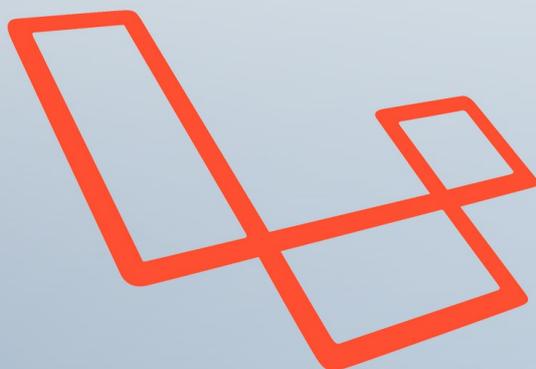


# Manual de Laravel 5



**Carlos Ruiz Ruso**  
**Miguel Angel Alvarez**

 desarrolloweb.com

[desarrolloweb.com/manuales/manual-laravel-5.html](http://desarrolloweb.com/manuales/manual-laravel-5.html)

# Introducción: Manual de Laravel 5

Manual del framework PHP Laravel, centrándonos en versión Laravel 5 (concretamente Laravel 5.1), que nos trae diversas mejoras en rendimiento y cambios en la organización de los archivos y proyectos. Es un manual que explica paso por paso los elementos más importantes que forman parte de este popular framework, a la vez que nos ofrece una guía para comenzar a crear aplicaciones web basadas en él.

Comenzamos con la instalación, usando una máquina virtual llamada Homestead, que es la plataforma oficial de desarrollo de Laravel 5. Luego usamos Composer para bajarnos e instalar el framework y a partir de ahí ya nos dedicamos a analizar el framework en detalle.

En este manual usamos la versión de Laravel 5.1 que además es la primera versión del framework ofrecida como LTS (Long Term Support), lo que te asegura actualizaciones de seguridad para los próximos años.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-laravel-5.html>

## Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

---

### Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



### Carlos Ruiz Ruso

Consultor tecnológico para el desarrollo de proyectos online especializado en WordPress y el framework Laravel.



# Instalación y configuración de Laravel 5

En esta primera parte del manual te enseñamos a instalar Laravel 5 y a configurar tu entorno de trabajo. Abordamos con detalle la plataforma oficial para desarrollo de Laravel, basada en una máquina virtual Vagrant, con el nombre de Homestead. Esta alternativa te ofrece las condiciones para desarrollar más parecidas a un entorno de producción y es la opción recomendada para tu entorno de desarrollo, que te ayudará a familiarizarte con la administración de servidores y evitará problemas cuando publiques tu web en el servidor definitivo.

## Homestead de Laravel

**Procedimiento de instalación del entorno de desarrollo ideal para la creación de aplicaciones usando el framework PHP Laravel y la virtualización de una máquina con Vagrant y Homestead.**

En este artículo vamos a ofrecer las pautas para contar con el mejor entorno de desarrollo posible para Laravel 5, el más profesional, fácil de instalar y de mayor versatilidad, ayudándonos de la virtualización de la máquina de desarrollo y Vagrant.

La fórmula tiene el nombre de Homestead, una herramienta oficial creada por el equipo de Laravel para facilitar la vida a los desarrolladores que quieran usar este framework PHP para comenzar un proyecto rápidamente.

La idea es que los desarrolladores tengamos que implementar el menor número de configuraciones en nuestras máquinas y que con dos o tres comandos podamos conseguir todo lo que necesitaría el entorno de desarrollo más exigente.

**Nota:** Con este texto damos inicio al [Manual de Laravel 5](#), en el que pretendemos enseñar a los lectores a trabajar con este importante framework PHP. Así que siéntate cómodo y aprovecha para comenzar con nosotros esta zambullida en profundidad a Laravel 5.1.



## Qué es exactamente Homestead

Es una "Box de Vagrant". Por si no lo sabes, [Vagrant es una capa por encima de Virtualbox](#) (o de otros sistemas de virtualización como VMWare) que nos sirve para crear entornos de desarrollo y las Box en su terminología son imágenes de sistemas operativos ya instalados. Generalmente vienen "peladas", tal como tendríamos el ordenador si hubiéramos acabado de instalar el sistema operativo en ese momento. O en algunos casos, como Laravel Homestead, ya configurados con distintos paquetes de software.

Homestead te instala software como: Ubuntu, PHP, HHVM, Nginx, MySQL, PostgreSQL, NodeJS, Redis y algunas librerías como Memcached, Laravel Envoy, Beanstalkd...

## Por qué Homestead

Porque la comunidad de creadores de Laravel quieren acercarte al framework. Facilitarte no solo la entrada como desarrollador de Laravel, sino también proporcionarte el entorno de desarrollo más adecuado para acometer cualquier tipo de proyecto.

Realmente puedes contar con Laravel funcionando en un Xampp, Mamp o similares, pero si virtualizas tendrás varias ventajas como contar con un entorno mucho más parecido al de producción, independiente de otros proyectos con los que estés desarrollando y además el mismo sistema, con los mismos paquetes de software. No tiene sentido extenderse mucho en las [ventajas de los entornos virtualizados, porque ya hemos hablado de ellos en otras ocasiones](#). Pero piensa además que puedes conseguir todo eso con la facilidad de lanzar unos pocos comandos para generar automáticamente toda la infraestructura perfectamente configurada.

## Qué necesitas

Instalar VirtualBox como sistema de virtualización, o VMware si lo prefieres (aunque en ese caso tendrías que instalar un plugin adicional). Luego instalas Vagrant. Ambos programas son gratuitos y multiplataforma.

Una vez instalados ambos programas (Primero se recomienda instalar Virtualbox y luego Vagrant) se debe descargar la caja "Box" de Homestead, para lo que lanzarás el siguiente comando de consola.

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

Ese comando lo podrás lanzar desde cualquier carpeta, desde el terminal de línea de comandos, claro está.

**Nota:** Si tienes una versión antigua de Vagrant quizás tengas que darle la URL completa de la localización de la Box.

```
vagrant box add laravel/homestead https://atlas.hashicorp.com/laravel/boxes/homestead
```

### Instalar Homestead

El tercer paso sería instalar el propio Homestead. Para ello usaremos la box que acabamos de descargar y Vagrant. Pero necesitamos un archivo de configuración que se encuentra en Github que contiene toda la

información de configuración que Vagrant necesitaría para generar y aprovisionar la máquina virtual para desarrollar con Laravel.

Para ello abrimos un terminal y nos situamos en nuestra carpeta de proyectos habitual y desde allí clonaremos el repositorio de Homestead con Git.

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

**Nota:** Ese comando clona un repositorio. El repositorio está en Github en la ruta <https://github.com/laravel/homestead.git>. El parámetro final del comando anterior git clone que está definido como "Homestead" es la carpeta donde va a colocar los archivos del repositorio que estás clonando. Si quisieras podrías cambiarle el nombre a la carpeta de destino, por ejemplo, "proyecto-prueba" en vez de "Homestead", para lo que el comando sería el siguiente:

```
git clone https://github.com/laravel/homestead.git proyecto-prueba
```

La estructura de carpetas que tengas tú en tu ordenador de momento es indiferente y responderá a tus propias preferencias. Lo que te interesa es todo lo que tiene el repositorio, que te vas a descargar con el comando "git clone". El lugar donde esté localizada la carpeta donde te descargas el repo de Homestead es lo de menos, por ejemplo muchas personas prefieren colocarla en la carpeta "Dropbox" para que cuando se realicen cambios en el proyecto se realicen copias de seguridad automáticas en la nube.

Dentro de ese repositorio clonado tenemos en un archivo bash con el nombre "init.sh" que tiene el código con los comandos para crear el archivo de configuración Homestead.yaml. Simplemente lo tienes que ejecutar, con el comando:

```
bash init.sh
```

Obviamente el comando "bash init.sh" lo debes ejecutar desde la ruta donde se encuentra el archivo init.sh, en la carpeta del repositorio que acabas de clonar. Verás que simplemente te da un mensaje de respuesta "Homestead initialized!". Eso quiere decir que ha ido bien y ha podido crear los archivos de configuración necesarios para los siguientes pasos.

Entre los archivos de configuración que se crean al ejecutar el init.sh hay uno que necesitarás editar, llamado "**Homestead.yaml**". Este archivo de configuración lo encontrarás en tu carpeta personal (la de tu usuario dentro de tu sistema operativo) y dentro de ella en la carpeta oculta ".homestead". Si lo abres podrás encontrar los datos de configuración de la máquina virtual Homestead, el servidor de desarrollo virtualizado que vamos a crear, que incluye datos como la IP del servidor, las CPUs, memoria asignada, carpetas que se compartirán entre la máquina de desarrollo y el ordenador anfitrión (nuestro ordenador físico) y cosas así.

**Nota:** Por ejemplo en Mac mi directorio del usuario personal es /Users/midesweb y la carpeta .homestead estará por tanto en /Users/midesweb/.homestead

En un principio necesitamos tocar pocas cosas del Homestead.yaml, porque la mayoría ya se ha configurado automáticamente, acorde con nuestro sistema operativo gracias al archivo init.sh, pero podrás desear cambiar los siguientes ítem:

- IP del servidor virtual, por defecto aparece ip: "192.168.10.10".
- Llave para la conexión SSH con la máquina virtual (ver información más abajo)
- Folders, que son las carpetas que vas a tener en tu sitio, mapeadas en dos direcciones, el servidor virtual y el ordenador local (luego te decimos cómo).
- Sites, para definir el nombre del sitio para el virtualhost.

## Crear la llave de SSH

Esta es la llave para la conexión con el servidor virtual por línea de comandos. Si no tienes una la puedes crear. Desde Mac o Linux es tan sencillo como lanzar el siguiente comando:

```
ssh-keygen -t rsa -C "you@homestead"
```

Desde Windows la recomendación es usar el "Bash de Git" (el programa de terminal que te viene incorporado cuando instalas Git) y lanzar el mismo comando anterior. Aunque también podrías usar el popular software PuTTY y PuTTYgen.

En el anterior comando no necesitas editar nada, lo pones tal cual. Enseguida verás que el programa que te genera la clave te pide la ruta donde va a colocar esta key generada. El propio programa te muestra una ruta predeterminada donde sugiere se coloquen las llaves. Podemos pulsar enter sin escribir nada para aceptar esa ruta o bien escribir cualquier otra ruta donde se generará la clave. Puedes poner lo que quieras, con tal que te acuerdes de la ruta que has seleccionado.

**Nota:** en cuanto a la ruta donde se van a colocar las claves es totalmente personalizable. Pon la que quieras, simplemente acuérdate donde están los archivos. Para ser más específicos y por si alguien se pierde, pueden darse tres casos:

- Que te parezca bien la ruta donde te sugiere que va a crear las claves. En ese caso puedes apretar enter y listo. En mi caso la ruta donde se colocarían las claves de manera predeterminada es /Users/midesweb/.ssh/id\_rsa. Por tanto se crearán los archivos id\_rsa e id\_rsa.pub en la carpeta /Users/midesweb/.ssh.
- Que escribas el nombre de un archivo simplemente, algo como "homesteadkey". En ese caso te creará las claves en el directorio donde estás situado cuando hiciste el comando ssh-keygen, archivos homesteadkey y homesteadkey.pub.
- Que escribas una ruta absoluta, algo como /Users/midesweb/.ssh/nuevaclave, en cuyo caso te colocará la clave en esa ruta absoluta, tanto la clave privada /Users/midesweb/.ssh/nuevaclave como la pública /Users/midesweb/.ssh/nuevaclave.pub

El programa de generar las key te pedirá también insertar una frase como clave, que puedes poner la que desees. O bien una cadena vacía si quieres ahorrarte indicar una frase que te puedas olvidar. Como es solo para trabajar en local sería suficiente.

Como decíamos, una vez generada la clave lo que te interesa es saber la ruta donde se ha colocado, que tendrás que usar para configurar el archivo Homestead.yaml.

El lugar donde vas a tener que introducir la ruta de la clave será algo como esto:

```
authorize: ~/.ssh/id_rsa.pub

keys:
  - ~/.ssh/id_rsa
```

Esa ruta es la que tienes que configurar según la localización de las llaves creadas. Generalmente si seleccionaste la carpeta predeterminada que sugería el comando ssh-keygen ni siquiera tendrías que editar nada. Si las claves las guardaste en otro lugar, entonces tendrás que indicarlo aquí. De todos modos, no tiene ninguna complicación, es saber la ruta donde está tu clave e indicarla en el Homestead.yaml.

**Nota:** Recuerda que ~/ es un alias de tu ruta personal (ver otra nota más abajo donde se habla sobre este tema). Pero si te lía lo puedes cambiar por una ruta absoluta y punto.

## Carpetas del servidor virtual y mapeo a las carpetas locales

Una de las características de los entornos con Vagrant es que hay determinadas carpetas del servidor virtual (invitado) que están mapeadas a directorios locales de tu máquina real (anfitrión). Esto te permite editar archivos de tu proyecto como los editarías en cualquier otra situación, con tus editores de código preferidos, trabajando en local como si lo hicieras con un sistema más tradicional tipo Xampp o Mamp, Wamp, etc. Las carpetas locales están enlazadas con las carpetas del servidor, por lo que no tienes que subir los archivos por FTP, SCP ni nada parecido. Cuando modifiques algún archivo, o crees archivos nuevos en tu carpeta local, automáticamente esos cambios serán producidos en las carpetas enlazadas del servidor virtualizado. Podrías configurar tantas carpetas compartidas como necesites en tu proyecto.

Ahora en el archivo Homestead.yaml tendrás que definir esa estructura de carpetas, que se adaptará a tus costumbres habituales. Como hemos comentado antes, tendrás una carpeta de tu ordenador donde guardas tus proyectos y seguirá siendo así. Por tanto, es simplemente guardar este proyecto en una carpeta en una ruta habitual para ti.

```
folders:
  - map: ~/carpeta_de_proyectos/proyecto_x/codigo
    to: /home/projects
```

En este lugar colocamos en "map" el valor de nuestra carpeta en el ordenador anfitrión y en "to" la ruta de la carpeta del ordenador virtualizado que vamos a crear con Vagrant y Homestead. La carpeta que estás indicando en "map", que corresponde con tu ordenador real, debería existir.

Ten en cuenta que en una máquina virtual de Homestead podrías [mantener varios proyectos Laravel](#), por tanto no sería descabellado que uses una declaración Folders abierta a esa posibilidad, donde mapeas una

carpeta de proyectos en el anfitrión a otra de proyectos en la máquina virtual.

folders:

```
- map: ~/carpeta_instalacion_homestead/proyectos/  
to: /home/vagrant/proyectos/
```

**Nota:** Una aclaración acerca de las rutas para los de Windows. "~/" corresponde con la carpeta raíz de tu usuario. Por ejemplo, si tu usuario (con el que entras en Windows es "micromante" esa ruta correspondería con una carpeta como "C:/Users/micromante". La carpeta "~/.homestead" correspondería físicamente con "C:/Users/micromante/.homestead". Si usas el terminal básico del Windows no reconocerá la ruta ~/, así que tendrás que usar la ruta real de tu carpeta de usuario, la física o ruta real. Si usas un terminal un poco mejor, por ejemplo el Git Bash sí que te reconocerá ese nombre lógico ~/ y te llevará a tu carpeta de usuario, sea cual sea tu nombre de usuario, igual que ocurre en el terminal de Linux o Mac.

De nuevo, no hay necesidad de colocar algo específico en la sección folders, sino que será tal como tú desees organizar las carpetas, tanto en el servidor Homestead como en tu ordenador local.

## Definir la configuración de los sitios que vamos a albergar en esta máquina Homestead

Nos queda un valor que configurar, que es el nombre del host virtual que nos va a configurar Homestead en esta máquina virtual que estamos a punto de crear. Este host virtual o "virtualhost" lo usaremos para entrar en el proyecto desde un navegador, con un nombre de dominio, como si fuera un servidor remoto. Lógicamente ese dominio solo estará disponible desde tu ordenador y tendrás que configurar el conocido archivo "hosts" para asociar la IP del servidor al nombre de dominio virtual.

sites:

```
- map: homestead.dw  
to: /home/projects/mi_proyecto/public
```

En este caso en "map" colocamos el nombre de dominio para el virtualhost y en "to" indicamos la carpeta del servidor virtual donde va a estar el "document root". Si te fijas, ese document root debe colgar de la carpeta que tienes en la máquina virtual que has enlazado con tu carpeta real, de ese modo podrás editar en local los archivos de tu proyecto que se servirá desde el ordenador virtual.

**Nota:** En una instalación de Homestead serías capaz de albergar varios proyectos si así lo deseas, con diversos host virtuales. Insistimos en que cada uno de los host virtuales tendrás que configurarlo en el archivo de hosts de tu ordenador, asociando la IP que hayas configurado para esta máquina virtual con el nombre de dominio de tus virtualhost.

Tu archivo hosts tendrás que editarlo agregando esta líneas, que quedará más o menos de esta forma:

```
192.168.10.10 homestead.dw
```

Aunque lógicamente, tendrás que poner la IP que hayas definido en el campo IP de Homestead.yaml y el nombre de dominio que hayas definido en los "sites", en el campo "map".

**Nota:** La configuración del archivo Hosts está explicada en el artículo: [Modificar el archivo de Hosts en Windows, Linux y Mac](#)

## Lanzar la máquina virtual

Una vez configurado el Homestead.yaml tienes que lanzar la máquina virtual, es decir, tendrás que ponerla en marcha, con el conocido comando de Vagrant:

```
vagrant up
```

Ese comando lo tienes que lanzar desde la carpeta donde has puesto el repositorio de Homestead, que clonaste con Git. La primera vez que se ejecute va a tardar un poco, porque tiene que instanciar y configurar toda la máquina, pero luego iniciar la máquina virtual será muy rápido, también con el comando "vagrant up".

**Nota:** En los comentarios del artículo nos han preguntado en qué momento hacer el "vagrant init". Ese comando no lo necesitas hacer tú mismo. Sirve para que se cree el Vagrantfile y configurar una carpeta para que albergue una máquina virtual que vas a crear con Vagrant. Pero básicamente el Vagrantfile es lo que te descargas al clonar el repositorio y las configuraciones que necesita Vagrant y que realizas predeterminadamente al hacer al "Vagrant init" son las que has aprendido a definir en el archivo Homestead.yaml.

Por tanto, si en algún momento haces un "vagrant up" y te pide hacer un "vagrant init" es que estás haciendo el "vagrant up" desde una carpeta incorrecta. En ese caso fíjate que estás situado en el repositorio de Homestead que has clonado. En esa carpeta debería haber un Vagrantfile, ya que es uno de los archivos del repositorio clonado.

Una vez lanzada la máquina virtual podríamos hacer una primera prueba accediendo a la IP del servidor virtual con tu navegador.

```
http://192.168.10.10/
```

(Ojo, la URL podría ser otra si tocaste en el Homestead.yaml el dato ip: "192.168.10.10". Si no es esa, será la IP que hayas configurado en tu caso)

De momento no veremos nada, pero al menos el navegador no debería recibir tampoco el típico error del navegador para informar que el host no ha sido encontrado. En mi caso aparece el mensaje "No input file

specified.". Esto es porque no tenemos todavía ningún archivo en el servidor, como código del proyecto, así como tampoco se ha creado la carpeta "public" que es "document root", o aquella carpeta raíz que hayas configurado en el yaml.

Ahora se trataría de colocar un archivo dentro de la carpeta que has definido como raíz de tu dominio en el campo "to:" de la configuración "sites". Recuerda que esa carpeta está enlazada desde el servidor virtual con una ruta en tu directorio de los proyectos, tal como se ha configurado.

Puedes colocar un index.php con un "Hola mundo" para probar. Lo colocarás en una ruta tal como lo hayas configurado en tu caso, siguiendo los valores definidos en este artículo nos quedaría una como esta:

Ordenador virtualizado:

```
/home/projects/mi_proyecto/public
```

Equivalente en el host anfitrión (ordenador real):

```
~/carpeta_de_proyectos/codigo/mi_proyecto/public
```

Ahora accediendo a la IP del servidor podrías encontrar tu index.php. Además, si has podido configurar correctamente tu virtualhost, podrás acceder a partir del nombre del dominio creado en tu máquina.

```
http://homestead.dw/
```

Con esto es todo, de momento tenemos nuestra máquina funcionando, lista para instalar Laravel, pero aún nos queda toda la [instalación del framework PHP Laravel](#) y algún trabajo más para comenzar a desarrollar. Lo veremos en próximos artículos.

Antes de acabar sería solo comentar que [tenemos un #programadorIO sobre Laravel Homestead](#) que amplía la información relatada en este artículo. Recuerda también que puedes encontrar la [documentación oficial de Homestead para Laravel 5 en este enlace](#).

## Código completo del Homestead.yaml

En los comentarios del artículo nos han pedido reproducir el código completo de un archivo Homestead.yaml. La verdad es que no lo habíamos hecho porque las configuraciones dependen mucho de las preferencias, costumbres, sistemas operativos y rutas de cada uno, por lo que no sería adecuado hacer un "copia-pegar" del archivo. Para evitar confusiones y posibles copias literales lo habíamos dejado pasar. No obstante, para que sirva de referencia para comparar vuestros archivos con los nuestros, dejo aquí el código completo.

```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1
```

```
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub

keys:
  - ~/.ssh/id_rsa

folders:
  - map: ~/html/Homestead/codigo
    to: /home/projects

sites:
  - map: homestead.dw
    to: /home/projects/homestead/public
  - map: testlaravel.com
    to: /home/projects/testlaravel/public

databases:
  - homestead

variables:
  - key: APP_ENV
    value: local

# blackfire:
#   - id: foo
#     token: bar
#     client-id: foo
#     client-token: bar

# ports:
#   - send: 93000
#     to: 9300
#   - send: 7777
#     to: 777
#   protocol: udp
```

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en *03/06/2015*  
Disponible online en <http://desarrolloweb.com/articulos/instalar-homestead-para-laravel5.html>

## Instalar Laravel 5

**Tutorial para aprender a instalar el popular framework PHP Laravel 5, usando Composer.**

En este artículo del [Manual de Laravel](#) vamos a abordar la instalación del framework PHP Laravel 5 usando la conocida herramienta Composer, el gestor de dependencias de PHP, que si no conoces de antemano te recomendamos estudiar con el [Manual de Composer de DesarrolloWeb.com](#).

Además usaremos una máquina virtual para instalar Laravel, que es la denominada Homestead, la

plataforma oficial de desarrollo de Laravel 5, que ya te explicamos en el [artículo sobre Homestead](#). No obstante, lo cierto es que este proceso de instalación lo podrías realizar sobre cualquier ordenador, usando virtualización o usando PHP instalado a mano o con softwares como Xampp o Mamp. El proceso es exactamente el mismo, puesto que se usa Composer también y éste es independiente de la máquina donde lo tengamos instalado, su sistema operativo, etc.



Al final del artículo además encontrarás un vídeo que te explica el proceso de manera visual y agrega otra serie de informaciones de utilidad acerca de la instalación de Laravel 5.

## Requisitos

Para instalar Laravel 5 necesitas estos requisitos:

- PHP 5.5.9
- Las extensiones de PHP:
- OpenSSL
- Mbstring
- Tokenizer

Además requiere que tengas instalado en tu sistema el gestor de dependencias Composer, ya que Laravel lo usa para la instalación.

Como hemos dicho anteriormente también, instalando la máquina virtual Homestead te aseguras de tener todo lo necesario para instalar Laravel 5 sin necesidad de configurar a mano ninguno de los paquetes o extensiones necesarias.

## Iniciar la máquina virtual Homestead

Esta instalación la vamos a realizar usando Homestead, así que os dejamos un par de apreciaciones que necesitarás tener en cuenta.

Debes arrancar la máquina virtual, para lo que te situarás en la línea de comandos en la carpeta donde instalaste Homestead. Desde allí lanzarás el comando:

```
vagrant up
```

Seguidamente tendrás que conectarte por SSH con la máquina virtual de Homestead, porque la instalación de Laravel la vamos a realizar sobre esa máquina y no sobre tu ordenador "real". Lo consigues con el comando:

```
vagrant ssh
```

**Nota:** Si por despiste intentas instalar Laravel 5 más adelante sin haber entrado en la máquina virtual de Homestead por SSH, lo más seguro es que por un tema de dependencias no puedas descargar el instalador de Laravel y te arrojará un error parecido al siguiente:

```
[RuntimeException] Could not load package guzzlehttp/guzzle in http://packagist.org:  
[UnexpectedValueException] Could not parse version constraint ^1.1: Invalid version string "^1.1"  
  
[UnexpectedValueException] Could not parse version constraint ^1.1: Invalid version string "^1.1"
```

Luego tendrás que recordar cómo instalaste Homestead y las carpetas de proyecto que fueron configuradas, así como el virtualhost que fue definido (datos de configuración "folders" y "sites"). Eso se definió en el archivo Homestead.yaml que tienes en la carpeta: ~/.homestead.

**Nota:** Fíjate que la carpeta ~/.homestead. es una carpeta oculta en Linux o Mac, por lo que para localizarla tendrás que listar archivos ocultos con "ls -la". Los que estéis en Windows lo tenéis en esa misma carpeta, en la home de tu usuario de Windows (es lo que significa "~/"), la diferencia es que .homestead no será una carpeta oculta.

## Instalación

Se trata de un par de sencillos pasos (sencillos ya que son comandos de Composer, que es quien hace el trabajo bruto para ti).

**1.- Primero descargamos el instalador de Laravel** y lo disponibilizamos de manera global. Este es un paso que tendrás que hacer una vez únicamente, independientemente del número de instalaciones de Laravel que quieras crear en una máquina.

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

O bien:

```
composer global require "laravel/installer=~1.1"
```

Recuerda que ese comando lo lanzas en la máquina virtual Homestead!!

**Nota:** Si no estás trabajando con Homestead, que ya te lo da todo hecho, tienes que cerciorarte de disponer el directorio ~/.composer/vendor/bin en tu variable de entorno PATH, de modo que el ejecutable de Laravel se pueda localizar en cualquier lugar de tu sistema. Será necesario cuando queramos crear la instalación del framework a partir del instalador descargado.

**Actualizado:** En marzo de 2016, al instalar Laravel, una vez realizado el comando `composer global require "laravel/installer"` me da el problema "laravel: command not found", a pesar de estar en una máquina virtual Homestead, que se supone que ya viene configurada. Ese error indica que el programa instalador de Laravel que te crea Composer no está en el Path. Sin embargo, las rutas han cambiado, porque deberemos confirmar donde está realmente Composer dejando el instalador de Laravel.

Ahora el instalador lo hemos localizado en otro PATH. Te informa de ello el propio Composer al lanzar el comando para bajarse el instalador de Laravel. "Changed current directory to /home/vagrant/.config/composer", así que será esa carpeta la que tengamos que meter en el Path. Esto lo consigues editando un archivo que se llama ".bash\_profile" que tienes en la ruta donde se abre la máquina virtual al conectarte con "vagrant ssh". Puedes editar ese archivo con Vim, por ejemplo lanzando el comando:

```
vim ~/.bash_profile
```

A continuación debes indicar el path que quieres agregar al sistema. Dada la ruta de los ejecutables de Composer, el contenido que tienes que colocar en ese archivo es el siguiente:

```
export PATH=~/.config/composer/vendor/bin:$PATH
```

Para que ese cambio tenga efecto tienes que reiniciar la conexión con ssh. Primero haces "exit" y luego conectas de nuevo con "vagrant ssh".

**2.- Instalamos una instancia del framework en nuestra carpeta de proyecto.** A través del comando "laravel new Nombre\_De\_Proyecto" creamos una instalación limpia del framework. Imagina que tu proyecto se llama "test\_desarrollo", entonces lanzarás este comando:

```
laravel new test_desarrollo
```

Eso nos creará un nuevo directorio en el sistema en el que tendremos la instalación de Laravel lista para usar. Además, todas las dependencias que usa Laravel para funcionar se instalarán en el mismo proceso, con lo que no tendrás que preocuparte por instalar por separado nada más, ni configurar ninguna librería.

Ese comando lo tienes que realizar desde la carpeta de tus proyectos, dentro de la máquina virtual de Homestead. Por ejemplo, si definiste esta configuración en el Homestead.yaml:

```
folders: - map: ~/proyectos/codigo to: /home/projects
```

```
sites: - map: mi_proyecto.dw to: /home/projects/mi_proyecto/public
```

Ese comando lo tendrás que lanzar en el directorio definido en "folders: -> to:". Dada la configuración anterior sería en /home/projects. Insistimos, dentro de la máquina virtual con Homestead.

## Alternativa tradicional de instalación de Laravel

Antes de la versión 5 Laravel se instalaba con un comando diferente, que también puedes seguir usando si es tu preferencia.

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

Eso se conectará con Git para traerse el código de Laravel 5 y lo copiará en tu carpeta, aunque esta opción será un poco más lenta.

## Comprobar la instalación

Finalmente queremos comprobar la instalación. Es tan sencillo como dirigirse con el navegador a la URL donde está Laravel instalado. Si obtenemos el mensaje de bienvenida, como el de esta imagen:



Laravel 5

Very little is needed to make a happy life. - Marcus Antoninus

Si no has tenido suerte a la primera y la instalación no está funcionando no te preocupes puesto que en tu caso pueden quedar algunas cosas por hacer. De hecho lo más seguro es que sea así y que tengas que crear al menos tu archivo de entorno (.env) y la llave de la aplicación. Todo esto está detallado en el siguiente artículo en el que [explicamos posibles tareas a realizar para resolver problemas comunes](#).

Más información y actualizada en la [página de la instalación, dentro de la documentación oficial](#).

Os dejamos con este vídeo, en el que se puede ver cómo configurar Homestead y realizar la instalación de Laravel 5.

Para ver este vídeo es necesario visitar el artículo original en:

<http://desarrolloweb.com/articulos/instalar-laravel5-composer.html>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado por primera vez en *10/06/2015*

Disponible online en <http://desarrolloweb.com/articulos/instalar-laravel5-composer.html>

## Videotutorial: Instalar Homestead y Laravel 5 en Windows

**Guía paso a paso en vídeo sobre la instalación de Laravel 5 en Windows, usando una máquina virtual Homestead, tal como se recomienda para entorno de desarrollo.**

En estos vídeos vamos a enseñar cómo instalar Homestead y cómo instalar Laravel en la máquina virtual de desarrollo (Homestead). Estos vídeos son complementarios a otros que ya hemos publicado en DesarrolloWeb.com, con la diferencia que en este caso vamos a usar el sistema operativo Windows para realizar las tareas.

¿Por qué Windows? porque indudablemente la mayoría de los desarrolladores trabajan con Windows y porque ya habíamos publicado artículos y vídeos en los que se mostraba el proceso en el sistema operativo Mac OS X. En OS X y en Linux la instalación es calcada, pero en Windows algunos lectores nos habían pedido instrucciones más precisas.

El procedimiento para instalar Laravel 5, según la recomendación oficial para entornos de desarrollo, está compuesto de dos pasos.

- Instalar Homestead (una máquina virtual de desarrollo que tiene todos los requisitos para que Laravel funcione perfectamente)
- Instalar Laravel 5, para poder comenzar el desarrollo con este framework PHP.



Estos dos pasos ya los hemos relatado en texto, con instrucciones detalladas, en artículos anteriores del [Manual de Laravel 5](#). Por ese motivo no vamos a repetir las explicaciones y vamos directamente a mostrar los vídeos.

## Instalar Homestead en Windows

En este vídeo realizamos el primero de los pasos, la instalación de Homestead, la plataforma de desarrollo basada en una virtualización de Linux con la "distro" Ubuntu. Osea, estamos diciendo que instalaremos Laravel sobre Windows, pero verdaderamente lo que vamos a crear es una máquina virtual en nuestro ordenador que tendrá el sistema operativo Linux. Homestead es el nombre que recibe el proyecto de esa máquina virtual configurada para instalar Laravel 5 para un entorno de desarrollo.

Los motivos de la instalación de Laravel sobre una máquina virtual Linux y de la existencia de Homestead en general está relatados en el artículo [Instalar Homestead](#).

Para ver este vídeo es necesario visitar el artículo original en:

<http://desarrolloweb.com/articulos/videtutorial-instalar-homestead-laravel-windows.html>

## Instalar Laravel 5 sobre Windows

Ahora vamos a instalar Laravel 5 en Homestead sobre una máquina anfitrión Windows. El trabajo lo hacemos sobre Homestead, por lo que este procedimiento verdaderamente sería exactamente igual en Windows, Linux o Mac, porque realmente estamos haciendo todo el proceso en la máquina virtual. Difieren pocas cosas, como la consola de comandos (terminal) que puedas usar.

El procedimiento ya se explicó en el artículo [Instalar Laravel 5](#), aunque ahora vamos a mostrar cómo se realiza todo esto de manera particular para los usuarios de Windows.

En una máquina Homestead puedes tener varios sitios web funcionando con Laravel, por lo que lo que

vamos a realizar nosotros ahora es la instalación de Laravel 5 para un proyecto en particular. Luego podríamos repetir un proceso de manera similar para administrar varios proyectos como se explica en el artículo [Mantener varios proyectos con Homestead](#).

En el vídeo que puedes ver a continuación realizamos además un pequeño "Hola mundo" para saber si la instalación de Laravel fue realizada con éxito.

Para ver este vídeo es necesario visitar el artículo original en:  
<http://desarrolloweb.com/articulos/videtutorial-instalar-homestead-laravel-windows.html>

Esperamos que estas indicaciones te hayan servido de utilidad y complementen la información presentada anteriormente sobre Laravel 5. Si tienes algún problema con la instalación, y para continuar con los siguientes pasos, te recomendamos también echar un vistazo al artículo sobre [Tareas y problemas comunes al instalar Laravel 5](#).

Acerca de problemas comunes de instalar Laravel, para la realización de estos vídeos me vi con una dificultad y es que, una vez instalado Homestead, no se iniciaba la máquina virtual y por tanto no se podía arrancar el servidor con "vagrant up". Solo tuve que actualizar la versión de VirtualBox instalada en mi ordenador, que estaba un poco viejita.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 09/09/2015  
Disponible online en <http://desarrolloweb.com/articulos/videtutorial-instalar-homestead-laravel-windows.html>

## Tareas adicionales en la instalación de Laravel 5 y problemas comunes

**Para completar la instalación tienes que realizar unas tareas adicionales, en muchos casos, así como resolver problemas comunes.**

En el artículo anterior del [Manual de Laravel](#) ya explicamos cómo realizar la [instalación de básica del framework PHP Laravel 5](#). Como has podido comprobar, instalar Laravel es muy fácil, pero en la mayoría de los casos habrá que completar algunas tareas básicas para dejarlo funcionando correctamente en nuestro sistema. En este artículo pretendemos ayudar explicando cuáles son esas posibles tareas, a la par que analizamos los problemas que te puedes encontrar durante la instalación.

Obviamente sería imposible recabar todos los tipos de problemas con los que te puedes encontrar cuando estés instalando Laravel 5, porque dependen de la configuración concreta del sistema donde lo vayamos a instalar. No obstante, vamos a hacer un listado de cosas que pueden fallar en este momento, en base a nuestra experiencia.

El temido enemigo en este caso es el mensaje "Whoops, looks like something went wrong.", que en la mayoría de los casos, en modo desarrollo, te debería advertir qué es lo que está funcionando mal. A partir de ese mensaje deberías encontrar la pista para saber como solucionar tu problema particular.

**Nota:** Además, recuerda que se recomienda instalar Laravel, al menos durante la [etapa de desarrollo, sobre Homestead](#) y ya con eso eliminaremos la mayoría de las fuentes comunes de problemas, ya que partes de una máquina virtual que tiene todo lo necesario para que Laravel funcione sin problemas.

Ahora van algunas de las posibles causas y soluciones de problemas que hemos detectado.



