedes limitar un trabajo programado a sólo ejecutarse en un servidor. Por ejemplo, asume que se tiene una tarea programada que genera un reporte nuevo cada viernes en la noche. Si el programador de tareas está siendo ejecutado en tres servidores de worker, la tarea programada se ejecutará en todos y generará el reporte tres veces. ¡No es bueno!

Para indicar que la tarea debe ejecutarse sólo en un servidor, usa el método `onOneServer` al definir la tarea programada. El primer servidor en obtener la tarea asegurará un bloqueo atómico en el trabajo para prevenir que otros servidores ejecuten la misma tarea al mismo tiempo:

```
$schedule->command('report:generate')
    ->fridays()
    ->at('17:00')
    ->onOneServer();
```

php

Tareas en segundo plano

Por defecto, múltiples comandos programados al mismo tiempo se ejecutarán secuencialmente. Si tienes comandos de ejecución larga, esto puede causar que los siguientes comandos sean ejecutados mucho más tarde que lo esperado. Si deseas ejecutar comandos en segundo plano para que todos funcionen de forma simultánea, puedes usar el método `runInBackground` :

```
$schedule->command('analytics:report')
    ->daily()
    ->runInBackground();
```

php

Nota

El método `runInBackground` sólo puede ser usado al programar tareas mediante los métodos `command` y `exec`.

Modo de mantenimiento

Las tareas programadas de Laravel no serán ejecutadas cuando Laravel está en [modo de mantenimiento](#), dado que no queremos que tus tareas interfieran con cualquier mantenimiento inacabado que puedes estar realizando en tu servidor. Sin embargo, si quieres forzar la ejecución de una tarea incluso en modo de mantenimiento, puedes usar el método `evenInMaintenanceMode` :

```
$schedule->command('emails:send')->evenInMaintenanceMode();
```

php

Resultado de la tarea

El programador de Laravel proporciona múltiples métodos convenientes para trabajar con el resultado generado por una tarea programada. Primero, usando el método `sendOutputTo`, puedes enviar el resultado a un archivo para una inspección posterior:

```
$schedule->command('emails:send')
    ->daily()
    ->sendOutputTo($filePath);
```

php

Si quieres agregar el resultado a un archivo dado, puedes usar el método `appendOutputTo` :

```
$schedule->command('emails:send')
    ->daily()
    ->appendOutputTo($filePath);
```

php

Usando el método `emailOutputTo`, puedes enviar el resultado a una dirección de correo electrónico de tu preferencia. Antes de enviar por correo electrónico el resultado de una tarea, debes configurar los [servicios de correo electrónico](#) de Laravel:

```
$schedule->command('foo')
    ->daily()
    ->sendOutputTo($filePath)
    ->emailOutputTo('foo@example.com');
```

php

Si solo quieres enviar el resultado por correo electrónico si el comando falla, usa el método

`emailOutputOnFailure` :

```
$schedule->command('foo')
    ->daily()
    ->emailOutputOnFailure('foo@example.com');
```

php

Nota

Los métodos `emailOutputTo`, `emailOutputOnFailure`, `sendOutputTo` y `appendOutputTo` son exclusivos para los métodos `command` y `exec`.

Hooks de tareas

Usando los métodos `before` y `after`, puedes especificar código que será ejecutado antes y después de que la tarea programada sea completada:

```
$schedule->command('emails:send')
    ->daily()
    ->before(function () {
        // Task is about to start...
    })
    ->after(function () {
        // Task is complete...
    });
});
```

php

Los métodos `onSuccess` y `onFailure` te permiten especificar código a ejecutar si la tarea programada tiene éxito o falla:

```
$schedule->command('emails:send')
    ->daily()
    ->onSuccess(function () {
        // The task succeeded...
    })
    ->onFailure(function () {
        // The task failed...
    });
});
```

php

Haciendo ping a URLs

Usando los métodos `pingBefore` y `thenPing`, el programador de tareas puede automáticamente hacer ping a una URL dada antes o después de que una tarea sea completada. Este método es útil para notificar a un servicio externo, como [Laravel Envoyer](#), que tu tarea programada está comenzando o ha finalizado su ejecución:

```
$schedule->command('emails:send')
    ->daily()
    ->pingBefore($url)
    ->thenPing($url);
```

php

Los métodos `pingBeforeIf` y `thenPingIf` pueden ser usados para hacer ping a una URL dada sólo si la condición dada es `true`:

```
$schedule->command('emails:send')
    ->daily()
    ->pingBeforeIf($condition, $url)
    ->thenPingIf($condition, $url);
```

php

Los métodos `pingOnSuccess` y `pingOnFailure` pueden ser usados para hacer ping a una URL dada sólo si la tarea tiene éxito o falla:

```
$schedule->command('emails:send')
    ->daily()
    ->pingOnSuccess($successUrl)
    ->pingOnFailure($failureUrl);
```

php

Todos los métodos de ping requieren el paquete HTTP Guzzle. Puedes agregar Guzzle a tu proyecto usando el gestor de paquetes Composer:

```
composer require guzzlehttp/guzzle
```

php

Bases de datos: primeros pasos

- Introducción
 - Configuración
 - Conexiones de lectura y escritura
 - Usando múltiples conexiones de bases de datos
- Ejecutando consultas SQL nativas
- Listeners de eventos de consultas
- Transacciones de bases de datos

Introducción

Laravel hace que la interacción con las bases de datos sea extremadamente fácil a través de una variedad de backends de bases de datos usando SQL nativo, el constructor de consultas query builder y el ORM Eloquent. Actualmente, Laravel soporta cuatro bases de datos.

- MySQL 5.6+ ([Política de la versión](#))
- PostgreSQL 9.4+ ([Política de la versión](#))
- SQLite 3.8.8+
- SQL Server 2017+ ([Política de la versión](#))

Configuración

La configuración de base de datos para tu aplicación está localizada en `config/database.php`.

Puedes definir todo lo concerniente a tus conexiones de bases de datos, y también especificar qué conexión debería ser usada por defecto. Ejemplos para la mayoría de los sistemas de bases de datos soportados son proporcionados en este archivo.

Por defecto, la configuración de entorno de muestra de Laravel está lista para usar con Laravel Homestead, la cual es una máquina virtual conveniente para el desarrollo con Laravel en tu máquina local. Eres libre de modificar esta configuración de acuerdo a tus necesidades de tu base de datos local.

Configuración de SQLite

Después de la creación de una nueva base de datos SQLite usando un comando tal como `touch database/database.sqlite`, puedes configurar fácilmente tus variables de entorno para apuntar a esta base de datos creada recientemente al usar la ruta absoluta a la base de datos.

```
DB_CONNECTION=sqlite  
DB_DATABASE=/absolute/path/to/database.sqlite
```

php

Para habilitar las claves foráneas en conexiones de SQLite, debes establecer la variable de entorno

```
DB_FOREIGN_KEYS a true :
```

```
DB_FOREIGN_KEYS=true
```

php

Configuración usando URLs

Típicamente, las conexiones a bases de datos son configuradas usando múltiples valores de configuración como `host`, `database`, `username`, `password`, etc. Cada uno de esos valores de configuración tiene su propia variable de entorno correspondiente. Esto quiere decir que al configurar tu información de conexión a la base de datos en un servidor de producción, necesitas administrar múltiples variables de entorno.

Algunos proveedores de bases de datos administrados como Heroku proporcionan una única "URL" de base de datos que contiene toda la información de conexión para la base de datos en una única cadena. Una URL de base de datos de ejemplo podría verse de la siguiente manera:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

php

Estas URLs típicamente siguen una convención de esquema estándar:

```
driver://username:password@host:port/database?options
```

php

Por conveniencia, Laravel soporta dichas URLs como alternativa a configurar tu base de datos con múltiples opciones de configuración. Si la opción de configuración `url` (o variable de entorno `DATABASE_URL` correspondiente) está presente, esta será usada para extraer la conexión a la base de datos y la información de credenciales.

Conexiones de lectura y escritura

Algunas veces puedes desear contar con una conexión de base de datos para los comandos SELECT y otra para los comandos UPDATE y DELETE. Laravel hace esto una tarea fácil, y las conexiones propias siempre serán usadas si estás usando consultas nativas, el constructor de consultas Query Builder o el ORM Eloquent.

Para ver cómo las conexiones de lectura / escritura deberían ser configuradas, vamos a mirar este ejemplo:

```
mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
        ],
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
],
```

Observa que tres claves han sido agregadas al arreglo de configuración: `read` , `write` y `sticky` . Las claves `read` y `write` tienen valores de arreglo contenido una sola clave: la dirección ip del `host` . El resto de las opciones de la base de datos para las conexiones `read` y `write` serán tomadas del arreglo principal `mysql` .

Únicamente necesitas colocar elementos en los arreglos `read` y `write` si deseas sobreescribir los valores del arreglo principal. Así, en este caso, `192.168.1.1` será usado como la máquina para la conexión de "lectura", mientras que `192.168.1.3` será usada para la conexión de "escritura". Las

credenciales de bases de datos, prefijo, conjunto de caracteres, y todas las demás opciones en el arreglo principal `mysql` serán compartidas a través de ambas conexiones.

La opción `sticky`

La opción `sticky` es un valor *opcional* que puede ser usado para permitir la lectura inmediata de registros que han sido escritos a la base de datos durante el ciclo de solicitudes actual. Si la opción `sticky` está habilitada y una operación de "escritura" ha sido ejecutada contra la base de datos durante el ciclo de solicitudes actual, cualquiera de las operaciones de "lectura" hasta aquí usarán la conexión "write". Esto asegura que cualquier dato escrito durante el ciclo de solicitud pueda ser leído inmediatamente de la base de datos durante la misma solicitud. Es posible para ti decidir si este es el comportamiento deseado para tu aplicación.

Usando conexiones de bases de datos múltiples

Cuando estamos usando conexiones múltiples, puedes acceder a cada conexión por medio del método `connection` en el Facade `DB`. El nombre `name` pasado al método de conexión `connection` debería corresponder a una de las conexiones listadas en tu archivo de configuración `config/database.php`:

```
$users = DB::connection('foo')->select(...);
```

php

También puedes acceder a los datos nativos de la instancia PDO subyacente que usa el método `getPdo` en una instancia de conexión:

```
$pdo = DB::connection()->getPdo();
```

php

Ejecutando consultas SQL nativas

Una vez que has configurado tu conexión de base de datos, puedes ejecutar consultas usando el Facade `DB`. La clase facade `DB` proporciona métodos para cada tipo de consulta: `select`, `update`, `insert`, `delete` y `statement`.

Ejecutando una consulta select

Para ejecutar una consulta básica, puedes usar el método `select` en el facade `DB`:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use Illuminate\Support\Facades\DB;  
  
class UserController extends Controller  
{  
    /**  
     * Muestra una lista de todos los usuarios de la aplicación.  
     *  
     * @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 SQL nativa; en este caso está parametrizada, mientras el segundo argumento es cualquier parámetro a enlazar que necesita estar conectado a la consulta. Típicamente, estos son los valores de las restricciones de cláusula `where`. El enlazamiento de parámetro proporciona protección contra ataques de inyección SQL.

El método `select` siempre devolverá un arreglo de resultados. Cada resultado dentro del arreglo será un objeto `stdClass` de PHP, permitiendo que accedas a los valores de los resultados:

```
foreach ($users as $user) {  
    echo $user->name;  
}
```

Usando enlaces nombrados

En lugar de usar `?` para representar tus enlaces (bindings) de parámetros, puedes ejecutar una consulta usando enlaces nombrados:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

php

Ejecutando una instrucción insert

Para ejecutar una instrucción `insert`, puedes usar el método `insert` en la clase facade `DB`. Igual que `select`, este método toma la consulta SQL nativa como su argumento inicial y lo enlaza con su argumento secundario:

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

php

Ejecutando una instrucción update

El método `update` debería ser usado para actualizar los registros existentes en la base de datos. El número de filas afectadas por la instrucción serán devueltas:

```
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

php

Ejecutando una instrucción delete

El método `delete` debería ser usado para eliminar registros de la base de datos. Al igual que `update`, el número de filas afectadas será devuelto:

```
$deleted = DB::delete('delete from users');
```

php

Ejecutando una instrucción general

Algunas instrucciones de bases de datos no devuelven algún valor. Para estos tipos de operaciones, puedes usar el método `statement` en la clase facade `DB`:

```
DB::statement('drop table users');
```

php

Listeners de eventos de consultas

Si prefieres recibir cada consulta SQL ejecutada por tu aplicación, puedes usar el método `listen`. Este método es útil para registrar consultas o depurar. Puedes registrar tus listeners de consultas en un proveedor de servicio:

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\Facades\DB;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Inicializa cualquiera de los servicios de la aplicación.  
     *  
     * @return void  
     */  
    public function boot()  
    {  
        DB::listen(function ($query) {  
            // $query->sql  
            // $query->bindings  
            // $query->time  
        });  
    }  
  
    /**  
     * Registra el proveedor de servicio.  
     *  
     * @return void  
     */  
    public function register()  
    {  
        //  
    }  
}
```

php

Transacciones de bases de datos

Puedes usar el método `transaction` en la clase facade `DB` para ejecutar un conjunto de operaciones dentro de una transacción de base de datos. Si un error de excepción es arrojado dentro del código `Closure` de la transacción, la transacción automáticamente terminará con un rollback. Si el código `Closure` se ejecuta correctamente, la transacción terminará automáticamente con un commit. No necesitas preocuparte por hacer rollback o commit manualmente mientras estés usando el método `transaction`:

```
DB::transaction(function () {  
    DB::table('users')->update(['votes' => 1]);  
  
    DB::table('posts')->delete();  
});
```

php

Manejando deadlocks (bloqueo mutuo)

El método `transaction` acepta un segundo argumento opcional el cual define el número de veces que la ejecución de una transacción debería ser reintentada cuando un ocurra un deadlock. Una vez que estos intentos hayan sido completados sin éxito, un error de excepción será arrojado:

```
DB::transaction(function () {  
    DB::table('users')->update(['votes' => 1]);  
  
    DB::table('posts')->delete();  
}, 5);
```

php

Usando transacciones manualmente

Si prefieres empezar una transacción manualmente y tener control total sobre rollbacks y commits, podrías usar el método `beginTransaction` de la clase facade `DB`:

```
DB::beginTransaction();
```

php

Puedes hacer rollback de la transacción por medio del método `rollBack`:

```
DB::rollBack();
```

php

Finalmente, puedes confirmar una transacción por medio del método `commit` :

```
DB::commit();
```

php

TIP

Los métodos de transacción del Facade `DB` controlan las transacciones para ambos backends de bases de datos del constructor de consultas query builder y el ORM Eloquent.

Base de datos: constructor de consultas (query builder)

- Introducción
- Obteniendo los resultados
 - Particionando los resultados
 - Agrupamientos
- Selects
- Expresiones sin procesar (raw)
- Joins
- Uniones
- Cláusulas where
 - Agrupamiento de parámetros
 - Cláusulas exists where
 - Cláusulas where JSON
- Ordenamiento, agrupamiento, límite y desplazamiento
- Cláusulas condicionales
- Inserciones

- Actualizaciones
 - Actualizando columnas JSON
 - Incremento y decremento
- Eliminaciones
- Bloqueo pesimista
- Depuración

Introducción

El constructor de consultas (query builder) de Base de datos de Laravel, proporciona una interface fluida y conveniente para la creación y ejecución de consultas de bases de datos. Puede ser usado para ejecutar las principales operaciones de bases de datos en tu aplicación y funciona en todos los sistemas de bases de datos soportados.

El constructor de consultas de Laravel usa enlazamiento de parámetros PDO para proteger tu aplicación contra ataques de inyección SQL. No hay necesidad de limpiar cadenas que están siendo pasadas como enlaces.

Nota

PDO no admite nombres de columna de enlace (binding). Por lo tanto, nunca debes permitir que la entrada de usuario dicte los nombres de columna a los que hacen referencia tus consultas, incluidas las columnas "ordenar por", etc. Si debes permitir que el usuario seleccione ciertas columnas para consultar, valida siempre los nombres de las columnas con una lista blanca de columnas permitidas.

Obteniendo los resultados

Obteniendo todas las filas de una tabla

Puedes usar el método `table` de la clase facade `DB` para empezar una consulta. El método `table` devuelve una instancia para construir consultas fáciles de entender para la tabla dada, permitiendo que encadenes más restricciones dentro de la consulta y recibas finalmente los resultados usando el método `get` :

```
<?php
```

php

```
namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

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

El método `get` devuelve una colección de la clase `Illuminate\Support\Collection` que contiene los resultados donde cada resultado es una instancia del objeto `StdClass` de PHP. Puedes acceder al valor de cada columna accediendo a la columna como una propiedad del objeto:

```
foreach ($users as $user) {
    echo $user->name;
}
```

php

Obteniendo una sola fila / columna de una tabla

Si solamente necesitas recuperar una sola fila de la tabla de la base de datos, puedes usar el método `first`. Este método devolverá un solo objeto `StdClass` :

```
$user = DB::table('users')->where('name', 'John')->first();

echo $user->name;
```

php

Si no necesitas una fila completa, puedes extraer un solo valor de un registro usando el método `value`. Este método devolverá directamente el valor de la columna:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

php

Para obtener una sola fila por su valor de columna `id`, use el método `find`:

```
$user = DB::table('users')->find(3);
```

php

Obteniendo una lista de valores de columna

Si prefieres obtener una Colección que contenga los valores de una sola columna, puedes usar el método `pluck`. En el siguiente ejemplo, obtendrémos una colección de títulos de rol:

```
$titles = DB::table('roles')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

php

También puedes especificar una columna clave personalizada para la colección retornada:

```
$roles = DB::table('roles')->pluck('title', 'name');

foreach ($roles as $name => $title) {
    echo $title;
}
```

php

Particionando los resultados

Si necesitas trabajar con miles de registros de bases de datos, considera usar el método `chunk`. Este método obtiene una partición pequeña de los resultados cada vez y pone cada partición dentro de un `Closure` para su procesamiento. Este método es muy útil para escribir [comandos de Artisan](#) que procesan miles de registros. Por ejemplo, vamos a trabajar con la tabla completa `users` en particiones de 100 registros cada vez:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {  
    foreach ($users as $user) {  
        //  
    }  
});
```

php

Puedes parar de obtener particiones para que no sean procesadas al devolver `false` en el código `Closure`:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {  
    // Process the records...  
  
    return false;  
});
```

php

Si estás actualizando registros de base de datos mientras particionas resultados, los resultados de particiones podrían cambiar en formas inesperadas. Entonces, cuando se actualicen los registros mientras se partitiona, siempre es mejor usar el método `chunkById` en su lugar. Este método paginará automáticamente los resultados basándose en la llave primaria del registro:

```
DB::table('users')->where('active', false)  
->chunkById(100, function ($users) {  
    foreach ($users as $user) {  
        DB::table('users')  
            ->where('id', $user->id)  
            ->update(['active' => true]);  
    }  
});
```

php

Nota

Al actualizar o eliminar registros dentro del callback de la partición, cualquier cambio en la clave primaria o claves foráneas podría afectar a la consulta de la partición. Esto podría potencialmente dar lugar a que los registros no se incluyan en los resultados particionados.

Agrupamientos

El constructor de consultas también proporciona una variedad de métodos de agrupamiento tales como `count` , `max` , `min` , `avg` y `sum` . Puedes ejecutar cualquiera de estos métodos después de construir tu consulta:

```
$users = DB::table('users')->count();  
  
$price = DB::table('orders')->max('price');
```

php

Puedes combinar estos métodos con otras cláusulas:

```
$price = DB::table('orders')  
    ->where('finalized', 1)  
    ->avg('price');
```

php

Determinando si existen registros

EN vez de usar el método `count` para determinar si existen registros que coincidan con los límites de tu consulta, puedes usar los métodos `exists` y `doesntExist` :

```
return DB::table('orders')->where('finalized', 1)->exists();  
  
return DB::table('orders')->where('finalized', 1)->doesntExist();
```

php

Selects

Especificando una cláusula select

No siempre deseas seleccionar todas las columnas de una tabla de la base de datos. Usando el método `select` , puedes especificar una cláusula `select` personalizada para la consulta:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

php

El método `distinct` te permite forzar la consulta para que devuelva solamente resultados que sean distintos:

```
$users = DB::table('users')->distinct()->get();
```

php

Si ya tienes una instancia del constructor de consultas y deseas añadir una columna a su cláusula

`select` existente, puedes usar el método `addSelect` :

```
$query = DB::table('users')->select('name');
```

php

```
$users = $query->addSelect('age')->get();
```

Expresiones sin procesar (raw)

Algunas veces puedes necesitar usar una expresión sin procesar en una consulta. Para crear una expresión sin procesar, puedes usar el método `DB::raw` :

```
$users = DB::table('users')
    ->select(DB::raw('count(*) as user_count, status'))
    ->where('status', '<>', 1)
    ->groupBy('status')
    ->get();
```

php

Nota

Las instrucciones sin procesar serán inyectadas dentro de la consulta como cadenas, así que deberías ser extremadamente cuidadoso para no crear vulnerabilidades de inyección SQL.

Métodos Raw

En lugar de usar `DB::raw`, también puedes usar los siguientes métodos para insertar una expresión sin procesar dentro de varias partes de tu consulta.

`selectRaw`

El método `selectRaw` puede ser usado en lugar de `addSelect(DB::raw(...))`. Este método acepta un arreglo opcional de enlaces como su segundo argumento:

```
$orders = DB::table('orders')
    ->selectRaw('price * ? as price_with_tax', [1.0825])
    ->get();
```

php

whereRaw / orWhereRaw

Los métodos `whereRaw` y `orWhereRaw` pueden ser usados para injectar una cláusula `where` sin procesar dentro de tu consulta. Estos métodos aceptan un arreglo opcional de enlaces como segundo argumento:

```
$orders = DB::table('orders')
    ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
    ->get();
```

php

havingRaw / orHavingRaw

Los métodos `havingRaw` y `orHavingRaw` pueden ser usados para establecer una cadena sin procesar 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();
```

php

orderByRaw

El método `orderByRaw` puede ser usado para establecer una cadena sin procesar como el valor de la cláusula `order by`:

```
$orders = DB::table('orders')
    ->orderByRaw('updated_at - created_at DESC')
    ->get();
```

php

Joins

Cláusula inner join

El constructor de consultas también puede ser usado para escribir instrucciones joins. Para ejecutar un "inner join" básico, 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 necesitas juntar, mientras que los argumentos restantes especifican las restricciones de columna para el join. Ciertamente, como puedes ver, puedes hacer un join de 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();
```

php

Cláusula left join / right join

Si prefieres ejecutar un "left join" o un "right join" en vez de un "inner join", usa los métodos `leftJoin` o `rightJoin`. Estos métodos tienen la misma forma de uso de los argumentos que el método `join`:

```
$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();

$users = DB::table('users')
    ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

php

Cláusula cross join

Para ejecutar un cláusula "cross join" usa el método `crossJoin` con el nombre de la tabla a la que deseas hacerle un cross join. Los cross join generan un producto cartesiano entre la primera tabla y la tabla juntada:

```
$users = DB::table('sizes')
    ->crossJoin('colours')
    ->get();
```

php

Cláusulas de join avanzadas

También puedes especificar cláusulas join más avanzadas. Para empezar, pasa una función `closure` como el segundo argumento dentro del método `join`. La `Closure` recibirá un objeto `JoinClause` el cual permitirá que especifiques restricciones en la cláusula `join`:

```
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
    })
    ->get();
```

php

Si prefieres usar una cláusula estilo "where" en tus joins, puedes usar los métodos `where` y `orWhere` en un join. En lugar 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();
```

php

Subconsultas joins

Puedes utilizar los métodos `joinSub`, `leftJoinSub` y `rightJoinSub` para unir una consulta a una subconsulta. Cada uno de estos métodos recibe tres argumentos: la subconsulta, su alias de tabla y una Closure que define las columnas relacionadas:

```
$latestPosts = DB::table('posts')
    ->select('user_id', DB::raw('MAX(created_at) as last_post_cr
    ->where('is_published', true)
    ->groupBy('user_id');

$users = DB::table('users')
    ->joinSub($latestPosts, 'latest_posts', function ($join) {
        $join->on('users.id', '=', 'latest_posts.user_id');
    })->get();
```

php

Uniones

El constructor de consultas también proporciona una forma rápida para "unir" dos consultas. Por ejemplo, puedes crear una consulta inicial y usar el método `union` para unirlo con una segunda consulta:

```
$first = DB::table('users')
    ->whereNull('first_name');

$users = DB::table('users')
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

php

TIP

El método `unionAll` también está disponible y tiene la misma forma de uso que `union`.

Cláusulas where

Cláusula where simple

Puedes usar el método `where` en una instancia del constructor de consultas para añadir cláusulas `where` a la consulta. La ejecución más básica de `where` requiere tres argumentos. El primer argumento es el nombre de la columna. El segundo argumento es un operador, el cual 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, aquí está una consulta que verifica que el valor de la columna "votes" sea igual a 100:

```
$users = DB::table('users')->where('votes', '=', 100)->get();
```

php

Por conveniencia, si quieras verificar que una columna sea igual a un valor dado, puedes pasar directamente el valor como el segundo argumento del método `where`.

```
$users = DB::table('users')->where('votes', 100)->get();
```

php

Puedes usar otros operadores cuando estés escribiendo una cláusula `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();
```

php

También puedes pasar un arreglo de condiciones a la función `where` :

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

php

Instrucciones or

Puedes encadenar en conjunto las restricciones `where` así como añadir 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();
```

php

Cláusulas where adicionales

`whereBetween / orWhereBetween`

El método `whereBetween` verifica que un valor de columna esté en un intervalo de valores:

```
$users = DB::table('users')
    ->whereBetween('votes', [1, 100])
    ->get();
```

php

whereNotBetween / orWhereNotBetween

El método `whereNotBetween` verifica que un valor de columna no esté en un intervalo de valores:

```
$users = DB::table('users')
    ->whereNotBetween('votes', [1, 100])
    ->get();
```

php

whereIn / whereNotIn / orWhereIn / orWhereNotIn

El método `whereIn` verifica que un valor de una columna dada esté contenido dentro del arreglo dado:

```
$users = DB::table('users')
    ->whereIn('id', [1, 2, 3])
    ->get();
```

php

El método `whereNotIn` verifica que el valor de una columna dada **no** esté contenido en el arreglo dado:

```
$users = DB::table('users')
    ->whereNotIn('id', [1, 2, 3])
    ->get();
```

php

whereNull / whereNotNull / orWhereNull / orWhereNotNull

El método `whereNull` verifica que el valor de una columna dada sea `NULL`:

```
$users = DB::table('users')
    ->whereNull('updated_at')
    ->get();
```

php

El método `whereNotNull` verifica que el valor dado de una columna no sea `NULL`:

```
$users = DB::table('users')
    ->whereNotNull('updated_at')
    ->get();
```

php

`whereDate / whereMonth / whereDay / whereYear / whereTime`

El método `whereDate` puede ser usado para comparar el valor de una columna contra una fecha:

```
$users = DB::table('users')
    ->whereDate('created_at', '2016-12-31')
    ->get();
```

php

El método `whereMonth` puede ser usado para comparar el valor de una columna contra un mes específico de un año:

```
$users = DB::table('users')
    ->whereMonth('created_at', '12')
    ->get();
```

php

El método `whereDay` puede ser usado para comparar el valor de una columna contra un día específico de un mes:

```
$users = DB::table('users')
    ->whereDay('created_at', '31')
    ->get();
```

php

El método `whereYear` puede ser usado para comparar el valor de una columna contra un año específico:

```
$users = DB::table('users')
    ->whereYear('created_at', '2016')
    ->get();
```

php

El método `whereTime` puede ser usado para comparar el valor de una columna contra una hora específica:

```
$users = DB::table('users')
    ->whereTime('created_at', '=', '11:20')
    ->get();
```

php

whereColumn / orWhereColumn

El método `whereColumn` puede ser usado para verificar que dos columnas son iguales:

```
$users = DB::table('users')
    ->whereColumn('first_name', 'last_name')
    ->get();
```

php

También puedes pasar un operador de comparación al método:

```
$users = DB::table('users')
    ->whereColumn('updated_at', '>', 'created_at')
    ->get();
```

php

Al método `whereColumn` también le puede ser pasado un arreglo de condiciones múltiples. Estas condiciones serán juntadas usando el operador `and`:

```
$users = DB::table('users')
    ->whereColumn([
        ['first_name', '=', 'last_name'],
        ['updated_at', '>', 'created_at'],
    ])->get();
```

php

Agrupando parámetros

Algunas veces puedes necesitar crear cláusulas where más avanzadas como cláusulas "where exists" o grupos de parámetros anidados. El constructor de consultas de Laravel puede manejar éstos también.

Para empezar, vamos a mirar un ejemplo de grupos de restricciones encerrado por llaves:

```
$users = DB::table('users')
    ->where('name', '=', 'John')
    ->where(function ($query) {
        $query->where('votes', '>', 100)
            ->orWhere('title', '=', 'Admin');
    })
    ->get();
```

php

Como puedes ver, al pasar una `Closure` dentro del método `orWhere`, instruyes al constructor de consultas para empezar un grupo de restricción. La `Closure` recibirá una instancia del constructor de consultas la cual puedes usar para establecer las restricciones que deberían estar contenidas dentro del grupo encerrado por llaves. El ejemplo de arriba producirá la siguiente instrucción SQL:

```
select * from users where name = 'John' and (votes > 100 or title = 'Admin')php
```

TIP

Siempre debes agrupar llamadas `orWhere` para evitar comportamiento inesperado cuando se apliquen alcances globales.

Cláusulas where exists

El método `whereExists` permite que escribas cláusulas de SQL `whereExists`. El método `whereExists` acepta un argumento de tipo `Closure`, el cual recibirá una instancia del constructor de consultas permitiendo que definas la consulta que debería ser puesta dentro de la cláusula "exists":

```
$users = DB::table('users')
    ->whereExists(function ($query) {
        $query->select(DB::raw(1))
            ->from('orders')
            ->whereRaw('orders.user_id = users.id');
    })
    ->get();php
```

La consulta anterior producirá el siguiente SQL:

```
select * from users
where exists (
    select 1 from orders where orders.user_id = users.id
)php
```

Cláusulas where JSON

Laravel también soporta consultar tipos de columna JSON en bases de datos que proporcionan soporte para tipos de columna JSON. Actualmente, esto incluye MySQL 5.7, PostgreSQL, SQL Server 2016, y SQLite 3.9.0 (con la extensión JSON1¹). Para consultar una columna JSON, usa el operador `->`:

```
$users = DB::table('users')
    ->where('options->language', 'en')
    ->get();

$users = DB::table('users')
    ->where('preferences->dining->meal', 'salad')
    ->get();
```

php

Puedes usar `whereJsonContains` para consultar arreglos JSON (sin soporte en SQLite):

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', 'en')
    ->get();
```

php

MySQL and PostgreSQL proveen soporte para `whereJsonContains` con múltiples valores:

```
$users = DB::table('users')
    ->whereJsonContains('options->languages', ['en', 'de'])
    ->get();
```

php

Puedes usar `whereJsonLength` para consultar arreglos JSON por su longitud:

```
$users = DB::table('users')
    ->whereJsonLength('options->languages', 0)
    ->get();

$users = DB::table('users')
    ->whereJsonLength('options->languages', '>', 1)
    ->get();
```

php

Ordenamiento, agrupamiento, límite y desplazamiento

orderBy

El método `orderBy` permite que ordenes los resultados de la consulta por una columna dada. El primer argumento para el método `orderBy` debería ser la columna por la cual deseas ordenar, mientras que el segundo argumento controla la dirección del ordenamiento y puede ser `asc` o `desc` :

```
$users = DB::table('users')
    ->orderBy('name', 'desc')
    ->get();
```

php

latest / oldest

Los métodos `latest` y `oldest` te permiten ordenar fácilmente los resultados por fecha. Por defecto, el resultado será ordenado por la columna `created_at`. También, puedes pasar el nombre de la columna por la cual deseas ordenar:

```
$user = DB::table('users')
    ->latest()
    ->first();
```

php

inRandomOrder

El método `inRandomOrder` puede ser usado para ordenar los resultados de la consulta aleatoriamente. Por ejemplo, puedes usar este método para obtener un usuario aleatorio:

```
$randomUser = DB::table('users')
    ->inRandomOrder()
    ->first();
```

php

groupBy / having

Los métodos `groupBy` y `having` pueden ser usados para agrupar los resultados de la consulta. La forma que distingue el uso del método `having` es similar a la que tiene el método `where` :

```
$users = DB::table('users')
    ->groupBy('account_id')
    ->having('account_id', '>', 100)
    ->get();
```

php

Puedes pasar argumentos múltiples al método `groupBy` para agrupar por múltiples columnas:

```
$users = DB::table('users')
    ->groupBy('first_name', 'status')
    ->having('account_id', '>', 100)
    ->get();
```

php

Para instrucciones `having` más avanzadas, echa un vistazo al método `havingRaw`.

skip / take

Para limitar el número de resultados devueltos desde la consulta, o para avanzar un número dado de resultados en la consulta, puedes usar los métodos `skip` y `take`:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

php

Alternativamente, puedes usar los métodos `limit` y `offset`:

```
$users = DB::table('users')
    ->offset(10)
    ->limit(5)
    ->get();
```

php

Cláusulas condicionales

Algunas podrías querer que las cláusulas apliquen solamente a una consulta cuando alguna cosa más se cumple. Por ejemplo, puedes querer que solamente se aplique una instrucción `where` si un valor de entrada dado está presente en la solicitud entrante. Puedes acompañar esto usando el método `when`:

```
$role = $request->input('role');

$users = DB::table('users')
    ->when($role, function ($query) use ($role) {
        return $query->where('role_id', $role);
    })
    ->get();
```

php

El método `when` ejecuta solamente la Closure dada cuando el primer parámetro es `true`. Si el primer parámetro es `false`, la Closure no será ejecutada.

Puedes pasar otra Closure como tercer parámetro del método `when`. Esta Closure se ejecutará si el primer parámetro se evalúa como `false`. Para ilustrar cómo esta característica puede ser usada, la usaremos para configurar el ordenamiento predeterminado de una consulta:

```
$sortBy = null;  
  
$users = DB::table('users')  
    ->when($sortBy, function ($query) use ($sortBy) {  
        return $query->orderBy($sortBy);  
    }, function ($query) {  
        return $query->orderBy('name');  
    })  
    ->get();
```

php

Inserciones

El constructor de consultas también proporciona un método `insert` para insertar registros dentro de la base de datos. El método `insert` acepta un arreglo de nombres de columna y valores:

```
DB::table('users')->insert(  
    ['email' => 'john@example.com', 'votes' => 0]  
)
```

php

Incluso puedes insertar varios registros dentro de la tabla con una sola llamada a `insert` pasando un arreglo de arreglos. Cada arreglo representa una fila a ser insertada dentro de la tabla:

```
DB::table('users')->insert([  
    ['email' => 'taylor@example.com', 'votes' => 0],  
    ['email' => 'dayle@example.com', 'votes' => 0]  
]);
```

php

El método `insertOrIgnore` ignorará los errores de registros duplicados al momento de insertar registros en la base de datos:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'taylor@example.com'],
    ['id' => 2, 'email' => 'dayle@example.com']
]);
```

php

IDs de auto-incremento

Si la tabla tiene un id de auto-incremento, usa el método `insertGetId` para insertar un registro y recibir el ID:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

php

Nota

Cuando estás usando PostgreSQL el método `insertGetId` espera que la columna de auto-incremento sea llamada `id`. Si prefieres obtener el ID con una “secuencia” distinta, puedes pasar el nombre de la columna como segundo parámetro del método `insertGetId`.

Actualizaciones

Además de insertar registros dentro de la base de datos, el constructor de consultas también puede actualizar registros existentes usando el método `update`. El método `update`, como el método `insert`, acepta un arreglo de pares de columna y valor que contienen las columnas a ser actualizadas. Puedes restringir la consulta `update` usando cláusulas `where`:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['votes' => 1]);
```

php

Actualizar o insertar

A veces es posible que deseas actualizar un registro existente en la base de datos o crearlo si no existe un registro coincidente. En este escenario, se puede usar el método `updateOrInsert`. El método

`updateOrInsert` acepta dos argumentos: un arreglo de condiciones para encontrar el registro y un arreglo de columnas y pares de valores que contienen las columnas que se actualizarán.

El método `updateOrInsert` intentará primero buscar un registro de base de datos que coincida con los pares de columna y valor del primer argumento. Si el registro existe, se actualizará con los valores del segundo argumento. Si no se encuentra el registro, se insertará un nuevo registro con los atributos combinados de ambos argumentos:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

php

Actualizando columnas JSON

Cuando estamos actualizando una columna JSON, deberías usar la sintaxis `->` para acceder a la clave apropiada en el objeto JSON. Esta operación es soportada solamente en MySQL 5.7+ y PostgreSQL 9.5+:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['options->enabled' => true]);
```

php

Incremento y decremento

El constructor de consultas también proporciona métodos convenientes para incrementar o decrementar el valor de una columna dada. Esto es un atajo, 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 a modificar. Un segundo argumento puede ser pasado opcionalmente para controlar la cantidad con la cual la columna debería ser incrementada o decrementada:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);
```

php

```
DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

También puedes especificar columnas adicionales para actualizar durante la operación:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

php

Eliminaciones

El constructor de consultas también puede ser usado para eliminar registros de la tabla por medio del método `delete`. Puedes restringir instrucciones `delete` al agregar cláusulas `where` antes de ejecutar el método `delete`:

```
DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();
```

php

Si deseas truncar la tabla completa, lo cual remueve todas las filas y reinicia el ID de auto-incremento a cero, puedes usar el método `truncate`:

```
DB::table('users')->truncate();
```

php

Bloqueo pesimista

El constructor de consultas también incluye algunas funciones que ayudan a que hagas el "bloqueo pesimista" en tus instrucciones `select`. Para ejecutar la instrucción con un "bloqueo compartido", puedes usar el método `sharedLock` en una consulta. Un bloqueo compartido previene las filas seleccionadas para que no sean modificadas hasta que tu transacción se confirme:

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

php

Alternativamente, puedes usar el método `lockForUpdate`. Un bloqueo "para actualización" evita que las filas se modifiquen o que se seleccionen con otro bloqueo compartido:

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

php

Puedes usar los métodos `dd` o `dump` al construir una consulta para vaciar los enlaces de consulta y SQL. El método `dd` mostrará la información de depuración y luego dejará de ejecutar la solicitud. El método `dump` mostrará la información de depuración pero permitirá que la solicitud se siga ejecutando:

```
DB::table('users')->where('votes', '>', 100)->dd();
```

php

```
DB::table('users')->where('votes', '>', 100)->dump();
```

Base de datos: Paginación

- Introducción
- Uso básico
 - Paginando los resultados del constructor de consultas
 - Paginando los resultados de Eloquent
 - Creando un paginador manualmente
- Mostrando los resultados de la paginación
 - Convertiendo los resultados a JSON
- Personalizando la vista de la paginación
- Métodos de instancia del paginador

Introducción

En otros frameworks, la paginación puede ser muy difícil. El paginador de Laravel está integrado con el constructor de consultas y el ORM Eloquent, proporcionando una conveniente y fácil manera de usar

paginación de resultados de forma predeterminada. El HTML generado por el paginador es compatible con el [Framework de CSS Bootstrap](#).

Uso básico

Paginando los resultados del constructor de consultas

Hay varias formas de paginar los elementos. La más simple es usando el método `paginate` en el constructor de consultas o una Consulta de Eloquent. El método `paginate` se encarga automáticamente de la configuración del límite y desplazamiento apropiado de la página actual que está siendo vista por el usuario. Por defecto, la página actual es detectada por el valor del argumento de cadena de consulta `page` en la solicitud HTTP. Este valor es detectado automáticamente por Laravel y también es insertado automáticamente dentro de los enlaces generados por el paginador.

En este ejemplo, el único argumento pasado al método `paginate` es el número de elementos que prefieres que sean mostrados "por página". En este caso, vamos a especificar que nos gustaría mostrar 15 elementos por página:

```
<?php  
  
namespace App\Http\Controllers;  
  
use App\Http\Controllers\Controller;  
use Illuminate\Support\Facades\DB;  
  
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]);  
    }  
}
```

Nota

Actualmente, las operaciones de paginación que usan una instrucción `GroupBy` no pueden ser ejecutados eficientemente por Laravel. Si necesitas usar una cláusula `GroupBy` con un conjunto de resultados paginados, es recomendable que consultes la base de datos y crees un paginador manualmente.

Paginación sencilla

Si necesitas mostrar solamente enlaces "Siguiente" y "Anterior" en tu vista de paginación, puedes usar el método `simplePaginate` para ejecutar una consulta más eficiente. Esto es muy útil para grandes colecciones de datos cuando no necesitas mostrar un enlace para cada número de página al momento de renderizar tu vista.

```
$users = DB::table('users')->simplePaginate(15);
```

php

Paginando resultados de eloquent

También puedes paginar consultas de [Eloquent](#). En este ejemplo, paginaremos el modelo `User` con 15 elementos por página. Como puedes ver, la sintaxis es casi idéntica a la paginación de los resultados del constructor de consultas.

```
$users = App\User::paginate(15);
```

php

Puedes ejecutar `paginate` después de configurar otras restricciones en la consulta, tal como las cláusulas `where`:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

php

También puedes usar el método `simplePaginate` al momento de paginar los modelos de Eloquent.

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

php

Creando un paginador manualmente

Algunas veces puedes desear crear una instancia de paginación manualmente, pasándole un arreglo de elementos. Puedes hacer eso al crear una instancia de la clase `Illuminate\Pagination\Paginator` o `Illuminate\Pagination\LengthAwarePaginator`, dependiendo de tus necesidades.

La clase `Paginator` no necesita conocer el número total de elementos en el conjunto de resultados; sin embargo, debido a esto, la clase no tiene métodos para obtener el índice de la última página. La clase `LengthAwarePaginator` acepta casi los mismos argumentos que la clase `Paginator`; sin embargo, si requiere una cuenta del total del número de elementos en el conjunto de resultados.

En otras palabras, la clase `Paginator` corresponde al método `simplePaginate` en el constructor de consultas y Eloquent, mientras la clase `LengthAwarePaginator` corresponde al método `paginate`.

Nota

Cuando creas manualmente una instancia del paginador, deberías manualmente "recortar en partes" el arreglo de resultados que pasas al paginador. Si estás inseguro de cómo hacer esto, inspecciona la función de PHP `array_slice`.

Mostrando los resultados de la paginación

Cuando ejecutas el método `paginate`, recibirás una instancia de la clase `Illuminate\Pagination\LengthAwarePaginator`. Cuando ejecutas el método `simplePaginate`, recibirás una instancia de la clase `Illuminate\Pagination\Paginator`. Estos objetos proporcionan varios métodos que afectan la presentación del conjunto de resultados. Además de estos métodos helpers, las instancias del paginador son iteradoras, es decir, pueden ser recorridas por un ciclo repetitivo igual que un arreglo. Así, una vez que has obtenido los resultados, puedes mostrar los resultados y renderizar los enlaces de página usando `Blade`:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

php

El método `links` renderizará los enlaces para el resto de las páginas en el conjunto de resultados. Cada uno de estos enlaces ya contendrá la variable de cadena de consulta `page` apropiada. Recuerda, el HTML generado por el método `links` es compatible con el [Framework de CSS Bootstrap](#).

Personalizando la URI del paginador

El método `withPath` permite personalizar la URI usada por el paginador al momento de generar enlaces. Por ejemplo, si quieras que el paginador genere enlaces como

`http://example.com/custom/url?page=N`, deberías pasar `custom/url` al método `withPath`:

```
Route::get('users', function () {  
    $users = App\User::paginate(15);  
  
    $users->withPath('custom/url');  
  
    //  
});
```

Agregando enlaces de paginación

Puedes agregar la cadena de consulta a los enlaces de paginación usando el método `appends`. Por ejemplo, para agregar `sort=votes` a cada enlace de paginación, deberías hacer la siguiente ejecución del método `appends`:

```
{{ $users->appends(['sort' => 'votes'])->links() }}
```

Si deseas agregar un "fragmento con el símbolo numeral `#`" a las URLs del paginador, puedes usar el método `fragment`. Por ejemplo, para agregar `#foo` al final de cada enlace de paginación, haz la siguiente ejecución del método `fragment`:

```
{{ $users->fragment('foo')->links() }}
```

Ajustando la ventana de enlace de paginación

Puedes controlar cuántos enlaces adicionales son mostrados en cada lado de la "ventana" de la URL del paginador. Por defecto, tres enlaces son mostrados en cada lado de los enlaces primarios del paginador. Sin embargo, puedes controlar este número usando el método `onEachSide`:

```
 {{ $users->onEachSide(5)->links() }}
```

php

Convirtiendo resultados a JSON

Las clases resultantes del paginador de Laravel implementan la interfaz

`Illuminate\Contracts\Support\Jsonable` y exponen el método `toJson`, así es muy fácil convertir los resultados de tu paginación a JSON. También puedes convertir una instancia del paginador al devolverlo desde una ruta o acción de controlador:

```
Route::get('users', function () {
    return App\User::paginate();
});
```

php

El JSON devuelto por el paginador incluirá meta información tal como `total`, `current_page`, `last_page` y más. Los objetos de resultados reales estarán disponibles por medio de la clave `data` en el arreglo JSON. Aquí está un ejemplo del JSON creado al regresar una instancia del paginador desde 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
        }
    ]
}
```

php

Personalizando la vista de la paginación

De forma predeterminada, las vistas que son renderizadas para mostrar los enlaces de paginación son compatibles con el framework de CSS Bootstrap. Sin embargo, si no estás usando Bootstrap, eres libre de definir tus propias vistas para renderizar esos enlaces. Al momento de ejecutar el método `links` en una instancia del paginador, pasa el nombre de la vista como primer argumento del método:

```
 {{ $paginator->links('view.name') }}  
  
// Passing data to the view...  
 {{ $paginator->links('view.name', ['foo' => 'bar']) }}  
php
```

Sin embargo, la forma más fácil de personalizar las vistas de paginación es exportándolas a tu directorio `resources/views/vendor` usando el comando `vendor:publish`:

```
php artisan vendor:publish --tag=laravel-pagination  
php
```

Este comando ubicará las vistas dentro del directorio `resources/views/vendor/pagination`. El archivo `default.blade.php` dentro de este directorio corresponde a la vista de paginación predeterminada. Edita este archivo para modificar el HTML de paginación.

Si quieras designar un archivo distinto como la vista de paginación por defecto, se pueden usar los métodos de paginador `defaultView` y `defaultSimpleView` en tu `AppServiceProvider`:

```
use Illuminate\Pagination\Paginator;  
  
public function boot()  
{  
    Paginator::defaultView('view-name');  
  
    Paginator::defaultSimpleView('view-name');  
}  
php
```

Métodos de la instancia paginadora

Cada instancia del paginador proporciona información de paginación adicional por medio de los siguientes métodos:

Método	Descripción
<code>\$results->count()</code>	Obtiene el número de elementos para la página actual.
<code>\$results->currentPage()</code>	Obtiene el número de la página actual.
<code>\$results->firstItem()</code>	Obtiene el número de resultado del primer elemento en los resultados.
<code>\$results->getOptions()</code>	Obtiene las opciones del paginador.
<code>\$results->getUrlRange(\$start, \$end)</code>	Crea un rango de URLs de paginación.
<code>\$results->hasMorePages()</code>	Determina si hay suficientes elementos para dividir en varias páginas.
<code>\$results->items()</code>	Obtener los elementos de la página actual.
<code>\$results->lastItem()</code>	Obtiene el número de resultado del último elemento en los resultados.
<code>\$results->lastPage()</code>	Obtiene el número de página de la última página disponible. (No disponible cuando se utiliza <code>simplePaginate</code>).
<code>\$results->nextPageUrl()</code>	Obtiene la URL para la próxima página.
<code>\$results->onFirstPage()</code>	Determine si el paginador está en la primera página.
<code>\$results->perPage()</code>	El número de elementos a mostrar por página.

Método	Descripción
<code>\$results->previousPageUrl()</code>	Obtiene la URL de la página anterior.
<code>\$results->total()</code>	Determine el número total de elementos coincidentes en el almacén de datos. (No disponible cuando se utiliza <code>simplePaginate</code>).
<code>\$results->url(\$page)</code>	Obtiene la URL para un número de página dado.
<code>\$results->getPageName()</code>	Obtiene la variable string usada para almacenar la página.
<code>\$results->setPageName(\$name)</code>	Determina la variable string usada para almacenar la página.

Base de datos: Migraciones

- Introducción
- Generando migraciones
- Estructura de migración
- Ejecutando migraciones
 - Revertir migraciones
- Tablas
 - Creando tablas
 - Renombrando / Eliminando tablas
- Columnas
 - Creando columnas
 - Modificadores de columna

- Modificando columnas
- Eliminando columnas
- Índices
 - Creación de indices
 - Renombrando indices
 - Eliminando indices
 - Restricciones de clave foránea

Introducción

Las migraciones son como el control de versión para tu base de datos, permiten que tu equipo modifique y comparta fácilmente el esquema de base de datos de la aplicación. Las migraciones son emparejadas típicamente con el constructor de esquema de Laravel para construir fácilmente el esquema de base de datos de tu aplicación. Si inclusive has tenido que decirle a un miembro de equipo que agregue una columna manualmente a sus esquemas de bases de datos local, has encarado el problema que solucionan las migraciones de base de datos.

La clase facade `Schema` de Laravel proporciona soporte de base de datos orientado a la programación orientada a objetos para la creación y manipulación de tablas a través de todos los sistemas de bases de datos soportados por Laravel.

Generando migraciones

Para crear una migración, usa el comando Artisan `make:migration` :

```
php artisan make:migration create_users_table
```

php

La nueva migración estará ubicada en tu directorio `database/migrations` . Cada nombre de archivo de migración contiene una marca de tiempo la cual permite que Laravel determine el orden de las migraciones.