## Archivo de entorno inexistente

Hemos observado que si instalas Laravel 5 a partir del instalador (`laravel new nombre_proyecto`) no se crea el archivo de entorno. Ese es un archivo que está en la raíz del proyecto, que se llama `".env"`.

Fíjate que en la carpeta raíz debería haber un archivo llamado `.env.example`. Ese archivo es un ejemplo de configuración. La solución sería simplemente duplicar el archivo y llamarle `".env"`.

Atención aquí a la variable de entorno `APP_DEBUG`, que debería estar a `"true"` para que te de una descripción completa de los errores que se puedan producir al ejecutar Laravel. Si no está el archivo de la configuración del entorno, o dentro de él `APP_DEBUG` está a `false`, verás que el mensaje de error de Laravel no aparece con descripción detallada.

Como alternativa, via Composer también se puede acceder a la creación de este archivo de configuración del entorno. Si te fijas en el `composer.json` hay una sección de scripts que sirven para correr comandos post-instalación. Entre ellos hay uno que hace justamente la copia del `.env.example` al `.env`. El script se llama `"post-root-package-install"`. Para ejecutarlo con Composer lanzamos el comando siguiente:

```
composer run-script post-root-package-install
```

**Nota:** Todas las variables de entorno se pueden acceder via `$_ENV` que es una variable PHP superglobal, que mediante un array asociativo te permite acceder sus elementos. Hay un helper llamado `"env"` que justamente está para facilitarte el acceso a las variables de entorno sin usar la superglobal de PHP. Por ejemplo `env('APP_DEBUG')` te daría el valor de la variable de entorno `APP_DEBUG`. Puedes verlo en funcionamiento en el archivo `config/app.php`.

## Permisos de las carpetas

Otra situación que puede dar lugar a errores es que no tengas permisos de escritura en las carpetas que lo necesitan. Directorios dentro de "storage" y "bootstrap/cache" deben tener permisos de escritura para el servidor web. Si has instalado Laravel en una máquina Homestead no deberías preocuparte por este detalle, pero si lo estás haciendo en una máquina distinta quizás tengas que activarlos.

Como estamos en el ordenador de desarrollo podríamos simplemente asignar 777 a los permisos y así nos aseguramos que no nos de problemas este detalle.

```
chmod -R 777 storage
```

Este asunto está documentado en la documentación oficial, aunque no sugieren poner los permisos a un valor concreto. Nosotros sugerimos solo 777 porque es tu máquina local, **nunca se debería hacer eso en el servidor remoto donde va a estar la aplicación en producción.**

## Llave de aplicación (Application Key)

En el caso de la instalación via el instalador de Laravel 5 también hemos observado que falta la llave de aplicación. En la documentación oficial menciona que esa llave debería haberse generado, tanto instalando con el instalador de Laravel (laravel new) como via Composer con la alternativa tradicional. Via composer sí se generó la llave, pero no via el instalador.

La solución es sencilla porque mediante el comando de Artisan key:generate se realiza todo el trabajo para ti.

**Nota:** Ojo, porque la llave se genera en el archivo de entorno .env y si ese archivo no existe quizás no funcione el comando de Artisan key:generate. Lee el punto anterior "Archivo de entorno inexistente" para encontrar más información.

Dentro de la raíz de tu proyecto, donde se encuentra el .env lanza el comando:

```
php artisan key:generate
```

Eso te debería lanzar algo como la siguiente salida:

```
Application key [dpGvjrnKLGszdgck1YSLrSMGeN61dy] set successfully.
```

Además el propio comando te actualiza el archivo .env, pero si no es así podrías hacerlo a mano tú mismo, en el epígrafe APP\_KEY:

```
APP_KEY=dpGvjrnKLGszdgck1YSLrSMGeN61dy
```

Como nos avisan en la documentación, ten cuidado con este detalle porque si la llave de aplicación no ha sido generada los datos de las sesiones de usuario y otras informaciones encriptadas no estarán seguras.

## Revisar config/app.php

En este archivo encontrarás información de configuración de la aplicación. Generalmente no necesitas

tocarlo si estás impaciente y quieres ver ya el framework funcionando. Sin embargo encontrarás algunas cosas útiles como la variable "timezone" y "locale" que en una aplicación en producción desearás editar.

Otros elementos que se pueden configurar en otros archivos son Chache, Database, Session, que veremos más adelante.

## URLs amigables a usuarios / buscadores

Laravel es capaz de mostrar las URL de aplicación de una manera amistosa para el usuario, y para los buscadores que puedan recorrer la página. Todo en Laravel comienza con un index.php que está dentro de la carpeta "public" (que debería ser tu document root). Para eliminar ese index.php de las URL y que éstas queden más limpias es posible que necesites tocar alguna cosa. Aunque si estás instalando via Homestead no deberías encontrarte con ningún problema en este sentido.

Laravel 5 viene con un archivo .htaccess que sirve para generar las URL amigables en Apache. Solo ten en cuenta que tu apache debe tener activado el mod\_rewrite para que las redirecciones de .htaccess funcionen correctamente.

Además del .htaccess que encuentras en la carpeta "public" en la documentación de Laravel, en la sección de instalación <http://laravel.com/docs/#installation> y luego en la subsección "Pretty URL" te ofrecen una versión reducida del .htaccess que podrías probar si la que provee el framework no hace correctamente su trabajo.

Para los que están en Nginx se debe agregar una directiva en la configuración del sitio:

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

Esto es todo por el momento. Si queréis profundizar en el tema os recomendamos que os veáis los vídeos que encontraréis en el siguiente enlace: [Primeros pasos con Laravel 5](#)

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 17/06/2015  
Disponible online en <http://desarrolloweb.com/articulos/tareas-instalacion-laravel5-problemas.html>

## Mantener varios proyectos con Homestead

### Tutorial para albergar varios proyectos realizados con Laravel en una misma máquina virtual Homestead, con varias instalaciones del framework PHP.

Estás trabajando con Laravel y te has decantado, tal como se recomienda, trabajar en el entorno Homestead. Tienes una máquina virtual con una instalación de Laravel, pero quieres manejar diferentes proyectos, de manera independiente, reutilizando la misma máquina virtual, para no tener que gastar mayor espacio en disco o tiempo en crear una máquina nueva.

Esta es una situación muy común y que tiene una sencilla solución, simplemente a través de la configuración del archivo Homestead.yaml, creando distintos host para cada proyecto, asociados a otras carpetas del

servidor.

Obviamente, para llegar a este artículo tienes que saber [Qué es Homestead](#) y saber cómo [instalar Laravel](#). Lo que vamos a aprender de nuevo es algo muy sencillo, simplemente configurar varios host virtuales para cada proyecto.



## Configuración de "sites" en Homestead.yaml

Recuerda que el archivo Homestead.yaml está en la carpeta ".homestead" que está situada en la raíz de tu usuario. Tanto en Windows, Linux o Mac: "~/.homestead".

**Nota:** Solo un detalle para los de Windows, la carpeta "~/.homestead" físicamente estará en una ruta del disco donde tengas instalado Windows. Si tu usuario se llama "carlos" y has instalado Windows en el disco C, tu carpeta estaría físicamente en una ruta absoluta como esta C:\Users\carlos.homestead. Recuerda que si usas un terminal medianamente bueno como el que te viene con Git (git bash) sí que te reconocerá las rutas comenzando con ~/ como rutas a la carpeta de tu usuario Windows.

Dentro del Homestead.yaml está el epígrafe "sites", sobre el que podemos construir cualquier número de hosts virtuales (virtualhost en la terminología de servidores, que significa que el servidor web reconocerá esa carpeta como un sitio aparte, al que se accederá por un nombre de dominio independiente).

Si queremos crear varios host independientes, uno para cada proyecto Laravel, simplemente tenemos que listar ese número de virtualhost tal como sigue:

```
sites:
  - map: proyecto1.local.com
    to: /home/projects/proyecto1/public
  - map: proyecto2.local.com
    to: /home/projects/proyecto2/public
  - map: proyecto3.local.com
    to: /home/projects/proyecto3/public
```

Podemos tener tantos virtualhost como queramos. Luego se trata de crear las carpetas mapeadas a cada uno de esos host virtuales en la máquina Homestead, algo que seguramente harás solamente cuando llegue el paso de la [instalación del framework Laravel 5](#) tal como se explicó anteriormente.

## Aprovisionar los cambios en el archivo Homestead.yaml

Si simplemente cambias el archivo Homestead.yaml las nuevas configuraciones no funcionarán, ni tan siquiera si reinicias la máquina virtual. Por ello tendrás que hacer un paso adicional para aprovisionar de nuevo la máquina atendiendo a los nuevos valores de configuración.

Eso se consigue con un comando como el que sigue:

```
vagrant reload --provision
```

Ese comando para la máquina virtual y luego la reconfigura, tomando los nuevos datos indicados en el archivo de configuración. Luego la vuelve a encender ya con las nuevas configuraciones.

**Nota:** Solo para mencionarlo por si a alguien se le ocurre, no hace falta hacer el `vagrant destroy` para eliminar la máquina virtual de Homestead y luego volverla a crear. Eso podría funcionar, pero se te eliminarán los proyectos que has creado.

## Volver a configurar el archivo hosts

También podrá resultar obvio, pero hay que comentar que, inmediatamente después de crear los nuevos virtualhost y antes de poder acceder a ellos por los nombres de dominio configurados, deberás editar tu archivo hosts para agregar la IP del servidor asociada al nuevo nombre de dominio que se está definiendo. Colocarás una línea como esta:

```
192.168.10.10 proyecto2.local.com
```

También ojo con esa línea porque tu máquina Homestead podría tener otra IP distinta si lo editaste en el archivo Homestead.yaml y lógicamente el nombre del dominio para el virtualhost también será otra, que hayas configurado en tu archivo yaml.

## Conclusión

Con eso es todo! en lugar de crear máquinas nuevas para colocar proyectos distintos de Laravel, lo que generalmente harás es crear nuevos host virtuales sobre la misma máquina, así compartes el mismo entorno de desarrollo para diferentes proyectos Laravel 5.

Es una situación bastante común, pero no implica necesariamente que lo tengas que hacer así. Para un proyecto, por cualquier motivo, podrías preferir crear una máquina virtual totalmente independiente, creando una nueva instancia de Homestead, en lugar de reutilizar la que ya tenías.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en 25/06/2015  
Disponible online en <http://desarrolloweb.com/articulos/mantener-proyectos-homestead.html>



# Primeros pasos con Laravel

En los siguientes artículos ponemos las manos en el código para introducirnos en Laravel. Encontrarás una vista de pájaro de lo que te ofrece el framework PHP. Además te explicamos con detalle cómo funciona el sistema de rutas de Laravel y te ayudamos a realizar tus primeras páginas basadas en este sistema.

## Primera prueba de Laravel con el sistema de rutas

**Este sería un típico Hola Mundo realizado en Laravel, en el que podremos construir una primera ruta dentro de nuestra aplicación.**

Imaginamos que estás ansioso por pasar de la pantalla inicial de Laravel y ver ya algún tipo de salida que se haya producido mediante un código tuyo propio. En ese caso en este artículo podremos ayudarte, porque vamos a probar el sistema de enrutamiento de solicitudes Laravel y hacer una primera página que muestre una salida.

Obviamente, las cosas que vamos a hacer en este ejemplo no son del todo correctas, pero Laravel nos las permite. Nos referimos a crear salida sin usar, o dejando de lado, lo que sería el patrón de diseño MVC. Vamos a intentar ser breves, así que nos vamos a dar algunas licencias en ese sentido.

## Identificar el archivo donde se generan las rutas de la aplicación

Comenzamos viendo dónde se generan las rutas que nuestra aplicación va a responder, a través del framework Laravel. Estamos trabajando con la versión 5.1, aunque en todo Laravel 5 debe de ser igual.

Navega con tu explorador de archivos a la carpeta `app/Http` y encontrarás allí un fichero llamado `routes.php`. Ese es el archivo que contiene todas las rutas de la aplicación.



Una vez abierto observarás que tiene una llamada a un método `Route::get()`. Ese es el método que usaremos para generar las rutas.

**Nota:** El orden en el que pongamos las rutas es importante, puesto que primero se gestionarán las que aparezcan antes en el código.

## Crear nuestra primera ruta

Guiándonos por el ejemplo de ruta que encontramos podemos crear nuestra propia ruta.

```
Route::get('/test', function(){
    echo "Esto es una simple prueba!!";
});
```

Aunque tendremos que volver en breve sobre las explicaciones del sistema de rutas, es importante señalar algunos puntos.

1. Hay que fijarse que cualquier ruta corresponde con un verbo del HTTP (get, post, etc.), que es el nombre del método que estamos invocando sobre la clase Route. En nuestro caso usamos el método get, que es el más común de las acciones del protocolo HTTP.
2. Además observa que ese método recibe dos parámetros, el primero de ellos es el patrón que debe cumplirse para que esa ruta se active. En nuestra ruta hemos colocado "/test" que quiere decir que desde la home de la aplicación y mediante el nombre "test" se activará esa ruta. En este caso el patrón es una simple cadena, pero en general podrá ser mucho más complejo, generando partes de la ruta que sean parámetros variables. Todo eso lo estudiaremos más adelante.
3. Como segundo parámetro al método get() para definir la ruta indicamos una función con el código a ejecutar cuando Laravel tenga que procesarla. Observarás que es una "función anónima" cuyo código es un simple echo para generar una salida.

**Nota:** Para quien no lo sepa, una función anónima es simplemente una función que no tiene nombre. Están disponibles en PHP a partir de la versión 5.3. Estas funciones se usan típicamente como parámetros en funciones o como valores de retorno y quizás quien venga de Javascript las conocerá más que de sobra, porque en ese lenguaje se usan intensivamente. Otro nombre con el que te puedes referir a funciones anónimas es "closure".

## Acceder a nuestra ruta

Viene la parte más fácil, que es acceder con el navegador a la nueva ruta que acabamos de definir. Si tu proyecto estaba en un virtualhost, usarás el nombre de tu dominio asociado a ese virtualhost, seguido con el patrón de la ruta que acabamos de crear ("/test"), quedando algo como esto:

```
http://example.com/test
```

Accediendo a esa dirección deberías ver el mensaje configurado en la closure enviada al generar la ruta.

## Observar las rutas posibles dentro de una aplicación

Antes de acabar queremos comentar un comando de Artisan que sirve para mostrar todas las rutas disponibles en un proyecto o aplicación web creada con Laravel. Es un comando muy útil para saber qué tipos de rutas puedes tener y qué valores te van a permitir, parámetros, etc.

```
php artisan route:list
```

Ese comando te producirá una salida como la que puedes ver en la siguiente imagen.



Domain	Method	URI	Name	Action	Middleware
	GET HEAD	test		Closure	
	GET HEAD	/		Closure	

## Conclusión

De momento se trata de una simple ruta, pero al menos ya hemos podido tocar algo dentro del framework y cerciorarnos que todo está funcionando. En adelante podremos complicar los ejemplos todo lo que queramos y producir cualquier tipo de ruta, puesto que el sistema de routing de Laravel es muy potente.

Además, como ya hemos dejado entrever, está preparado para el trabajo con APIs REST ya que te admite directamente los distintos verbos del HTTP que se suelen usar dentro de estos sistemas. Que no te lie esto último, en realidad no es necesario construir ningún tipo de API REST para trabajar con Laravel 5, pero si lo necesitas lo tendrás bien fácil.

En el siguiente vídeo puedes ver cómo se ha creado una primera ruta como la de este artículo, simplemente de prueba, que nos sirve para saber que Laravel está funcionando correctamente.

Para ver este vídeo es necesario visitar el artículo original en:

<http://desarrolloweb.com/articulos/primer-prueba-laravel-sistema-rutas.html>

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado por primera vez en 03/07/2015

Disponible online en <http://desarrolloweb.com/articulos/primer-prueba-laravel-sistema-rutas.html>

## Estructura de carpetas de Laravel 5

Un resumen de la estructura de carpetas del framework Laravel 5, a vista de pájaro, sin entrar en demasiado detalle, pero que nos ayude a ubicar los componentes principales.

Laravel, así como cualquier framework PHP, propone una estructura de carpetas, con la que organizar el código de los sitios o aplicaciones web. Como ya somos desarrolladores aplicados ;) deberíamos saber que cuando más separemos el código por responsabilidades, más facilidad de mantenimiento tendrán nuestras aplicaciones. Por lo tanto, en el desarrollo del framework se cuidan mucho estos detalles, para que aquellos desarrolladores que lo utilicen sean capaces también de organizarse de una manera correcta.

En este artículo del [Manual de Laravel](#) pretendemos dar un rápido recorrido a las carpetas que vas a encontrar en la instalación de partida de Laravel 5, teniendo en cuenta que en futuras ocasiones quedará pendiente un análisis en profundidad de cada uno de estos elementos.

El objetivo de este artículo es que sepamos más o menos dónde están las cosas en el framework y que tengamos una ligerísima idea de lo que es cada cosa.



## Archivos en la carpeta raíz

Los archivos que tenemos sueltos en la carpeta raíz de Laravel son los siguientes (o al menos los más importantes que debes ir conociendo).

**.env** Es la definición de las variables de entorno. Podemos tener varios entornos donde vamos a mantener la ejecución de la aplicación con varias variables que tengan valores diferentes. Temas como si estamos trabajando con el debug activado, datos de conexión con la base de datos, servidores de envío de correo, caché, etc.

**Nota:** Si `.env` no se ha generado en tu sistema lo puedes generar a mano mediante el archivo `.env.example`, renombrando ese fichero o duplicándolo y renombrando después. Esto se explicó anteriormente en el artículo llamado [Tareas para completar la instalación y problemas comunes](#).

**composer.json** Que contiene información para Composer. Para conocer algo más de este fichero es mejor que te leas el [Tutorial de Composer](#).

Además en la raíz hay una serie de archivos que tienen que ver con Git, el readme, o del lado frontend el `package.json` o incluso un `gulpfile.js` que no vendría muy al caso comentar aquí porque no son cosas específicas de Laravel.

## Carpeta vendor

Esta carpeta contiene una cantidad de librerías externas, creadas por diversos desarrolladores que son dependencias de Laravel. La carpeta vendor no la debemos tocar para nada, porque la gestiona Composer, que es nuestro gestor de dependencias.

Si nosotros tuviésemos que usar una librería que no estuviera en la carpeta vendor la tendríamos que especificar en el archivo composer.json en el campo require. Luego hacer un "composer update" para que la nueva dependencia se instale.

## Carpeta storage

Es el sistema de almacenamiento automático del framework, donde se guardan cosas como la caché, las sesiones o las vistas, logs, etc. Esta carpeta tampoco la vamos a tocar directamente, salvo que tengamos que vaciarla para que todos esos archivos se tengan que generar de nuevo.

También podemos configurar Laravel para que use otros sistemas de almacenamiento para elementos como la caché o las sesiones.

En cuanto a las vistas cabe aclarar que no son las vistas que vamos a programar nosotros, sino las vistas una vez compiladas, algo que genera Laravel automáticamente en función de nuestras vistas que meteremos en otro lugar.

## Carpeta resources

En Laravel 5 han creado esta carpeta, englobando distintos tipos de recursos, que antes estaban dentro de la carpeta app. En resumen, en esta carpeta se guardan assets, archivos de idioma (lang) y vistas.

Dentro de views tienes las vistas que crearás tú para el desarrollo de tu aplicación. En la instalación básica encontrarás una serie de subcarpetas con diversos tipos de vistas que durante el desarrollo podrías crear, vistas de emails, errores, de autenticación. Nosotros podremos crear nuevas subcarpetas para organizar nuestras vistas.

A propósito, en Laravel se usa el motor de plantillas Blade. Lo estudiaremos más adelante.

## Carpeta Public

Es el denominado "document root" del servidor web. Es el único subdirectorio que estará accesible desde fuera mediante el servidor web. Dentro encontrarás ya varios archivos:

**.htaccess** En el caso de Apache, este es el archivo que genera las URL amigables a buscadores.

**favicon.ico** Es el icono de nuestra aplicación, que usará el navegador para el título de la página o al agregar la página a favoritos.

**index.php** Este es un archivo muy importante, que hace de embudo por el cual pasan todas las solicitudes a archivos dentro del dominio donde se está usando Laravel. Estaría bien que abrieras ese index.php para observar lo que tiene dentro. Para el que conozca el patrón "controlador frontal" o "front controller" cabe decir que este index.php forma parte de él.

**robots.txt** Que es algo que indica las cosas que puede y no puede hacer a la araña de Google y la de otros

motores de búsqueda.

En la carpeta public podrás crear todas las subcarpetas que necesites en tu sitio web para contener archivos con código Javascript, CSS, imágenes, etc.

## Carpeta database

Contiene las alimentaciones y migraciones de la base de datos que veremos más adelante.

## Carpeta bootstrap

Permite el sistema de arranque de Laravel, es otra carpeta que en principio no necesitamos tocar.

## Carpeta config

Esta carpeta contiene toda una serie de archivos de configuración. La configuración de los componentes del framework se hace por separado, por lo que encontraremos muchos archivos PHP con configuraciones específicas de varios elementos que seguramente reconoceremos fácilmente.

La configuración principal está en app.php y luego hay archivos aparte para configurar la base de datos, las sesiones, vistas, caché, mail, etc.

## Carpeta app

Es la última que nos queda y es la más importante. Tiene a su vez muchas carpetas adicionales para diversas cosas. Encuentras carpetas para comandos, para comandos de consola, control de eventos, control de excepciones, proveedores y servicios, etc.

Es relevante comentar que en esta carpeta no existe un subdirectorio específico para los modelos, sino que se colocan directamente colgando de app, como archivos sueltos.

La carpeta Http que cuelga de App contiene a su vez diversas carpetas importantes, como es el caso de aquella donde guardamos los controladores, middleware, así como el archivo de rutas de la aplicación routes.php que vimos anteriormente, entre otras cosas.

## Conclusión

Hemos podido conocer de una manera breve los componentes principales del framework Laravel. Como todo en la vida, visto así rápidamente, nos puede parecer que son muchas cosas y difíciles de entender y usar, pero no debemos preocuparnos porque los iremos conociendo poco a poco. Si tienes una base razonable de conocimientos de PHP y patrones de diseño orientado a objetos te resultará fácil usarlos leyendo los siguientes artículos del [Manual de Laravel](#).

Este artículo es obra de *Miguel Angel Alvarez*.  
Fue publicado por primera vez en 08/07/2015  
Disponible online en <http://desarrolloweb.com/articulos/estructura-carpetas-laravel5.html>

## Verbos en las rutas de Laravel

---

Explicamos cómo el HTTP Routing System de Laravel 5 permite la configuración de diversos verbos con los que especificar qué tipo de operación se desea realizar.

El sistema de rutas de Laravel es bastante sencillo de utilizar, por lo que en pocos pasos podremos crear todo tipo de rutas bastante complejas. No obstante, al ser tan potente, si quieres analizarlo en profundidad gastarás tiempo. En este artículo pretendemos hacer una primera aproximación al sistema de enrutado de solicitudes HTTP, analizando lo que son los verbos de las solicitudes. En futuras entregas analizaremos otros elementos fundamentales.

Para comenzar no está de más una primera pasada por la documentación oficial, que de un vistazo rápido te dará una idea sobre la multitud de cosas que podrás realizar y configurar. Nosotros estamos utilizando como referencia para este artículo la [documentación del sistema de rutas de Laravel 5.1](#).

Primero queremos recordar que en el [Manual de Laravel 5](#) ya explicamos cómo realizar una primera ruta simple y vimos cosas como la configuración de las funciones anónimas o closures para especificar el tipo de tratamiento que se debe realizar para cada ruta. Esto lo puedes encontrar en el artículo [Primera prueba de Laravel](#) con el sistema de rutas y para no repetirnos hay cosas que no volveremos a explicar.



### Verbos HTTP

Ya los mencionamos, pero cabe incidir de nuevo sobre los verbos HTTP porque forman parte de las rutas Laravel. Los verbos del HTTP son algo relativo al protocolo de comunicación HTTP, usado en la web, por lo tanto no tienen que ver con Laravel específicamente.

Sirven para decir el tipo de acción que quieres realizar con un recurso (la URL), siendo posible especificar en el protocolo (la formalidad de la conexión por HTTP entre distintas máquinas) el verbo o acción que deseamos realizar.

Son 8 verbos en el protocolo: Head, Get, Post, Put, Delete, Trace, Options, Connect. Y si ya eres desarrollador web y no te suena haberlos usado nunca te diré que realmente no es así. Get es la acción que se usa en el protocolo habitualmente, cuando se consulta un recurso. Sirve para recuperar información. Post por su parte es la acción que se realiza cuando se mandan datos, de un formulario generalmente. Los otros verbos no se usan de una manera muy habitual, pero sí se han dado utilidad en el desarrollo de lo que se conoce como [API REST](#).

Laravel ya viene preparado para implementar APIs REST, así que usa los verbos del protocolo para generar

sus rutas de aplicación. De momento nos debe quedar claro que el acceso a un URI como `"/test"` puede tener diversos verbos a la hora de realizarse por parte de un sistema. Para Laravel el acceso a `"/test"` (o cualquier otro URI) usando la acción `Get` no es el mismo que el acceso a `"/test"` usando la acción `Post`.

En resumen, el significado de los verbos es:

- **Get:** recuperar información, podemos enviar datos para indicar qué se desea recuperar, pero mediante `get` en principio no se debería generar nada, ningún tipo de recurso en el servidor o aplicación, porque los datos se verán en la URL y puede ser inseguro.
- **Post:** enviar datos que se indicarán en la propia. Esos datos no se verán en la solicitud, puesto que viajan con la información del protocolo.
- **Put:** esto sirve para enviar un recurso, subir archivos al servidor, por ejemplo. No está activo en muchas configuraciones de servidores web. Con `put` se supone que los datos que estamos enviando son para que se cree algún tipo de recurso en el servidor.
- **Delete:** borrado de algo.
- **Trace, patch, link, unlink, options y connect...** no están entre los verbos comunes de Laravel, por lo que [te referimos a la Wikipedia para más información.](#)

## Registrando rutas con sus verbos

Las rutas se registran con los verbos, usando el método correspondiente de la fachada `Route`. Ya se ha mencionado pero lo repetimos: **Todo el listado de rutas que queramos producir se debe escribir en el fichero `app/Http/routes.php`**

**Nota:** Es la primera vez en este manual que usamos la terminología "fachada" que es un patrón de diseño de software orientado a objetos usada de manera intensiva en Laravel. Podrás encontrar referencias a este patrón a partir de la palabra "facade", en inglés. De momento vamos a obviar esa palabra para tratarla en detalle más adelante.

La clase `Route` (la fachada del sistema de rutas) tiene varios métodos estáticos con los verbos sobre los que queremos dar de alta una ruta de la aplicación. Anteriormente comentamos que estos métodos reciben dos parámetros, uno el patrón o URI que queremos registrar y otro la función con el código a ejecutar.

```
Route::get('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo
    return 'get';
});

Route::post('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo POST
    return 'post';
});
```

```
Route::put('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo PUT
    return 'put';
});

Route::delete('/probando/ruta', function(){
    //código a ejecutar cuando se produzca esa ruta y el verbo DELETE
    return 'delete';
});
```

Usar varios verbos a la vez al registrar una ruta

Como puedes imaginar, existen casos en los que te puede interesar registrar una ruta que tenga validez con varios verbos, indicando una única función que los resuelva todos. Esto se puede conseguir mediante dos métodos distintos.

```
Route::match(['get', 'post', 'put'], '/testing', function () {
    echo 'Ruta testing para los verbos GET, POST, PUT';
});
```

Como puedes apreciar, el método `match` requiere un parámetro adicional, el primero, en el que indicamos un array con los verbos que queremos aplicar a esta ruta.

Por otra parte tenemos el método `any`, que es como un comodín, que sirve para cualquier tipo de verbo del HTTP.

```
Route::any("/cualquiercosa", function(){
    echo 'La ruta /cualquiercosa asociada a cualquier verbo';
});
```

## Cómo probar los verbos de HTTP

Desde un navegador podrás probar fácilmente cualquier ruta con el verbo GET. Será simplemente escribir la URL en la barra de direcciones del navegador. Sin embargo, para probar el método post necesitarás crearte un formulario. Y si se trata de probar PUT o DELETE la cosa es más complicada porque necesitarías algún tipo de script especial.

Pero hay una alternativa muy cómoda que te permitirá probar cualquier tipo de verbo y enviar cualquier tipo de parámetro en la solicitud HTTP. Se trata de utilizar algún cliente Rest que podemos descargar como complemento en el navegador.

Una posibilidad es Postman, un cliente REST que es muy fácil de usar. Esta extensión disponible en [Chrome](#) o también en [Firefox](#).

A través de esta herramienta eres capaz de escribir una URL, el método o verbo de la conexión y a través del envío de la solicitud (botón send) recibir y visualizar la respuesta de Laravel.

Así que con postman ya podemos crear todo tipo de rutas en el sistema de enrutamiento y comprobar qué

es lo que está pasando. Como sugerencia os indico justamente eso, probar distintos verbos, crear el código para registrar las rutas y ejecutarlas a través de Postman.

**Nota:** Si quieres probar rutas diferentes de GET observarás que no se puede. Al intentar hacer post, put o delete te sale un mensaje: "TokenMismatchException".

Esto es debido a que en Laravel está activado un sistema antispam para solicitudes diferentes de GET, potencialmente más peligrosas, por el cual se debe comprobar un token. Hablaremos de ello más adelante. De momento vamos simplemente a desactivarlo.

En el archivo `app/Http/Kernel.php` encontrarás el "global HTTP middleware stack", osea la pila de middlewares que se ejecutan de manera global en las solicitudes HTTP. Busca la línea que hace la carga de `VerifyCsrfToken::class`, y coméntala.

```
//\App\Http\Middleware\VerifyCsrfToken::class,
```

## Conclusión

Hemos hablado de verbos en la solicitud HTTP, y a la vez hemos continuado aprendiendo del sistema de routing de Laravel. Pero nos gustaría que quedase claro que desde el navegador generalmente vas a realizar siempre solicitudes GET, o POST si envías datos de formulario.

Para usar los otros verbos que puedes controlar en Laravel realmente necesitas de un cliente REST, que sea capaz de ejecutar acciones como PUT o DELETE. Tendrán sentido usar en tus rutas cuando estés construyendo un API REST con Laravel.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en 16/07/2015  
Disponible online en <http://desarrolloweb.com/articulos/verbos-rutas-laravel.html>

## Parámetros en las rutas de Laravel 5

### Explicaciones detalladas sobre cómo trabajar con parámetros en las rutas del framework PHP Laravel, versión 5.

En el Manual de Laravel 5 hemos comenzado a tratar el sistema de rutas en profundidad. En el artículo anterior os hablábamos sobre las rutas y sus verbos HTTP, en este momento vamos a dedicarnos a algunos temas relacionados con el paso de parámetros.

Las rutas en las aplicaciones web corresponden con patrones en los que en algunas ocasiones se encuentran textos fijos y en otras ocasiones textos que van a ser variables. A lo largo de este artículo nos vamos a referir a esos textos variables con el nombre de parámetros y se pueden producir con una sintaxis de llaves.

Veamos varias rutas de una supuesta aplicación para entender qué es esto de los parámetros.



```
example.com/colaboradores/miguel
example.com/colaboradores/carlos
example.com/tienda/productos/34
example.com/agenda/julio/2015
example.com/categoria/php
example.com/categoria/php/2
```

Podrás apreciar esas rutas y verás que hay zonas que son variables, por ejemplo, el nombre del colaborador que se pretende ver, el identificador del producto de una tienda, el mes y el año de una agenda, la categoría que se desea ver o la página de un hipotético listado de artículos de una categoría.

## Crear rutas con parámetros

A continuación vamos a mostrar cómo se podrían crear algunas de esas rutas anteriores en el sistema de routing de Laravel.

```
Route::get('colaboradores/{nombre}', function($nombre){
    return "Mostrando el colaborador $nombre";
});
```

Como puedes observar, en este ejemplo el nombre del colaborador es variable, por ello se expresa como un parámetro, encerrado entre llaves. En la función closure recibimos el parámetro y podemos trabajar con él, como con cualquier parámetro de una función.

El nombre del parámetro (variable con la que recibimos ese parámetro en la función anónima o closure) es independiente, como puedes ver en el siguiente ejemplo. Podríamos tener en el patrón de la ruta definido el `{id}` y luego recibir ese dato en una variable `$id_producto`. Lo que importa en este caso es el orden con el que se han definido los parámetros en el patrón del URI.

```
Route::get('tienda/productos/{id}', function($id_producto){
    return "Mostrando el producto $id_producto de la tienda";
});
```

Se pueden enviar varios parámetros si se desea. Simplemente los recogeremos en la función, de una manera

similar.

```
Route::get('agenda/{mes}/{ano}', function($mes, $ano){
    return "Viendo la agenda de $mes de $ano";
});
```

Aquí lo importante no son el nombre de las variables con las que recoges los parámetros, sino el orden en el que fueron declarados en el patrón de la URI. Primero recibimos el parámetro \$mes porque en la URI figura con anterioridad. Recuerda que el patrón era algo como "agenda/julio/2015".

## Enviar los parámetros a los controladores

Aunque todavía no hemos hablado de los controladores, queremos poner un ejemplo aquí, aunque realmente no cambia mucho sobre lo que hemos visto.

**Nota:** Aunque no hayamos explicado qué son los controladores estamos seguros que muchos de los lectores están familiarizados con esa terminología, en ellos estamos pensando cuando escribimos estas líneas. Luego hablaremos de lo que hacen los controladores con detalle pero para aclarar conceptos el lector que lo necesite puede ir revisando el [artículo sobre Qué es MVC](#).

Al definir una ruta a un controlador tenemos que indicarlo en el método que registra la ruta. En ese método, en lugar de definir una función anónima indicaremos el nombre y método del controlador a ejecutar.

```
Route::get('tienda/productos/{id}', 'TiendaController@producto');
```

Luego en el controlador recibimos el parámetro definido en el patrón de URI de la ruta registrada.

```
public function producto($id)
{
    return "Esto muestra un producto. Recibiendo $id";
}
```

## Parámetros opcionales

Hay ocasiones en las que se especifican parámetros que tienen valores opcionales. Mira las siguientes rutas:

```
example.com/categoria/php
example.com/categoria/php/2
```

Como puedes comprobar, a veces indicamos la página de categoría que se desea mostrar y a veces no se indica nada. Esto imitaría el funcionamiento de un paginador, en la que, si no se recibe nada, se mostraría la primera página y si se indica un número de página se mostraría esa en concreto.

Los parámetros opcionales en Laravel se indican con un símbolo de interrogación. En este código en el patrón de la URI observarás que la página es opcional.

```
Route::get('categoria/{categoria}/{pagina?}', function($categoria, $pagina = 1){
    return "Viendo categoría $categoria y página $pagina";
});
```

Resultará de utilidad, al definir el closure, indicar un valor por defecto para aquellos parámetros que son opcionales. Si no lo hacemos, a la hora de usar ese parámetro dentro de la función anónima, nos mostrará un mensaje de error en caso que no lo enviemos "Missing argument 2 for...".

## Precedencia de las rutas

Con dos rutas registradas que tienen patrones distintos, si por un casual una URI puede encajar en el formato definido por ambos patrones, el que se ejecutará será el que primero se haya escrito en el archivo routes.php.

```
Route::get('categoria/{categoria}', function($categoria){
    return "Ruta 1- Viendo categoría $categoria y no recibo página";
});

Route::get('categoria/{categoria}/{pagina?}', function($categoria, $pagina=1){
    return "Ruta 2 - Viendo categoría $categoria y página $pagina";
});
```

En esas dos rutas tenemos patrones diferentes de URI, pero si alguien escribe:

```
example.com/categoria/laravel/
```

Esa URL podría casar con ambos patrones de URI. En este caso, el mensaje que obtendremos será:

```
Ruta 1 - Viendo categoría laravel y no recibo página
```

En este caso no necesitaríamos haber indicado el valor por defecto en \$pagina para la segunda ruta registrada, pues nunca se invocaría a esa ruta sin enviarle algún valor de página.