Las opciones `--table` y `--create` también pueden ser usadas para indicar el nombre de la tabla y si la migración estará creando una nueva tabla. Estas opciones rellenan previamente el archivo stub de migración generado con la tabla especificada:

```
php artisan make:migration create_users_table --create=users
```

php

```
php artisan make:migration add_votes_to_users_table --table=users
```

Si prefieres especificar una ruta de directorio de salida personalizada para la migración generada, puedes usar la opción `--path` al momento de ejecutar el comando `make:migration`. La ruta de directorio dada debe ser relativa a la ruta de directorio base de tu aplicación.

Estructura de migración

Una clase de migración contiene dos métodos: `up` y `down`. El método `up` es usado para agregar nuevas tablas, columnas, o índices para tu base de datos, mientras el método `down` debería revertir las operaciones ejecutadas por el método `up`.

Dentro de ambos métodos puedes usar el constructor de esquema de Laravel para crear y modificar expresivamente las tablas. Para aprender sobre todos los métodos disponibles en el constructor

[Schema](#), [inspecciona su documentación](#). Por ejemplo, este ejemplo de migración crea una tabla `flights`:

```
<?php
```

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->bigIncrements('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('flights');
    }
}
```

```
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Ejecutando migraciones

Para ejecutar todas tus maravillosas migraciones, ejecuta el comando Artisan `migrate` :

```
php artisan migrate
```

php

Nota

Si estás usando La máquina virtual de Homestead, deberías ejecutar este comando desde dentro de tu máquina virtual.

Forzando las migraciones para ejecutar en producción

Algunas operaciones de migración son destructivas, lo que significa que pueden causar que pierdas tus datos. Con el propósito de protegerte de ejecutar estos comandos contra tu base de datos de producción, recibirás un mensaje de confirmación antes que los comandos sean ejecutados. Para forzar que los comandos se ejecuten sin retardo, usa el indicador `--force`.

```
php artisan migrate --force
```

php

Revertir migraciones

Para revertir la operación de migración más reciente, puedes usar el comando `rollback`. Este comando reversa el último "lote" de migraciones, los cuales pueden incluir archivos de migración múltiples.

```
php artisan migrate:rollback
```

php

Puedes revertir un número limitado de migraciones proporcionando la opción `step` al comando `rollback`. Por ejemplo, el siguiente comando revertirá los cinco "lotes" de migraciones más recientes:

```
php artisan migrate:rollback --step=5
```

php

El comando `migrate:reset` revertirá todas las migraciones de tu aplicación:

```
php artisan migrate:reset
```

php

rollback & migrate en un único comando

El comando `migrate:refresh` revertirá todas tus migraciones y después ejecutará el comando `migrate`. Este comando vuelve a crear efectivamente tu base de datos entera:

```
php artisan migrate:refresh
```

php

```
// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

Puedes revertir y volver a migrar un número limitado de migraciones proporcionando la opción `step` al comando `refresh`. Por ejemplo, el siguiente comando revertirá y volverá a migrar las cinco migraciones más recientes:

```
php artisan migrate:refresh --step=5
```

php

Eliminando todas las tablas y migrar

El comando `migrate:fresh` eliminará todas las tablas de la base de datos y después ejecutará el comando `migrate`:

```
php artisan migrate:fresh  
  
php artisan migrate:fresh --seed
```

php

Tablas

Creando tablas

Para crear una nueva tabla en la base de datos, usa el método `create` en la clase facade `Schema`. El método `create` acepta dos argumentos. El primero es el nombre de la tabla, mientras que el segundo es una `Closure` la cual recibe un objeto de la clase `Blueprint` que puede ser usado para definir la nueva tabla:

```
Schema::create('users', function (Blueprint $table) {  
    $table->bigIncrements('id');  
});
```

php

Al momento de crear la tabla, puedes usar cualquiera de [los métodos de columna](#) del constructor de esquemas para definir las columnas de la tabla.

Inspeccionando la tabla / Existencia de columna

Puedes inspeccionar fácilmente la existencia de una tabla o columna usando los métodos `hasTable` y `hasColumn`:

```
if (Schema::hasTable('users')) {  
    //  
}  
  
if (Schema::hasColumn('users', 'email')) {  
    //  
}
```

php

Conexión de base de datos & Opciones de tabla

Si quieres ejecutar una operación de esquema en una conexión de base de datos que no es tu conexión predeterminada, usa el método `connection`:

```
Schema::connection('foo')->create('users', function (Blueprint $table) {  
    $table->bigIncrements('id');  
});
```

php

Puedes usar los siguientes comandos en el constructor de esquema para definir las opciones de tabla:

Comando	Descripción
<code>\$table->engine = 'InnoDB';</code>	Especifica el motor de almacenamiento de la tabla. (Sólo en MySQL).
<code>\$table->charset = 'utf8';</code>	Especifica un conjunto de caracteres. (Sólo en MySQL).
<code>\$table->collation = 'utf8_unicode_ci';</code>	Especifica un orden predeterminado para la tabla. (Sólo en MySQL)
<code>\$table->temporary();</code>	Crea una tabla temporal (excepto en SQL Server).

Renombrando / Eliminando tablas

Para renombrar una tabla de base de datos existente, usa el método `rename` :

```
Schema::rename($from, $to);
```

php

Para eliminar una tabla existente, puedes usar los métodos `drop` o `dropIfExists` :

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

php

```
Schema::dropIfExists('users');
```

Renombrando tablas con claves foráneas

Antes de renombrar una tabla, deberías verificar que cualquiera de las restricciones de clave foránea en la tabla tenga un nombre explícito en tus archivos de migración en caso de permitir que Laravel asigne un nombre basado en la convención. De otra manera, el nombre de restricción de clave foránea se referirá al nombre que tenía la tabla.

Columnas

Creando columnas

El método `table` en la clase facade `Schema` puede ser usado para actualizar tablas existentes. Igual que el método `create` acepta dos argumentos: el nombre de la tabla y una `Closure` que recibe una instancia de la clase `Blueprint` que puedes usar para agregar columnas a la tabla:

```
Schema::table('users', function (Blueprint $table) {  
    $table->string('email');  
});
```

php

Tipos de columna permitidos

El constructor de esquema contiene una variedad de tipos de columna que puedes especificar al momento de construir tus tablas:

Comando	Descripción
<code>\$table->bigIncrements('id');</code>	Tipo de columna equivalente a Auto-incremento UNSIGNED BIGINT (clave primaria).
<code>\$table->bigInteger('votes');</code>	Tipo de columna equivalente a BIGINT equivalent.
<code>\$table->binary('data');</code>	Tipo de columna equivalente a BLOB.
<code>\$table->boolean('confirmed');</code>	Tipo de columna equivalente a BOOLEAN.
<code>\$table->char('name', 100);</code>	Tipo de columna equivalente a CHAR con una longitud.
<code>\$table->date('created_at');</code>	Tipo de columna equivalente a DATE.
<code>\$table->dateTime('created_at', 0);</code>	Tipo de columna equivalente a DATETIME con precisión (el total de dígitos).

Comando	Descripción
\$table->dateTimeTz('created_at', 0);	Tipo de columna equivalente a DATETIME (con hora de la zona) y con una precisión (el total de dígitos).
\$table->decimal('amount', 8, 2);	Tipo de columna equivalente a DECIMAL con una precisión (el total de dígitos) y escala de dígitos decimales.
\$table->double('amount', 8, 2);	Tipo de columna equivalente a DOUBLE con una precisión (el total de dígitos) y escala de dígitos decimales.
\$table->enum('level', ['easy', 'hard']);	Tipo de columna equivalente a ENUM.
\$table->float('amount', 8, 2);	Tipo de columna equivalente a FLOAT con una precisión (el total de dígitos) y escala de dígitos decimales.
\$table->geometry('positions');	Tipo de columna equivalente a GEOMETRY.
\$table->geometryCollection('positions');	Tipo de columna equivalente a GEOMETRYCOLLECTION.
\$table->increments('id');	Tipo de columna equivalente a Auto-incremento UNSIGNED INTEGER (clave primaria).
\$table->integer('votes');	Tipo de columna equivalente a INTEGER.
\$table->ipAddress('visitor');	Tipo de columna equivalente a dirección IP.
\$table->json('options');	Tipo de columna equivalente a JSON.
\$table->jsonb('options');	Tipo de columna equivalente a JSONB.
\$table->lineString('positions');	Tipo de columna equivalente a LINESTRING.

Comando	Descripción
<code>\$table->longText('description');</code>	Tipo de columna equivalente a LONGTEXT.
<code>\$table->macAddress('device');</code>	Tipo de columna equivalente a dirección MAC.
<code>\$table->mediumIncrements('id');</code>	Tipo de columna equivalente a Auto-incremento UNSIGNED MEDIUMINT (clave primaria).
<code>\$table->mediumInteger('votes');</code>	Tipo de columna equivalente a MEDIUMINT.
<code>\$table->mediumText('description');</code>	Tipo de columna equivalente a MEDIUMTEXT.
<code>\$table->morphs('taggable');</code>	Agrega los tipos de columna equivalente a UNSIGNED INTEGER <code>taggable_id</code> y VARCHAR <code>taggable_type</code> .
<code>\$table->uuidMorphs('taggable');</code>	Agrega las columnas UUID equivalentes <code>taggable_id</code> CHAR(36) y <code>taggable_type</code> VARCHAR(255).
<code>\$table->multiLineString('positions');</code>	Tipo de columna equivalente a MULTILINESTRING.
<code>\$table->multiPoint('positions');</code>	Tipo de columna equivalente a MULTIPOINT.
<code>\$table->multiPolygon('positions');</code>	Tipo de columna equivalente a MULTIPOLYGON.
<code>\$table->nullableMorphs('taggable');</code>	Permite que la columna <code>morphs()</code> acepte una versión de valor nulo.
<code>\$table->nullableUuidMorphs('taggable');</code>	Agrega versiones nullable de las columnas <code>uuidMorphs()</code> .
<code>\$table->nullableTimestamps();</code>	Método Alias de <code>timestamps()</code> .
<code>\$table->point('position');</code>	Tipo de columna equivalente a POINT.

Comando	Descripción
\$table->polygon('positions');	Tipo de columna equivalente a POLYGON.
\$table->rememberToken();	Permite nulos en el tipo de columna equivalente a VARCHAR(100) <code>remember_token</code> .
\$table->set('flavors', ['strawberry', 'vanilla']);	Establece una columna equivalente.
\$table->smallIncrements('id');	Tipo de columna equivalente a Auto-incremento UNSIGNED SMALLINT (clave primaria).
\$table->smallInteger('votes');	Tipo de columna equivalente a SMALLINT.
\$table->softDeletes(0);	Permite nulos en el tipo de columna equivalente a TIMESTAMP <code>deleted_at</code> para eliminaciones.
\$table->softDeletesTz();	Permite nulos en el tipo de columna equivalente a TIMESTAMP <code>deleted_at</code> (con la hora de la zona) para eliminaciones.
\$table->string('name', 100);	Tipo de columna equivalente a VARCHAR con una longitud opcional.
\$table->text('description');	Tipo de columna equivalente a TEXT.
\$table->time('sunrise');	Tipo de columna equivalente a TIME.
\$table->timeTz('sunrise');	Tipo de columna equivalente a TIME (con la hora de la zona).
\$table->timestamp('added_on');	Tipo de columna equivalente a TIMESTAMP.
\$table->timestampTz('added_on');	Tipo de columna equivalente a TIMESTAMP (con la hora de la zona).
\$table->timestamps();	Permite nulos en las columnas equivalentes TIMESTAMP <code>created_at</code> y <code>updated_at</code> .

Comando	Descripción
<code>\$table->timestampsTz();</code>	Permite nulos en las columnas equivalentes TIMESTAMP <code>created_at</code> y <code>updated_at</code> (con la hora de la zona).
<code>\$table->tinyIncrements('id');</code>	Tipo de columna equivalente a Auto-incremento UNSIGNED TINYINT (clave primaria).
<code>\$table->tinyInteger('votes');</code>	Tipo de columna equivalente a TINYINT.
<code>\$table->unsignedBigInteger('votes');</code>	Tipo de columna equivalente a UNSIGNED BIGINT.
<code>\$table->unsignedDecimal('amount', 8, 2);</code>	Tipo de columna equivalente a UNSIGNED DECIMAL con una precisión (total de dígitos) escala (dígitos decimales).
<code>\$table->unsignedInteger('votes');</code>	Tipo de columna equivalente a UNSIGNED INTEGER.
<code>\$table->unsignedMediumInteger('votes');</code>	Tipo de columna equivalente a UNSIGNED MEDIUMINT.
<code>\$table->unsignedSmallInteger('votes');</code>	Tipo de columna equivalente a UNSIGNED SMALLINT.
<code>\$table->unsignedTinyInteger('votes');</code>	Tipo de columna equivalente a UNSIGNED TINYINT.
<code>\$table->uuid('id');</code>	Tipo de columna equivalente a UUID.
<code>\$table->year('birth_year');</code>	Tipo de columna equivalente a YEAR.

Modificadores de columna

Además de los tipos de columna listados anteriormente, hay varios "modificadores" de columna que puedes usar al momento de agregar una columna a la tabla de base de datos. Por ejemplo, para hacer que la columna "acepte valores nulos", puedes usar el método `nullable`.

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

php

Debajo está una lista con todos los modificadores de columna disponibles. Esta lista no incluye los modificadores de índice:

Modificador	Descripción
->after('column')	Coloca la columna "después de" otra columna (MySQL)
->autoIncrement()	Establece las columnas tipo INTEGER como auto-incremento (clave primaria)
->charset('utf8')	Especifica un conjunto de caracteres para la columna (MySQL)
->collation('utf8_unicode_ci')	Especifica un ordenamiento para la columna (MySQL/PostgreSQL/SQL Server)
->comment('my comment')	Agrega un comentario a una columna (MySQL/PostgreSQL)
->default(\$value)	Especifica un valor "predeterminado" para la columna
->first()	Coloca la columna al "principio" en la tabla (MySQL)
->nullable(\$value = true)	Permite que valores NULL (por defecto) sean insertados dentro de la columna
->storedAs(\$expression)	Crea una columna almacenada generada por la expresión (MySQL)
->unsigned()	Establece las columnas tipo INTEGER como UNSIGNED (MySQL)
->useCurrent()	Establece las columnas tipo TIMESTAMP para usar CURRENT_TIMESTAMP como valor predeterminado

Modificador	Descripción
<code>->virtualAs(\$expression)</code>	Crea una columna virtual generada por la expresión (MySQL)
<code>->generatedAs(\$expression)</code>	Crea una columna de identidad con opciones de secuencia especificadas (PostgreSQL)
<code>->always()</code>	Define la prioridad de los valores de secuencia sobre la entrada para una columna de identidad (PostgreSQL)

Expresiones por defecto

El modificador `default` acepta un valor o una instancia

`\Illuminate\Database\Query\Expression`. Usar una instancia `Expression` evitará envolver el valor en comillas y te permitirá usar funciones específicas de la base de datos. Una situación donde esto es particularmente útil es al momento de asignar valores por defecto a columnas JSON:

```
<?php                                         php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->json('movies')->default(new Expression('JSON_ARRAY()'));
            $table->timestamps();
        });
    }
}
```

Nota

El soporte para expresiones por defecto depende del driver de tu base de datos, la versión de la base de datos y el tipo de campo. Por favor refiérete a la documentación apropiada por compatibilidad. También ten en cuenta que usar funciones específicas de la base de datos puede vincularte estrechamente a un driver específico.

Modificando columnas

Prerequisitos

Antes de modificar una columna, asegúrate de agregar la dependencia `doctrine/dbal` a tu archivo `composer.json`. La biblioteca DBAL de Doctrine es usada para determinar el estado actual de la columna y crear las consultas SQL necesarias para hacer los ajustes especificados a la columna:

```
composer require doctrine/dbal
```

php

Actualizando los atributos de columna

El método `change` permite que modifiques algunos tipos de columna existentes a un nuevo tipo o modifiques los atributos de la columna. Por ejemplo, puedes querer aumentar el tamaño de una columna tipo cadena. Para ver el método `change` en acción, vamos a aumentar el tamaño de la columna `name` de 25 a 50 caracteres:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

php

También podríamos modificar una columna para que acepte valores nulos:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 50)->nullable()->change();
});
```

php

Nota

Solamente los siguientes tipos de columna pueden ser "cambiados": bigInteger, binary, boolean, date, dateTime, dateTimeTz, decimal, integer, json, longText, mediumText, smallInteger, string, text, time, unsignedBigInteger, unsignedInteger y unsignedSmallInteger.

Renombrando columnas

Para renombrar una columna, puedes usar el método `renameColumn` en el constructor de esquemas. Antes de renombrar una columna, asegúrate de agregar la dependencia `doctrine/dbal` a tu archivo `composer.json`:

```
Schema::table('users', function (Blueprint $table) {  
    $table->renameColumn('from', 'to');  
});
```

php

Nota

Renombrar alguna columna en una tabla que también tiene una columna de tipo `enum` no es soportado actualmente.

Eliminando columnas

Para eliminar una columna, usa el método `dropColumn` en el constructor de esquemas. Antes de eliminar columnas de una base de datos SQLite, necesitarás agregar la dependencia `doctrine/dbal` a tu archivo `composer.json` y ejecutar el comando `composer update` en tu terminal para instalar la biblioteca:

```
Schema::table('users', function (Blueprint $table) {  
    $table->dropColumn('votes');  
});
```

php

Puedes eliminar múltiples columnas de una tabla al pasar un arreglo de nombres de columna al método `dropColumn`:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

php

Nota

Eliminar o modificar múltiples columnas dentro de una sola migración al momento de usar una base de datos SQLite no está soportado.

Alias de comandos disponibles

Comando	Descripción
<code>\$table->dropMorphs('morphable');</code>	Elimina las columnas <code>morphable_id</code> y <code>morphable_type</code> .
<code>\$table->dropRememberToken();</code>	Eliminar la columna <code>remember_token</code> .
<code>\$table->dropSoftDeletes();</code>	Eliminar la columna <code>deleted_at</code> .
<code>\$table->dropSoftDeletesTz();</code>	Alias del método <code>dropSoftDeletes()</code> .
<code>\$table->dropTimestamps();</code>	Eliminar las columnas <code>created_at</code> y <code>updated_at</code> .
<code>\$table->dropTimestampsTz();</code>	Alias del método <code>dropTimestamps()</code> .

Indices

Creando índices

El constructor de esquemas soporta varios tipos de índices. Primero, veamos un ejemplo que especifica que los valores de una columna deben ser únicos. Para crear el índice, podemos encadenar el método `unique` en la definición de columna:

```
$table->string('email')->unique();
```

php

Alternativamente, puedes crear el índice después de la definición de la columna. Por ejemplo:

```
$table->unique('email');
```

php

Incluso puedes pasar un arreglo de columnas a un método de índice para crear un índice compuesto (o combinado) :

```
$table->index(['account_id', 'created_at']);
```

php

Laravel generará automáticamente un nombre de índice razonable, pero puedes pasar un segundo argumento al método para especificar el nombre por ti mismo.

```
$table->unique('email', 'unique_email');
```

php

Tipos de indice disponibles

Cada método de índice acepta un segundo argumento opcional para especificar el nombre del índice. Si se omite, el nombre se derivará de los nombres de la tabla y la(s) columna(s).

Comando	Descripción
<code>\$table->primary('id');</code>	Agrega una clave primaria.
<code>\$table->primary(['id', 'parent_id']);</code>	Agrega claves compuestas.
<code>\$table->unique('email');</code>	Agrega un índice único.
<code>\$table->index('state');</code>	Agrega un índice con valores repetidos.
<code>\$table->spatialIndex('location');</code>	Agrega un índice espacial. (excepto SQLite)

Longitudes de indices & MySQL / MariaDB

Laravel usa el conjunto de caracteres `utf8mb4` por defecto, el cual incluye soporte para almacenar "emojis" en la base de datos. Si estás ejecutando una versión de MySQL más antigua que la versión 5.7.7 o más vieja que la versión 10.2.2 de MariaDB, puedes que necesites configurar manualmente la longitud de cadena predeterminada generada por las migraciones con el propósito de que MySQL cree los índices

para estos. Puedes configurar esto ejecutando el método `Schema::defaultStringLength()` dentro de tu `AppServiceProvider`:

```
use Illuminate\Support\Facades\Schema;  
  
/**  
 * Bootstrap any application services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    Schema::defaultStringLength(191);  
}
```

php

Alternativamente, puedes habilitar la opción `innodb_large_prefix` para tu base de datos. Debes referirte a la documentación de tu base de datos para conocer las instrucciones de cómo habilitar ésta apropiadamente.

Renombrando índices

Para renombrar un índice, puedes usar el método `renameIndex()`. Este método acepta el nombre del índice actual como primer argumento y el nombre deseado como segundo argumento:

```
$table->renameIndex('from', 'to')
```

php

Eliminando índices

Para eliminar un índice, debes especificar el nombre del índice. De forma predeterminada, Laravel asigna automáticamente un nombre razonable para los índices. Concatena el nombre de la tabla, el nombre de la columna indexada y el tipo de índice. Aquí están algunos ejemplos:

Comando	Descripción
<pre>\$table->dropPrimary('users_id_primary');</pre>	Eliminar una clave primaria de la tabla "users".

Comando	Descripción
<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".
<code>\$table->dropSpatialIndex('geo_location_spatialindex');</code>	Elimina un índice espacial de la tabla "geo" (excepto SQLite).

Si pasas un arreglo de columnas dentro de un método que elimina los índices, el nombre de índice convencional será generado basado en el nombre de la tabla, columnas y tipo de clave:

```
Schema::table('geo', function (Blueprint $table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

php

Restricciones de clave foránea

Laravel también proporciona soporte para la creación de restricciones de clave foránea, las cuales son usadas para forzar la integridad referencial a nivel de base de datos. Por ejemplo, vamos a definir una columna `user_id` en la tabla `posts` que referencia la columna `id` en una tabla `users` :

```
Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

php

También puedes especificar la acción deseada para las propiedades "on delete" y "on update" de la restricción:

```
$table->foreign('user_id')
    ->references('id')->on('users')
    ->onDelete('cascade');
```

php

Para eliminar una clave foránea, puedes usar el método `dropForeign`. Las restricciones de clave foránea usan la misma convención de nombres que los índices. Así, concatenaremos el nombre de la tabla y el de columna en la restricción luego agrega el sufijo ”_foreign” al nombre:

```
$table->dropForeign('posts_user_id_foreign');
```

php

O, puedes pasar un arreglo de valores el cual usará automáticamente el nombre de restricción convencional al momento de eliminar:

```
$table->dropForeign(['user_id']);
```

php

Puedes habilitar o deshabilitar las restricciones de clave foránea dentro de tus migraciones usando los siguientes métodos:

```
Schema::enableForeignKeyConstraints();
```

php

```
Schema::disableForeignKeyConstraints();
```

Nota

SQLite deshabilita las restricciones de clave foránea de forma predeterminada. Al usar SQLite, asegúrese de habilitar el soporte de clave foránea en la configuración de tu base de datos antes de intentar crearlos en sus migraciones.

Base de datos: Seeding

- [Introducción](#)

- [Escribiendo seeders](#)
 - [Usando model factories](#)
 - [Registrando seeders adicionales](#)
- [Ejecutando seeders](#)

Introducción

Laravel incluye un método sencillo para alimentar tu base de datos con datos de prueba usando clases `Seeder`. Todas las clases `Seeder` son almacenadas en el directorio `database/seeds`. Las clases `Seeder` pueden tener cualquier nombre que deseas, pero deberías seguir probablemente alguna convención razonable, tales como `UsersTableSeeder` etc. De forma predeterminada, una clase `DatabaseSeeder` se define para tí. A partir de esta clase, puedes usar el método `call` para registrar otras clases seeder, permitiendo que controles el orden en que se ejecutan.

Escribiendo seeders

Para generar un seeder, ejecuta el Comando Artisan `make:seeder`. Todos los seeders generados por el framework serán colocados en el directorio `database/seeds`:

```
php artisan make:seeder UsersTableSeeder
```

php

Una clase seeder contiene solamente un método de forma predeterminada: `run`. Este método es ejecutado cuando el Comando Artisan `db:seed` se ejecuta. Dentro del método `run`, puedes insertar datos en tu base de datos en la forma que deseas. Puedes usar el constructor de consultas para insertar datos manualmente o puedes usar los Model Factories de Eloquent.

TIP

La protección de asignación en masa es deshabilitada automáticamente durante el seeding de la base de datos.

Como un ejemplo, vamos a modificar la clase `DatabaseSeeder` predeterminada y agregar una instrucción `insert` al método `run`:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

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' => Hash::make('password'),
        ]);
    }
}
```

TIP

Puede escribir cualquier dependencia que necesite dentro de la firma del método `run`. Se resolverán automáticamente a través del [contenedor de servicio](#) de Laravel.

Usando model factories

Ciertamente, especificar manualmente los atributos para cada seeder de modelos es lento y complicado. En lugar de eso, puedes usar [Model Factories](#) para generar convenientemente cantidades grandes de registros de bases de datos. Primero, revisa la [documentación sobre model factories](#) para aprender cómo definir tus factories. Una vez que hayas definido tus factories, puedes usar la función helper `factory` para insertar registros dentro de tu base de datos.

Por ejemplo, vamos a crear 50 usuarios y establecer una asociación con los posts para cada usuario:

```
/***
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    factory(App\User::class, 50)->create()->each(function ($user) {
        $user->posts()->save(factory(App\Post::class)->make());
    });
}
```

php

Registrando seeders adicionales

Dentro de la clase `DatabaseSeeder`, puedes usar el método `call` para ejecutar clases seeder adicionales. Usar el método `call` te permite separar el seeding de tu base de datos en varios archivos con el propósito de que no exista una clase seeder única que se vuelva extremadamente grande. Pasa el nombre de la clase seeder que deseas ejecutar:

```
/***
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    $this->call([
        UsersTableSeeder::class,
        PostsTableSeeder::class,
        CommentsTableSeeder::class,
    ]);
}
```

php

Ejecutando seeders

Una vez que hayas escrito tu seeder, puedes necesitar regenerar el cargador automático de Composer usando el comando `dump-autoload`:

```
composer dump-autoload
```

php

Ahora puedes usar el comando Artisan `db:seed` para alimentar tu base de datos. De forma predeterminada, el comando `db:seed` ejecuta la clase `DatabaseSeeder`, la cual puede ser usada para ejecutar otras clases seeder. Sin embargo, puedes usar la opción `--class` para especificar que una clase seeder específica se ejecute individualmente:

```
php artisan db:seed
```

php

```
php artisan db:seed --class=UsersTableSeeder
```

También puedes alimentar tu base de datos usando el comando `migrate:fresh`, el cual eliminará todas las tablas y volverá a ejecutar todas tus migraciones. Este comando es útil para reconstruir tu base de datos completamente:

```
php artisan migrate:fresh --seed
```

php

Forzar la ejecución de seeders en producción

Algunas operaciones de seeding pueden causar que alteres o pierdas datos. Para protegerte de ejecutar comandos de seeding en tu base de datos de producción, te será solicitada una confirmación antes de que los seeders sean ejecutados. Para forzar la ejecución de los seeders sin confirmación, usa la opción

```
--force :
```

```
php artisan db:seed --force
```

php

Redis

- Introducción
 - Configuración
 - Predis
 - PhpRedis
- Interactuar con redis
 - Canalizar comandos
- Pub / Sub

Introducción

Redis es un almacenamiento avanzado de pares clave-valor y de código abierto. A menudo se le denomina como un servidor de estructura de datos ya que los pares pueden contener [cadenas, hashes, listas, sets y sets ordenados](#).

Antes de utilizar Redis con Laravel, te recomendamos que instales y uses la extensión de PHP [PhpRedis](#) mediante PECL. La extensión es mas difícil de instalar pero contribuirá a un mejor rendimiento en aplicaciones que hacen un uso intensivo de Redis.

Alternativamente, puedes instalar el paquete [predis/predis](#) mediante Composer:

```
composer require predis/predis
```

php

NOTA

El mantenimiento de Predis se ha abandonado por su autor original y puede que sea eliminado de Laravel en futuras versiones.

Configuración

La configuración de redis para tu aplicación está ubicada en el archivo de configuración [config/database](#). Dentro de este archivo, podrás ver el arreglo [redis](#) que contiene los servidores de Redis utilizados por tu aplicación:

```
'redis' => [  
    'client' => env('REDIS_CLIENT', 'phpredis'),
```

php

```

'default' => [
    'host' => env('REDIS_HOST', '127.0.0.1'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => env('REDIS_DB', 0),
],
]

'cache' => [
    'host' => env('REDIS_HOST', '127.0.0.1'),
    'password' => env('REDIS_PASSWORD', null),
    'port' => env('REDIS_PORT', 6379),
    'database' => env('REDIS_CACHE_DB', 1),
],
]

```

La configuración del servidor por defecto deberá ser suficiente para el entorno de desarrollo. Sin embargo, puedes modificar este arreglo según tu entorno. Cada servidor de Redis definido en tu configuración debe contener un nombre, host y puerto.

Configuración de clusters

Si tu aplicación está utilizando un cluster de servidores Redis, debes definir este cluster en la clave `clusters` de tu configuración de Redis:

```

'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),

    'clusters' => [
        'default' => [
            [
                'host' => env('REDIS_HOST', 'localhost'),
                'password' => env('REDIS_PASSWORD', null),
                'port' => env('REDIS_PORT', 6379),
                'database' => 0,
            ]
        ],
        'clusters' => [
            'default' => [
                [
                    'host' => env('REDIS_HOST', 'localhost'),

```

```
'password' => env('REDIS_PASSWORD', null),
'port' => env('REDIS_PORT', 6379),
'database' => 0,
],
],
],
],
```

Por defecto, los clusters realizarán la división del lado del cliente en sus nodos, permitiéndote agrupar nodos y crear una gran cantidad de RAM disponible. Sin embargo, ten en cuenta que la división del lado del cliente no gestiona el failover; por lo tanto, es principalmente adecuado para datos en caché que estén disponibles desde otro almacenamiento de datos primario. Si deseas utilizar el agrupamiento nativo de Redis, debes especificarlo en la clave `options` de tu configuración de Redis:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'options' => [
        'cluster' => env('REDIS_CLUSTER', 'redis'),
    ],
    'options' => [
        'cluster' => 'redis',
    ],
    'clusters' => [
        // ...
    ],
],
```

Predis

Para utilizar la extensión Predis, debes de cambiar la variable de entorno `REDIS_CLIENT` de `phpredis` a `predis`:

```
'redis' => [
    'client' => env('REDIS_CLIENT', 'predis'),
```

```
'client' => env('REDIS_CLIENT', 'predis'),
```

```
// Resto de la configuración de Redis...
```

```
],
```

Además de las opciones predeterminadas de la configuración del servidor `host`, `port`, `database` y `password`, Predis admite [parámetros de conexión](#) adicionales que pueden ser definidos para cada uno de tus servidores de Redis. Para utilizar estas opciones de configuración adicionales, agrégalos a la configuración del 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,  
,
```

php

PhpRedis

La extensión PhpRedis esta configurada por defecto en el fichero env como `REDIS_CLIENT` y en tu archivo de configuración `config/database.php`:

```
'redis' => [  
  
    'client' => env('REDIS_CLIENT', 'phpredis'),  
  
    // Resto de la configuración de Redis...  
,
```

php

Si planeas usar la extension junto con el llamado `Redis` Facade, deberias renombrarlo como

`RedisManager` para evitar el conflicto con la clase Redis. Puedes hacerlo en la sección de alias de tu archivo de configuracion `app.php`.

```
'RedisManager' => Illuminate\Support\Facades\Redis::class,
```

php

Además de las opciones predeterminadas de configuración del servidor `host`, `port`, `database` y `password`, PhpRedis admite los siguientes parámetros de conexión adicionales: `persistent`, `prefix`, `read_timeout` y `timeout`. Puedes agregar cualquiera de estas opciones a la configuración del 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_timeout' => 60,
],
```

Facade Redis

Para evitar conflictos de nombramiento de clases con la propia extensión de Redis PHP, necesitarás eliminar o renombrar el alias Facade `Illuminate\Support\Facades\Redis` de la configuración de tu `app` en el apartado o vector `aliases`. Generalmente, deberás eliminar este alias completamente y solo referenciar la Facade por su nombre de clase completo mientras que uses la extensión Redis PHP.

Interactuar con redis

Puedes interactuar con Redis llamando varios métodos en el `facade Redis`. El facade `Redis` admite métodos dinámicos, lo que significa que puedes llamar a cualquier comando de Redis en el facade y el comando será pasado directamente a Redis. En este ejemplo, vamos a llamar al comando `GET` de Redis llamando al método `get` en el facade `Redis`:

```
<?php

namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *

```

```
* @param int $id
* @return Response
*/
public function showProfile($id)
{
    $user = Redis::get('user:profile:'.$id);

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

Como lo mencionamos anteriormente, puedes llamar a cualquier comando de Redis en el facade `Redis`. Laravel utiliza métodos mágicos para pasar los comandos al servidor de Redis, para que pases los argumentos que espera el comando de Redis:

```
Redis::set('name', 'Taylor');

$values = Redis::lrange('names', 5, 10);
```

php

Alternativamente, también puedes pasar comandos al servidor usando el método `command`, el cual acepta el nombre del comando como primer argumento, y un arreglo de valores como segundo argumento:

```
$values = Redis::command('lrange', ['name', 5, 10]);
```

php

Utilizar múltiples conexiones de Redis

Puedes obtener una instancia de Redis llamando al método `Redis::connection`:

```
$redis = Redis::connection();
```

php

Esto te dará una instancia del servidor de Redis predeterminado. También puedes pasar la conexión o nombre del cluster al método `connection` para obtener un servidor o cluster en específico según lo definido en tu configuración de Redis:

```
$redis = Redis::connection('my-connection');
```

php

Canalizar comandos

La canalización debe ser utilizada cuando necesites enviar muchos comandos al servidor. El método `pipeline` acepta un argumento: un `Closure` que reciba una instancia de Redis. Puedes emitir todos tus comandos a esta instancia de Redis y después éstos serán enviados al servidor proporcionando mejor rendimiento:

```
Redis::pipeline(function ($pipe) {  
    for ($i = 0; $i < 1000; $i++) {  
        $pipe->set("key:$i", $i);  
    }  
});
```

Pub / Sub

Laravel proporciona una interfaz conveniente para los comandos `publish` y `subscribe` de Redis. Estos comandos de Redis te permiten escuchar mensajes en un "canal" dado. Puedes publicar mensajes en el canal desde otra aplicación, o incluso utilizando otro lenguaje de programación, lo que permite una comunicación sencilla entre aplicaciones y procesos.

Primero, configuremos un listener para el canal usando el método `subscribe`. Vamos a colocar una llamada a este método en un [comando de Artisan](#) ya que llamar al método `subscribe` comienza un proceso de larga ejecución:

```
<?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;
        });
    }
}

```

Ahora podemos publicar mensajes en el canal usando el método `publish` :

```

Route::get('publish', function () {
    // Route logic...

    Redis::publish('test-channel', json_encode(['foo' => 'bar']));
});

```

php

Suscripciones de comodines

Usando el método `psubscribe` , puedes suscribirte a un canal comodín, el cual puede ser útil para capturar todos los mensajes en todos los canales. El nombre del canal `$channel` será pasado como segundo argumento al callback `Closure` proporcionado:

```

Redis::psubscribe(['*'], function ($message, $channel) {
    echo $message;
});

Redis::psubscribe(['users.*'], function ($message, $channel) {

```

php

```
echo $message;  
});
```

Eloquent: Primeros Pasos

- Introducción
- Definiendo modelos
 - Convenciones del modelo Eloquent
 - Valores de atributo predeterminados
- Obteniendo modelos
 - Colecciones
 - Resultados divididos en partes (chunk)
 - Subconsultas avanzadas
- Obteniendo modelos individuales / Agrupamientos
 - Obteniendo agrupamientos
- Insertando y actualizando modelos
 - Inserciones
 - Actualizaciones
 - Asignación masiva
 - Otros métodos de creación
- Eliminando modelos
 - Eliminación lógica
 - Consultando modelos eliminados lógicamente
- Alcances de consulta
 - Alcances globales
 - Alcances locales
- Comparando modelos
- Eventos
 - Observadores

Introducción

El ORM Eloquent incluido con Laravel proporciona una genial y simple implementación básica de ActiveRecord para trabajar con tu base de datos. Cada tabla de base de datos tiene un correspondiente "Modelo" el cual es usado para interactuar con la tabla. Los modelos permiten que consultes los datos en tus tablas, así como también insertar nuevos registros dentro de la tabla.

Antes de empezar, asegúrate de configurar una conexión de base de datos en `config/database.php`. Para mayor información sobre la configuración de tu base de datos, revisa la documentación.

Definiendo modelos

Para empezar, vamos a crear un modelo de Eloquent. Los modelos residen típicamente en el directorio `app`, pero eres libre de colocarlos en cualquier parte que pueda ser auto-cargada de acuerdo a tu archivo `composer.json`. Todos los modelos de Eloquent extienden la clase `Illuminate\Database\Eloquent\Model`.

La forma más fácil de crear una instancia del modelo es usando el Comando Artisan `make:model`:

```
php artisan make:model Flight
```

php

Si prefieres generar una migración de base de datos cuando generes el modelo, puedes usar la opción `--migration` o `-m`:

```
php artisan make:model Flight --migration
```

php

```
php artisan make:model Flight -m
```

Convenciones del modelo Eloquent

Ahora, vamos a mirar un modelo `Flight` de ejemplo, el cual usaremos para obtener y guardar información desde nuestra tabla de base de datos `flights`:

```
<?php
```

php

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Nombres de tabla

Observa que no le dijimos a Eloquent cuál tabla usar para nuestro modelo `Flight`. Por convención, el nombre de la clase en plural y en formato "snake_case" será usado como el nombre de tabla a menos que otro nombre sea especificado expresamente. Así, en este caso, Eloquent asumirá que el modelo `Flight` guarde los registros en la tabla `flights`. Puedes especificar una tabla personalizada al definir una propiedad `table` en tu 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 asumirá que cada tabla tiene una columna de clave primaria denominada `id`. Puedes definir una propiedad `$primaryKey` protegida para sobrescribir esta convención:

```
<?php  
  
namespace App;  
use Illuminate\Database\Eloquent\Model;  
  
class Flight extends Model  
{  
    /**  
     * The primary key associated with the table.  
     *  
     * @var string  
     */  
    protected $primaryKey = 'flight_id';  
}
```

Además, Eloquent asume que la clave primaria es un valor entero autoincremental, lo que significa que de forma predeterminada la clave primaria será convertida a un tipo `int` automáticamente. Si deseas usar una clave primaria que no sea de autoincremeneto o numérica debes establecer la propiedad pública `$incrementing` de tu modelo a `false` :

```
<?php  
  
class Flight extends Model  
{  
    /**  
     * Indicates if the IDs are auto-incrementing.  
     *  
     * @var bool  
     */  
    public $incrementing = false;  
}
```

Si tu clave primaria no es un entero, debes establecer la propiedad protegida `$keyType` de tu modelo a `string` :

```
<?php  
  
class Flight extends Model  
{
```

```
 /**
 * The "type" of the auto-incrementing ID.
 *
 * @var string
 */
protected $keyType = 'string';
}
```

Marcas de tiempo o timestamps

De forma predeterminada, Eloquent espera que las columnas `created_at` y `updated_at` existan en tus tablas. Si no deseas tener estas columnas manejadas automáticamente por Eloquent, establece la propiedad `$timestamps` de tu 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;  
}
```

php

Si necesitas personalizar el formato de tus marcas de tiempo, establece la propiedad `$dateFormat` de tu modelo. Esta propiedad determina como los atributos de fecha son guardados en la base de datos, también como su formato cuando el modelo es serializado a un arreglo o JSON:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;
```

php

```
class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Si necesitas personalizar los nombres de las columnas usadas para guardar las marcas de tiempo, puedes establecer las constantes `CREATED_AT` y `UPDATED_AT` en tu modelo:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'last_update';
}
```

php

Conexión de base de datos

De forma predeterminada, todos los modelos Eloquent usarán la conexión de base de datos configurada por tu aplicación. Si quieres especificar una conexión diferente para el modelo, usa la propiedad

`$connection :`

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
}
```

php

```
    protected $connection = 'connection-name';  
}
```

Valores de atributo predeterminados

Si deseas definir los valores predeterminados para algunos de los atributos de su modelo, puedes definir una propiedad `$attributes` en tu modelo:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Flight extends Model  
{  
    /**  
     * The model's default values for attributes.  
     *  
     * @var array  
     */  
    protected $attributes = [  
        'delayed' => false,  
    ];  
}
```

Obteniendo modelos

Una vez que has creado un modelo y su tabla de base de datos asociada, estás listo para empezar a obtener datos de tu base de datos. Piensa en cada modelo de Eloquent como un constructor de consultas muy poderoso que te permite consultar fluidamente la tabla de base de datos asociada con el modelo. Por ejemplo:

```
<?php  
  
$flights = App\Flight::all();  
  
foreach ($flights as $flight) {
```

```
echo $flight->name;  
}
```

Añadiendo restricciones adicionales

El método `all` de Eloquent devolverá todos los resultados en la tabla del modelo. Ya que cada modelo de Eloquent sirve como un constructor de consultas, también puedes añadir restricciones a las consultas y entonces usar el método `get` para obtener los resultados:

```
$flights = App\Flight::where('active', 1)  
    ->orderBy('name', 'desc')  
    ->take(10)  
    ->get();
```

TIP

Ya que los modelos de Eloquent son constructores de consultas, deberías revisar todos los métodos disponibles en el constructor de consultas. Puedes usar cualquiera de estos métodos en tus consultas de Eloquent.

Actualizando modelos

Podemos actualizar modelos usando los métodos `fresh` y `refresh`. El método `fresh` volverá a recuperar el modelo de la base de datos. La instancia de modelo existente no será afectada:

```
$flight = App\Flight::where('number', 'FR 900')->first();  
  
$freshFlight = $flight->fresh();
```

El método `refresh` "rehidratará" el modelo existente usando datos nuevos de la base de datos. Además, todas sus relaciones cargadas previamente serán también actualizadas:

```
$flight = App\Flight::where('number', 'FR 900')->first();  
  
$flight->number = 'FR 456';  
  
$flight->refresh();
```

```
$flight->number; // "FR 900"
```

Colecciones

Para métodos de Eloquent como `all` y `get` que obtienen varios resultados, se devolverá una instancia de `Illuminate\Database\Eloquent\Collection`. La clase `Collection` proporciona una variedad de métodos útiles para trabajar con los resultados de Eloquent:

```
$flights = $flights->reject(function ($flight) {  
    return $flight->cancelled;  
});
```

También puedes recorrer la colección como un arreglo:

```
foreach ($flights as $flight) {  
    echo $flight->name;  
}
```

Resultados divididos en partes (chunk)

Si necesitas procesar miles de registros de Eloquent, usa el comando `chunk`. El método `chunk` obtendrá una "porción" de los modelos de Eloquent, incorporándolos a una `Closure` dada para procesamiento. Usando el método `chunk` ahorrarás memoria al momento de trabajar con grandes conjuntos de resultados:

```
Flight::chunk(200, function ($flights) {  
    foreach ($flights as $flight) {  
        //  
    }  
});
```

El primer argumento pasado al método es el número de registros que deseas obtener por cada "porción". La Closure pasada como segundo argumento será ejecutada para cada porción que sea obtenida de la base de datos. Una consulta de base de datos será ejecutada para obtener cada porción de registros pasados a la Closure.

Usando cursos

El método `cursor` permite que iteres a través de registros de tu base de datos usando un cursor, el cual ejecutará solamente una consulta única. Al momento de procesar grandes cantidades de datos, puedes usar el método `cursor` para reducir en gran medida el uso de la memoria:

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {  
    //  
}
```

php

El `cursor` retorna una instancia `Illuminate\Support\LazyCollection`. Las `colecciones lazy` te permiten usar muchos de los métodos de colección disponibles en colecciones típicas de Laravel mientras que sólo carga un único modelo en memoria a la vez:

```
$users = App\User::cursor()->filter(function ($user) {  
    return $user->id > 500;  
});  
  
foreach ($users as $user) {  
    echo $user->id;  
}
```

php

Subconsultas avanzadas

Selects de subconsultas

Eloquent también ofrece soporte avanzado para subconsultas, lo que te permite extraer información de tablas relacionadas en una única consulta. Por ejemplo, imaginemos que tenemos una tabla

`destinations` y una tabla `flights`. La tabla `flights` contiene una columna `arrived_at` que indica cuando el vuelo ha arribado al destino.

Usando la funcionalidad de subconsulta disponible en los métodos `select` y `addSelect`, podemos seleccionar todos los destinos y el nombre del vuelo que más recientemente arribó a dicho destino usando una única consulta:

```
use App\Flight;  
use App\Destination;
```

php

```
return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderBy('arrived_at', 'desc')
    ->limit(1)
])->get();
```

Ordenando subconsultas

Adicionalmente, la función `orderBy` del constructor de consultas soporta subconsultas. Podemos usar esta funcionalidad para ordenar los destinos en base a cuando llegó el último vuelo a dicho destino. De nuevo, esto puede ser hecho ejecutando una única consulta en la base de datos:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderBy('arrived_at', 'desc')
        ->limit(1)
)->get();
```

php

Obteniendo modelos individuales / Agrupamientos

Además de obtener todos los registros de una tabla dada, también puedes obtener registros individuales usando `find`, `first` o `firstWhere`. En lugar de devolver una colección de modelos, estos métodos devuelven una única instancia de modelo:

```
// Recupera un modelo por su clave primaria...
$flight = App\Flight::find(1);

// Recupera el primer modelo que coincide con las restricciones de consulta...
$flight = App\Flight::where('active', 1)->first();

// Shorthand for retrieving the first model matching the query constraints...
$flight = App\Flight::firstWhere('active', 1);
```

php

También puedes ejecutar el método `find` con un arreglo de claves primarias, el cual devolverá una colección de los registros que coincidan:

```
$flights = App\Flight::find([1, 2, 3]);
```

php

Algunas veces puedes querer retornar el primer resultado de una consulta o realizar alguna otra acción si ningún resultado es encontrado. El método `firstOr` retornará el primer resultado encontrado o, si ningún resultado es encontrado, ejecutará el callback dado. El resultado del callback será considerado el resultado del método `firstOr`:

```
$model = App\Flight::where('legs', '>', 100)->firstOr(function () {
    // ...
});
```

php

El método `firstOr` también acepta un arreglo de columnas a ser retornadas:

```
$model = App\Flight::where('legs', '>', 100)
    ->firstOr(['id', 'legs'], function () {
        // ...
});
```

php

Excepciones not found (no encontrado)

Algunas veces, puedes desear arrojar una excepción si un modelo no es encontrado. Es particularmente útil en rutas o controladores. Los métodos `findOrFail` y `firstOrFail` obtendrán el primer resultado de la consulta; sin embargo, si nada es encontrado, una excepción de

`Illuminate\Database\Eloquent\ModelNotFoundException` será arrojada:

```
$model = App\Flight::findOrFail(1);
```

php

```
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

Si la excepción no es atrapada, una respuesta HTTP `404` es enviada automáticamente de regreso al usuario. No es necesario escribir verificaciones explícitas para devolver respuestas `404` cuando uses estos métodos:

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
```

php

```
});
```

Obteniendo agrupamientos

También puedes usar los métodos `count` , `sum` , `max` y otros [métodos de agrupamiento](#) proporcionados por el [constructor de consulta](#). Estos métodos devuelven el valor escalar apropiado en lugar de una completa instancia de modelo:

```
$count = App\Flight::where('active', 1)->count();  
  
$max = App\Flight::where('active', 1)->max('price');
```

php

Insertando Y actualizando modelos

Inserciones

Para agregar un nuevo registro en la base de datos crea una nueva instancia de modelo, establece los atributos del modelo y después ejecuta el método `save` :

```
<?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...
```

php

```
$flight = new Flight;

$flight->name = $request->name;

$flight->save();
}

}
```

En este ejemplo, asignamos el parámetro `name` de la solicitud entrante al atributo `name` de la instancia del modelo `App\Flight`. Cuando ejecutamos el método `save`, un registro será insertado en la base de datos. Las marcas de tiempo `created_at` y `updated_at` serán automáticamente establecidas cuando el método `save` sea ejecutado, no hay necesidad de establecerlos manualmente.

Actualizaciones

El método `save` también puede ser usado para actualizar modelos que ya existen en la base de datos. Para actualizar un modelo, debes obtenerlo, establecer cualquiera de los atributos que deseas actualizar y después ejecutar el método `save`. Otra vez, la marca de tiempo `updated_at` será actualizada automáticamente, no hay necesidad de establecer su valor manualmente.

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

Actualizaciones masivas

Las actualizaciones también pueden ser ejecutadas contra cualquier número de modelos que coincidan con un criterio de consulta dada. En este ejemplo, todos los vuelos que están activos o con `active` igual a 1 y tienen un atributo `destination` igual a `San Diego` serán marcados como retrasados:

```
App\Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

El método `update` espera un arreglo de pares de columna y valor representando las columnas que deberían ser actualizadas.

Nota

Al momento de utilizar una actualización masiva por medio de Eloquent, los eventos de modelo `saving`, `saved`, `updating` y `updated` no serán disparados para los modelos actualizados. Esto es debido a que los modelos nunca son obtenidos en realidad al momento de hacer una actualización masiva.

Examinando cambios en los atributos

Eloquent proporciona los métodos `isDirty`, `isClean` y `wasChanged` para examinar el estado interno de tus modelos y determinar cómo sus atributos han cambiado desde que fueron originalmente cargados.

El método `isDirty` determina si algún atributo ha cambiado desde que el modelo fue cargado. Puedes pasar un nombre de atributo específico para determinar si un atributo particular está sucio. El método `isClean` es el opuesto a `isDirty` y también acepta un atributo opcional como argumento:

```
php
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

El método `wasChanged` determina si algún atributo fue cambiado cuando el modelo fue guardado por última vez dentro del ciclo de solicitud actual. También puedes pasar un nombre de atributo para ver si un atributo particular ha cambiado:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';
$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged('first_name'); // false
```

Asignación masiva

También puedes usar el método `create` para guardar un nuevo modelo en una sola línea. La instancia de modelo insertada te será devuelta por el método. Sin embargo, antes de hacer eso, necesitarás especificar o un atributo `fillable` o `guarded` del modelo, de modo que todos los modelos de Eloquent se protejan contra la asignación masiva de forma predeterminada.

Una vulnerabilidad en la asignación masiva ocurre cuando un usuario pasa un parámetro HTTP inesperado a través de una solicitud y ese parámetro cambia una columna en tu base de datos que no esperabas. Por ejemplo, un usuario malicioso podría enviar un parámetro `is_admin` a través de una solicitud HTTP, la cual es entonces pasada en el método `create` de tu modelo, permitiendo que el usuario se promueva a si mismo como un usuario administrador.

Así que, para empezar, debes definir cuáles atributos del modelo quieras que se asignen de forma masiva. Puedes hacerlo usando la propiedad `$fillable` del modelo. Por ejemplo, vamos a hacer el atributo `name` de nuestro modelo `Flight` sea asignado masivamente.

```
<?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 hemos indicado los atributos asignables en masa, podemos usar el método `create` para insertar un nuevo registro en la base de datos. El método `create` devuelve la instancia de modelo guardada:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

php

Si ya tienes una instancia del modelo, puedes usar el método `fill` para llenarla con un arreglo de atributos:

```
$flight->fill(['name' => 'Flight 22']);
```

php

Protección de atributos

Mientras `$fillable` sirve como una "lista blanca" de atributos que deben ser asignados en forma masiva, también puedes elegir usar `$guarded`. La propiedad `$guarded` debe contener un arreglo de atributos que no deseas que sean asignados en forma masiva. El resto de atributos que no estén en el arreglo serán asignados masivamente. `$guarded` funciona como una "lista negra". Importante, debes usar `$fillable` o `$guarded` - pero no ambos. En el ejemplo siguiente, todos los atributos `excepto price` serán asignados en forma masiva:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
```

php

```
{  
    /**  
     * The attributes that aren't mass assignable.  
     *  
     * @var array  
     */  
    protected $guarded = ['price'];  
}
```

Si prefieres hacer todos los atributos asignados masivamente, puedes definir la propiedad `$guarded` como un arreglo vacío:

```
/**  
 * The attributes that aren't mass assignable.  
 *  
 * @var array  
 */  
protected $guarded = [];
```

php

Otros Métodos De Creación

`firstOrCreate / firstOrNew`

Hay otros dos métodos que puedes usar para crear modelos con atributos de asignación masiva:

`firstOrCreate` y `firstOrNew`. El método `firstOrCreate` intentará localizar un registro de base de datos usando los pares columna / valor dados. Si el modelo no puede ser encontrado en la base de datos, un registro será insertado con los atributos del primer parámetro, junto con aquellos del segundo parámetro opcional.

El método `firstOrNew`, al igual que `firstOrCreate`, intentará localizar un registro en la base de datos que coincida con los atributos dados. Sin embargo, si un modelo no es encontrado, una nueva instancia de modelo será devuelta. Nota que el modelo devuelto por `firstOrNew` todavía no ha sido enviado a la base de datos. Necesitarás ejecutar `save` manualmente para hacerlo persistente:

```
// Recupera el vuelo por nombre, o lo crea si no existe...  
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);  
  
// Recupera vuelo por nombre o lo crea con los atributos name, delayed y arrival  
$flight = App\Flight::firstOrCreate(
```

php

```
[ 'name' => 'Flight 10'],
['delayed' => 1, 'arrival_time' => '11:30']
);

// Recupera por nombre, o instancia...
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

// Recupera por nombre o crea una instancia con los atributos name, delayed y ar
$flight = App\Flight::firstOrNew(
    ['name' => 'Flight 10'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

updateOrCreate

También puedes encontrar situaciones donde quieras actualizar un modelo existente o crear un nuevo modelo si no existe. Laravel proporciona un método `updateOrCreate` para hacer esto en un paso. Al igual que el método `firstOrCreate`, `updateOrCreate` persiste el modelo, para que no haya necesidad de ejecutar `save()`:

```
// Si hay un vuelo desde Oakland a San Diego, establece el precio a $99.          php
// Si no existe un modelo que coincida, crea uno.
$flight = App\Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Eliminando modelos

Para eliminar un modelo, ejecuta el método `delete` en una instancia del modelo:

```
$flight = App\Flight::find(1);          php

$flight->delete();
```

Eliminando un modelo existente por clave

En el ejemplo anterior, estamos obteniendo el modelo de la base de datos antes de ejecutar el método `delete`. Sin embargo, si conoces la clave primaria del modelo, puedes eliminar el modelo sin obtenerlo primero. Para hacer eso, ejecuta el método `destroy`. Además de recibir una sola clave primaria como argumento, el método `destroy` aceptará múltiples claves primarias, o una `collection` de claves primarias:

```
App\Flight::destroy(1);  
  
App\Flight::destroy(1, 2, 3);  
  
App\Flight::destroy([1, 2, 3]);  
  
App\Flight::destroy(collect([1, 2, 3]));
```

php

Eliminando modelos por consultas

También puedes ejecutar una instrucción de eliminar en un conjunto de modelos. En este ejemplo, eliminaremos todos los vuelos que están marcados como inactivos. Al igual que las actualizaciones masivas, las eliminaciones masivas no dispararán cualquiera de los eventos de modelo para los modelos que son eliminados:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

php

Nota

Al momento de ejecutar una instrucción de eliminación masiva por medio de Eloquent, los eventos de modelo `deleting` and `deleted` no serán ejecutados para los modelos eliminados. Esto es debido a que los modelos nunca son obtenidos realmente al momento de ejecutar la instrucción de eliminación.

Eliminación lógica (Soft Deleting)

Además de eliminar realmente los registros de tu base de datos, Eloquent también puede "eliminar lógicamente" los modelos. Cuando los modelos son borrados lógicamente, no son removidos realmente de tu base de datos. En lugar de eso, un atributo `deleted_at` es establecido en el modelo e insertado en la base de datos. Si un modelo tiene un valor `deleted_at` no nulo, el modelo ha sido eliminado

lógicamente. Para habilitar eliminaciones lógicas en un modelo, usa el trait

`Illuminate\Database\Eloquent\SoftDeletes` en el modelo:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\SoftDeletes;  
  
class Flight extends Model  
{  
    use SoftDeletes;  
}
```

TIP

El trait `SoftDeletes` convertirá (cast) automáticamente el atributo `deleted_at` a una instancia de `DateTime` / `Carbon` para ti.

Debes añadir la columna `deleted_at` a tu tabla de base de datos. El [constructor de esquemas](#) de Laravel contiene un método helper para crear esta columna:

```
Schema::table('flights', function (Blueprint $table) {  
    $table->softDeletes();  
});
```

Ahora, cuando ejecutes el método `delete` en el modelo, la columna `deleted_at` será establecida con la fecha y hora actual. Y, al momento de consultar un modelo que use eliminaciones lógicas, los modelos eliminados lógicamente serán excluidos automáticamente de todos los resultados de consultas.

Para determinar si una instancia de modelo ha sido eliminada lógicamente, usa el método `trashed` :

```
if ($flight->trashed()) {  
    //  
}
```

Consultando Modelos Eliminados Lógicamente

Incluyendo modelos eliminados lógicamente

Como se apreció anteriormente, los modelos eliminados lógicamente serán excluidos automáticamente de los resultados de las consultas. Sin embargo, puedes forzar que los modelos eliminados lógicamente aparezcan en un conjunto resultante usando el método `withTrashed` en la consulta:

```
$flights = App\Flight::withTrashed()  
    ->where('account_id', 1)  
    ->get();
```

php

El método `withTrashed` también puede ser usado en una consulta de [relación de Eloquent](#):

```
$flight->history()->withTrashed()->get();
```

php

Obteniendo modelos individuales eliminados lógicamente

El método `onlyTrashed` obtendrá **sólo** modelos eliminados lógicamente:

```
$flights = App\Flight::onlyTrashed()  
    ->where('airline_id', 1)  
    ->get();
```

php

Restaurando modelos eliminados lógicamente

Algunas veces puedes desear "deshacer la eliminación" de un modelo eliminado lógicamente. Para restaurar un modelo eliminado lógicamente a un estado activo, usa el método `restore` en una instancia de modelo:

```
$flight->restore();
```

php

También puedes usar el método `restore` en una consulta para restaurar rápidamente varios modelos. Otra vez, al igual que otras operaciones "masivas", esto no disparará cualquiera de los eventos de modelo para los modelos que sean restaurados:

```
App\Flight::withTrashed()
    ->where('airline_id', 1)
    ->restore();
```

php

Al igual que con el método `withTrashed`, el método `restore` también puede ser usado en relaciones de Eloquent:

```
$flight->history()->restore();
```

php

Eliminando modelos permanentemente

Algunas veces puedes necesitar eliminar verdaderamente un modelo de tu base de datos. Para remover permanentemente un modelo eliminado lógicamente de la base de datos, usa el método

`forceDelete` :

```
// Obliga la eliminación de una instancia de un solo modelo...
$flight->forceDelete();

// Obliga la eliminación de todos los modelos relacionados...
$flight->history()->forceDelete();
```

php

Alcances (Scopes) de consultas

Alcances (scopes) globales

Los alcances globales permiten que añadas restricciones a todas las consultas para un modelo dado. La propia funcionalidad de la [eliminación lógica](#) de Laravel utiliza alcances globales para extraer solamente los modelos "no-eliminados" de la base de datos. Escribiendo tus propios alcances globales puede proporcionarte una forma conveniente y fácil de asegurar que cada consulta para un modelo dado reciba ciertas restricciones.

Escribiendo scopes globales

Escribir un alcance global es simple. Define una clase que implemente la interfaz `Illuminate\Database\Eloquent\Scope`. Esta interfaz requiere que implementes un método: `apply`. El método `apply` puede añadir restricciones `where` a la consulta como sea necesario:

php

```
<?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);
    }
}
```

TIP

Si tu scope global está agregando columnas a la cláusula select de la consulta, deberías usar el método `addSelect` en lugar de `select`. Esto evitara el reemplazo no intencional de la cláusula select existente de la consulta.

Aplicando scopes globales

Para asignar un scope global a un modelo, debes sobrescribir el método `boot` del modelo dado y usar el método `addGlobalScope` :

php

```
<?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 código SQL:

```
select * from `users` where `age` > 200
```

php

Alcances globales anónimos

Eloquent también permite que definas scopes globales usando Closures, lo cual es particularmente útil para scopes simples que no se crean en una clase separada:

```
<?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()
    {
```

php

```
parent::boot();

    static::addGlobalScope('age', function (Builder $builder) {
        $builder->where('age', '>', 200);
    });
}

}
```

Eliminar scopes globales

Si prefieres remover un scope global para una consulta dada, puedes usar el método `withoutGlobalScope`. El método acepta el nombre de clase del scope global como su único argumento:

```
User::withoutGlobalScope(AgeScope::class)->get();
```

php

O, si definiste el scope global usando un Closure:

```
User::withoutGlobalScope('age')->get();
```

php

Si prefieres eliminar varios o incluso todos los scopes globales, puedes usar el método `withoutGlobalScopes`:

```
// Elimina todos los scopes globales...
User::withoutGlobalScopes()->get();

// Elimina algunos de los scopes globales...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

php

Alcances (scopes) locales

Los scopes locales permiten que definas conjuntos de restricciones comunes que puedes reusar fácilmente a traves de tu aplicación. Por ejemplo, puedes necesitar obtener frecuentemente todos los usuarios que son considerados "populares". Para definir un scope, agrega el prefijo `scope` a un método de modelo de Eloquent.

Los scopes deberían devolver siempre una instancia del constructor de consultas:

```
<?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);  
    }  
}
```

php

Utilizando un scope local

Una vez que el scope ha sido definido, puedes ejecutar los métodos de scope al momento de consultar el modelo. Sin embargo, no debes incluir el prefijo `scope` cuando ejecutas el método. Incluso puedes encadenar las ejecuciones a varios scopes, por ejemplo:

```
$users = App\User::popular()->active()->orderBy('created_at')->get();
```

php

La combinación de múltiples scopes de modelo Eloquent a través de un operador de consulta `or` puede requerir el uso de funciones de retorno Closure como:

```
$users = App\User::popular()->orWhere(function (Builder $query) {  
    $query->active();  
})->get();
```

php

Sin embargo, dado que esto puede ser engorroso, Laravel proporciona un método de "orden superior" `orWhere` que te permite encadenar estos scopes con fluidez sin el uso de Closure:

```
$users = App\User::popular()->orWhere->active()->get();
```

php

Scopes dinámicos

Algunas veces, puedes desear definir un scope que acepte parámetros. Para empezar, sólo agrega tus parámetros adicionales a tu scope. Los parámetros de scope deben ser definidos después del parámetro `$query`:

```
<?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);  
    }  
}
```

php

Ahora, puedes pasar los parámetros cuando llamas al scope:

```
$users = App\User::ofType('admin')->get();
```

php

Comparando modelos

En ocasiones necesitarás determinar si dos modelos son "el mismo". El método `is` puede ser usado para verificar rápidamente dos modelos que comparten llave principal, tabla y conexión a base de datos:

```
if ($post->is($anotherPost)) {  
    //  
}
```

php

Eventos

Los modelos de Eloquent ejecutan varios eventos, permitiendo que captes los siguientes puntos en un ciclo de vida del modelo: `retrieved` , `creating` , `created` , `updating` , `updated` , `saving` , `saved` , `deleting` , `deleted` , `restoring` y `restored` . Los eventos permiten que ejecutes fácilmente código cada vez que una clase de modelo específica es guardada o actualizada en la base de datos.

El evento `retrieved` se disparará cuando un modelo existente es obtenido de la base de datos.

Cuando un nuevo modelo es guardado la primera vez, los eventos `creating` y `created` se disparan. Si un modelo ya existe en la base de datos y el método `save` es ejecutado, los eventos

`updating` / `updated` se dispararán. Sin embargo, en ambos casos, los eventos `saving` / `saved` se dispararán.

Nota

Al realizar una actualización o eliminación masiva a través de Eloquent, los eventos de modelo

`saved` , `updated` , `deleting` y `deleted` no se activarán para los modelos actualizados.

Esto se debe a que los modelos nunca se recuperan cuando se emite una actualización masiva.

Para empezar, define una propiedad `$dispatchesEvents` en tu modelo Eloquent que mapee varios puntos del ciclo de vida de modelo de Eloquent a tus propias [clases de eventos](#):

php

```
<?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,
    ];
}
```

Después de definir y mapear tus eventos Eloquent, puedes usar [listeners de eventos](#) para manejar los eventos.

Observadores

Definiendo observadores

Si estás escuchando muchos eventos en un modelo dado, puedes usar observadores para agrupar todos tus listeners dentro de una sola clase. Las clases observadoras tienen nombres de métodos que reflejan los eventos de Eloquent que deseas escuchar. Cada uno de estos métodos reciben el modelo como su único argumento. El comando `make:observer` Artisan es la forma más sencilla de crear una nueva clase de observador:

php

```
php artisan make:observer UserObserver --model=User
```

Este comando colocará el nuevo observador en tu directorio `App/Observers`. Si este directorio no existe, Artisan lo creará por ti. Tu nuevo observador lucirá como lo siguiente:

```
<?php  
  
namespace App\Observers;  
  
use App\User;  
  
class UserObserver  
{  
    /**  
     * Handle the User "created" event.  
     *  
     * @param \App\User $user  
     * @return void  
     */  
    public function created(User $user)  
    {  
        //  
    }  
  
    /**  
     * Handle the User "updated" event.  
     *  
     * @param \App\User $user  
     * @return void  
     */  
    public function updated(User $user)  
    {  
        //  
    }  
  
    /**  
     * Handle the User "deleted" event.  
     *  
     * @param \App\User $user  
     * @return void  
     */  
    public function deleted(User $user)  
    {  
        //  
    }  
}
```

```
/**
 * Handle the User "forceDeleted" event.
 *
 * @param \App\User $user
 * @return void
 */
public function forceDeleted(User $user)
{
    //
}
```

Para registrar un observador, usa el método `observe` en el modelo que deseas observar. Puedes registrar los observadores en el método `boot` de uno de tus proveedores de servicio. En este ejemplo, registraremos el observador en el `AppServiceProvider`:

```
<?php  
  
namespace App\Providers;  
  
use App\User;  
use App\Observers\UserObserver;  
use Illuminate\Support\ServiceProvider;  
  
class AppServiceProvider extends ServiceProvider  
{  
    /**  
     * Register any application services.  
     *  
     * @return void  
     */  
    public function register()  
    {  
        //  
    }  
  
    /**  
     * Bootstrap any application services.  
     *  
     * @return void  
     */  
}
```

```
public function boot()
{
    User::observe(UserObserver::class);
}
```

Eloquent: Relaciones

- Introducción
- Definiendo relaciones
 - Uno a uno
 - Uno a muchos
 - Uno a muchos (inverso)
 - Muchos a muchos
 - Definiendo modelos de tabla intermedia personalizados
 - Tiene uno a través de
 - Tiene muchos a través de
- Relaciones polimórficas
 - Uno a uno
 - Uno a muchos
 - Muchos a muchos
 - Tipos polimórficos personalizados
- Consultando relaciones
 - Métodos de relación vs. propiedades dinámicas
 - Consultando la existencia de relación
 - Consultando la ausencia de relación
 - Consultando relaciones polimórficas
 - Contando modelos relacionados
- Precarga (eager loading)

- Restringiendo precargas
- Precarga diferida (lazy eager loading)
- Insertando y actualizando modelos relacionados
 - El método `save`
 - El método `create`
 - Actualizando relaciones pertenece a (BelongsTo)
 - Actualizando relaciones muchos a muchos
- Tocando marcas de tiempo del padre

Introducción

Las tablas de base de datos frecuentemente están relacionadas a otra tabla. Por ejemplo, un post de un blog puede tener muchos comentarios o un pedido podría estar relacionado al usuario que lo ordenó. Eloquent hace que manejar y trabajar con estas relaciones sea fácil y soporta varios tipos de relaciones:

- Uno a Uno
- Uno a Muchos
- Muchos a Muchos
- Uno a Tráves de
- Muchos a Tráves de
- Uno a Uno (Polimórfica)
- Uno a Muchos (Polimórfica)
- Muchos a Muchos (Polimórfica)

Definiendo relaciones

Las relaciones de Eloquent se definen como métodos en tus clases de modelo de Eloquent. Debido a que, como los mismos modelos Eloquent, las relaciones también sirven como poderosos constructores de consultas, puesto que definir relaciones como métodos proporciona potentes capacidades de encadenamiento de métodos y consultas. Por ejemplo, podemos encadenar restricciones adicionales en esta relación `posts` :

```
$user->posts()->where('active', 1)->get();
```

php

Pero, antes de profundizar demasiado en el uso de relaciones, aprendamos cómo definir cada tipo.

Nota

Los nombres de las relaciones no pueden colisionar con nombres de atributos dado que eso podría ocasionar que tu modelo no sea capaz de saber cuál resolver.

Uno A Uno

Una relación de uno a uno es una relación muy sencilla. Por ejemplo, un modelo `User` podría estar asociado con un `Phone`. Para definir esta relación, colocaremos un método `phone` en el modelo `User`. El método `phone` debería llamar al método `hasOne` y devolver 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');  
    }  
}
```

php

El primer argumento pasado al método `hasOne` es el nombre del modelo relacionado. Una vez que la relación es definida, podemos obtener el registro relacionado usando propiedades dinámicas de Eloquent. Las propiedades dinámicas permiten que accedas a métodos de relación como si fueran propiedades definidas en el modelo:

```
$phone = User::find(1)->phone;
```

php

Eloquent determina la clave foránea de la relación en base al nombre del modelo. En este caso, se asume automáticamente que el modelo `Phone` tenga una clave foránea `user_id`. Si deseas sobrescribir esta convención, puedes pasar un segundo argumento al método `hasOne`:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

php

Adicionalmente, Eloquent asume que la clave foránea debería tener un valor que coincida con la columna `id` (o `$primaryKey` personalizada) del padre. En otras palabras, Eloquent buscará el valor de la columna `id` del usuario en la columna `user_id` de `Phone`. Si prefieres que la relación use un valor distinto de `id`, puedes pasar un tercer argumento al método `hasOne` especificando tu clave personalizada:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

php

Definiendo el inverso de la relación

Así, podemos acceder al modelo `Phone` desde nuestro `User`. Ahora, vamos a definir una relación en el modelo `Phone` que nos permitirá acceder al `User` que posee el teléfono. Podemos definir el inverso de una relación `hasOne` usando 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');  
    }  
}
```

php

En el ejemplo anterior, Eloquent intentará hacer coincidir el `user_id` del modelo `Phone` con un `id` en el modelo `User`. Eloquent determina el nombre de la clave foránea de forma predeterminada al examinar el nombre del método de la relación y agregando el sufijo al nombre del método con `_id`.

Sin embargo, si la clave foránea en el modelo `Phone` no es `user_id`, puedes pasar un nombre de clave personalizada como segundo argumento al método `belongsTo`:

```
/** php
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key');
}
```

Si tu modelo padre no usa `id` como su clave primaria, o deseas hacer join al modelo hijo con una columna diferente, puedes pasar un tercer argumento al método `belongsTo` especificando la clave personalizada de tu tabla padre:

```
/** php
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo('App\User', 'foreign_key', 'other_key');
}
```

Uno a muchos

Una relación de "uno-a-muchos" es usada para definir relaciones donde un solo modelo posee cualquier cantidad de otros modelos. Por ejemplo, un post de un blog puede tener un número infinito de comentarios. Al igual que todas las demás relaciones de Eloquent, las relaciones uno-a-muchos son definidas al colocar una función en tu modelo Eloquent:

```
<?php 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');  
}  
}
```

Recuerda, Eloquent determinará automáticamente la columna de clave foránea apropiada en el modelo `Comment`. Por convención, Eloquent tomará el nombre "snake_case" del modelo que la contiene y le agregará el sufijo `_id`. Para este ejemplo, Eloquent asumirá que la clave foránea del modelo `Comment` es `post_id`.

Una vez que la relación ha sido definida, podemos acceder a la colección de comentarios al acceder a la propiedad `comments`. Recuerda, ya que Eloquent proporciona "propiedades dinámicas", podemos acceder a los métodos de la relación como si fueran definidos como propiedades en el modelo:

```
$comments = App\\Post::find(1)->comments;  
  
foreach ($comments as $comment) {  
    //  
}
```

Debido a que todas las relaciones también sirven como constructores de consultas (query builders), puedes agregar restricciones adicionales a cuyos comentarios sean obtenidos ejecutando el método `comments` y encadenando condiciones en la consulta:

```
$comment = App\\Post::find(1)->comments()->where('title', 'foo')->first();
```

Igual que el método `hasOne`, también puedes sobrescribir las claves foráneas y locales al pasar 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 puedes acceder a todos los comentarios de un post, vamos a definir una relación para permitir a un comentario acceder a su post padre. Para definir el inverso de una relación `hasMany`, define una función de relación en el modelo hijo que ejecute el 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');  
    }  
}
```

php

Una vez que la relación ha sido definida, podemos obtener el modelo `Post` para un `Comment` accediendo a la "propiedad dinámica" de `post`:

```
$comment = App\Comment::find(1);  
  
echo $comment->post->title;
```

php

En el ejemplo anterior, Eloquent tratará de hacer coincidir el `post_id` del modelo `Comment` con un `id` en el modelo `Post`. Eloquent determina el nombre de la clave foránea por defecto, examinando el nombre del método de la relación y agregando un sufijo `_` al nombre del método, seguido del nombre de la columna principal de la llave. Sin embargo, si la clave foránea en el modelo `Comment` no es `post_id`, puedes pasar un nombre de clave personalizado como segundo argumento al método `belongsTo`:

```
/**  
 * Get the post that owns the comment.  
 */
```

php

```
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key');
}
```

Si tu modelo padre no usa `id` como su clave primaria, o deseas hacer join al modelo hijo con una columna diferente, puedes pasar un tercer argumento al método `belongsTo` especificando la clave personalizada de tu tabla padre.

```
/*
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo('App\Post', 'foreign_key', 'other_key');
}
```

php

Muchos a muchos

Las relaciones de muchos-a-muchos son ligeramente más complicadas que las relaciones `hasOne` y `hasMany`. Un ejemplo de tal relación es un usuario con muchos roles, donde los roles también son compartidos por otros usuarios. Por ejemplo, muchos usuarios pueden tener el rol "Admin".

Estructura de la tabla

Para definir esta relación, tres tablas de bases de datos son necesitadas: `users`, `roles`, y `role_user`. La tabla `role_user` es derivada del orden alfabético de los nombres de modelo relacionados y contiene las columnas `user_id` y `role_id`.

```
users
    id - entero
    name - cadena
roles
    id - entero
    name - cadena
role_user
    user_id - entero
    role_id - entero
```

php

Estructura del modelo

Las relaciones de muchos-a-muchos son definidas escribiendo un método que devuelve el resultado del método `belongsToMany`. Por ejemplo, vamos a definir el método `roles` en nuestro 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 que la relación es definida, puedes acceder a los roles del usuario usando la propiedad dinámica `roles`:

```
$user = App\User::find(1);  
  
foreach ($user->roles as $role) {  
    //  
}
```

Como con los otros tipos de relación, puedes ejecutar el método `roles` para continuar encadenando las restricciones de consulta en la relación:

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

Como mencionamos previamente, para determinar el nombre de la tabla asociativa, Eloquent juntará los dos nombres de modelo en orden alfabético. Sin embargo, eres libre de sobrescribir esta convención.

Puedes hacer eso pasando un segundo argumento al método `belongsToMany`:

```
return $this->belongsToMany('App\Role', 'role_user');
```

php

Además de personalizar el nombre de la tabla asociativa, también puedes personalizar los nombres de columna de las claves en la tabla pasando argumentos adicionales al método `belongsToMany`. El tercer argumento es el nombre de clave foránea del modelo en el cual estás definiendo la relación, mientras el cuarto argumento es el nombre de la clave foránea del modelo que estás asociando:

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

php

Definiendo el inverso de la relación

Para definir el inverso de una relación de muchos-a-muchos, puedes colocar otra llamada de `belongsToMany` en tu modelo relacionado. Para continuar con nuestro ejemplo de roles de usuario, vamos a definir el método `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');  
    }  
}
```

php

Como puedes ver, la relación es definida exactamente de la misma forma que su contraparte `User`, con la excepción de que referencia al modelo `App\User`. Ya que estamos reusando el método `belongsToMany`, todas las tablas y opciones de personalización de claves usuales están disponibles al momento de definir el inverso de las relaciones de muchos-a-muchos.

Obteniendo columnas de tablas intermedias (Pivote)

Como ya has aprendido, trabajar con relaciones de muchos-a-muchos requiere la presencia de una tabla intermedia o pivote. Eloquent proporciona algunas formas muy útiles de interactuar con esta tabla. Por ejemplo, vamos a asumir que nuestro objeto `User` tiene muchos objetos `Role` al que está relacionado. Después de acceder a esta relación, podemos acceder a la tabla intermedia usando el atributo `pivot` en los modelos:

```
$user = App\User::find(1);  
  
foreach ($user->roles as $role) {  
    echo $role->pivot->created_at;  
}
```

Ten en cuenta que a cada modelo `Role` que obtenemos se le asigna automáticamente un atributo `pivot`. Este atributo contiene un modelo que representa la tabla intermedia y puede ser usado como cualquier otro modelo de Eloquent.

De forma predeterminada, solo las claves del modelo estarán presentes en el objeto `pivot`. Si tu tabla pivote contiene atributos extras, debes especificarlos cuando definas la relación.

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

Si quieres que tu tabla pivote automáticamente mantenga las marcas de tiempo `created_at` y `updated_at`, usa el método `withTimestamps` en la definición de la relación:

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Personalizando el nombre del atributo `pivot`

Como se señaló anteriormente, los atributos de la tabla intermedia pueden ser accedidos en modelos usando el atributo `pivot`. Sin embargo, eres libre de personalizar el nombre de este atributo para que refleje mejor su propósito dentro de tu aplicación.

Por ejemplo, si tu aplicación contiene usuarios que pueden suscribirse a podcasts, probablemente tengas una relación de muchos-a-muchos entre usuarios y podcasts. Si éste es el caso, puedes desear

renombrar tu tabla pivote intermedia como `subscription` en lugar de `pivot`. Esto puede ser hecho usando el método `as` al momento de definir la relación:

```
return $this->belongsToMany('App\\Podcast')
    ->as('subscription')
    ->withTimestamps();
```

php

Una vez hecho esto, puedes acceder a los datos de la tabla intermedia usando el nombre personalizado:

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

php

Filtrando relaciones a través de columnas de tablas intermedias

También puedes filtrar los resultados devueltos por `belongsToMany` usando los métodos `wherePivot`, `wherePivotIn` y `wherePivotNotIn` al momento de definir la relación:

```
return $this->belongsToMany('App\\Role')->wherePivot('approved', 1);

return $this->belongsToMany('App\\Role')->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany('App\\Role')->wherePivotNotIn('priority', [1, 2]);
```

php

Definiendo modelos de tabla intermedia personalizados

Si prefieres definir un modelo personalizado para representar la tabla intermedia o pivote de tu relación, puedes ejecutar el método `using` al momento de definir la relación. Los modelos de tablas intermedias de muchos-a-muchos personalizados deben extender la clase

`Illuminate\Database\Eloquent\Relations\Pivot` mientras que los modelos polimórficos muchos-a-muchos deben extender la clase

`Illuminate\Database\Eloquent\Relations\MorphPivot`. Por ejemplo, podemos definir un `Role` que use un modelo pivote `RoleUser` personalizado:

```
<?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\RoleUser');  
    }  
}
```

Al momento de definir el modelo `RoleUser`, extenderemos la clase `Pivot`:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Relations\Pivot;  
  
class RoleUser extends Pivot  
{  
    //  
}
```

Puedes combinar `using` y `withPivot` para retornar columnas de la tabla intermedia. Por ejemplo, puedes retornar las columnas `created_by` y `updated_by` desde la tabla pivot `RoleUser` pasando los nombres de las columnas al método `withPivot`:

```
<?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\RoleUser')
            ->withPivot([
                'created_by',
                'updated_by',
            ]);
    }
}
```

Nota

Nota: Los modelos Pivot no pueden usar el trait `SoftDeletes`. Si necesitas hacer soft delete de registros pivot considera convertir tu modelo pivot a un modelo de Eloquent.

Modelos de pivote personalizados e IDs incrementales

Si has definido una relación de muchos a muchos que usa un modelo de pivote personalizado, y ese modelo de pivote tiene una clave primaria de incremento automático, debes asegurarte de que su clase de modelo de pivote personalizado defina una propiedad `incrementing` que se establece en `true`.

```
/**
 * Indicates if the IDs are auto-incrementing.
 *
 * @var bool
 */
public $incrementing = true;
```

php

Tiene uno a través de (hasOneThrough)

La relación "tiene uno a través" vincula los modelos a través de una única relación intermedia. Por ejemplo, si cada proveedor (supplier) tiene un usuario (user) y cada usuario está asociado con un registro

del historial (history) de usuarios, entonces el modelo del proveedor puede acceder al historial del usuario *a través* del usuario. Veamos las tablas de base de datos necesarias para definir esta relación:

```
users
  id - integer
  supplier_id - integer

suppliers
  id - integer

history
  id - integer
  user_id - integer
```

php

Aunque la tabla `history` no contiene una columna `supplier_id`, la relación `hasOneThrough` puede proporcionar acceso al historial del usuario desde el modelo del proveedor. Ahora que hemos examinado la estructura de la tabla para la relación, vamos a definirla en el modelo `Supplier`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Supplier extends Model
{
    /**
     * Get the user's history.
     */
    public function userHistory()
    {
        return $this->hasOneThrough('App\History', 'App\User');
    }
}
```

php

El primer argumento pasado al método `hasOneThrough` es el nombre del modelo final al que queremos acceder, mientras que el segundo argumento es el nombre del modelo intermedio.

Se utilizarán las convenciones típicas de clave foránea de Eloquent al realizar las consultas de la relación. Si deseas personalizar las claves de la relación, puedes pasárselas como el tercer y cuarto argumento al

método `hasOneThrough`. El tercer argumento es el nombre de la clave foránea en el modelo intermedio. El cuarto argumento es el nombre de la clave foránea en el modelo final. El quinto argumento es la clave local, mientras que el sexto argumento es la clave local del modelo intermedio:

```
class Supplier extends Model
{
    /**
     * Get the user's history.
     */
    public function userHistory()
    {
        return $this->hasOneThrough(
            'App\History',
            'App\User',
            'supplier_id', // Foreign key on users table...
            'user_id', // Foreign key on history table...
            'id', // Local key on suppliers table...
            'id' // Local key on users table...
        );
    }
}
```

php

Tiene muchos a través de (hasManyThrough)

La relación "tiene-muchos-a-través-de" proporciona una abreviación conveniente para acceder a relaciones distantes por medio de una relación intermedia. Por ejemplo, un modelo `Country` podría tener muchos modelos `Post` a través de un modelo `User` intermedio. En este ejemplo, podrías traer todos los posts de un blog para un país dado. Vamos a buscar las tablas requeridas para definir esta relación:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

posts
```

php

```
id - integer
user_id - integer
title - string
```

Aunque los `posts` no contienen una columna `country_id`, la relación `hasManyThrough` proporciona acceso a los posts de un país por medio de `$country->posts`. Para ejecutar esta consulta, Eloquent inspecciona el `country_id` de la tabla intermedia `users`. Después de encontrar los ID de usuario que coincidan, serán usados para consultar la tabla `posts`.

Ahora que hemos examinado la estructura de la tabla para la relación, vamos a definirla en 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');  
    }  
}
```

php

El primer argumento pasado al método `hasManyThrough` es el nombre del modelo final que deseamos acceder, mientras que el segundo argumento es el nombre del modelo intermedio.

Las convenciones de clave foránea típicas de Eloquent serán usadas al momento de ejecutar las consultas de la relación. Si prefieres personalizar las claves de la relación, puedes pasarlo como tercer y cuarto argumentos del método `hasManyThrough`. El tercer argumento es el nombre de la clave foránea en el modelo intermedio. El cuarto argumento es el nombre de la clave foránea en el modelo final. El quinto argumento es la clave local, mientras el sexto argumento 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...
        );
    }
}
```

php

Relaciones polimórficas

Una relación polimórfica permite que el modelo objetivo pertenezca a más de un tipo de modelo usando una sola asociación.

Una a una (polimórfica)

Estructura de tabla

Una relación polimórfica de uno-a-uno es similar a una relación de uno-a-uno simple; sin embargo, el modelo objetivo puede pertenecer a más de un tipo de modelo en una sola asociación. Por ejemplo, un `Post` de un blog y un `User` pueden compartir una relación polimórfica con un modelo `Image`.

Usando una relación polimórfica de uno-a-uno te permite tener una sola lista de imágenes únicas que son usadas tanto los posts del blog como por las cuentas de los usuarios. Primero, vamos a examinar la estructura de la tabla:

```
posts
    id - integer
    name - string

users
    id - integer
    name - string
```

php

```
images
  id - integer
  url - string
  imageable_id - integer
  imageable_type - string
```

Observa las columnas `imageable_id` y `imageable_type` en la tabla `images`. La columna `imageable_id` contendrá el valor del ID del post o el usuario, mientras que la columna `imageable_type` contendrá el nombre de clase del modelo padre. La columna `imageable_type` es usada por Eloquent para determinar cuál "tipo" de modelo padre retornar al acceder a la relación `imageable`.

Estructura del modelo

A continuación, vamos a examinar las definiciones de modelo necesarias para construir esta relación:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Image extends Model  
{  
    /**  
     * Get the owning imageable model.  
     */  
    public function imageable()  
    {  
        return $this->morphTo();  
    }  
}  
  
class Post extends Model  
{  
    /**  
     * Get the post's image.  
     */  
    public function image()  
    {  
        return $this->morphOne('App\Image', 'imageable');  
    }  
}
```

```
}

class User extends Model
{
    /**
     * Get the user's image.
     */
    public function image()
    {
        return $this->morphOne('App\Image', 'imageable');
    }
}
```

Retornando la relación

Una vez que tu base de datos y modelos son definidos, puedes acceder a las relaciones mediante tus modelos. Por ejemplo, para retornar la imagen para un post, podemos usar la propiedad dinámica

`image` :

```
$post = App\Post::find(1);  
  
$image = $post->image;
```

Puedes también retornar el padre del modelo polimórfico accediendo al nombre del método que realiza la llamada a `morphTo`. En nuestro caso, éste es el método `imageable` en el modelo `Image`. Entonces, accederemos al método como una propiedad dinámica:

```
$image = App\Image::find(1);  
  
$imageable = $image->imageable;
```

La relación `imageable` en el modelo `Image` retornará ya sea una instancia de `Post` o `User`, dependiendo del tipo de modelo que posea la imagen.

Uno a muchos (Polimórfica)

Estructura de tabla

Una relación polimórfica de uno-a-muchos es similar a una relación de uno-a-muchos sencilla; sin embargo, el modelo objetivo puede pertenecer a más de un tipo de modelo en una sola asociación. Por ejemplo, imagina que usuarios de tu aplicación pueden comentar tanto en posts como en videos. Usando relaciones polimórficas, puedes usar una sola tabla `comments` para ambos escenarios. Primero, vamos a examinar la estructura de tabla requerida para esta relación:

```
posts                                php
  id - integer
  title - string
  body - text

videos
  id - integer
  title - string
  url - string

comments
  id - integer
  body - text
  commentable_id - integer
  commentable_type - string
```

Estructura de modelo

A continuación, vamos a examinar las definiciones de modelos necesarias para construir esta relación:

```
<?php                                php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the owning commentable model.
     */
    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');
    }
}

```

Retornando la relación

Una vez que tu base de datos y modelos son definidos, puedes acceder a las relaciones mediante tus modelos. Por ejemplo, para acceder a todos los comentarios de un post podemos usar la propiedad dinámica `comments` :

```

$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}

```

php

También puedes retornar al propietario de una relación polimórfica desde un modelo polimórfico accediendo al nombre del método que realiza la llamada a `morphTo`. En nuestro caso, éste es el método `commentable` en el modelo `Comment`. Así que, accederemos a dicho método como una propiedad dinámica:

```
$comment = App\Comment::find(1);  
  
$commentable = $comment->commentable;
```

php

La relación `commentable` en el modelo `Comment` retornará ya sea una instancia `Post` o `Video`, dependiendo de qué tipo de modelo es el propietario del comentario.

Muchos A Muchos (Polimórfica)

Estructura de tabla

Las relaciones polimórficas de muchos-a-muchos son un poco más complicadas que las relaciones `morphOne` y `morphMany`. Por ejemplo, un modelo `Post` de un blog y un modelo `Video` pueden compartir una relación polimórfica con un modelo `Tag`. Usando una relación polimórfica de muchos-a-muchos te permite tener una única lista de etiquetas que son compartidas a través de posts y videos. Primero, vamos a examinar la estructura de tabla:

```
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
```

php

Estructura del modelo

Seguidamente, estamos listos para definir las relaciones en el modelo. Ambos modelos `Post` y `Video` tendrán un método `tags` que ejecuta el método `morphToMany` en la clase base de Eloquent:

php

```
<?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 el inverso de la relación

A continuación, en el modelo `Tag`, debes definir un método para cada uno de sus modelos relacionados. Así, para este ejemplo, definiremos un método `posts` y un método `videos` :

php

```
<?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');
}
```

Obteniendo la relación

Una vez que tu tabla en la base de datos y modelos son definidos, puedes acceder las relaciones por medio de tus modelos. Por ejemplo, para acceder a todos los tags de un post, puedes usar la propiedad dinámica `tags` :

```
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

También puedes obtener el propietario de una relación polimórfica desde el modelo polimórfico accediendo al nombre del método que ejecutó la llamada a `morphedByMany`. En nuestro caso, estos son los métodos `posts` o `videos` en el modelo `Tag`. Así, accederemos a esos métodos como propiedades dinámicas:

```
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

Tipos polimórficos personalizados

Por defecto, Laravel usará el nombre completo de clase para almacenar el tipo del modelo relacionado. Por ejemplo, dado el ejemplo uno-a-muchos de arriba donde un `Comment` puede pertenecer a un `Post` o a un `Video`, el `commentable_type` por defecto será `App\Post` o `App\Video`, respectivamente. Sin embargo, puedes querer desacoplar tu base de datos de la estructura interna de tu aplicación. En dicho caso, puedes definir un "mapa de morfología (morph map)" para indicarle a Eloquent que use 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',  
]);
```

php

TIP

Al agregar un "morph map" a tu aplicación existente, cada valor de la columna de morfología `*_type` en tu base de datos que aún contenga una clase completamente calificada necesitará ser convertida a su nombre de "mapa".

Consultando relaciones

Ya que todos los tipos de relaciones Eloquent son definidas por medio de métodos, puedes ejecutar esos métodos para obtener una instancia de la relación sin ejecutar realmente las consultas de la relación. Además, todos los tipos de relaciones Eloquent también sirven como [constructores de consultas](#), permitiendo que continúes encadenando restricciones dentro de la consulta de la relación antes de ejecutar finalmente el SQL contra la base de datos.

Por ejemplo, imagina un sistema de blog en el cual un modelo `User` tiene muchos modelos `Post` 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');
```

php

```
    }  
}
```

Puedes consultar la relación `posts` y agregar limitaciones a la relación de la siguiente forma:

```
$user = App\User::find(1);  
  
$user->posts()->where('active', 1)->get();
```

php

Puedes usar cualquiera de los métodos de [constructor de consultas](#) y así que asegúrate de revisar la documentación del constructor de consultas para aprender sobre todos los métodos disponibles.

Encadenando cláusulas `orWhere` en relaciones

Como se demostró en el ejemplo superior, eres libre de agregar restricciones adicionales a las relaciones al momento de realizar peticiones. Sin embargo, ten cuidado al encadenar cláusulas `orWhere` a una relación, dado que las cláusulas `orWhere` serán agrupadas lógicamente en el mismo nivel que la restricción de la relación:

```
$user->posts()  
    ->where('active', 1)  
    ->orWhere('votes', '>=', 100)  
    ->get();  
  
// select * from posts  
// where user_id = ? and active = 1 or votes >= 100
```

php

En la mayoría de los casos, probablemente pretendes usar [grupos de restricciones](#) para agrupar lógicamente las comprobaciones condicionales entre parentesis:

```
use Illuminate\Database\Eloquent\Builder;  
  
$user->posts()  
    ->where(function (Builder $query) {  
        return $query->where('active', 1)  
            ->orWhere('votes', '>=', 100);  
    })  
    ->get();
```

php

```
// select * from posts  
// where user_id = ? and (active = 1 or votes >= 100)
```

Métodos de relación Vs. propiedades dinámicas

Si no necesitas agregar restricciones adicionales a una consulta de relación de Eloquent, puedes acceder a la relación como si fuera una propiedad. Por ejemplo, continuando con el uso de nuestros modelos de ejemplo `User` y `Post`, podemos acceder a todos los posts de un usuario como sigue:

```
$user = App\User::find(1);  
  
foreach ($user->posts as $post) {  
    //  
}
```

Las propiedades dinámicas son de "carga diferida (lazy loading)", lo que significa que cargarán solamente los datos de su relación cuando realmente accedas a ellas. Debido a esto, los desarrolladores con frecuencia usan [carga previa \(eager loading\)](#) para precargar las relaciones que ellos saben que serán accedidas después de cargar el modelo. La carga previa proporciona una reducción significativa en consultas SQL que deben ser ejecutadas para cargar las relaciones de un modelo.

Consultando la existencia de una relación

Cuando accedes a los registros de un modelo, puedes desear limitar sus resultados basados en la existencia de una relación. Por ejemplo, imagina que quieres obtener todos los posts de blog que tienen al menos un comentario. Para hacer eso, puedes pasar el nombre de la relación a los métodos `has` y `orHas`:

```
// Retrieve all posts that have at least one comment...  
$posts = App\Post::has('comments')->get();
```

También puedes especificar un operador y la cantidad para personalizar aún más la consulta:

```
// Retrieve all posts that have three or more comments...  
$App/posts = Post::has('comments', '>=', 3)->get();
```

Las instrucciones `has` anidadas también pueden ser construidas usando la notación de "punto". Por ejemplo, puedes obtener todos los posts que tienen al menos un comentario con votos:

```
// Retrieve posts that have at least one comment with votes...
$posts = Post::has('comments.votes')->get();
```

php

Incluso si necesitas más potencia, puedes usar los métodos `whereHas` y `orWhereHas` para poner condiciones "where" en tus consultas `has`. Estos métodos permiten que agregues restricciones personalizadas a una restricción de relación, tal como verificar el contenido de un comentario:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like foo%
$post = App\Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'foo%');
})->get();

// Retrieve posts with at least ten comments containing words like foo%
$post = App\Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'foo%');
}, '>=', 10)->get();
```

php

Consultando la ausencia de una relación

Al momento de acceder a los registros de un modelo, puedes desear limitar tus resultados en base a la ausencia de una relación. Por ejemplo, imagina que quieras obtener todos los posts de blogs que **no** tienen algún comentario. Para hacer eso, puedes pasar el nombre de la relación a los métodos

`doesntHave` y `orWhereDoesntHave`:

```
$posts = App\Post::doesntHave('comments')->get();
```

php

Incluso si necesitas más potencia, puedes usar los métodos `whereDoesntHave` y `orWhereDoesntHave` para poner condiciones "where" en tus consultas `doesntHave`. Estos métodos permiten que agregues restricciones personalizadas a una restricción de relación, tal como verificar el contenido de un comentario:

```
use Illuminate\Database\Eloquent\Builder;                                php

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Puedes usar notación "de puntos" para ejecutar una consulta contra una relación anidada. Por ejemplo, la siguiente consulta entregará todos los posts con comentarios de autores que no están vetados:

```
use Illuminate\Database\Eloquent\Builder;                                php

$posts = App\Post::whereDoesntHave('comments.author', function (Builder $query)
    $query->where('banned', 0);
)->get();
```

Consultando relaciones polimórficas

Para consultar la existencia de relaciones `MorphTo`, puedes usar el método `whereHasMorph` y sus métodos correspondientes:

```
use Illuminate\Database\Eloquent\Builder;                                php

// Retrieve comments associated to posts or videos with a title like foo...
$comments = App\Comment::whereHasMorph(
    'commentable',
    ['App\Post', 'App\Video'],
    function (Builder $query) {
        $query->where('title', 'like', 'foo%');
    }
)->get();

// Retrieve comments associated to posts with a title not like foo...
$comments = App\Comment::whereDoesntHaveMorph(
    'commentable',
    'App\Post',
    function (Builder $query) {
        $query->where('title', 'like', 'foo%');
    }
)->get();
```

Puedes usar el parametro `$type` para agregar diferentes restricciones dependiendo del modelo relacionado:

```
use Illuminate\Database\Eloquent\Builder;                                         php

$comments = App\Comment::whereHasMorph(
    'commentable',
    ['App\Post', 'App\Video'],
    function (Builder $query, $type) {
        $query->where('title', 'like', 'foo%');

        if ($type === 'App\Post') {
            $query->orWhere('content', 'like', 'foo%');
        }
    }
)->get();
```

En lugar de pasar un arreglo de posibles modelos polimorficos, puedes proporcionar un `*` como comodín y dejar que Laravel retorne todos los posibles tipos polimorficos desde la base de datos. Laravel ejecutará una solicitud adicional para poder realizar esta operación:

```
use Illuminate\Database\Eloquent\Builder;                                         php

$comments = App\Comment::whereHasMorph('commentable', '*', function (Builder $qu
    $query->where('title', 'like', 'foo%');
})->get();
```

Contando modelos relacionados

Si quieres contar el número de resultados de una relación sin cargarlos realmente puedes usar el método `withCount`, el cual coloca una columna `{relation}_count` en tus modelos resultantes. Por ejemplo:

```
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
```

```
echo $post->comments_count;  
}
```

Puedes agregar las "cuentas" para múltiples relaciones así como también agregar restricciones a las consultas:

```
$posts = Post::withCount(['votes', 'comments' => function ($query) {  
    $query->where('content', 'like', 'foo%');  
}])->get();  
  
echo $posts[0]->votes_count;  
echo $posts[0]->comments_count;
```

php

También puedes poner un alias al resultado de la cuenta de la relación, permitiendo múltiples cuentas en la misma relación:

```
use Illuminate\Database\Eloquent\Builder;  
  
$posts = App\Post::withCount([  
    'comments',  
    'comments as pending_comments_count' => function (Builder $query) {  
        $query->where('approved', false);  
    },  
])->get();  
  
echo $posts[0]->comments_count;  
  
echo $posts[0]->pending_comments_count;
```

php

Si estás combinando `withCount` con una instrucción `select`, asegúrate de llamar a `withCount` después del método `select`:

```
$posts = App\Post::select(['title', 'body'])->withCount('comments')->get();  
  
echo $posts[0]->title;  
echo $posts[0]->body;  
echo $posts[0]->comments_count;
```

php

Además, utilizando el método `loadCount`, puedes cargar un conteo de la relación después de que el modelo padre haya sido obtenido:

```
$book = App\Book::first();
$book->loadCount('genres');
```

php

Si necesitas establecer restricciones adicionales de consulta en la consulta de carga previa, puedes pasar un arreglo clave por las relaciones que deseas cargar. Los valores del arreglo deben ser instancias de `Closure` que reciben la instancia del generador de consulta:

```
$book->loadCount(['reviews' => function ($query) {
    $query->where('rating', 5);
}])
```

php

Carga previa (eager loading)

Al momento de acceder a las relaciones Eloquent como propiedades, los datos de la relación son "cargados diferidamente (lazy loading)". Esto significa que los datos de la relación no son cargados realmente hasta que primero accedas a la propiedad. Sin embargo, Eloquent puede "cargar previamente (eager loading)" las relaciones al mismo tiempo que consultas el modelo padre. La carga previa alivia el problema de la consulta N + 1. Para ilustrar el problema de la consulta N + 1, considera un modelo `Book` que está relacionado a `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');
    }
}
```

php

```
    }  
}
```

Ahora, vamos a obtener todos los libros y sus autores:

```
$books = App\Book::all();  
  
foreach ($books as $book) {  
    echo $book->author->name;  
}
```

php

Este ciclo ejecutará una consulta para obtener todos los libros en la tabla, después otra consulta para cada libro para obtener el autor. Así, si tenemos 25 libros, este ciclo debería ejecutar 26 consultas: 1 para el libro original y 25 consultas adicionales para obtener el autor de cada libro.

Afortunadamente, podemos usar la carga previa para reducir esta operación a solo 2 consultas. Al momento de consultar, puedes especificar cuáles relaciones deberían ser precargadas usando el método `with` :

```
$books = App\Book::with('author')->get();  
  
foreach ($books as $book) {  
    echo $book->author->name;  
}
```

php

Para esta operación, solo dos consultas serán ejecutadas:

```
select * from books  
  
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

php

Carga previa de múltiples relaciones

Algunas veces puedes necesitar la carga previa de varias relaciones diferentes en una operación única. Para hacer eso, pasa sólo los argumentos adicionales al método `with` :

```
$books = App\Book::with(['author', 'publisher'])->get();
```

php

Carga previa anidada

Para precargar relaciones anidadas, puedes usar la sintaxis de "punto". Por ejemplo, vamos a precargar todos los autores de los libros y todos los contactos personales del autor en una instrucción de Eloquent:

```
$books = App\Book::with('author.contacts')->get();
```

php

Eager Load anidado de relaciones `morphTo`

Si te gustaría hacer eager load de relaciones `morphTo`, así como de relaciones anidadas en varias entidades que podrían ser retornadas por dicha relación, puedes usar el método `with` en combinación con el método `morphWith` de la relación `morphTo`. Para ayudarte a ilustrar este método, vamos a considerar el siguiente método:

```
<?php  
  
use Illuminate\Database\Eloquent\Model;  
  
class ActivityFeed extends Model  
{  
    /**  
     * Get the parent of the activity feed record.  
     */  
    public function parentable()  
    {  
        return $this->morphTo();  
    }  
}
```

php

En este ejemplo, vamos a asumir que los modelos `Event`, `Photo` y `Post` podrían crear moelos `ActivityFeed`. Adicionalmente, vamos a asumir que los modelos `Event` pertenecen a una modelo `Calendar`, que los modelos `Photo` están asociados con modelos `Tag` y los modelos `Post` pertenecen a una modelo `Author`.