## Aceptar solamente determinados valores de parámetros

Hay una posibilidad muy útil con los parámetros de las rutas que consiste en definir expresiones regulares para especificar qué tipo de valores aceptas en los parámetros. Por ejemplo, un identificador de producto debe ser un valor numérico o un nombre de un colaborador te debe aceptar solamente caracteres alfabéticos. Si has entendido esta situación observarás que hasta el momento, tal como hemos registrado las rutas, se aceptarían todo tipo de valores en los parámetros, generando rutas que muchas veces no deberían devolver un valor de página encontrada. Por ejemplo:

```
example.com/colaboradores/666  
example.com/tienda/productos/kkk
```

Recuerda que hemos dicho que colaboradores debería ser un valor de tipo alfabético y que el identificador de producto solo puede ser numérico. Así que vamos a modificar las rutas para poder agregarle el código que nos permita no aceptar valores que no deseamos.

```
Route::get('colaboradores/{nombre}', function($nombre){  
    return "Mostrando el colaborador $nombre";  
})->where(array('nombre' => '[a-zA-Z]+'));
```

Como puedes apreciar, se le coloca encadena, sobre la ruta generada, un método `where()` que nos permite especificar en un array todas las reglas que se le deben aplicar a cada uno de los parámetros que queremos restringir.

**Nota:** Si la ruta define varios parámetros no estamos obligados a colocar expresiones regulares para todos, solo aquellos que queramos restringir a determinados tipos de valores.

Ahora mira este código, donde tenemos dos rutas definidas:

```
Route::get('tienda/productos/{id}', function($id_producto){  
    return "Mostrando el producto $id_producto de la tienda";  
})->where(['id' => '[0-9]+']);  
  
Route::get('tienda/productos/{id}', 'PrimerController@test');
```

En la primera ruta estamos obligando a que el parámetro `id` sea un número. Sin embargo, en la segunda ruta tenemos el mismo patrón sin definir el tipo de valor al que queremos restringir. En este caso, si el `id` fuera un valor numérico se iría por la primera ruta y si es un valor diferente se iría por la segunda. En fin, que podemos configurar las rutas para hacer muchas cosas distintas y definir innumerables comportamientos atendiendo a las necesidades de la aplicación y a la manera en la que prefiramos organizar nuestro código.

De momento con lo que hemos visto sobre rutas podemos jugar bastante, así que para avanzar en el uso de Laravel 5 vamos a cambiar de tercio en los próximos artículos para tratar otros asuntos que a buen seguro estarás impaciente por conocer.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en 20/07/2015  
Disponible online en <http://desarrolloweb.com/articulos/parametros-rutas-laravel5.html>

# Introducción a los componentes principales de Laravel

En los siguientes artículos vamos a abordar, uno a uno, los componentes principales que encuentras en el framework: controladores, vistas, modelos, sistemas request y response, etc. El objetivo es que conozcas las piezas fundamentales para el desarrollo de aplicaciones web con Laravel y comiences a usarlas en ejemplos más elaborados.

## Introducción a las vistas en Laravel 5

**Cómo se trabaja con vistas en Laravel 5, creamos las primeras vistas y las llamamos desde el sistema de enrutado.**

En este artículo del [Manual de Laravel 5](#) vamos a comenzar a tratar el tema de las vistas. Las vistas no son más que los archivos PHP desde donde tenemos que realizar la salida de la aplicación. Es un concepto que esperamos que ya se tenga en la cabeza cuando estamos introduciéndonos en Laravel, puesto que no pertenece en sí al framework PHP sino al MVC en general.

En este artículo veremos cómo crear nuestras primeras vistas en Laravel 5 y como invocarlas para crear una salida de la aplicación web totalmente personalizada.



## Qué son las vistas

Como decíamos, la mayoría seguro entenderá este concepto, no obstante, vamos a exponerlo rápidamente para aquel que no sepa de qué estamos hablando. Las vistas son una de las capas que tiene el sistema MVC, que trata de la separación del código según sus responsabilidades. En este caso, las vistas mantienen el código de lo que sería la capa de presentación.

Como capa de presentación, las vistas se encargan de realizar la salida de la aplicación que generalmente en el caso de PHP será código HTML. Por tanto, una vista será un archivo PHP que contendrá mayoritariamente código HTML, que se enviará al navegador para que éste renderice la salida para el usuario.

**Nota:** En la práctica una vista podrá tener cualquier tipo de salida, no solo HTML. Hay ocasiones que será código PHP para generar una imagen, un archivo de texto o cualquier otra necesidad. Por ejemplo, si ante una solicitud el servidor debe enviar como respuesta datos en notación JSON, esos datos se escribirán mediante una vista.

## Dónde almacenar las vistas en Laravel 5

Existe una carpeta en el proyecto que es donde debemos colocar las vistas en Laravel. Está en "resources/views". Si navegas a esa carpeta observarás que dentro ya hay diversos archivos, incluso directorios. Esto es porque las vistas se pueden organizar por carpetas, para mantener agrupadas las de cada una de las secciones de la aplicación. Nosotros podemos hacer lo mismo, o dejarlas sueltas en el directorio view.

Además verás que las vistas tiene una extensión ".blade.php". Esto hace referencia a que es un archivo de salida que usa el motor de plantillas "Blade", el oficial de Laravel.

Ten en cuenta que, a pesar de la extensión ".blade.php", las vistas no dejan de ser archivos PHP donde podríamos colocar HTML plano mezclado con códigos PHP. De hecho, podríamos perfectamente nombrar a nuestras vistas como ".php" y el tratamiento que haremos para invocarlas será exactamente el mismo que si tienen la extensión ".blade.php". Es más, usar Blade es opcional, por lo que para mantener la simplicidad de nuestras primeras vistas no vamos a usar el motor de plantillas.

Así pues, para construir nuestra primera vista lo tenemos fácil. Simplemente creamos un archivo llamado "algo.php" en la carpeta "resources/views/". Dentro le colocamos cualquier documento HTML, con cualquier contenido. Con eso ya tenemos la vista lista para ser utilizada.

Como decíamos, el código de ese archivo "algo.php" de momento es indiferente, pero por si alguien necesita la aclaración sería algo como esto:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Esto es una vista de prueba</title>
</head>
<body>
  <h1>Vista "algo"</h1>
  <p>Esta es mi primera vista en Laravel</p>
</body>
</html>
```

Como has observado, la vista no contiene ningún código PHP, porque de momento no lo necesitamos para probar. Es un HTML plano, pero no obstante, sí debemos respetar la extensión ".php" o ".blade.php".

## Cómo invocar una vista en Laravel 5

Ahora toca usar la vista creada en el paso anterior. Lo vamos a hacer desde el propio sistema de rutas, por simplificar la vida en este primer ejemplo. De hecho es como se invoca a la vista "welcome" en la ruta predeterminada que podremos ver al instalar el framework.

**Nota:** Aunque en una arquitectura MVC muchos desarrolladores prefieren que sean los controladores quienes invoquen a las vistas, nosotros nos vamos a saltar de momento ese paso porque realmente Laravel no lo necesita, pero sobre todo porque todavía no hemos llegado a explicar los controladores. En lugar de cargar la vista desde el controlador la vamos a invocar directamente desde el sistema de routing.

La corrección de esta operación sería discutible. No es lo habitual pero en ocasiones puede ser adecuada, por ejemplo que sea una vista que no necesita datos externos para mostrarse a si misma. Pero esto es otra discusión diferente al objetivo de este artículo.

En el siguiente código realizamos la invocación a una vista. Debes apreciar la función `view()` dentro del closure, a la que enviamos el nombre de la vista a mostrar.

```
Route::get('algo', function () {  
    return view('algo');  
});
```

Como has podido ver, `view()` es una función global, un helper global en Laravel, que se encarga de cargar una vista y devolver la salida producida por ella.

El nombre de la vista que estamos cargando es "algo". Para que no nos de un error la vista deberá estar creada dentro de la carpeta "resources/views", con esta ruta completa:

```
resources/views/algo.php
```

O bien:

```
resources/views/algo.blade.php
```

**Nota:** Si en algún momento queremos preguntar si una vista existe antes de cargarla, podemos usar la función `view()` de esta manera:

```
view()->exists('calendario.mes')
```

Si no se le envían parámetros a `view()` se recibe un objeto que tiene una serie de métodos útiles para vistas. El método `exists()` devuelve un `true` o `false`, dependiendo de si existe o no una vista indicada.

## Organizar las vistas por carpetas

Es común que queramos poner todas las vistas que tengan que ver con la misma sección del sitio en una misma carpeta, o todas las vistas de los correos electrónicos enviados por la aplicación, por ejemplo.

Las carpetas las situaremos dentro de "resources/views" y en ellas colocaremos los archivos php de las vistas, como se ha descrito antes.

Por ejemplo crea una vista llamada index.php y colócala en otro directorio dentro de views. Su ruta sería algo como esto:

```
resources/views/otro/index.php
```

Ahora invocaremos esa vista indicando la ruta donde se encuentra, desde el directorio views. Omitimos de nuevo la extensión, ya sea ".php" o ".blade.php".

```
Route::get('/otro', function () {  
    return view('otro/index');  
});
```

Como alternativa podemos especificar la ruta de la vista con un punto en lugar de una barra.

```
Route::get('/otro', function () {  
    return view('otro.index');  
});
```

## Pasar datos a las vistas en Laravel

Seguro que lo siguiente que te preguntabas era cómo pasar los datos a las vistas. Si tienes que pasar datos para que las vistas los representen, los enviarás a través de la función global view() como un array asociativo.

```
view('calendario.eventos', [  
    'mes' => $mes,  
    'ano' => $ano,  
    'eventos' => $eventos  
]);
```

Una vez dentro de la vista los recoges en el ámbito global, a partir de las llaves de los elementos del array de datos. En el ejemplo anterior \$mes, \$ano o \$eventos.

```
<p>  
    Estás viendo el mes <?= $mes; ?> y el año <?= $ano; ?>.  
</p>
```

**Nota:** Hemos hablado de las plantillas Blade, pero advirtiéndote que las vamos a ver en detalle más adelante. Sin embargo es útil que mencionemos una estructura de sintaxis de este tipo de plantillas que verás sin duda en varios ejemplos por ahí y que nosotros también pensamos usar en el futuro, que es el volcado de datos que tienes en variables en el contenido de la vista.

El mismo código que acabamos de ver en el párrafo anterior a esta nota, en el que mostramos el valor del mes y el año, podríamos haberlo escrito así con la sintaxis de Blade:

```
<p>
    Estás viendo el mes {{ $mes }} y el año {{ $ano }}.
</p>
```

Como se puede apreciar, simplemente sustituyes los tag de cierre y de apertura de código PHP y el "echo" por unas dobles llaves que engloban aquello que quieres volcar a la vista. Es una sintaxis que mejora la legibilidad del código.

Para usar esa sintaxis recuerda que debes tener una plantilla Blade, por lo que el archivo lo tendrás que nombrar necesariamente con extensión ".blade.php".

De momento es todo. Con esto ya conoces lo básico de las vistas en Laravel 5. No obstante quedan muchas cosas que aprender que veremos en los siguientes artículos de este manual.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 02/09/2015  
Disponible online en <http://desarrolloweb.com/articulos/introduccion-vistas-laravel5.html>

## Controladores en Laravel 5

### Explicaciones y ejemplos sobre controladores en Laravel 5. Crear controladores, invocarlos desde las rutas.

Aunque en el [Manual de Laravel 5](#) ya nos hemos referido anteriormente a los controladores solo fue muy de pasada. En este artículo comenzaremos a trabajar con ellos, conociéndolos un poco más a fondo.

Aunque suponemos que todos los lectores deben tener unas nociones básicas generales sobre el concepto de controlador, cabe aclarar que éstos son una de las piezas que, junto con los modelos y las vistas, forman parte del patrón MVC. En Laravel, como en cualquier otro de los frameworks PHP populares, son una importante parte de las aplicaciones. Su función es la de definir el código a ejecutar como comportamiento frente a una acción solicitada dentro de la aplicación.



Generalmente para poder desempeñar su labor se apoyan en los modelos y las vistas. El controlador sabe qué métodos del modelo debe invocar, ya sea para actualizar cierta información o para obtener ciertos datos, así como las vistas que deben presentar la información como respuesta al usuario, después de la realización de las acciones necesarias.

**Nota:** Para más información sobre MVC te referimos al artículo [Qué es MVC](#).

## Controladores en Laravel

Los controladores están localizados en la carpeta `app/Http/Controllers` y podemos organizarlos en subcarpetas si lo deseamos. Como otras clases de Laravel están dentro del sistema de autocarga de clases, por lo que estarán disponibles siempre que los necesitemos en la aplicación.

A continuación puedes encontrar el código de un controlador básico. Merece la pena pararse a analizarlo brevemente para comentar algunos detalles.

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class ArticulosController extends Controller
{
    public function ver($id)
    {
        return view('articulos.ver', ['id' => $id]);
    }
}
```

Las dos primeras líneas son concernientes a los [espacios de nombres](#) donde estamos trabajando. Luego encontramos la propia clase que define el controlador, en este caso `ArticulosController`. Como cualquier clase, por convención, usamos la primera letra del nombre en mayúscula y además en Laravel tenemos por costumbre apellidar a las clases con el sufijo "Controller". También es una convención que te ayudará a ti a identificar el código de los controladores y a otras personas que puedan trabajar en el proyecto.

Dentro de la clase colocamos las acciones que deseemos. Cada acción la implementamos a partir de un método, al que podemos invocar desde el sistema de routing, como se verá a continuación.

**Nota:** Dentro del código de la acción de momento estamos colocando una llamada a una vista, que supuestamente mostraría un artículo. Generalmente los controladores acceden primero a los modelos para recuperar la información que se les debe pasar a las vistas para que representen. Aquí le estamos pasando como dato a la vista simplemente el id del artículo, lo que no resultaría de mucha utilidad, pero más adelante aprenderemos a acceder a la base de datos para recuperar esa información y poderla pasar completa hacia la vista.

Otro detalle que se puede observar es que los controladores son clases, de programación orientada a objetos, y como tales podríamos incluir en ellas cualquier miembro/s que consideremos oportuno para el funcionamiento interno, como métodos privados, propiedades, etc.

## Invocar un controlador desde el sistema de rutas

A los controladores los vamos a invocar normalmente desde el sistema de rutas, indicando el nombre del controlador y la acción (método) que debe ejecutarse para procesar una solicitud. En la siguiente línea de código registramos una ruta que llamaría a la acción "ver" sobre el controlador "ArticulosController".

```
Route::get('articulos/{id}', 'ArticulosController@ver');
```

Las acciones pueden recibir parámetros, tal como se explicó cuando [aprendimos a crear nuestras rutas en Laravel](#). En esa ruta estaríamos definiendo que se debe recibir el parámetro {id}. El valor de ese parámetro será pasado al método o acción ver() de "ArticulosController". Obviamente, podemos pasar tantos parámetros como sea necesario y los recibiremos en la acción del controlador en el mismo orden como fueron definidos en el patrón de la URI registrada.

## Generar los controladores con automáticamente con artisan

Crear desde cero un controlador es una tarea repetitiva dentro de Laravel, por lo que existen atajos. El ya conocido comando "artisan" nos ofrece una utilidad para crear una nueva clase controlador de una manera automática. Para ejecutarlo lanzamos en la consola este comando.

```
php artisan make:controller CategoriasController
```

Dentro de la carpeta del proyecto, invocamos a artisan. La operación solicitada es make:controller y luego le indicamos el nombre del controlador a crear, en este caso "CategoriasController".

**Nota:** Si en cualquier momento queremos ver la lista de utilidades que nos facilita artisan, escribimos este comando en la consola:

```
php artisan
```

Pero ojo, para que funcione artisan tienes que estar en la home del proyecto, donde verás que se encuentra un archivo llamado "artisan".

Ahora verás, en la carpeta de los controladores ("app/Http/Controllers"), que ha aparecido el nuevo archivo del controlador, generado automáticamente. Te darás cuenta que tiene ya una serie de acciones definidas que son las típicas cuando quieres crear un recurso "RESTful". De momento no las usaremos pero más adelante verás que es un atajo potente, ya que te dan declarados todos los métodos que necesitarías para realizar las operaciones típicas con un recurso en un API Rest.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en 18/08/2015  
Disponible online en <http://desarrolloweb.com/articulos/controladores-laravel5-ejemplos.html>

## HTTP Request en Laravel 5

Laravel nos facilita todos los datos de la solicitud actual a través HTTP Request, un objeto sobre el que podremos consultar información sobre el cliente que realiza la solicitud y datos que pueda estar enviando.

Toda aplicación web recibe solicitudes para completar todo tipo de acciones. Cada solicitud que recibe el servidor viene acompañada de una serie de datos, que se envían en el protocolo HTTP. Entre la información que recibe PHP podemos encontrar desde el user-agent del visitante o su IP, hasta datos que viajan en las cabeceras ante una operación post.

Esos datos en Laravel se acceden a través del objeto Request, que podemos recibir en el controlador mediante la inyección de dependencias, o mediante la correspondiente facade. En este artículo del [Manual de Laravel](#) te vamos a explicar las bases del trabajo con el objeto Request.



### Recibir la solicitud (request)

Comencemos observando cómo se consigue el objeto Request en un controlador, para lo que vamos a hacer mención a un patrón de diseño de programación orientada a objetos, usado en Laravel así como en otra serie de frameworks populares: [Inyección de dependencias](#). En este patrón se busca separar la responsabilidad de creación de los objetos de su uso, simplificando y abstrayéndonos de toda la complejidad que puede suponer crear todas y cada una de las dependencias que tenga un código. Cuando una clase

necesite de un objeto para completar sus operaciones no lo construye él, sino que lo recibe en el constructor o en los métodos que lo necesiten.

El hecho de recibir los objetos de los que depende una clase por parámetro es lo que se conoce como inyección. La dependencia es aquello que necesita, que no es más que un objeto de una clase. Además el patrón incluye lo que se llama el contenedor de dependencias o contenedor de servicios, que es la pieza de software encargado de instanciar los objetos que se deben inyectar e inyectarlos en los métodos necesarios. Como este patrón ya lo hemos explicado anteriormente en DesarrolloWeb.com nos vamos a ahorrar volver otra vez sobre lo mismo, remitiendo a los interesados al artículo enlazado en el párrafo anterior.

La clase Request está en el namespace "Illuminate\Http\Request", por lo que un primer paso sería declarar que vamos a usarlo.

```
use Illuminate\Http\Request;
```

A continuación podemos definir, en la cabecera del método donde queramos recibir el objeto request, la correspondiente dependencia, de la clase Illuminate\Http\Request.

```
public function recibirPost(Request $request){  
    // en este punto del código $request es mi objeto HTTP Request. (inyectado)  
}
```

Como puedes ver, para que el inyector de dependencias sepa qué es lo que debe inyectar, estamos definiendo la clase del parámetro \$request que no es otra que la clase Request.

**Nota:** La posibilidad de escribir el tipo de datos de los parámetros de métodos o funciones es algo relativamente nuevo en PHP y toma el nombre de Type Hinting, "Implicación de tipos" en español.

Al informar el tipo observarás que no hemos colocado toda la ruta en la jerarquía del namespace "Illuminate\Http\Request", el motivo es simplemente porque ya habíamos definido un alias de esa clase mediante la sentencia "use Illuminate\Http\Request".

**Nota:** Si no hiciéramos el alias del namespace con la sentencia use Illuminate\Http\Request, entonces estaríamos obligados a definir la jerarquía de espacios de nombres donde está situada la clase Request, con un código como el que puedes ver a continuación.

```
public function recibirPost(\Illuminate\Http\Request $request){  
    // Esto es equivalente y funcionará aunque no hagas el "use"  
}
```

Suponemos que estas nociones de espacios de nombres las tendrás claras, si no es así te recomendamos la lectura de los artículos de DesarrolloWeb.com sobre [Namespaces en PHP](#).

En otros artículos también veremos cómo trabajar con el HTTP Request sin necesidad de la inyección de dependencias, directamente a través de la "facade" (fachada) Request.

## Usar el HTTP Request

Una vez ya disponemos del objeto request podemos consultar toda la información disponible acerca de la solicitud. Es tan sencillo como enviar mensajes, invocando sus métodos. En este artículo haremos un ejercicio básico de envío por post, pero antes vamos a probar un par de métodos sencillos.

1. Para recuperar la URI actual de una solicitud (lo que va después del dominio), sobre nuestro objeto request invocaremos el método `path()`.
2. Para recuperar la URL completa de una solicitud (toda la ruta completa, incluyendo el dominio), invocamos el método `url()`.

Así quedaría un método de un controlador que mostrase ambas informaciones de la solicitud.

```
public function mostrarUriUrl(Request $request){
    echo $request->path();
    echo "<br>";
    echo $request->url();
}
```

Ahora vamos a ver el procedimiento completo, desde la creación de la ruta hasta la codificación del controlador, para recuperar un dato que enviarían por post en una solicitud, recogiendo los valores enviados usando HTTP Request.

En primer paso, debo de registrar una ruta en mi sistema, archivo `routes.php`.

```
Route::post('recibir', 'PrimerController@recibirPost');
```

Segundo paso defino el método `recibirPost`, que debemos de crear dentro de `PrimerController` (es un controlador que creamos en un ejemplo anterior de este manual de Laravel). En ese método inyecto la dependencia y la recibo con `$request`.

```
public function recibirPost(Request $request){
    // código de mi método
}
```

Como tercer paso, ya en el código de implementación del método, puedo acceder a uno de los campos que me envíen por post, de manera independiente.

```
public function recibirPost(Request $request){
    echo $request->input('id');
}
```

En el caso anterior estamos accediendo a un campo enviado por formulario llamado "id". Pero también existen métodos para recuperar de una sola vez todos los datos enviados en la solicitud.

```
public function recibirPost(Request $request){
    $todos_los_datos = $request->all();
}
```

En este caso recuperas todos los datos en forma de array, por lo que si deseas ver su contenido tendrás que usar una función como `print_r()` o `var_dump()`, o quizás mejor `dd()` que es una función específica de Laravel que muestra el contenido de una variable y a continuación para la ejecución del script.

```
public function recibirPost(Request $request){
    dd($request->all());
}
```

**Nota:** Para poner en marcha este ejemplo de una manera cómoda tendrás que usar una extensión como postman, que te permite generar formularios al vuelo. Algo que se explicó en el artículo Verbos en las rutas de Laravel. Otra cosa que tendrás que hacer será comentar la línea donde se accede al middleware de protección csrf, también explicado en el mencionado artículo.

## Recibir parámetros de la ruta

Aunque estemos inyectando en el controlador el objeto de la solicitud, clase HTTP Request, no impide recibir otros parámetros en el método de la acción. Esto lo podemos conseguir simplemente creando los parámetros en la acción como siempre.

Nuestra ruta sería algo así:

```
Route::post('editar/{id}', 'PrimerController@editar');
```

Ahora podremos recibir tanto la request como el parámetro de la ruta `{id}` en el método de la acción del controlador.

```
public function editar(Request $request, $id){
    echo "Recibo $id como parámetro de la ruta.";
    echo "Además recibimos estos datos por formulario: " . implode(', ', $request->all());
}
```

## Conclusión

Con lo que hemos visto tenemos material suficiente para poder continuar viendo otras utilidades del framework Laravel. Por supuesto, más adelante volveremos sobre el tema de las solicitudes, pues es muy

importante.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 13/08/2015  
Disponible online en <http://desarrolloweb.com/articulos/http-request-laravel5.html>

## Introducción a modelos en Laravel

### Introducción a los modelos, parte del patrón MVC, en el framework PHP Laravel 5.

Siguiendo con una introducción básica a los componentes principales de Laravel, queremos hacer una primera aproximación a los modelos, de los que no habíamos hablado todavía en el [Manual de Laravel 5](#). Pero antes que nada, conviene hacer una aclaración conceptual sobre qué es un modelo:

Los modelos son uno de los componentes principales de las aplicaciones desarrolladas bajo el patrón MVC, que tienen la responsabilidad de acceder a los datos, modificarlos, etc. En el patrón además los modelos mantienen lo que se llama la lógica de negocio, que son las reglas que deben cumplirse para trabajar con los datos.

Por tanto, el tipo de acciones que le vamos a solicitar a un modelo es por ejemplo, obtener datos, insertarlos, modificarlos, etc. En las operaciones que modifiquen los datos además se tendrá que realizar cierta validación de esos datos, para asegurarnos que tienen la forma que es necesaria antes de guardarlos.



Cuando pensamos en modelos muchos hacemos una conexión directa con la base de datos: "un modelo guarda el código de acceso a la base de datos". Pero no es exactamente así, ya que un modelo trabaja con datos que pueden venir de varias fuentes. Generalmente será la base de datos, pero podría ser un API, Servicio web, sistema de archivos, etc.

### Modelos en Laravel 5

En Laravel 5 los modelos se gestionan en la carpeta "app", colocando los archivos de nuestro modelo sueltos ahí. En la instalación limpia de Laravel 5.1 tenemos un primer modelo que podemos abrir para echar un primer vistazo rápido sobre ellos. Es el archivo que está en la ruta "app/User.php".

**Nota:** Una de las modificaciones principales que aparecieron en la versión 5 de Laravel es que han quitado la carpeta de los modelos. Esto es porque te animan a que uses la estructura de carpetas que

preferias para los modelos. En principio pueden ir todos colgando de la carpeta "app", pero también podría crearse una carpeta "models" para situarlos allí, o crear cualquier otra estructura si lo ves conveniente.

Laravel además separa código que en el patrón MVC se ubica en la responsabilidad del modelo en diversas clases dispersas por diversos directorios. Por ejemplo, las validaciones y filtrado de los datos que se reciben se pueden colocar en el middleware o en los archivos de Requests para validación que no hemos visto todavía. Así que, los que conocemos otras arquitecturas de aplicaciones basadas en MVC debemos de abrir un poco la mente cuando entramos en Laravel.

Como puedes ver, los modelos tienen la primera letra en mayúscula, por implementarse mediante clases. Los archivos donde guardamos el código de los modelos también deben tener esa primera letra en mayúscula.

En Laravel los modelos se controlan por un ORM llamado Eloquent, al menos los modelos que están implementados como datos en una base de datos, pero no es un requisito, de modo que podríamos trabajar con otros ORM o incluso bajar a un nivel más bajo y trabajar con PDO directamente, o con las extensiones de nuestra base de datos en particular.

En el caso de ser un modelo Eloquent, los modelos están directamente asociados a una entidad y a su vez a una tabla de la base de datos, por lo que un modelo que se llama User está directamente relacionado con una tabla llamada con el mismo nombre en la base de datos, pero en minúscula y acabado en plural, ej "users".

Como siempre, te recomendamos comenzar por abrir el mencionado archivo con el modelo User.php para ver cómo se implementan en Laravel. Ese modelo está bien pero tiene un par de modificaciones un poco avanzadas que nos pueden despistar, así que preferimos explicarte los modelos con respecto a un ejemplo más vacío.

## Crear un modelo vacío en Laravel

Como en otras ocasiones, podemos ayudarnos de artisan para crear un modelo de partida. Con el comando `make:model`, seguido del nombre del nuevo modelo, creamos un modelo vacío.

```
php artisan make:model Article
```

Eso nos crea en el directorio "app" el correspondiente modelo de Eloquent, que contiene un código como el que puedes ver a continuación.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
}
```

Eso es todo lo que necesita un modelo básico en Laravel. Como te puedes imaginar, la explicación de su sencillez es que esta clase extiende la clase Model de Laravel.

Estos modelos ya vienen con funcionalidades para solicitar datos que estén en la base de datos. Los modelos de Eloquent estarán asociados directamente con una tabla llamada "articles", sin que tengamos que configurar nada en nuestro código, aunque más adelante aprenderemos a cambiar el nombre de la tabla asociada a un modelo, por si nos resultase necesario.

Este modelo usará el motor del ORM de Eloquent y lo puedes ver porque está haciendo uso de la clase Model que está en el namespace Illuminate\Database\Eloquent. Esa clase se le asigna un alias llamado "Model" (el mismo nombre de la clase que luego hacemos el extends) gracias a la sentencia:

```
use Illuminate\Database\Eloquent\Model;
```

**Nota:** Si nuestro modelo trabajase con otro ORM, o con otra base de datos que no soporte Eloquent como MondoDB, no usaríamos esa clase Model para extenderlo, sino otra, y por tanto el trabajo sería diferente al que realizamos con Eloquent.

Fíjate también que el modelo se crea dentro del namespace "App", definido por la primera línea de código:

```
namespace App;
```

## Acceder a datos del modelo

Desde los controladores queremos acceder a datos que mantienen los modelos: consultas, modificaciones, etc. Esas operaciones se hacen a través de la clase del modelo que acabamos de implementar.

De momento veamos cómo implementar una selección de todos los datos que tenemos en el modelo, invocando el método all() sobre el modelo que acabamos de crear.

```
\App\Article::all()
```

Este código estaría en un controlador, o en otra clase desde la que queramos acceder a los datos del modelo. Como puedes ver, para referirnos al modelo debemos indicar el espacio de nombres donde lo podemos encontrar, que en nuestro caso era "App".

Esa línea de código, como decíamos, nos devuelve una colección con los datos encontrados. Aunque de momento todavía no nos va a funcionar, porque la tabla "articles" no está creada en nuestro sistema gestor. En cambio obtendremos un error como este: "[...] Base table or view not found: 1146 Table 'proyecto.articles' [...]".

En futuros artículos veremos cómo crear nuestras tablas, con el sistema de migraciones y podremos comenzar a usar más a fondo los modelos. Pero como seguro estamos impacientes por comprobar si esa instrucción verdaderamente funciona, vamos a adelantar alguna cosa.

## Crear una tabla manualmente de MySQL

Podemos crear manualmente la tabla que estamos necesitando en la base de datos. Como decimos, no sería el modo correcto de proceder pero de momento con lo que sabemos vamos a conformarnos. Usaremos nuestro cliente MySQL de preferencia, como MySQL Workbench, Sequel Pro o incluso podríamos instalar PhpMyAdmin. Nosotros no vamos a usar ninguna de esas posibilidades, sino que vamos a [conectar MySQL por línea de comandos](#), que así no hay manera de fallar.

A continuación realizaremos una pequeña recetilla para crear esa tabla, que nos servirá para explicar el proceso.

1. Primero arrancamos la máquina virtual, si no estaba ya: (desde el directorio de homestead en tu disco local)

```
vagrant up
```

2. Conectas por SSH con la máquina Homestead.

```
vagrant ssh
```

3. Conectas con MySQL por línea de comandos. El host es "localhost", el usuario es "homestead" y la clave es "secret".

```
mysql -h localhost -u homestead -p
```

**Nota:** Puedes mirar el usuario y contraseña de la base de datos en el archivo .env que está en la raíz del proyecto. Otras configuraciones de bases de datos, como el sistema gestor de base de datos usado se indican en archivo config/database.php. Por defecto Laravel en Homestead viene configurado para usar MySQL, pero hay otros motores de base de datos que se encuentran instalados en la máquina virtual.

4. Ya dentro del cliente MySQL por línea de comandos lanzas el comando para usar la base de datos que tengas creada.

```
use homestead;
```

5. Ahora creas la tabla y los datos de prueba.

```
CREATE TABLE `articles` (  
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,  
  `name` varchar(200) COLLATE utf8_unicode_ci DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;  
  
INSERT INTO `articles` (`id`, `name`)  
VALUES  
  (1, 'Probando'),  
  (2, 'Algo'),  
  (3, 'Lindo');
```

Teóricamente ahora ya podrás acceder a la página de antes, donde habías puesto en el controlador la instrucción para mostrar todos los artículos. Solo recuerda que para mostrar la salida por la página y así poder leerla debes hacer un `print_r()` o `var_dump()` porque es una colección. También puedes usar la función `dd()` que te ofrece Laravel.

```
dd(\App\Article::all());
```

**Nota:** Realmente no necesitas ni usar un controlador, podrías hacerlo directamente con una closure dentro del sistema de rutas.

```
Route::get('articulos', function(){  
    dd(\App\Article::all());  
});
```

## Conclusión

Insistimos en que más adelante vamos a conocer mecanismos por los que se crean las tablas o se insertan datos de prueba directamente desde Laravel, cuando hablemos de "migrations y seeders". Aunque para trabajar en Laravel podríamos tener el schema de la base de datos hecho a mano directamente con SQL en el gestor de base de datos que estemos usando, no es la manera más habitual de proceder.

Además, hay otros métodos de acceder al sistema gestor de base de datos, como ya hemos advertido, con programas profesionales como MySQL Workbench que dan muchas mejores prestaciones y aumentan la productividad, en comparación con trabajar directamente por el terminal.

De momento creemos que es suficiente para cumplir con lo que sabemos nuestro objetivo de poner en marcha esa llamada al modelo y recuperar información que hay en MySQL.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en *14/09/2015*  
Disponible online en <http://desarrolloweb.com/articulos/introduccion-modelos-laravel.html>

## Laravel middleware

Qué son los Http Middleware, una de las piezas principales del framework PHP Laravel. Cómo trabajar con Middlewares en Laravel, creando uno nuevo.

En el [Manual de Laravel](#) hemos podido conocer varias de las principales capas de aplicación. Hemos preferido comenzar por describir las partes más sencillas y de las cuales estamos seguros muchos tenían nociones, como son las vistas o controladores. Del MVC nos faltan por ver los modelos, pero antes de ponernos con ellos nos vamos a detener en los middleware.

Hasta ahora hemos visto que desde el sistema de routing podemos invocar a los controladores, o incluso también a través de un "closure" a las vistas. Sin embargo no habíamos mencionado que, antes llegar el flujo de ejecución a éstos hay una capa intermedia que se ejecuta en toda solicitud: el middleware.

**Los middleware en términos generales son partes del software que actúan de mediadores** entre distintos actores, permanecen en medio para facilitar la interacción o comunicación entre distintas partes de un sistema. Aquí en Laravel se quedan entre medias del sistema de routing y los controladores, permitiendo hacer cosas al pasar de unos a otros, generalmente operaciones de filtrado.



Según la definición de la documentación oficial de Laravel, "HTTP middleware provee un mecanismo adecuado para filtrar solicitudes HTTP entrantes a la aplicación [...] hay diversos middleware incluidos en el framework Laravel, como middleware para mantenimiento, autenticación, protección CSRF mediante token, etc."

Además nosotros podemos hacer nuestros propios middleware para diversas tareas, como añadir una salida en las cabeceras de toda respuesta, realizar un log de todas las solicitudes al servidor, etc. Como puedes ver, el middleware es el sitio ideal para incluir código que quieres ejecutar al principio de toda solicitud, aunque si quieres también los puedes asociar solamente a determinadas rutas.

### Archivo Kernel.php, donde se registran los middleware

Antes de ponernos a realizar nuestro propio middleware es una buena idea echar un vistazo a los que ya tenemos funcionando de manera predeterminada. Estamos seguros que ésto ayudará a aclarar el concepto de middleware en Laravel. Para ver los middleware configurados en nuestro sistema tenemos que entrar en el archivo `app/Http/Kernel.php`.

El Kernel es el que le dice a Laravel qué middlewares tiene que cargar. Allí encontrarás una clase que tiene

registrados los middleware que se van a ejecutar en el sistema, tanto de manera global (propiedad `$middleware`) como para rutas particulares (propiedad `$routeMiddleware`).

**Nota:** Quizás recuerdas que en este archivo `Kernel.php` ya habíamos entrado anteriormente, cuando descubrimos el sistema de rutas. En el artículo [Verbos en las rutas de Laravel](#) vimos que al realizarse una ruta de tipo `post` se activa una comprobación de un token, solicitado para aumentar la seguridad frente a CSRF (Cross-site request forgery o falsificación de petición desde otros sitios). Esa comprobación se realiza en el middleware `"VerifyCsrfToken"`. En el mencionado artículo habíamos pedido comentar esa línea, para que no se pusiera en marcha ese middleware y poder comprobar si funcionaban las rutas `post`.

## Crear un middleware

Para crear un middleware podríamos tomar como punto de partida uno de los que ya vienen por defecto en Laravel, pero realmente hay una manera más apropiada, usando el asistente `"artisan"`.

Desde la línea de comandos, en la carpeta raíz del proyecto, podemos invocar `artisan` y solicitarle el comando `"make:middleware"` indicando a continuación el nombre del middleware que se desea crear.