Usando estas definiciones de modelos y relaciones, podríamos retornar instancias del modelo `ActivityFeed` y hacer eager load de todos los modelos `parentable` y sus respectivas relaciones anidadas:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;  
  
$activities = ActivityFeed::query()  
    ->with(['parentable' => function (MorphTo $morphTo) {  
        $morphTo->morphWith([  
            Event::class => ['calendar'],  
            Photo::class => ['tags'],  
            Post::class => ['author'],  
        ]);  
    }])->get();
```

php

Cargando previamente columnas específicas

No siempre necesitas todas las columnas de las relaciones que estás obteniendo. Por esta razón, Eloquent te permite que especificar cuáles columnas de la relación te gustaría obtener:

```
$books = App\Book::with('author:id,name')->get();
```

php

Nota

Al momento de usar esta característica, siempre debes incluir la columna `id` en la lista de columnas que deseas obtener.

Para relaciones "tiene muchos" necesitas especificar tanto `id` como `foreign_key`

```
$books = App\Book::with('chapter:id,book_id,name')->get();
```

php

Nota

Al usar esta característica, siempre debes incluir la columna `id` y cualquier columna de clave foranea relevante en la lista de columnas que deseas retornar.

Carga previa por defecto

Algunas veces vas a querer cargar siempre algunas relaciones al retornar un modelo. Para lograr esto, puedes definir una propiedad `$with` en el modelo:

```
<?php
```

php

```
namespace App;  
use Illuminate\Database\Eloquent\Model;  
  
class Book extends Model  
{  
    /**  
     * Always load the related author when retrieving a book  
     * The relationships that should always be loaded.  
     *  
     * @var array  
     */  
    protected $with = ['author'];  
  
    /**  
     * Get the author that wrote the book.  
     */  
    public function author()  
    {  
        return $this->belongsTo('App\Author');  
    }  
}
```

Si te gustaría remover un elemento de la propiedad `$with` para una sola petición, puedes usar el método `without` :

```
$books = App\Book::without('author')->get();
```

php

Restringiendo cargas previas

Algunas veces puedes desear cargar previamente una relación, pero también especificar condiciones de consulta para la consulta de carga previa. Aquí está un ejemplo:

```
use Illuminate\Database\Eloquent\Builder;  
  
$users = App\User::with(['posts' => function ($query) {  
    $query->where('title', 'like', '%first%');  
}])->get();
```

php

En este ejemplo, Eloquent solamente precargará los posts donde la columna `title` del post contenga la palabra `first`. Puedes ejecutar otros métodos del [constructor de consulta](#) para personalizar más la operación de carga previa:

```
use Illuminate\Database\Eloquent\Builder; php

$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

Nota

Los métodos del constructor de consultas `limit` y `take` no se pueden usar al restringir las cargas previas.

Carga previa diferida (lazy eager loading)

Algunas veces puedes necesitar precargar una relación después de que el modelo padre ya ha sido obtenido. Por ejemplo, esto puede ser útil si necesitas decidir dinámicamente si se cargan modelos relacionados:

```
$books = App\Book::all(); php

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

Si necesitas establecer restricciones de consultas adicionales en la consulta de carga previa, puedes pasar un arreglo clave / valor con las relaciones que deseas cargar. Los valores del arreglo deberían ser instancias de `Closure`, las cuales reciben la instancia de consulta:

```
use Illuminate\Database\Eloquent\Builder; php

$author->load(['books' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

Para cargar una relación solo cuando aún no se ha cargado, usa el método `loadMissing` :

```
public function format(Book $book)
{
    $book->loadMissing('author');

    return [
        'name' => $book->name,
        'author' => $book->author->name,
    ];
}
```

php

Carga previa diferida anidada y `morphTo`

Si deseas cargar previamente una relación `morphTo`, así como relaciones anidadas en las diversas entidades que pueden ser devueltas por esa relación, puedes usar el método `loadMorph`.

Este método acepta el nombre de la relación `morphTo` como su primer argumento, y un arreglo de pares modelo / relación como su segundo argumento. Para ayudar a ilustrar este método, consideremos el siguiente modelo:

```
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

php

En este ejemplo, asumamos que los modelos `Event`, `Photo` y `Post` pueden crear modelos `ActivityFeed`. Además, supongamos que los modelos `Event` pertenecen a un modelo `Calendar`, los modelos `Photo` están asociados con los modelos `Tag` y los modelos `Post` pertenecen a un modelo `Author`.

Usando estas definiciones y relaciones de modelo, podemos recuperar instancias de modelo

`ActivityFeed` y cargar previamente todos los modelos `parentables` y sus respectivas relaciones anidadas:

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);

```

php

Insertando y actualizando modelos relacionados

El método save

Eloquent proporciona métodos convenientes para agregar nuevos modelos a las relaciones. Por ejemplo, quizás necesites insertar un nuevo `Comment` para un modelo `Post`. En lugar de establecer manualmente el atributo `post_id` en el `Comment`, puedes insertar el `Comment` directamente con el método `save` de la relación:

```
$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);
```

php

Observa que no accedimos a la relación `comments` como una propiedad dinámica. En su lugar, ejecutamos el método `comments` para obtener una instancia de la relación. El método `save` automáticamente agregará el valor `post_id` apropiado al nuevo modelo `Comment`.

Si necesitas guardar múltiples modelos relacionados, puedes usar el método `saveMany`:

```
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),

```

php

```
new App\Comment(['message' => 'Another comment.']),
]);
```

Guardando modelos y relaciones recursivamente

Si quieres hacer `save` a tu modelo y a todas sus relaciones asociadas, puedes usar el método `push`:

```
$post = App\Post::find(1);  
  
$post->comments[0]->message = 'Message';  
$post->comments[0]->author->name = 'Author Name';  
  
$post->push();
```

php

El método `create`

En adición a los métodos `save` y `saveMany`, también puedes usar el método `create`, el cual acepta un arreglo de atributos, crea un modelo y lo inserta dentro de la base de datos. Otra vez, la diferencia entre `save` y `create` es que `save` acepta una instancia de modelo Eloquent llena mientras `create` acepta un `array` PHP plano:

```
$post = App\Post::find(1);  
  
$comment = $post->comments()->create([  
    'message' => 'A new comment.',  
]);
```

php

TIP

Antes de usar el método `create`, asegurate de revisar la documentación sobre la [asignación masiva de atributos](#).

Puedes usar el método `createMany` para crear múltiples modelos relacionados:

```
$post = App\Post::find(1);  
  
$post->comments()->createMany([
```

php

```
[  
    'message' => 'A new comment.',  
,  
 [  
    'message' => 'Another new comment.',  
,  
]);
```

También puedes usar los métodos `findOrFail` , `firstOrFail` , `firstOrCreate` y `updateOrCreate` para [crear y actualizar modelos en relaciones](#).

Actualizar relación pertenece a (belongsTo)

Al momento de actualizar una relación `belongsTo` , puedes usar el método `associate` . Este método establecerá la clave foránea en el modelo hijo:

```
$account = App\Account::find(10);  
  
$user->account()->associate($account);  
  
$user->save();
```

Al momento de eliminar una relación `belongsTo` , puedes usar el método `dissociate` . Este método establecerá la clave foránea de la relación a `null` :

```
$user->account()->dissociate();  
  
$user->save();
```

Modelos predeterminados

Las relaciones `belongsTo` , `hasOne` , `hasOneThrough` y `morphOne` te permiten definir un modelo predeterminado que se devolverá si la relación dada es `null` . A este patrón se le conoce comúnmente como [patrón Null Object](#) y puede ayudar a quitar comprobaciones condicionales en tu código. En el ejemplo siguiente, la relación `user` devolverá un modelo `App\User` vacío si no hay un `user` adjunto a la publicación:

```
/*
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo('App\User')->withDefault();
}
```

php

Para llenar el modelo predeterminado con atributos, puedes pasar un arreglo 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, $post) {
        $user->name = 'Guest Author';
    });
}
```

php

Relaciones muchos a muchos

Adjuntando (attach) / Quitando (detach)

Eloquent también proporciona unos cuantos métodos helper para hacer que el trabajo con los modelos relacionados sea más conveniente. Por ejemplo, vamos a imaginar que un usuario tiene muchos roles y un rol puede tener muchos usuarios. Para adjuntar un rol a un usuario insertando un registro en la tabla intermedia que vincula los modelos, usa el método `attach` :

```
$user = App\User::find(1);  
  
$user->roles()->attach($roleId);
```

php

Al momento de adjuntar una relación a un modelo, también puedes pasar un arreglo de datos adicionales para ser insertados dentro de la tabla intermedia:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

php

Algunas veces puede ser necesario quitar un rol de un usuario. Para remover un registro de una relación de muchos-a-muchos, usa 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();
```

php

Por conveniencia, `attach` y `detach` también aceptan arreglos de IDs como entrada:

```
$user = App\User::find(1);  
  
$user->roles()->detach([1, 2, 3]);  
  
$user->roles()->attach([  
    1 => ['expires' => $expires],  
    2 => ['expires' => $expires],  
]);
```

php

Sincronizando asociaciones

También puedes usar el método `sync` para construir asociaciones muchos-a-muchos. El método `sync` acepta un arreglo de IDs para colocar en la tabla intermedia. Algunos IDs que no estén en el arreglo dado serán removidos de la tabla intermedia. Por tanto, después que esta operación se complete, solamente los IDs en el arreglo dado existirán en la tabla intermedia:

```
$user->roles()->sync([1, 2, 3]);
```

php

También puedes pasar valores adicionales de tabla intermedia con los IDs:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

php

Si no quieres desatar IDs existentes, puedes usar el método `syncWithoutDetaching`:

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

php

Alternar asociaciones

La relación de muchos-a-muchos también proporciona un método `toggle` el cual "alterna" el estado adjunto de los IDs dados. Si el ID está actualmente adjuntado, será removido. De igual forma, si está actualmente removido, será adjuntado:

```
$user->roles()->toggle([1, 2, 3]);
```

php

Guardando datos adicionales en una tabla pivot

Al momento de trabajar con una relación de muchos-a-muchos, el método `save` acepta un arreglo de atributos adicionales de tabla intermedia como su segundo argumento:

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

php

Actualizando un registro en una tabla pivot

Si necesitas actualizar una fila existente en tu tabla pivot, puedes usar el método

`updateExistingPivot`. Este método acepta la clave foránea del registro pivot y un arreglo de atributos para actualizar:

```
$user = App\User::find(1);
```

php

```
$user->roles()->updateExistingPivot($roleId, $attributes);
```

Tocando marcas de tiempo del padre

Cuando un modelo `belongsTo` o `belongsToMany` a otro modelo, tal como un `Comment` el cual pertenece a un `Post`, algunas veces es útil actualizar la marca de tiempo del padre cuando el modelo hijo es actualizado. Por ejemplo, cuando un modelo `Comment` es actualizado, puedes querer "tocar" automáticamente la marca de tiempo `updated_at` del `Post` que lo posee. Eloquent hace esto fácil. Simplemente agrega una propiedad `touches` conteniendo los nombres de las relaciones 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');  
    }  
}
```

php

Ahora, cuando actualices un `Comment`, el `Post` que lo posee tendrá su columna `updated_at` actualizada también, haciéndolo más conveniente para saber cuándo invalidar una caché del modelo `Post`:

```
$comment = App\Comment::find(1);  
  
$comment->text = 'Edit to this comment!';
```

php

```
$comment->save();
```

Eloquent: Colecciones

- [Introducción](#)
- [Métodos disponibles](#)
- [Colecciones personalizadas](#)

Introducción

Todos los conjuntos de multi-resultados retornados por Eloquent son instancias del objeto

`Illuminate\Database\Eloquent\Collection`, incluyendo los resultados obtenidos por medio del método `get` o accedidos por medio de una relación. El objeto de la colección Eloquent extiende la colección base de Laravel, así hereda naturalmente docenas de métodos usados para trabajar fluidamente con el arreglo subyacente de modelos de Eloquent.

Todas las colecciones tambien sirven como iteradores, permitiendo que iteres sobre ellas como si fueran simples arreglos de PHP:

```
$users = App\User::where('active', 1)->get();  
  
foreach ($users as $user) {  
    echo $user->name;  
}
```

Sin embargo, las colecciones son mucho más poderosas que los arreglos y exponen una variedad de mapeos / reduce operaciones que pueden ser encadenadas usando una interfaz intuitiva. Por ejemplo, vamos a remover todos los modelos inactivos y traeremos el primer nombre para cada usuario restante:

```
$users = App\User::all();  
  
$names = $users->reject(function ($user) {  
    return $user->active === false;  
})  
->map(function ($user) {  
    return $user->name;  
});
```

php

Nota

Mientras los métodos de colección de Eloquent devuelven una nueva instancia de una colección de Eloquent, los métodos `pluck`, `keys`, `zip`, `collapse`, `flatten` y `flip` devuelven una instancia de colección base. De igual forma, si una operación devuelve una colección que no contiene modelos Eloquent, será automáticamente convertida a una colección base.

Métodos Disponibles

La colección base

Todas las colecciones de Eloquent extienden el objeto de [colección de Laravel](#) base; sin embargo, heredan todos los métodos poderosos proporcionados por la clase de colección base:

Adicionalmente, la clase `Illuminate\Database\Eloquent\Collection` proporciona una serie de métodos para ayudarte a administrar tus colecciones de modelos. La mayoría de los métodos retornan instancias de `Illuminate\Database\Eloquent\Collection`; sin embargo, algunos métodos retornan una instancia base `Illuminate\Support\Collection`.

contains
diff
except
find
fresh
intersect
load
loadMissing

`modelKeys`

`makeVisible`

`makeHidden`

`only`

`unique`

```
contains($key, $operator = null, $value = null)
```

El método `contains` puede ser usado para determinar si una instancia de modelo dada es contenida por la colección. Este método acepta una clave primaria o una instancia de modelo:

```
$users->contains(1);
```

php

```
$users->contains(User::find(1));
```

```
diff($items)
```

El método `diff` retorna todos los modelos que no están presentes en la colección dada:

```
use App\User;
```

php

```
$users = $users->diff(User::whereIn('id', [1, 2, 3])->get());
```

```
except($keys)
```

El método `except` retorna todos los modelos que no tienen las claves primarias dadas:

```
$users = $users->except([1, 2, 3]);
```

php

```
find($key) {#collection-method .first-collection-method}
```

El método `find` encuentra un modelo que tienen una clave primaria dada. Si `$key` es una instancia de modelo, `find` intentará retornar un modelo que coincida con la clave primaria. Si `$key` es un arreglo de claves, `find` retornará todos los modelos que coincidan con las `$keys` usando `whereIn()`:

```
$users = User::all();
```

php

```
$user = $users->find(1);
```

`fresh($with = [])`

El método `fresh` retorna una instancia nueva de cada modelo en la colección desde la base de datos. Adicionalmente, cualquier relación especificada será cargada por adelantado:

```
$users = $users->fresh();
```

php

```
$users = $users->fresh('comments');
```

`intersect($items)`

El método `intersect` retorna todos los modelos que también están presentes en la colección dada:

```
use App\User;
```

php

```
$users = $users->intersect(User::whereIn('id', [1, 2, 3])->get());
```

`load($relations)`

El método `load` carga por adelantado las relaciones dadas para todos los modelos en la colección:

```
$users->load('comments', 'posts');
```

php

```
$users->load('comments.author');
```

`loadMissing($relations)`

El método `loadMissing` carga por adelantado las relaciones dadas para todos los modelos en la colección si las relaciones aún no han sido cargadas:

```
$users->loadMissing('comments', 'posts');
```

php

```
$users->loadMissing('comments.author');
```

`modelKeys()`

El método `modelKeys` retorna las claves primarias para todos los modelos en la colección:

```
$users->modelKeys();  
// [1, 2, 3, 4, 5]                                     php
```

`makeVisible($attributes)`

El método `makeVisible` hace visibles los atributos que normalmente están "ocultados" en cada modelo de la colección:

```
$users = $users->makeVisible(['address', 'phone_number']);          php
```

`makeHidden($attributes)`

El método `makeHidden` oculta los atributos que normalmente están "visibles" en cada modelo de la colección:

```
$users = $users->makeHidden(['address', 'phone_number']);          php
```

`only($keys)`

El método `only` retorna todos los modelos que tienen las claves primarias dadas:

```
$users = $users->only([1, 2, 3]);                                     php
```

`unique($key = null, $strict = false)`

El método `unique` retorna todos los modelos únicos en la colección. Cualquier modelo del mismo tipo con las mismas claves primarias que otro modelo en la colección es removido.

```
$users = $users->unique();                                         php
```

Colecciones personalizadas

Si necesitas usar un objeto `Collection` personalizado con tus propios métodos de extensión, puedes sobrescribir el método `newCollection` en tu 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);  
    }  
}
```

php

Una vez que has definido un método `newCollection`, recibirás una instancia de tu colección personalizada cada vez que Eloquent devuelva una instancia `Collection` de ese modelo. Si prefieres usar una colección personalizada para cada modelo en tu aplicación, deberías sobrescribir el método `newCollection` en una clase del modelo base que es extendida por todos tus modelos.

Eloquent: Mutators

- Introducción
- Accesadores y mutadores
 - Definiendo un accesador
 - Definiendo un mutador
- Mutadores de fecha
- Conversión de atributos
 - Conversión de arreglos y JSON
 - Conversión de fechas

Introducción

Los accesadores y mutadores permiten que des formato a los valores de atributos de Eloquent cuando los obtienes o estableces en las instancias de modelo. Por ejemplo, puede que te guste usar el [criptador de Laravel](#) para cifrar un valor mientras es almacenado en la base de datos y después descifrar automáticamente el atributo cuando accedes a él en un modelo de Eloquent.

Además de los accesadores y los mutadores personalizados, Eloquent también puede convertir automáticamente campos de fecha a instancias [Carbon](#) o incluso [convertir campos de texto a JSON](#).

Accesadores y mutadores

Definiendo un accesador

Para definir un accesador crea un método `getFooAttribute` en tu modelo, donde `Foo` es el nombre de la columna que deseas acceder en el formato Studly Case (Primera letra de cada palabra en mayúscula). En este ejemplo, definiremos un accesador para el atributo `first_name`. El accesador automáticamente será ejecutado por Eloquent al momento de intentar obtener el valor del atributo `first_name`:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class User extends Model  
{
```

php

```
/**
 * Get the user's first name.
 *
 * @param string $value
 * @return string
 */
public function getFirstNameAttribute($value)
{
    return ucfirst($value);
}
```

Como puedes ver, el valor original de la columna es pasado al accesador, permitiéndote manipular y devolver el valor. Para acceder al valor del accesador, puedes acceder al atributo `first_name` en una instancia del modelo:

```
$user = App\User::find(1);  
  
$firstName = $user->first_name;
```

También puedes usar accesadores para retornar nuevos valores computados de atributos existentes:

```
/**
 * Get the user's full name.
 *
 * @return string
 */
public function getFullNameAttribute()
{
    return "{$this->first_name} {$this->last_name}";
}
```

TIP

Si deseas que estos valores computados sean agregados a las representaciones de arreglo / JSON de tu modelo, [necesitarás adjuntarlos](#).

Definiendo un mutador

Para definir un mutador, define un método `setFooAttribute` en tu modelo, donde `Foo` es el nombre de la columna que deseas acceder en el formato Studly Case (Primera letra de cada palabra en mayúscula). Así, otra vez, vamos a definir un mutador para el atributo `first_name`. Este mutador será ejecutado automáticamente cuando intentamos establecer el valor del atributo `first_name` en el 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 que está siendo establecido en el atributo, permitiéndote manipular el valor y establecer el valor manipulado en la propiedad `$attributes` interna del modelo Eloquent. Así, por ejemplo, si intentamos establecer el atributo `first_name` como `Sally` :

```
$user = App\User::find(1);  
  
$user->first_name = 'Sally';
```

En este ejemplo, la función `setFirstNameAttribute` será ejecutada con el valor `Sally`. El mutador entonces aplicará la función `strtolower` al nombre y establecerá su valor resultante en el arreglo `$attributes` interno.

Mutadores de fecha

De forma predeterminada, Eloquent convertirá las columnas `created_at` y `updated_at` a instancias de [Carbon](#), la cual extiende la clase `DateTime` de PHP para proporcionar una variedad de métodos útiles. Puedes agregar atributos de fecha adicionales estableciendo la propiedad `$dates` de tu 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 = [  
        'seen_at',  
    ];  
}
```

php

TIP

Puedes desactivar las marcas de tiempo (timestamps) predeterminadas `created_at` y `updated_at` configurando la propiedad pública `$timestamps` de tu modelo en `false`.

Cuando una columna es considerada una fecha, puedes establecer su valor a una marca de tiempo UNIX, cadena de fecha (`Y-m-d`), cadena fecha-hora o una instancia `DateTime` / `Carbon` . El valor de la fecha será convertido y almacenado correctamente en tu base de datos:

```
$user = App\User::find(1);  
  
$user->deleted_at = now();  
  
$user->save();
```

php

Como se apreció anteriormente, al momento de obtener atributos que están listados en tu propiedad `$dates`, éstos serán automáticamente convertidos a instancias [Carbon](#), permitiendo que uses cualquiera de los métodos de Carbon en tus atributos:

```
$user = App\User::find(1);  
  
return $user->deleted_at->getTimestamp();
```

php

Formatos de fecha

De forma predeterminada, las marcas de tiempo son formateadas como `'Y-m-d H:i:s'`. Si necesitas personalizar el formato de marca de tiempo, establece la propiedad `$dateFormat` en tu modelo. Esta propiedad determina como los atributos de fecha son almacenados en la base de datos así como también su formato cuando el modelo es serializado a un arreglo 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';  
}
```

php

Conversión (casting) de atributos

La propiedad `$casts` en tu modelo proporciona un método conveniente de convertir atributos a tipos de datos comunes. La propiedad `$casts` debería ser un arreglo donde la clave es el nombre del atributo que está siendo convertido y el valor es el tipo al que deseas convertir la columna. Los tipos de conversión soportados son: `integer`, `real`, `float`, `double`, `decimal:<digits>`, `string`,

`boolean`, `object`, `array`, `collection`, `date`, `datetime`, and `timestamp`. Al convertir en `decimal`, debes definir el número de dígitos (`decimal:2`).

Para demostrar la conversión de atributos, vamos a convertir el atributo `is_admin`, el cual es almacenado en nuestra base de datos como un entero (`0` 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` será siempre convertido a un booleano cuando lo accedas, incluso si el valor subyacente es almacenado en la base de datos como un entero:

```
$user = App\User::find(1);  
  
if ($user->is_admin) {  
    //  
}
```

Conversión de arreglos y JSON

El tipo de conversión `array` es particularmente útil al momento de trabajar con columnas que son almacenadas como JSON serializado. Por ejemplo, si tu base de datos tiene un tipo de campo `JSON` o `TEXT` que contiene JSON serializado, agregar la conversión `array` a ese atributo deserializará automáticamente el atributo a un arreglo PHP cuando lo accedas en tu 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',  
    ];  
}
```

php

Una vez que la conversión es definida, puedes acceder al atributo `options` y será automáticamente deserializado desde JSON a un arreglo PHP. Cuando establezcas el valor del atributo `options`, el arreglo dado será automáticamente serializado de vuelta en JSON para almacenamiento:

```
$user = App\User::find(1);  
  
$options = $user->options;  
  
$options['key'] = 'value';  
  
$user->options = $options;  
  
$user->save();
```

php

Conversión de fechas

Al usar el tipo de conversión `date` o `datetime`, puedes especificar el formato de la fecha. Este formato se utilizará cuando el [modelo se serializa a un arreglo o JSON](#):

```
/**  
 * The attributes that should be cast to native types.
```

php

```
* @var array
*/
protected $casts = [
    'created_at' => 'datetime:Y-m-d',
];
```

Eloquent: Recursos API

- Introducción
- Generación de recursos
- Descripción general del concepto
 - Colecciones de recursos
- Escritura de recursos
 - Envoltura de datos
 - Paginación
 - Atributos condicionales
 - Relaciones condicionales
 - Añadiendo metadatos
- Respuestas de recursos

Introducción

Al crear una API, es posible que necesites una capa de transformación que se ubique entre tus modelos Eloquent y las respuestas JSON que realmente se devuelven a los usuarios de tu aplicación. Las clases de recursos de Laravel te permiten transformar tus modelos y colecciones de modelos de forma expresiva y sencilla en JSON.

Generación de recursos

Para generar un clase recurso, puedes usar el comando de Artisan `make:resource`. Por defecto, los recursos estará localizado en el directorio `app/Http/Resources` de tu aplicación. Los Recursos extiende de la clase `Illuminate\Http\Resources\Json\JsonResource`:

```
php artisan make:resource User
```

php

Colecciones de recurso

Además de generar recursos que transforman modelos individuales, puedes generar recursos que sean responsables de transformar colecciones de modelos. Esto permite que tu respuesta incluya enlaces y otra metainformación relevante para una colección completa de un recurso determinado.

Para crear una colección de recursos, debes utilizar la opción `--collection` al crear el recurso. O, incluir la palabra `Colección` en el nombre del recurso que le indicará a Laravel que debe crear un recurso de colección. Los recursos de colección extienden la clase

`Illuminate\Http\Resources\Json\ResourceCollection`:

```
php artisan make:resource Users --collection
```

php

```
php artisan make:resource UserCollection
```

Descripción general del concepto

TIP

Esta es una explicación general de recursos y colecciones de recursos. Te recomendamos que leas las otras secciones de esta documentación para obtener una comprensión más profunda de la personalización y el poder que te ofrecen los recursos.

Antes de sumergirse en todas las opciones disponibles para escribir recursos, primero analicemos cómo se utilizan los recursos dentro de Laravel. Una clase de recurso representa un modelo único que debe transformarse en una estructura JSON. Por ejemplo, aquí hay una clase de recurso `User` simple:

```
<?php
```

php

```

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

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

```

Cada clase de recurso define un método `toArray` que devuelve el arreglo de atributos que deben convertirse a JSON al enviar la respuesta. Observa que podemos acceder a las propiedades del modelo directamente desde la variable `$this`. Esto es porque la clase del recurso va a redirigir de manera automática el acceso de propiedades y métodos al modelo asignado. Una vez que se define el recurso, se puede devolver desde una ruta o controlador:

```

use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return new UserResource(User::find(1));
});

```

php

Colecciones de recurso

Si estás devolviendo una colección de recursos o una respuesta paginada, puedes usar el método `collection` al crear la instancia de recursos en tu ruta o controlador:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all());
});
```

php

Observa que esto no permite ninguna adición de metadatos que pueden necesitar ser retornados con la colección. Si deseas personalizar la respuesta de la colección de recursos, puedes crear un recurso dedicado para representar la colección:

```
php artisan make:resource UserCollection
```

php

Una vez que se ha generado la clase de colección de recursos, puedes definir fácilmente los metadatos que deben 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 $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

php

```
    ],
];
}
}
```

Después de definir tu colección de recursos, ésta la puedes devolver desde una ruta o controlador:

```
use App\Http\Resources\UserCollection;
use App\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

php

Preservando la colección de llaves

Cuando se retorna un recurso de colección desde una ruta, Laravel reinicia las llaves de la colección para que éstas estén en un simple orden numérico. Sin embargo, puedes añadir una propiedad

`preserveKeys`

a tu clase de recurso indicando si esta colección de llaves debería preservarse:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Indicates if the resource's collection keys should be preserved.
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

php

Cuando la propiedad `preserveKeys` es colocada en `true`, la colección de llaves será preservada:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all()->keyBy->id);
});
```

php

Personalización de la clase de recurso subyacente

Normalmente, la propiedad `$this->collection` de una colección de recursos se rellena automáticamente con el resultado de la asignación de cada elemento de la colección a su clase de recurso singular. Se asume que la clase de recurso singular es el nombre de clase de la colección sin la cadena `Collection` al final.

Por ejemplo, `UserCollection` intentará asignar las instancias de usuario dadas al recurso `User`. Para personalizar este comportamiento, puedes anular la propiedad `$collects` de tu colección de recursos:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * The resource that this resource collects.
     *
     * @var string
     */
    public $collects = 'App\Http\Resources\Member';
}
```

php

Escritura de recursos

TIP

Si no has leído la [descripción general del concepto](#), te recomendamos que lo hagas antes de continuar con esta documentación.

En esencia, los recursos son simples. Solo necesitan transformar un modelo dado en un arreglo. Por lo tanto, cada recurso contiene un método `toArray` que traduce los atributos de tu modelo en un arreglo amigable con la API que se puede devolver a sus usuarios:

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

Una vez que has definido un recurso, lo puedes devolver directamente desde una ruta o controlador:

```
use App\Http\Resources\User as UserResource;  
use App\User;  
  
Route::get('/user', function () {
```

```
        return new UserResource(User::find(1));
    });
}
```

Relaciones

Si deseas incluir recursos relacionados en tu respuesta, puedes agregarlos al arreglo devuelto por tu método `toArray`. En este ejemplo, usaremos el método `collection` del recurso `Post` para agregar las publicaciones del blog del usuario a la respuesta del recurso:

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

TIP

Si deseas incluir relaciones solo cuando ya se han cargado, consulte la documentación sobre [relaciones condicionales](#).

Colecciones de recurso

Si bien los recursos traducen un modelo único en un arreglo, las colecciones de recursos traducen una colección de modelos en un arreglo. No es absolutamente necesario definir una clase de colección de recursos para cada uno de los tipos de modelo ya que todos los recursos proporcionan un método `collection` para generar una colección de recursos "ad-hoc" sobre la marcha:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all());
});
```

php

Sin embargo, si necesitas personalizar los metadatos devueltos con la colección, será necesario definir una colección de recursos:

```
<?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 $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

php

Al igual que los recursos singulares, las colecciones de recursos se pueden devolver directamente desde las rutas o los controladores:

```
use App\Http\Resources\UserCollection;
use App\User;
```

php

```
Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

Envoltura de datos

Por defecto, tu recurso más externo está envuelto en una clave `data` cuando la respuesta del recurso se convierte a JSON. Entonces, por ejemplo, una respuesta típica de colección de recursos 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 deseas deshabilitar la envoltura del recurso más externo, puedes usar el método `withoutWrapping` en la clase de recurso base. Por lo general, debes llamar a este método desde su

`AppServiceProvider` u otro `proveedor de servicios` que se carga en cada solicitud a tu 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` solo afecta a la respuesta más externa y no eliminará las claves `data` que agregues manualmente a tus propias colecciones de recursos.

Envoltura de recursos anidados

Tienes total libertad para determinar cómo se envuelven las relaciones de tus recursos. Si deseas que todas las colecciones de recursos se envuelvan en una clave `data`, independientemente de su anidamiento, debes definir una clase de colección de recursos para cada recurso y devolver la colección dentro de una clave `data`.

Puedes que te estés preguntando si esto hará que tu recurso más externo se incluya en dos claves `data`. No te preocupes, Laravel nunca permitirá que tus recursos se envuelvan por error, por lo que no tienes que preocuparte por el nivel de anidamiento de la colección de recursos que estás transformando:

```
<?php  
  
namespace App\Http\Resources;
```

php

```
use Illuminate\Http\Resources\Json\ResourceCollection;

class CommentsCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param \Illuminate\Http\Request $request
     * @return array
     */
    public function toArray($request)
    {
        return ['data' => $this->collection];
    }
}
```

Envoltura de datos y paginación

Al devolver colecciones paginadas en una respuesta de recursos, Laravel ajustará tus datos de recursos en una clave `data` incluso si se ha llamado al método `withoutWrapping`. Esto se debe a que las respuestas paginadas siempre contienen claves `meta` y `links` con información sobre el 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 puedes pasar una instancia del paginador al método `collection` de un recurso o a una colección de recursos personalizada:

```
use App\Http\Resources\UserCollection;  
use App\User;  
  
Route::get('/users', function () {  
    return new UserCollection(User::paginate());  
});
```

Las respuestas paginadas siempre contienen claves `meta` y `links` con información sobre el 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
    }
}
```

Atributos condicionales

En ocasiones, es posible que desees incluir solo un atributo en una respuesta de recurso si se cumple una condición determinada. Por ejemplo, es posible que desee incluir solo un valor si el usuario actual es un "administrador". Laravel proporciona una variedad de métodos de ayuda para ayudarlo en esta situación. El método `when` se puede usar para agregar condicionalmente un atributo a una respuesta de recurso:

```
/**
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'secret' => $this->when(Auth::user()->isAdmin(), 'secret-value'),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

En este ejemplo, la clave `secret` solo se devolverá en la respuesta final del recurso si el método `isAdmin` del usuario autenticado devuelve `true`. Si el método devuelve `false`, la clave `secret` se eliminará de la respuesta del recurso por completo antes de que se envíe de nuevo al cliente. El método `when` te permite definir expresivamente tus recursos sin tener que recurrir a sentencias condicionales al construir el arreglo.

El método `when` también acepta un Closure como segundo argumento, lo que te permite calcular el valor resultante solo si la condición dada es `true`:

```
'secret' => $this->when(Auth::user()->isAdmin(), function () {
    return 'secret-value';
}),
```

php

Fusionar atributos condicionales

En ocasiones, es posible que tenga varios atributos que solo deben incluirse en la respuesta del recurso según la misma condición. En este caso, puede usar el método `mergeWhen` para incluir los atributos en la respuesta solo cuando la condición dada es `true`:

```
/*
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        $this->mergeWhen(Auth::user()->isAdmin(), [
            'first-secret' => 'value',
            'second-secret' => 'value',
        ]),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

php

Nuevamente, si la condición dada es `false`, estos atributos se eliminarán de la respuesta del recurso por completo antes de que se envíe al cliente.

Nota

El método `mergeWhen` no debe usarse dentro de arreglos que mezclen claves de cadenas de caracteres y claves numéricas. Además, no se debe utilizar dentro de arreglos con claves numéricas que no están ordenadas secuencialmente.

Relaciones condicionales

Además de cargar condicionalmente los atributos, puedes incluir condicionalmente relaciones en tus respuestas de recursos en función de si la relación ya se ha cargado en el modelo. Esto permite que tu controlador decida qué relaciones deben cargarse en el modelo y tu recurso puede incluirlas fácilmente solo cuando realmente se hayan cargado.

Fundamentalmente, esto hace que sea más fácil evitar los problemas de consulta "N + 1" dentro de tus recursos. El método `whenLoaded` puede usarse para cargar condicionalmente una relación. Para evitar cargar relaciones innecesariamente, este método acepta el nombre de la relación en lugar de la relación en sí:

```
/*
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'posts' => PostResource::collection($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 de la respuesta del recurso por completo antes de que se envíe al cliente.

Información de pivot condicional

Además de incluir condicionalmente la información de la relación en tus respuestas de recursos, puedes incluir condicionalmente datos de las tablas intermedias de relaciones de muchos a muchos utilizando el método `whenPivotLoaded`. El método `whenPivotLoaded` acepta el nombre de la tabla pivot como su primer argumento. El segundo argumento debe ser un Closure que defina el valor que se devolverá si la información pivot está disponible en el modelo:

```
/*
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'expires_at' => $this->whenPivotLoaded('role_user', function () {
            return $this->pivot->expires_at;
        }),
    ];
}
```

Si tu tabla intermedia utiliza un accesador distinto de `pivot`, puede usar el método `whenPivotLoadedAs`:

```
/*
 * Transform the resource into an array.
 *
 * @param \Illuminate\Http\Request $request
 * @return array
 */
public function toArray($request)
{
    return [
```

```
'id' => $this->id,  
'name' => $this->name,  
'expires_at' => $this->whenPivotLoadedAs('subscription', 'role_user', function ($query)  
    return $query->where('subscription_id', $this->id);  
,  
]);  
}  
];
```

Añadiendo metadatos

Algunos estándares de API de JSON requieren la adición de metadatos a tus respuestas de recursos y colecciones de recursos. Esto a menudo incluye cosas como `links` al recurso o recursos relacionados, o metadatos sobre el recurso en sí. Si necesitas devolver metadatos adicionales sobre un recurso, inclúyelos en tu método `toArray`. Por ejemplo, puedes incluir información de `link` al transformar una colección de recursos:

```
/** @return array */  
public function toArray($request)  
{  
    return [  
        'data' => $this->collection,  
        'links' => [  
            'self' => 'link-value',  
        ],  
    ];  
}
```

php

Al devolver metadatos adicionales de sus recursos, nunca tendrás que preocuparte por anular accidentalmente las claves `links` o `meta` que Laravel agrega automáticamente al devolver las respuestas paginadas. Cualquier `links` adicional que definas se fusionará con los enlaces proporcionados por el paginador.

Metadatos de nivel superior

A veces, es posible que deseas incluir solo ciertos metadatos con una respuesta de recurso si el recurso es el recurso más externo que se devuelve. Por lo general, esto incluye información meta sobre la respuesta como un todo. Para definir estos metadatos, agrega un método `with` a tu clase de recurso. Este método debería devolver un arreglo de metadatos que se incluirá con la respuesta del recurso solo cuando el recurso sea el recurso más externo que se está llamando:

```
<?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 $request  
     * @return array  
     */  
    public function toArray($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',  
            ],  
        ];  
    }  
}
```

Añadiendo metadatos al construir recursos

También puedes agregar datos de nivel superior al construir de instancias de recursos en tu ruta o controlador. El método `additional`, que está disponible en todos los recursos, acepta un arreglo de datos que deberían agregarse a la respuesta del recurso:

```
return (new UserCollection(User::all()->load('roles')))  
    ->additional(['meta' => [  
        'key' => 'value',  
    ]]);
```

php

Respuestas de Recurso

Como ya has leído, los recursos pueden devolverse directamente desde las rutas y los controladores:

```
use App\Http\Resources\User as UserResource;  
use App\User;  
  
Route::get('/user', function () {  
    return new UserResource(User::find(1));  
});
```

php

Sin embargo, a veces es posible que necesites personalizar la respuesta HTTP saliente antes de enviarla al cliente. Hay dos maneras de lograr esto. Primero, puedes encadenar el método `response` en el recurso. Este método devolverá una instancia de `Illuminate\Http\JsonResponse`, que te permite un control total de los encabezados de la respuesta:

```
use App\Http\Resources\User as UserResource;  
use App\User;  
  
Route::get('/user', function () {  
    return (new UserResource(User::find(1))  
        ->response()  
        ->header('X-Value', 'True');  
});
```

php

Alternativamente, puedes definir un método `withResponse` dentro del propio recurso. Este método se llamará cuando el recurso se devuelva como el recurso más externo en una respuesta:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

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

    /**
     * Customize the outgoing response for the resource.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Illuminate\Http\Response $response
     * @return void
     */
    public function withResponse($request, $response)
    {
        $response->header('X-Value', 'True');
    }
}
```

Eloquent: Serialización

- Introducción
- Serializando modelos y colecciones
 - Serializando a arreglos
 - Serializando a JSON
- Ocultando atributos de JSON
- Añadiendo valores a JSON
- Serialización de fechas

Introducción

Al momento de construir APIs JSON, con frecuencia necesitas convertir tus modelos y relaciones a arreglos o JSON. Eloquent incluye métodos convenientes para hacer estas conversiones, también como controlar cuáles atributos están incluidos en tus serializaciones.

Serializando modelos y colecciones

Serializando a arreglos

Para convertir un modelo y sus relaciones cargadas a un arreglo, debes usar el método `toArray`. Este método es recursivo, ya que todos los atributos y todas las relaciones (incluyendo las relaciones de relaciones) serán convertidas a arreglos:

```
$user = App\User::with('roles')->first();  
  
return $user->toArray();
```

Para convertir solo los atributos de un modelo a un arreglo, usa el método `attributedToArray`:

```
$user = App\User::first();
```

```
return $user->attributesToArray();
```

También puedes convertir colecciones completas de modelos en arreglos:

```
$users = App\User::all();  
  
return $users->toArray();
```

php

Para convertir únicamente los atributos de un modelo a arreglo, usa el método `attributesToArray`:

```
$user = App\User::first();  
  
return $user->attributesToArray();
```

php

Serializando a JSON

Para convertir un modelo a JSON, deberías usar el método `toJson`. Igual que `toArray`, el método `toJson` es recursivo, así todos los atributos y relaciones serán convertidas a JSON. También puedes especificar las opciones de codificación JSON [soportadas por PHP](#):

```
$user = App\User::find(1);  
  
return $user->toJson();  
  
return $user->toJson(JSON_PRETTY_PRINT);
```

php

Alternativamente, puedes convertir un modelo o colección en una cadena, la cual ejecutará automáticamente el método `toJson` sobre el modelo o colección:

```
$user = App\User::find(1);  
  
return (string) $user;
```

php

Debido a que los modelos y colecciones son convertidos a JSON al momento de conversión a una cadena, puedes devolver objetos de Eloquent directamente desde las rutas o controladores de tu aplicación:

```
Route::get('users', function () {
    return App\User::all();
});
```

php

Relaciones

Cuando un modelo de Eloquent es convertido a JSON, las relaciones que sean cargadas serán incluidas automáticamente como atributos en el objeto JSON. Además, aunque los métodos de relación de Eloquent sean definidos usando "camel case", un atributo JSON de la relación en su lugar se verá como "snake case".

Ocultando Atributos de JSON

Algunas veces puedes querer limitar los atributos, tales como contraseñas, que están incluidos en la representación de arreglo o JSON de tu modelo. Para hacer eso, agrega una propiedad `$hidden` en tu 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'];
}
```

php

Nota

Al momento de ocultar relaciones, usa el nombre de método de la relación.

Alternativamente, puedes usar la propiedad `visible` para definir una lista blanca de atributos que deberían ser incluidos en la representación de arreglo y JSON de tu modelo. Todos los demás atributos estarán ocultos cuando el modelo sea convertido a un arreglo 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'];  
}
```

php

Modificando la visibilidad de atributos temporalmente

Si prefieres hacer visible algunos atributos típicamente ocultos en una instancia de modelo dado, puedes usar el método `makeVisible`. El método `makeVisible` devuelve la instancia de modelo para encadenar métodos de forma conveniente:

```
return $user->makeVisible('attribute')->toArray();
```

php

De igual manera, si prefieres ocultar algunos atributos típicamente visibles en una instancia de modelo dado, puedes usar el método `makeHidden`.

```
return $user->makeHidden('attribute')->toArray();
```

php

Añadiendo Valores a JSON

Ocasionalmente, al momento de convertir modelos a un arreglo o JSON, puedes querer agregar atributos que no tienen una columna correspondiente en tu base de datos. Para hacer eso, primero define un

accesador para el valor:

```
<?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';  
    }  
}
```

php

Después de crear el accesador, agrega el nombre del atributo a la propiedad `appends` en el modelo. Nota que los nombres de atributo son referenciados típicamente en "snake_case", aun cuando el accesador sea definido 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'];  
}
```

php

Una vez que el atributo ha sido agregado a la lista `appends`, será incluido en ambas representaciones de arreglo y JSON del modelo. Los atributos en el arreglo `appends` también respetarán las configuraciones `visible` y `hidden` configuradas en el modelo.

Añadiendo en tiempo de ejecución

Puedes indicar una única instancia de modelo que agregue atributos utilizando el método `append`. También usar el método `setAppends` para sobrescribir el arreglo completo de propiedades adjuntadas para una instancia de un modelo dado:

```
return $user->append('is_admin')->toArray();  
  
return $user->setAppends(['is_admin'])->toArray();
```

php

Serialización de Fecha

Personalizar el formato de la fecha por atributo

Puedes personalizar el formato de serialización de atributos de fecha de Eloquent individuales especificando el formato de la fecha en la [declaración de la conversión](#):

```
protected $casts = [  
    'birthday' => 'date:Y-m-d',  
    'joined_at' => 'datetime:Y-m-d H:00',  
];
```

php

Pruebas: Primeros Pasos

- [Introducción](#)

- Entorno
- Creando y ejecutando pruebas

Introducción

Laravel está construido pensando en las pruebas. De hecho, el soporte para pruebas con PHPUnit es incluido de forma predeterminada y un archivo `phpunit.xml` ya está configurado para tu aplicación. El framework también viene con métodos de ayuda convenientes que permiten que pruebes tus aplicaciones de forma expresiva.

De forma predeterminada, el directorio `tests` de tu aplicación contiene dos directorios: `Feature` y `Unit`. Las pruebas unitarias (Unit) son pruebas que se enfocan en una muy pequeña porción aislada de tu código. De hecho, la mayoría de las pruebas unitarias se enfocan probablemente en un solo método. Las pruebas funcionales (Feature) pueden probar una porción más grande de tu código, incluyendo la forma en la que varios objetos interactúan entre sí e incluso una solicitud HTTP completa para un endpoint de JSON.

Un archivo `ExampleTest.php` es proporcionado en ambos directorios de prueba `Feature` y `Unit`. Después de instalar una nueva aplicación de Laravel, ejecuta `phpunit` en la línea de comandos para ejecutar tus pruebas.

Entorno

Al momento de ejecutar las pruebas por medio de `phpunit`, Laravel establecerá automáticamente el entorno de configuración a `testing` debido a las variables de entorno definidas en el archivo `phpunit.xml`. Laravel también configura automáticamente la sesión y cache del manejador `array` al momento de ejecutar las pruebas, lo que significa que ninguna sesión o cache de datos será conservada mientras las pruebas son ejecutadas.

Eres libre de definir otros valores de configuración del entorno de pruebas cuando sea necesario. Las variables de entorno `testing` pueden ser configuradas en el archivo `phpunit.xml`, pero asegúrate de limpiar tu cache de configuración usando el comando Artisan `config:clear` antes de ejecutar tus pruebas!

Además, puedes crear un archivo `.env.testing` en la raíz de tu proyecto. Este archivo anulará el archivo `.env` cuando ejecute las pruebas PHPUnit o cuando ejecute los comandos de Artisan con la opción `--env = testing`.

Creando y ejecutando pruebas

Para crear un nuevo caso de prueba, usa el comando 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
```

php

Una vez que la prueba ha sido generada, puedes definir métodos de pruebas como lo harías normalmente usando PHPUnit. Para ejecutar tus pruebas, ejecuta el comando `phpunit` desde tu terminal:

```
<?php
```

php

```
namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

Nota

Si defines tus propios métodos `setUp` / `tearDown` dentro de una clase de prueba, asegúrate de ejecutar los respectivos `parent::setUp()` / `parent::tearDown()` métodos en la clase padre.

Pruebas HTTP

- Introducción
 - Personalizando encabezados de solicitud
 - Depurando respuestas
- Sesión y autenticación
- Probando APIs JSON
- Probando subidas de archivos
- Aserciones disponibles
 - Aserciones de respuesta
 - Aserciones de autenticación

Introducción

Laravel proporciona una API muy fluida para hacer solicitudes HTTP a tu aplicación y examinar la salida. Por ejemplo, echemos un vistazo a la prueba definida 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.  
     */
```

php

```
* @return void
*/
public function testBasicTest()
{
    $response = $this->get('/');

    $response->assertStatus(200);
}
}
```

El método `get` simula una solicitud `GET` dentro de la aplicación, mientras que el método `assertStatus` comprueba que la respuesta devuelta debería tener el código de estado HTTP dado. Además de esta sencilla aserción, Laravel también contiene una variedad de aserciones para inspeccionar de la respuesta los encabezados, contenidos, estructura JSON y más.