```
php artisan make:middleware DomingoMiddleware
```

Nuestro middleware se llama `DomingoMiddleware` y se encargará de hacer cosas cuando detecte que el día actual es un domingo. Ese comando de `artisan` genera un middleware básico en el que solo tenemos un método, que enseguida explicamos.

La localización del archivo que se ha creado, y en general de todos los middlewares en Laravel, está en `app/Http/Middleware/`. Allí encontrarás ahora el archivo `DomingoMiddleware` que como también corresponde con el nombre de la clase que implementa el middleware, tiene la primera letra en mayúscula.

Este es el código de nuestro primer middleware:

```
<?php

namespace App\Http\Middleware;

use Closure;

class DomingoMiddleware
{
    public function handle($request, Closure $next)
    {
        if(date('w')=='0'){
            echo "Es domingo!";
        }else{
            echo "No es domingo";
        }
        return $next($request);
    }
}
```

```
}  
}
```

Sobre el código anterior queremos que te fijes en el método `handle()`, que es el que se ejecutará cuando se ponga en marcha este middleware. Tiene unas cuantas cosas que conviene explicar detenidamente, pero de momento solo queremos que se aprecie el `if`, donde se comprueba si el día de la semana actual es domingo, realizando dos acciones distintas si era o no era ese día de la semana.

**Nota:** Como acción de respuesta dependiendo de si es o no domingo realizamos una salida con un `echo`. Queremos remarcar que ese `echo` no tiene mucho sentido, porque no es el momento adecuado de lanzar salida al navegador, pero ahora mismo nos sirve en este middleware de prueba, para verlo en funcionamiento y recibir alguna salida como respuesta. El único motivo por tanto de realizar ese `echo` es para saber si se está ejecutando. Como ya sabes, la salida la generamos desde las vistas. En el middleware acciones típicas son redirigir al usuario a una página o realizar cualquier tarea de mantenimiento, etc.

## Registrar el middleware de manera global

Ahora vamos a registrar el middleware para ver cómo se ejecuta en el sistema. Empezaremos registrando en modo global, para que se ejecute en todas y cada una de las solicitudes que atienda nuestra aplicación.

Esto es muy sencillo y ya lo hemos dejado entrever al principio del artículo. Se hace desde el archivo `Kernel.php` que está en la ruta `app/Http/Kernel.php`.

Hay una propiedad de la clase `Kernel` donde se coloca toda la lista de middleware a ejecutar de manera global;

```
protected $middleware = [  
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,  
    // otros middleware  
    // ...  
    \App\Http\Middleware\DomingoMiddleware::class,  
];
```

Como puedes ver, es un array en el que debemos indicar el listado de los middleware a ejecutar en cada solicitud, en el orden en el que van a ser invocados. Nosotros hemos agregado el middleware creado anteriormente en el último elemento de la lista, pero si necesitase mayor prioridad sería solo ponerlo antes en el array.

Una vez registrado el middleware de manera global podrías entrar en cualquier página de la aplicación y debería verse la salida del middleware, informando si es o no domingo.

## Registrar el middleware para una ruta determinada o un controlador

Hay varias maneras de hacer este paso, desde el sistema de rutas o incluso desde los controladores, pero siempre debemos comenzar por asignar un nombre a nuestro middleware en el archivo `Kernel.php`. Allí, en la clase `Kernel`, hay una segunda propiedad llamada `$routeMiddleware`, donde tenemos un array asociativo

con los middlewares que deseemos usar en las rutas.

Cada elemento del array tiene un índice y ese índice será el nombre que le demos al middleware para referirnos a él desde el sistema de routing.

```
protected $routeMiddleware = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    // otros middleware de rutas...  
    'domingo' => \App\Http\Middleware\DomingoMiddleware::class,  
];
```

Ahora podemos ejecutar el middleware a través del índice que le hemos dado en el array anterior. Lo veremos hacer mediante tres alternativas distintas:

1. Desde el sistema de routing, al definir la ruta, podemos indicar un middleware que queremos usar. Esto lo tenemos que hacer desde el registro de rutas, realizado en el archivo routes.php, mediante una sintaxis como la que puedes ver a continuación.

```
Route::get('/test', ['middleware' => 'domingo', function(){  
    return 'Probando ruta con middleware';  
}]);
```

2. También desde el sistema de routing podemos generar un grupo de rutas donde se ejecute este middleware.

```
Route::group(['middleware' => 'domingo'], function(){  
    Route::get('/probando/ruta', function(){  
        //código a ejecutar cuando se produzca esa ruta y el verbo  
        return 'get';  
    });  
  
    Route::post('/probando/ruta', function(){  
        //código a ejecutar cuando se produzca esa ruta y el verbo POST  
        return 'post';  
    });  
});
```

Así hemos indicado dos rutas donde se ejecutará ese middleware etiquetado como "domingo".

3. Desde un controlador también podemos llamar a un middleware. Lo podemos hacer desde el constructor, por lo que ese middleware afectará a todas las acciones dentro del controlador.

```
class PrimerController extends Controller  
{  
    public function __construct(){  
        $this->middleware('domingo');  
    }  
}
```

```
}
```

En este caso no necesitamos mencionar el middleware desde el sistema de rutas, solo lo mencionamos desde el constructor del controlador, para ejecutarse en cualquiera de las acciones declaradas en él.

## Encadenar un middleware con el siguiente

Según la filosofía de los middlewares en Laravel, como ya se comentó, pueden existir varios que se ejecuten en cadena, uno detrás de otro, para cada solicitud. Por ello tenemos que asegurarnos que este encadenamiento se pueda producir. Realmente este trabajo nos lo dan ya hecho en el middleware que se genera con artisan, por lo que no debemos preocuparnos nosotros. Sin embargo queremos que se vea dónde se consigue ese procesamiento en cadena.

Echemos un vistazo al método `handle()` del middleware generado automáticamente:

```
public function handle($request, Closure $next)
{
    return $next($request);
}
```

Como se puede ver, en este método se devuelve con `return` una llamada a `$next()` pasándole `$request` como parámetro. Esa llamada es la que permite que se ejecute el siguiente middleware en la lista de middlewares globales y además enviarle la Request completa al siguiente middleware para que siga filtrándola.

Realmente no tenemos que hacer nada manualmente, porque Laravel ya le pasa en el método `handle` el `$request` y el `$next` como parámetro. Además, Laravel sabe si ese middleware es el último de la lista, en cuyo caso no debería haber otro encadenamiento. Simplemente sería asegurarse de dejar ese `return` que aparece de manera predeterminada, para que todo siga funcionando.

## Conclusión

Los middlewares son una herramienta potente para realizar muchos tipos de acciones. Hemos visto las posibilidades más básicas que nos ofrece Laravel 5. Para los que vengan de versiones anteriores, cabe decir que vienen a sustituir el sistema de filtrado de Laravel 4 y realizando la operación habitual de comprobación de la solicitud mediante el objeto Request visto anteriormente.

Sin embargo, las utilidades del middleware se extienden para cualquier cosa que podamos necesitar. En futuros artículos nos detendremos en otras cosas y configuraciones que se pueden realizar con los middleware.

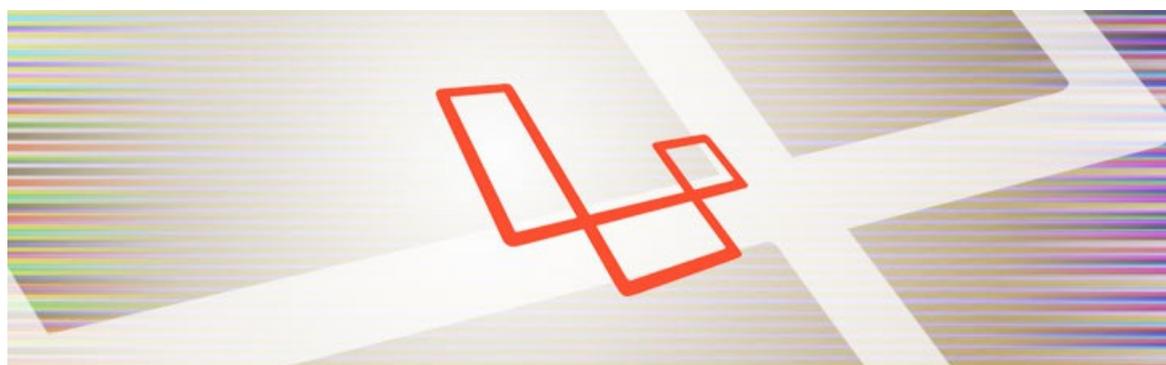
Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en *18/09/2015*  
Disponible online en <http://desarrolloweb.com/articulos/laravel-middleware.html>

## Responses en Laravel 5

Qué son las responses, uno de los elementos fundamentales de Laravel 5 y algunos ejemplos de uso.

Realmente el sistema de "Responses" es algo que abordamos en este momento de manera particular, pero que venimos usando a lo largo de diversos puntos de este manual de Laravel. Cuando devolvemos datos en forma de página web, o hacemos redirecciones por poner otro ejemplo, estamos usando el sistema response.

Las responses son las "respuestas" que nos debe devolver Laravel ante cualquier solicitud que se realice al servidor. Según la documentación oficial "Cualquier ruta [del sistema de routing] y controlador debe devolver algún tipo de respuesta al navegador del usuario". Existen diversos modos de devolver respuestas y en este artículo analizaremos algunos de ellos.



El sistema de Response depende de la clase `Illuminate\Http\Response` aunque muchas veces nos saltamos realmente el uso de esa clase y simplemente le dejamos a Laravel que la use internamente. Por ejemplo cuando escribimos salida desde el sistema de routing:

```
Route::get('ruta/de/ejemplo', function(){
    return "Respuesta desde sistema de routing.";
});
```

En este caso, como decimos, internamente Laravel recibirá esa cadena de respuesta la convertirá en un HTTP response para enviarla al cliente.

Cuando devolvemos una vista, ya sea desde un controlador o un closure en el sistema de routing también se pone en marcha el sistema de response de Laravel sin que el programador necesite usarlo directamente.

```
Route::get('cualquier/ruta', function () {
    return view('una_vista');
});
```

¿Entonces se puede abordar directamente el sistema de Response de Laravel y en ese caso, qué utilidad tiene? Efectivamente, nosotros podemos usar directamente una instancia de Response y ello nos permite personalizar todavía más elementos de la respuesta HTTP, enviando códigos de status particulares o

cabeceras de HTTP.

**Nota:** La clase Response de Laravel hereda directamente de la clase `Symfony\Component\HttpFoundation\Response` que ya provee una buena cantidad de métodos para construir respuestas HTTP. En este caso nos pueden interesar las documentaciones de [Response de Laravel](#) y por supuesto de [Response de Symfony](#).

## Ejemplo de respuesta usando una instancia de Response

A través del helper `response()` podemos crear fácilmente una instancia del objeto Response con el que podemos hacer varios ejemplos de uso de personalización de la respuesta.

En este primer ejemplo tenemos una respuesta exactamente igual a la que conseguiríamos con un simple `return 'Hola respuesta!'`.

```
Route::get('respuesta', function(){
    return response('Hola respuesta', 200);
});
```

El helper `response()`, como puedes ver, recibe dos parámetros:

1. Contenido de la respuesta
2. Código de status

El contenido es el texto que devuelve como respuesta la solicitud HTTP. El código de status son los típicos que debes de conocer del protocolo HTTP. 200 significa "todo correcto".

Después de la invocación al helper `response()` recibimos como valor de devolución un objeto de la clase Response. Éste objeto es el que devuelve el closure y que Laravel toma para componer la respuesta HTTP.

En este segundo ejemplo devolvemos un error de página no encontrada, status 404, con contenido 'Esto es un error'.

```
Route::get('respuesta2', function(){
    return response('Esto es un error', 404);
});
```

**Nota:** Los códigos de status los puedes ver en las herramientas para desarrolladores de tu navegador de preferencia. En Chrome por ejemplo los podrás apreciar en la sección "Network" de las developer tools, en la columna "Status".

The screenshot shows the Chrome DevTools Network tab. The 'Network' panel is active, displaying a list of requests. The first request is 'favicon.ico' with a status of 200 and type 'x-icon'. The second request is 'respuesta2' with a status of 404 and type 'document'. The 'Timeline' panel shows a red vertical line indicating the time of the error. The 'Filter' field is empty, and the 'View' and 'Options' menus are visible.

Name	Method	Status	Type
favicon.ico	GET	200	x-icon
respuesta2	GET	404	document

Si quieres conocer otros códigos de respuesta puedes consultar este artículo de la MDN: [Response codes](#).

En este ejemplo realizamos algo también simple pero más elaborado, ya que enviamos una respuesta a la que le agregamos una cabecera adicional para especificar que el contenido es una hoja de cálculo, archivo CSV.

```
Route::get('respuesta3', function(){
    return response("1,2,3,4\n5,6,7,8", 200)
        ->header('Content-Type', 'text/csv');
});
```

Como decíamos, el helper `response()` devuelve un objeto de la clase `Response` y sobre él estamos encadenando la invocación de su método `header()`, que sirve para agregar nuevas cabeceras a la respuesta HTTP. En este caso el nombre de la cabecera es `'Content-Type'` y el valor que estamos aplicando es `'text/csv'`.

En el ejemplo siguiente probamos otra cabecera diferente, para realizar una redirección. Tómalo simplemente como un test, puesto que existe otro método más rápido y sencillo para enviar respuestas de tipo `redirect`.

```
Route::get('respuesta4', function(){
    return response("", 301)
        ->header('location', 'http://desarrolloweb.com');
});
```

**Nota:** Existe un helper llamado `redirect()` que forma parte de los componentes del sistema de response de Laravel y está pensado para hacer ese trabajo, aportando diversos tipos de redirecciones posibles. Un ejemplo podría ser este:

```
return redirect('/uri/de/redireccion');
```

En esta ocasión lo usamos para redirigir a otra URI del mismo sitio web, pero veremos otros ejemplos en futuros artículos.

Podemos encadenar varias llamadas al método `header()` si quisiéramos agregar varias cabeceras diferentes a la respuesta. como vemos en el siguiente ejemplo:

```
Route::get('respuesta5', function(){
    return response("Esta página se refrescará en 5 segundos hacia...", 200)
        ->header('Cache-Control', 'max-age=3600')
        ->header('Refresh', '5; url=http://www.desarrolloweb.com');
});
```

Hemos puesto dos cabeceras de HTTP response, la primera para definir la antigüedad máxima del elemento cacheado y la segunda para decir que en 5 segundos se refresque la página enviando al navegador a una nueva URL.

## Enviar una vista mediante una instancia Response

Habrás advertido que resulta especialmente incómodo enviar como respuesta una cadena en el primer parámetro del helper `response()`. Si la respuesta es muy larga para colocarla, así tal cual, en una cadena, o simplemente prefieres separar la salida en una vista, tal como has aprendido, tenemos una alternativa muy útil.

Se trata de invocar al helper de `response()` sin enviar ningún parámetro. La respuesta (objeto `Response`) estará limpia para configurar todos sus detalles. Uno de los métodos del objeto `Response` que te devuelve el helper `response()` es `view()`, en el que indicamos la vista que queremos procesar.

Luego podemos indicar nuevas cabeceras HTTP en la respuesta, encadenando métodos, tal como hemos visto en ejemplos anteriores en este artículo.

```
Route::get('respuesta6', function(){
    return response()
        ->view('error')
        ->header('status', 404)
        ->header('Refresh', '5; url=/');
});
```

En el código anterior tendrías un ejemplo de `response` en la que cargamos una vista llamada "error" y luego enviamos dos cabeceras adicionales, una para mandar un código de status (404 de error "página no encontrada") y una redirección pasados 5 segundos a la home del dominio.

## Conclusión

De momento dejamos por aquí esta introducción al sistema de `Response` de Laravel, hay bastante más que ver en los siguientes artículos, como responder en formato JSON, generar cookies, enviar archivos para descarga, etc. pero como nuestro objetivo de momento es la presentación de los componentes principales del framework, es más que suficiente.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 24/09/2015  
Disponible online en <http://desarrolloweb.com/articulos/responses-laravel5.html>

# Validación de formularios en Laravel 5.1

A lo largo de los próximos artículos vamos a recorrer el sistema de validación incorporado en Laravel y los mecanismos que nos ofrece el framework para realizar formularios usables, capaces de recordar su estado en diferentes llamadas a la página.

## Recibiendo datos en Laravel 5

### Metodos y alternativas para recibir datos enviados en la solicitud HTTP con Request en Laravel 5.

Laravel 5 nos ofrece toda la información relativa a la solicitud por medio de un objeto Request, el cual estudiamos ya en el artículo de [HTTP Request en Laravel](#). Ahora queremos revisar y poner ejemplos de una serie de métodos útiles para recibir los datos que nos envían.

El envío de datos puede realizarse tanto por get como por post y siempre los recibiremos por medio del objeto Request, usando los mismos métodos, independientemente del método de envío. Como ya sabemos, cuando se envían datos por get, viajan en variables dentro de la URL y cuando se envían por post, viajan en las cabeceras del HTTP, de manera invisible para el usuario.



Antes de ponernos a ver contenido de este artículo, refresquemos algo nuestra memoria sobre los request y el flujo de trabajo de Laravel 5. Vamos a trabajar con una ruta como la siguiente:

```
Route::match(['get', 'post'], 'input', 'TestRequestController@recibir');
```

Hemos definido una ruta hacia un controlador llamado "TestRequestController" con una acción llamada "recibir" ("TestRequestController@recibir") que es donde tenemos que codificar la respuesta ante la solicitud "input". Esta acción se pondrá en marcha cuando el verbo del HTTP sea "post" o "get". Todo esto lo vimos en el [artículo sobre routing en Laravel](#).

Ahora veamos el código de este controlador, que define el método recibir() para codificar el comportamiento ante la ruta anterior.

```
class TestRequestController extends Controller
{
    public function recibir(Request $request)
    {
        // código para esta acción del controlador
    }
}
```

Dentro del controlador tenemos el método `recibir()`, al que le inyectamos como dependencia el objeto de la clase `Request`. De esa manera somos capaces de acceder a los datos de la solicitud. Todo el resto del código PHP que vamos a ver en este artículo se sitúa dentro de ese método y cuando hagamos referencia a `$request` ya sabemos que se trata de una instancia del objeto `HTTP Request`.

## Averiguar el método en una solicitud

Como hemos mencionado, la manera de recoger los datos que nos envíen es la misma en cualquier método (`get` si se envían por la URL, `post` si se envían por formulario). Por tanto, si en algún momento necesitamos averiguar cuál es (`get` o `post`), podemos consultarlo a través del método de `Request` llamado `method()`.

```
$metodo = $request->method();
```

Esto, en la ruta anterior, que hemos definido tanto para el verbo `"post"` como para el verbo `"get"` nos devolvería el método con el que se ha producido la solicitud, entre los dos posibles.

Si en cualquier momento queremos preguntar si estamos ante un método en particular podemos preguntarlo con `isMethod()`.

```
if($request->isMethod('post')){
    echo "Estoy recibiendo por post";
}
```

## Recuperando datos enviados por GET / POST

Ahora vamos a ver una serie de métodos de `Request` que nos sirven para traernos las variables enviadas por `GET` o `POST`.

**Obtener un dato suelto:** método `input()`, indicando la cadena que queremos recibir.

```
$nombre = $request->input("nombre");
```

**Nota:** Recuerda que para probar este ejercicio lo más cómodo será usar una extensión como `Postman`, que permite indicar la URL y definir el verbo del `HTTP` e incluso mandarle datos por formulario. [Esto lo explicamos en el artículo Verbos en las rutas de Laravel](#). También se mencionó en ese artículo que para que los envíos por `post` nos funcionen habrá que desactivar el middleware de protección `CSRF`

```
"VerifyCsrfToken".
```

**Valor predeterminado:** Ahora también podemos especificar un valor predeterminado para recibir como respuesta en la consulta, si es que ese dato no fue recibido en el Request.

```
$edad = $request->input("edad", 18);
```

**Recibir datos que vienen en arrays:** Como debes saber, es posible agrupar en arrays datos que se envían por formularios o por la URL. Hay diversos modos, el más habitual es ponerle el name del campo input un valor con unos corchetes vacíos al final.

```
Categoria0: <input name="categorias[]">  
<br />  
Categoria1: <input name="categorias[]">  
<br />  
Categoria2: <input name="categorias[]">
```

Otra opción es que te vengan los datos de un SELECT "multiple".

```
<select name="categorias" multiple>  
...  
</select>
```

La cosa se puede complicar todo lo que desees, con campos que se organizan en arrays de varias dimensiones.

```
<input name="categorias[0][nombre]">  
<input name="categorias[0][votos]">  
<input name="categorias[1][nombre]">  
<input name="categorias[1][votos]">  
...
```

Todas esas alternativas las construyes con el HTML, no vamos a seguir dando ejemplos. Ahora, si lo desees, con Laravel lo podrías recoger con:

```
$categorias = $request->input("categorias");
```

Así recibes un array con todas las categorías, pero podrías acceder a alguna de las posiciones del array en concreto, mediante la notación "punto".

```
$categorias = $request->input("categorias.1.voto");
```

**Saber si se recibe un dato:** Puedes saber si estás recibiendo un dato en concreto en el conjunto de los datos recibidos con HTTP, mediante el método `has()`.

```
if ($request->has("edad")){
    echo "Encontré la edad";
}
```

Recibir un grupo de datos: otra posibilidad, cuando quieres recibir varios datos a la vez, es usar el método `only()`, en el que indicas un listado de las claves (key) de aquellos datos que deseas recibir. Lo que te devuelve `only()` es un array asociativo con pares llave/valor de aquellos elementos indicados.

```
$datos_grupo = $request->only('nombre', 'id');
```

Mediante el anterior código recibirás un array asociativo con la llave "nombre" e "id". Sus valores serán aquellos valores que se reciban en Request. Si alguno de ellos se ha especificado pero no se ha recibido, entonces simplemente recibirás null como valor.

**Recibir todos los datos menos algunos:** Es parecido al anterior método, pero en vez de indicar qué queremos recibir, se indica qué se quiere excluir.

```
$datos_excepto = $request->except('categorias');
```

**Recibir todos los datos del Request:** Este método ya lo vimos anteriormente, simplemente nos devuelve un array asociativo con la totalidad de los datos que se encuentren en la solicitud.

```
$todos = $request->all();
```

**Recibir un campo de tipo file:** Si estamos haciendo upload de archivos a través de un formulario, los campos los tenemos que recoger de una manera diferente, usando el método `file()`.

```
$archivo = $request->file('archivo');
```

También podemos comprobar si existe el archivo usando `hasFile()`.

```
if($request->hasFile('archivo')){
    $archivo = $request->file('archivo');
}
```

Existen una serie de métodos para trabajar con archivos enviados desde formulario, pero preferimos verlos más adelante cuando hagamos un ejercicio completo de upload de archivos con Laravel, porque tiene varios detalles que sería largo de tratar en este momento e innecesario con lo que sabemos hasta el momento de este framework PHP.

De momento en el próximo artículo nos vamos a introducir en la validación de la entrada con Laravel, un tema mucho más básico e importante.

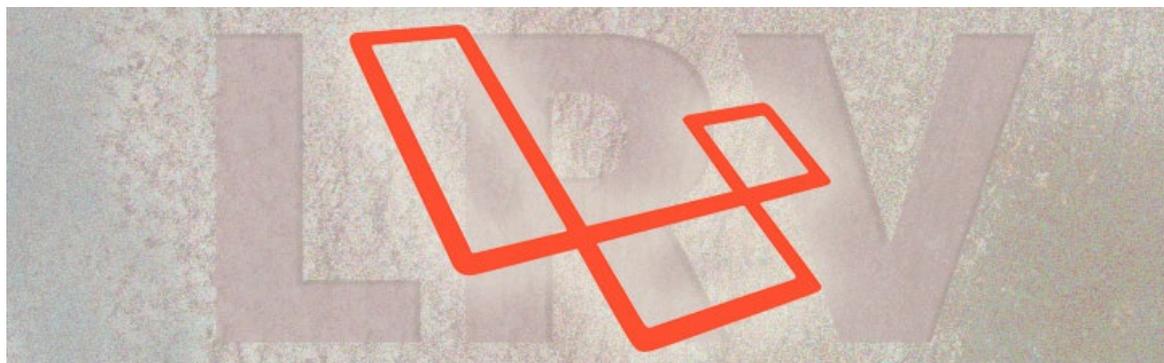
Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 30/09/2015  
Disponible online en <http://desarrolloweb.com/articulos/recibiendo-datos-laravel5.html>

## Volcado de la entrada de datos de usuario a la sesión

El volcado de datos de entrada del usuario en el sistema de sesión, por medio de las funciones **Old Input** de Laravel 5 permite acceder a la entrada de datos de una solicitud en la siguiente.

Habíamos quedado que en este artículo comenzaríamos con la validación de formularios, pero para poder abordar ese asunto con mayor facilidad, convendría detenerse antes en aprender una técnica que pondremos en práctica en las validaciones. Se trata de flashear los datos de input en la sesión, para recordarlos en la próxima solicitud. Es algo que hace Laravel de manera sencilla, y que veremos resulta muy útil.

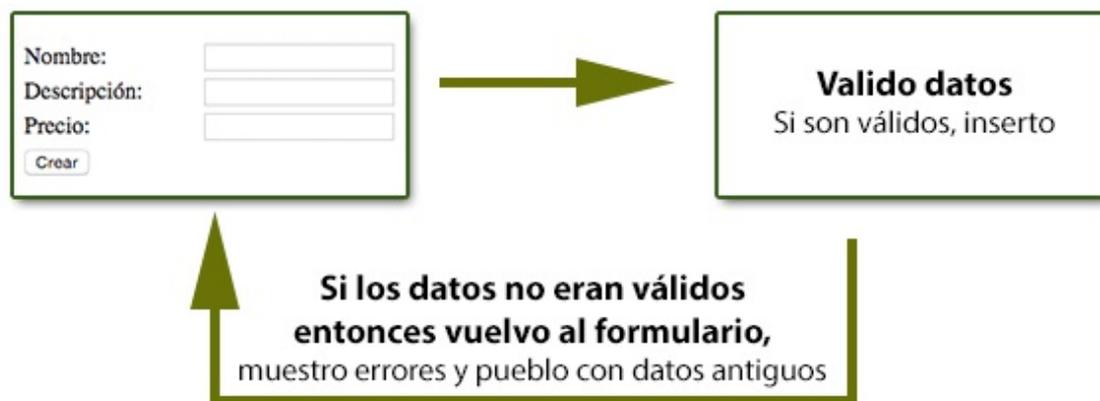
Volcar los datos de entrada del usuario en la sesión es una práctica habitual cuando se trabaja con formularios. Su objetivo normalmente está ligado a los procesos de validación y permiten realizar las comprobaciones en una página y redirigir a otra memorizando la entrada de datos producida con anterioridad.



En términos generales, las funciones "old input" de Laravel permiten acceder a los datos enviados por el usuario, ya sea GET o POST, en una solicitud y también en la siguiente. En este artículo del [Manual de Laravel 5](#) aprenderemos a hacerlo.

### Por qué volcar los datos

Por si todavía no queda clara la necesidad, podemos explicarlo brevemente. Con el siguiente diagrama quedará más fácil de entender.



Tenemos una página con un formulario. Esa página envía los datos de la solicitud a otra página y en ella realizamos validaciones. Si todo se validó correctamente hacemos las operaciones que nos hayan pedido y listo. Puede ser una inserción, enviar un email o lo que sea que tengamos que hacer como respuesta a esa solicitud. Pero si los datos no eran válidos, lo ideal es mostrar de nuevo el formulario y los errores de validación encontrados y para ello lo más cómodo es retornar a la página donde estaba el formulario.

Volviendo a la página donde estaba el formulario tenemos un problema y es que los datos del usuario, los que había escrito en el formulario, ya no se encuentran disponibles. Al redirigir de nuevo al formulario se genera una segunda solicitud y aquella información enviada solo está disponible en la primera, cuando se validaron los datos.

La solución que se propone es volcar los datos de la solicitud a la sesión, lo que nos permite acceder a ellos más tarde. Y esto no es algo propio de Laravel, sino que se trata de una estrategia bastante habitual cuando se desarrolla aplicaciones web.

## Old Input

En Laravel 5 los datos enviados a una solicitud se pueden leer en esa solicitud y en la siguiente, si realizamos el correspondiente volcado a la sesión. Es una de las funcionalidades que nos ofrece el objeto Request.

En la mayoría de los casos esto es algo que Laravel hace de manera automática. Es decir, en un proceso de validación, si utilizamos el sistema de validación integrado en Laravel, el flash de los datos de Request a la sesión se hace sin que el desarrollador tenga que intervenir, pero además nosotros podemos hacerlo explícitamente, si no validamos con las herramientas del framework o si nos viene bien para cualquier otro tipo de situación.

El método que nos permite volcar los datos de HTTP Request a la sesión se llama `flash()`. Lo hacemos sobre el objeto `$request` que ya como costumbre venimos inyectando en el controlador.

```
$request->flash();
```

**Nota:** También como alternativa puedes flashear solo unos campos o bien todos menos aquellos que se indiquen.

```
$request->flashOnly('nombre', 'edad');  
$request->flashExcept('clave');
```

Luego querrás redirigir hacia una nueva ruta. Esta es una operación que depende del sistema de respuesta (response) y que se puede hacer de varias maneras. Comúnmente harás algo como:

```
return redirect('/uri/a/la/que/redirijo');
```

Como generalmente estas dos operaciones, el flasheo y la redirección se suelen hacer juntas, Laravel nos ofrece métodos para poder hacerlas de una sola vez.

```
return redirect('uri')->withInput();
```

Si no indicas nada como parámetro a `withInput()`, simplemente se mandan todos los datos del Request. Aunque también puedes flashear solo una porción de la Request, simplemente indicando aquellos campos que quieres que se memoricen en la siguiente solicitud.

```
return redirect('redirijo')->withInput($request->except("nombre"));
```

**Nota:** Recuerda que el método `except()` devuelve todos los datos del Request menos aquellos campos que se indiquen.

## Recuperar los datos de la solicitud antigua

En aquella solicitud donde rediriges y para la cual has querido memorizar la Request puedes recuperar la información de varias maneras. Solo un detalle, los datos del usuario ya no van a estar en la HTTP Request, sino en la sesión. Sin embargo, en ningún caso necesitaremos acceder nosotros directamente a la sesión.

Laravel ofrece distintos métodos para acceder a esa información. Uno de ellos a través del objeto Request, aunque no desde los métodos que ya conocemos como `input()`, `except()`, `all()`, etc. (donde se encuentran los datos de la solicitud normalmente), sino a través del método `old()`.

```
$request->old('nombre');
```

Además Laravel ofrece una función global llamada `old()` a la que tenemos que indicar como parámetro el campo que queremos recuperar y nos devuelve su valor.

```
old('nombre');
```

Al ser un helper global podrás usarlo dentro de una vista sin necesidad de enviarte ningún dato. Lo podrás volcar en un campo input o en cualquier lugar donde lo necesites.

```
<input type="text" name="nombre" value="<?=old("nombre")?>">
```

Si estás en un template de Blade puedes usar la sintaxis de las dos llaves, que no hemos visto todavía, pero te quedaría como esto.

```
<input type="text" name="nombre" value="{{old('nombre')}}">
```

La función `old()` además tiene un segundo parámetro opcional que permite indicar el valor por defecto en caso que no exista ese campo entre los elementos flasheados en la sesión.

```
old("nombre", "Anonimo");
```

Dada la invocación anterior, si no hay datos flasheados en la sesión, o no existe el campo "nombre" entre los elementos existentes, `old()` devolverá el valor "Anónimo".

## Conclusión

Has aprendido a recuperar los datos de Request no solo desde una solicitud, sino también desde la siguiente, por medio del flasheo en la sesión. En el siguiente artículo comenzaremos a hablar de las validaciones dentro de Laravel y usaremos el sistema que el framework nos ofrece. Entonces verás que la parte del flash y la redirección son asuntos que Laravel hace ya de manera automática, pero te vendrá bien conocer el modo de recuperar los datos de la solicitud antigua, porque lo necesitarás para componer los formularios con la conveniente usabilidad.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 08/10/2015  
Disponible online en <http://desarrolloweb.com/articulos/volcado-entrada-datos-usuario-sistema.html>

## Validaciones con Laravel 5

### Introducción a las validaciones de datos del usuario, entrada mediante HTTP Request, con el framework PHP Laravel 5.

Una de las necesidades fundamentales de cualquier aplicación en general, y aplicaciones web en particular, son las validaciones de la entrada de datos del usuario. Obviamente en Laravel se toman muchas molestias por proporcionarnos métodos de validación sencillos y potentes. Además no existe una única vía, por lo que habrá distintos caminos para realizar una validación.

Para comenzar el apartado de validaciones en el [Manual de Laravel](#), vamos a analizar la manera más rápida de validar, manteniendo la lógica en el controlador. Más adelante veremos alternativas que varían un poco esta propuesta, permitiendo separar el código de validaciones a clases diferentes, pensadas para ello, que colocaremos en la carpeta Requests (fíjate la "s" al final para distinguirlo de Request).

El ejemplo completo de validación que vamos a realizar está basado en las prácticas que nos sugiere la documentación oficial de [Laravel 5.1 en la sección "Validation Quickstart"](#).



## Creamos las rutas

Vamos a necesitar dos rutas en nuestra aplicación. La primera de ellas nos mostraría un formulario donde estarán aquellos campos que necesitemos que el usuario rellene. La segunda para validar los datos y realizar las acciones necesarias si son correctos o no.

En cuanto a la primera ruta, donde estará el formulario, tendremos algo como esto:

```
Route::get('producto/crear', 'ProductoController@create');
```

Esta ruta llamará a la acción "create" sobre el controlador "ProductoController", siempre que se acceda a la URI "producto/crear" con el verbo HTTP get.

En la segunda ruta realizaremos las acciones de validación y la parte de insertar la información en la base de datos, en el caso que la validación fuera correcta.

```
Route::post('producto', 'ProductoController@store');
```

Como puedes ver, esa ruta se activa con un envío por post sobre la URI "producto". Entonces se invoca a la acción "store" sobre el controlador "ProductoController".

Como has podido comprobar, en "ProductoController" concentramos todo el trabajo del ejemplo de validación, es decir, las dos rutas van a diferentes acciones del mismo controlador.

## Controlador con validación ProductoController

Ahora vamos a observar el controlador donde vamos a realizar las acciones invocadas por las rutas.

```
class ProductoController extends Controller
{
    // Aquí van los métodos del controlador
    // el código de los métodos está más abajo en el artículo
}
```

Comenzaremos por ver el método `create()`, que es invocado por la primera ruta. En tal método tenemos que invocar la vista donde está el formulario mediante el cual el usuario insertará la información que luego vamos a validar.