Personalizando encabezados de solicitud

Puedes usar el método `withHeaders` para personalizar los encabezados de la solicitud antes que sean enviados a la aplicación. Esto permitirá que agregues algunos encabezados personalizados de tu preferencia 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,
            ]);
    }
}
```

```
    }  
}
```

TIP

El middleware CSRF es automáticamente deshabilitado cuando se ejecutan las pruebas.

Depurando respuestas

Luego de hacer una solicitud de prueba a tu aplicación, los métodos `dump` y `dumpHeaders` pueden ser usados para examinar y depurar el contenido de la respuesta:

```
<?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->dumpHeaders();  
        $response->dump();  
    }  
}
```

Sesión y autenticación

Laravel proporciona varias funciones helper para trabajar con la sesión durante las pruebas HTTP. Primero, puedes colocar los datos de la sesión en un arreglo dado usando el método `withSession`.

Esto es útil para cargar la sesión con los datos antes de realizar una solicitud a tu aplicación:

```
<?php  
  
class ExampleTest extends TestCase  
{  
    public function testApplication()  
    {  
        $response = $this->withSession(['foo' => 'bar'])  
                  ->get('/');  
    }  
}
```

php

Un uso común de la sesión es para mantener el estado del usuario autenticado. El método helper `actingAs` proporciona una forma sencilla de autenticar un usuario dado como el usuario actual. Por ejemplo, podemos usar un `model factory` para generar y autenticar 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('/');  
    }  
}
```

php

También puedes especificar que "guard" debe ser usado para autenticar el usuario dado al pasar el nombre del guard como segundo argumento del método `actingAs`:

```
$this->actingAs($user, 'api')
```

php

Probando APIs JSON

Laravel también proporciona varios helpers para probar APIs JSON y sus respuestas. Por ejemplo, los métodos `json`, `get`, `post`, `put`, `patch`, `delete` y `option` pueden ser usados para hacer solicitudes con varios verbos HTTP. También puedes pasar datos y encabezados fácilmente a estos métodos. Para empezar, vamos a escribir una prueba para hacer una solicitud `POST` a `/user` y comprobar que los datos esperados fueron devueltos:

```
<?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,  
            ]);  
    }  
}
```

TIP

El método `assertJson` convierte la respuesta a un arreglo y utiliza `PHPUnit::assertArraySubset` para verificar que el arreglo dado exista dentro de la respuesta JSON devuelta por la aplicación. Así, si hay otras propiedades en la respuesta JSON, esta prueba aún pasará siempre y cuando el fragmento dado esté presente.

Verificando una coincidencia JSON exacta

Si prefieres verificar que el arreglo dado esté contenido **exactamente** en la respuesta JSON devuelta por la aplicación, deberías 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,  
            ]);  
    }  
}
```

php

Probando subidas de archivos

La clase `Illuminate\Http\UploadedFile` proporciona un método `fake` el cual puede ser usado para generar archivos de prueba o imágenes para prueba. Esto, combinado con el método `fake` de la clase facade `Storage` simplifica grandemente la prueba de subidas de archivos. Por ejemplo, puedes combinar estas 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;
```

php

```
class ExampleTest extends TestCase
{
    public function testAvatarUpload()
    {
        Storage::fake('avatars');

        $file = UploadedFile::fake()->image('avatar.jpg');

        $response = $this->json('POST', '/avatar', [
            'avatar' => $file,
        ]);

        // Assert the file was stored...
        Storage::disk('avatars')->assertExists($file->hashName());

        // Assert a file does not exist...
        Storage::disk('avatars')->assertMissing('missing.jpg');
    }
}
```

Personalización de archivo fake

Al momento de crear archivos usando el método `fake`, puedes especificar el ancho, la altura y el tamaño de la imagen con el propósito de probar mejor tus reglas de validación:

```
UploadedFile::fake()->image('avatar.jpg', $width, $height)->size(100);
```

php

Además de crear imágenes, puedes crear archivos de cualquier otro tipo usando el método `create`:

```
UploadedFile::fake()->create('document.pdf', $sizeInKilobytes);
```

php

Aserciones disponibles

Aserciones de respuesta

Laravel proporciona una variedad de métodos de aserción personalizados para tus pruebas [PHPUnit](#). Estas aserciones pueden ser accedidas en la respuesta que es retornada por los métodos de prueba

json , get , post , put y delete :

assertCookie	assertJsonMissingExact	assertSessionHasInput
assertCookieExpired	assertJsonMissingValidationErrors	assertSessionHasAll
assertCookieNotExpired	assertJsonStructure	assertSessionHasErrors
assertCookieMissing	assertJsonValidationErrors	assertSessionHasErrorsIn
assertDontSee	assertLocation	assertSessionHasNoErrors
assertDontSeeText	assertNotFound	assertSessionDoesntHaveErrors
assertExactJson	assertOk	assertSessionMissing
assertForbidden	assertPlainCookie	assertStatus
assertHeader	assertRedirect	assertSuccessful
assertHeaderMissing	assertSee	assertSuccessful
assertJson	assertSeeInOrder	assertViewHas
assertJsonCount	assertSeeText	assertViewHasAll
assertJsonFragment	assertSeeTextInOrder	assertViewIs
assertJsonMissing	assertSessionHas	assertViewMissing

assertCookie

Comprueba que la respuesta contenga el cookie dado:

```
$response->assertCookie($cookieName, $value = null);
```

php

assertCookieExpired

Comprueba que la respuesta contenga el cookie dado y que esté vencido:

```
$response->assertCookieExpired($cookieName);
```

php

assertCookieNotExpired

Comprueba que la respuesta contenga la cookie dada y que no haya expirado:

```
$response->assertCookieNotExpired($cookieName);
```

php

assertCookieMissing

Comprueba que la respuesta no contenga el cookie dado:

```
$response->assertCookieMissing($cookieName);
```

php

assertDontSee

Comprueba que la cadena dada no esté contenida dentro de la respuesta:

```
$response->assertDontSee($value);
```

php

assertDontSeeText

Comprueba que la cadena dada no esté contenida dentro del texto de la respuesta:

```
$response->assertDontSeeText($value);
```

php

assertExactJson

Comprueba que la respuesta contenga una coincidencia exacta de los datos JSON dados:

```
$response->assertExactJson(array $data);
```

php

assertForbidden

Comprueba que la respuesta tenga un código de estado "prohibido":

```
$response->assertForbidden();
```

php

assertHeader

Comprueba que el encabezado dado esté presente en la respuesta:

```
$response->assertHeader($headerName, $value = null);
```

php

assertHeaderMissing

Comprueba que el encabezado dado no esté presente en la respuesta:

```
$response->assertHeaderMissing($headerName);
```

php

assertJson

Comprueba que la respuesta contenga los datos JSON dados:

```
$response->assertJson(array $data);
```

php

assertJsonCount

Comprueba que la respuesta JSON tenga un arreglo con el número esperado de elementos en la llave dada:

```
$response->assertJsonCount($count, $key = null);
```

php

assertJsonFragment

Comprueba que la respuesta contenga el fragmento JSON dado:

```
$response->assertJsonFragment(array $data);
```

php

assertJsonMissing

Comprueba que la respuesta no contenga el fragmento JSON dado:

```
$response->assertJsonMissing(array $data);
```

php

assertJsonMissingExact

Comprueba que la respuesta no contenga el fragmento exacto JSON:

```
$response->assertJsonMissingExact(array $data);
```

php

assertJsonMissingValidationErrors

Comprueba que la respuesta no contenga errores de validación JSON para la llaves dadas:

```
$response->assertJsonMissingValidationErrors($keys);
```

php

assertJsonStructure

Comprueba que la respuesta tenga una estructura JSON dada:

```
$response->assertJsonStructure(array $structure);
```

php

assertJsonValidationErrors

Comprueba que la respuesta tenga los errores de validación JSON dados:

```
$response->assertJsonValidationErrors(array $data);
```

php

assertLocation

Comprueba que la respuesta tenga el valor URI dado en el encabezado `Location` :

```
$response->assertLocation($uri);
```

php

assertNotFound

Comprueba que la respuesta tenga un código de estado "no encontrado":

```
$response->assertNotFound();
```

php

assertOk

Comprueba que la respuesta tenga un código de estado 200:

```
$response->assertOk();
```

php

assertPlainCookie

Comprueba que la respuesta contenga el cookie dado (descriptado):

```
$response->assertPlainCookie($cookieName, $value = null);
```

php

assertRedirect

Comprueba que la respuesta es una redirección a una URI dada:

```
$response->assertRedirect($uri);
```

php

assertSee

Comprueba que la cadena dada esté contenida dentro de la respuesta:

```
$response->assertSee($value);;
```

php

assertSeeInOrder

Comprueba que las cadenas dadas estén en orden dentro de la respuesta:

```
$response->assertSeeInOrder(array $values);
```

php

assertSeeText

Comprueba que la cadena dada esté contenida dentro del texto de la respuesta:

```
$response->assertSeeText($value);
```

php

assertSeeTextInOrder

Comprueba que las cadenas dadas estén en orden dentro del texto de respuesta:

```
$response->assertSeeTextInOrder(array $values);
```

php

assertSessionHas

Comprueba que la sesión contenga la porción dada de datos:

```
$response->assertSessionHas($key, $value = null);
```

php

assertSessionHasInput

Comprueba que la sesión tiene un valor dado en los datos del arreglo proporcionado:

```
$response->assertSessionHasInput($key, $value = null);
```

php

assertSessionHasAll

Comprueba que la sesión tenga una lista dada de valores:

```
$response->assertSessionHasAll(array $data);
```

php

assertSessionHasErrors

Comprueba que la sesión contenga un error para el campo dado:

```
$response->assertSessionHasErrors(array $keys, $format = null, $errorBag = 'defa
```

php

assertSessionHasErrorsIn

Comprueba que la sesión tenga los errores dados:

```
$response->assertSessionHasErrorsIn($errorBag, $keys = [], $format = null);
```

php

assertSessionHasNoErrors

Comprueba que la sesión no contenga errores:

```
$response->assertSessionHasNoErrors();
```

php

assertSessionDoesntHaveErrors

Comprueba que la sesión no contenga errores para las llaves dadas:

```
$response->assertSessionDoesntHaveErrors($keys = [], $format = null, $errorBag =
```

assertSessionMissing

Comprueba que la sesión no contenga la llave dada:

```
$response->assertSessionMissing($key);
```

assertStatus

Comprueba que la respuesta tenga un código dado:

```
$response->assertStatus($code);
```

assertSuccessful

Comprueba que la respuesta tenga un código de estado de éxito (200):

```
$response->assertSuccessful();
```

assertUnauthorized

Comprueba que la respuesta tiene un código de estado sin autorización (401):

```
$response->assertUnauthorized();
```

assertViewHas

Comprueba que la vista de la respuesta dada contiene los valores indicados:

```
$response->assertViewHas($key, $value = null);
```

php

assertViewHasAll

Comprueba que la vista de la respuesta tiene una lista de datos:

```
$response->assertViewHasAll(array $data);
```

php

assertViewIs

Comprueba que la vista dada fue retornada por la ruta:

```
$response->assertViewIs($value);
```

php

assertViewMissing

Comprueba que a la vista de la respuesta le está faltando una porción de datos enlazados:

```
$response->assertViewMissing($key);
```

php

Aserciones de autenticación

Laravel también proporciona una variedad de aserciones relacionadas con la autenticación para tus pruebas [PHPUnit](#):

Método	Descripción
<pre>\$this->assertAuthenticated(\$guard = null);</pre>	Comprueba que el usuario está autenticado.
<pre>\$this->assertGuest(\$guard = null);</pre>	Comprueba que el usuario no está autenticado.
<pre>\$this->assertAuthenticatedAs(\$user, \$guard = null);</pre>	Comprueba que el usuario dado está autenticado.

Método	Descripción
<code>\$this->assertCredentials(array \$credentials, \$guard = null);</code>	Comprueba que las credenciales dadas son válidas.
<code>\$this->assertInvalidCredentials(array \$credentials, \$guard = null);</code>	Comprueba que las credenciales dadas no son válidas.

Pruebas de consola

- [Introducción](#)
- [Esperando entrada / salida](#)

Introducción

Además de simplificar las pruebas de HTTP, Laravel proporciona una API simple para probar las aplicaciones de consola que solicitan información al usuario.

Esperando entrada / salida

Laravel te permite "simular" (mock) fácilmente la entrada de datos por parte del usuario mediante la consola utilizando el método `expectsQuestion`. Además, puedes especificar el código de salida y el texto que esperas que genere el comando de la consola utilizando los métodos `assertExitCode` y `expectsOutput`. Por ejemplo, considera el siguiente comando de consola:

```
php
Artisan::command('question', function () {
    $name = $this->ask('What is your name?');

    $language = $this->choice('Which language do you program in?', [
```

```
'PHP',
'Ruby',
'Python',
]);

$this->line('Your name is '.$name.' and you program in '.$language.'.');
});
```

Puedes probar este comando con la siguiente prueba que utiliza los métodos

`expectsQuestion` , `expectsOutput` y `assertExitCode` :

```
/*
 * Test a console command.
 *
 * @return void
 */
public function testConsoleCommand()
{
    $this->artisan('question')
        ->expectsQuestion('What is your name?', 'Taylor Otwell')
        ->expectsQuestion('Which language do you program in?', 'PHP')
        ->expectsOutput('Your name is Taylor Otwell and you program in PHP.')
        ->assertExitCode(0);
}
```

Laravel Dusk

- [Introducción](#)
- [Instalación](#)
 - [Administrando las instalaciones de ChromeDriver](#)
 - [Usando otros navegadores](#)

- Primeros pasos
 - Generando pruebas
 - Ejecutar pruebas
 - Manejo de entorno
 - Creando navegadores
 - Macros de navegador
 - Autenticación
 - Migraciones de base de datos
- Interactuando con elementos
 - Selectores de Dusk
 - Haciendo clic en enlaces
 - Texto, valores y atributos
 - Usando formularios
 - Adjuntando archivos
 - Usando el teclado
 - Usando el ratón
 - Diálogos de JavaScript
 - Alcance de selectores
 - Esperando por elementos
 - Haciendo aserciones de Vue
- Aserciones disponibles
- Páginas
 - Generando páginas
 - Configurando páginas
 - Visitando páginas
 - Selectores abreviados
 - Métodos de página
- Componentes
 - Generando componentes
 - Usando componentes
- Integración continua
 - CircleCI
 - Codeship
 - Heroku CI
 - Travis CI

Introducción

Laravel Dusk proporciona una API de automatización y prueba para navegador expresiva y fácil de usar. De forma predeterminada, Dusk no requiere que instales JDK o Selenium en tu computador. En su lugar, Dusk usa una instalación de [ChromeDriver](#) independiente. Sin embargo, siéntete libre de utilizar cualquier otro driver compatible con Selenium que deseas.

Instalación

Para empezar, debes agregar la dependencia de Composer `laravel/dusk` a tu proyecto:

```
composer require --dev laravel/dusk
```

php

Nota

Si estás registrando manualmente el proveedor de servicio de Dusk, **nunca** deberías registrarlo en tu entorno de producción, ya que hacerlo así podría conducir a que usuarios arbitrarios sean capaces de autenticarse en tu aplicación.

Después de la instalación del paquete Dusk, ejecuta el comando Artisan `dusk:install` :

```
php artisan dusk:install
```

php

Un directorio `Browser` será creado dentro de tu directorio `tests` y contendrá una prueba de ejemplo. Seguido, establece la variable de entorno `APP_URL` en tu archivo `.env`. Este valor debería coincidir con la URL que uses para acceder a tu aplicación en un navegador.

Para ejecutar tus pruebas, usa el comando de Artisan `dusk`. El comando `dusk` acepta cualquier argumento que también sea aceptado por el comando `phpunit` :

```
php artisan dusk
```

php

Si tuviste fallos en las pruebas la última vez que se ejecutó el comando `dusk`, puedes ahorrar tiempo volviendo a ejecutar las pruebas fallidas usando el comando `dusk: fail` :

```
php artisan dusk:fails
```

php

Administrando las instalaciones de ChromeDriver

Si te gustaría instalar una versión diferente de ChromeDriver a la incluida con Laravel Dusk, puedes usar el comando `dusk:chrome-driver` :

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 74

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all
```

php

Nota

Dusk requiere que los binarios de `chromedriver` sean ejecutables. Si tienes problemas para ejecutar Dusk, asegurate de que los binarios sean ejecutables con el siguiente comando: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Usando otros navegadores

De forma predeterminada, Dusk usa Google Chrome y una instalación de [ChromeDriver](#) independiente para ejecutar tus pruebas de navegador. Sin embargo, puedes iniciar tu propio servidor Selenium y ejecutar tus pruebas en cualquier navegador que deseas.

Para empezar, abre tu archivo `tests/DuskTestCase.php`, el cual es el caso de prueba de Dusk básico para tu aplicación. Dentro de este archivo, puedes remover la ejecución del método `startChromeDriver`. Esto evitará que Dusk inicie automáticamente ChromeDriver:

```
/**
 * Prepare for Dusk test execution.
 *
 * @beforeClass
 * @return void
```

php

```
 */
public static function prepare()
{
    // static::startChromeDriver();
}
```

Luego de esto, puedes modificar el método `driver` para conectar a la URL y puerto de tu preferencia. Además, puedes modificar las "capacidades deseadas" que deberían ser pasadas al WebDriver:

```
/**
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()
    );
}
```

php

Primeros pasos

Generando pruebas

Para generar una prueba de Dusk, usa el comando de Artisan `dusk:make`. La prueba generada será colocada en el directorio `tests/Browser`:

```
php artisan dusk:make LoginTest
```

php

Ejecutando pruebas

Para ejecutar tus pruebas de navegador, usa el comando Artisan `dusk`:

```
php artisan dusk
```

php

Si tuviste fallos en las pruebas la última vez que se ejecutó el comando `dusk`, puedes ahorrar tiempo volviendo a ejecutar las pruebas fallidas usando el comando `dusk: fail`:

```
php artisan dusk:fails
```

php

El comando `dusk` acepta cualquier argumento que sea aceptado normalmente por el administrador de pruebas de PHPUnit, permitiendo que ejecutes solamente las pruebas para un [grupo](#) dado, etc:

```
php artisan dusk --group=foo
```

php

Iniciando manualmente ChromeDriver

De forma predeterminada, Dusk intentará automáticamente iniciar ChromeDriver. Si esto no funciona para tu sistema en particular, puedes iniciar manualmente ChromeDriver antes de ejecutar el comando `dusk`. Si eliges iniciar manualmente ChromeDriver, debes comentar la siguiente línea de tu archivo `tests/DuskTestCase.php`:

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 * @return void  
 */  
public static function prepare()  
{  
    // static::startChromeDriver();  
}
```

php

Además, si inicias ChromeDriver en un puerto diferente a 9515, deberías modificar el método `driver` de la misma clase:

```
/**  
 * Create the RemoteWebDriver instance.  
 *  
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver  
 */  
protected function driver()
```

php

```
{  
    return RemoteWebDriver::create(  
        'http://localhost:9515', DesiredCapabilities::chrome()  
    );  
}
```

Manejo de entorno

Para forzar que Dusk use su propio archivo de entorno al momento de ejecutar las pruebas, crea un archivo `.env.dusk.{environment}` en el directorio raíz de tu proyecto. Por ejemplo, si estás iniciando el comando `dusk` desde tu entorno `local`, deberías crear un archivo `.env.dusk.local`.

Al momento de ejecutar pruebas, Dusk respaldará tu archivo `.env` y renombrará tu entorno Dusk a `.env`. Una vez que las pruebas han sido completadas, tu archivo `.env` será restaurado.

Creando navegadores

Para empezar, vamos a escribir una prueba que verifica que podemos entrar a nuestra aplicación. Después de generar una prueba, podemos modificarla para visitar la página de login, introducir algunas credenciales y presionar el botón "Login". Para crear una instancia del navegador, ejecuta el método `browse`:

```
<?php  
  
namespace Tests\Browser;  
  
use App\User;  
use Illuminate\Foundation\Testing\DatabaseMigrations;  
use Laravel\Dusk\Chrome;  
use Tests\DuskTestCase;  
  
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', 'password')
            ->press('Login')
            ->assertPathIs('/home');
    });
}
```

Como puedes ver en el ejemplo anterior, el método `browse` acepta una función callback. Una instancia de navegador será pasada automáticamente a esta función de retorno por Dusk y es el objeto principal utilizado para interactuar y hacer aserciones en la aplicación.

Creando múltiples navegadores

Algunas veces puedes necesitar múltiples navegadores con el propósito de ejecutar apropiadamente una prueba. Por ejemplo, múltiples navegadores pueden ser necesitados para probar una pantalla de conversaciones que interactúa con websockets. Para crear múltiples navegadores, "solicita" más de un navegador en la firma del callback dado al método `browse` :

```
$this->browse(function ($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');
});
```

Redimensionando las ventanas del navegador

Puedes usar el método `resize` para ajustar el tamaño de la ventana del navegador:

```
$browser->resize(1920, 1080);
```

php

El método `maximize` puede ser usado para maximizar la ventana del navegador:

```
$browser->maximize();
```

php

Macros de navegador

Si desea definir un método de navegador personalizado que puedas reutilizar en una variedad de tus pruebas, puedes usar el método `macro` en la clase `Browser`. Normalmente, deberías llamar a este método desde el método `boot` del [proveedor de servicios](#):

```
<?php  
  
namespace App\Providers;  
  
use Illuminate\Support\ServiceProvider;  
use Laravel\Dusk\Browser;  
  
class DuskServiceProvider extends ServiceProvider  
{  
    /**  
     * Register the Dusk's browser macros.  
     *  
     * @return void  
     */  
    public function boot()  
    {  
        Browser::macro('scrollToElement', function ($element = null) {  
            $this->script("$( 'html, body' ).animate({ scrollTop: $($element).of  
  
                return $this;  
            }));  
        }  
    }  
}
```

php

La función `macro` acepta un nombre como primer argumento y un Closure como segundo. El Closure del macro se ejecutará cuando se llame al macro como un método en una implementación de `Browser`:

```
$this->browse(function ($browser) use ($user) {  
    $browser->visit('/pay')  
        ->scrollToElement('#credit-card-details')  
        ->assertSee('Enter Credit Card Details');  
});
```

php

Autenticación

Frecuentemente, estarás probando páginas que requieren autenticación. Puedes usar el método `loginAs` de Dusk con el propósito de evitar interactuar con la pantalla de login durante cada prueba. El método `loginAs` acepta un ID de usuario o una instancia de modelo de usuario:

```
$this->browse(function ($first, $second) {  
    $first->loginAs(User::find(1))  
        ->visit('/home');  
});
```

php

Nota

Después de usar el método `loginAs`, la sesión de usuario será mantenida para todas las pruebas dentro del archivo.

Migraciones de bases de datos

Cuando tu prueba requiere migraciones, como el ejemplo de autenticación visto antes, nunca deberías usar el trait `RefreshDatabase`. El trait `RefreshDatabase` se apoya en transacciones de base de datos, las cuales no serán aplicables a través de las solicitudes HTTP. En su lugar, usa el trait `DatabaseMigrations`:

```
<?php  
  
namespace Tests\Browser;
```

php

```
use App\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

class ExampleTest extends DuskTestCase
{
    use DatabaseMigrations;
}
```

Interactuando con elementos

Selectores de Dusk

Elegir buenos selectores CSS para interactuar con elementos es una de las partes más difíciles de escribir las pruebas de Dusk. Con el tiempo, los cambios del diseño frontend pueden causar que los selectores CSS como los siguientes dañen tus pruebas:

```
// HTML...
<button>Login</button>

// Test...

$browser->click('.login-page .container div > button');
```

php

Los selectores de Dusk permiten que te enfoques en la escritura de pruebas efectivas en vez de recordar selectores CSS. Para definir un selector, agrega un atributo `dusk` a tu elemento HTML. Después, agrega un prefijo al selector con `@` para manipular el elemento conectado dentro de una prueba de Dusk:

```
// HTML...
<button dusk="login-button">Login</button>

// Test...

$browser->click('@login-button');
```

php

Haciendo clic en enlaces

Para hacer clic sobre un enlace, puedes usar el método `clickLink` en la instancia del navegador. El método `clickLink` hará clic en el enlace que tiene el texto dado en la pantalla:

```
$browser->clickLink($linkText);
```

php

Nota

Este método interactúa con jQuery. Si jQuery no está disponible en la página, Dusk lo inyectará automáticamente de modo que esté disponible por la duración de la prueba.

Texto, Valores y Atributos

Obteniendo y estableciendo valores

Dusk proporciona varios métodos para interactuar con el texto de pantalla, valor y atributos de elementos en la página actual. Por ejemplo, para obtener el "valor" de un elemento que coincide con un selector dado, usa el método `value` :

```
// Retrieve the value...
$value = $browser->value('selector');

// Set the value...
$browser->value('selector', 'value');
```

php

Obteniendo texto

El método `text` puede ser usado para obtener el texto de pantalla de un elemento que coincide con el selector dado:

```
$text = $browser->text('selector');
```

php

Obteniendo atributos

Finalmente, el método `attribute` puede ser usado para obtener un atributo de un elemento que coincide con el selector dado:

```
$attribute = $browser->attribute('selector', 'value');
```

php

Usando Formularios

Escribiendo valores

Dusk proporciona una variedad de métodos para interactuar con formularios y elementos de entrada. Primero, vamos a echar un vistazo a un ejemplo de escribir texto dentro de un campo de entrada:

```
$browser->type('email', 'taylor@laravel.com');
```

php

Nota que, aunque el método acepta uno si es necesario, no estamos obligados a pasar un selector CSS dentro del método `type`. Si un selector CSS no es proporcionado, Dusk buscará un campo de entrada con el atributo `name` dado. Finalmente, Dusk intentará encontrar un `textarea` con el atributo `name` dado.

Para agregar texto a un campo sin limpiar su contenido, puedes usar el método `append`:

```
$browser->type('tags', 'foo')
->append('tags', 'bar, baz');
```

php

Puedes limpiar el valor de un campo usando el método `clear`:

```
$browser->clear('email');
```

php

Listas desplegables

Para seleccionar un valor en un cuadro de selección de lista desplegable, puedes usar el método `select`. Al momento de pasar un valor al método `select`, deberías pasar el valor de opción a resaltar en lugar del texto mostrado en pantalla:

```
$browser->select('size', 'Large');
```

php

Puedes seleccionar una opción aleatoria al omitir el segundo parámetro:

```
$browser->select('size');
```

php

Casillas de verificación

Para "marcar" un campo de casilla de verificación, puedes usar el método `check`. Al igual que muchos otros métodos relacionados con entradas, un selector CSS completo no es obligatorio. Si un selector que coincide exactamente no puede ser encontrado, Dusk buscará una casilla de verificación con un atributo `name` coincidente.

```
$browser->check('terms');
```

php

```
$browser->uncheck('terms');
```

Botones de radio

Para "seleccionar" una opción de botón de radio, puedes usar el método `radio`. Al igual que muchos otros métodos relacionados con campos, un selector CSS completo no es obligatorio. Si un selector que coincide exactamente no puede ser encontrado, Dusk buscará un radio con atributos `name` y `value` coincidentes:

```
$browser->radio('version', 'php7');
```

php

Adjuntando archivos

El método `attach` puede ser usado para adjuntar un archivo a un elemento `file`. Al igual que muchos otros métodos relacionados con campos, un selector CSS completo no es obligatorio. Si un selector que coincide exactamente no puede ser encontrado, Dusk buscará un campo de archivo con atributo `name` coincidente:

```
$browser->attach('photo', __DIR__. '/photos/me.png');
```

php

Nota

La función `attach` requiere que la extensión de PHP `Zip` esté instalada y habilitada en tu servidor.

Usando el teclado

El método `keys` permite que proporciones secuencias de entrada más complejas para un elemento dado que lo permitido normalmente por el método `type`. Por ejemplo, puedes mantener presionada las teclas modificadoras al introducir valores. En este ejemplo, la tecla `shift` será mantenida presionada mientras la palabra `taylor` es introducida dentro del elemento que coincide con el selector dado. Después de que la palabra `taylor` sea tipeada, la palabra `otwell` será tipeada sin alguna tecla modificadora:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

php

Incluso puedes enviar una "tecla de función" al selector CSS principal que contiene tu aplicación:

```
$browser->keys('.app', ['{command}', 'j']);
```

php

TIP TIP

Todas las teclas modificadoras se envuelven entre corchetes `{}` y coinciden con las constantes definidas en la clase `Facebook\WebDriver\WebDriverKeys`, la cual puede ser [encontrada en GitHub](#).

Usando el Ratón

Haciendo clic sobre elementos

El método `click` puede ser usado para "clickear" sobre un elemento que coincide con el selector dado:

```
$browser->click('.selector');
```

php

Mouseover

El método `mouseover` puede ser usado cuando necesitas mover el ratón sobre un elemento que coincide con el selector dado:

```
$browser->mouseover('.selector');
```

php

Arrastrar y soltar

El método `drag` puede ser usado para arrastrar un elemento que coincide con el selector dado hasta otro elemento:

```
$browser->drag('.from-selector', '.to-selector');
```

php

O, puedes arrastrar un elemento en una única dirección:

```
$browser->dragLeft('.selector', 10);
$browser->dragRight('.selector', 10);
$browser->dragUp('.selector', 10);
$browser->dragDown('.selector', 10);
```

php

Diálogos de JavaScript

Dusk provee de varios métodos para interactuar con Diálogos de JavaScript:

```
// Wait for a dialog to appear:
$browser->waitForDialog($seconds = null);

// Assert that a dialog has been displayed and that its message matches the given value:
$browser->assertDialogOpened('value');

// Type the given value in an open JavaScript prompt dialog:
$browser->typeInDialog('Hello World');
```

php

Para cerrar un Diálogo de JavaScript abierto, haga clic en el botón Aceptar o OK:

```
$browser->acceptDialog();
```

php

Para cerrar un Diálogo de JavaScript abierto, haga clic en el botón Cancelar (solo para un diálogo de confirmación):

```
$browser->dismissDialog();
```

php

Alcance de selectores

Algunas veces puedes querer ejecutar varias operaciones dentro del alcance de un selector dado. Por ejemplo, puedes querer comprobar que algunos textos existen únicamente dentro de una tabla y después presionar un botón dentro de la tabla. Puedes usar el método `with` para completar esta tarea. Todas las operaciones ejecutadas dentro de la función de retorno dada al método `with` serán exploradas en el selector original:

```
$browser->with('.table', function ($table) {
    $table->assertSee('Hello World')
        ->clickLink('Delete');
});
```

php

Esperando por elementos

Al momento de probar aplicaciones que usan JavaScript de forma extensiva, frecuentemente se vuelve necesario "esperar" por ciertos elementos o datos estén disponibles antes de proceder con una prueba. Dusk hace esto fácilmente. Usando una variedad de métodos, puedes esperar que los elementos estén visibles en la página e incluso esperar hasta que una expresión de JavaScript dada se evalúe como `true`.

Esperando

Si necesitas pausar la prueba por un número de milisegundos dado, usa el método `pause`:

```
$browser->pause(1000);
```

php

Esperando por selectores

El método `waitFor` puede ser usado para pausar la ejecución de la prueba hasta que el elemento que coincide con el selector CSS dado sea mostrado en la página. De forma predeterminada, esto pausará la

prueba por un máximo de cinco segundos antes de arrojar una excepción. Si es necesario, puedes pasar un umbral de tiempo de expiración personalizado como segundo argumento del 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);
```

php

También puede esperar hasta que el selector dado no se encuentre en la página:

```
$browser->waitUntilMissing('.selector');

$browser->waitUntilMissing('.selector', 1);
```

php

Estableciendo el alcance de selectores cuando estén disponibles

Ocasionalmente, puedes querer esperar por un selector dado y después interactuar con el elemento que coincida con el selector. Por ejemplo, puedes querer esperar hasta que una ventana modal esté disponible y después presionar el botón "OK" dentro de esa ventana. El método `whenAvailable` puede ser usado en este caso. Todas las operaciones de elementos ejecutadas dentro de la función de retorno dada serán ejecutadas dentro del selector original:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

php

Esperando por texto

El método `waitForText` puede ser usado para esperar hasta que el texto dado sea mostrado en la página:

```
// Wait a maximum of five seconds for the text...
$browser->waitForText('Hello World');
```

php

```
// Wait a maximum of one second for the text...
$browser->waitForText('Hello World', 1);
```

Esperando por enlaces

El método `waitForLink` puede ser usado para esperar hasta que un enlace dado sea mostrada 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);
```

php

Esperando por la localización de la página

Al momento de hacer una comprobación de ruta tal como `$browser->assertPathIs('/home')`, la comprobación puede fallar si `window.location.pathname` está siendo actualizada asincrónicamente. Puedes usar el método `waitForLocation` para esperar por la localización que tenga un valor dado:

```
$browser->waitForLocation('/secret');
```

php

También puede esperar la localización de una ruta con nombre:

```
$browser->waitForRoute($routeName, $parameters);
```

php

Esperando por recargas de página

Si necesita hacer aserciones después de que se ha recargado una página, usa el método

`waitForReload` :

```
$browser->click('.some-action')
    ->waitForReload()
    ->assertSee('something');
```

php

Esperando por expresiones de JavaScript

Algunas veces puedes querer pausar la ejecución de una prueba hasta que una expresión de JavaScript dada se evalúe a `true`. Puedes completar fácilmente esto usando el método `waitFor`. Al momento de pasar una expresión a este método, no necesitas incluir al final la palabra clave `return` o un punto y coma `;`:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitFor('App.dataLoaded');

$browser->waitFor('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitFor('App.data.servers.length > 0', 1);
```

php

Esperando por expresiones de Vue

Los siguientes métodos puedes ser usados para esperar hasta que un atributo de componente de Vue dado tenga un determinado valor:

```
// Wait until the component attribute contains the given value...
$browser->waitForVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitForVueIsNot('user.name', null, '@user');
```

php

Esperando por una función de retorno

Muchos de los métodos de "espera" en Dusk confían en el método `waitForUsing` subyacente. Puedes usar este método directamente para esperar por una función de retorno dada que devuelva `true`. El método `waitForUsing` acepta el máximo número de segundos para esperar la Closure, el intervalo en el cual la Closure debería ser evaluada y un mensaje opcional de falla:

```
$browser->waitForUsing(10, 1, function () use ($something) {
    return $something->isReady();
}, "Something wasn't ready in time.");
```

php

Haciendo aserciones de Vue

Inclusive Dusk permite que hagas comprobaciones en el estado de componente de datos de Vue. Por ejemplo, imagina que tu 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'  
            }  
        };  
    }  
});
```

php

Puedes comprobar el estado del componente de Vue de esta manera:

```
/**  
 * A basic Vue test example.  
 *  
 * @return void  
 */  
public function testVue()  
{  
    $this->browse(function (Browser $browser) {  
        $browser->visit('/')  
            ->assertVue('user.name', 'Taylor', '@profile-component');  
    });  
}
```

php

Aserciones disponibles

Dusk proporciona una variedad de aserciones que puedes hacer en tu aplicación. Todas las aserciones disponibles están documentadas en la tabla de abajo:

assertTitle	assertHasCookie	assertSelected
assertTitleContains	assertCookieMissing	assertNotSelected
assertUrlIs	assertCookieValue	assertSelectHasOptions
assertSchemels	assertPlainCookieValue	assertSelectMissingOptions
assertSchemelsNot	assertSee	assertSelectHasOption
assertHostIs	assertDontSee	assertValue
assertHostIsNot	assertSeeIn	assertVisible
assertPortIs	assertDontSeeIn	assertPresent
assertPortIsNot	assertSourceHas	assertMissing
assertPathBeginsWith	assertSourceMissing	assertDialogOpened
assertPathIs	assertSeeLink	assertEnabled
assertPathIsNot	assertDontSeeLink	assertDisabled
assertRouteIs	assertInputValue	assertFocused
assertQueryStringHas	assertInputValuesNot	assertNotFocused
assertQueryStringMissing	assertChecked	assertVue
assertFragments	assertNotChecked	assertVuelsNot
assertFragmentBeginsWith	assertRadioSelected	assertVueContains
assertFragmentsIsNot	assertRadioNotSelected	assertVueDoesNotContain

assertTitle

Comprueba que el título de la página coincida con el texto dado:

```
$browser->assertTitle($title);
```

php

assertTitleContains

Comprueba que el título de página contenga el texto dado:

```
$browser->assertTitleContains($title);
```

php

assertUrlIs

Comprueba que la URL actual (sin la cadena de consulta) coincida con la cadena dada:

```
$browser->assertUrlIs($url);
```

php

assertSchemeIs

Comprueba que el esquema de la URL actual coincide con el esquema dado:

```
$browser->assertSchemeIs($scheme);
```

php

assertSchemeIsNot

Comprueba que el esquema de la URL actual no coincide con el esquema dado:

```
$browser->assertSchemeIsNot($scheme);
```

php

assertHostIs

Comprueba que el Host de la URL actual coincide con el Host dado:

```
$browser->assertHostIs($host);
```

php

assertHostIsNot

Comprueba que el Host de la URL actual no coincide con el Host dado:

```
$browser->assertHostIsNot($host);
```

php

assertPortIs

Comprueba que el puerto de la URL actual coincide con el puerto dado:

```
$browser->assertPortIs($port);
```

php

assertPortIsNot

Comprueba que el puerto de la URL actual no coincide con el puerto dado:

```
$browser->assertPortIsNot($port);
```

php

assertPathBeginsWith

Comprueba que la ruta de la URL actual comience con la ruta dada:

```
$browser->assertPathBeginsWith($path);
```

php

assertPathIs

Comprueba que la ruta actual coincida con la ruta dada:

```
$browser->assertPathIs('/home');
```

php

assertPathIsNot

Comprueba que la ruta actual no coincide con la ruta dada:

```
$browser->assertPathIsNot('/home');
```

php

assertRouteIs

Comprueba que la URL actual coincide con la URL de ruta nombrada dada:

```
$browser->assertRouteIs($name, $parameters);
```

php

assertQueryStringHas

Comprueba que el parámetro de cadena para una consulta dada está presente:

```
$browser->assertQueryStringHas($name);
```

php

Comprueba que el parámetro de cadena para una consulta dada está presente y tiene un valor dado:

```
$browser->assertQueryStringHas($name, $value);
```

php

assertQueryStringMissing

Comprueba que el parámetro de cadena para una consulta dada está ausente:

```
$browser->assertQueryStringMissing($name);
```

php

assertFragmentsIs

Comprueba que el fragmento actual coincide con el fragmento dado:

```
$browser->assertFragmentIs('anchor');
```

php

assertFragmentBeginsWith

Comprueba que el fragmento actual comienza con el fragmento dado:

```
$browser->assertFragmentBeginsWith('anchor');
```

php

assertFragmentIsNot

AComprueba que el fragmento actual no coincide con el fragmento dado:

```
$browser->assertFragmentIsNot('anchor');
```

php

assertHasCookie

Comprueba que el cookie dado está presente:

```
$browser->assertHasCookie($name);
```

php

assertCookieMissing

Comprueba que el cookie dado no está presente:

```
$browser->assertCookieMissing($name);
```

php

assertCookieValue

Comprueba que un cookie tenga un valor dado:

```
$browser->assertCookieValue($name, $value);
```

php

assertPlainCookieValue

Comprueba que un cookie desencriptado tenga un valor dado:

```
$browser->assertPlainCookieValue($name, $value);
```

php

assertSee

Comprueba que el texto dado está presente en la página:

```
$browser->assertSee($text);
```

php

assertDontSee

Comprueba que el texto dado no está presente en la página:

```
$browser->assertDontSee($text);
```

php

assertSeeIn

Comprueba que el texto dado está presente dentro del selector:

```
$browser->assertSeeIn($selector, $text);
```

php

assertDontSeeIn

Comprueba que el texto dado no está presente dentro del selector:

```
$browser->assertDontSeeIn($selector, $text);
```

php

assertSourceHas

Comprueba que el código fuente dado está presente en la página:

```
$browser->assertSourceHas($code);
```

php

assertSourceMissing

Comprueba que el código fuente dado no está presente en la página:

```
$browser->assertSourceMissing($code);
```

php

assertSeeLink

Comprueba que el enlace dado está presente en la página:

```
$browser->assertSeeLink($linkText);
```

php

assertDontSeeLink

Comprueba que el enlace dado está no presente en la página:

```
$browser->assertDontSeeLink($linkText);
```

php

assertInputValue

Comprueba que el campo de entrada dado tiene el valor dado:

```
$browser->assertInputValue($field, $value);
```

php

assertInputValueIsNot

Comprueba que el campo de entrada dado no tiene el valor dado:

```
$browser->assertInputValueIsNot($field, $value);
```

php

assertChecked

Comprueba que la casilla de verificación está marcada:

```
$browser->assertChecked($field);
```

php

assertNotChecked

Comprueba que la casilla de verificación no está marcada:

```
$browser->assertNotChecked($field);
```

php

assertRadioSelected

Comprueba que el campo de radio está seleccionado:

```
$browser->assertRadioSelected($field, $value);
```

php

assertRadioNotSelected

Comprueba que el campo de radio no está seleccionado:

```
$browser->assertRadioNotSelected($field, $value);
```

php

assertSelected

Comprueba que la lista desplegable tiene seleccionado el valor dado:

```
$browser->assertSelected($field, $value);
```

php

assertNotSelected

Comprueba que la lista desplegable no tiene seleccionado el valor dado:

```
$browser->assertNotSelected($field, $value);
```

php

assertSelectHasOptions

Comprueba que el arreglo dado de valores están disponibles para ser seleccionados:

```
$browser->assertSelectHasOptions($field, $values);
```

php

assertSelectMissingOptions

Comprueba que el arreglo dado de valores no están disponibles para ser seleccionados:

```
$browser->assertSelectMissingOptions($field, $values);
```

php

assertSelectHasOption

Comprueba que el valor dado está disponible para ser seleccionado en el campo dado:

```
$browser->assertSelectHasOption($field, $value);
```

php

assertValue

Comprueba que el elemento que coincide con el selector dado tenga el valor dado:

```
$browser->assertValue($selector, $value);
```

php

assertVisible

Comprueba que el elemento que coincide con el selector dado sea visible:

```
$browser->assertVisible($selector);
```

php

assertPresent

Comprueba que el elemento que coincide con el selector dado está presente:

```
$browser->assertPresent($selector);
```

php

assertMissing

Comprueba que el elemento que coincide con el selector dado no sea visible:

```
$browser->assertMissing($selector);
```

php

assertDialogOpened

Comprueba que un diálogo JavaScript con un mensaje dado ha sido abierto:

```
$browser->assertDialogOpened($message);
```

php

assertEnabled

Comprueba que el campo dado está activado:

```
$browser->assertEnabled($field);
```

php

assertDisabled

Comprueba que el campo dado está desactivado:

```
$browser->assertDisabled($field);
```

php

assertFocused

Comprueba que el campo dado está enfocado:

```
$browser->assertFocused($field);
```

php

assertNotFocused

Comprueba que el campo dado no está enfocado:

```
$browser->assertNotFocused($field);
```

php

assertVue

Comprueba que una propiedad de datos de un componente de Vue dado coincide con el valor dado:

```
$browser->assertVue($property, $value, $componentSelector = null);
```

php

assertVueIsNot

Comprueba que una propiedad de datos de un componente de Vue dado no coincide con el valor dado:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

php

assertVueContains

Comprueba que una propiedad de datos de un componente de Vue dado es un arreglo y contiene el valor dado:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

php

assertVueDoesNotContain

Comprueba que una propiedad de datos de un componente de Vue dado es un arreglo y no contiene el valor dado:

```
$browser->assertVueDoesNotContain($property, $value, $componentSelector = null);
```

php

Páginas

Alguna veces, las pruebas requieren que varias acciones complicadas sean ejecutadas en secuencia. Esto puede hacer tus pruebas más difíciles de leer y entender. Las páginas permiten que definas acciones expresivas que entonces pueden ser ejecutadas en una página dada usando un solo método. Las páginas también permiten que definas abreviaturas para selectores comunes para tu aplicación o una página única.

Generando páginas

Para generar un objeto de página, usa el comando Artisan `dusk:page`. Todos los objetos de página serán colocados en el directorio `tests/Browser/Pages`:

```
php artisan dusk:page Login
```

php

Configurando páginas

De forma predeterminada, las páginas tienen tres métodos: `url`, `assert`, y `elements`.

Discutiremos los métodos `url` y `assert` ahora. El método `elements` será [discutido con más detalle debajo](#).

El método `url`

El método `url` debería devolver la ruta de la URL que representa la página. Dusk usará esta URL al momento de navegar a la página en el navegador:

```
/**  
 * Get the URL for the page.  
 *  
 * @return string  
 */  
public function url()  
{  
    return '/login';  
}
```

php

El método `assert`

El método `assert` puede hacer algunas aserciones necesarias para verificar que el navegador en realidad está en la página dada. Completar este método no es necesario; sin embargo, eres libre de hacer estas aserciones si lo deseas. Estas aserciones serán ejecutadas automáticamente al momento de navegar hacia la página:

```
/**  
 * Assert that the browser is on the page.  
 *  
 * @return void  
 */
```

php

```
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

Navegando hacia las páginas

Una vez que se ha configurado una página, puedes navegar a ella utilizando el método `visit` :

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

php

A veces es posible que ya estés en una página determinada y necesitas "cargar" los selectores y métodos dentro del contexto de prueba actual. Esto es común al momento de presionar un botón y ser redireccionado a una página dada sin navegar explícitamente a ésta. En esta situación, puedes usar el método `on` para cargar la página.

```
use Tests\Browser\Pages\CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

php

Selectores abreviados

El método `elements` de páginas permite que definas abreviaturas rápidas, fáciles de recordar para cualquier selector CSS en tu página. Por ejemplo, vamos a definir una abreviación para el campo "email" de la página login de la aplicación:

```
/**
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
```

php

```
return [
    '@email' => 'input[name=email]',
];
}
```

Ahora, puedes usar este selector de abreviación en cualquier lugar que usarías un selector de CSS completo:

```
$browser->type('@email', 'taylor@laravel.com');
```

php

Selectores de abreviaturas globales

Después de instalar Dusk, una clase `Page` básica será colocada en tu directorio

`tests/Browser/Pages`. Esta clase contiene un método `siteElements` el cual puede ser usado para definir selectores de abreviaturas globales que deberían estar disponibles en cada página en cada parte de tu aplicación:

```
/*
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

php

Métodos de página

Además de los métodos predeterminados definidos en páginas, puedes definir métodos adicionales, los cuales pueden ser usados en cualquier parte de tus pruebas. Por ejemplo, vamos a imaginar que estamos construyendo una aplicación para administración de música. Una acción común para una página de la aplicación podría ser crear una lista de reproducción. En lugar de volver a escribir la lógica para crear una lista de reproducción en cada prueba, puedes definir un método `createPlaylist` en una clase de página:

php

```
<?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');
    }
}
```

php

```
use Tests\Browser\Pages\Dashboard;

$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```

Componentes

Los componentes son similares a “los objetos de página” de Dusk, pero son planeados para partes de UI y funcionalidades que sean reusadas en otras partes de tu aplicación, tal como una barra de navegación o ventana de notificación. Como tal, los componentes no son enlazados a URLs específicas.

Generando componentes

Para generar un componente, usa el comando Artisan `dusk:component`. Los nuevos componentes son colocados en el directorio `test/Browser/Components`:

```
php artisan dusk:component DatePicker
```

php

Como se muestra antes, un "calendario" es un ejemplo de un componente que puede existir en cualquier parte de tu aplicación en una variedad de páginas. Puede volverse complejo escribir manualmente lógica de automatización de navegador para seleccionar una fecha entre docenas de pruebas en cualquier parte de tu software de prueba. En lugar de esto, podemos definir un componente de Dusk para representar el calendario, permitiendo encapsular esa lógica dentro del componente:

```
<?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());  
    }  
}
```

php

```

}

/**
 * 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 $day
 * @return void
 */
public function selectDate($browser, $month, $day)
{
    $browser->click('@date-field')
        ->within('@month-list', function ($browser) use ($month) {
            $browser->click($month);
        })
        ->within('@day-list', function ($browser) use ($day) {
            $browser->click($day);
        });
}
}

```

Usando componentes

Una vez que el componente ha sido definido, fácilmente podemos seleccionar una fecha dentro del calendario desde cualquier prueba. Y, si la lógica necesaria para seleccionar una fecha cambia, solamente necesitaremos actualizar el componente:

```
<?php  
  
namespace Tests\Browser;  
  
use Illuminate\Foundation\Testing\DatabaseMigrations;  
use Laravel\Dusk\Browser;  
use Tests\Browser\Components\DatePicker;  
use Tests\DuskTestCase;  
  
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');  
        });  
    }  
}
```

php

Integración continua

CircleCI

Si estás usando CircleCI para ejecutar tus pruebas de Dusk, puedes usar este archivo de configuración como punto de partida. Al igual que con TravisCI, usaremos el comando `php artisan serve` para ejecutar el servidor web integrado de PHP:

```
version: 2  
jobs:  
  build:  
    steps:
```

php

```
- run: sudo apt-get install -y libsqlite3-dev
- run: cp .env.testing .env
- run: composer install --no-interaction --prefer-dist
- 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

- store_artifacts:
  path: tests/Browser/screenshots
```

Codeship

Para ejecutar pruebas de Dusk en [Codeship](#), agrega los siguientes comandos a tu proyecto Codeship. Estos comandos son sólo un punto de partida y eres libre de agregar los comandos adicionales que necesites:

```
phpenv local 7.2
cp .env.testing .env
mkdir -p ./bootstrap/cache
composer install --no-interaction --prefer-dist
php artisan key:generate
nohup bash -c "php artisan serve 2>&1 && sleep 5
php artisan dusk
```

Heroku CI

Para ejecutar tus pruebas de Dusk en [Heroku CI](#), agrega el siguiente buildpack de Google Chrome y scripts a tu archivo `app.json` de Heroku:

```
{  
  "environments": {  
    "test": {  
      "buildpacks": [  
        { "url": "heroku/php" },  
        { "url": "https://github.com/heroku/heroku-buildpack-google-chro  
      ],  
      "scripts": {  
        "test-setup": "cp .env.testing .env",  
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-1  
      }  
    }  
  }  
}
```

Travis CI

Para ejecutar tus pruebas de Dusk en [Travis CI](#), usa la siguiente configuración en el archivo

`.travis.yml`. Ya que Travis CI no es un entorno gráfico, necesitaremos tomar algunos pasos extras con el propósito de ejecutar un navegador Chrome. En adición a esto, usaremos `php artisan serve` para ejecutar el servidor web integrado de PHP:

```
language: php  
  
php:  
  - 7.3  
  
addons:  
  chrome: stable  
  
install:  
  - cp .env.testing .env  
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest  
  - php artisan key:generate  
  
before_script:  
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222
```

```
- php artisan serve &
```

```
script:
```

```
- php artisan dusk
```

En tu archivo `.env.testing`, ajusta el valor de `APP_URL`:

```
APP_URL=http://127.0.0.1:8000
```

php

GitHub Actions

Si estás usando [acciones de GitHub](#) para ejecutar tus pruebas de Dusk, puedes usar este archivo de configuración como punto de partida. Igual que TravisCI, usaremos el comando `php artisan serve` para ejecutar el servidor integrado de PHP:

```
name: CI
on: [push]
jobs:
  dusk-php:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v1
      - name: Prepare The Environment
        run: cp .env.example .env
      - name: Create Database
        run: mysql --user="root" --password="root" -e "CREATE DATABASE my-database"
      - name: Install Composer Dependencies
        run: composer install --no-progress --no-suggest --prefer-dist --optimize-autoloader
      - name: Generate Application Key
        run: php artisan key:generate
      - name: Upgrade Chrome Driver
        run: php artisan dusk:chrome-driver
      - name: Start Chrome Driver
        run: ./vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &
      - name: Run Laravel Server
        run: php artisan serve > /dev/null 2>&1 &
      - name: Run Dusk Tests
        run: php artisan dusk
```

En tu archivo `.env.testing`, ajusta el valor de `APP_URL`:

```
APP_URL=http://127.0.0.1:8000
```

php

Pruebas de Base de Datos

- Introducción
- Generando factories
- Reiniciando la base de datos después de cada prueba
- Escribiendo factories
 - Estados de un factory
 - Llamadas de retorno de un factory
- Usando factories
 - Creando modelos
 - Persistiendo modelos
 - Relaciones
- Aserciones disponibles

Introducción

Laravel proporciona una variedad de herramientas útiles para hacer que sea más fácil probar tus aplicaciones que manejan base de datos. Primero, puedes usar el método (helper)

`assertDatabaseHas` para comprobar que los datos existentes en la base de datos coinciden con un conjunto dado de criterios. Por ejemplo, si quisieras verificar que hay un registro en la tabla `users` con el valor `email` de `sally@example.com`, puedes hacer lo siguiente:

```
public function testDatabase()
{
    // Make call to application...

    $this->assertDatabaseHas('users', [
        'email' => 'sally@example.com',
    ]);
}
```

php

También podrías usar el método `assertDatabaseMissing` para comprobar que esos datos no existen en la base de datos.

El método `assertDatabaseHas` y otros métodos como éste son por conveniencia. Eres libre de usar cualquiera de los métodos de aserción de PHPUnit integrados para complementar tus pruebas.

Generando factories

Para crear un factory, usa el comando Artisan `make:factory` :

```
php artisan make:factory PostFactory
```

php

El nuevo factory será colocado en tu directorio `database/factories`.

La opción `--model` puede ser usada para indicar el nombre del modelo creado por el factory. Esta opción pre-llenará el archivo de factory generado con el modelo dado:

```
php artisan make:factory PostFactory --model=Post
```

php

Reiniciando la base de datos después de cada prueba

Con frecuencia es útil reinicializar tu base de datos después de cada prueba de modo que los datos de una prueba previa no interfieran con las pruebas subsecuentes. El trait `RefreshDatabase` toma el enfoque más óptimo para migrar tu base de datos de pruebas, dependiendo de si estás usando una base de datos en memoria o una base de datos tradicional. Usa el trait en tu clase de prueba y todo será manejado por ti:

php

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * A basic functional test example.
     *
     * @return void
     */
    public function testBasicExample()
    {
        $response = $this->get('/');

        // ...
    }
}
```

Escribiendo factories

Al momento de probar, puedes necesitar insertar unos pocos registros dentro de tu base de datos antes de ejecutar tu prueba. En lugar de especificar manualmente el valor de cada columna cuando crees estos datos de prueba, Laravel permite que definas un conjunto de atributos predeterminados para cada uno de tus modelos de Eloquent usando factories de modelos. Para empezar, echemos un vistazo al archivo `database/factories/UserFactory.php` en tu aplicación. De forma predeterminada, este archivo contiene una definición de factory:

php

```
use Faker\Generator as Faker;
use Illuminate\Support\Str;

$factory->define(App\User::class, function (Faker $faker) {
    return [
        'name' => $faker->name(),
        'email' => $faker->safeEmail(),
        'password' => Str::random(10),
        'remember_token' => Str::random(10),
    ];
});
```

```
'name' => $faker->name,  
'email' => $faker->unique()->safeEmail,  
'email_verified_at' => now(),  
'password' => '$2y$10$TKh8H1.PfQx37YgCzwIKb.KjNyWgaHb9cbcQgdIVFlYg7B77U  
'remember_token' => Str::random(10),  
];  
});
```

Dentro del Closure, la cual sirve como la definición del factory, puedes devolver los valores de prueba predeterminados de todos los atributos del modelo. El Closure recibirá una instancia de la biblioteca PHP [Faker](#), la cual permitirá que generes convenientemente varios tipos de datos aleatorios para las pruebas.

También puedes crear archivos de factories adicionales para cada modelo para una mejor organización. Por ejemplo, podrías crear archivos `UserFactory.php` y `CommentFactory.php` dentro de tu directorio `database/factories`. Todos los archivos dentro del directorio `factories` serán cargados automáticamente por Laravel.

TIP TIP

Puedes establecer la configuración regional de Faker agregando una opción `faker_locale` a tu archivo de configuración `config/app.php`.

Estados de un factory

Los estados te permiten definir modificaciones discretas que pueden ser aplicadas a tus factories de modelos en cualquier combinación. Por ejemplo, tu modelo `User` podría tener un estado

`delinquent` que modifique uno de sus valores de atributo predeterminados. Puedes definir tus transformaciones de estado usando el método `state`. Para estados simples, puedes pasar un arreglo de modificaciones de atributos:

```
$factory->state(App\User::class, 'delinquent', [  
    'account_status' => 'delinquent',  
]);
```

php

Si tu estado requiere cálculo o una instancia `$faker`, puedes usar un Closure para calcular las modificaciones de los atributos del estado:

```
$factory->state(App\User::class, 'address', function ($faker) {  
    return [  
        'address' => $faker->address,  
    ];  
});
```

php

LLamadas de retorno de un factory

Las llamadas de retorno de un Factory son registradas usando los métodos `afterMaking` y `afterCreating` y te permiten realizar tareas adicionales de hacer o crear un modelo. Por ejemplo, puedes usar llamadas de retorno para relacionar modelos adicionales con el modelo creado:

```
$factory->afterMaking(App\User::class, function ($user, $faker) {  
    // ...  
});  
  
$factory->afterCreating(App\User::class, function ($user, $faker) {  
    $user->accounts()->save(factory(App\Account::class)->make());  
});
```

php

También puedes definir llamadas de retorno para [estados de un factory](#):

```
$factory->afterMakingState(App\User::class, 'delinquent', function ($user, $faker) {  
    // ...  
});  
  
$factory->afterCreatingState(App\User::class, 'delinquent', function ($user, $faker) {  
    // ...  
});
```

php

Usando factories

Creando modelos

Una vez que has definido tus factories, puedes usar la función global `factory` en tus pruebas o en archivos seeder para generar instancias de un modelo. Así, vamos a echar un vistazo en unos pocos

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...
}
```

php

También puedes crear una colección de muchos modelos o crear modelos de un tipo dado:

```
// Create three App\User instances...
$users = factory(App\User::class, 3)->make();
```

php

Aplicando estados

También puedes aplicar cualquiera de tus [estados](#) a los modelos. Si prefieres aplicar múltiples transformaciones de estado a los modelos, deberías especificar el nombre de cada estado que quisieras aplicar:

```
$users = factory(App\User::class, 5)->states('delinquent')->make();

$users = factory(App\User::class, 5)->states('premium', 'delinquent')->make();
```

php

Sobrescribiendo atributos

Si prefieres sobreescribir algunos de los valores predeterminados de tus modelos, puedes pasar un arreglo de valores al método `make`. Solamente, los valores especificados serán reemplazados mientras que el resto de los valores permanecerán con sus valores predeterminados como se especificó en el factory:

```
$user = factory(App\User::class)->make([
    'name' => 'Abigail',
]);
```

php

Persistiendo modelos

El método `create` no solamente crea las instancias de un modelo sino que también los almacena en la base de datos usando el método `save` de Eloquent:

```
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...
}
```

php

Puedes sobrescribir atributos en el modelo al pasar un arreglo al método `create`:

```
$user = factory(App\User::class)->create([
    'name' => 'Abigail',
]);
```

php

Relaciones

En este ejemplo, adjuntaremos una relación para algunos modelos creados. Al momento de usar el método `create` para crear múltiples modelos, una [instancia de colección](#) de Eloquent es devuelta, permitiendo que uses cualquiera de las funciones convenientes proporcionadas por la colección, tales como `each`:

```
$users = factory(App\User::class, 3)
    ->create()
    ->each(function ($user) {
        $user->posts()->save(factory(App\Post::class)->make());
    });

```

php

Relaciones y closures de atributos

También puedes adjuntar relaciones a los modelos en tus definiciones del factory. Por ejemplo, si prefieres crear una nueva instancia `User` al momento de crear un `Post`, puedes hacer lo siguiente:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => factory(App\User::class),
    ];
});
```

php

Si la relación depende del factory que las define debes proporcionar un callback que acepta el arreglo de atributos evaluado:

```
$factory->define(App\Post::class, function ($faker) {
    return [
        'title' => $faker->title,
        'content' => $faker->paragraph,
        'user_id' => factory(App\User::class),
        'user_type' => function (array $post) {
            return App\User::find($post['user_id'])->type;
        },
    ];
});
```

php

Usando Seeders

Si te gustaría usar [seeders de bases de datos](#) para llenar tu base de datos al momento de realizar una prueba, puedes usar el método `seed`. Por defecto, el método `seed` retornará `DatabaseSeeder`, que debería ejecutar todos tus otros seeders. De forma alternativa, pasas un nombre de clase seeder específico al método `seed`:

```
<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
```

php

```

use OrderStatusesTableSeeder;
use Tests\TestCase;

class ExampleTest extends TestCase
{
    use RefreshDatabase;

    /**
     * Test creating a new order.
     *
     * @return void
     */
    public function testCreatingANewOrder()
    {
        // Run the DatabaseSeeder...
        $this->seed();

        // Run a single seeder...
        $this->seed(OrderStatusesTableSeeder::class);

        // ...
    }
}

```

Aserciones disponibles

Laravel proporciona varias aserciones de base de datos para tus pruebas [PHPUnit](#):

Método	Descripción
<code>\$this->assertDatabaseHas(\$table, array \$data);</code>	Comprueba que una tabla en la base de datos contiene los datos dados.
<code>\$this->assertDatabaseMissing(\$table, array \$data);</code>	Comprueba que una tabla en la base de datos no contiene los datos dados.
<code>\$this->assertDeleted(\$table, array \$data);</code>	Comprueba que el registro dado ha sido eliminado.

Método	Descripción
<code>\$this->assertSoftDeleted(\$table, array \$data);</code>	Comprueba que el registro dado ha sido borrado lógicamente.

Por conveniencia, puedes pasar un modelo a los helpers `assertDeleted` y `assertSoftDeleted` para comprobar que el registro fue eliminado o borrado lógicamente, respectivamente, de la base de datos en base a la clave primaria del modelo.

Por ejemplo, si estás usando un model factory en tu prueba, puedes pasar este modelo a uno de estos helpers para probar si tu aplicación borró de forma apropiada el registro de la base de datos:

```
public function testDatabase()
{
    $user = factory(App\User::class)->create();
    // Make call to application...
    $this->assertDeleted($user);
}
```

php

Mocking

- Introducción
- Mocking de objetos
- Fake de trabajos (Jobs)
- Fake de eventos
 - Fake de eventos con alcance
- Fake de correos electrónicos
- Fake de notificaciones
- Fake de colas

- [Fake de almacenamiento de archivos](#)
- [Clases facade](#)

Introducción

Al momento de probar aplicaciones de Laravel, puedes querer "simular" (mock) ciertos aspectos de tu aplicación de modo que realmente no sean ejecutados durante una prueba dada. Por ejemplo, al momento de probar un controlador que despacha un evento, puedes querer simular los listeners de eventos de modo que realmente no se ejecuten durante la prueba. Esto te permite probar solamente la respuesta HTTP del controlador sin preocuparte por la ejecución de los listeners de eventos, ya que los listeners de eventos pueden ser evaluados en sus propios casos de prueba.

Laravel provee funciones helpers para simular eventos, tareas y clases facades predeterminadas. Estos helpers proporcionan principalmente una capa conveniente sobre la clase Mockery de modo que no tengas que hacer manualmente llamadas complicadas a métodos Mockery. Puedes también usar [Mockery](#) o [PHPUnit](#) para crear tus propios mocks o spies.

Mocking de objetos

Cuando hagas mocking de un objeto que vas a injectar en tu aplicación a través del contenedor de servicio de Laravel, debes enlazar tu instancia a la que le has hecho mocking al contenedor como un enlace de `instance`. Esto le indicará al contenedor que use tu instancia "mockeada" del objeto en lugar de construir el propio objeto:

```
use Mockery;  
use App\Service;  
  
$this->instance(Service::class, Mockery::mock(Service::class, function ($mock) {  
    $mock->shouldReceive('process')->once();  
}));
```

Para hacer esto más conveniente, puedes usar el método `mock`, que es proporcionado por la clase `TestCase` base de Laravel:

```
use App\Service;
```

```
$this->mock(Service::class, function ($mock) {
    $mock->shouldReceive('process')->once();
});
```

Puedes usar el método `partialMock` cuando sólo necesitas simular algunos métodos de un objeto. Los métodos que no son simulados serán ejecutados de forma normal al ser llamados:

```
use App\Service;

$this->partialMock(Service::class, function ($mock) {
    $mock->shouldReceive('process')->once();
});
```

php

De forma similar, si quieres espiar un objeto, la clase de prueba base de Laravel ofrece un método `spy` como un wrapper conveniente del método `Mockery::spy`:

```
use App\Service;

$this->spy(Service::class, function ($mock) {
    $mock->shouldHaveReceived('process');
});
```

php

Fake de trabajos (jobs)

Como una alternativa a mocking, puedes usar el método `fake` de la clase facade `Bus` para evitar que determinadas tareas sean despachadas. Al momento de usar fakes, las aserciones serán hechas después de que el código bajo prueba sea ejecutado.

```
<?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;
```

php

```
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;
        });

        // Comprueba que un trabajo no fue enviado...
        Bus::assertNotDispatched(AnotherJob::class);
    }
}
```

Fake de eventos

Como una alternativa a mocking, puedes usar el método `fake` de la clase facade `Event` para prevenir la ejecución de todos los listeners de eventos. Después puedes comprobar que los eventos fueron despachados e incluso inspeccionar los datos que recibieron. Al momento de usar fakes, las aserciones son hechas después de que el código bajo prueba sea ejecutado:

```
<?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;
    });

    // Comprueba que un evento fue enviado dos veces...
    Event::assertDispatched(OrderShipped::class, 2);

    // Comprueba que un evento no fue enviado...
    Event::assertNotDispatched(OrderFailedToShip::class);
}

}
```

Nota

Después de llamar a `Event::fake()`, no se ejecutarán listeners de eventos. Entonces, si tus pruebas usan model factories que dependen de eventos, cómo crear una UUID durante el evento de modelo `creating`, debes llamar `Event::fake()` **después** de usar tus factories.

Haciendo fake a un subconjunto de eventos

Si sólo si deseas hacer fake a listeners de eventos para un grupo específico de eventos, puedes pasarlo a los métodos `fake` o `fakeFor`:

```
/**
 * Test order process.
 */
public function testOrderProcess()
{
    Event::fake([
        OrderCreated::class,
    ]);

    $order = factory(Order::class)->create();
```

php

```
Event::assertDispatched(OrderCreated::class);

// Otros eventos se envían de forma normal...
$order->update([...]);
}
```

Fake de eventos con alcance

Si sólo quieres hacer fake a oyentes de eventos para una porción de la prueba, se puede usar el método

`fakeFor` :

```
<?php                                         php

namespace Tests\Feature;

use App\Order;
use Tests\TestCase;
use App\Events\OrderCreated;
use Illuminate\Support\Facades\Event;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    /**
     * Test order process.
     */
    public function testOrderProcess()
    {
        $order = Event::fakeFor(function () {
            $order = factory(Order::class)->create();

            Event::assertDispatched(OrderCreated::class);

            return $order;
        });

        // Los eventos se envían normalmente y los observadores se ejecutarán...
        $order->update([...]);
    }
}
```

Fake de correos electrónicos

Puedes usar el método `fake` de la clase facade `Mail` para prevenir que los correos sean enviados. Después puedes comprobar qué [correos de clases mailables](#) fueron enviados a los usuarios e incluso inspeccionar los datos que recibieron. Al momento de usar fakes, las aserciones son hechas después de que el código bajo prueba sea ejecutado.

```
<?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();  
  
        // Comprueba que no se enviaron mailables...  
        Mail::assertNothingSent();  
  
        // Perform order shipping...  
  
        Mail::assertSent(OrderShipped::class, function ($mail) use ($order) {  
            return $mail->order->id === $order->id;  
        });  
  
        // Comprueba que un mensaje fue enviado a los usuarios dados...  
        Mail::assertSent(OrderShipped::class, function ($mail) use ($user) {  
            return $mail->hasTo($user->email) &&  
                $mail->hasCc('...') &&  
                $mail->hasBcc('...');  
        });  
  
        // Comprueba que un correo electrónico fue enviado dos veces...  
        Mail::assertSent(OrderShipped::class, 2);  
    }  
}
```

```
// Comprueba que un correo electrónico no fue enviado...
Mail::assertNotSent(AnotherMailable::class);
}

}
```

Si estás haciendo colas de mailables para su entrega en segundo plano, deberías usar el método `assertQueued` en lugar de `assertSent` :

```
Mail::assertQueued(...);
Mail::assertNotQueued(...);
```

php

Fake de notificaciones

Puedes usar el método `fake` de la clase facade `Notification` para prevenir que se envíen las notificaciones. Después puedes comprobar qué `notificaciones` fueron enviadas a los usuarios e incluso inspeccionar los datos que recibieron. Al momento de usar fakes, las aserciones son hechas después de que el código bajo prueba es ejecutado:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Notifications\OrderShipped;
use Illuminate\Support\Facades\Notification;
use Illuminate\Notifications\AnonymousNotifiable;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithoutMiddleware;

class ExampleTest extends TestCase
{
    public function testOrderShipping()
    {
        Notification::fake();

        // Comprueba que no se enviaron notificaciones...
        Notification::assertNothingSent();
    }
}
```

php

```

// Perform order shipping...

Notification::assertSentTo(
    $user,
    OrderShipped::class,
    function ($notification, $channels) use ($order) {
        return $notification->order->id === $order->id;
    }
);

// Comprueba que una notificación fue enviada a los usuarios dados...
Notification::assertSentTo(
    [$user], OrderShipped::class
);

// Comprueba que una notificación no fue enviada...
Notification::assertNotSentTo(
    [$user], AnotherNotification::class
);

// Comprueba que se envió una notificación mediante el método Notification::route()
Notification::assertSentTo(
    new AnonymousNotifiable, OrderShipped::class
);

// Comprueba que el método Notification::route() envió la notificación a
Notification::assertSentTo(
    new AnonymousNotifiable,
    OrderShipped::class,
    function ($notification, $channels, $notifiable) use ($user) {
        return $notifiable->routes['mail'] === $user->email;
    }
);
}

}

```

Fake de colas

Como alternativa a mocking, puedes usar el método `fake` de la clase facade `Queue` para prevenir que las tareas sean encoladas. Después puedes comprobar que tareas fueron agregadas a la cola e

incluso inspeccionar los datos que recibieron. Al momento de usar fakes, las aserciones son hechas después de que el código bajo prueba es ejecutado:

```
<?php  
  
namespace Tests\Feature;  
  
use Tests\TestCase;  
use App\Jobs\AnotherJob;  
use App\Jobs\FinalJob;  
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();  
  
        // Comprueba que no se agregaron trabajos...  
        Queue::assertNothingPushed();  
  
        // Perform order shipping...  
        Queue::assertPushed(ShipOrder::class, function ($job) use ($order) {  
            return $job->order->id === $order->id;  
        });  
  
        // Comprueba que un trabajo fue agregado a una cola dada...  
        Queue::assertPushedOn('queue-name', ShipOrder::class);  
  
        // Comprueba que un trabajo fue agregado dos veces...  
        Queue::assertPushed(ShipOrder::class, 2);  
  
        // Comprueba que un trabajo no fue agregado...  
        Queue::assertNotPushed(AnotherJob::class);  
  
        // Comprueba que un trabajo fue agregado con una cadena dada de trabajos  
        Queue::assertPushedWithChain(ShipOrder::class, [  
            AnotherJob::class,  
            FinalJob::class  
        ]);  
    }  
}
```

```
// Configuración para trabajos emparejados por clase y propiedades
$expectedAnotherJob = new AnotherJob("foo");
$expectedFinalJob = new FinalJob("bar");

// Comprueba que un trabajo fue agregado con una cadena dada de trabajos
// como propiedades...
Queue::assertPushedWithChain(ShipOrder::class, [
    new AnotherJob('foo'),
    new FinalJob('bar'),
]);
}

}
```

Fake de almacenamiento de archivos

El método fake de la clase facade `Storage` permite que generes fácilmente un disco falso que, combinado con las utilidades de generación de archivo de la clase `UploadedFile`, simplifica mucho la prueba de subidas 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 testAlbumUpload()
    {
        Storage::fake('photos');

        $response = $this->json('POST', '/photos', [
            UploadedFile::fake()->image('photo1.jpg'),
            UploadedFile::fake()->image('photo2.jpg')
        ]);
    }
}
```

```
// Comprueba que uno o más archivos fueron almacenados...
Storage::disk('photos')->assertExists('photo1.jpg');
Storage::disk('photos')->assertExists(['photo1.jpg', 'photo2.jpg']);

// Comprueba que uno o más archivos no fueron almacenados...
Storage::disk('photos')->assertMissing('missing.jpg');
Storage::disk('photos')->assertMissing(['missing.jpg', 'non-existing.jpg'])

}
```

TIP

De forma predeterminada, el método `fake` borrará todos los archivos en su directorio temporal. Si prefieres mantener estos archivos, puedes usar en su lugar el método "persistentFake".

Las clases facade

Diferente de las llamadas de métodos estáticos tradicionales, [las clases facade](#) pueden ser simuladas (mock). Esto proporciona una gran ventaja sobre los métodos estáticos tradicionales y te concede la misma capacidad de prueba que tendrías si estuvieras usando inyección de dependencias. Al momento de probar, con frecuencia puedes querer simular una llamada a una clase facade de Laravel en uno de tus controladores. Por ejemplo, considera la siguiente acción de 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 (mock) la ejecución de la clase facade `Cache` usando el método `shouldReceive`, el cual devolverá una instancia mock de la clase [Mockery](#). Ya que las clases facades realmente son resueltas y administradas por el [contenedor de servicios](#) de Laravel, tendrán mucho más capacidad de prueba que una clase estática típica. Por ejemplo, vamos a simular (mock) nuestra llamada al método `get` de la clase facade `Cache`:

```
<?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');  
  
        // ...  
    }  
}
```

Nota

No deberías hacer mock a la clase facade `Request`. En lugar de eso, pasa la entrada que deseas dentro de los métodos helper HTTP tales como `get` y `post` al momento de ejecutar tus pruebas. Del mismo modo, en lugar de simular (mock) la clase facade `Config`, ejecuta el método `Config::set` en tus pruebas.

Laravel Cashier

- [Introducción](#)
- [Actualizando cashier](#)
- [Instalación](#)
- [Configuración](#)
 - [Migraciones de base de datos](#)
 - [Modelo Billable](#)
 - [API Keys](#)
 - [Configuración de moneda](#)
 - [Webhooks](#)
- [Suscripciones](#)
 - [Creando suscripciones](#)
 - [Verificando el estado de suscripción](#)
 - [Cambiando planes](#)
 - [Cantidad de suscripción](#)
 - [Impuestos de suscripción](#)
 - [Fecha de anclaje de suscripción](#)
 - [Cancelando suscripciones](#)
 - [Reanudando suscripciones](#)
- [Periodos de prueba de suscripción](#)
 - [Con tarjeta de crédito](#)
 - [Sin tarjeta de crédito](#)

- Clientes
 - Creando clientes
- Tarjetas
 - Retornando tarjetas de crédito
 - Determinando si una tarjeta está en el archivo
 - Actualizando tarjetas de crédito
 - Eliminando tarjetas de crédito
- Manejando webhooks de Stripe
 - Definiendo manejadores de eventos de webhooks
 - Suscripciones fallidas
 - Verificando las firmas del webhook
- Cargos únicos
 - Carga simple
 - Carga con factura
 - Reembolsar cargos
- Facturas
 - Generando PDFs de facturas

Introducción

Laravel Cashier proporciona una expresiva interfaz fluida para los servicios de pagos en línea por suscripción de [Stripe](#). Maneja casi todo el código de facturación de suscripción que estás teniendo pavor de escribir. Además de la gestión de suscripción, Cashier puede manejar cupones, cambio de suscripciones, "cantidades" de suscripción, cancelación de períodos de gracia e incluso generar PDFs de facturas.

Nota

Si solamente estás trabajando con cargos de "un pago-único" y no ofreces suscripciones, no deberías usar Cashier. En lugar de eso, usa directamente los SDKs de Stripe.

Actualizando cashier

Al actualizar a una nueva versión mayor de Cashier, es importante que revises cuidadosamente [la guía de actualización](#).

Instalación

Primero, instala el paquete de Cashier para Stripe Con Composer:

```
composer require laravel/cashier
```

php

Configuración

Migraciones de bases de datos

Antes de usar Cashier, también necesitaremos [preparar la base de datos](#). Necesitas agregar varias columnas a tu tabla `users` y crear una nueva tabla `subscriptions` para manejar todas las subscripciones de nuestros clientes:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('stripe_id')->nullable()->collation('utf8mb4_bin');
    $table->string('card_brand')->nullable();
    $table->string('card_last_four', 4)->nullable();
    $table->timestamp('trial_ends_at')->nullable();
});

Schema::create('subscriptions', function (Blueprint $table) {
    $table->bigIncrements('id');
    $table->unsignedBigInteger('user_id');
    $table->string('name');
    $table->string('stripe_id')->collation('utf8mb4_bin');
    $table->string('stripe_plan');
    $table->integer('quantity');
    $table->timestamp('trial_ends_at')->nullable();
    $table->timestamp('ends_at')->nullable();
    $table->timestamps();
});
```

php

Una vez que las migraciones han sido creadas, ejecuta el comando Artisan `migrate`.

Modelo Billable

A continuación, agrega el trait `Billable` a tu definición de modelo. Este trait proporciona varios métodos para permitirte realizar tareas comunes de facturación, tales como creación de subscripciones,

aplicación de cupones y actualización de la información de la tarjeta de crédito:

```
use Laravel\Cashier\Billable;  
  
class User extends Authenticatable  
{  
    use Billable;  
}
```

php

Claves de API

Finalmente, deberías configurar tu clave de Stripe en tu archivo de configuración `services.php`.

Puedes obtener tus claves de API de Stripe desde el panel de control de Stripe:

```
'stripe' => [  
    'model' => App\User::class,  
    'key' => env('STRIPE_KEY'),  
    'secret' => env('STRIPE_SECRET'),  
    'webhook' => [  
        'secret' => env('STRIPE_WEBHOOK_SECRET'),  
        'tolerance' => env('STRIPE_WEBHOOK_TOLERANCE', 300),  
    ],  
],
```

php

Configuración de moneda

La moneda predeterminada de Cashier es Dólares estadounidenses (USD). Puedes cambiar la moneda predeterminada al ejecutar el método `Cashier::useCurrency` dentro del método `boot` de uno de tus proveedores de servicio. El método `Cashier::useCurrency` acepta dos parámetros de cadena: la moneda y el símbolo de la moneda:

```
use Laravel\Cashier\Cashier;  
  
Cashier::useCurrency('eur', '€');
```

php

Webhooks

Para asegurarte de que Cashier maneja apropiadamente todos los eventos de Stripe, recomendamos profundamente [configurar el manejador de webhook de Cashier](#).

Subscripciones

Creando suscripciones

Para crear una suscripción, primero obtén una instancia de tu modelo facturable, el cual será típicamente una instancia de `App\User`. Una vez que has obtenido la instancia de modelo, puedes usar el método `newSubscription` para crear la suscripción del modelo:

```
$user = User::find(1);  
  
$user->newSubscription('main', 'premium')->create($token);
```

El primer argumento pasado al método `newSubscription` debería ser el nombre de la suscripción. Si tu aplicación sólo ofrece una única suscripción, puedes llamarla `main` o `primary`. El segundo argumento es el plan específico al que el usuario se está suscribiendo. Este valor debería corresponder al identificador del plan en Stripe.

El método `create` el cual acepta una tarjeta de crédito / token source de Stripe, comenzará la suscripción al igual que actualizará tu base de datos con el ID del cliente y otra información de facturación relevante.

Detalles de usuario adicionales

Si prefieres especificar detalles de cliente adicionales, puedes hacerlo pasándolos como segundo argumento del método `create`:

```
$user->newSubscription('main', 'monthly')->create($token, [  
    'email' => $email,  
]);
```

Para aprender más sobre los campos adicionales soportados por Stripe, revisa la [documentación sobre la creación de clientes](#).

Cupones

Si prefieres aplicar un cupón al momento de crear la suscripción, puedes usar el método `withCoupon`:

```
$user->newSubscription('main', 'monthly')
    ->withCoupon('code')
    ->create($token);
```

php

Verificando el estado de la suscripción

Una vez que un usuario está suscrito a tu aplicación, puedes verificar fácilmente su estado de suscripción usando una variedad conveniente de métodos. Primero, el método `subscribed` devuelve `true` si el usuario tiene una suscripción activa, incluso si la suscripción está actualmente en su período de prueba:

```
if ($user->subscribed('main')) {
    //
}
```

php

El método `subscribed` también constituye un gran candidato para un [middleware de ruta](#), permitiéndote filtrar el acceso a rutas y controladores basados en el estado de suscripción:

```
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);
}
```

php

Si prefieres determinar si un usuario está aún dentro de su período de prueba, puedes usar el método `onTrial`. Este método puede ser útil para mostrar una advertencia al usuario que todavía está en su período de prueba:

```
if ($user->subscription('main')->onTrial()) {
    //
}
```

php

El método `subscribedToPlan` puede ser usado para determinar si el usuario está suscrito a un plan dado basado en un ID de plan Stripe proporcionado. En este ejemplo, determinaremos si la suscripción `main` del usuario está activa para al plan `monthly` :

```
if ($user->subscribedToPlan('monthly', 'main')) {  
    //  
}
```

php

El método `recurring` puede ser usado para determinar si el usuario está actualmente suscrito y ya no está dentro de su periodo de prueba:

```
if ($user->subscription('main')->recurring()) {  
    //  
}
```

php

Estado de suscripción cancelada

Para determinar si el usuario fue alguna vez un suscriptor activo, pero que ha cancelado su suscripción, puedes usar el método `cancelled` :

```
if ($user->subscription('main')->cancelled()) {  
    //  
}
```

php

También puedes determinar si un usuario ha cancelado su suscripción, pero todavía está en su "periodo de gracia" hasta que la suscripción caduque totalmente. Por ejemplo, si un usuario cancela una suscripción el 5 de Marzo que fue planificada para expirar originalmente el 10 de Marzo, el usuario está en su "periodo de gracia" hasta el 10 de Marzo. Nota que el método `subscribed` aún devuelve `true` durante esta tiempo:

```
if ($user->subscription('main')->onGracePeriod()) {  
    //  
}
```

php

Para determinar si el usuario que ha cancelado su suscripción ya no está dentro del "periodo de gracia", puedes usar el método `ended` :

```
if ($user->subscription('main')->ended()) {  
    //  
}
```

php

Cambiando planes

Después que un usuario esté suscrito en tu aplicación, ocasionalmente puede querer cambiar a un nuevo plan de suscripción. Para cambiar un usuario a una nueva suscripción, pasa el identificador de plan al método `swap` :

```
$user = App\User::find(1);  
  
$user->subscription('main')->swap('provider-plan-id');
```

php

Si el usuario está en período de prueba, se mantendrá el período de prueba. También, si una "cantidad" existe para la suscripción esa cantidad también será conservada.

Si prefieres cambiar planes y cancelar cualquier período de prueba en donde esté el usuario actualmente, puedes usar el método `skipTrial` :

```
$user->subscription('main')  
    ->skipTrial()  
    ->swap('provider-plan-id');
```

php

Cantidad de la suscripción

Algunas veces las suscripciones son afectadas por la "cantidad". Por ejemplo, tu aplicación podría cargar 10\$ por mes **por usuario** en una cuenta. Para incrementar o disminuir fácilmente tu cantidad de suscripción, usa 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);
```

php

```
$user->subscription('main')->decrementQuantity();  
  
// Subtract five to the subscription's current quantity...  
$user->subscription('main')->decrementQuantity(5);
```

Alternativamente, puedes establecer una cantidad específica usando el método `updateQuantity` :

```
$user->subscription('main')->updateQuantity(10);
```

php

El método `noProrate` puede ser usado para actualizar la cantidad de la suscripción sin prorratear los cargos:

```
$user->subscription('main')->noProrate()->updateQuantity(10);
```

php

Para más información sobre cantidades de suscripción, consulta la [documentación de Stripe](#).

Impuestos de suscripción

Para especificar el porcentaje de impuesto que un usuario paga en una suscripción, implementa el método `taxPercentage` en tu modelo facturable y devuelve un valor numérico entre 0 y 100, sin más de 2 posiciones decimales.

```
public function taxPercentage()  
{  
    return 20;  
}
```

php

El método `taxPercentage` le permite aplicar una tasa de impuesto modelo por modelo, lo que puede ser útil para una base de usuarios que abarca varios países y tasas de impuestos.

Nota

El método `taxPercentage` solamente aplica para cargos por suscripción. Si usas Cashier para hacer cargos de "pago único", necesitarás especificar manualmente la tasa de impuesto en ese momento.

Sincronizando los porcentajes del impuesto

Al cambiar el valor returned por el método `taxPercentage`, las configuraciones de impuesto en cualquier suscripción existente del usuario permanecerán igual. Si deseas actualizar el valor del impuesto para un suscripción existente con el valor `taxPercentage` returned, debes llamar al método `syncTaxPercentage` en la instancia de suscripción del usuario:

```
$user->subscription('main')->syncTaxPercentage();
```

php

Fecha de anclaje de la suscripción

Nota

Modificar la fecha de suscripción sólo es soportado por la versión de Stripe de Cashier.

Por defecto, el anclaje del ciclo de facturación es la fecha en que se creó la suscripción o, si se usa un período de prueba, la fecha en que finaliza la prueba. Si deseas modificar la fecha de anclaje de facturación, puedes usar el método `anchorBillingCycleOn`:

```
use App\User;
use Carbon\Carbon;

$user = User::find(1);

$anchor = Carbon::parse('first day of next month');

$user->newSubscription('main', 'premium')
    ->anchorBillingCycleOn($anchor->startOfDay())
    ->create($token);
```

php

Para más información sobre administrar ciclos de facturación, consulta la [documentación del ciclo de facturación de Stripe](#) ↗

Cancelando suscripciones

Para cancelar una suscripción, ejecuta el método `cancel` en la suscripción del usuario:

```
$user->subscription('main')->cancel();
```

php

Cuando una suscripción es cancelada, Cashier establecerá automáticamente la columna `ends_at` en tu base de datos. Esta columna es usada para conocer cuando el método `subscribed` debería empezar, devolviendo `false`. Por ejemplo, si un cliente cancela una suscripción el primero de Marzo, pero la suscripción no estaba planificada para finalizar sino para el 5 de Marzo, el método `subscribed` continuará devolviendo `true` hasta el 5 de Marzo.

Puedes determinar si un usuario ha cancelado su suscripción pero aún está en su "período de gracia" usando el método `onGracePeriod`:

```
if ($user->subscription('main')->onGracePeriod()) {  
    //  
}
```

php

Si deseas cancelar una suscripción inmediatamente, ejecuta el método `cancelNow` en la suscripción del usuario:

```
$user->subscription('main')->cancelNow();
```

php

Reanudando suscripciones

Si un usuario ha cancelado su suscripción y deseas reanudarla, usa el método `resume`. El usuario **debe** estar aún en su período de gracia con el propósito de reanudar una suscripción:

```
$user->subscription('main')->resume();
```

php

Si el usuario cancela una suscripción y después reanuda esa suscripción antes que la suscripción haya expirado completamente, no será facturada inmediatamente. En lugar de eso, su suscripción será reactivada y será facturada en el ciclo de facturación original.

Períodos de prueba de suscripción

Con información anticipada de la tarjeta de crédito

Si prefieres ofrecer períodos de prueba a tus clientes mientras continuas colecciónando información anticipada del método de pago, deberías usar el método `trialDays` al momento de crear tus suscripciones:

```
$user = User::find(1);  
  
$user->newSubscription('main', 'monthly')  
    ->trialDays(10)  
    ->create($token);
```

Este método establecerá la fecha de finalización del período de prueba del registro de suscripción dentro de la base de datos, al igual que le indicará a Stripe a no empezar a facturar al cliente hasta después de esta fecha. Al usar el método `trialDays`, Cashier sobrescribirá cualquier periodo de prueba por defecto configurado para el plan en Stripe.

Nota

Si la suscripción del cliente no es cancelada antes de la fecha de finalización del período de prueba, será cargada tan pronto como expire el período de prueba, así que deberías asegurarte de notificar a tus usuarios de la fecha de finalización de su período de prueba.

El método `trialUntil` te permite proporcionar una instancia `DateTime` para especificar cuando el período de prueba debería terminar:

```
use Carbon\Carbon;  
  
$user->newSubscription('main', 'monthly')  
    ->trialUntil(Carbon::now()->addDays(10))  
    ->create($token);
```

Puedes determinar si el usuario está dentro de su período de prueba utilizando el método `onTrial` de la instancia del usuario o el método `onTrial` de la instancia de suscripción. Los dos ejemplos que siguen son idénticos:

```
if ($user->onTrial('main')) {  
    //  
}
```

```
if ($user->subscription('main')->onTrial()) {  
    //  
}
```

Sin información anticipada de la tarjeta de crédito

Si prefieres ofrecer períodos de prueba sin coleccionar la información anticipada del método de pago del usuario, puedes establecer la columna `trial_ends_at` del registro del usuario con la fecha de finalización del período de prueba deseado. Esto es hecho típicamente durante el registro del usuario:

```
$user = User::create([  
    // Populate other user properties...  
    'trial_ends_at' => now()->addDays(10),  
]);
```

php

Nota

Asegúrate de agregar un mutador de fecha para `trial_ends_at` en tu definición de modelo.

Cashier se refiere a este tipo de período de prueba como un "período de prueba genérico", debido a que no está conectado a ninguna suscripción existente. El método `onTrial` en la instancia `User` devolverá `true` si la fecha actual no es mayor al valor de `trial_ends_at`:

```
if ($user->onTrial()) {  
    // User is within their trial period...  
}
```

php

También puedes usar el método `onGenericTrial` si deseas conocer específicamente que el usuario está dentro de su período de prueba "genérico" y no ha creado una suscripción real todavía:

```
if ($user->onGenericTrial()) {  
    // User is within their "generic" trial period...  
}
```

php

Una vez que estés listo para crear una suscripción real para el usuario, puedes usar el método `newSubscription` como es usual:

```
$user = User::find(1);  
  
$user->newSubscription('main', 'monthly')->create($token);
```

php

Clientes

Creando clientes

Ocasionalmente, puedes desear crear un cliente de Stripe sin iniciar una suscripción. Puedes lograr esto usando el método `createAsStripeCustomer` :

```
$user->createAsStripeCustomer();
```

php

Una vez el cliente ha sido creado en Stripe, puedes iniciar una suscripción en una fecha posterior.

Tarjetas

Recuperando tarjetas de crédito

El método `card` en la instancia del modelo facturable retorna una colección de instancias `Laravel\Cashier\Card` :

```
$cards = $user->cards();
```

php

Para recuperar la tarjeta por defecto, puedes usar el método `defaultCard` :

```
$card = $user->defaultCard();
```

php

Determinando si una tarjeta está en el archivo

Puedes comprobar si un cliente tiene una tarjeta de credito agregada a su cuenta usando el método `hasCardOnFile` :

```
if ($user->hasCardOnFile()) {  
    //  
}
```

php

Actualizando tarjetas de crédito

El método `updateCard` puede ser usado para actualizar la información de tarjeta de crédito de un cliente. Este método acepta un token de Stripe y asignará la nueva tarjeta de crédito como el método de pago por defecto:

```
$user->updateCard($token);
```

php

Para sincronizar tu información de tarjeta con la información de la tarjeta por defecto del cliente en Stripe, puedes usar el método `updateCardFromStripe` :

```
$user->updateCardFromStripe();
```

php

Eliminando tarjetas de crédito

Para eliminar una tarjeta, debes primero recuperar las tarjetas del cliente con el método `cards`. Luego, puedes llamar al método `delete` en la instancia de la tarjeta que deseas eliminar:

```
foreach ($user->cards() as $card) {  
    $card->delete();  
}
```

php

Nota

Si eliminás la tarjeta por defecto, por favor asegurate de que sincronizas la nueva tarjeta por defecto con tu base de datos usando método `updateCardFromStripe`.

El método `deleteCards` eliminará toda la información de la tarjeta almacenada por tu aplicación:

```
$user->deleteCards();
```

php

Nota

Si el usuario tiene una suscripción activa, debes considerar evitar que eliminan la última forma de pago restante.

Manejando webhooks de Stripe

Stripe puede notificar tu aplicación de una variedad de eventos por medio de Webhooks. Para manejar webhooks, define una ruta que apunte al controlador de webhook de Cashier. Este controlador manejará todas las solicitudes de webhook entrantes y despacharlos al método de controlador apropiado.

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

Nota

Una vez que hayas registrado tu ruta, asegúrate de configurar la URL de webhook en tus opciones de configuración de panel de control de Stripe.

De forma predeterminada, este controlador manejará automáticamente la cancelación de suscripciones que tengan demasiados cargos fallidos (como sean definidos por tus opciones de configuración de Stripe), actualizaciones de clientes, eliminaciones de clientes, actualizaciones de suscripciones y cambios de tarjetas de crédito; sin embargo, como vamos a descubrir pronto, puedes extender este controlador para manejar cualquier evento de webhook que quieras.

Nota

Asegurate de proteger las peticiones entrantes con el middleware [webhook de verificación de firma][/billing.html#verifying-webhook-signatures] incluido en Cashier.

Webhooks & Protección CSRF

Ya que los webhooks de Stripe necesitan pasar por alto la [protección CSRF](#) de Laravel, asegurate de listar la URI como una excepción en tu middleware `VerifyCsrfToken` o lista la ruta fuera del grupo de middleware `web` :

```
protected $except = [  
    'stripe/*',  
];
```

php

Definiendo manejadores de eventos de webhooks

Cashier maneja automáticamente la cancelación de suscripción por cargos fallidos, pero si tienes eventos de webhook adicionales que te gustaría manejar, extiende el controlador de Webhook. Tus nombres de métodos deberían corresponder con la convención esperada por Cashier, específicamente, los métodos deberían tener como prefijo `handle` y el nombre "camel case" del webhook que deseas manejar. Por ejemplo, si deseas manejar el webhook `invoice.payment_succeeded`, deberías agregar un método `handleInvoicePaymentSucceeded` al controlador:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;  
  
class WebhookController extends CashierController  
{  
    /**  
     * Handle invoice payment succeeded.  
     *  
     * @param array $payload  
     * @return \Symfony\Component\HttpFoundation\Response  
     */  
    public function handleInvoicePaymentSucceeded($payload)  
    {  
        // Handle The Event  
    }  
}
```

php

Luego, define una ruta a tu controlador de Cashier dentro de tu archivo `routes/web.php` :

```
Route::post(
    'stripe/webhook',
    '\App\Http\Controllers\WebhookController@handleWebhook'
);
```

php

Suscripciones fallidas

¿Qué sucedería si una tarjeta de crédito expira? No importa - Cashier incluye un controlador de Webhook que puede cancelar fácilmente la suscripción del cliente por ti. Como notaste antes, todo lo que necesitas hacer es apuntar una ruta al controlador:

```
Route::post(
    'stripe/webhook',
    '\Laravel\Cashier\Http\Controllers\WebhookController@handleWebhook'
);
```

php

¡Y eso es todo! Los pagos fallidos serán capturados y manejados por el controlador. El controlador cancelará la suscripción del cliente cuando Stripe determina que la suscripción ha fallado (normalmente después de tres intentos de pagos fallidos).

Verificando las firmas de los webhooks

Para asegurar tus webhooks, puedes usar [las firmas de webhook de Stripe](#). Por conveniencia, Cashier automáticamente incluye un middleware que verifica si la petición del webhook de Stripe entrante es válida.

Para habilitar la verificación de webhook, asegurate de que el valor de configuración `stripe.webhook.secret` está establecido en tu archivo de configuración `services`. El valor `secret` del webhook puede ser retornado desde el dashboard de tu cuenta de Stripe.

Cargos únicos

Cargo simple

Nota

El método `charge` acepta la cantidad que prefieras cargar en el **denominador más bajo de la moneda usada por tu aplicación**.

Si desea realizar un "cargo único" en la tarjeta de crédito de un cliente suscrito, puedes usar el método `charge` en una instancia de modelo facturable.

```
// Stripe Accepts Charges In Cents...
$userCharge = $user->charge(100);
```

php

El método `charge` acepta un arreglo como segundo argumento, permitiendo que pases algunas opciones que deseas para la creación de cargo de Stripe subyacente. Consulte la documentación de Stripe sobre las opciones disponibles al crear cargos:

```
$user->charge(100, [
    'custom_option' => $value,
]);
```

php

El método `charge` arrojará una excepción si el cargo falla. Si el cargo es exitoso, la respuesta completa de Stripe será devuelta por el método:

```
try {
    $response = $user->charge(100);
} catch (Exception $e) {
    //
}
```

php

Cargo con factura

Algunas veces puedes necesitar hacer un cargo único pero también generar una factura por el cargo de modo que puedas ofrecer un recibo PDF a tu cliente. El método `invoiceFor` permite que hagas justamente eso. Por ejemplo, vamos a facturar al cliente \$5.00 por una "cuota única":

```
// Stripe Accepts Charges In Cents...
$user->invoiceFor('One Time Fee', 500);
```

php

La factura será cargada inmediatamente contra la tarjeta de crédito del usuario. El método `invoiceFor` también acepta un arreglo como su tercer argumento. Este arreglo contiene las opciones de facturación para el elemento de la factura. El cuarto argumento aceptado por el método es también un arreglo. Este argumento final acepta las opciones de facturación de la factura en sí:

```
$user->invoiceFor('Stickers', 500, [
    'quantity' => 50,
], [
    'tax_percent' => 21,
]);
```

Nota

El método `invoiceFor` creará una factura de Stripe la cual reintentará intentos de facturación fallidos. Si no quieres que las facturas reintenten cargos fallidos, necesitarás cerrarlas usando la API de Stripe después del primer cargo fallido.

Reembolsando cargos

Si necesitas reembolsar un cargo de Stripe, puedes usar el método `refund`. Este método acepta el id del cargo de Stripe como su único argumento:

```
$stripeCharge = $user->charge(100);

$user->refund($stripeCharge->id);
```

Facturas

Puedes obtener fácilmente un arreglo de facturas de modelo facturables usando el método `invoices`:

```
$invoices = $user->invoices();

// Include pending invoices in the results...
$invoices = $user->invoicesIncludingPending();
```

Al momento de listar las facturas para el cliente, puedes usar los métodos helper de factura para mostrar la información de factura relevante. Por ejemplo, puedes querer listar todas las facturas en una tabla, permitiendo que el usuario descargue fácilmente algunas 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>
```

php

Generando PDFs de facturas

Dentro de una ruta o controlador, usa el método `downloadInvoice` para generar una descarga en PDF de la factura. Este método generará automáticamente 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',
    ]);
});
```

php

- Introducción
- Instalación
 - Administrando las instalaciones de ChromeDriver
 - Usando otros navegadores
- Primeros pasos
 - Generando pruebas
 - Ejecutar pruebas
 - Manejo de entorno
 - Creando navegadores
 - Macros de navegador
 - Autenticación
 - Migraciones de base de datos
- Interactuando con elementos
 - Selectores de Dusk
 - Haciendo clic en enlaces
 - Texto, valores y atributos
 - Usando formularios
 - Adjuntando archivos
 - Usando el teclado
 - Usando el ratón
 - Diálogos de JavaScript
 - Alcance de selectores
 - Esperando por elementos
 - Haciendo aserciones de Vue
- Aserciones disponibles
- Páginas
 - Generando páginas
 - Configurando páginas
 - Visitando páginas
 - Selectores abreviados
 - Métodos de página
- Componentes
 - Generando componentes
 - Usando componentes
- Integración continua
 - CircleCI

- Codeship
- Heroku CI
- Travis CI

Introducción

Laravel Dusk proporciona una API de automatización y prueba para navegador expresiva y fácil de usar. De forma predeterminada, Dusk no requiere que instales JDK o Selenium en tu computador. En su lugar, Dusk usa una instalación de [ChromeDriver](#) independiente. Sin embargo, siéntete libre de utilizar cualquier otro driver compatible con Selenium que deseas.