```
public function create(Request $request)
{
    return view('producto.create');
}
```

**Nota:** De momento nos importa poco la vista del formulario. Puedes poner un archivo que tendrá básicamente el código HTML de cualquier formulario con cualquier número de campos. Al final del artículo os pasaré el código completo de la vista que he usado yo en mi ejemplo.

Este método del controlador no necesita más código, porque solo se encarga de mostrar la vista del formulario. Donde tenemos que hacer las validaciones será en el siguiente método, que enseguida os mostramos.

## Trait `ValidatesRequests` en el controlador base

Laravel 5 incluye un trait que se carga en el controlador base (clase `Controller`), llamado "`ValidatesRequests`". Ese trait contiene código que estará disponible en todos los controladores que nosotros creamos, puesto que todos extienden la clase `Controller`.

**Nota:** Puedes encontrar el código del controlador base (clase `Controller`) en la ruta "`app/Http/Controllers/Controller.php`" y encontrarás la declaración de uso del trait en la línea:

```
use DispatchesJobs, ValidatesRequests;
```

No confundas ese "use" que es la primera línea de código de una clase, con un "use" que aparece en cualquier lugar del código para definir que estás usando cosas declaradas en otros namespaces.

Dentro del trait "`ValidatesRequests`" hay un método que nos sirve para hacer validaciones de los datos que nos llegan mediante HTTP Request, llamado `validate()`. Por tanto, en todos nuestros controladores podremos invocar este método para realizar validaciones de datos de entrada.

El método `validate()` recibe dos parámetros:

1. El objeto Request
2. Las reglas de validación que queremos definir.

El objeto Request ya lo conocemos de sobra, puesto que ya lo hemos tratado muchas veces. Lo inyectarás en el método del controlador como ya hemos detallado en el artículo [HTTP Request en Laravel 5](#).

Las reglas de validación son diversas y perfectamente configurables por nosotros mediante una sencilla sintaxis. De momento vamos a conocer algunas reglas elementales, pero Laravel tiene muchas otras que trataremos más adelante. Aunque este tema lo puedes consultar de una manera muy rápida en la documentación oficial de Laravel 5.1 en la sección [Available Validation Rules](#).

Para especificar las reglas, como veremos en el código del método store(), un poco más abajo, las indicamos con un array asociativo. Como llaves (key) del array usamos el nombre del campo que deseamos validar (atributo "name" de los campos de formulario) y como valores indicamos todas las reglas que se deben comprobar en dicho campo para considerarlo correcto.

Este sería el código de nuestro método store() del controlador, donde puedes ver la llamada a validate() para la validación del formulario de producto que has debido de crear en la anterior acción.

```
public function store(Request $request)
{
    $this->validate($request, [
        'nombre' => 'required|max:255',
        'descripcion' => 'required',
        'precio' => 'required|numeric',
    ]);

    echo 'Ahora sé que los datos están validados. Puedo insertar en la base de datos';
}
```

## Comportamientos ante un válido / inválido

Cuando se llama el método validate(), como resultado de pasar un proceso de comprobación de los datos de entrada, pueden ocurrir dos cosas:

1. **Los datos eran válidos:** en cuyo caso simplemente se continúa el procesamiento del método, ejecutando las instrucciones que se hayan después de la llamada a validate().
2. **Los datos eran inválidos:** en cuyo caso se hace una redirección a la ruta anterior, donde estaba el formulario. (Aunque cabe señalar que en conexiones Ajax este comportamiento puede tener variaciones, puesto que se tendrá que generar un JSON y enviar como respuesta al cliente, algo que no vamos a tratar ahora).

En el código del método store(), que presentamos unos párrafos atrás, comprobarás que después de invocar el método validate() se pone el código que realiza las acciones pertinentes. Nosotros hemos colocado un simple "echo", lanzando un mensaje, pero generalmente aquí habría que llamar al modelo que se encarga de guardar esos datos en la base de datos.

## Mostrar los errores de validación

Con todo el código anterior ya tenemos implementado todo el proceso de validación, en esta dinámica de hacerlo dentro del controlador. Realmente has observado que el corazón del proceso es el método `validate()` presente en todos los controladores. Sin embargo, para completar este ejercicio debemos realizar algunas tareas sobre la vista para mejorar la usabilidad del formulario, ayudando al usuario a identificar qué es lo que ha ocurrido y volviendo a poblar el formulario. Serán por tanto dos tareas las que nos faltan:

- Mostrar los errores de validación
- Recuperar los datos en los campos de formulario que se habían escrito anteriormente

Los errores de validación los vamos a extraer de una variable llamada **`$errors`**, que está disponible en toda vista.

Laravel automáticamente crea la variable `$errors` flasheando en la sesión todos los errores de validación que se hayan podido producir. Si no hubo tal error de validación Laravel crea igualmente esa variable, pero vacía, por lo que podemos usarla siempre que queramos, sin preocuparnos si está o no está definida, porque siempre lo va a estar.

Además el bindeo se realiza automáticamente a la vista, por lo que nosotros no tenemos que hacer absolutamente nada para pasar esa variable.

Por tanto, en nuestra vista podemos fácilmente recorrer los errores definidos en `$errors` y mostrarlos en el formato que nosotros deseemos. A continuación tendrías un pedazo de código que se encarga de recorrer esos errores, mostrando un elemento DIV con los errores en el caso que haya alguno, y listando cada uno en un elemento LI de una lista UL.

```
@if(count($errors) > 0)
    <div class="errors">
        <ul>
            @foreach($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

De momento este código te puede parecer un poco extraño, porque no hemos hablado de la sintaxis del sistema de templates Blade, el oficial de Laravel. No te preocupes porque lo tocaremos más adelante y podrás entenderlo perfectamente. No obstante, **para que te funcione, es muy importante que nombres el archivo de esta vista con extensión ".blade.php"**, porque si no, el código "Blade" no se ejecutaría.

**Nota:** Los mensajes de error aparecen en inglés, pero es muy fácil obtener las traducciones. [En Github hay un proyecto que te las ofrece ya listas en una serie de idiomas](#). Te explicaremos más adelante en DesarrolloWeb.com cómo debes instalar las traducciones en tu proyecto.

## Poblar de nuevo los valores de los campos de formulario

Primero decir que esta otra parte del trabajo en la vista realmente nos la podríamos ahorrar en Laravel, ya que hay un método muy sencillo por el que Laravel nos lo hace automáticamente. Se trataría de usar la librería que viene con Laravel para generar código de los formularios, pero como no la hemos visto todavía, nos toca hacerlo "a mano", aunque comprobarás que tampoco es nada doloroso.

Simplemente en los campos INPUT hemos creado el value, asignando lo que nos devuelve el helper `old()`, indicando el campo que queremos recuperar. Esa función se encarga de traerse aquello que estaba escrito anteriormente en el campo del formulario, tal como explicamos en el artículo [Volcado de la entrada de datos de usuario a la sesión](#).

```
<input type="text" name="nombre" value="{{old('nombre')}}">
```

Este código no necesita muchas más explicaciones, pero simplemente tómalolo como una alternativa, sabiendo que hay mejores maneras de hacer esto, si le dejas a Laravel la responsabilidad de crear el HTML de tu formulario.

## Código completo de la vista

Como prometí, aquí va el listado completo de la vista que he usado para este ejercicio.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Crear un producto</title>
  <style>
    .errors{
      background-color: #fcc;
      border: 1px solid #966;
    }
    form{
      margin-top: 20px;
      line-height: 1.5em;
    }
    label{
      display: inline-block;
      width: 120px;
    }
  </style>
</head>
<body>
  <h1>Crear un producto</h1>
  @if(count($errors) > 0)
    <div class="errors">
      <ul>
        @foreach($errors->all() as $error)
          <li>{{ $error }}</li>
        @endforeach
      </ul>
    </div>
  @endif
</body>
```

```
</ul>
</div>
@endif

<form action="/producto" method="post">
  <label for="nombre">Nombre:</label> <input type="text" name="nombre" value="{{old('nombre')}}">
  <br />
  <label for="descripcion">Descripción:</label> <input type="text" name="descripcion" value="{{old('descripcion')}}">
  <br />
  <label for="precio">Precio:</label> <input type="text" name="precio" value="{{old('precio')}}">
  <br />
  <input type="submit" value="Crear">
</form>
</body>
</html>
```

Por favor, ten en cuenta que esta vista tiene un código muy elemental, simplemente para salir del paso. Observa estos detalles:

- No estoy pasando el token para protección CSRF, por lo que tendrás que desactivar el middleware asociado. Esto lo hemos explicado ya en varias ocasiones, así que debes saber a qué me refiero. Si no, revisa el artículo [Verbos en las rutas de Laravel](#).
- Guarda el archivo en la carpeta correspondiente. Tal como quedó el código del método create() del controlador, sería en la carpeta "resources/views/producto".
- El nombre del archivo será create.blade.php. Ese nombre lo marcamos al invocar la vista, desde el controlador ProductoController, en la acción create().

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en *15/10/2015*  
Disponible online en <http://desarrolloweb.com/articulos/validaciones-con-laravel5.html>

## Validación reutilizable por Requests en Laravel 5

La validación por Requests de Laravel 5 es un estilo de validación más avanzado que puede ser reutilizable y nos libera a los controladores de las operaciones de comprobación de la entrada de usuario.

En el artículo anterior del [Manual de Laravel](#) estuvimos estudiando una [primera vía para construir sistemas de validación de la entrada de usuario](#). Observamos que es bastante potente, ya que la mayoría de las acciones típicas durante la validación las hace el framework directamente, por lo que nuestro código se queda muy sencillo.

No obstante, todavía puede mejorar la situación. En este artículo explicamos las validaciones por el mecanismo de extender la Request, creando Request especializadas que son capaces no solo de entregar la información del usuario, sino también de validarla convenientemente.

No confundir "Request" en singular con "Requests" en plural. La primera es la que hemos venido usando a lo largo de todo el curso de Laravel, que nos ha proporcionado diversas informaciones y funcionalidades sobre la solicitud del cliente. Esa Request, en singular, siempre está presente en este framework PHP como parte de los recursos que te entregan para realizar tus aplicaciones. Pero por otra parte, nosotros como desarrolladores, podemos crear un número indeterminado de Requests, en plural, para aquellas variantes de entrada de datos que necesite nuestra aplicación.

Las Requests que podemos crear, al extender la Request, disponen de la misma funcionalidad que ya conocemos. Pero además, en esas clases Request que crearemos a partir de ahora vamos a incluir código necesario para las validaciones específicas de ese tipo de entrada. Por ejemplo, una de las Requests podrá ser para validar los datos del usuario, otra para validar datos de un producto, un post, o lo que sea necesario.



## Ventajas de la validación por Requests

Son muchos los puntos que mejora este sistema de validación, pero básicamente debemos señalar:

1. Separación del código. Es la base de muchos patrones de diseño y nos permite mantener clases más simples, más desacopladas y por tanto de más fácil mantenimiento.
2. Reutilización, al poder usar esa misma request en cualquier parte donde estemos recibiendo ese tipo de entrada de usuario. Por ejemplo, cada vez que reciba datos de un usuario podré usar la request del usuario.

## Cómo crear una Request personalizada

Existe una carpeta específica para guardar las Requests que creemos para nuestro proyecto, en la ruta "app/Http/Requests".

Además existe un comando de artisan "make:request" que nos hace el trabajo de crear el código base de estas clases para validar una solicitud dada.

```
php artisan make:request ProductoRequest
```

Con esto obtendremos una clase llamada ProductoRequest que se sitúa en la ruta de las requests. Su ruta completa por tanto será: "app/Http/Requests/ProductoRequest.php". Si abres este archivo comprobarás que tiene un par de métodos:

```
class ProductoRequest extends Request
{
    public function authorize()
    {
        return false;
    }

    public function rules()
    {
        return [
            //
        ];
    }
}
```

Ambos métodos son para realizar validaciones de la información, pero su responsabilidad es de distinto tipo:

- En la parte de `authorize()` se comprobará si el usuario está autorizado a realizar esa request. Por ejemplo podrás comprobar si el usuario tiene los permisos adecuados, si aquello que intenta modificar le pertenece y cosas así. Si te fijas, cuando creas el Requests se devuelve un `return false`, y eso querrá decir que no se le deja pasar.
- En la parte de `rules()` indicarás las reglas que deben de cumplirse para que se considere que los datos son válidos. Si los campos existen y tienen los valores que se esperan, numéricos, cadenas, de unas dimensiones, etc. [Estas reglas de validación son las mismas con las que trabajamos en el anterior artículo sobre validaciones.](#)

Laravel primero hará la comprobación de `authorize` y luego verá si cumple las reglas que se han indicado en `rules()`.

A continuación tienes el listado de una Requests que hemos creado para el formulario de producto que usamos en el artículo anterior.

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

use App\Http\Requests\Request;
use App\Http\Requests\Requests;

class ProductoRequest extends Request
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'nombre' => 'required|max:255',
```

```
'descripcion' => 'required|min:10',
'precio' => 'required|numeric',
];
}
}
```

Como puedes ver, en `authorize()` hemos colocado de momento un `return true;`. Esto producirá que siempre se autorice a realizar esta acción. Más adelante escribiremos alguna cosa en este método.

Por su parte, en `rules()` tenemos las mismas reglas de validación que ya conocemos, en el array que se devuelve.

## Cómo usar una de nuestras Requests personales

Después de crear una requests podemos usarla en el controlador. Solo necesitamos realizar dos tareas:

Primero agregar el "use" correspondiente para que se conozca el nombre de la clase `ProductoRequest`.

```
use App\Http\Requests\ProductoRequest;
```

Luego podremos usar esa clase en la cabecera de los métodos que deban realizar la validación de la entrada de datos.

```
public function store(ProductoRequest $request)
{
    echo 'Estoy usando la Request personalizada ProductoRequest. ';
    echo 'No necesito invocar explícitamente las funciones para validar. ';
    echo 'Si estoy aquí sé que los datos están validados.';
    dd($request->all());
}
```

Como puedes ver, al definir el método `store()` colocamos el parámetro que ahora se especifica con el nombre de nuestra request personalizada: `ProductoRequest`.

Laravel ya se encarga por su cuenta de llamar a los procedimientos de validación automáticamente, sin que tengamos que intervenir. Si no pasa la validación realizará una de estas dos acciones:

1. Si no pasa el `authorize()` mostrará un "Forbidden". Este tipo de error se supone que será por una acción no permitida a nivel de aplicación y teóricamente porque están queriendo hacer alguna cosa, burlando la seguridad. Por ello no se realiza un tratamiento especial, mostrando simplemente un feo mensaje que no da muchas pistas al usuario de lo que pueda estar pasando.
2. Si no pasa el `rules()` hace lo mismo que el método `validate` cuando no se pasan las reglas, es decir, redirige a la página anterior flasheando la entrada de datos a la sesión y generando los errores de validación para que luego podamos mostrarlos, bien claritos para ayudar al usuario a localizar los problemas en la entrada de datos. [Para más aclaraciones consulta el anterior artículo de introducción a las validaciones.](#)

Luego, cualquier código dentro del método se ejecutará solamente si las validaciones pasaron correctamente. Así que el método del controlador nos queda liberado de cualquier proceso de validación y podemos escribir solo el código de los procesos necesarios para ejecutar las acciones deseadas.

Además, dentro del método del controlador podemos usar tranquilamente el objeto de la clase `ProductoRequest` como si fuera un objeto de la clase `Request` normal, ya que la clase `ProductoRequest` hereda de `Request`. Podremos acceder a los datos de usuario y a todas las otras operaciones que hay en un `Request` de las que ya conocemos.

En todos los controladores que necesitemos validar la entrada de datos podremos hacer el mismo procedimiento para validar implícitamente, usando el `Request` creado para el caso. Por tanto, no es necesario repetir el código de validaciones en varias partes de la aplicación y se produce la reutilización.

## Qué sentido tienen las validaciones en el controller

Si has apreciado lo limpios que quedan nuestros controladores con este estilo de validación y valoras positivamente la separación del código y la reutilización de las funciones de validación, quizás te preguntarás ¿Y entonces para qué necesito, o qué sentido tienen, las validaciones en el controller que vimos en el artículo anterior?

Bueno, la verdad es que continúan siendo útiles en varios casos, primero y menos importante es que en determinado momento puede que nos resulte más cómodo realizar una validación sencilla de alguna cosa en el controlador, sin que tengamos que generar una nueva clase para realizar algo rápidamente.

Pero la razón de más peso es que hay acciones que pueden requerir validaciones más especializadas. Por ejemplo al crear una factura podemos validar una serie de informaciones básicas. Pero quizás al editarla hay otra serie de reglas que se deben comprobar. En ese caso, esas validaciones específicas tenemos que colocarlas en la acción del controlador que las necesita.

De momento es todo, esperamos que apreciéis la potencia y simplicidad de este nuevo método de validar y lo podáis practicar.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en 22/10/2015  
Disponible online en <http://desarrolloweb.com/articulos/validacion-reutilizable-request-laravel5.html>

# Trabajar con cookies en Laravel

Teoría y práctica sobre el trabajo con cookies y algunas de las funciones útiles que nos ofrece este soporte para guardar información.

## Cookies en Laravel

**Tutorial y referencia para el trabajo con cookies en Laravel 5. Crear cookies, recuperar el valor de cookies guardadas, etc.**

Las cookies son de sobra conocidas por todos. Sirven para almacenar pequeñas cantidades de información textual en el navegador del usuario, que podrán persistir entre diferentes visitas a un mismo sitio web. Por supuesto, en Laravel se hacen un buen cargo de ellas, facilitando procedimientos como la creación o lectura de cookies de una manera sencilla.

Este artículo te servirá como referencia y guía para el trabajo con Cookies, aprendiendo a crearlas desde el sistema de "Response" y a acceder a ellas mediante el sistema de "Request".

## Cómo funcionan las Cookies

Antes de explicar los mecanismos que nos ofrece Laravel creo que unas pequeñas notas sobre cómo funcionan las cookies aclararán muchas dudas. Como no es el objetivo de este artículo hablar de cookies en general, lo voy a esquematizar en cuatro puntos:

- La creación de cookies desde el servidor, en lenguajes como PHP, se debe producir en la respuesta de una solicitud HTTP (Response). Osea, solo cuando devuelvo los datos de una solicitud al servidor soy capaz de escribir una cookie en el navegador.
- La lectura de cookies se realiza a través de la petición. Si el navegador del usuario tiene alguna cookie almacenada en el sistema, llega al servidor en la solicitud emitida por el cliente web (Request).
- Las cookies viajan en las cabeceras del protocolo HTTP, de manera transparente para el usuario. La información para la creación o actualización de cookies se escribe dentro de las cabeceras del HTTP de respuesta. La información para la lectura de las cookies viaja al servidor en las cabeceras HTTP de la solicitud.
- Por todo ello, cuando proceso una página en el servidor que escribe una cookie, esa cookie solo estará presente para lectura en la siguiente solicitud que se realice al servidor.



## Cookies en Laravel

Como comprenderás según lo anterior, las cookies forman parte de dos sistemas de Laravel, el de Request para acceder a las cookies creadas, así como el de Response para almacenar nuevas o actualizar antiguas cookies.

Además es interesante saber que todas las cookies gestionadas por Laravel están encriptadas y firmadas con un código de autenticación, de ese modo podemos asegurar que no han sido modificadas a mano por el usuario.

## Crear cookies en el navegador con Laravel

Las cookies que creamos desde Laravel son instancias de una clase de Symfony: `Symfony\Component\HttpFoundation\Cookie`. Para crearlas Laravel ofrece diversos mecanismos que veremos. El más sencillo y directo es un helper llamado `cookie()` que realiza la tarea por nosotros.

Los parámetros básicos del helper `cookie()`, que querrás utilizar al crear la mayoría de cookies, serán el nombre de la cookie, su valor y el tiempo de persistencia (en este caso expresado en minutos).

```
$nueva_cookie = cookie('nombre', 'valor', $minutos);
```

Si queremos que la cookie se recuerde por el mayor tiempo posible (lo máximo son 5 años) puedes usar esta otra alternativa de código.

```
$nueva_cookie = cookie()->forever('micookie', 'mivalor');
```

Con eso obtengo el objeto de la clase `Cookie`, pero todavía me falta enviarlo al navegador, algo que veremos enseguida.

Además también puedes crear cookies con otra serie de parámetros, opcionales, que listamos a continuación en el orden en el que tienen que ser entregados. Como verás, son prácticamente los mismos parámetros que para hacer cookies en PHP "nativo".

Nombre de la cookie (string) Valor (string) Minutos de persistencia (entero) Camino/ruta donde va a estar disponible (string) Dominio o subdominio de existencia (string) Si es segura, https (booleano) Si es solo insegura, no https (booleano)

Por defecto están disponibles para el subdominio donde se creó, y son sólo para http normal. Enviar una Cookie al navegador en el sistema de "Response"

Para enviar una Cookie al navegador se debe adjuntar a la respuesta. Esto lo tienes que hacer desde el sistema de Response, para lo que necesitas un objeto de la clase Response.

**Nota:** Los objetos de la clase Response los obtienes de varias maneras y es algo que ya explicamos en el artículo sobre [Responses en Laravel 5](#).

La clase Response tiene un método llamado `withCookie()` que adjunta un objeto Cookie en la respuesta de la solicitud HTTP. Así Laravel, al construir la respuesta que enviará al navegador será capaz de enviarle también la cookie adjuntada.

```
$response = response("Voy a enviarte una cookie");  
$response->withCookie($nueva_cookie);
```

Aquí estamos usando el helper `response()` para obtener el objeto de la clase Response de Laravel, luego estamos enviando en la respuesta la cookie `$nueva_cookie`. Obviamente, `$nueva_cookie` es una referencia a un objeto Cookie como los que acabamos de enseñar a crear.

Ahora puedes ver un sencillo ejemplo de creación de una cookie, procedimiento que hemos colocado en una closure del sistema routing, para facilitar las cosas.

```
Route::get('pruebacookie', function(){  
    $nueva_cookie = cookie('probando', 'valorprobando', 60);  
    $response = response("Voy a enviarte una cookie");  
    $response->withCookie($nueva_cookie);  
    return $response;  
});
```

## Lectura de una Cookie mediante el sistema de request

Ahora veamos cómo leer una cookie ya creada. Es algo que hacemos a través del sistema de Request. Podríamos leerla de varias maneras:

1. A partir de la "facade" (fachada) de Request.

```
Request::cookie('nombre_de_la_cookie');
```

**Nota:** Recuerda hacer el "use Request;" para tener acceso a esta fachada. Otra alternativa sería anteponer una contrabarra en Request para que Laravel sepa que te estás refiriendo al namespace global.

```
\Request::cookie('nombre_de_la_cookie');
```

2. Como alternativa, si hemos inyectado el correspondiente objeto de la clase Request en el método del controlador, tenemos acceso al método `cookie()` a partir de ese objeto:

```
public function personalizar(Request $request){  
    $cookie_leida = $request->cookie('nombre');  
    // resto del código del controlador  
}
```

Si la cookie que intentamos recibir no estaba en el navegador del usuario, este método devolverá `null` como valor de la cookie. Para evitarnos tener que comprobar si el valor es `null` y en ese caso establecer un valor predeterminado, podemos llamar al método `cookie()` indicando como segundo parámetro el valor a cargar por "default" (en caso de no exista la cookie).

```
Request::cookie('cookie_lectura', "valor_x_default");
```

## Dejar cookies en la cola de escritura

Ahora vamos a hablar de otro mecanismo interesante de trabajo con Cookies que nos relatan en la documentación de Laravel 5.0.

**Nota:** Es importante señalar que este mecanismo de encolado de la cookie ha sido borrado de la documentación de Laravel 5.1. Quizás lo han recolocado en algún otro lugar o quizás piensan eliminarlo del framework. Sin embargo, el código está probado y sigue funcionando perfectamente en la versión 5.1.

Puede darse el caso que quieras generar una cookie y todavía no se haya construido el objeto `response` y por cualquier motivo no lo desee hacer en este momento. En ese caso puedo dejar las cookies encoladas para su escritura en la siguiente respuesta (`Response`) que se genere.

Eso se consigue fácilmente con la fachada `Cookie` y el método estático `queue()`.

```
Cookie::queue($una_cookie_a_encolar);
```

**Nota:** Para que `Cookie` esté disponible recuerda hacer el "use `Cookie`";". O bien la alternativa de comenzar con una contrabarra, para indicar que esta clase se encuentra en el namespace global.

```
\Cookie::queue(\Cookie::make('cuco', 'cucurucu', 60 * 24 * 365));
```

## Redirigir con envío de cookie

En ocasiones podemos necesitar crear una cookie en un response que no muestre contenido, sino que redirija al usuario a otra página. Eso es perfectamente posible en Laravel, solo que no usarás un objeto response tal cual como hemos aprendido anteriormente, sino que lo harás con una instancia de un objeto "redirector".

El siguiente código crea una ruta que redirige a otra. En la redirección se añade una cookie usando el mismo método withCookie() que hemos invocado sobre un objeto Response, solo que ahora lo invocamos sobre el objeto Redirector que nos devuelve el helper redirect().

```
Route::get('redirigecookie', function(){
    return redirect('/otra/ruta/a/redirigir')->withCookie(cookie('redir', '1234', 20));
});
```

En este caso, como la cookie se genera en la Response que realiza la redirección, cuando el usuario accede a /otra/ruta/a/redirigir la cookie ya está disponible en el sistema.

**Nota:** No hemos hablado todavía en profundidad de redirecciones, pero lo haremos en breve. De todos modos, la cosa es bastante sencilla, si te apoyas en el helper redirect(), como habrás podido apreciar.

En el siguiente artículo vamos a seguir trabajando con Cookies pero de manera más práctica. Realizaremos un ejemplo completo en Laravel en el que haremos diversas cosas que nos ayudarán a repasar todo lo aprendido.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 30/10/2015  
Disponible online en <http://desarrolloweb.com/articulos/trabajo-cookies-laravel.html>

## Ejemplo completo de uso de cookies en Laravel

**Ejercicio práctico para ilustrar el uso de cookies en el framework PHP Laravel, así como de request y response, controladores, acciones, rutas, etc.**

En este ejemplo completo de uso de cookies vamos a practicar con muchas de las cosas que hemos visto hasta el momento en el [manual de Laravel 5](#). Nos servirá para conocer mejor las cookies, pero también para realizar rutas, controladores, vistas y formularios, recibir datos, validar campos, etc.

En realidad el ejemplo es bien sencillo, ya que lo interesante es practicar con el flujo de trabajo con el que

construir aplicaciones en Laravel, no tanto complicarse con formularios llenos de datos que recibir. Sin embargo, con lo que vamos a ver podremos construir muchos tipos de formularios ya personalizados con nuestras preferencias, con los datos que necesitemos recibir. Así que ahí vamos!



## Objetivo: Sistema de personalización de aspecto

El objetivo es hacer un **sistema de personalización del tamaño de la fuente en un supuesto sitio web**.

Básicamente tenemos un formulario donde el usuario puede seleccionar el tamaño de fuente que prefiere para visualizar el sitio. Una vez seleccionado el tamaño de la fuente **se envía el formulario y se guardará el dato en una cookie, de modo que en siguientes accesos el sitio web sea capaz de memorizar la preferencia del usuario**.

## Rutas de la aplicación

Tendremos dos rutas simplemente, una que muestra el formulario de personalización y otra que recibe el dato y si es correcto se encarga de almacenar la cookie.

```
Route::get('personalizacion', 'PersonalizacionController@personalizar');
```

```
Route::post('personalizacion', 'PersonalizacionController@guardarpersonalizacion');
```

Fíjate que la URI es la misma, pero cambian los verbos HTTP. La primera se activa cuando no se envían datos por formulario (verbo get) y la segunda cuando sí se reciben datos de formulario (verbo post). Como ves, ambas invocan al mismo controlador, con diferentes acciones.

Tienes más información de las rutas y dónde se coloca este código en el [artículo sobre rutas de Laravel](#).

## Controlador PersonalizacionController

Nuestro controlador se llama PersonalizacionController, como sabes se puede crear la base del controlador inicial con el comando artisan. Eso lo aprendimos ya en el [artículo de los controladores en Laravel](#). Así que nos vamos a limitar a colocar el código de las acciones que tenemos que crear dentro.

Comenzamos con la acción "personalizar" que es la que se invoca cuando se usa el verbo HTTP get. En esta acción vamos a tener que mostrar el formulario para personalizar, que permite al usuario escoger el

tamaño de la fuente.

```
public function personalizar(){
    $tamano_fuente = \Request::cookie('fuente', '16pt');

    return view('personalizacion.formulario', [
        'fuente' => $tamano_fuente
    ]);
}
```

Esta primera acción realmente solo llama a una vista. Lo que pasa es que la vista requiere un dato que es la fuente con la que queremos personalizar el sitio web, por lo tanto tenemos que pasárselo. Para recuperar el dato necesitamos acceder a la cookie de personalización. La primera vez que accede el usuario a la web esa cookie no estará creada, por lo que en ese caso tendremos que darle un valor por defecto a la fuente del sitio web.

Por tanto, en la primera línea del método accedemos al valor de la cookie a través de la fachada de Request. En la llamada al método `cookie()` enviamos el nombre de la cookie que queremos leer y el valor predeterminado en caso que esa cookie no exista.

**Nota:** Recuerda que estuvimos explicando todas estas cosas sobre cookies en el [artículo cookies en Laravel 5](#).

Luego mostramos la vista del formulario, enviando el dato que necesita la vista para mostrarse con la fuente personalizada.

La segunda acción es la que se procesa cuando se recibe alguna cosa del formulario de personalización. Como podrás apreciar en el siguiente código, hacemos una validación de la información que necesitamos para trabajar y luego nos encargamos de crear la cookie. Usamos el sistema de Response, pero en esta ocasión nos basamos en una redirección. Esto es así porque realmente cuando recibimos la personalización no queremos mostrar ningún mensaje en concreto, solo queremos devolver al usuario a la página anterior y en esa página se podrá apreciar la nueva fuente configurada para el sitio. El código es un poco más complicado que el del método anterior, pero seguramente lo podrás entender.

```
public function guardarpersonalizacion(Request $request){
    $this->validate($request, [
        'fuente' => 'required'
    ]);
    return redirect('/personalizacion')
        ->withCookie(cookie('fuente', $request->input('fuente'), 60 * 24 * 365));
}
```

El sistema de validación en este caso solo comprueba el campo de formulario "fuente". Ese campo es requerido. Sobre este punto encuentras más explicaciones en el [artículo de Validaciones en Laravel](#).

**Nota:** Hay que recordar que, según el código del controlador y gracias a la validación "required" del campo fuente, el código que hay a continuación de la llamada al método validate() solo se ejecutará cuando realmente se reciba la fuente a cambiar en el sistema de personalización.

Como habrás podido apreciar, se usa el helper redirect(), que explicaremos con detalle en un futuro artículo. Ese helper nos devuelve un objeto response especial llamado "Redirector", a partir del cual se crea la Cookie (pues las cookies solo se pueden crear cuando se devuelven datos al cliente en el response).

Al objeto Redirector le indicamos que se genere la cookie donde se almacena la fuente. Esa cookie tiene como nombre "fuente", como valor el dato recibido en el input y como duración 1 año expresado en minutos.

**Nota:** Sobre Response aprendimos en el artículo de Response HTTP en Laravel. Sobre las Cookies en el artículo anterior Cookies en Laravel.

## Vistas de la aplicación

Para acabar nos faltaría mostrar el código de la vista que hemos creado para esta práctica. Es la parte más sencilla porque básicamente es código HTML. Es una vista de tipo Blade, por lo que tendrá extensión .blade.php (ya que hemos usado sintaxis blade para embutir datos que recibimos en la vista en el código HTML {{{variable}}}). Vimos más sobre vistas en el artículo de [introducción a las vistas en Laravel](#).