Instalación

Para empezar, debes agregar la dependencia de Composer `laravel/dusk` a tu proyecto:

```
composer require --dev laravel/dusk
```

php

Nota

Si estás registrando manualmente el proveedor de servicio de Dusk, **nunca** deberías registrarlo en tu entorno de producción, ya que hacerlo así podría conducir a que usuarios arbitrarios sean capaces de autenticarse en tu aplicación.

Después de la instalación del paquete Dusk, ejecuta el comando Artisan `dusk:install` :

```
php artisan dusk:install
```

php

Un directorio `Browser` será creado dentro de tu directorio `tests` y contendrá una prueba de ejemplo. Seguido, establece la variable de entorno `APP_URL` en tu archivo `.env`. Este valor debería coincidir con la URL que uses para acceder a tu aplicación en un navegador.

Para ejecutar tus pruebas, usa el comando de Artisan `dusk`. El comando `dusk` acepta cualquier argumento que también sea aceptado por el comando `phpunit` :

```
php artisan dusk
```

php

Si tuviste fallos en las pruebas la última vez que se ejecutó el comando `dusk`, puedes ahorrar tiempo volviendo a ejecutar las pruebas fallidas usando el comando `dusk: fail`:

```
php artisan dusk:fails
```

php

Administrando las instalaciones de ChromeDriver

Si te gustaría instalar una versión diferente de ChromeDriver a la incluida con Laravel Dusk, puedes usar el comando `dusk:chrome-driver`:

```
# Install the latest version of ChromeDriver for your OS...
php artisan dusk:chrome-driver

# Install a given version of ChromeDriver for your OS...
php artisan dusk:chrome-driver 74

# Install a given version of ChromeDriver for all supported OSs...
php artisan dusk:chrome-driver --all
```

php

Nota

Dusk requiere que los binarios de `chromedriver` sean ejecutables. Si tienes problemas para ejecutar Dusk, asegurate de que los binarios sean ejecutables con el siguiente comando: `chmod -R 0755 vendor/laravel/dusk/bin/`.

Usando otros navegadores

De forma predeterminada, Dusk usa Google Chrome y una instalación de [ChromeDriver](#) independiente para ejecutar tus pruebas de navegador. Sin embargo, puedes iniciar tu propio servidor Selenium y ejecutar tus pruebas en cualquier navegador que deseas.

Para empezar, abre tu archivo `tests/DuskTestCase.php`, el cual es el caso de prueba de Dusk básico para tu aplicación. Dentro de este archivo, puedes remover la ejecución del método `startChromeDriver`. Esto evitará que Dusk inicie automáticamente ChromeDriver:

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 * @return void  
 */  
public static function prepare()  
{  
    // static::startChromeDriver();  
}
```

php

Luego de esto, puedes modificar el método `driver` para conectar a la URL y puerto de tu preferencia. Además, puedes modificar las "capacidades deseadas" que deberían ser pasadas al WebDriver:

```
/**  
 * Create the RemoteWebDriver instance.  
 *  
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver  
 */  
protected function driver()  
{  
    return RemoteWebDriver::create(  
        'http://localhost:4444/wd/hub', DesiredCapabilities::phantomjs()  
    );  
}
```

php

Primeros pasos

Generando pruebas

Para generar una prueba de Dusk, usa el comando de Artisan `dusk:make`. La prueba generada será colocada en el directorio `tests/Browser`:

```
php artisan dusk:make LoginTest
```

php

Ejecutando pruebas

Para ejecutar tus pruebas de navegador, usa el comando Artisan `dusk` :

```
php artisan dusk
```

php

Si tuviste fallos en las pruebas la última vez que se ejecutó el comando `dusk`, puedes ahorrar tiempo volviendo a ejecutar las pruebas fallidas usando el comando `dusk: fail` :

```
php artisan dusk:fail
```

php

El comando `dusk` acepta cualquier argumento que sea aceptado normalmente por el administrador de pruebas de PHPUnit, permitiendo que ejecutes solamente las pruebas para un [grupo](#) dado, etc:

```
php artisan dusk --group=foo
```

php

Iniciando manualmente ChromeDriver

De forma predeterminada, Dusk intentará automáticamente iniciar ChromeDriver. Si esto no funciona para tu sistema en particular, puedes iniciar manualmente ChromeDriver antes de ejecutar el comando `dusk`. Si eliges iniciar manualmente ChromeDriver, debes comentar la siguiente línea de tu archivo `tests/DuskTestCase.php` :

```
/**  
 * Prepare for Dusk test execution.  
 *  
 * @beforeClass  
 * @return void  
 */  
public static function prepare()  
{  
    // static::startChromeDriver();  
}
```

php

Además, si inicias ChromeDriver en un puerto diferente a 9515, deberías modificar el método `driver` de la misma clase:

```
/** php
 * Create the RemoteWebDriver instance.
 *
 * @return \Facebook\WebDriver\Remote\RemoteWebDriver
 */
protected function driver()
{
    return RemoteWebDriver::create(
        'http://localhost:9515', DesiredCapabilities::chrome()
    );
}
```

Manejo de entorno

Para forzar que Dusk use su propio archivo de entorno al momento de ejecutar las pruebas, crea un archivo `.env.dusk.{environment}` en el directorio raíz de tu proyecto. Por ejemplo, si estás iniciando el comando `dusk` desde tu entorno `local`, deberías crear un archivo `.env.dusk.local`.

Al momento de ejecutar pruebas, Dusk respaldará tu archivo `.env` y renombrará tu entorno Dusk a `.env`. Una vez que las pruebas han sido completadas, tu archivo `.env` será restaurado.

Creando navegadores

Para empezar, vamos a escribir una prueba que verifica que podemos entrar a nuestra aplicación. Después de generar una prueba, podemos modificarla para visitar la página de login, introducir algunas credenciales y presionar el botón "Login". Para crear una instancia del navegador, ejecuta el método `browse`:

```
<?php php

namespace Tests\Browser;

use App\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Dusk\Chrome;
use Tests\DuskTestCase;

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', 'password')
            ->press('Login')
            ->assertPathIs('/home');
    });
}
}
```

Como puedes ver en el ejemplo anterior, el método `browse` acepta una función callback. Una instancia de navegador será pasada automáticamente a esta función de retorno por Dusk y es el objeto principal utilizado para interactuar y hacer aserciones en la aplicación.

Creando múltiples navegadores

Algunas veces puedes necesitar múltiples navegadores con el propósito de ejecutar apropiadamente una prueba. Por ejemplo, múltiples navegadores pueden ser necesitados para probar una pantalla de conversaciones que interactúa con websockets. Para crear múltiples navegadores, "solicita" más de un navegador en la firma del callback dado al método `browse` :

```
$this->browse(function ($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');
});
```

Redimensionando las ventanas del navegador

Puedes usar el método `resize` para ajustar el tamaño de la ventana del navegador:

```
$browser->resize(1920, 1080);
```

php

El método `maximize` puede ser usado para maximizar la ventana del navegador:

```
$browser->maximize();
```

php

Macros de navegador

Si desea definir un método de navegador personalizado que puedas reutilizar en una variedad de tus pruebas, puedes usar el método `macro` en la clase `Browser`. Normalmente, deberías llamar a este método desde el método `boot` del proveedor de servicios:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Laravel\Dusk\Browser;

class DuskServiceProvider extends ServiceProvider
{
    /**
     * Register the Dusk's browser macros.
     *
     * @return void
     */
    public function boot()
```

php

```
{  
    Browser::macro('scrollToElement', function ($element = null) {  
        $this->script("$( 'html, body' ) .animate( { scrollTop: $($element).of  
  
            return $this;  
        } );  
    }  
}
```

La función `macro` acepta un nombre como primer argumento y un Closure como segundo. El Closure del macro se ejecutará cuando se llame al macro como un método en una implementación de

`Browser` :

```
$this->browse(function ($browser) use ($user) {  
    $browser->visit('/pay')  
        ->scrollToElement('#credit-card-details')  
        ->assertSee('Enter Credit Card Details');  
});
```

php

Autenticación

Frecuentemente, estarás probando páginas que requieren autenticación. Puedes usar el método `loginAs` de Dusk con el propósito de evitar interactuar con la pantalla de login durante cada prueba. El método `loginAs` acepta un ID de usuario o una instancia de modelo de usuario:

```
$this->browse(function ($first, $second) {  
    $first->loginAs(User::find(1))  
        ->visit('/home');  
});
```

php

Nota

Después de usar el método `loginAs`, la sesión de usuario será mantenida para todas las pruebas dentro del archivo.

Migraciones de bases de datos

Cuando tu prueba requiere migraciones, como el ejemplo de autenticación visto antes, nunca deberías usar el trait `RefreshDatabase`. El trait `RefreshDatabase` se apoya en transacciones de base de datos, las cuales no serán aplicables a través de las solicitudes HTTP. En su lugar, usa el trait `DatabaseMigrations`:

```
<?php  
  
namespace Tests\Browser;  
  
use App\User;  
use Illuminate\Foundation\Testing\DatabaseMigrations;  
use Laravel\Dusk\Chrome;  
use Tests\DuskTestCase;  
  
class ExampleTest extends DuskTestCase  
{  
    use DatabaseMigrations;  
}
```

php

Interactuando con elementos

Selectores de Dusk

Elegir buenos selectores CSS para interactuar con elementos es una de las partes más difíciles de escribir las pruebas de Dusk. Con el tiempo, los cambios del diseño frontend pueden causar que los selectores CSS como los siguientes dañen tus pruebas:

```
// HTML...  
  
<button>Login</button>  
  
// Test...  
  
$browser->click('.login-page .container div > button');
```

php

Los selectores de Dusk permiten que te enfoques en la escritura de pruebas efectivas en vez de recordar selectores CSS. Para definir un selector, agrega un atributo `dusk` a tu elemento HTML. Después,

agrega un prefijo al selector con `@` para manipular el elemento conectado dentro de una prueba de Dusk:

```
// HTML...  
  
<button dusk="login-button">Login</button>  
  
// Test...  
  
$browser->click('@login-button');
```

Haciendo clic en enlaces

Para hacer clic sobre un enlace, puedes usar el método `clickLink` en la instancia del navegador. El método `clickLink` hará clic en el enlace que tiene el texto dado en la pantalla:

```
$browser->clickLink($linkText);
```

Nota

Este método interactúa con jQuery. Si jQuery no está disponible en la página, Dusk lo inyectará automáticamente de modo que esté disponible por la duración de la prueba.

Texto, Valores y Atributos

Obteniendo y estableciendo valores

Dusk proporciona varios métodos para interactuar con el texto de pantalla, valor y atributos de elementos en la página actual. Por ejemplo, para obtener el "valor" de un elemento que coincide con un selector dado, usa el método `value` :

```
// Retrieve the value...  
$value = $browser->value('selector');  
  
// Set the value...  
$browser->value('selector', 'value');
```

Obteniendo texto

El método `text` puede ser usado para obtener el texto de pantalla de un elemento que coincide con el selector dado:

```
$text = $browser->text('selector');
```

php

Obteniendo atributos

Finalmente, el método `attribute` puede ser usado para obtener un atributo de un elemento que coincide con el selector dado:

```
$attribute = $browser->attribute('selector', 'value');
```

php

Usando Formularios

Escribiendo valores

Dusk proporciona una variedad de métodos para interactuar con formularios y elementos de entrada. Primero, vamos a echar un vistazo a un ejemplo de escribir texto dentro de un campo de entrada:

```
$browser->type('email', 'taylor@laravel.com');
```

php

Nota que, aunque el método acepta uno si es necesario, no estamos obligados a pasar un selector CSS dentro del método `type`. Si un selector CSS no es proporcionado, Dusk buscará un campo de entrada con el atributo `name` dado. Finalmente, Dusk intentará encontrar un `textarea` con el atributo `name` dado.

Para agregar texto a un campo sin limpiar su contenido, puedes usar el método `append`:

```
$browser->type('tags', 'foo')
    ->append('tags', ' ', bar, baz');
```

php

Puedes limpiar el valor de un campo usando el método `clear`:

```
$browser->clear('email');
```

php

Listas desplegables

Para seleccionar un valor en un cuadro de selección de lista desplegable, puedes usar el método `select`. Al momento de pasar un valor al método `select`, deberías pasar el valor de opción a resaltar en lugar del texto mostrado en pantalla:

```
$browser->select('size', 'Large');
```

php

Puedes seleccionar una opción aleatoria al omitir el segundo parámetro:

```
$browser->select('size');
```

php

Casillas de verificación

Para "marcar" un campo de casilla de verificación, puedes usar el método `check`. Al igual que muchos otros métodos relacionados con entradas, un selector CSS completo no es obligatorio. Si un selector que coincide exactamente no puede ser encontrado, Dusk buscará una casilla de verificación con un atributo `name` coincidente.

```
$browser->check('terms');
```

php

```
$browser->uncheck('terms');
```

Botones de radio

Para "seleccionar" una opción de botón de radio, puedes usar el método `radio`. Al igual que muchos otros métodos relacionados con campos, un selector CSS completo no es obligatorio. Si un selector que coincide exactamente no puede ser encontrado, Dusk buscará un radio con atributos `name` y `value` coincidentes:

```
$browser->radio('version', 'php7');
```

php

Adjuntando archivos

El método `attach` puede ser usado para adjuntar un archivo a un elemento `file`. Al igual que muchos otros métodos relacionados con campos, un selector CSS completo no es obligatorio. Si un selector que coincide exactamente no puede ser encontrado, Dusk buscará un campo de archivo con atributo `name` coincidente:

```
$browser->attach('photo', __DIR__. '/photos/me.png');
```

php

Nota

La función `attach` requiere que la extensión de PHP `Zip` esté instalada y habilitada en tu servidor.

Usando el teclado

El método `keys` permite que proporciones secuencias de entrada más complejas para un elemento dado que lo permitido normalmente por el método `type`. Por ejemplo, puedes mantener presionada las teclas modificadoras al introducir valores. En este ejemplo, la tecla `shift` será mantenida presionada mientras la palabra `taylor` es introducida dentro del elemento que coincide con el selector dado. Después de que la palabra `taylor` sea tipeada, la palabra `otwell` será tipeada sin alguna tecla modificadora:

```
$browser->keys('selector', ['{shift}', 'taylor'], 'otwell');
```

php

Incluso puedes enviar una "tecla de función" al selector CSS principal que contiene tu aplicación:

```
$browser->keys('.app', ['{command}', 'j']);
```

php

TIP TIP

Todas las teclas modificadoras se envuelven entre corchetes `{}` y coinciden con las constantes definidas en la clase `Facebook\WebDriver\WebDriverKeys`, la cual puede ser [encontrada en GitHub](#).

Usando el Ratón

Haciendo clic sobre elementos

El método `click` puede ser usado para "clickear" sobre un elemento que coincide con el selector dado:

```
$browser->click('.selector');
```

php

Mouseover

El método `mouseover` puede ser usado cuando necesitas mover el ratón sobre un elemento que coincide con el selector dado:

```
$browser->mouseover('.selector');
```

php

Arrastrar y soltar

El método `drag` puede ser usado para arrastrar un elemento que coincide con el selector dado hasta otro elemento:

```
$browser->drag('.from-selector', '.to-selector');
```

php

O, puedes arrastrar un elemento en una única dirección:

```
$browser->dragLeft('.selector', 10);
$browser->dragRight('.selector', 10);
$browser->dragUp('.selector', 10);
$browser->dragDown('.selector', 10);
```

php

Diálogos de JavaScript

Dusk provee de varios métodos para interactuar con Diálogos de JavaScript:

```
// Wait for a dialog to appear:
$browser->waitForDialog($seconds = null);
```

php

```
// Assert that a dialog has been displayed and that its message matches the give
$browser->assertDialogOpened('value');
```

```
// Type the given value in an open JavaScript prompt dialog:  
$browser->typeInDialog('Hello World');
```

Para cerrar un Diálogo de JavaScript abierto, haga clic en el botón Aceptar o OK:

```
$browser->acceptDialog();
```

php

Para cerrar un Diálogo de JavaScript abierto, haga clic en el botón Cancelar (solo para un diálogo de confirmación):

```
$browser->dismissDialog();
```

php

Alcance de selectores

Algunas veces puedes querer ejecutar varias operaciones dentro del alcance de un selector dado. Por ejemplo, puedes querer comprobar que algunos textos existen únicamente dentro de una tabla y después presionar un botón dentro de la tabla. Puedes usar el método `with` para completar esta tarea. Todas las operaciones ejecutadas dentro de la función de retorno dada al método `with` serán exploradas en el selector original:

```
$browser->with('.table', function ($table) {  
    $table->assertSee('Hello World')  
        ->clickLink('Delete');  
});
```

php

Esperando por elementos

Al momento de probar aplicaciones que usan JavaScript de forma extensiva, frecuentemente se vuelve necesario "esperar" por ciertos elementos o datos estén disponibles antes de proceder con una prueba. Dusk hace esto fácilmente. Usando una variedad de métodos, puedes esperar que los elementos estén visibles en la página e incluso esperar hasta que una expresión de JavaScript dada se evalúe como `true`.

Esperando

Si necesitas pausar la prueba por un número de milisegundos dado, usa el método `pause` :

```
$browser->pause(1000);
```

php

Esperando por selectores

El método `waitFor` puede ser usado para pausar la ejecución de la prueba hasta que el elemento que coincide con el selector CSS dado sea mostrado en la página. De forma predeterminada, esto pausará la prueba por un máximo de cinco segundos antes de arrojar una excepción. Si es necesario, puedes pasar un umbral de tiempo de expiración personalizado como segundo argumento del 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);
```

php

También puede esperar hasta que el selector dado no se encuentre en la página:

```
$browser->waitForMissing('.selector');

$browser->waitForMissing('.selector', 1);
```

php

Estableciendo el alcance de selectores cuando estén disponibles

Ocasionalmente, puedes querer esperar por un selector dado y después interactuar con el elemento que coincide con el selector. Por ejemplo, puedes querer esperar hasta que una ventana modal esté disponible y después presionar el botón "OK" dentro de esa ventana. El método `whenAvailable` puede ser usado en este caso. Todas las operaciones de elementos ejecutadas dentro de la función de retorno dada serán ejecutadas dentro del selector original:

```
$browser->whenAvailable('.modal', function ($modal) {
    $modal->assertSee('Hello World')
        ->press('OK');
});
```

php

Esperando por texto

El método `waitForText` puede ser usado para esperar hasta que el texto dado sea mostrado 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);
```

php

Esperando por enlaces

El método `waitForLink` puede ser usado para esperar hasta que un enlace dado sea mostrada 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);
```

php

Esperando por la localización de la página

Al momento de hacer una comprobación de ruta tal como `$browser->assertPathIs('/home')`, la comprobación puede fallar si `window.location.pathname` está siendo actualizada asincrónicamente. Puedes usar el método `waitForLocation` para esperar por la localización que tenga un valor dado:

```
$browser->waitForLocation('/secret');
```

php

También puede esperar la localización de una ruta con nombre:

```
$browser->waitForRoute($routeName, $parameters);
```

php

Esperando por recargas de página

Si necesitas hacer aserciones después de que se ha recargado una página, usa el método

`waitForReload` :

```
$browser->click('.some-action')
    ->waitForReload()
    ->assertSee('something');
```

php

Esperando por expresiones de JavaScript

Algunas veces puedes querer pausar la ejecución de una prueba hasta que una expresión de JavaScript dada se evalúe a `true`. Puedes completar fácilmente esto usando el método `waitForUntil`. Al momento de pasar una expresión a este método, no necesitas incluir al final la palabra clave `return` o un punto y coma `;`:

```
// Wait a maximum of five seconds for the expression to be true...
$browser->waitForUntil('App.dataLoaded');

$browser->waitForUntil('App.data.servers.length > 0');

// Wait a maximum of one second for the expression to be true...
$browser->waitForUntil('App.data.servers.length > 0', 1);
```

php

Esperando por expresiones de Vue

Los siguientes métodos puedes ser usados para esperar hasta que un atributo de componente de Vue dado tenga un determinado valor:

```
// Wait until the component attribute contains the given value...
$browser->waitForUntilVue('user.name', 'Taylor', '@user');

// Wait until the component attribute doesn't contain the given value...
$browser->waitForUntilVueIsNot('user.name', null, '@user');
```

php

Esperando por una función de retorno

Muchos de los métodos de "espera" en Dusk confían en el método `waitUsing` subyacente. Puedes usar este método directamente para esperar por una función de retorno dada que devuelva `true`. El

método `waitUsing` acepta el máximo número de segundos para esperar la Closure, el intervalo en el cual la Closure debería ser evaluada y un mensaje opcional de falla:

```
$browser->waitUsing(10, 1, function () use ($something) {  
    return $something->isReady();  
}, "Something wasn't ready in time.");
```

php

Haciendo aserciones de Vue

Inclusive Dusk permite que hagas comprobaciones en el estado de componente de datos de [Vue](#). Por ejemplo, imagina que tu 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'  
            }  
        };  
    }  
});
```

php

Puedes comprobar el estado del componente de Vue de esta manera:

```
/**  
 * A basic Vue test example.  
 *  
 * @return void  
 */  
public function testVue()  
{
```

php

```
$this->browse(function (Browser $browser) {
    $browser->visit('/')
        ->assertVue('user.name', 'Taylor', '@profile-component');
});
```

Aserciones disponibles

Dusk proporciona una variedad de aserciones que puedes hacer en tu aplicación. Todas las aserciones disponibles están documentadas en la tabla de abajo:

assertTitle	assertHasCookie	assertSelected
assertTitleContains	assertCookieMissing	assertNotSelected
assertUrls	assertCookieValue	assertSelectHasOptions
assertSchemes	assertPlainCookieValue	assertSelectMissingOptions
assertSchemesNot	assertSee	assertSelectHasOption
assertHostIs	assertDontSee	assertValue
assertHostIsNot	assertSeeln	assertVisible
assertPortIs	assertDontSeeln	assertPresent
assertPortIsNot	assertSourceHas	assertMissing
assertPathBeginsWith	assertSourceMissing	assertDialogOpened
assertPathIs	assertSeeLink	assertEnabled
assertPathIsNot	assertDontSeeLink	assertDisabled
assertRouteIs	assertInputValue	assertFocused
assertQueryStringHas	assertInputValuesNot	assertNotFocused
assertQueryStringMissing	assertChecked	assertVue
assertFragments	assertNotChecked	assertVuelsNot
assertFragmentBeginsWith	assertRadioSelected	assertVueContains
assertFragmentsNot	assertRadioNotSelected	assertVueDoesNotContain

assertTitle

Comprueba que el título de la página coincide con el texto dado:

```
$browser->assertTitle($title);
```

php

assertTitleContains

Comprueba que el título de página contenga el texto dado:

```
$browser->assertTitleContains($title);
```

php

assertUrls

Comprueba que la URL actual (sin la cadena de consulta) coincida con la cadena dada:

```
$browser->assertUrlIs($url);
```

php

assertSchemeIs

Comprueba que el esquema de la URL actual coincide con el esquema dado:

```
$browser->assertSchemeIs($scheme);
```

php

assertSchemeIsNot

Comprueba que el esquema de la URL actual no coincide con el esquema dado:

```
$browser->assertSchemeIsNot($scheme);
```

php

assertHostIs

Comprueba que el Host de la URL actual coincide con el Host dado:

```
$browser->assertHostIs($host);
```

php

assertHostIsNot

Comprueba que el Host de la URL actual no coincide con el Host dado:

```
$browser->assertHostIsNot($host);
```

php

assertPortIs

Comprueba que el puerto de la URL actual coincide con el puerto dado:

```
$browser->assertPortIs($port);
```

php

assertPortIsNot

Comprueba que el puerto de la URL actual no coincide con el puerto dado:

```
$browser->assertPortIsNot($port);
```

php

assertPathBeginsWith

Comprueba que la ruta de la URL actual comience con la ruta dada:

```
$browser->assertPathBeginsWith($path);
```

php

assertPathIs

Comprueba que la ruta actual coincida con la ruta dada:

```
$browser->assertPathIs('/home');
```

php

assertPathIsNot

Comprueba que la ruta actual no coincide con la ruta dada:

```
$browser->assertPathIsNot('/home');
```

php

assertRouteIs

Comprueba que la URL actual coincide con la URL de ruta nombrada dada:

```
$browser->assertRouteIs($name, $parameters);
```

php

assertQueryStringHas

Comprueba que el parámetro de cadena para una consulta dada está presente:

```
$browser->assertQueryStringHas($name);
```

php

Comprueba que el parámetro de cadena para una consulta dada está presente y tiene un valor dado:

```
$browser->assertQueryStringHas($name, $value);
```

php

assertQueryStringMissing

Comprueba que el parámetro de cadena para una consulta dada está ausente:

```
$browser->assertQueryStringMissing($name);
```

php

assertFragmentIs

Comprueba que el fragmento actual coincide con el fragmento dado:

```
$browser->assertFragmentIs('anchor');
```

php

assertFragmentBeginsWith

Comprueba que el fragmento actual comienza con el fragmento dado:

```
$browser->assertFragmentBeginsWith('anchor');
```

php

assertFragmentIsNot

AComprueba que el fragmento actual no coincide con el fragmento dado:

```
$browser->assertFragmentIsNot('anchor');
```

php

assertHasCookie

Comprueba que el cookie dado está presente:

```
$browser->assertHasCookie($name);
```

php

assertCookieMissing

Comprueba que el cookie dado no está presente:

```
$browser->assertCookieMissing($name);
```

php

assertCookieValue

Comprueba que un cookie tenga un valor dado:

```
$browser->assertCookieValue($name, $value);
```

php

assertPlainCookieValue

Comprueba que un cookie desencriptado tenga un valor dado:

```
$browser->assertPlainCookieValue($name, $value);
```

php

assertSee

Comprueba que el texto dado está presente en la página:

```
$browser->assertSee($text);
```

php

assertDontSee

Comprueba que el texto dado no está presente en la página:

```
$browser->assertDontSee($text);
```

php

assertSeeln

Comprueba que el texto dado está presente dentro del selector:

```
$browser->assertSeeIn($selector, $text);
```

php

assertDontSeeIn

Comprueba que el texto dado no está presente dentro del selector:

```
$browser->assertDontSeeIn($selector, $text);
```

php

assertSourceHas

Comprueba que el código fuente dado está presente en la página:

```
$browser->assertSourceHas($code);
```

php

assertSourceMissing

Comprueba que el código fuente dado no está presente en la página:

```
$browser->assertSourceMissing($code);
```

php

assertSeeLink

Comprueba que el enlace dado está presente en la página:

```
$browser->assertSeeLink($linkText);
```

php

assertDontSeeLink

Comprueba que el enlace dado está no presente en la página:

```
$browser->assertDontSeeLink($linkText);
```

php

assertInputValue

Comprueba que el campo de entrada dado tiene el valor dado:

```
$browser->assertInputValue($field, $value);
```

php

assertInputValueIsNot

Comprueba que el campo de entrada dado no tiene el valor dado:

```
$browser->assertInputValueIsNot($field, $value);
```

php

assertChecked

Comprueba que la casilla de verificación está marcada:

```
$browser->assertChecked($field);
```

php

assertNotChecked

Comprueba que la casilla de verificación no está marcada:

```
$browser->assertNotChecked($field);
```

php

assertRadioSelected

Comprueba que el campo de radio está seleccionado:

```
$browser->assertRadioSelected($field, $value);
```

php

assertRadioNotSelected

Comprueba que el campo de radio no está seleccionado:

```
$browser->assertRadioNotSelected($field, $value);
```

php

assertSelected

Comprueba que la lista desplegable tiene seleccionado el valor dado:

```
$browser->assertSelected($field, $value);
```

php

assertNotSelected

Comprueba que la lista desplegable no tiene seleccionado el valor dado:

```
$browser->assertNotSelected($field, $value);
```

php

assertSelectHasOptions

Comprueba que el arreglo dado de valores están disponibles para ser seleccionados:

```
$browser->assertSelectHasOptions($field, $values);
```

php

assertSelectMissingOptions

Comprueba que el arreglo dado de valores no están disponibles para ser seleccionados:

```
$browser->assertSelectMissingOptions($field, $values);
```

php

assertSelectHasOption

Comprueba que el valor dado está disponible para ser seleccionado en el campo dado:

```
$browser->assertSelectHasOption($field, $value);
```

php

assertValue

Comprueba que el elemento que coincide con el selector dado tenga el valor dado:

```
$browser->assertValue($selector, $value);
```

php

assertVisible

Comprueba que el elemento que coincide con el selector dado sea visible:

```
$browser->assertVisible($selector);
```

php

assertPresent

Comprueba que el elemento que coincide con el selector dado está presente:

```
$browser->assertPresent($selector);
```

php

assertMissing

Comprueba que el elemento que coincide con el selector dado no sea visible:

```
$browser->assertMissing($selector);
```

php

assertDialogOpened

Comprueba que un diálogo JavaScript con un mensaje dado ha sido abierto:

```
$browser->assertDialogOpened($message);
```

php

assertEnabled

Comprueba que el campo dado está activado:

```
$browser->assertEnabled($field);
```

php

assertDisabled

Comprueba que el campo dado está desactivado:

```
$browser->assertDisabled($field);
```

php

assertFocused

Comprueba que el campo dado está enfocado:

```
$browser->assertFocused($field);
```

php

assertNotFocused

Comprueba que el campo dado no está enfocado:

```
$browser->assertNotFocused($field);
```

php

assertVue

Comprueba que una propiedad de datos de un componente de Vue dado coincide con el valor dado:

```
$browser->assertVue($property, $value, $componentSelector = null);
```

php

assertVueIsNot

Comprueba que una propiedad de datos de un componente de Vue dado no coincide con el valor dado:

```
$browser->assertVueIsNot($property, $value, $componentSelector = null);
```

php

assertVueContains

Comprueba que una propiedad de datos de un componente de Vue dado es un arreglo y contiene el valor dado:

```
$browser->assertVueContains($property, $value, $componentSelector = null);
```

php

assertVueDoesNotContain

Comprueba que una propiedad de datos de un componente de Vue dado es un arreglo y no contiene el valor dado:

```
$browser->assertVueDoesNotContain($property, $value, $componentSelector = null);
```

php

Páginas

Alguna veces, las pruebas requieren que varias acciones complicadas sean ejecutadas en secuencia. Esto puede hacer tus pruebas más difíciles de leer y entender. Las páginas permiten que definas acciones expresivas que entonces pueden ser ejecutadas en una página dada usando un solo método. Las páginas también permiten que definas abreviaturas para selectores comunes para tu aplicación o una página única.

Generando páginas

Para generar un objeto de página, usa el comando Artisan `dusk:page`. Todos los objetos de página serán colocados en el directorio `tests/Browser/Pages`:

```
php artisan dusk:page Login
```

php

Configurando páginas

De forma predeterminada, las páginas tienen tres métodos: `url`, `assert`, y `elements`.

Discutiremos los métodos `url` y `assert` ahora. El método `elements` será [discutido con más detalle debajo](#).

El método `url`

El método `url` debería devolver la ruta de la URL que representa la página. Dusk usará esta URL al momento de navegar a la página en el navegador:

```
/**  
 * Get the URL for the page.  
 *  
 * @return string  
 */  
public function url()  
{  
    return '/login';  
}
```

php

El método `assert`

El método `assert` puede hacer algunas aserciones necesarias para verificar que el navegador en realidad está en la página dada. Completar este método no es necesario; sin embargo, eres libre de hacer estas aserciones si lo deseas. Estas aserciones serán ejecutadas automáticamente al momento de navegar hacia la página:

```
/*
 * Assert that the browser is on the page.
 *
 * @return void
 */
public function assert(Browser $browser)
{
    $browser->assertPathIs($this->url());
}
```

php

Navegando hacia las páginas

Una vez que se ha configurado una página, puedes navegar a ella utilizando el método `visit` :

```
use Tests\Browser\Pages\Login;

$browser->visit(new Login);
```

php

A veces es posible que ya estés en una página determinada y necesitas "cargar" los selectores y métodos dentro del contexto de prueba actual. Esto es común al momento de presionar un botón y ser redireccionado a una página dada sin navegar explícitamente a ésta. En esta situación, puedes usar el método `on` para cargar la página.

```
use Tests\Browser\Pages>CreatePlaylist;

$browser->visit('/dashboard')
    ->clickLink('Create Playlist')
    ->on(new CreatePlaylist)
    ->assertSee('@create');
```

php

Selectores abreviados

El método `elements` de páginas permite que definas abreviaturas rápidas, fáciles de recordar para cualquier selector CSS en tu página. Por ejemplo, vamos a definir una abreviación para el campo "email" de la página login de la aplicación:

```
/*
 * Get the element shortcuts for the page.
 *
 * @return array
 */
public function elements()
{
    return [
        '@email' => 'input[name=email]',
    ];
}
```

php

Ahora, puedes usar este selector de abreviación en cualquier lugar que usarías un selector de CSS completo:

```
$browser->type('@email', 'taylor@laravel.com');
```

php

Selectores de abreviaturas globales

Después de instalar Dusk, una clase `Page` básica será colocada en tu directorio

`tests/Browser/Pages`. Esta clase contiene un método `siteElements` el cual puede ser usado para definir selectores de abreviaturas globales que deberían estar disponibles en cada página en cada parte de tu aplicación:

```
/*
 * Get the global element shortcuts for the site.
 *
 * @return array
 */
public static function siteElements()
{
    return [
        '@element' => '#selector',
    ];
}
```

php

Métodos de página

Además de los métodos predeterminados definidos en páginas, puedes definir métodos adicionales, los cuales pueden ser usados en cualquier parte de tus pruebas. Por ejemplo, vamos a imaginar que estamos construyendo una aplicación para administración de música. Una acción común para una página de la aplicación podría ser crear una lista de reproducción. En lugar de volver a escribir la lógica para crear una lista de reproducción en cada prueba, puedes definir un método `createPlaylist` en una clase de página:

```
<?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');  
    }  
}
```

Una vez que el método ha sido definido, puedes usarlo dentro de cualquier prueba que utilice la página. La instancia de navegador será pasada automáticamente al método de la página:

```
use Tests\Browser\Pages\Dashboard;
```

```
$browser->visit(new Dashboard)
    ->createPlaylist('My Playlist')
    ->assertSee('My Playlist');
```

Componentes

Los componentes son similares a “los objetos de página” de Dusk, pero son planeados para partes de UI y funcionalidades que sean reusadas en otras partes de tu aplicación, tal como una barra de navegación o ventana de notificación. Como tal, los componentes no son enlazados a URLs específicas.

Generando componentes

Para generar un componente, usa el comando Artisan `dusk:component`. Los nuevos componentes son colocados en el directorio `test/Browser/Components`:

```
php artisan dusk:component DatePicker
```

php

Como se muestra antes, un “calendario” es un ejemplo de un componente que puede existir en cualquier parte de tu aplicación en una variedad de páginas. Puede volverse complejo escribir manualmente lógica de automatización de navegador para seleccionar una fecha entre docenas de pruebas en cualquier parte de tu software de prueba. En lugar de esto, podemos definir un componente de Dusk para representar el calendario, permitiendo encapsular esa lógica dentro del componente:

```
<?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()
```

php

```
{
    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 $day
 * @return void
 */
public function selectDate($browser, $month, $day)
{
    $browser->click('@date-field')
        ->within('@month-list', function ($browser) use ($month) {
            $browser->click($month);
        })
        ->within('@day-list', function ($browser) use ($day) {

```

```
        $browser->click($day);
    });
}
}
```

Usando componentes

Una vez que el componente ha sido definido, fácilmente podemos seleccionar una fecha dentro del calendario desde cualquier prueba. Y, si la lógica necesaria para seleccionar una fecha cambia, solamente necesitaremos actualizar el componente:

```
<?php  
  
namespace Tests\Browser;  
  
use Illuminate\Foundation\Testing\DatabaseMigrations;  
use Laravel\Dusk\Browser;  
use Tests\Browser\Components\DatePicker;  
use Tests\DuskTestCase;  
  
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');  
        });  
    }  
}
```

php

Integración continua

CircleCI

Si estás usando CircleCI para ejecutar tus pruebas de Dusk, puedes usar este archivo de configuración como punto de partida. Al igual que con TravisCI, usaremos el comando `php artisan serve` para ejecutar el servidor web integrado de PHP:

```
version: 2
jobs:
  build:
    steps:
      - run: sudo apt-get install -y libsqlite3-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

      - store_artifacts:
          path: tests/Browser/screenshots
```

Codeship

Para ejecutar pruebas de Dusk en [Codeship](#), agrega los siguientes comandos a tu proyecto Codeship. Estos comandos son sólo un punto de partida y eres libre de agregar los comandos adicionales que necesites:

```
phpenv local 7.2
cp .env.testing .env
mkdir -p ./bootstrap/cache
composer install --no-interaction --prefer-dist
php artisan key:generate
nohup bash -c "php artisan serve 2>&1 &" && sleep 5
php artisan dusk
```

php

Heroku CI

Para ejecutar tus pruebas de Dusk en [Heroku CI](#), agrega el siguiente buildpack de Google Chrome y scripts a tu archivo `app.json` de Heroku:

```
{
  "environments": {
    "test": {
      "buildpacks": [
        { "url": "heroku/php" },
        { "url": "https://github.com/heroku/heroku-buildpack-google-chro
      ],
      "scripts": {
        "test-setup": "cp .env.testing .env",
        "test": "nohup bash -c './vendor/laravel/dusk/bin/chromedriver-1
      }
    }
  }
}
```

php

Travis CI

Para ejecutar tus pruebas de Dusk en [Travis CI](#), usa la siguiente configuración en el archivo

`.travis.yml`. Ya que Travis CI no es un entorno gráfico, necesitaremos tomar algunos pasos extras con el propósito de ejecutar un navegador Chrome. En adición a esto, usaremos `php artisan serve` para ejecutar el servidor web integrado de PHP:

```
language: php
php:
```

php

```
- 7.3

addons:
  chrome: stable

install:
  - cp .env.testing .env
  - travis_retry composer install --no-interaction --prefer-dist --no-suggest
  - php artisan key:generate

before_script:
  - google-chrome-stable --headless --disable-gpu --remote-debugging-port=9222
  - php artisan serve &

script:
  - php artisan dusk
```

En tu archivo `.env.testing`, ajusta el valor de `APP_URL`:

```
APP_URL=http://127.0.0.1:8000
```

php

GitHub Actions

Si estás usando [acciones de GitHub](#) para ejecutar tus pruebas de Dusk, puedes usar este archivo de configuración como punto de partida. Igual que TravisCI, usaremos el comando `php artisan serve` para ejecutar el servidor integrado de PHP:

```
name: CI
on: [push]
jobs:

dusk-php:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v1
    - name: Prepare The Environment
      run: cp .env.example .env
    - name: Create Database
      run: mysql --user="root" --password="root" -e "CREATE DATABASE my-database"
    - name: Install Composer Dependencies
```

```
run: composer install --no-progress --no-suggest --prefer-dist --optimize--  
- name: Generate Application Key  
  run: php artisan key:generate  
- name: Upgrade Chrome Driver  
  run: php artisan dusk:chrome-driver  
- name: Start Chrome Driver  
  run: ./vendor/laravel/dusk/bin/chromedriver-linux > /dev/null 2>&1 &  
- name: Run Laravel Server  
  run: php artisan serve > /dev/null 2>&1 &  
- name: Run Dusk Tests  
  run: php artisan dusk
```

En tu archivo `.env.testing`, ajusta el valor de `APP_URL`:

```
APP_URL=http://127.0.0.1:8000
```

php

Laravel Envoy

- [Introducción](#)
 - [Instalación](#)
- [Escribir tareas](#)
 - [Setup](#)
 - [Variables](#)
 - [Historias](#)
 - [Múltiples servidores](#)
- [Ejecutar tareas](#)
 - [Confirmar ejecución de tarea](#)
- [Notificaciones](#)
 - [Slack](#)

- o Discord

Introducción

Laravel Envoy proporciona una sintaxis limpia y mínima para definir las tareas comunes que ejecutas en tus servidores remotos. Utilizando el estilo de sintaxis de Blade, puedes configurar fácilmente tareas para deploy, comandos de Artisan y más. Envoy solamente es compatible con sistemas operativos Mac y Linux.

Instalación

Primero, instala Envoy utilizando el comando de composer `global require` :

```
composer global require laravel/envoy
```

php

Dado que las librerías globales de Composer ocasionalmente pueden causar conflictos en la versión del paquete, puedes considerar utilizar `cgr`, el cual es un reemplazo directo para el comando `composer global require`. Las instrucciones de instalación de la librería `gcr` pueden ser [encontradas en GitHub](#).

Nota

Asegurate de colocar el directorio `~/.composer/vendor/bin` en tu PATH para que el ejecutable `envoy` pueda ser localizado cuando se ejecute el comando `envoy` en tu terminal.

Actualizar envoy

También puedes usar Composer para mantener tu instalación de Envoy actualizada. Ejecutar el comando `composer global update` actualizará todos tus paquetes de Composer instalados globalmente:

```
composer global update
```

php

Escribir tareas

Todas tus tareas de Envoy deberán definirse en un archivo `Envoy.blade.php` en la raíz de tu proyecto. Aquí un ejemplo para comenzar:

```
@servers(['web' => ['user@192.168.1.1']])  
  
@task('foo', ['on' => 'web'])  
    ls -la  
@endtask
```

Como puedes ver, un arreglo `@servers` es definido en la parte superior del archivo, permitiéndote hacer referencia a estos servidores en la opción `on` en la declaración de tus tareas. Dentro de tus declaraciones `@task`, deberás colocar el código Bash que se deberá ejecutar en tu servidor una vez que la tarea sea ejecutada.

Puedes forzar que un script se ejecute localmente especificando la dirección IP del servidor como `127.0.0.1`:

```
@servers(['localhost' => '127.0.0.1'])
```

Setup

En ocasiones, puede que necesites ejecutar algún código PHP antes de tus tareas de Envoy. Puedes hacer uso de la directiva `@setup` para declarar variables y hacer uso de PHP en general antes de que tus otras tareas sean ejecutadas:

```
@setup  
    $now = new DateTime();  
  
    $environment = isset($env) ? $env : "testing";  
@endsetup
```

Si necesitas de otros archivos PHP antes de ejecutar tus tareas, puedes utilizar la directiva `@include` en la parte superior de tu archivo `Envoy.blade.php`:

```
@include('vendor/autoload.php')  
  
@task('foo')
```

```
# ...
@endtask
```

También puedes importar otros archivos de Envoy para que así sus historias y tareas sean agregadas a los tuyos. Luego de que han sido importados, puedes ejecutar las tareas en dichos archivos como si estuvieran definidas en los tuyos. Debes usar la directiva `@import` al principio de tu archivo

`Envoy.blade.php` :

```
@import('package/Envoy.blade.php')
```

php

Variables

Si es necesario, puedes pasar valores de opciones a las tareas de Envoy usando la línea de comandos:

```
envoy run deploy --branch=master
```

php

Puedes acceder a las opciones en tus tareas por medio de la sintaxis "echo" de Blade. También puedes usar declaraciones `if` y bucles dentro de tus tareas. Por ejemplo, para verificar la presencia de la 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
```

php

Historias

Las historias agrupan un conjunto de tareas con un nombre único y conveniente, permitiendo agrupar tareas pequeñas enfocándose en tareas más grandes. Por ejemplo, una historia `deploy` puede ejecutar las tareas `git` y `composer` al listar los nombres de las tareas en tu definición:

```
@servers(['web' => '192.168.1.1'])
```

php

```
@story('deploy')
```

```
    git
```

```
    composer
```

```
@endstory
```

```
@task('git')
```

```
    git pull origin master
```

```
@endtask
```

```
@task('composer')
```

```
    composer install
```

```
@endtask
```

Una vez que hayas finalizado de escribir tu historia, puedes ejecutarla como una tarea típica:

```
envoy run deploy
```

php

Múltiples servidores

Envoy te permite fácilmente ejecutar tareas a través de múltiples servidores. Primero, agrega servidores adicionales a tu declaración `@servers`. A cada servidor se le debe asignar un nombre único. Una vez definidos los servidores adicionales, deberás indicar en cuáles servidores se van a ejecutar las tareas, esto puede hacerse en el arreglo `on` de cada tarea:

```
@servers(['web-1' => '192.168.1.1', 'web-2' => '192.168.1.2'])
```

php

```
@task('deploy', ['on' => ['web-1', 'web-2']])
```

```
    cd site
```

```
    git pull origin {{ $branch }}
```

```
    php artisan migrate
```

```
@endtask
```

Ejecución paralela

Por defecto, las tareas serán ejecutadas en cada servidor en serie. En otras palabras, una tarea finaliza su ejecución en el primer servidor antes de proceder a ejecutarse en el segundo servidor. Si deseas ejecutar

una tarea a través de múltiples servidores en paralelo, agrega la opción `parallel` a la declaración de tu tarea:

```
@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
```

php

Ejecutar tareas

Para ejecutar una tarea o historia que esté definida en tu archivo `Envoy.blade.php`, ejecuta el comando de Envoy `run`, pasando el nombre de la tarea o historia que deseas ejecutar. Envoy va a ejecutar la tarea y mostrará el resultado de los servidores mientras se ejecuta la tarea:

```
envoy run deploy
```

php

Confirmar ejecución de tarea

Si deseas que se solicite confirmación antes de ejecutar una tarea en tus servidores, deberás añadir la directiva `confirm` a la declaración de tu 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
```

php

Notificaciones

Slack

Envoy también permite enviar notificaciones a [Slack](#) después de ejecutar cada tarea. La directiva `@slack` acepta una URL de webhook a Slack y un nombre de canal. Puedes recuperar tu URL de webhook creando una integración "Incoming WebHooks" en el panel de control de Slack. Debes pasar la URL de webhook completa en la directiva `@slack` :

```
@finished  
  @slack('webhook-url', '#bots')  
@endfinished
```

php

Puedes proporcionar uno de los siguientes como el argumento del canal:

- Para enviar notificaciones a un canal: `#canal`
- Para enviar notificaciones a un usuario: `@usuario`

Discord

Envoy también soporta el envío de notificaciones a [Discord](#) después de que cada tarea es ejecutada. La directiva `@discord` acepta una URL WebHook y un mensaje de Discord. Puedes recuperar tu URL webhook creando una "Webhook" en los ajustes de tu servidor y seleccionando en cuál canal publicar la webhook. También deberías pasar la URL de Webhook completa en la directiva `@discord` :

```
@finished  
  @discord('discord-webhook-url')  
@endfinished
```

php

Laravel Horizon

- [Introducción](#)
- [Actualización de Horizon](#)

- Instalación
 - Configuración
 - Autorización del dashboard
- Ejecutando Horizon
 - Usando Horizon
- Etiquetas
- Notificaciones
- Métricas

Introducción

Horizon proporciona un bonito panel de control y sistema de configuración controlado por código para Laravel, potenciado por colas de Redis. Horizon te permite monitorear fácilmente métricas clave de tu sistema de colas tales como tasa de rendimiento, tiempo de ejecución y fallas de tareas.

Toda la configuración de tu worker es almacenada en un solo archivo de configuración sencillo, permitiendo que tu configuración quede en el código fuente donde tu equipo completo pueda colaborar.

The screenshot shows the Laravel Horizon dashboard with the following sections:

- Overview:** A summary table with the following data:

JOBS PER MINUTE	JOBS PAST HOUR	FAILED JOBS PAST HOUR	STATUS
50	100	0	Active

TOTAL PROCESSES	MAX WAIT TIME	MAX RUNTIME	MAX THROUGHPUT
3	-	default	default
- Current Workload:** A table showing the current state of the default queue:

Queue	Processes	Jobs	Wait
default	3	0	A Few Seconds
- risa-MaDd:** A table showing supervisor details:

Supervisor	Processes	Queues	Balancing
supervisor-1	3	default	✓ (Simple)

At the bottom, a footer note reads: "Laravel is a trademark of Taylor Otwell. Copyright © Laravel LLC. All rights reserved."

Instalación

Nota

Debes asegurarte de que tu conexión de cola está establecido a `redis` en tu archivo de configuración `queue`.

Puedes usar Composer para instalar Horizon en tu proyecto de Laravel:

```
composer require laravel/horizon
```

php

Después de instalar Horizon, publica sus assets usando el comando Artisan `horizon:install`:

```
php artisan horizon:install
```

php

Debes también crear la tabla `failed_jobs` que Laravel usará para almacenar cualquier `trabajo en cola fallido`:

```
php artisan queue:failed-table
```

php

```
php artisan migrate
```

Actualización de horizon

Al actualizar a una nueva versión mayor de Horizon, es importante que revises cuidadosamente [la guía de actualización](#).

Además, debes volver a publicar los assets de Horizon:

```
php artisan horizon:assets
```

php

Configuración

Después de publicar los assets de Horizon, su principal archivo de configuración será colocado en `config/horizon.php`. Este archivo de configuración permite que configures las opciones del worker

y cada opción de configuración incluye una descripción de su propósito, así que asegurate de explorar con gran detalle este archivo.

Nota

Debes asegurarte de que la porción `environments` de tu archivo de configuración `horizon` contiene una entrada para cada entorno en el que planeas ejecutar Horizon.

Opciones de balance

Horizon permite que elijas entre tres estrategias de balance: `simple`, `auto` y `false`. La estrategia `simple`, que es la opción por defecto del archivo de configuración, divide los trabajos entrantes de manera uniforme entre procesos:

```
'balance' => 'simple',
```

php

La estrategia `auto` ajusta el número de procesos trabajadores por cola basado en la carga de trabajo de la cola. Por ejemplo, si tu cola `notifications` tiene 1.000 trabajos esperando mientras tu cola `render` está vacía, Horizon asignará más trabajadores a tu cola `notifications` hasta que esté vacía. Cuando la opción `balance` esté establecida a `false`, el comportamiento predeterminado de Laravel será usado, el cual procesa las colas en el orden que son listadas en tu configuración.