Esta vista es la del formulario:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Personalizacion</title>
  <style>
  body{
    font-size: {{{fuente}}};
  }
  </style>
</head>
<body>
  <p>fuente: {{{fuente}}}</p>
  <form action="/personalizacion" method="post">
    Fuente:
    <select name="fuente">
      <option value="24pt">Grande</option>
      <option value="16pt">Mediana</option>
      <option value="12pt">Pequeña</option>
    </select>
    <br />
    <input type="submit" value="Enviar">
  </form>
</body>
</html>
```

```
</form>
</body>
</html>
```

Observa que en la declaración de estilos, etiqueta STYLE en el HEAD, usamos la variable de la fuente para personalizar el tamaño de letra del BODY. Luego hay un párrafo ya en el cuerpo de la página donde usamos esa misma variable que nos mandan a la vista `{{ $fuente }}`. El resto es un formulario HTML de toda la vida.

Con eso es todo, si has seguido con atención el [manual de Laravel](#), estamos convencidos que podrás reproducir este ejemplo y mejorarlo por tu cuenta con adicionales opciones de personalización de estilos en la página.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en *05/11/2015*  
Disponible online en <http://desarrolloweb.com/articulos/ejemplo-completo-uso-cookies-laravel.html>

# Bases de datos con Laravel 5.1

Las bases de datos son el corazón de las aplicaciones web y el modo de trabajo con bases de datos en Laravel dista bastante del modo de trabajo con PHP básico, así que los desarrolladores tendremos que aprender muchas cosas nuevas cuando nos lanzamos a este framework. Como ventaja muy representativa en Laravel, tenemos la posibilidad de trabajar con bases de datos a diferentes niveles y de diferentes formas, de más alto o bajo nivel, por lo que estamos seguros que cada proyecto y cada desarrollador tendrá un modo que se adapte a sus necesidades, aunque a veces tendremos que combinarlos en una misma aplicación si fuera necesario. Tendremos que aprender también a usar las migraciones, una herramienta fundamental para un sencillo trabajo de actualización y mantenimiento del modelo de base de datos de una aplicación web. Todo esto y más lo iremos aprendiendo en los próximos puntos.

## Bases de datos con Laravel

**Introducción a las bases de datos con el framework PHP Laravel, qué es Eloquent, motivos por los que existe y primeros pasos.**

Con este artículo comenzamos a introducirnos en el mundo de las bases de datos en Laravel, componentes indispensables para cualquier aplicación web por sencilla que sea. En Laravel tenemos varios caminos válidos para trabajar con bases de datos, de más o menos alto nivel, para cubrir todas las necesidades de proyectos, situaciones o perfiles de desarrolladores.

A lo largo de este manual podremos aprender a trabajar con bases de datos, aunque el tema es amplio como para poderlo ver todo de golpe. De momento, los objetivos que nos planteamos en este artículo son dos:

1. Proporcionar una vista de pájaro sobre las vías herramientas y existentes para trabajar con base de datos, sin abordar detalladamente ninguna de ellas.
2. Que el estudiante sepa ubicar perfectamente cada una de las piezas de software con las que se va a trabajar en el acceso a bases de datos.

**Nota:** Para los más observadores, cabe aclarar que hemos pasado por alto en el [Manual de Laravel](#) muchas secciones importantes que veremos con detalle más adelante, como los redirects, envío de email, vistas con Blade, etc. pero no hay que preocuparse porque las iremos retomando.

El motivo de comenzar ahora las bases de datos es nuestra preocupación por hacer ejemplos más interesantes, en los que podamos practicar en condiciones similares a las que encontrarás en un proyecto real. Debido que cualquier proyecto en el mundo de las aplicaciones web pasa por usar alguna base de datos, preferimos introducirnos ahora este asunto.



## Cómo se gestiona el trabajo con bases de datos en Laravel

En Laravel existen diversas vías para trabajar con bases de datos, a distintos niveles. Principalmente podremos trabajar con:

**Nota:** Cuando decimos "a distintos niveles" nos referimos a lo mismo que se refieren los lenguajes de alto y bajo nivel. Los de alto nivel son más cercanos al lenguaje de las personas y los de bajo nivel más cercanos a la máquina. En este caso son distintos niveles con respecto al sistema gestor de base de datos. Raw Sql es el de más bajo nivel y Eloquent es el de más alto nivel.

- **Raw SQL:** consultas nativas de nuestro sistema gestor de bases de datos.
- **Fluent Query Builder:** un sistema de construcción de las consultas por medio de código, independiente del sistema gestor de bases de datos usado, y sin tener que escribir a mano el código SQL.
- **Eloquent ORM:** un ORM avanzado, pero sencillo de usar, para trabajar con los datos como si fueran objetos y abstraernos de que realmente estén almacenados en tablas de la base de datos.

El desarrollador es el responsable de saber de qué manera le interesa trabajar para cumplir mejor sus objetivos. Generalmente, en la mayor parte de los casos, trabajar con Eloquent será más sencillo, por lo que habitualmente será la solución preferida. Sin embargo, las necesidades de nuestro proyecto, ya sea por complejidad del modelo de negocio o por buscar una mejora del rendimiento, podremos preferir opciones de más bajo nivel como Query Builder o incluso Raw SQL.

## Bases de datos compatibles

Laravel trabaja con diversos motores de base de datos por debajo. Son los siguientes:

- MySQL
- Postgres
- SQLite
- SQL Server

La elección del sistema de base de datos corre por nuestra cuenta y aunque sean motores diferentes, a la hora de la práctica el tratamiento que haremos, a nivel de código, para trabajar con cualquiera de ellos será el mismo, si trabajamos en las capas de más alto nivel como el query builder o el ORM.

## Configuración de la base de datos

El primer paso que tendremos que hacer para trabajar con una base de datos es configurarla en nuestro sistema. Esto se realiza desde un par de lugares.

**Archivo .env:** Este archivo se encuentra en la carpeta raíz del proyecto y contiene información sobre el entorno. Son básicamente una serie de variables de entorno que tienen entre otras información sobre la base de datos. Este archivo de entorno podrá en un mismo proyecto tener valores distintos, en distintas máquinas, por ejemplo cuando está en un servidor en producción y uno en desarrollo.

**Nota:** Como te podrás imaginar, este archivo no se suele incluir en el sistema de control de versiones, porque cada máquina tendrá sus variables de entorno y no se desean compartir entre unas y otras. Por eso, cuando usas Git, lo más normal es que lo metas en el archivo "gitignore".

**Archivo config/database.php:** Es un archivo donde se definen los valores de conexión de la base de datos. Muchos de ellos los toma directamente de las definiciones del archivo .env (el entorno), pero además se definen asuntos como el driver, codificación, etc. Si tuviéramos varias conexiones con base de datos deberíamos definirlas todas en este archivo.

**Nota:** Si has instalado Laravel mediante Homestead la inmensa mayoría de las configuraciones ya están realizadas. En los archivos .env y config/database.php no tendrás necesidad de editar nada. Si has instalado Laravel por otros medios tendrás que configurarlos a mano. No obstante, aunque Homestead te ahorre este paso, merece la pena echarle un vistazo a ambos ficheros para ver cómo y dónde se especifican todos los valores de conexión.

Los valores de conexión con MySQL predeterminados que tenemos en Homestead son los siguientes:

- Host: localhost
- Usuario: homestead
- Clave: secret
- Base de datos: homestead

Aunque esta información la puedes ver y confirmar en el archivo .env de la carpeta raíz de tu instalación de Laravel.

## Modelos en Laravel

Los modelos son las clases que tienen el código donde se realizará el acceso a los datos, por tanto, los accesos a las bases de datos son responsabilidad de los modelos. En Laravel se han diseñado para que funcionen con muy poco código y para ello el programador debe respetar una serie de convenciones.

Esas convenciones son las que permiten que podamos ahorrar código. Si no las respetamos simplemente nos veríamos obligados a escribir algo de código para adaptar los modelos a nuestras circunstancias. Estas reglas son importantes para comenzar:

- El nombre del modelo tiene que ser en singular y mayúscula y la tabla a la que acceden estará en plural y minúscula. Por ejemplo: modelo "User" para tabla "users". Modelo "Product" para tabla "products".
- Por defecto busca siempre la llave primaria con el nombre de campo "id".
- Eloquent usa las columnas `created_at` y `updated_at` para actualizar automáticamente esos valores de tiempo.

**Nota:** Obviamente, hay mucha más información que tendremos que abordar sobre los modelos, que veremos más adelante. No obstante recuerda que tuvimos una [introducción a los modelos en Laravel](#) en la que ofrecimos más información y vimos algo de código.

## Migraciones y seeders

Otra cosa a la que tendremos que adaptarnos cuando trabajamos con Laravel, y que a corto y largo plazo nos ahorrará muchas tareas de mantenimiento cuando trabajemos con bases de datos, son las migraciones. En Laravel no defines tú las tablas directamente con SQL o un programa tipo PhpMyAdmin o MySQL Workbench. En lugar de ello se crean unos archivos con código de clases llamados migraciones.

Con las migraciones puedes definir la creación de tablas, su modificación, etc., llevando un registro de cada uno de los cambios que se han producido en el esquema de la base de datos. Con los seeders puedes incluso crear datos de prueba para poder poblar las tablas. Quizás podría ser más rápido aplicar todos esos cambios con un programa de acceso a las bases de datos, sin embargo la potencia de las migraciones las hacen especialmente útiles en varias situaciones:

- Cualquiera puede actualizar la forma de la base de datos o los datos en sí y compartir los archivos de migraciones con el resto del equipo, de modo que todos se puedan sincronizar y tener la misma versión de la base de datos.
- Cualquier migración se puede volver para atrás, de modo que nos sirven como una especie de control de versiones de la forma o definición de nuestra base de datos.
- Realiza una abstracción de la base de datos en cuanto a la creación y modificación de su forma. De modo que, si se cambia el motor, las migraciones crearían la misma base de datos en el nuevo sistema gestor.
- Facilita mucho la implantación de un sitio web en un nuevo servidor o en un nuevo componente del equipo de trabajo.

Sobre todo es clave el tema de la sincronización con todos los componentes de un equipo de trabajo. Si alguien cambia una tabla no necesita pasar el create table nuevo y asegurarse que todos los componentes lo apliquen a mano. Será solo dejar el archivo de las migraciones en el sistema de control de versiones usado para el proyecto (git o el que sea), todos los desarrolladores se sincronizan y ejecutan un comando para procesar las migraciones. Sin embargo, aunque trabajemos solos, las migraciones también nos ayudarán a ser más productivos.

En el próximo artículo comenzaremos a explorar a fondo las bases de datos en Laravel analizando con detalle el [tema de las migraciones](#).

## Conectar con un software profesional de administración de bases de datos

Hemos dicho que la definición de las tablas la realizaremos por medio del sistema de migraciones. También la carga de datos de prueba, sin embargo, es útil que durante el desarrollo podamos comprobar en las tablas si las operaciones que se realizan mediante la aplicación han tenido los cambios esperados. Por ello seguimos necesitando algún software para el acceso a la base de datos a bajo nivel.

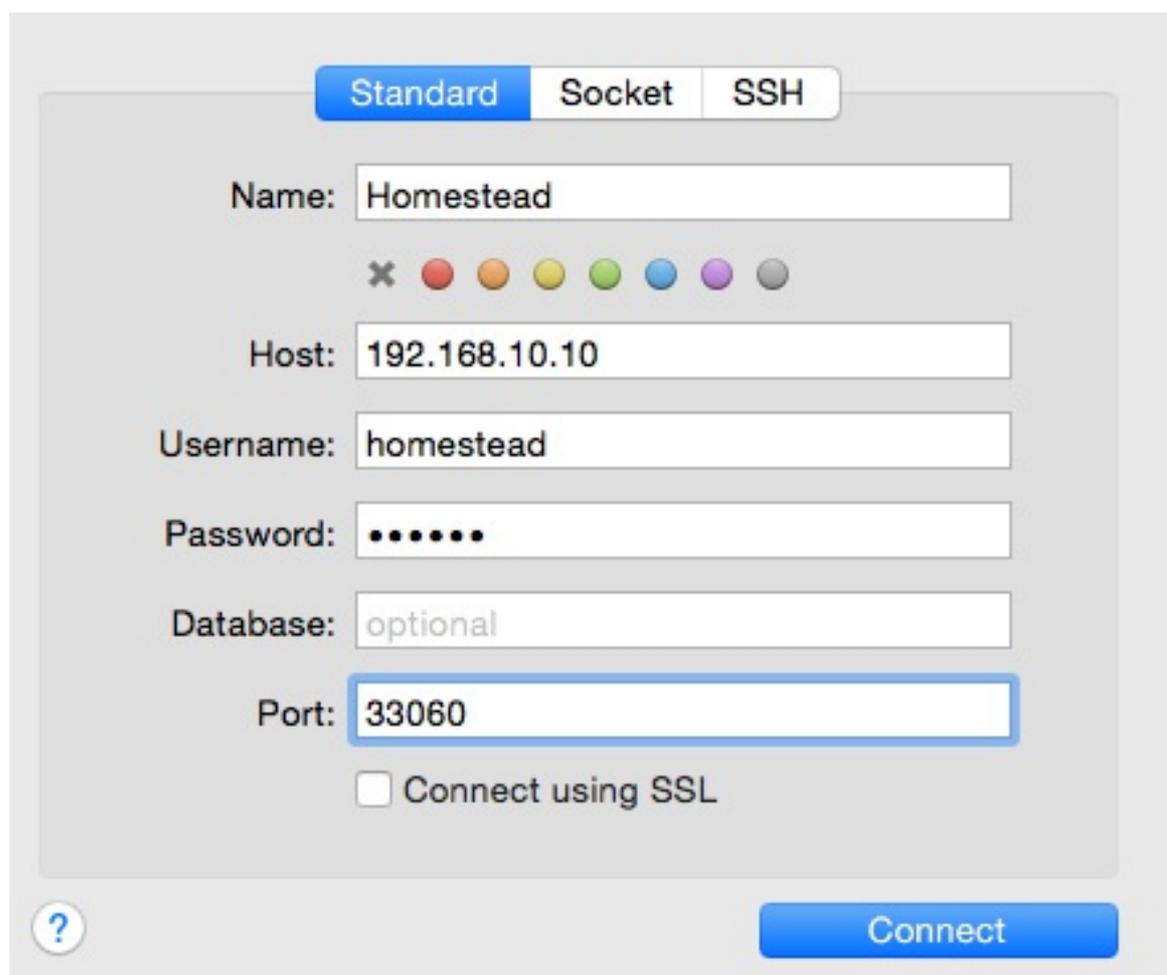
Puedes acceder de varias formas, pero lo ideal es que vayas familiarizándote con algún software profesional de administración de bases de datos, como Sequel Pro, MySQL Workbench, etc.

**Nota:** Otro método sería conectar con MySQL, u otro sistema gestor de base de datos que estés usando, a través de línea de comandos. Por ejemplo, para conectar con MySQL por el terminal ya explicamos cómo hacerlo en la Introducción a los modelos de Laravel.

El mecanismo para conectar será diferente dependiendo del programa. **Por ejemplo en Sequel Pro haríamos lo siguiente.** "File / New Connection Window" y luego en la ventana de conexión usamos los valores:

- **Name:** Homestead (o lo que queramos)
- **Host:** 127.0.0.1 (Ojo en este punto, porque la IP de nuestro homestead es 192.168.10.10, o aquella que hayamos asignado a la máquina virtual Homestead. En mi caso he conseguido conectar por esta IP en algunas ocasiones, pero no siempre. Con la IP de localhost 127.0.0.1 consigues conectar también, solo ten cuidado no estés conectando a una instalación que tengas en local de MySQL, si es que lo tienes instalado.)
- **Username:** homestead
- **Password:** secret
- **Port:** 33060

Opcionalmente podemos indicar ya la base de datos que ha sido creada, llamada "homestead". Luego pulsamos el botón de "Connect". Además puedes guardar esa conexión para no tener que escribir los datos más adelante con el botón "Add to Favorites".



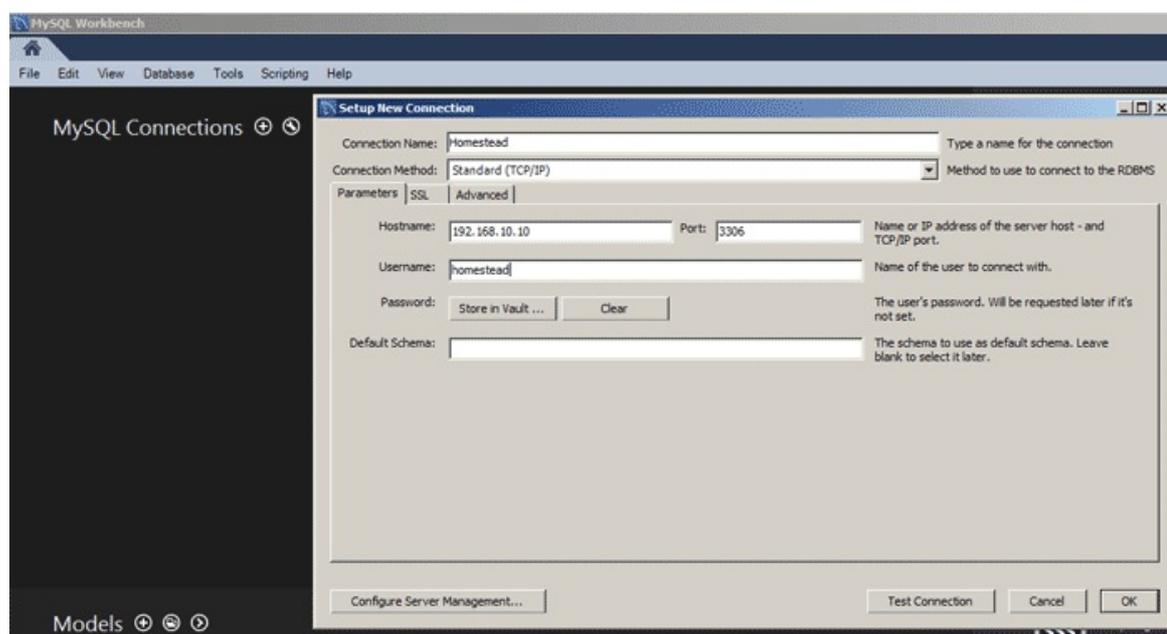
The image shows a MySQL connection configuration window with the following fields and options:

- Tab: Standard (selected), Socket, SSH
- Name: Homestead
- Host: 192.168.10.10
- Username: homestead
- Password: masked with dots
- Database: optional
- Port: 33060
- Connect using SSL
- Buttons: ? (help), Connect

**Nota:** Para que la conexión funcione ten en cuenta que la máquina virtual Homestead debe estar arrancada. Osea, haber hecho el "vagrant up" desde la carpeta donde tienes el Vagrantfile.

Los usuarios de Windows pueden optar por MySQL Workbench que también es gratuito, es multiplataforma y su acceso es bastante similar. Los datos de conexión son los mismos y las ventajas de la aplicación para realizar el enlace con la base de datos difieren poco.

En MySQL Workbench hacemos una nueva conexión desde la pestaña "Home" (con una casita en la parte de arriba y a la izquierda), luego pulsando el botón "+" al lado de "Connections". Nos aparece una ventana donde configuramos la conexión.



Connection name será lo que deseemos. El connection method será "Standard (TCP/IP)" y luego los parámetros de conexión los ya conocidos, 127.0.0.1 (o quizás 192.168.10.10 si es tu caso) para el "Hostname", homestead para "Username" y listo.

Si no ponemos la clave en la primera ventana de conexión nos aparecerá una nueva ventana más tarde para indicarla.



Como decimos, desde estos programas en principio nos limitaremos a observar cómo han ido las migraciones al ejecutarse o volverse hacia atrás, así como la intervención de los modelos para modificar los datos de la aplicación. No obstante, si lo preferimos, podríamos usar también estos programas para definir las bases de datos, en cuyo caso perderíamos las ventajas de trabajar con el sistema de migraciones.

En el siguiente artículo entraremos de lleno en la parte práctica y conoceremos el sistema de migraciones para poder comenzar a crear las nuestras.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en 18/11/2015  
Disponible online en <http://desarrolloweb.com/articulos/bases-datos-laravel.html>

## Migraciones en Laravel

Trabajo con bases de datos y el sistema de migraciones para definir tablas y modificarlas a través del código de diferentes clases.

Las migraciones son como un control de versiones del estado de nuestras tablas. Están preparadas para ejecutarse en "ida y vuelta". Osea, podemos correr el sistema de migraciones para obtener la base de datos en su estado final, así como también podríamos volver hacia atrás, para recuperar cualquier estado de la base de datos hasta llegar al estado inicial, en el que se supone que no habría tabla alguna.

En el artículo anterior del [Manual de Laravel 5](#) estuvimos haciendo una [introducción a las migraciones en particular y al tema de las bases de datos en Laravel en general](#), por lo que no vamos a repetir lo mismo. Suponemos que el concepto se entiende, así como sus ventajas, y lo que queremos ahora es trasladarlo al mundo de lo concreto.

Las migraciones están dentro de la carpeta database/migrations y en la instalación de base de Laravel ya incorporan un par de archivos de migraciones, por lo que podemos observar cómo se han construido. Son un par de archivos para creación de la tabla de usuarios "users" y la tabla "password\_resets", componentes del sistema de login de usuarios de Laravel.

En breve veremos cómo crear estos archivos de migraciones con nuestro propio código, pero antes vamos a aprender algunos comandos para operar con estas migraciones.



### Comandos "artisan" para ejecutar las migraciones ya creadas

Ya que la instalación de Laravel viene con un par de migraciones, vamos a jugar un poco con ellas para empezar, ejecutándolas y volviendo hacia atrás.

El asistente de comandos artisan tiene una serie de instrucciones para trabajar con las migraciones. Como siempre, podemos encontrar la lista de comandos disponibles en artisan con "php artisan".

En la sección "migrate" encontramos varios items que nos interesarán para trabajar con estas primeras migraciones que ya vienen creadas en el framework.

**migrate** Este comando sirve para ejecutar todo el sistema de migraciones y poner en marcha cada una de las migraciones generadas en el proyecto. Crea el repositorio de migraciones y luego las ejecuta una por una para generar toda la estructura de la base de datos y los datos de prueba que puedan haberse cargado.

**Nota:** Estos comandos los ejecutamos desde la línea de comandos del servidor. Si estás usando Homestead, los ejecutas una vez conectado por ssh con la máquina virtual, desde la raíz de tu proyecto. Como ya se ha mencionado en alguna ocasión los comandos artisan necesitan del intérprete de PHP, por lo que la sintaxis para invocarlos es la siguiente:

```
php artisan migrate
```

Prueba a ejecutar ese comando y verás como se generan tres tablas en tu sistema. Una de ellas llamada "migrations" sirve de control del sistema de migraciones. Luego verás otras dos tablas, que son las tablas de la aplicación para controlar a los usuarios y los resets de claves.

**migrate:install** Sirve para crear el repositorio de migraciones. Lo que hace es crear una tabla en la base de datos que sirve para llevar el control de las migraciones realizadas y su orden. Su ejecución se realiza con el comando completo "php artisan migrate:install". Este comando va implícito en el comando anterior, migrate.

**migrate:reset** Realiza una marcha atrás de todas las migraciones en el sistema, volviendo el schema de la base de datos al estado inicial.

**migrate:refresh** Vuelve atrás el sistema de migraciones y las vuelve a ejecutar todas.

**migrate:rollback** Vuelve hacia atrás la última migración realizada. Una migración puede incluir varias tablas que fueron migradas de una sola vez en el paso anterior.

**migrate:status** Te informa de las migraciones presentes en el sistema y si están ejecutadas o no.

En la siguiente imagen puedes ver un reporte de status de migraciones, con algunas migraciones ejecutadas y otras no. No corresponde con las migraciones que tendrás ahora en tu sistema, ya que no hemos creado ninguna nueva aparte de las dos que venían originalmente en Laravel, pero dentro de poco vamos a crear esas migraciones que estás viendo en el status de la imagen:

```
+-----+-----+
| Ran? | Migration |
+-----+-----+
| Y    | 2014_10_12_000000_create_users_table |
| Y    | 2014_10_12_100000_create_password_resets_table |
| N    | 2015_09_05_003208_create_posts_table |
| N    | 2015_09_05_171742_articles_create_table |
+-----+-----+
```

Puedes probar estos comandos con total garantía que no vas a romper nada, ejecutando las migraciones y volviendo hacia atrás, consultando el "status" en cada paso. Eso te ayudará a entender mejor las mecánicas de las migraciones y a empezar a apreciar la potencia que hay detrás de este sistema.

## Crear una nueva migración

Ahora vamos a crear nuestra primera migración, para agregar una nueva tabla en la base de datos. Como en otras ocasiones, el esqueleto de nuestra migración lo podemos crear fácilmente ayudados por un comando artisan: `make:migration`.

```
php artisan make:migration create_posts_table
```

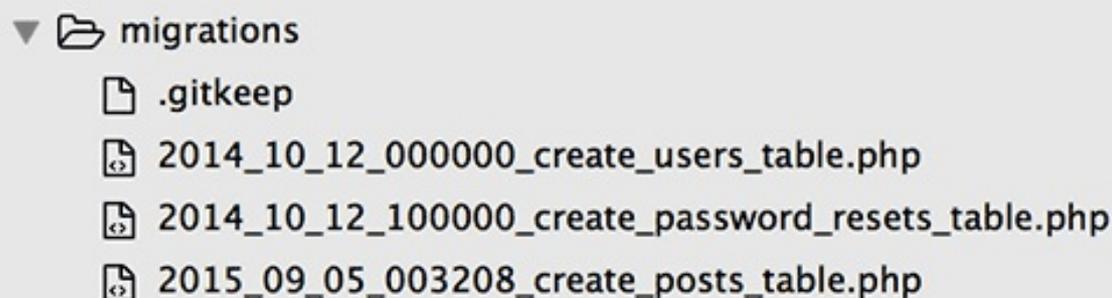
Habitualmente en el nombre de las migraciones indicas qué tipo de operación estás realizando al ejecutar esa migración. En este caso el nombre de la migración era "create\_posts\_table", osea, el "create table" para la entidad "post".

Además, a este comando le podemos pasar un parámetro adicional para indicar a artisan el nombre de la tabla que está relacionada con la migración, que servirá para que lo incluya en el código esqueleto de la clase para esta migración.

```
php artisan make:migration create_posts_table --table=posts
```

Ahora vamos a la carpeta donde se encuentran las migraciones: "database/migrations". Allí encontrarás el archivo de nuestra nueva migración. Aprenderás que todos los archivos de las migraciones tienen, en su nombre, un timestamp que ayudará a Laravel a saber en qué orden deben ejecutarse. En mi caso tiene este nombre de archivo:

```
2015_10_05_003208_create_posts_table.php
```



Obviamente, cuando generes tu migración se le asignará otro timestamp, acorde con el instante en el que se creó.

Puedes abrir el archivo ahora. El código te permitirá identificar rápidamente que todas las migraciones son hijas de la clase "Migration" (extends Migration). Además encontrarás dos métodos ya generados que ahora te explicamos.

## Métodos para procesar una migración o volver hacia atrás

El corazón de las migraciones, y aquello que tendrás que definir mediante código cuando las estás creando, son dos métodos que realizan la tarea de ejecutar la migración o volverla hacia atrás.

**up()** Este método se ejecutará cuando la migración se corra, así que tiene todo el código necesario para

hacer las modificaciones en el esquema de la base de datos. Por ejemplo, si estás creando una migración porque necesitas una nueva tabla, este método tendrá el código para crearla.

**down()** Este método se ejecutará cuando se vuelva hacia atrás y se tengan que desechar los cambios producidos por esta migración. Tendrá el código necesario para volver hacia atrás y recuperar el estado anterior del schema de la base de datos. Si la migración se hiciera para crear una tabla, en este método pondrías el código para eliminarla.

## Código para realizar cambios en la definición de la base de datos

En los métodos `up()` y `down()` colocas código PHP que se encargue de realizar los cambios pertinentes en la definición de la base de datos. Podrás hacer cosas como crear o eliminar tablas, cambiar su nombre, cambiar sus campos poniendo o quitando columnas, crear o eliminar índices, claves, etc. Sin embargo, para realizar todas estas operaciones no vas a trabajar con el lenguaje SQL del sistema gestor de la base de datos que hayas escogido, sino con un API de funciones de Laravel.

El API para hacer migraciones tiene la ventaja de ser común para cualquiera de los sistemas gestores de base de datos para los que Laravel es compatible, por lo que el código de migraciones será perfectamente válido para cualquier base de datos. Incluso si a mitad de proyecto se cambia el sistema gestor, tus migraciones seguirán sirviendo.

En este artículo vamos a hacer un ejemplo de código para crear una tabla, y su correspondiente "rollback". En futuros artículos revisaremos otros tipos de operaciones de modificación del schema, pero de momento puedes verlos en la propia [documentación de Laravel "Writing Migrations"](#).

Si queremos crear la migración para incorporar la tabla "posts", comenzaremos por definir el código para crear dicha tabla en el método `up()`. Las operaciones de alteración del schema de la base de datos se hacen a través de la fachada "Schema", por lo que básicamente dentro de este método realizaremos uso de esa fachada.

Para crear una tabla usamos el método estático `create()` de la fachada Schema, cuya invocación tendrá esta forma:

```
Schema::create('posts', function (Blueprint $table) {  
    // código para definir esta tabla  
});
```

Como se ve, el método requiere el nombre de la tabla que se va a crear y luego una función anónima (closure) que contiene el código para definir los campos de la tabla. Esta función recibe un parámetro de la clase Blueprint que tiene los métodos que necesitarás para definir la tabla.

**Nota:** Ese objeto de clase Blueprint nos lo inyecta Laravel en el closure, no tenemos que hacer nada para enviarlo a la función.

Como código para definir la tabla, ayudados por el objeto `$table`, de la clase Blueprint, realizaremos la

declaración de cada uno de los campos, usando métodos como estos:

- `increments()` genera un campo auto-increment.
- `string()` genera un campo de tipo cadena (VARCHAR).
- `text()` genera un campo largo de cadena (TEXT).
- `integer()` genera un campo de tipo entero.
- `boolean()` genera un campo booleano.
- `timestamp()` genera un campo timestamp.
- `timestamps()`, observa que está en plural, genera las columnas `created_at` y `updated_at` para que Laravel lleve el control de estos campos automáticamente.
- ...

Tienes decenas de tipos de columnas. Insistimos que los tipos de columnas no dependen del sistema gestor, sino que forman parte del API de migraciones. Por tanto, Laravel será encargado de definir el campo con el tipo nativo de la base de datos usada que mejor considere. Tienes la [lista completa de tipos de columnas en la documentación de Laravel](#).

El código de ejemplo para la creación de la tabla de usuarios sería el siguiente:

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->increments('id');
        $table->string('titulo', 100);
        $table->text('cuerpo');
        $table->integer('visitas');
    });
}
```

Hemos indicado una serie de campos variados. Observa que no hemos dicho por ningún lado que el campo "id" debe de ser considerado como índice o clave primaria, dado que Laravel ya lo sobreentiende, tal como comentamos en el artículo anterior.

Ahora vamos a implementar el método `down()`, que debe hacer el paso contrario que `up()`, en este caso eliminar la tabla "posts". Es muy sencillo:

```
public function down()
{
    Schema::drop('posts');
}
```

En este caso la operación de borrado de una tabla, método `drop()` de la facade "Schema", solamente requiere el nombre de la tabla que se debe eliminar.

Ahora podrás realizar los comandos para migrar y volver atrás en esta migración, que por si no te acuerdas son:

Ejecutar las migraciones pendientes, que como acabamos de crear una, te la pondrá en marcha. "php artisan

migrate" Volver atrás, deshaciendo la última migración. "php artisan migrate:rollback"

En cada paso detente a observar la base de datos y podrás ver cómo se genera y se destruye la tabla al trabajar con el sistema de migraciones.

De momento es todo. Con lo que hemos visto podrás practicar varias cosas con el sistema de migraciones. En futuros artículos haremos nuevos ejemplos para abarcar más posibilidades.

Este artículo es obra de *Carlos Ruiz Rufo*  
Fue publicado por primera vez en 10/12/2015  
Disponible online en <http://desarrolloweb.com/articulos/migraciones-laravel.html>

## Tratamiento de índices y claves al escribir migraciones en Laravel

**Cómo trabajar con claves, primarias, foráneas, así como índices al escribir las migraciones para definición de la base de datos en Laravel.**

Seguimos con el tema de las migraciones en Laravel 5, profundizando acerca de los índices y la definición de claves en el código de las migraciones. Hay varios métodos y mecanismos que debemos de conocer para poder completar nuestros conocimientos para poder escribir migraciones.

Ya sabemos que los índices nos sirven a la hora de definir el esquema de la base de datos, permitiendo que el sistema gestor nos ayude optimizando las tablas y mejorando la velocidad de acceso a los datos, o permitiendo asuntos como que no se repitan los valores en una columna. Hay varios tipos de índices y claves que nos imaginamos que llegados a este punto conoces de antemano, así que vamos directamente a ver cómo trabajar con estos elementos en las migraciones.

The image shows the word "Laravel" in a white, sans-serif font, centered on a dark red background.

### Claves primarias

Los tipos de índices más comunes son los que se crean cuando se definen las claves primarias. **En Laravel las claves primarias se definen automáticamente sobre los campos definidos como "increments"**. Este es un detalle que ocurre sin que tengamos que intervenir o especificar nada en las migraciones, no obstante, si no tenemos un campo autoincremental, se puede definir la clave de otra forma.

```
Schema::create('facturas', function (Blueprint $table) {
```

```
$table->increments('id');  
$table->date('fecha');  
});
```

Esta tabla tendrá como clave primaria el campo "id". Pero si al campo le hubiésemos llamado "id\_factura" por ejemplo, también sería definida como clave primaria por el hecho de haber sido definido con "increments".

**Nota:** Existe un método alternativo para crear campos auto-increment, pero en este caso con posibilidad de llegar a valores de enteros muy grandes.

```
$table->bigIncrements('id');
```

Si queremos definir como clave primaria otro campo, que no sea un incremental, podemos realizarlo con el método `primary()`.

Tendríamos dos maneras de hacer esto, encadenando el método `primary()` a la definición del campo.

```
$table->char('codigo', 25)->primary();
```

O definiendo el `primary()` una vez el campo está creado, en instrucciones diferentes.

```
$table->char('codigo', 25);  
$table->primary('codigo');
```

Por último, **podemos también crear claves primarias con una reunión de campos**, simplemente indicando los campos sobre los que se genera el índice en un array que se enviará al método `primary()`.

```
$table->char('area', 25);  
$table->char('bloque', 2);  
$table->primary(['area', 'bloque']);
```

## Claves foráneas

En la realización de migraciones mediante Laravel existe también la posibilidad de especificar claves foráneas (foreign key), que se usarán para asegurar la integridad referencial de los datos.

Por ejemplo, podemos tener la clave `id_cliente` en la tabla de factura, que corresponde con el campo `id` de la tabla clientes. Entonces podemos definir la clave de esta manera:

```
$table->integer('id_cliente')->unsigned();
```

```
$table->foreign('id_cliente')->references('id')->on('clientes');
```

Aquí hay dos detalles que merece la pena remarcar. Primero, al construir el campo `id_cliente`, de tipo `integer`, le hemos indicado el modificador `unsigned()`. Es porque los campos de identificador son siempre sin signo, así que la columna `id_cliente`, para hacerla corresponder con el `id` de la tabla `clientes`, debe ser también `unsigned`. El otro detalle es en la llamada al método `foreign()`, observar como se encadenan una serie de métodos, para decirle que referencia al "id" que hay en la tabla "clientes". Creo que el código es suficientemente auto-explicativo.

**Nota:** Para poder hacer una referencia en una foreign key a otro campo en otra tabla, tal campo en esa tabla debe de existir. Osea, para que funcione el ejemplo anterior de migración debería haber una tabla "clientes" con un campo "id".

Aquí hay otro asunto interesante, los "delete cascade", que trataremos de manera práctica más adelante, pero no queremos desaprovechar la oportunidad de comentarlos. Como se puede saber sirve para borrar elementos relacionados en claves foráneas cuando se produce un delete. Simplemente se encadena una llamada a un método adicional `onDelete('cascade')`:

```
$table->foreign('id_cliente')->references('id')->on('clientes')->onDelete('cascade');
```

## Escribir otros índices en las migraciones

Existe la posibilidad de crear otros índices, ya no relacionados con claves de una tabla, pero que nos permitan realizar búsquedas optimizadas o controlar los duplicados.

Los índices básicos se crean con el método `index()`. Por ejemplo, en la tabla de facturas queremos hacer consultas por fecha y queremos crear un índice en ese campo.

```
$table->date('fecha')->index();
```

Los índices únicos se crean con el método `unique()`. Por ejemplo en la tabla de clientes, el email podría ser único.

```
$table->string('email', 200)->unique();
```

Estos dos métodos también pueden invocarse una vez creado el campo en la tabla. Por ejemplo:

```
$table->date('fecha');  
$table->index('fecha');
```

## Eliminar índices creados

Podemos querer eliminar índices creados con anterioridad, para deshacer una migración en la que creamos el índice (cuando definimos el método `down()` del `rollback`) o bien para quitar un índice que ya no necesitamos en la base de datos.

Los cuatro métodos para eliminar índices, para los 4 tipos de índices que hemos visto, son:

- `dropIndex('índice')` elimina un índice básico
- `dropUnique('índice')` elimina un índice único
- `dropPrimary('índice')` elimina una clave primaria
- `dropForeign('índice')` elimina una clave foránea

Cualquiera de estos métodos recibe un único parámetro, que es el nombre del índice que se debe eliminar. Ese nombre lo crea automáticamente Laravel al ejecutarse el sistema de migraciones y está compuesto por varios segmentos separados por un guión bajo:

- Nombre de la tabla
- Campo o columna sobre la que se crea el índice
- Tipo de índice

Nosotros podemos componer el nombre del índice fácilmente juntando esos datos, algo como "facturas\_fecha\_index", "clientes\_email\_unique" o "facturas\_id\_cliente\_foreign". Pero si no damos con el nombre exacto, podemos ver los nombres asignados a los índices directamente con un programa de acceso a bases de datos, tipo PhpMyAdmin, MySQL Workbench o Sequel Pro. Asimismo, lo podemos hacer desde el terminal, conectando con el intérprete de MySQL o nuestro motor escogido y lanzando esta sentencia SQL:

```
show index from nombre_de_tabla;
```

De momento eso es todo, con lo que sabes deberías poder crear todo tipo de migraciones. Te recomendamos buscar en la documentación oficial otras informaciones o referencias.

Este artículo es obra de *Carlos Ruiz Ruso*  
Fue publicado por primera vez en *14/12/2015*  
Disponible online en <http://desarrolloweb.com/articulos/indices-claves-migraciones-laravel.html>

## Seeders en Laravel 5

**Conoce los seeders, para alimentar una base de datos y crear datos de prueba o configurar el estado inicial de las tablas para un proyecto.**

Seguimos con asuntos relacionados con las bases de datos en el Manual de Laravel 5. En esta ocasión vamos a hablar de los "seeders". Seed es "semilla" en inglés y "seeders" serían algo así como "sembradores", aunque prefiero traducir como "alimentador", ya que sería como yo llamaría a estos sistemas corrientemente.

Los seeders no son más que componentes del framework Laravel que sirven para inicializar las tablas con datos. Así como las migraciones nos permiten especificar el esquema de la base de datos, los seeders nos permiten también por medio de código alimentar las tablas con datos.

Su uso, como decimos puede ser para:

1. Crear datos de prueba con los que trabajar durante el desarrollo de la aplicación
2. Configurar el estado de las tablas que necesita nuestra aplicación para comenzar a trabajar



Para aclarar este segundo caso piensa en el ejemplo de los artículos. Cuando subes un artículo puede tener varios estados: "borrador", "publicado", "borrado", etc. Puede que esos estados estén definidos en una tabla, así luego puedes crear otros estados si lo necesitas, como "obsoleto". Pero quizás no vas a hacer en el backoffice para administrar los cambios sobre esa tabla y prefieres dejar una configuración inicial de los estados que prevees vas a necesitar para que la aplicación funcione según los requisitos pedidos. Entonces usarás el seeder para generar los estados de los artículos iniciales que te han pedido.

## Crear un seeder

El proceso para crear un seeder comienza, como tantos otros, con el asistente artisan. Podemos pedirle que nos genere el esqueleto de un alimentador mediante el comando `make:seeder` indicando luego el nombre del seeder que queremos crear:

```
php artisan make:seeder ReviewsSeeder
```

Los seeder pueden tener cualquier nombre que necesites. La recomendación es indicar un nombre que te sirva para saber exactamente para qué se creó ese seeder. Por ejemplo podrás usar el nombre de la tabla que se va a alimentar y el sufijo "Seeder". Además como los seeders en código son clases, colocaremos la primera letra en mayúscula por convención. Algo como "BookSeeder" podría ser buen nombre o "BookTableSeeder".

```
php artisan make:seeder BookSeeder
```

Los seeders una vez creados se colocan en la carpeta "database/seeds". Ejecuta el comando anterior y allí encontrarás el seeder creado por artisan.

## Escribir el código de los seeder

Los seeder no son más que clases, de programación orientada a objetos, en las que tendrás el código de los datos que quieras insertar para alimentar tus tablas inicialmente. Estas clases tendrán un método llamado `run()` que contiene el código de los inserts que desees realizar dentro de tus tablas.

Existen varias maneras de "atacar" a la base de datos y producir los inserts que necesitas. Puedes usar sentencias construidas con "Query Builder" o directamente "Eloquent" a través de las operaciones disponibles en un modelo, eso es indiferente.

**Nota:** Esta parte la podemos describir de manera muy esquemática, porque todavía no hemos profundizado en el conocimiento que necesitamos para trabajar con la base de datos. Por ello nos vamos a limitar a insertar los datos valiéndonos de las operaciones del modelo. Para entender lo que viene a continuación tendrás que haber leído el capítulo de [introducción a los modelos en Laravel 5](#). Si hace tiempo que lo leíste y no has practicado lo suficiente te recomendamos hacer una segunda lectura antes de proseguir.

Utilizando las operaciones de un modelo, y el ORM Eloquent, podemos insertar datos a través del método `create()` del modelo. Por ejemplo, si tenemos el modelo "Review", que afecta a la tabla "reviews". Entonces podrás hacer `Review::create()` para acceder a la operación de inserción de datos.

Este método `create()` recibe un array asociativo, que contiene cada uno de los campos que se quieren aplicar con los valores a definir. Son pares claves / valor, la clave es el nombre del campo y el valor es el valor que pretendemos asignar.

```
Review::create([
    'campo'          => 'Valor campo',
    'campo_num'     => 0
]);
```

**Nota:** Recuerda que los modelos son clases, que se llaman con el mismo nombre que la tabla (la diferencia es que la primera letra de la tabla es en minúsculas y la primera letra de la clase de un modelo es en mayúsculas, así como la tabla está en plural y el modelo en singular). Esas clases en principio pueden estar vacías de código, puesto que la herencia de la clase principal Model ya nos ofrece la mayoría de las operaciones listas para usar. Para crear un modelo simplemente lanzas el correspondiente comando de artisan `make:model`. Será algo que necesitarás realizar antes de poder invocar el método `create()` del modelo. De nuevo, lee el artículo de los modelos si no te suena lo que estamos diciendo.

Además, en el caso que te apoyes en el modelo para crear los alimentadores, tendrás que asegurarte que haces el correspondiente "use" para tener disponible el espacio de nombres (namespace) de la clase del modelo que vas a usar.

Por ejemplo, este sería el código para nuestro seeder de Reviews. El tema del namespace, para conocer el modelo Review está en la línea "use App\Review;".

```
<?php
```

```
use Illuminate\Database\Seeder;
use App\Review;

class ReviewsSeeder extends Seeder
{
    public function run()
    {
        Review::create([
            'name'          => 'Vacas locas',
            'votes'         => 33,
            'fulldescription' => 'lalala'
        ]);
    }
}
```

## Ejecutar los seeders

En un proyecto podemos haber creado varios seeders, para alimentar de manera individual varias tablas. Estos alimentadores se pueden ejecutar de manera global (todos a la vez) configurando el código de una clase que está en la carpeta "database/seeds" llamada "DatabaseSeeder".

**Nota:** DatabaseSeeder la encontrarás de manera predeterminada en toda instalación de Laravel 5 y con un código básico para comenzar. Esta clase podríamos decir que es el seeder "global". Podríamos situar todo el código de las alimentaciones en este mismo archivo, en el método run(), pero no es lo habitual, puesto que resulta de mayor utilidad separar los seeders en diversos archivos como hemos aprendido.

Dentro de DatabaseSeeder, en el método run() podremos hacer tantas llamadas a seeders particulares a través del método call(), indicando qué seeders queremos ejecutar a través del nombre de la clase.

```
public function run()
{
    Model::unguard();

    $this->call(BookSeeder::class);
    $this->call(ReviewsSeeder::class);

    Model::reguard();
}
```

Las distintas clases seeder que hayas especificado se ejecutarán en el orden en el que estén escritas en este código.

**Nota:** Los métodos de Model unguard() y reguard() sirven para anular temporalmente y luego reactivar ciertas protecciones de seguridad en el modelo, que evitan que te inyecten datos indeseados en las tablas. Hablaremos más adelante de ellas.

Una vez configurado el DatabaseSeeder, podemos lanzar toda la secuencia de seeders a través del comando de artisan:

```
php artisan db:seed
```

**Nota:** Para realizar el "migrate" y a la vez procesar los seeders, en un solo paso, tienes como alternativa de lanzar el siguiente comando.

```
php artisan migrate:refresh --seed
```

Este comando lo podemos ejecutar un número indeterminado de veces. Si se repite su ejecución simplemente volverá a insertar los datos de nuevo en las tablas.

## Ejecutar seeders de manera individual

En ocasiones podemos necesitar realizar la ejecución de un seeder en concreto y no todos los generados para un proyecto. Para conseguir esto podemos usar el comando de artisan "db:seed" seguido del parámetro --class, al que le asignamos como valor el nombre del seeder a ejecutar de manera individual.

Quedaría algo como puedes ver en siguiente código:

```
php artisan db:seed --class=BookSeeder
```

Quizás nos ocurra entonces que recibamos un error:

```
[Illuminate\Database\Eloquent\MassAssignmentException]
```

Eso es por una protección que tiene Laravel para evitar que introduzcan datos que no deseamos en tablas. Lo veremos con detalle más adelante pero se soluciona colocando una pequeña declaración en la propiedad \$fillable del modelo. No te preocupes de momento por ello, nosotros vamos a saltarnos las protecciones para este paso de la siguiente manera. Vamos a avisar al modelo que vamos a realizar la alimentación saltándonos la protección con el método unguard() de la clase Model.

```
Model::unguard();
```

**Nota:** La llamada a unguard() es una operación que encontrarás en el código del "seeder global" DatabaseSeeder. Verás ahí el código para levantar las protecciones de nuevo.

Claro, si vas a usar la clase Model debes indicarlo con el correspondiente "use" para conocer su namespace.

```
use Illuminate\Database\Eloquent\Model;
```

Con estas dos líneas, el código de un seeder para que puedas ejecutarlo de manera individual te quedaría parecido a esto:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;
use App\Book;

class BookSeeder extends Seeder
{
    public function run()
    {
        Model::unguard();

        Book::create([
            'name'      =>  'Viaje al centro de la tierra',
            'author'    =>  'Julio Verne',
            'isbn'      =>  '14445884'
        ]);
    }
}
```

## Conclusión

De los seeders no hay mucho más que hablar. Pero lógicamente nuestro código se puede complicar mucho en función de diversas situaciones que queramos resolver, como cargar juegos de datos (varios reg en vez de uno), borrar datos que pudiera haber que no se desean en el estado inicial, etc.

Antes de terminar te damos una sugerencia de código rápido para poder alimentar una tabla con una gran cantidad de elementos, simplemente a través de un bucle for.

```
for ($i=0; $i<100; $i++){
    Book::create([
        'name'      =>  'Libro ' . $i,
        'author'    =>  'Autor ' . $i,
        'isbn'      =>  '14445884' . $i
    ]);
}
```

Para mejorar estos resultados podrías aprender a manejar Faker, una librería pensada para generar datos de prueba de modo que parezcan más reales, pero no es algo que de momento no vamos a tocar, pues es una librería externa a Laravel.

Además nos quedaría explorar toda la parte de acceso a datos con los métodos de los modelos, pero eso es algo que ya no depende del sistema de seeder y que veremos en los siguientes artículos.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 17/12/2015  
Disponible online en <http://desarrolloweb.com/articulos/seeders-laravel5.html>

## Práctica de acceso a base de datos en Laravel

**Realizaremos una práctica de acceso a base de datos en Laravel, con varias operaciones a partir de modelos de Eloquent en Laravel 5.**

Vamos a hacer una pausa para la práctica en la sección de bases de datos del [Manual de Laravel 5](#). Hemos visto de manera detallada la creación y modificación de tablas mediante las [migraciones](#) y además hemos conseguido insertar datos de prueba en las tablas gracias a los [seeders](#), así que nos resta jugar un poco con todo ello.

Nuestro ejercicio nos permitirá poner en práctica los conocimientos que ya poseemos hasta el momento y explorar algún área nueva, sobre todo en lo que respecta a los modelos. El objetivo es realizar un sencillo sistema para listar datos que tenemos en una tabla y para insertar datos nuevos en dicha tabla. Resultará bastante sencillo de entender si digo que nos quedaremos a la mitad del camino de lo que se conoce como CRUD. Osea, veremos solamente el "CR" Create y Read, dejando para más adelante el "UD" Update y Delete.

**Nota:** Todo lo que vamos a conocer de nuevo en este artículo son cosas que nos ofrece el ORM de Laravel, Eloquent. No es nuestra intención entrar en detalle con este asunto, porque es algo que veremos dentro de poco y para lo que necesitaremos varios artículos. La idea es comenzar a hacer algunos ejemplos en los que podamos ver la sencillez del acceso a datos de Laravel y podamos empezar a generar páginas que nos respondan entregando información que hemos guardado en nuestras tablas.



Aprovecharemos que en los artículos anteriores sobre los seeders poblamos la tabla de libros para usarla en esta práctica. Partimos pues sobre una tabla, ya rellena con información de prueba, llamada "books". Si no la tienes, por favor, sigue los pasos el [artículo sobre los Seeders](#).

### Rutas

Comenzaremos observando las rutas nuevas que hemos creado para este ejemplo.

```
route::get('libros', 'BookController@index');
route::get('libros/{id}', 'BookController@show')->where(['id' => '[0-9]+']);
route::get('libros/crear', 'BookController@create');
route::post('libros/crear', 'BookController@store');
```

Por orden de aparición, las rutas serán usadas para las siguientes páginas:

- libros: para ver un listado de todos los libros
- libros/{id}: para ver un detalle de un libro en particular
- libros/crear (GET): para mostrar el formulario con el que crearemos libros
- libros/crear (POST): para recibir el formulario con los datos de un nuevo libro y si valida correctamente, insertarlo

Con la lectura del código de las rutas debería quedar claro que el controlador encargado de ejecutar todas las solicitudes se llama BookController. Los métodos que tendremos que definir, con el código de cada operación, se llaman index(), show(), create(), store().

**Nota:** Recuerda que tienes más información sobre las rutas en capítulos anteriores. Puedes ver el artículo [Parámetros en las rutas](#), así como [Verbos en las rutas](#).

## Controlador BookController

Vamos a contar con un único controlador que realizará todas las acciones que serán necesarias para resolver nuestros objetivos.

El controlador, como sabes, se puede generar con un comando de artisan: “make:controller”. Es parte te la dejamos para ti, porque ya lo vimos en el capítulo de [introducción a los controladores](#), pero sin embargo quiero que prestes atención a los nombres de las acciones en el controlador por defecto que se ha creado. Son los mismos que hemos proyectado en las rutas anteriores, así que ya tenemos mucho del trabajo adelantado.

Nuestro código para gestionar esas acciones se puede ver a continuación:

**El listado de todos los libros:** La primera acción debe mostrar todos los libros de la tabla books.

```
public function index()
{
    $libros = Book::all();
    return view('libro.todos', ['libros' => $libros->toArray()]);
}
```

Hacemos una búsqueda de los libros a través del modelo Book. Luego llamamos a una vista que se

encargará de mostrar el listado de todos los libros.

**Nota:** El modelo Book lo debes haber creado al poner en práctica artículos anteriores, puesto que ya lo usamos para crear el seeder de la tabla de libros (books). De todos modos, de momento es un modelo vacío de contenido, por lo que si no lo tienes lo generas directamente con el comando “make:model” del asistente artisan.

Cuando solicitamos información a un modelo como en este caso, `Book::all()`, lo que nos devuelve el método `all()` es un objeto colección “collection”. No hemos llegado a explicar qué hay de útil detrás de las colecciones y por ahora vamos a dejarlo aparte, pero comenzaremos a usarlas. De momento lo único que hacemos con la colección es invocar el método `toArray()`, que nos devuelve los datos que contiene, pero en formato de array.

Ese array generado a partir de la colección se lo pasamos a la vista para que sea capaz de pintar los libros que se han encontrado en la consulta a la tabla.

Por cierto, en el controlador no se te debe olvidar hacer el “use” de la clase donde está el modelo Book.

```
use App\Book;
```

**El detalle de un libro:** La segunda acción que vamos a observar es la que se invoca cuando se solicita el detalle de un libro, método `show()`.

```
public function show($id)
{
    $libro = Book::find($id);
    if (!is_null($libro))
        return view('libro.mostrar', ['libro' => $libro->toArray()]);
    else
        return response('no encontrado', 404);
}
```

Este método recibe como parámetro el id del libro que se desea visualizar. Ese id se lo pasamos a la ruta y gracias al `where()` del código de la ruta ya está comprobado que sea un número.

**Nota:** Recuerda el código de la ruta:

```
route::get('libros/{id}', 'BookController@show')->where(['id' => '[0-9]+']);
```

Tal como está construida, solo se activará cuando el parámetro id sea un número.

En esta acción del controlador solicitamos información al modelo de los libros (Book) mediante el método `find()`, que sirve para buscar un elemento a partir de su clave primaria. Si encontró alguna cosa me devolverá

la colección con aquello que se haya encontrado. Si no encuentra nada me devolverá null. En función del valor de retorno, se realizan cosas distintas.

- Devuelvo una vista para mostrar el libro, en caso que no sea nulo, enviándole a la vista los datos del libro en un array
- O bien, si no se encontró nada, devuelvo un objeto response con un error 404 de página no encontrada y un mensaje

**Formulario de creación de un libro:** Ahora veamos la acción que comienza el proceso de creación de un libro, que se encarga de mostrar el formulario para introducir los datos de un libro.

```
public function create()
{
    return view('libro.formLibro');
}
```

Este es el más sencillo de todos los métodos del controller, porque solo tiene la llamada a una vista, que tendrá el HTML del formulario de alta de un libro.

**Creación del libro, a partir de los datos del formulario:** Para acabar el controlador nos queda ver el método store, que es el que realmente se encarga de crear el nuevo libro.

```
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|min:5',
        'author' => 'required|min:8',
        'isbn' => 'required'
    ]);

    Book::create($request->all());
    return redirect('/libros');
}
```

El método store() recibe la Request, dependencia que se inyecta en la llamada al método de la acción, necesario porque vamos a tener que recuperar los datos de la solicitud POST (los datos enviados en el formulario).

Dentro del método realizamos una validación de los datos recibidos por formulario y si todo ha ido bien, es cuando creamos el recurso, gracias al método create() del modelo Book, enviándole los datos de la solicitud, todos convertidos a array mediante el método all().

Una vez creado el libro en la tabla, se redirige a la página “home” de libros, que se encarga de mostrar el listado de los libros que hay en el sistema.

Esta es la parte más compleja de la práctica, porque entran en juego las validaciones, sin embargo esperamos que puedas recordar los mecanismos que ya fueron relatados en el [artículo de validaciones en Laravel](#).

## Modificación en el modelo Book

Tal como está nuestro código y el uso que hacemos del modelo con el método `Book::create()`, vamos avisando que Laravel nos arrojará una excepción cuando se intente crear el nuevo libro.

```
MassAssignmentException in Model.php line 424:
```

No es la primera vez que nos aparece un error similar, así que vamos a intentar concretar ya de una manera sencilla qué es lo que está pasando.

El método `Book::create()` nos permite crear un elemento e insertarlo en la base de datos. Para ello recibe un array y usa todos los datos que le estamos pasando. Esos datos en nuestro ejemplo vienen directamente de la solicitud y los volcamos tal cual para poder crear ese nuevo libro:

```
Book::create($request->all());
```

Imagina que viene a través de la solicitud un dato que realmente no estábamos esperando, por ejemplo un dato de control que usas internamente para saber el estado de un libro y no deseas que sea parte de la entrada del usuario. ¿Cómo podría ocurrir que te manden otros datos, además de los que deseas? pues simplemente un usuario con conocimientos mínimos podría crear otros campos en el formulario, editando la página con las herramientas de desarrolladores y así se enviarían también en la solicitud por el método POST al enviar ese formulario normalmente.

Así que Laravel, para evitar estas cosas, al usar el método `create()` que es potencialmente peligroso por una posible inyección de datos no deseados, obliga a que tengas declarados en el modelo aquellos campos que se van a poder insertar al hacer un `create()`.

**Nota:** Hay otros métodos de Eloquent para crear elementos en las tablas, que no tendrían este problema porque de manera explícita se indican campo a campo las columnas a insertar con los valores.

Esta configuración se realiza a través de una propiedad que tendremos que crear dentro del modelo, llamada `$fillable`. El código te quedará así:

```
class Book extends Model
{
    protected $fillable = ['name', 'author', 'isbn'];
}
```

## Vistas de la aplicación

Ya solo nos queda ver las vistas que hemos creado para esta práctica. Todas las hemos situado en la carpeta `resources/views/libro`.

**Nota:** Todavía nos quedan cosas por ver de las vistas, sobre todo del motor de plantillas Blade, pero recuerda que ya pudimos encontrar una [introducción a las vistas](#).

**Vista libro.todos: (archivo todos.php)** Esta vista hace el recorrido a array de libros que se le ha pasado y los va mostrando uno a uno.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Todos los libros</title>
</head>
<body>
  <?php foreach ($libros as $libro): ?>
    <p>
      <?=$libro['name']?>, por <?=$libro['author']?>
      <br>
      ISBN: <?=$libro['isbn']?>
    </p>
  <?php endforeach ?>
</body>
</html>
```

En este caso usamos un “foreach” clásico de PHP (con el código sugerido como alternativa a las vistas), aunque más tarde al aprender Blade veremos una forma más clara de escribir este mismo código.

**Vista libro.mostrar: (archivo mostrar.php)** Esta segunda vista muestra el detalle de un libro.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Mostrar un libro</title>
</head>
<body>
  <h1>
    <?=$libro['name']?>
  </h1>
  <p>
    Por <?=$libro['author']?>
    <br>
    ISBN: <?=$libro['isbn']?>
  </p>
</body>
</html>
```

**Nota:** Es cierto que había otros campos en la tabla de libro, pero no los hemos querido usar por ahora, para no incrementar la dificultad de esta práctica.

vista libro.formlibro: archivo formlibro.blade.php Esta tercera vista, y última, es la que usaremos para mostrar el formulario de un libro, que de momento estamos usando para la creación de libros. Hemos elegido como extensión .blade.php porque vamos a usar un pedazo de código con sintaxis blade para mostrar los posibles errores de validación de los datos del formulario.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Crear un libro</title>
</head>
<body>
  <h1>Crear un libro</h1>
  @if(count($errors) > 0)
    <div class="errors">
      <ul>
        @foreach($errors->all() as $error)
          <li>{{ $error }}</li>
        @endforeach
      </ul>
    </div>
  @endif
  <form action="/libros/crear" method="post">
    Nombre: <input type="text" name="name" value="{{old('name')}}">
    <br>
    Autor: <input type="text" name="author" value="{{old('author')}}">
    <br>
    ISBN: <input type="text" name="isbn" value="{{old('isbn')}}">
    <br>
    <input type="submit" value="Crear">
  </form>
</body>
</html>
```

Esta vista tiene varios detalles importantes, primero el recorrido a los errores de validación y luego la “memoria” en los campos de formulario para que sean capaces de recordar su estado cuando fue enviado el formulario la última vez. Todo esto no lo vamos a explicar porque se vio con suficiente detalle en el [capítulo de validaciones en Laravel](#).

Con esto termina esta práctica, ya solo faltaría poner en marcha el ejercicio accediendo a las rutas de la aplicación definidas.

Este artículo es obra de *Miguel Angel Alvarez*

Fue publicado por primera vez en 29/12/2015

Disponible online en <http://desarrolloweb.com/articulos/practica-acceso-base-datos-laravel.html>

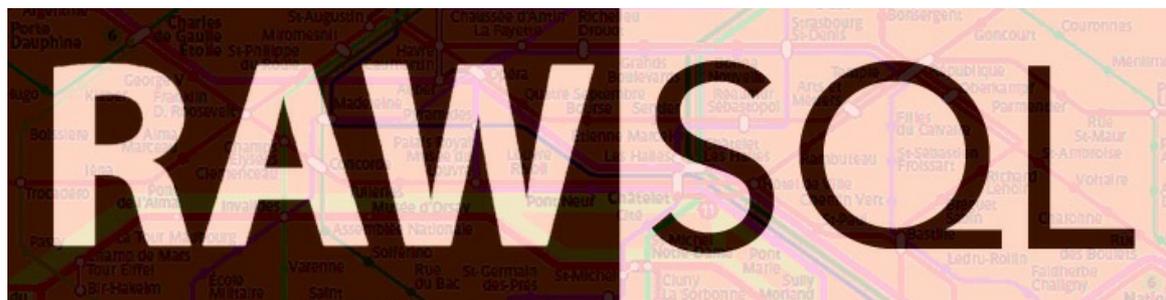
## Raw SQL en Laravel 5.1

Aprendemos a escribir consultas con SQL crudo, Raw SQL, en Laravel 5.1, el método de más bajo nivel para acceder a los datos mediante este framework PHP.

De entre todos los mecanismos que nos propone Laravel para acceso a datos, el de más bajo nivel es el que vamos a tratar en este artículo: "Raw SQL", que sería como escribir las consultas SQL "en crudo". En este tipo de acceso realizamos directamente el SQL de las consultas, que ejecutamos vía la clase BD que es una fachadas que nos ofrece Laravel.

Las consultas que escribes tendrán SQL puro, por lo que usarás el SQL propio del sistema gestor que estás utilizando en este momento. Otros mecanismos que veremos más adelante como Query Builder o el propio ORM Eloquent son independientes de la base de datos que se use.

Este no es el mecanismo más versátil, ni tampoco el más productivo. De hecho, en los anteriores artículos del [Manual de Laravel 5.1](#), pudimos comprobar que, casi sin código en los modelos éramos capaces de realizar muchas de las operaciones típicas con los datos de las tablas. Por ese motivo, en la mayoría de los casos resultará más interesante usar Eloquent que escribir nuestras propias sentencias SQL.



El motivo de la existencia de Raw SQL es que hay ocasiones en las que las operaciones con las bases de datos son complejas y se requiere la escritura de consultas muy apuradas, que hagan cosas que no puedas a través de otros mecanismos, o bien porque deseas que se ejecuten más rápido. En ese caso podemos bajar hasta el nivel de la base de datos y ejecutar las consultas directamente contra el sistema gestor.

**Nota:** Realmente con PHP podríamos acceder a los datos a niveles más cercanos a la base de datos, usando extensiones "nativas" para la base de datos en concreto que se desee, como por ejemplo las de MySQL: MySQLi. Sin embargo, llegar a ese nivel es innecesario, porque con las Raw SQL tenemos prácticamente el mismo nivel de control. Sería también contraproducente, porque no podríamos usar nada del código que ya incorpora Laravel para el acceso a datos, pero si aún así lo deseamos, lo podríamos hacer ya que Laravel es PHP al final de cuentas.

## Facade DB

La fachada DB nos sirve para ejecutar las consultas crudas. Tiene diversos métodos que debemos invocar

en función de la operación dentro de la base de datos: `select()`, `insert()`, `update()`, `delete()`. Además hay un método `statement()` para operaciones con el sistema gestor que no devuelven datos, como un "drop table".

El motivo de la existencia de estos métodos es que Laravel te ayuda un poco más al devolverte diversas cosas como resultado al ejecutar las consultas. Dicho en otras palabras, va un poco más allá que ejecutar una sentencia SQL, tal cual, con harías con `mysqli_query()` por ejemplo.

- `select()` siempre devuelve un array de resultados
- `insert()` devuelve un booleano indicando si se insertó o no
- `update()` devuelve el número de elementos actualizados
- `delete()` devuelve el número de elementos borrados

Ahora veamos unos simples pedazos de código que nos servirán como ejemplo.

**Selección (select):** Comenzaremos por este, en el que recibimos todos los libros de la tabla "books".

```
$libros = DB::select('select * from books where 1');
```

Después de la ejecución de esta línea, en `$libros` encontraremos un array con todos los libros que haya en la tabla.

```
array:121 [▼
  0 => {#231 ▼
    +"id": 1
    +"name": "Viaje al centro de la tierra"
    +"author": "Julio Verne"
    +"description": ""
    +"date_published": "0000-00-00"
    +"isbn": "14445884"
    +"created_at": "2015-09-09 20:16:47"
    +"updated_at": "2015-09-09 20:16:47"
  }
  1 => {#232 ▶}
  2 => {#233 ▶}
  3 => {#234 ▶}
  4 => {#235 ▶}
  5 => {#236 ▶}
  6 => {#237 ▶}
```

Una interesante alternativa para selección de elementos nos la ofrece el método `selectOne()`. La única diferencia es que solo nos selecciona un elemento y por tanto nos evita usar un array. Lo que nos devuelve en su lugar es el objeto del único elemento encontrado (o bien null si no ha encontrado nada).

```
$libroUnico = DB::selectOne('select * from books where id=1');
```

```
{#233 ▾
  +"id": 1
  +"name": "Viaje al centro de la tierra"
  +"author": "Julio Verne"
  +"description": ""
  +"date_published": "0000-00-00"
  +"isbn": "14445884"
  +"created_at": "2015-09-09 20:16:47"
  +"updated_at": "2015-09-09 20:16:47"
}
```

**Inserción (insert):** En el siguiente código insertamos un libro. Además hemos agregado una comprobación para saber si se llegó a insertar y en ese caso devolver un mensaje de respuesta.

```
$insertado = DB::insert('insert into books (name) values ("La Colmena)');
if($insertado){
    return 'insertado';
}
```

**Actualización (update):** En este ejemplo encontrarás una actualización de unos libros y un mensaje indicando cuántos se actualizaron al ejecutarse la consulta.

```
$num = DB::update('update books set name="Encantos de mujer" where id=55 or id=38');
return "Se han actualizado $num elementos";
```

**Borrado (delete):** Y por último una sentencia de borrado, con una línea de código adicional para informar cuántos se han borrado.

```
$num = DB::update('delete from books where id=65');
return "Se han borrado $num elementos";
```

## "Bindeo" de parametros para filtrado

Una de las preocupaciones más importantes de cualquier desarrollador debería ser la seguridad y en el trabajo con bases de datos lo más crucial es **evitar los ataques por inyección de SQL**. Muchos de los mecanismos de Laravel para acceso a datos ya tienen implícitos sistemas de filtrado de la información, de modo que no tengas que preocuparte por filtrar tú mismo las variables antes de usarlas en consultas SQL. En el caso de las consultas con Raw SQL, la protección pasa por usar el "bindeo" de parametros en las consultas.

Si no bindeas los parámetros tu código puede ser vulnerable y es un error en el que resutaría fácil de caer cuando se realizan consultas con Raw SQL. Para entender cómo podemos poner en riesgo una aplicación web mediante inyección SQL podemos ver dos códigos que funcionarían aparentemente igual.

### 1) Código vulnerable:

```
route::get('rawsql/libros/buscarid', function(){
    $id = \Request::input('id');
    $libros = DB::select('select * from books where id=' . $id);
    dd($libros);
});
```

El código anterior es vulnerable, porque estamos concatenando directamente en el SQL un valor, sin hacer el bindeo del parámetro. Especialmente, si el valor viene de una entrada de datos por parte del usuario, como es el caso, podría contener caracteres que produzcan SQLs diferentes de las que en principio estamos suponiendo que vamos a enviar.

**Nota:** No es el objetivo explicar cómo burlar la seguridad de ese código con una inyección de SQL, pero si te interesa puedes buscar documentación en Internet, que hay bastante.

Este segundo código hace exactamente lo mismo. Pero en este caso sí se realiza el binding del parámetro usando el procedimiento que te asegura la protección de tu sentencia.

## 2) Código seguro frente a SQL injection:

```
route::get('rawsql/libros/buscarid', function(){
    $id = \Request::input('id');
    $libros = DB::select('select * from books where id= ?', [$id]);
    dd($libros);
});
```

**Nota:** El bindeo de los parámetros no es algo específico de Laravel. Si usas MySQLi por ejemplo también puedes hacer bindeo de parámetros preparando las consultas. O bien con PDO. Quiere decir que las protecciones para el paso de parámetros como datos en las consultas son propias de PHP nativo y que las debes usar aunque no trabajes con ningún framework. Por ese motivo imaginamos que conoces un poco este asunto y no necesitamos explicar mucho más.

Puedes bindear varios parámetros en una consulta, simplemente agregando más variables al array, que harán corresponderse con los caracteres "?" por orden de aparición.

```
$titulo = "El Quijote";
$autor = "Cervantes";
$insertado = DB::insert('insert into books (name, author) values (?, ?)', [$titulo, $autor]);
```

Así mismo, puedes realizar el bindeo de datos a la consulta usando parámetros con nombres, en cuyo caso usarás un array asociativo para indicar los valores.

```
$libros = DB::select('select * from books where name like :busqueda limit :cuantos', array(
'busqueda' => "%$cadena%",
'cuantos' => $num
));
```

**Nota:** Un error común en el bindeo de parametros es colocar el binding entre comillas. Por ejemplo "select from books where name = '?' limit 1". Esto está mal escrito, aunque name tenga valores que son alfanuméricos, puesto que el sistema de incorporación de los parámetros debe ser el encargado de saber si algo va entre comillas o no, y componer la sentencia correctamente. Nunca entrecorramos los bindeos. Tampoco estaría bien "select from books where name like '%:busqueda%' limit 3", por el mismo motivo.

## Conclusión

Hemos hecho un rápido repaso a el acceso a base de datos a través de la escritura de nuestras propias sentencias SQL. Laravel es capaz de ayudarnos bastante para recuperar la información de las tablas, actualizar, insertar o borrar, entre otras acciones, pero el grado de automatización del acceso a datos no llega a compararse al que tienes mediante otros procedimientos.

Así mismo, el procedimiento que tendrás que realizar en algunos casos puede ser distinto dependiendo del sistema gestor, ya que muchas veces se encuentran cambios en las SQL de uno y otro motor de bases de datos. Sin embargo, si usamos mecanismos de más alto nivel para acceder a BBDD (como Query Builder o el ORM Eloquent), nos aseguramos que el código sea el mismo, independientemente del gestor de bases de datos que estemos usando. Esto creo que sería interesante verlo con un ejemplo para que nos demos cuenta.

Por ejemplo vamos a **recuperar el id de la última inserción**, con Raw SQL para el motor de bases de datos MySQL.

```
$titulo = "El Padrino";
$autor = "Mario Puzo";
$insertado = DB::insert('insert into books (name, author) values (?, ?)', [$titulo, $autor]);
if($insertado){
    $id = DB::selectOne('SELECT LAST_INSERT_ID() as "id"');
    return 'Insertado correctamente con id ' . $id->id;
}
```

No es lo más fácil, ni tampoco lo más legible, pero el problema principal de este código para obtener el identificador del insert realizado es que sólo funcionará para MySQL o MaríaDB. Si queremos cambiar el sistema gestor deberíamos revisar esas líneas de código, lo que es poco atractivo.

No obstante, queremos remarcar que este tampoco es un problema serio en Laravel, puesto como decimos existen otros mecanismos de acceso a las bases de datos que sí permiten crear un código independiente del sistema de base de datos que estemos usando en un momento dado. Continuaremos viendo Query Builder para ir avanzando en nuestro conocimiento y aprender a manejar otros sistemas de Laravel más versátiles que las Raw SQL.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 14/01/2016  
Disponible online en <http://desarrolloweb.com/articulos/raw-sql-laravel.html>

## Query Builder Laravel 5

### Qué es Query Builder, junto con una guía de uso en el framework PHP Laravel 5.1.

Como ya señalamos en la [introducción a las bases de datos en Laravel](#), existen diversas vías de acceso a los datos, con distintos niveles de abstracción. Query Builder es uno de ellos. Su nivel no es tan bajo como [escribir las consultas en crudo \(Raw SQL\)](#), pero sí más cercano al sistema gestor de base de datos de lo que sería el ORM.

Query Builder contiene una serie de funciones listas para realizar las operaciones más comunes con una base de datos, pero sin usar el lenguaje SQL directamente, sino el API de Laravel. En este artículo vamos a ver una completa introducción a Query Builder de Laravel 5.1, que nos sirva para conocer la operativa de trabajo con el sistema. Pero antes de comenzar conviene señalar un par de características importantes.



1. Query Builder trabaja con todos los tipos de bases de datos soportadas por Laravel. Por tanto, el código que nosotros usaremos se podrá ejecutar para cualquier gestor compatible, obteniendo los mismos resultados. Por tanto, este sistema permite abstraerse del motor de base de datos que estemos usando por abajo.
2. Query Builder usa internamente los mecanismos de PDO, incluido el bindeo de parámetros en las consultas. Por tanto, no es necesario filtrar los datos que vamos a usar en las sentencias, ya que éstos serán filtrados automáticamente para protegernos de ataques por inyección SQL.

Como observarás, el uso del ORM Eloquent simplifica todavía más el trabajo con las bases de datos, sin embargo Query Builder consigue completar las funciones en una menor cantidad de tiempo. Para un juego de datos o número de operaciones pequeñas ese tiempo será imperceptible, pero a medida que sube la carga se podrá ver que Query Builder es un poco más rápido.

### Objetos "fluent" Query Builder

Para trabajar con Query Builder seguimos requiriendo el uso de la "DB Facade", pero en este momento, en vez de usar los métodos para [ejecutar consultas crudas \("Raw SQL"\)](#) que vimos en el anterior artículo, usaremos el método `table()`.

Al método `table()` le pasamos por parámetro el nombre de la tabla con la que pretendemos operar y nos devuelve un objeto de clase `Illuminate\Database\Query\Builder`, al que se conoce habitualmente con el nombre de "Fluent Query Builder" o simplemente "Fluent".

```
$fluent = DB::table('books');
```

En `$fluent` tenemos ahora una referencia a un objeto "Fluent Query Builder" que ha sido enlazado con la tabla "books". Ese objeto ya estará enlazado para trabajar sobre la tabla indicada y le podremos pasar mensajes (invocar sus métodos) para hacer las operaciones precisas que necesitemos en cada momento sobre esa tabla.

**Nota:** no te olvides de hacer el correspondiente "Use DB;" en caso que estés trabajando con esta fachada desde un namespace en el que no se conozca esa clase, por ejemplo desde un controlador.

Ahora podemos invocar otros métodos para poder realizar distintas operaciones. Por ejemplo, `get()` para recibir los datos de una tabla.

```
$libros = $fluent->get();
```

Eso nos devolverá un array de elementos, donde en cada casilla tendremos referencias a objetos, de clase `stdClass`, con los datos de un libro.

**Nota:** `stdClass` es una clase incorporada en PHP nativo que se usa cuando se convierte una variable (que no es tipo objeto) en un objeto, por ejemplo, haciendo un casting de un array asociativo a un objeto. Osea, es un objeto "genérico" por decirlo de alguna manera, que no hace ninguna cosa en concreto.