Recorte de trabajos

El archivo de configuración `horizon` te permite configurar cuánto tiempo los trabajos de recientes y fallidos deben ser persistidos (en minutos). Por defecto, los trabajos recientes son mantenidos por una hora mientras que los trabajos fallidos son mantenidos por una semana:

```
'trim' => [
    'recent' => 60,
    'failed' => 10080,
],
```

php

Autorización del dashboard

Horizon muestra un dashboard o panel de control en `/horizon`. Por defecto, sólo serás capaz de acceder a este dashboard en el entorno `local`. Dentro de tu archivo

`app/Providers/HorizonServiceProvider.php`, hay un método `gate`. Este gate de autorización controla el acceso a Horizon en entornos **no locales**. Eres libre de modificar este gate como sea necesario para restringir el acceso a tu instalación de Horizon:

```
/*
 * Register the Horizon gate.
 *
 * This gate determines who can access Horizon in non-local environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewHorizon', function ($user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

php

Nota

Recuerda que Laravel inyecta el usuario *autenticado* al Gate de forma automática. Si tu aplicación está proporcionando seguridad de Horizon mediante algún otro método, como restricciones de IP, entonces tus usuarios de Horizon pueden no necesitar iniciar sesión. Por lo tanto, necesitarás cambiar el `function ($user)` de arriba por `function ($user = null)` para forzar a Laravel a que no requiera autenticación.

Ejecutando Horizon

Una vez que has configurado tus workers en el archivo de configuración `config/horizon.php`, puedes ejecutar Horizon usando el comando Artisan `horizon`. Este único comando iniciará todos tus workers configurados:

```
php artisan horizon
```

php

Puedes pausar los procesos de Horizon e instruirlo para continuar procesando trabajos usando los comandos Artisan `horizon:pause` y `horizon:continue`:

```
php artisan horizon:pause
```

php

```
php artisan horizon:continue
```

Puedes terminar elegantemente el proceso maestro de Horizon en tu máquina usando el comando Artisan `horizon:terminate`. Cualquiera de los trabajos que Horizon esté procesando actualmente será completado y después Horizon parará:

```
php artisan horizon:terminate
```

php

Usando Horizon

Si estás usando Horizon en un servidor activo, deberías configurar un monitor de proceso para monitorear el comando `php artisan horizon` y reiniciarlo si éste sale inesperadamente. Al momento de usar código reciente en tu servidor, necesitarás instruir el proceso maestro de Horizon para que termine así puede ser reiniciado por tu monitor de proceso y recibir tu cambios de código.

Configuración de Supervisor

Si estás usando el monitor de procesos de Supervisor para administrar tu proceso `horizon`, el siguiente archivo de configuración debería 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
```

php

TIP

Si no estás cómodo administrando tus propios servidores, considera usar [Laravel Forge](#). Forge aprovisiona tus propios servidores PHP 7+ con todo lo que necesitas para administrar modernas aplicaciones robustas de Laravel con Horizon.

Etiquetas

Horizon permite que asigne “etiquetas” a los trabajos, incluyendo correos válidos, difusiones de eventos, notificaciones y listeners de eventos encolados. De hecho, Horizon etiquetará inteligente y automáticamente la mayoría de los trabajos dependiendo de los modelos Eloquent que estén adjuntos al trabajo. Por ejemplo, echemos un vistazo al siguiente worker:

```
<?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 es encolado con una instancia `App\Video` que tenga un `id` de `1`, recibirá automáticamente la etiqueta `App\Video:1`. Esto es debido a que Horizon examinará las propiedades del trabajo para cualquier modelo Eloquent. Si los modelos Eloquent son encontrados, Horizon etiquetará inteligentemente el trabajo usando el nombre de la clase y la clave primaria del modelo.

```
$video = App\Video::find(1);  
  
App\Jobs\RenderVideo::dispatch($video);
```

php

Etiquetado manual

Si prefieres definir manualmente las etiquetas para uno de tus objetos encolables, puedes 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];  
    }  
}
```

php

```
    }  
}
```

Notificaciones

Note: Al momento de configurar Horizon para enviar notificaciones de Slack o SMS, también deberías revisar los [prerequisitos para el manejador de notificaciones relevante](#).

Si prefieres ser notificado cuando una de tus colas tenga un largo tiempo de inactividad, puedes usar los métodos `Horizon::routeMailNotificationsTo`, `Horizon::routeSlackNotificationsTo` y `Horizon::routeSmsNotificationsTo`. Puedes ejecutar estos métodos desde el `HorizonServiceProvider` de tu aplicación:

```
Horizon::routeMailNotificationsTo('example@example.com');  
Horizon::routeSlackNotificationsTo('slack-webhook-url', '#channel');  
Horizon::routeSmsNotificationsTo('15556667777');
```

php

Configurando las notificaciones de umbrales de tiempo de inactividad

Puedes configurar cuántos segundos son considerados un "tiempo de inactividad" dentro de tu archivo de configuración `config/horizon.php`. La opción de configuración `waits` dentro de este archivo permite que controles el umbral de tiempo de inactividad para cada combinación conexión / cola:

```
'waits' => [  
    'redis:default' => 60,  
],
```

php

Métricas

Horizon incluye un panel de métricas, el cual proporciona información de tus tiempos de trabajo y de espera en cola y tasa de rendimiento. Con el propósito de agregar contenido a este panel, deberías configurar el comando Artisan `snapshot` de Horizon para que se ejecute cada 5 minutos por medio del [planificador](#) de tu aplicación:

```
/**  
 * Define the application's command schedule.
```

php

```
* @param \Illuminate\Console\Scheduling\Schedule $schedule
* @return void
*/
protected function schedule(Schedule $schedule)
{
    $schedule->command('horizon:snapshot')->everyFiveMinutes();
}
```

Laravel Passport

- Introducción
- Actualizando Passport
- Instalación
 - Inicio rápido de frontend
 - Despliegue de passport
- Configuración
 - Duración de tokens
 - Sobrescribiendo modelos predeterminados
- Emitiendo tokens de acceso
 - Administrando clientes
 - Solicitando tokens
 - Actualización de tokens
- Tokens de permiso de contraseña
 - Creando un cliente con permiso de contraseña
 - Solicitando tokens
 - Solicitando todos los alcances
 - Personalizando el campo username
- Tokens de permiso implícitos
- Tokens de permiso de credenciales de cliente

- Tokens de acceso personal
 - Creando un cliente de acceso personal
 - Administrando tokens de acceso personal
- Protegiendo rutas
 - Por medio de middleware
 - Pasando el token de acceso
- Alcances de token
 - Definiendo alcances
 - Alcance predeterminado
 - Asignando alcances a los Tokens
 - Verificando alcances
- Consumiendo tu API con JavaScript
- Eventos
- Pruebas

Introducción

Laravel ya hace fácil ejecutar la autenticación por medio de los tradicionales formularios de inicio de sesión, pero ¿Qué información tenemos sobre APIs? Las APIs típicamente usan tokens para autenticar a los usuarios y no mantienen el estado de sesión entre solicitudes. Laravel hace de la autenticación de API algo muy simple usando Passport de Laravel, el cual proporciona una implementación de servidor OAuth2 completa para tu aplicación Laravel en sólo minutos. Passport está construido sobre el [servidor OAuth2](#) que es mantenido por Andy Millington y Simon Hamp..

Nota

Esta documentación asume que estás familiarizado con OAuth2. Si no sabes nada sobre OAuth2, considera familiarizarte con la terminología general y las características de OAuth2 antes de continuar.

Actualizando Passport

Al actualizar a una nueva versión mayor de Passport, es importante que revises detalladamente [la guía de actualización](#).

Instalación

Para empezar, instala Passport por medio del gestor de paquetes Composer:

```
composer require laravel/passport
```

php

El proveedor de servicio de Passport registra su propio directorio de migración de base de datos con el framework, así que deberías migrar tu base de datos después de registrar el paquete. Las migraciones de Passport crearán las tablas que tu aplicación necesita para almacenar clientes y tokens de acceso:

```
php artisan migrate
```

php

A continuación, debes ejecutar el comando `passport:install`. Este comando creará las claves de encriptación necesarias para generar tokens de acceso seguro. Además, el comando creará clientes de "acceso personal" y "permiso de contraseña" los cuales serán usados para generar tokens de acceso:

```
php artisan passport:install
```

php

Después de ejecutar este comando, agrega el trait `Laravel\Passport\HasApiTokens` a tu modelo `App\User`. Este trait proporcionará algunos métodos helper para tu modelo los cuales permitirán que inspecciones el token y alcances del usuario autenticado:

```
<?php  
  
namespace App;  
  
use Illuminate\Foundation\Auth\User as Authenticatable;  
use Illuminate\Notifications\Notifiable;  
use Laravel\Passport\HasApiTokens;  
  
class User extends Authenticatable  
{  
    use HasApiTokens, Notifiable;  
}
```

php

Lo próximo, deberías ejecutar el método `Passport::routes` dentro del método `boot` de tu `AuthServiceProvider`. Este método registrará las rutas necesarias para suministrar tokens y revocar tokens de acceso, clientes y tokens de acceso personal:

php

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;
use Laravel\Passport\Passport;

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();
    }
}
```

Finalmente, en tu archivo de configuración `config/auth.php`, debes establecer la opción `driver` del guardia de autenticación de `api` a `passport`. Esto indicará a tu aplicación que utilice el `TokenGuard` de Passport al momento de autenticar las solicitudes de API entrantes:

php

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

```
'api' => [
    'driver' => 'passport',
    'provider' => 'users',
],
],
```

Personalización de la migración

Si no vas a utilizar las migraciones predeterminadas de Passport, debes llamar al método `Passport::ignoreMigrations` en el método `register` de tu `AppServiceProvider`. Puedes exportar las migraciones por defecto usando `php artisan vendor:publish --tag=passport-migrations`.

Por defecto, Passport usa una columna de enteros para almacenar el `user_id`. Si tu aplicación utiliza un tipo de columna diferente para identificar a los usuarios (por ejemplo: UUID), debes modificar las migraciones de Passport predeterminadas después de publicarlas.

Inicio rápido de frontend

Nota

Para usar los componentes de Vue, debes estar usando el framework de JavaScript [Vue](#). Estos componentes también usarán el framework de CSS Bootstrap. Sin embargo, incluso si no estás usando estas herramientas, los componentes sirven como una referencia valiosa para tu propia implementación de frontend.

Passport viene con una API JSON que puedes usar para permitir que tus usuarios creen tokens de acceso de clientes y personal. Sin embargo, puede ser que consuma tiempo codificar un frontend para interactuar con estas APIs. Así que, Passport también incluye componentes de [Vue](#) pre-construidos que puedes usar como implementación de ejemplo o punto de inicio para tu propia implementación.

Para publicar los componentes de Vue de Passport, usa el comando Artisan `vendor:publish`:

```
php artisan vendor:publish --tag=passport-components
```

php

Los componentes publicados serán colocados en tu directorio `resources/js/components` . Una vez que los componentes han sido publicados, debes registrarlos en tu archivo `resources/js/app.js` :

```
Vue.component('passport-clients', require('./components/passport/Clients.vue').default);  
  
Vue.component('passport-authorized-clients', require('./components/passport/AuthorizedClients.vue').default);  
  
Vue.component('passport-personal-access-tokens', require('./components/passport/PersonalAccessTokens.vue').default);
```

Nota

Antes de Laravel v 5.7.19, anexar `.default` al registrar componentes da como resultado un error de consola. Una explicación para este cambio puedes encontrarla en las notas de lanzamiento de Laravel Mix v 4.0.0 ↗.

Después de registrar los componentes, asegurate de ejecutar `npm run dev` para recompilar tu código CSS/JS. Una vez que has recompilado tus código CSS/JS, puedes colocar los componentes dentro de una de las plantillas de tu aplicación para empezar a crear tokens de acceso clientes y personal:

```
<passport-clients></passport-clients>  
<passport-authorized-clients></passport-authorized-clients>  
<passport-personal-access-tokens></passport-personal-access-tokens>
```

Despliegue de passport

Al momento de usar Passport en tus servidores de producción por primera vez, es probable que debas ejecutar el comando `passport:keys` . Este comando genera las claves de encriptación que Passport necesita con el propósito de generar el token de acceso. Las claves generadas normalmente no son guardadas en control de código fuente:

```
php artisan passport:keys
```

php

De ser necesario, puedes definir la ruta en la que se deben cargar las claves de Passport. Para lograr esto puedes usar el método `Passport::loadKeysFrom` :

```
/**  
 * Register any authentication / authorization services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    $this->registerPolicies();  
  
    Passport::routes();  
  
    Passport::loadKeysFrom('/secret-keys/oauth');  
}
```

php

Configuración

Duración de tokens

De forma predeterminada, Passport emite tokens de acceso de larga duración que caducan después de un año. Si prefieres configurar una duración de token más larga o más corta, puedes usar los métodos `tokensExpireIn` , `refreshTokensExpireIn` y `personalAccessTokensExpireIn` . Estos métodos deberían ser ejecutados desde el método `boot` de tu `AuthServiceProvider` :

```
/**  
 * Register any authentication / authorization services.  
 *  
 * @return void  
 */  
public function boot()  
{  
    $this->registerPolicies();  
  
    Passport::routes();
```

php

```
Passport::tokensExpireIn(now()->addDays(15));

Passport::refreshTokensExpireIn(now()->addDays(30));

Passport::personalAccessTokensExpireIn(now()->addMonths(6));
}
```

Sobrescribiendo modelos predeterminados

Eres en libre de extender los modelos usados internamente por Passport. A continuación, puedes indicarle a Passport que utilice tus modelos personalizados a través de la clase `Passport` :

```
use App\Models\Passport\AuthCode;
use App\Models\Passport\Client;
use App\Models\Passport\PersonalAccessToken;
use App\Models\Passport\Token;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    Passport::useTokenModel(Token::class);
    Passport::useClientModel(Client::class);
    Passport::useAuthCodeModel(AuthCode::class);
    Passport::usePersonalAccessTokenModel(PersonalAccessToken::class);
}
```

Emitiendo tokens de acceso

Usar OAuth2 con códigos de autorización es la forma en que la mayoría de los desarrolladores están familiarizados con OAuth2. Al usar códigos de autorización, una aplicación cliente redireccionará un usuario a tu servidor donde aprobará o denegará la solicitud para emitir un token de acceso al cliente.

Administrando clientes

En primer lugar, los desarrolladores que crean aplicaciones que necesitan interactuar con la API de tu aplicación necesitarán registrar su aplicación con la tuya creando un "cliente". Normalmente, esto consiste en proporcionar el nombre de su aplicación y una dirección URL a la que tu aplicación puede redirigir después de que los usuarios aprueben su solicitud de autorización.

El comando `passport:client`

La forma más simple de crear un cliente es usando el comando Artisan `passport:client`. Este comando puede ser usado para crear tus propios clientes para probar tu funcionalidad OAuth2. Cuando ejecutes el comando `client`, Passport te pedirá más información sobre tu cliente y te proporcionará un ID y clave secreta de cliente:

```
php artisan passport:client
```

php

Redirigir URLs

Si deseas incluir en la lista blanca varias direcciones URL de redireccionamiento para tu cliente, puedes especificarlas mediante una lista delimitadas por comas cuando se le solicite la dirección URL mediante el comando `passport:client`:

```
http://example.com/callback,http://examplefoo.com/callback
```

php

Nota

Cualquier URL que contenga comas debe estar codificada.

API JSON

Debido a que tus usuarios no podrán utilizar el comando `client`, Passport proporciona una API JSON que puedes usar para crear clientes. Esto te ahorra la molestia de tener que codificar manualmente los controladores para crear, actualizar y eliminar clientes.

Sin embargo, necesitarás acoplar la API JSON de Passport con tu propio frontend para proporcionar un dashboard para que tus usuarios administren sus clientes. A continuación, revisaremos todos los

endpoints de API para administrar clientes. Por conveniencia, usaremos [Axios](#) para demostrar la realización de solicitudes HTTP a los endpoints.

La API JSON está protegida por los middleware `web` y `auth`; por lo tanto, sólo puede ser llamada desde tu propia aplicación. No se puede llamar desde una fuente externa.

TIP

Si no quieres implementar tu mismo el frontend completo para administra

Scout para Laravel

- [Introducción](#)
- [Instalación](#)
 - [Colas](#)
 - [Requisitos previos del driver](#)
- [Configuración](#)
 - [Configurando indices de modelo](#)
 - [Configurando datos de búsqueda](#)
 - [Configurando el ID de modelo](#)
- [Indexando](#)
 - [Importación en lote \(batch\)](#)
 - [Agregando registros](#)
 - [Actualizando registros](#)
 - [Eliminando registros](#)
 - [Pausando indexamiento](#)
 - [Instancias de modelos searchable condicionales](#)
- [Búsqueda](#)
 - [Cláusulas where](#)

- Paginación
- Eliminación lógica
- Personalizando motores de búsqueda
- Motores personalizados
- Macros de constructor (builder)

Introducción

Laravel Scout proporciona una sencilla solución para agregar búsquedas de texto completo a tus modelos Eloquent. Usando observadores de modelo, Scout mantendrá automáticamente tus índices de búsqueda sincronizados con tus registros de Eloquent.

Actualmente, Scout viene con el controlador (driver) [Algolia](#); sin embargo, la escritura de controladores personalizados es simple y eres libre de extender Scout con tus propias implementaciones de búsqueda.

Instalación

Primero, instala Scout por medio del paquete administrador de Composer:

```
composer require laravel/scout
```

php

Después de instalar Scout, debes publicar la configuración de Scout usando el comando Artisan `vendor:publish`. Este comando publicará el archivo de configuración `scout.php` en tu directorio `config`:

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

php

Finalmente, agrega el trait `Laravel\Scout\Searchable` al modelo en el que te gustaría hacer búsquedas. Este trait registrará un observador de modelo para mantener sincronizado con tu controlador de búsqueda:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;
```

php

```
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;
}
```

Colas

Aunque no es estrictamente necesario para usar Scout, deberías considerar fuertemente configurar un controlador de cola antes de usar el paquete. La ejecución de un trabajador (worker) de cola permitirá a Scout poner en cola todas las operaciones que sincronizan la información de tu modelo con tus índices de búsqueda, proporcionando mejores tiempos de respuesta para la interfaz web de tu aplicación.

Una vez que hayas configurado tu controlador de cola, establece el valor de la opción `queue` en tu archivo de configuración `config/scout.php` a `true`:

```
'queue' => true,
```

php

Requisitos previos del driver

Algolia

Al usar el controlador Algolia, debes configurar tus credenciales `id` y `secret` en tu archivo de configuración `config/scout.php`. Una vez que tus credenciales han sido configuradas, también necesitarás instalar Algolia PHP SDK por medio del gestor de paquetes Composer:

```
composer require algolia/algoliasearch-client-php:^2.2
```

php

Configuración

Configurando índices de modelo

Cada modelo Eloquent es sincronizado con un ”índice” de búsqueda dado, el cual contiene todos los registros que pueden ser encontrados para ese modelo. En otras palabras, puedes pensar en cada índice como una tabla MySQL. De forma predeterminada, cada modelo será persistido en un índice que

coincida con el típico nombre de la "tabla" del modelo. Típicamente, esta es la forma plural del nombre del modelo; sin embargo, eres libre de personalizar el índice del modelo sobrescribiendo el método `searchableAs` en el modelo:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
use Laravel\Scout\Searchable;  
  
class Post extends Model  
{  
    use Searchable;  
  
    /**  
     * Get the index name for the model.  
     *  
     * @return string  
     */  
    public function searchableAs()  
    {  
        return 'posts_index';  
    }  
}
```

Configuración de datos de búsqueda

De forma predeterminada, la forma `toArray` completa de un modelo dado será persistida en su índice de búsqueda. Si prefieres personalizar los datos que son sincronizados en el índice de búsqueda, puedes sobrescribir el método `toSearchableArray` en el modelo:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
use Laravel\Scout\Searchable;  
  
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;  
    }  
}
```

Configurando el ID del modelo

Por defecto, Scout usará la clave primaria del modelo como su ID única, almacenada en el índice de búsqueda. Si necesitas personalizar este comportamiento, se puede sobrescribir el método

`getScoutKey` en el modelo:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
use Laravel\Scout\Searchable;  
  
class User extends Model  
{  
    use Searchable;  
  
    /**  
     * Get the value used to index the model.  
     *  
     * @return mixed  
     */  
    public function getScoutKey()  
    {
```

php

```
        return $this->email;
    }
}
```

Indexando

Importación en lote (batch)

Si estás instalando Scout en un proyecto existente, puede que ya tengas registros de base de datos que necesites importar dentro de tu manejador de búsqueda. Scout proporciona un comando Artisan `import` que puedes usar para importar todos tus registros existentes a tus índices de búsqueda:

```
php artisan scout:import "App\Post"
```

php

El comando `flush` puede ser usado para eliminar todos los registros de un modelo de los índices de búsqueda:

```
php artisan scout:flush "App\Post"
```

php

Agregando registros

Una vez que has agregado el trait `Laravel\Scout\Searchable` a tu modelo, todo lo que necesitas hacer es llamar a `save` en una instancia de modelo y será agregada automáticamente a tu índice de búsqueda. Si has configurado Scout para `usar colas` esta operación será ejecutada en segundo plano por tu worker de cola:

```
$order = new App\Order;

// ...

$order->save();
```

php

Agregando por medio de consulta

Si prefieres agregar una colección de modelos a tu índice de búsqueda por medio de una consulta Eloquent, puedes encadenar el método `searchable` con una consulta Eloquent. El método `searchable` dividirá (chunk) los resultados de la consulta y agregará los registros a tu índice de búsqueda. Otra vez, si has configurado Scout para usar colas, todos estas porciones serán agregadas en segundo plano por tus workers de cola:

```
// Agregando Por Medio de consulta Eloquent...
App\Order::where('price', '>', 100)->searchable();

// Puedes también agregar registros a través de relaciones...
$user->orders()->searchable();

// Puedes también agregar registros a través de colecciones...
$orders->searchable();
```

php

El método `searchable` puede ser considerado una operación "upsert". En otras palabras, si el registro del modelo ya está en tu índice, será actualizado. Si no existe en el índice de búsqueda, será agregado al índice.

Actualizando registros

Para actualizar un modelo searchable, sólo necesitas actualizar las propiedades de la instancia del modelo y llamar a `save` en el modelo en tu base de datos. Scout persistirá automáticamente los cambios en tu índice de búsqueda:

```
$order = App\Order::find(1);

// Update the order...

$order->save();
```

php

También puedes usar el método `searchable` en una consulta Eloquent para actualizar una colección de modelos. Si los modelos no existen en tu índice de búsqueda, serán creados:

```
// Actualizando a través de consulta de Eloquent...
App\Order::where('price', '>', 100)->searchable();

// Puedes actualizar por medio de relaciones...
```

php

```
$user->orders()->searchable();  
  
// También puedes actualizar a través de colecciones...  
$orders->searchable();
```

Eliminando registros

Para eliminar un registro de tu índice, llama a `delete` en el modelo de la base de datos. Esta forma de eliminar es también compatible con los modelos eliminados lógicamente:

```
$order = App\Order::find(1);  
  
$order->delete();
```

Si no quieres obtener el modelo antes de eliminar el registro, puedes usar el método `unsearchable` en una instancia de consulta de Eloquent o una colección:

```
// 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();
```

Pausando el indexamiento

Algunas veces puedes necesitar ejecutar un lote de operaciones de Eloquent en un modelo sin sincronizar los datos del modelo con tu índice de búsqueda. Puedes hacer esto usando el método `withoutSyncingToSearch`. Este método acepta una sola función de retorno la cual será ejecutada inmediatamente. Cualquiera de las operaciones de modelo que ocurran dentro de la función de retorno no serán sincronizadas con el índice del modelo:

```
App\Order::withoutSyncingToSearch(function () {  
    // Perform model actions...  
});
```

Instancias de modelos searchable condicionales

A veces es posible que solo tengas que hacer que un modelo searchable bajo ciertas condiciones. Por ejemplo, imagina que tienes el modelo `App\Post` que puede estar en uno de dos estados: "borrador (draft)" y "publicado (published)". Es posible que solo desees permitir que las publicaciones "publicadas" puedan buscarse. Para lograr esto, puede definir un método `shouldBeSearchable` en su modelo:

```
public function shouldBeSearchable()
{
    return $this->isPublished();
}
```

php

El método `shouldBeSearchable` solo se aplica cuando se manipulan modelos a través del método `save`, las consultas o las relaciones. Puedes hacer que los modelos o las colecciones se puedan buscar directamente utilizando el método `searchable` que sobrescribirá el resultado del método `shouldBeSearchable`:

```
// Respetará "shouldBeSearchable"...
App\Order::where('price', '>', 100)->searchable();

$user->orders()->searchable();

$order->save();

// Sobrescribirá "shouldBeSearchable"...
$orders->searchable();

$order->searchable();
```

php

Búsqueda

Puedes empezar a buscar un modelo usando el método `search`. Este método acepta una sola cadena que será usada para buscar tus modelos. Luego debes encadenar el método `get` con la consulta de búsqueda para obtener los modelos Eloquent que coincidan con la consulta de búsqueda dada:

```
$orders = App\Order::search('Star Trek')->get();
```

php

Ya que las búsquedas de Scout devuelven una colección de modelos, incluso puedes devolver los resultados directamente desde una ruta o controlador y serán convertidos automáticamente a JSON:

```
use Illuminate\Http\Request;                                         php

Route::get('/search', function (Request $request) {
    return App\Order::search($request->search)->get();
});
```

Si prefieres obtener los resultados crudos (raw) antes de que sean convertidos a modelos de Eloquent, deberías usar el método `raw`:

```
$orders = App\Order::search('Star Trek')->raw();                         php
```

Las consultas de búsqueda son ejecutadas típicamente en el índice especificado por el método `searchableAs` del modelo. Sin embargo, puedes usar el método `within` para especificar un índice personalizado que debería ser buscado en su lugar:

```
$orders = App\Order::search('Star Trek')
    ->within('tv_shows_popularity_desc')
    ->get();
```

Cláusulas where

Scout permite que agregues cláusulas "where" sencillas a tus consultas de búsqueda. Actualmente, estas cláusulas solamente soportan verificaciones básicas de igualdad numérica y son útiles principalmente para establecer el alcance de las consultas de búsqueda por un ID. Ya que un índice de búsqueda no es una base de datos relacional, cláusulas "where" más avanzadas no están soportadas actualmente:

```
$orders = App\Order::search('Star Trek')->where('user_id', 1)->get();          php
```

Paginación

Además de obtener una colección de modelos, puedes paginar los resultados de tu búsqueda usando el método `paginate`. Este método devolverá una instancia `Paginator` justo como si hubieras paginada una consulta Eloquent tradicional:

```
$orders = App\Order::search('Star Trek')->paginate();
```

php

Puedes especificar cuántos modelos obtener por página al pasar la cantidad como primer argumento del método `paginate`:

```
$orders = App\Order::search('Star Trek')->paginate(15);
```

php

Una vez que has obtenido los resultados, puedes mostrar los resultados y renderizar los enlaces de página usando Blade justo como si hubieras paginado una consulta Eloquent tradicional:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

php

Eliminación lógica

Si tus modelos indexados son de eliminación lógica y necesitas buscar tus modelos eliminados lógicamente, establece la opción `soft_delete` del archivo `config/scout.php` en `true`:

```
'soft_delete' => true,
```

php

Cuando esta opción de configuración es `true`, Scout no removerá del índice los modelos eliminados lógicamente. En su lugar, establecerá un atributo escondido `_soft_deleted` en el registro indexado. Luego, puedes usar los métodos `withTrashed` o `onlyTrashed` para recuperar los registros eliminados lógicamente al realizar una búsqueda:

```
// Include trashed records when retrieving results...
$orders = App\Order::search('Star Trek')->withTrashed()->get();

// Only include trashed records when retrieving results...
$orders = App\Order::search('Star Trek')->onlyTrashed()->get();
```

php

TIP

Cuando un modelo eliminado lógicamente es eliminado permanentemente utilizando `forceDelete`, Scout lo removerá del índice de búsqueda automáticamente.

Personalizando motores de búsqueda

Si necesitas personalizar el comportamiento de un motor de búsqueda, puedes pasar una función de retorno (callback) como el segundo argumento al método `search`. Por ejemplo, podrías usar este callback para añadir datos de geolocalización a tus opciones de búsqueda antes de que la consulta de búsqueda sea pasada a Algolia:

```
use Algolia\AlgoliaSearch\SearchIndex;

App\Order::search('Star Trek', function (SearchIndex $algolia, string $query, array $options) {
    $options['body']['query']['bool']['filter']['geo_distance'] = [
        'distance' => '1000km',
        'location' => ['lat' => 36, 'lon' => 111],
    ];

    return $algolia->search($query, $options);
})->get();
```

php

Motores personalizados

Escribiendo el motor

Si ninguno de los motores de búsqueda integrados en Scout no se ajustan a tus necesidades, puedes escribir tu propio motor personalizado y registrarlo con Scout. Tu motor debería extender la clase

abstracta `Laravel\Scout\Engines\Engine`. Esta clase abstracta contiene ocho métodos que tu motor de búsqueda personalizado debe implementar:

```
use Laravel\Scout\Builder;                                         php

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function mapIds($results);
abstract public function map(Builder $builder, $results, $model);
abstract public function getTotalCount($results);
abstract public function flush($model);
```

Puedes encontrar útil revisar las implementaciones de estos métodos en la clase

`Laravel\Scout\Engines\AlgoliaEngine`. Esta clase te proporcionará un buen punto de inicio para aprender cómo implementar cada uno de estos métodos en tu propio motor.

Registrando el motor

Una vez que hayas escrito tu motor personalizado, puedes registrararlo con Scout usando el método

`extend` del administrador de motor de Scout. Deberías ejecutar el método `extend` desde el método `boot` de tu `AppServiceProvider` o cualquier otro proveedor de servicio usado por tu aplicación.

Por ejemplo, si has escrito un `MySqlSearchEngine`, puedes registrararlo como sigue:

```
use Laravel\Scout\EngineManager;                                         php

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

Una vez que tu motor ha sido registrado, puedes especificarlo como tu `driver` predeterminado de Scout en tu archivo de configuración `config/scout.php` :

```
'driver' => 'mysql',
```

php

Macros de constructor (builder)

Si deseas definir un método constructor personalizado, puedes usar el método `macro` en la clase `Laravel\Scout\Builder`. Típicamente, las "macros" deben ser definidas dentro de un método `boot` de un proveedor de servicios:

```
<?php
```

php

```
namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;
use Laravel\Scout\Builder;

class ScoutMacroServiceProvider extends ServiceProvider
{
    /**
     * Register the application's scout macros.
     *
     * @return void
     */
    public function boot()
    {
        Builder::macro('count', function () {
            return $this->engine->getTotalCount(
                $this->engine()->search($this)
            );
        });
    }
}
```

La función `macro` acepta un nombre como su primer argumento y un Closure como el segundo. El Closure del macro será ejecutado al momento de llamar el nombre del macro desde una implementación `Laravel\Scout\Builder` :

```
App\Order::search('Star Trek')->count();
```

php

Laravel Socialite

- [Introducción](#)
- [Actualizando Socialite](#)
- [Instalación](#)
- [Configuración](#)
- [Enrutamiento](#)
- [Parámetros opcionales](#)
- [Alcances de acceso](#)
- [Autenticación sin estado](#)
- [Obteniendo detalles de usuario](#)

Introducción

Además de la típica, autenticación basada en formularios, Laravel también proporciona una sencilla y conveniente forma de autenticar con proveedores OAuth usando [Laravel Socialite](#). Actualmente Socialite soporta autenticación con Facebook, Twitter, LinkedIn, Google, Github, GitLab y Bitbucket.

TIP

Los adaptadores para otras plataformas son listados en el sitio web de [Proveedores de Socialite](#) manejado por la comunidad.

Actualizando Socialite

Al actualizar a una nueva versión principal de Socialite, es importante que revise cuidadosamente [la guía de actualización](#).

Instalación

Para empezar con Socialite, usa Composer para agregar el paquete a las dependencias de tu proyecto:

```
composer require laravel/socialite
```

php

Configuración

Antes de usar Socialite, también necesitarás agregar las credenciales para los servicios OAuth que tu aplicación utiliza. Estas credenciales deberían estar colocadas en tu archivo de configuración

`config/services.php`, y debería usar la clave `facebook`, `twitter`, `linkedin`, `google`, `github`, `gitlab` o `bitbucket` dependiendo del proveedor que tu aplicación requiera. 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',
],
```

php

TIP

Si la opción `redirect` contiene una ruta relativa, será resuelta automáticamente a una URL completamente calificada.

Enrutamiento

A continuación, iestás listo para autenticar usuarios! Necesitarás dos rutas: una para redireccionar el usuario al proveedor OAuth y otra para recibir la función de retorno del proveedor después de la autenticación. Accederemos a Socialite usando la clase facade `Socialite`:

```
<?php
```

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 `redirect` se toma la tarea de enviar el usuario al proveedor OAuth, mientras que el método `user` leerá la solicitud entrante y obtendrá la información del usuario desde el proveedor.

Necesitarás definir las rutas para tus métodos de controlador:

```
Route::get('login/github', 'Auth\LoginController@redirectToProvider');
Route::get('login/github/callback', 'Auth\LoginController@handleProviderCallback')
```

php

Parámetros opcionales

Un número de proveedores OAuth soportan parámetros opcionales en la solicitud de redirección. Para incluir algunos de los parámetros opcionales en la solicitud, llama el método `with` con un arreglo asociativo:

```
return Socialite::driver('google')
    ->with(['hd' => 'example.com'])
    ->redirect();
```

php

Nota

Al momento de usar el método `with`, procura no pasar algunas palabras reservadas tales como `state` or `response_type`.

Alcances de acceso

Antes de redireccionar al usuario, también puedes agregar "alcances (scopes)" adicionales en la solicitud usando el método `scopes`. Este método mezclará todos los alcances existentes con los que suministras:

```
return Socialite::driver('github')
    ->scopes(['read:user', 'public_repo'])
    ->redirect();
```

php

Puedes sobrescribir todos los alcances existentes usando el método `setScopes`:

```
return Socialite::driver('github')
    ->setScopes(['read:user', 'public_repo'])
    ->redirect();
```

php

Autenticación sin estado

El método `stateless` puede ser usado para deshabilitar la verificación de estado de sesión. Esto es útil al momento de agregar la autenticación de una red social a una API.

```
return Socialite::driver('google')->stateless()->user();
```

php

Obteniendo detalles de usuario

Una vez que tengas una instancia de usuario, puedes aprovechar de obtener algunos detalles del 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();
```

php

Obteniendo Los detalles de usuario desde un token (OAuth2)

Si ya tienes un token de acceso válido de un usuario, puedes obtener sus detalles usando el método

`userFromToken` :

```
$user = Socialite::driver('github')->userFromToken($token);
```

php

Obteniendo los detalles de usuario desde un token y secreto (OAuth1)

Si ya tienes un par válido de token / secreto de un usuario, puedes obtener sus detalles usando el método `userFromTokenAndSecret` :

```
$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $secret);
```

php

Laravel Telescope

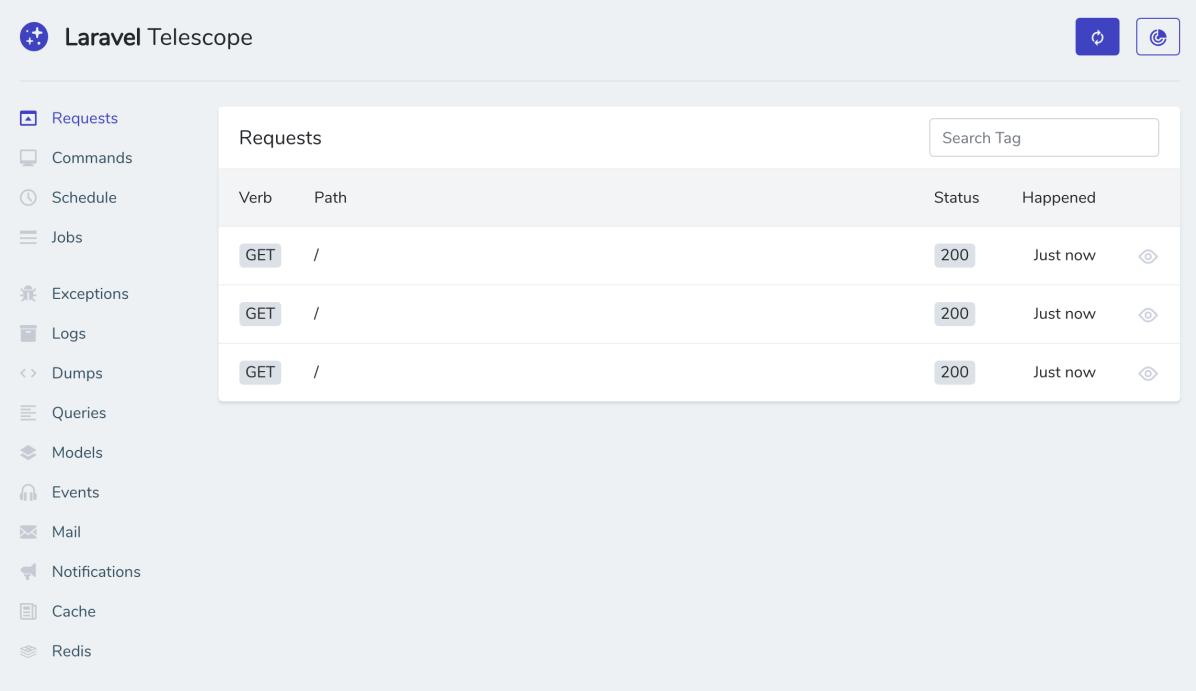
- Introducción
- Instalación
 - Configuración
 - Remover datos de entradas de Telescope
 - Personalizar la migración
- Autorización para el panel de control
- Filtros
 - Entradas
 - Lotes
- Etiquetado
 - Agregar etiquetas personalizadas
- Observadores disponibles
 - Observador De caché
 - Observador De comandos
 - Observador De variables
 - Observador De eventos
 - Observador De excepciones
 - Observador De gates
 - Observador De trabajos
 - Observador De registros (log)
 - Observador De correos
 - Observador De modelos
 - Observador De notificaciones

- Observador De consultas De Bases De Datos
- Observador De Redis
- Observador De solicitudes (request)
- Observador De tareas programadas

Introducción

Telescope de Laravel es un elegante asistente para depurar código para el framework de Laravel.

Telescope proporciona información detallada de las solicitudes entrantes de tu aplicación, excepciones, entradas de log, consultas de bases de datos, trabajos en cola, correos, notificaciones, operaciones de caché, tareas programadas, valores de variables y mucho más. Telescope acompaña maravillosamente tu entorno de desarrollo de Laravel.



The screenshot shows the Laravel Telescope dashboard. On the left is a sidebar with icons and labels for: Requests, Commands, Schedule, Jobs, Exceptions, Logs, Dumps, Queries, Models, Events, Mail, Notifications, Cache, and Redis. The main area is titled "Requests" and contains a table with columns: Verb, Path, Status, and Happened. There are three entries in the table:

Verb	Path	Status	Happened
GET	/	200	Just now
GET	/	200	Just now
GET	/	200	Just now

At the top right of the main area are two small blue buttons with white icons. At the bottom right of the main area is a search bar labeled "Search Tag".

Instalación

Puedes usar Composer para instalar Telescope dentro de tu proyecto de Laravel:

```
composer require laravel/telescope
```

php

Después de instalar Telescope, publica sus recursos usando el comando Artisan `telescope:install`. Después de instalar Telescope, también deberías ejecutar el comando `migrate`:

```
php artisan telescope:install
```

php

```
php artisan migrate
```

Actualizando Telescope

Si haces una actualización de Telescope, deberías volver a publicar los recursos de Telescope:

```
php artisan telescope:publish
```

php

Instalando únicamente en entornos específicos

Si planeas usar Telescope solamente para apoyar tu desarrollo local, puedes instalar Telescope usando la bandera `--dev` :

```
composer require laravel/telescope --dev
```

php

Después de ejecutar `telescope:install`, deberías remover el registro de proveedor de servicio `TelescopeServiceProvider` de tu archivo de configuración `app`. En su lugar, registra manualmente el proveedor de servicio en el método `register` de tu `AppServiceProvider`:

```
use App\Providers\TelescopeServiceProvider;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    if ($this->app->isLocal()) {
        $this->app->register(TelescopeServiceProvider::class);
    }
}
```

php

Personalización de la migración

Si no vas a usar las migraciones predeterminadas de Telescope, deberías ejecutar el método `Telescope::ignoreMigrations` en el método `register` de tu `AppServiceProvider`. Puedes exportar las migraciones predeterminadas usando el comando `php artisan vendor:publish --tag=telescope-migrations`.

Configuración

Después de publicar los recursos de Telescope, su archivo de configuración principal estará ubicado en `config/telescope.php`. Este archivo de configuración permite que configures tus opciones de observador (watcher) y cada opción de configuración incluye una descripción de su propósito, así que asegúrate de examinar meticulosamente este archivo.

Si lo deseas, puedes deshabilitar completamente la colección de datos de Telescope usando la opción de configuración `enabled`:

```
'enabled' => env('TELESCOPE_ENABLED', true),
```

php

Remover datos de entradas de Telescope

Sin la remoción, la tabla `telescope_entries` puede acumular registros muy rápidamente. Para mitigar esto, deberías programar el comando `telescope:prune` para que se ejecute diariamente:

```
$schedule->command('telescope:prune')->daily();
```

php

De forma predeterminada, aquellas entradas con más de 24 horas serán removidas. Puedes usar la opción `hours` al momento de ejecutar el comando para indicar cuánto tiempo retiene los datos Telescope. Por ejemplo, el siguiente comando eliminará todos los registros con más de 48 horas desde que fueron creados.

```
$schedule->command('telescope:prune --hours=48')->daily();
```

php

Autorización para el panel de control

Telescope viene con un panel de control en `/telescope`. De forma predeterminada, solamente serás capaz de acceder este panel de control en el entorno `local`. Dentro de tu archivo

`app/Providers/TelescopeServiceProvider.php`, hay un método `gate`. Esta gate de autorización controla el acceso a Telescope en los entornos **que no son locales**. Eres libre de modificar este gate de acuerdo a tus necesidades para restringir el acceso a tu instalación de Telescope:

```
/*
 * Register the Telescope gate.
 *
 * This gate determines who can access Telescope in non-local environments.
 *
 * @return void
 */
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            'taylor@laravel.com',
        ]);
    });
}
```

php

Filtros

Entradas

Puedes filtrar los datos que son guardados por Telescope por medio de la función de retorno (callback)

`filter` que está registrada en tu `TelescopeServiceProvider`. De forma predeterminada, esta función de retorno guarda todos los datos en el entorno `local` y las excepciones, trabajos que fallan, tareas programadas, y datos de las etiquetas monitoreadas en los demás entornos.

```
/*
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::filter(function (IncomingEntry $entry) {
```

php

```
        if ($this->app->isLocal()) {
            return true;
        }

        return $entry->isReportableException() ||
            $entry->isFailedJob() ||
            $entry->isScheduledTask() ||
            $entry->hasMonitoredTag();
    });

}
```

Lotes

Mientras la función de retorno `filter` filtra datos por entradas individuales, puedes usar el método `filterBatch` para registrar una función de retorno que filtra todos los datos para un comando de consola o solicitud dado. Si la función de retorno devuelve `true`, la totalidad de las entradas son guardadas por Telescope:

```
use Illuminate\Support\Collection;

/**
 * Register any application services.
 *
 * @return void
 */
public function register()
{
    $this->hideSensitiveRequestDetails();

    Telescope::filterBatch(function (Collection $entries) {
        if ($this->app->isLocal()) {
            return true;
        }

        return $entries->contains(function ($entry) {
            return $entry->isReportableException() ||
                $entry->isFailedJob() ||
                $entry->isScheduledTask() ||
                $entry->hasMonitoredTag();
        });
    });
}
```

```
});  
}
```

Etiquetado

Telescope te permite buscar entradas por "etiqueta". A menudo, las etiquetas son nombres de clases de modelos de Eloquent o IDs de usuarios autenticados que Telescope automáticamente agrega a entradas. Ocasionalmente, puede que quieras adjuntar tus propias etiquetas personalizadas a entradas. Para lograr esto, puedes usar el método `Telescope::tag`. El método `tags` acepta un callback que debe retornar un arreglo de etiquetas. Las etiquetas retornadas por el callback se fusionarán con cualquier etiqueta que Telescope automáticamente agregaría a la entrada. Debes llamar al método `tags` dentro de tu `TelescopeServiceProvider`:

```
use Laravel\Telescope\Telescope;  
/**  
 * Register any application services.  
 *  
 * @return void  
 */  
public function register()  
{  
    $this->hideSensitiveRequestDetails();  
    Telescope::tag(function (IncomingEntry $entry) {  
        if ($entry->type === 'request') {  
            return ['status:' . $entry->content['response_status']];  
        }  
        return [];  
    });  
}
```

php

Observadores disponibles

Los observadores de Telescope coleccionan los datos de la aplicación cuando una solicitud o comando de consola es ejecutado. Puedes personalizar la lista de observadores que deseas habilitar dentro de tu archivo de configuración `config/telescope.php`:

```
'watchers' => [
    Watchers\CacheWatcher::class => true,
    Watchers\CommandWatcher::class => true,
    ...
],
]

```

php

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 100,
    ],
    ...
],
]

```

php

Observador de caché

El observador de caché (Cache Watcher) guarda datos cuando una clave está presente, falta, es actualizada u olvidada en caché.

Observador de comandos

El observador de comandos (command watcher) guarda los argumentos, opciones, códigos de salida, información enviada a la pantalla cada vez que se ejecuta un comando Artisan. Si deseas excluir ciertos comandos para que no sean grabados por el observador, puedes especificar el comando junto con la opción `ignore` en tu archivo `config/telescope.php` :

```
'watchers' => [
    Watchers\CommandWatcher::class => [
        'enabled' => env('TELESCOPE_COMMAND_WATCHER', true),
        'ignore' => ['key:generate'],
    ],
    ...
],
]

```

php

Observador de variables

El observador de variables (dump watcher) guarda y muestra los valores de tus variables en Telescope. Al momento de usar Laravel, los valores de las variables pueden ser mostrados usando la función global `dump`. La pestaña del observador de variables debe estar abierta en un navegador para que los valores sean guardados, de lo contrario serán ignorados por el observador.

Observador de eventos

El observador de eventos (event watcher) guarda la carga, oyentes (listeners) y los datos de difusión (broadcast) para cualquier evento que sea despachado por tu aplicación. Los eventos internos del framework de Laravel son ignorados por el observador de eventos.

Observador de excepciones

El observador de excepciones (exception watcher) guarda los datos y el seguimiento de la pila para cualquier excepción reportable que sea lanzada por tu aplicación.

Observador de gates

El observador de gate (gate watcher) guarda los datos y el resultado de verificaciones de gates y políticas hechas por tu aplicación. Si deseas excluir ciertas habilidades para que no sean guardadas por el observador, puedes especificar aquellas en la opción `ignore_abilities` en tu archivo `config/telescope.php`:

```
'watchers' => [
    Watchers\GateWatcher::class => [
        'enabled' => env('TELESCOPE_GATE_WATCHER', true),
        'ignore_abilities' => ['viewNova'],
    ],
    ...
],
```

php

Observador de trabajos

El observador de trabajos (job watcher) guarda los datos y estado de los trabajos despachado por tu aplicación.

Observador de registros

El observador de registros (log watcher) guarda datos de los registros escritos por tu aplicación.

Observador de correos

El observador de correos (mail watcher) permite que veas una pre-visualización en el navegador de los correos junto con sus datos adjuntados. También puedes descargar los correos como un archivo

.eml .

Observador de modelos

El observador de modelos (model watcher) guarda los cambios del modelo cada vez que se despacha un evento `created` , `updated` , `restored` , o `deleted` de Eloquent. Puedes especificar cuáles eventos de modelos deberían ser guardados por medio de la opción `events` del observador:

```
'watchers' => [
    Watchers\ModelWatcher::class => [
        'enabled' => env('TELESCOPE_MODEL_WATCHER', true),
        'events' => ['eloquent.created*', 'eloquent.updated*'],
    ],
    ...
],
```

php

Observador de notificaciones

El observador de notificaciones (notification watcher) guarda todas las notificaciones enviadas por tu aplicación. Si la notificación dispara un correo y tienes el observador de correos habilitado, el correo también estará disponible para pre-visualizar en la pantalla del observador de correos.

Observador de consultas de bases de datos

El observador de consultas de bases de datos (query watcher) guarda los comandos SQL, enlaces, y tiempo de ejecución para todas las consultas de bases de datos que sean ejecutadas por tu aplicación. El observador también coloca una etiqueta `slow` a las consultas más lentas, aquellas que tardan más de 100 micro segundos. Puedes personalizar el umbral para las consultas lentas usando la opción `slow` del observador:

```
'watchers' => [
    Watchers\QueryWatcher::class => [
        'enabled' => env('TELESCOPE_QUERY_WATCHER', true),
        'slow' => 50,
    ],
    ...
],
],
```

Observador de Redis

Nota

Los eventos de Redis deben ser habilitados por el observador de Redis (Redis watcher) para que funcione de forma correcta. Puedes habilitar los eventos de Redis ejecutando

```
Redis::enableEvents() en el método boot de tu archivo  
app/Providers/AppServiceProvider.php .
```

El observador de Redis (redis watcher) guarda todos los comandos de Redis ejecutados por tu aplicación. Si estás usando Redis para el almacenamiento de caché, también los comandos de caché serán guardados por el observador de Redis.

Observador de solicitudes

El observador de solicitudes (request watcher) guarda la solicitud, encabezados, la sesión y los datos de respuesta asociados con las solicitudes manejadas por la aplicación. Puedes limitar tus datos de respuesta por medio de la opción `size_limit` (en KB):

```
'watchers' => [
    Watchers\RequestWatcher::class => [
        'enabled' => env('TELESCOPE_REQUEST_WATCHER', true),
        'size_limit' => env('TELESCOPE_RESPONSE_SIZE_LIMIT', 64),
    ],
    ...
],
```

Observador de tareas programadas

El observador de tareas programadas (schedule watcher) guarda el comando y la información enviada a la pantalla de las tareas programadas que ejecuta tu aplicación.