Estas dos operaciones se suelen encadenar en una única línea de código:

```
$libros = DB::table('books')->get();
```

Aparte de `get()`, existen otros métodos, para convertir un objeto Fluent Query Builder en una estructura de datos, por ejemplo `first()` te devolverá el primero de los elementos de la colección que contenga un objeto fluent.

```
$libro = DB::table('books')->first();
```

En este caso no recibes como retorno un array de libros, sino que recibes un único libro en un objeto de clase `stdClass`.

La diferencia entre `get()` y `first()` es que en el primero obtienes un array con un número indeterminado de elementos y en la segunda obtienes un objeto con un único elemento. Para comprobarlo, te damos dos pedazos de código realizarán la misma tarea. Solo difiere la manera de acceder al primer elemento para recuperar su nombre.

```
// Si uso first() accedo a las columnas del registro mediante un objeto
$libro = DB::table('books')->first();
echo $libro->name;
```

```
// si uso get() recibo un array, cuyo primer registro se indexa con [0]
$libros = DB::table('books')->get();
echo $libros[0]->name;
```

Esto es una pequeñísima parte de lo que se puede hacer con Query Builder, pero es el inicio de todo tipo de operaciones más complejas que vamos a ver. Sacarle todo su provecho se basa en encadenar métodos que nos sirven para hacer las cosas que necesitemos. Es esencial por tanto contar con la [documentación de Query Builder en Laravel 5.1](#), que es la versión que manejamos en este manual.

## Operaciones de selección con Query Builder

Al hacer operaciones de selección tienes la opción de especificar los campos que quieres recuperar con el método `select`:

```
$libros = DB::table('books')->select('name')->get();
```

Si hay más de un campo que quieras seleccionar, puedes enviar un array con los campos que te interesa recuperar.

```
$libros = DB::table('books')->select(['name', 'author'])->get();
```

Veamos ahora cómo seleccionar un grupo de usuarios concreto, con el método `where()`, que encadenamos a la creación del objeto fluent.

```
$libros_mario_puzo = DB::table('books')->where('author', '=', 'Mario Puzo')->get();
```

En este código tenemos como resultado un objeto fluent query builder con un grupo restringido de elementos de la tabla "books". Obtenemos la tabla entera y luego le encadenamos el método `where()` indicando que necesitamos aquellos libros con "author" igual a "Mario Puzo".

El método `where` puede recibir cualquier tipo de operador, incluso el "like".

```
$libros = DB::table('books')->where('author', 'like', '%julio%')->get();
```

Si queremos hacer dos condiciones, por ejemplo que el autor sea tal cosa y que el título sea tal otra, podemos encadenar dos métodos `where()`.

```
$libros = DB::table('books')->where('author', '=', 'Mario Puzo')->where('name', 'like', '%padrino%')->get();
```

**Nota:** Si quieres aumentar la legibilidad de tanto encadenamiento de mensajes, puedes escribir esto mismo en varias líneas.

```
$libros = DB::table('books')
->where('author', '=', 'Mario Puzo')
->where('name', 'like', '%padrino%')
->get();
```

Si queremos combinar varios `where()` con la función lógica "or", por ejemplo autores que se llamen "Mario Puzo" o "Cervantes", puedes hacerlo con el método `orWhere()`.

```
$libros = DB::table('books')
->where('author', '=', 'Mario Puzo')
->orWhere('author', '=', 'Cervantes')
->get();
```

Tienes muchas otras opciones como `whereNull()`, `whereNotNull()`, `whereIn()`, `whereNotin()`, `whereNotBetween()`.

Por ejemplo, el código anterior (autores que sean una cosa o la otra) podría haberse escrito:

```
$libros = DB::table('books')
->whereIn('author', ['Mario Puzo', 'Cervantes'])
->get();
```

## Joins con Query Builder

Si queremos juntar dos o más tablas con Query Builder también tenemos la posibilidad de usar las operaciones de "join", con el método `join()` invocado sobre el objeto Fluent Query Builder.

Este método recibe varios parámetros:

- La tabla a unir
- Campo de la tabla 1 por el que se relacionan
- Operador de la relación, usualmente "="
- Campo de la tabla 2 por el que se relacionan
- Tipo de join, por defecto "inner"
- Boolean, positivo para hacer un "join where", por defecto es false

Los parámetros clave son los 4 primeros, que veremos en uso en el siguiente ejemplo. Dada una tabla de facturas y una de clientes, donde las facturas tienen un `id_cliente`.

```
$facturasCliente = DB::table('clientes')
->join('facturas', 'facturas.id_cliente', '=', 'clientes.id', 'inner', true)
->select('clientes.*', 'facturas.id as id_factura', 'facturas.fecha')
->where('clientes.email', '=', 'miguel@desarrolloweb.com')
->get();
```

Se pueden unir más de dos tablas, especificando los distintos Joins. Al esquema anterior vamos a unirle una tabla de "item facturable", donde cada factura tiene un número indeterminado de items facturables, que serían los conceptos de la factura.

```
$facturasCliente = DB::table('clientes')
->join('facturas', 'facturas.id_cliente', '=', 'clientes.id')
->join('item_facturables', 'facturas.id', '=', 'item_facturables.id_factura')
->select('clientes.*', 'facturas.id as id_factura', 'facturas.fecha', 'concepto')
->where('clientes.email', '=', 'miguel@desarrolloweb.com')
->get();
```

Existen varios tipos de join y también varios métodos para hacer joins como `leftJoin()`, `rightJoin()`, `joinWhere()`, etc. También existe una variante del método `join` que permite configurar la relación por medio de una función anónima. Para todos estos usos, consultar la documentación.

## Inserts con Query Builder

Para insertar uno o varios campos en registros de una tabla podemos lanzar el mensaje `insert()` a un objeto Fluent Query Builder. Obtenemos el objeto como hasta ahora, con el método `table()` de la facade DB, indicando el nombre de la tabla: `DB::table('books')`. Entonces encadenamos la llamada al método `insert()`, de la siguiente manera:

```
$inserted = DB::table('books')
->insert([
    'name' => 'La Ciudad de los Prodigios',
    'author' => 'Eduardo Mendoza'
]);
```

La llamada al método `insert()` devuelve un booleano indicando si pudo completar la inserción con éxito.

Ahora podemos ver otra alternativa de llamada a `insert()`, enviando como parámetro un array de arrays para insertar varios elementos a la vez.

```
$inserted = DB::table('books')
->insert(
[
[
```

```
'name' => 'Cien años de soledad',
'author' => 'García Márquez'
],
[
    'name' => 'El Alquimista',
    'author' => 'Paulo Coelho'
]
]
);
```

Si queremos saber el Id del registro insertado tenemos la alternativa de usar el método `insertGetId()`, como se puede ver en el código siguiente:

```
$id = DB::table('books')
->insertGetId([
    'name' => 'La danza de la realidad',
    'author' => 'Alejandro Jodorowsky'
]);
```

## Updates con Query Builder

Hacer un update es bien sencillo y similar a lo que hemos realizado ya. Invocamos el método `update()` sobre el Query Builder, al que pasamos un array asociativo con los datos a actualizar. Este comando se suele combinar con el método `Where`, para restringir el registro o grupo de registros que quieres actualizar con estos datos.

```
$updates = DB::table('books')
->where('id', '=', '74')
->update([
    'name' => 'Sin noticias de Gurb',
    'author' => 'Eduardo Mendoza'
]);
```

En este caso, como valor de retorno del `update` obtendremos el número de registros que se actualizaron al ejecutarse la consulta.

Como utilidad adicional, existen unos shortcuts para hacer una operación de incremento o decremento, aplicable en campos numéricos. En lugar de escribir a mano el código para hacer el update se realiza la llamada al método `increment()` o `decrement()`. El campo que se quiere incrementar y decrementar se envía como primer parámetro y de manera opcional se puede indicar un segundo parámetro con las unidades de incremento o decremento.

```
$num_updates = DB::table('books')
->where('id', '=', '97')
->increment('lecturas');
```

## Deletes

Los deletes son exactamente iguales a los updates, solo que se tiene que invocar el método `delete()` para borrado.

```
$delete = DB::table('facturas')->delete();
```

No te olvides colocar el método `where()` para restringir los registros que serán borrados! Si lo que quieres es borrar todos los elementos de una tabla, y restaurar su estado inicial sin datos, también tienes el método `truncate()`.

## Debug simple con `toSql()` en las sentencias Query Builder

Las sentencias de selección se puede debugear fácilmente con el método `toSql()`. Simplemente lo usamos en vez de generar el conjunto de registros en un array con `get()` o `first()`.

`toSql()` nos devuelve la cadena de la sentencia en crudo (Raw SQL) que se ejecutará en el servidor, que luego podemos mostrar con una función como `var_dump()` o la incorporada `dd()` en Laravel.

```
$rawSql = DB::table('books')
    ->select(['name', 'author'])
    ->where('author', '=', 'Mario Puzo')
    ->orWhere('author', '=', 'Cervantes')
    ->toSql();
dd($rawSql);
```

Esto nos mostraría la cadena de la sentencia a ejecutar, pero antes de realizar el bindeo de los parámetros.

```
select `name`, `author` from `books` where `author` = ? or `author` = ?
```

En muchos de los casos será suficiente para obtener la información que necesitamos con el objetivo de saber si la consulta es la que se esperaba. No obstante, existen otros métodos más avanzados de debuggear la consulta que veremos más adelante cuando hablemos de escuchar eventos de consultas o debug en Laravel.

## Conclusión

Hemos aprendido a manejar Query Builder en Laravel 5.1, creando objetos Fluent y encadenando todo tipo de mensajes para hacer operaciones de selección, selección con relaciones entre tablas, inserciones, actualizaciones y borrados. Gracias a las características de Query Builder, todo el código que realicemos será compatible con cualquier sistema gestor de bases de datos que soporte Laravel.

La construcción de consultas SQL con Fluent Query Builder tiene, no obstante, otras muchas cosas interesantes que no hemos llegado a ver todavía. En próximos artículos y en ejemplos prácticos seguiremos trabajando con esta alternativa al acceso a bases de datos, aunque en adelante vamos intentar centrarnos

más en el ORM, Eloquent, ya que nos ofrece mayores facilidades. No obstante, tampoco debes perder de vista lo aprendido sobre Query Builder, porque la mayoría de métodos que has conocido, por no decir todos, te van a servir también cuando trabajas con Eloquent.

Recuerda que la documentación oficial también te puede aclarar puntos a los que no hemos llegado en el presente artículo.

Este artículo es obra de *Miguel Angel Alvarez*.  
Fue publicado por primera vez en *17/02/2016*  
Disponible online en <http://desarrolloweb.com/articulos/query-builder-laravel5.html>

# Eloquent: el ORM de Laravel

Comenzamos a explorar el ORM que incorpora Laravel para el trabajo con datos que vienen de tablas de la base de datos. De entre todos los disponibles en el framework, Eloquent es el mecanismo de acceso a bases de datos de más alto nivel. Resulta sencillo de usar y ahorra mucho código de acceso a la información de las tablas, permitiendo consultar los datos como si fueran objetos, implementando relaciones entre las tablas como propiedades de los mismos objetos.

## Laravel Eloquent

**Introducción a Eloquent, el ORM de Laravel que implementa el patrón Active Record para el trabajo con datos que llegan de bases de datos relacionales.**

En los capítulos anteriores del [Manual de Laravel](#) abordamos el [trabajo con bases de datos](#) a distintos niveles. Desde la creación de [consultas en SQL crudo](#), para el sistema gestor con el que estamos trabajando, hasta la creación de consultas por medio de programación orientada a objetos con [QueryBuilder](#). Vimos que estos distintos niveles de trabajo conllevan diferentes prestaciones, siendo lo más importante de QueryBuilder la posibilidad de trabajar con abstracción de la base de datos que estamos usando.

En este artículo y los siguientes vamos a abordar el nivel de acceso a bases de datos de más alto nivel, que sería Eloquent. Eloquent es lo que se conoce como "ORM" (Object Relational Mapping o Mapeo de Objeto Relacional). Básicamente es un sistema que nos permite llevar la capa de persistencia en bases de datos por medio de objetos y que nos ahorra el trabajo de comunicar directamente con la base de datos.



## Qué es un ORM, qué es Active Record y cómo es Eloquent en Laravel

Con Eloquent, y en general con cualquier ORM tenemos la posibilidad de trabajar con los datos que hay en las bases de datos por medio de objetos. Los datos de las tablas se mapean a objetos, evitando todo el trabajo de escribir las consultas para el acceso a la información. El acceso a sus relaciones también se realiza por medio de propiedades de objetos, lo que facilita mucho las dinámicas de acceso a la información, siempre que haya una relación de dependencia declarada entre las distintas entidades de nuestro modelo de datos.

La forma de trabajo de Eloquent implementa el patrón "Active Record", un patrón de arquitectura de software que permite almacenar en bases de datos relacionales el contenido de los objetos que se tiene en memoria. Esto se hace por medio de métodos como `save()`, `update()` o `delete()`, provocando internamente la escritura en la base de datos, pero sin que nosotros tengamos que componer las propias sentencias.

En Active Record una tabla está directamente relacionada con una clase. En ese caso se dice que la clase es una envoltura de la tabla. La clase en si es lo que conocemos en el framework como "modelo". Cuando nosotros creamos un nuevo objeto de ese modelo y decidimos salvarlo, se produce la creación de un registro de la tabla. Cuando el objeto se modifica y se salvan los datos, se produce el correspondiente `update` en ese registro. Cuando ese objeto se borra, se produce el `delete` sobre ese registro de la tabla.

ORM sería entonces la herramienta de persistencia y Active Record el patrón de arquitectura que se sigue para su construcción. Eloquent es el nombre con el que se conoce en Laravel esta parte del framework, que una vez nos acostumbramos a usar, nos agiliza la mayoría de las operaciones habituales del acceso a bases de datos.

**Nota:** Para el trabajo con el ORM Eloquent necesitamos la misma configuración que ya explicamos en el artículo de [Bases de datos con Laravel](#).

## Crear un modelo Eloquent

Cualquier modelo creado en Laravel es un "Eloquent Model", así que nos permitirá realizar las acciones de acceso y modificación de los datos típicas de las bases de datos relacionales. En realidad muchas cosas sobre los modelos ya las comentamos anteriormente en el artículo de [Introducción a los modelos en Laravel](#). Procuraremos no repetirnos demasiado, así que te recomendamos la lectura de ese artículo previamente.

Los modelos en Laravel se ubican sueltos, dentro de la carpeta "app", aunque si tenemos muchas tablas (y por tanto muchos modelos) sería adecuado realizar algún tipo de orden en esta carpeta. Lo adecuado entonces es que funcione con el modelo de carga de clases que nos ofrece el autoloader de Composer, que básicamente usa los namespaces para definir la localización de subdirectorio donde se encuentra cada uno de los archivos de las clases.

El modo más adecuado para construir nuestros modelos es usando el asistente de Artisan, con la instrucción `make:model` seguida del nombre de nuestro modelo.

```
php artisan make:model Product
```

**Nota:** Recuerda que por convención en Laravel se usan los nombres de las tablas en plural y minúsculas, mientras que en las clases de los modelos debemos colocar mayúscula en la primera letra (camelcase en realidad) y acabado en singular. Por ejemplo, para una tabla llamada "Groups", el nombre del modelo será "Group". Luego veremos cómo modificar esa convención en el caso que sea necesario.

Nuestro modelo "Product" como se ha dicho, corresponde con una clase, que se colocará en la carpeta "app" y tendrá el siguiente código.

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model
{
    //
}
```

A pesar que está prácticamente vacío de código es importante señalar que los modelos de por si ya hacen bastante cosas, gracias a que heredan el comportamiento a partir de la clase `Illuminate\Database\Eloquent\Model`.

## Modificar el nombre de la tabla

Por convención en Laravel, si no se especifica nada, el modelo Product se correspondería, o trabajaría, con la tabla products. ¿Pero qué se hace si ya nos vienen dados los nombres de las tablas y no se ajustan a esta convención?

Si queremos modificar el nombre de la tabla con la que trabaja un modelo podemos indicarlo a partir de la definición de la propiedad `$table`, asignando el valor que tiene la tabla asociada con este modelo.

```
protected $table = 'My_categories';
```

Esa declaración de propiedad se coloca generalmente al principio del código de la clase.

## Modificar la clave primaria

Las claves primarias en Laravel también tienen su convención. Básicamente toda tabla tiene una columna llamada "id" que será "autoincrement" y que hará las veces de clave primaria. De nuevo, si esto no se ajusta a nuestra realidad solo tenemos que definir una propiedad llamada `$primaryKey`, asignando el nombre del campo que actúe como "Primary Key".

```
protected $primaryKey = 'id_categoria';

Timestamps "created_at" y "updated_at"
```

Cuando hablamos de [Migraciones en Laravel](#), se mencionaron muy rápidamente los campos "timestamps". Estos campos, como su nombre indica, sirven para llevar marcas de tiempo.

**Nota:** Al crear una tabla podemos indicarle que nos cree los correspondientes campos timestamp "created\_at" y "updated\_at", simplemente indicándolo con el método `timestamps()`. Recuerda que una

creación de una tabla en el archivo de migraciones se definía más o menos así:

```
Schema::create('carpetas', function (Blueprint $table) {  
    $table->increments('id');  
    $table->string('nombre');  
    $table->text('descripcion');  
    $table->timestamps();  
});
```

Como resulta obvio, estos timestamps nos llevan la fecha de creación y actualización de un registro. Al nivel de un modelo y por convención son necesarios para que todo funcione correctamente, porque Eloquent los actualiza automáticamente. Si no existen esos campos, al realizar las operaciones de escritura sobre la tabla, nos arrojará un error diciendo que no ha encontrado tales columnas.

En el caso puntual, para los modelos donde no necesitemos que estos campos lleven la cuenta del tiempo, porque es algo que no vemos necesario en todas las tablas, podemos evitar este comportamiento automático simplemente definiendo la propiedad `$timestamps` al valor `false`.

```
public $timestamps = false;
```

Otra cosa que podemos modificar es el formato de la fecha almacenada en estos campos. Es algo que generalmente no necesitarás, pero se puede marcar a través de la propiedad `$dateFormat`. Consulta la documentación en este caso.

## Modelo que sobrescribe las convenciones

Veamos el código de un modelo que sobrescribe las mencionadas convenciones:

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;  
  
class CategoryRole extends Model  
{  
    protected $table = 'rol';  
    $primaryKey = 'id_rol';  
    public $timestamps = false;  
}
```

## Conclusión

Ya conoces las generalidades de los modelos en Laravel, usando Eloquent ORM. Has visto que no es necesario escribir mucho código y lo que no has visto todavía es cómo ya nos ofrecen la mayoría de las funcionalidades listas para usar. Esto es gracias a que en Eloquent se realizan las cosas más por convención que por configuración.

En el siguiente artículo estudiaremos cómo trabajar con un modelo desde un controlador para ejecución de las principales operaciones de acceso a los datos.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 12/05/2016  
Disponible online en <http://desarrolloweb.com/articulos/laravel-eloquent.html>

## Cómo usar modelos de Eloquent en Laravel 5

### Cómo podemos usar modelos para recuperar información de las tablas de la base de datos, con el ORM Eloquent en Laravel 5.

Ahora que sabemos [crear y configurar nuestros modelos](#) podemos comenzar a usarlos para recuperar información de la base de datos. Esta es una operación sencilla, ya que realmente nuestros modelos realizarán todo el trabajo pesado de conectarse a la base de datos y ejecutar las correspondientes consultas para obtener o modificar información.

Parte de la operativa de acceso a los datos ya se conoce, puesto que le dimos un repaso inicial en el capítulo de [QueryBuilder](#). En el caso de los modelos, cuando recuperas información tienes casi los mismos métodos que conoces de Query Builder, con la diferencia que no necesitas especificar la tabla, puesto que ya la sabe el modelo. Algunas cosas cambian pero pocas, como verás a continuación.

Obviamente, para que esto funcione tendrás que tener creado no solo el modelo, sino también la tabla correspondiente asociada en el sistema gestor de base de datos, mediante la correspondiente [migración](#).



### Namespace del modelo

Recuerda que los modelos están creados dentro de la carpeta "app" de la aplicación, sueltos ahí. Sus clases forman parte del Namespace App y para usarlos primero debemos indicar que vamos a usar tal espacio de nombres.

```
use App\User;
```

Eso nos indicará que estamos usando un modelo llamado User, que está en la carpeta "app", como todos los modelos. Gracias al Autoload se cargará ella sola. A partir de entonces ese modelo lo podremos usar

directamente con el nombre de la clase.

**Nota:** Si decidimos no declarar el "use" del namespace, podemos usar el modelo igualmente, siempre que indiquemos el nombre de la clase precedida de su espacio de nombres. "App\User".

## Usar un modelo

Un modelo se podrá usar desde cualquiera de las clases de Laravel, como middlewares o seeders, pero lo más habitual es usarlo desde los controladores. Una vez declarado su ruta mediante el espacio de nombres podremos ejecutar diversos métodos generalmente estáticos para recuperar información.

**Recuperar todos los registros:** Podremos recuperar todos los registros de una tabla con el método `all()`.

```
Articulo::all();
```

Lo que obtenemos será una colección, que podremos usar para lo que sea necesario. Por ejemplo lo podríamos enviar a una vista para que presentarlo en pantalla. Eso lo puedes ver en el siguiente controlador.

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

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Articulo;

class ArticulosController extends Controller
{
    public function index(){
        $articulos = Articulo::all();
        return view('articulo', [
            'articulos' => $articulos
        ]);
    }
}
```

**Nota:** El tema de las colecciones es muy interesante. Las podemos usar como si fueran arrays simples, pero realmente son objetos cuyas funcionalidades van mucho más allá, permitiendo una gama importante de operaciones. Hablaremos más adelante sobre colecciones con detalle.

**Recuperar un registro:** Si lo que quieres hacer es buscar un registro en concreto de este modelo existe un método muy sencillo que permite recuperar un único elemento dado un id de clave primaria.

```
Articulo::find(18);
```

Esto nos devolverá los datos del artículo con identificador 18. En concreto lo que devuelve esta función es un modelo, un objeto de la clase `App\Articulo`. Como es un modelo dispondremos de los métodos que nos facilitan los modelos de Eloquent, pero también lo podremos usar como si fuera un simple array con los valores del artículo en cuestión que acabamos de recuperar. Por ejemplo podríamos pasar esos datos a una vista, con el siguiente código de función en el controlador de antes "ArticulosController".

```
public function muestraId($id){
    $articulo = Articulo::find($id);
    return view('articulo.muestra', [
        'articulo' => $articulo
    ]);
}
```

**Recuperar artículos mediante condiciones complejas:** Como hemos dicho, los modelos permiten una gama de métodos que ya se conocen de QueryBuilder, así que podemos usarlos para acceder a juegos de datos más específicos.

Con este método estaremos recibiendo una colección con todos los artículos escritos dentro del último año.

```
public function index(){
    $articulos = Articulo::where('fecha', '>', time() - (365*24*3600))->get();
    return view('articulo.index', [
        'articulos' => $articulos
    ]);
}
```

Tal como aprendimos, mediante QueryBuilder podríamos encadenar diversos métodos para realizar búsquedas más complejas. Ahora vamos a variar este mismo código para sacar los 10 últimos artículos (según su fecha de publicación) que comienzan por la palabra "introducción"

```
public function index(){
    $articulos = Articulo::where('nombre_articulo', 'like', 'introduccion%')
        ->orderBy('fecha', 'desc')
        ->take(10)
        ->get();
    return view('articulo.index', [
        'articulos' => $articulos
    ]);
}
```

## Acceso a los campos de un registro mediante propiedades de un modelo

Cuando tenemos un objeto modelo, una instancia de la clase de un modelo de Eloquent, podemos acceder a sus propiedades y así estaremos accediendo a los campos del registro seleccionado en el modelo.

No deberíamos escribir salida directamente desde un controlador, pero sirva este ejemplo de muestra del acceso a propiedades de una instancia del modelo.

```
public function muestraId($id){
    $articulo = Articulo::find($id);
    echo $articulo->nombre_articulo;
    echo '<br>';
    echo $articulo->descripcion_articulo;
}
```

## Modificar un registro dado

En el artículo anterior explicamos brevemente el patrón Active Record de arquitectura de software. Dijimos que proponía realizar operaciones para la persistencia de datos con métodos como `save()` o `delete()`. Estos métodos nos servirán en Eloquent para poder almacenar información que haya dentro de un modelo.

Ponte el caso que queremos recuperar un artículo para modificar su título. Lo primero será hacerse con un objeto de la clase del modelo `App\Articulo`, que contenga el artículo que se desea modificar. Luego cambiaremos las propiedades del objeto tal cual, asignando nuevos valores a las propiedades deseadas. Por último invocaremos el método `save()` para que esas modificaciones se almacenen en la tabla asociada.

```
public function muestraId($id){
    $articulo = Articulo::find($id);
    $articulo->nombre_articulo = 'Este es el nuevo titulo';
    $articulo->save();
}
```

## Borrar un registro dado

De una manera muy similar borramos un registro de la tabla. Accediendo a la instancia del modelo, cargando el registro que se desea borrar, y luego invocando el método `delete()`.

```
public function borraId($id){
    $articulo = Articulo::find($id);
    $articulo->delete();
}
```

## Crear un nuevo registro

Para crear un nuevo registro en la tabla del modelo basta con crear una instancia del modelo, mediante su constructor. Luego asignamos todos los valores a ese nuevo modelo a través de sus propiedades y por último invocamos el método `save()` para que los datos se almacenen. Solo al hacer el `save()` el registro se guarda en la base de datos.

```
public function crear(){
    $articulo = new Articulo;
```

```
$articulo->nombre_articulo = "Este es el título";
$articulo->descripcion_articulo = "Esta es la descripción";
$articulo->save();
echo 'creado artículo con id: ' . $articulo->id;
}
```

Como alternativa podríamos enviar todos los datos del nuevo artículo como un array asociativo a la función constructora del modelo. Sin embargo, por motivos de seguridad esta operación tiene sus limitaciones.

El código de la función en el controlador tendría una forma como esta:

```
public function crear(){
    $articulo = new Articulo([
        'nombre_articulo' => 'Este es el título',
        'descripcion_articulo' => 'Esta es la descripción'
    ]);
    $articulo->save();
    echo 'creado artículo con id: ' . $articulo->id_articulo;
}
```

Sin embargo encontraremos que ejecutarlo tal cual nos devuelve el error "MassAssignmentException in Model.php". La solución pasa por especificar en el modelo qué campos de la tabla artículo queremos permitir que se seteen mediante esta alternativa con el constructor.

Este sería el código de mi modelo, una vez indicada la posibilidad de setear esos campos mediante el constructor del modelo.

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Articulo extends Model
{
    protected $fillable = ['nombre_articulo', 'descripcion_articulo'];
}
```

## Conclusión

Has aprendido a usar los modelos para la realización de un completo conjunto de posibilidades. Seguro que apreciarás las ventajas y facilidades de trabajar contra el ORM Eloquent.

Todavía nos quedan cosas importantes que estudiar, como el tema de las relaciones entre tablas. En breve nos pondremos con ello, de momento te recomendamos tomarte el tiempo para practicar lo aprendido.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en *18/05/2016*  
Disponible online en <http://desarrolloweb.com/articulos/usar-modelos-eloquent-laravel5.html>

## Relaciones en los modelos Eloquent

**Entender cómo un ORM en general gestiona las relaciones entre tablas y cómo Eloquent en particular nos permite acceder a información relacionada como si estuviera en el propio modelo.**

En modelos Eloquent, el ORM de Laravel que venimos abordando en diversos artículos del [Manual de Laravel](#), hemos comprobado que prácticamente no hemos colocado código alguno para que toda la "magia" de acceso a los datos funcione. Es algo que realmente llama la atención, porque nos permite mucha agilidad en el desarrollo.

Pero en el área de relaciones entre tablas vamos a tener que ayudar de algún modo al framework para que realice su increíble trabajo. De todos modos, como verás en este artículo y los siguientes, se tratará solamente de realizar unas pocas declaraciones y con ello podremos conseguir nuestro objetivo de acceder a datos relacionados, sin tener que realizar las consultas de acceso a la información.

El objetivo es conseguir que nuestros modelos, aparte de los datos que contienen las propias tablas de la base de datos con las que están asociadas, permitan acceder a los datos de todas las tablas que se hayan relacionado en la definición de la base de datos. Su acceso se realizará por medio de propiedades, de modo que en la práctica podrá parecer que esos datos pertenecen al propio modelo, aunque realmente sean datos que están en otras entidades.



## Relaciones soportadas por Eloquent

Como sabes, existen diversos tipos de relaciones en bases de datos, las básicas que todo el mundo debe conocer son:

- Relaciones de uno a uno: Por ejemplo, un usuario tiene un perfil de usuario. El usuario tiene un perfil y el perfil solo pertenece a un usuario.
- De uno a muchos (1 a n): Por ejemplo, un usuario tiene posts. El usuario puede cargar un número indeterminado de post, pero un post sólo pertenece a un usuario.
- De muchos a muchos (n a m): Por ejemplo, un artículo tiene varias etiquetas y una etiqueta puede tener varios artículos. Estas relaciones generan una tabla adicional generalmente que Eloquent maneja para ti.

A estas relaciones debemos agregarle otras que Eloquent soporta, no tan comunes pero que también se encuentran en la vida real.

- Relaciones de pertenencia a través de otras entidades: Estas relaciones no son más que las relaciones que conoces de toda la vida, pero saltando a través de otras tablas. En un esquema relacional la tabla "a" se relaciona con "b" y a su vez "b" se relaciona con "c". En un caso como este, con Eloquent podemos conseguir definir una relación directa desde "a" hacia "c", pasando a través de "b". Ese paso "a través" se realiza de manera transparente para el desarrollador. Es decir, es como si la tabla "a" estuviera directamente vinculada a "c".
- Relaciones polimórficas: esto es algo muy interesante también y se basa en un esquema de relaciones que quizás hayas tenido que resolver en algunas ocasiones "a mano" y que Laravel te hace de manera automática. Básicamente consiste en una relación donde aquella entidad con la que estás relacionando pueda ser variable. Por ejemplo, un usuario puede dar el "me gusta" a post de otro usuario, a un comentario de otro usuario, a un vídeo, a un artículo en venta... en general, a un número de entidades indeterminado y variable. De ahí el polimorfismo.

Ten en cuenta que las relaciones en la base de datos se definen a la hora de hacer migraciones. Tenemos un artículo entero dedicado al [tratamiento de índices en las migraciones](#) en el que hablamos de los índices de claves foráneas. Nuestro primer paso entonces sería establecer correctamente esas relaciones en el modelo de la base de datos, puesto que Eloquent lo necesita para su correcto funcionamiento.

A partir de ahí, podemos declarar relaciones en los modelos. Esto lo conseguimos por medio de funciones en las que básicamente informamos que tal entidad pertenece a tal otra, o que a tal entidad se le asocian muchos ejemplares de tal otra. Dependiendo del tipo de relación esta declaración de función se realizará de una manera o de otra.

## Ejemplo básico sobre las ventajas de un ORM

Antes de ponernos a ver los tipos de relaciones y cómo se declaran en Eloquent creo que puede estar interesante incidir de nuevo en cómo un ORM ayuda en el acceso a los datos que están relacionados.

Piensa en este sencillo ejemplo. Tenemos un comercio electrónico en el que manejamos productos. Los productos pueden tener muchos comentarios.

El ORM permite acceder a los productos y sus datos mediante objetos.

```
$producto1 = Producto::find(1);  
// Ahora el producto me permite acceder a sus datos  
echo $producto1->nombre; //muestra el nombre  
echo $producto1->descripcion; //muestra la descripción
```

Pero además, si definimos la relación en el modelo Eloquent, también podremos acceder a los datos de las tablas relacionadas. Acceder a los comentarios de un artículo es tan sencillo como acceder a un método. Esta sería una colección con todos sus comentarios:

```
$producto1->comentarios()
```

**Nota:** Además es posible que al hacer el primer acceso a los productos ya estés indicando que quieres

recibir sus comentarios como una de las propiedades de ese producto. Esto se consigue con una funcionalidad que se llama "Eager Loading" que también explicaremos más adelante porque resulta de mucha utilidad.

Podré usar esa colección para recorrer los comentarios, mostrarlos en la página o hacer lo que sea necesario en cada caso. En ningún caso tendré que escribir una consulta o realizar algún tipo de operación para traerme esos datos. Simplemente declaro la relación en el modelo y esos datos estarán a mi disposición siempre que lo necesite.

Es interesante mencionar que el acceso a las tablas relacionadas se hace mediante un esquema de trabajo que se conoce como "Lazy Load" o "Carga Perezosa". Eso quiere decir que, cuando se generó el modelo con los datos del producto no se cargaron directamente los comentarios. Solo en el momento en el que se intente acceder a la propiedad donde se encuentran talas comentarios es cuando Eloquent hace el trabajo de acceder a la tabla relacionada y recuperar la información.

Obviamente, este esquema de lazy load es beneficioso porque no siempre que accedes a los datos de un producto necesitas realmente que te entreguen sus comentarios. Por ejemplo, al mostrar un producto en el carrito de la compra, no necesitas saber qué comentarios dieron los usuarios a ese producto. Pero a veces sí que es interesante que esos datos relacionados se entreguen directamente y estén disponibles y precargados. Esto lo veremos más adelante, pero en Laravel podrás indicar que una consulta ya te entregue determinada información de sus tablas relacionadas. En resumen, lazy loading es el comportamiento predeterminado, pero al recuperar un modelo podemos decirle explícitamente que cargue también de inicio información de tablas que están relacionadas, como se comentó en la nota anterior.

En los siguientes artículos vamos a ir abordando cada uno de los tipos de relaciones para ver cómo se declaran en un modelo.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 26/05/2016  
Disponible online en <http://desarrolloweb.com/articulos/relaciones-modelos-eloquent.html>

## Relaciones de 1 a 1 en Laravel Eloquent

### Cómo implementar relaciones de 1 a 1 en modelos de Eloquent, desde la creación de las migraciones, modificación de los modelos y su uso.

En este artículo nos vamos a introducir en una parte más práctica, abordando cómo debemos configurar relaciones en una aplicación Laravel, usando el ORM Eloquent. Es importante que hasta este punto tengas una idea de lo que es Eloquent, como usar modelos en Laravel y otras cosas que hemos tratado ya en el [Manual de Laravel](#).

En este artículo veremos el flujo completo para tratar una relación de 1 a 1 en Laravel, comenzando por la creación de una migración y posteriormente la definición de las relaciones dentro de los modelos, así como el uso de esos modelos. Aunque Laravel te ayuda mucho y Eloquent necesita pocas configuraciones, no es un tema trivial, así que trataremos de explicarlo detenidamente.



## Relaciones de uno a uno

Las relaciones de uno a uno no son las más habituales, y de hecho muchas veces se evitan, pero tenemos ejemplos típicos que se repiten en diferentes aplicaciones: por ejemplo, un usuario tiene un perfil de usuario.

La relación de 1 a 1 se define en ese caso porque un usuario solo puede tener un perfil de usuario y porque un perfil de usuario solo puede pertenecer a un usuario.

**Nota:** Este tipo de relaciones a veces podrían suponer juntar ambas tablas en una. En el caso del usuario y su perfil podríamos tener ambos conjuntos de datos en la tabla users, pero podría haber diversos motivos por los cuales separar esta información en dos tablas. Por ejemplo motivos de almacenamiento, ya que no todos los usuarios pueden tener perfil definido y quizás no queramos meter en la tabla de usuario muchos campos que puede que no se rellenen la mayoría de las veces. Incluso así, juntar ambas tablas en una puede suponer mejora de rendimiento, cuando la información siempre se va a consultar al mismo tiempo.

En términos de programación no encontraremos muchas diferencias en lo que supone el trabajo de acceso a ambas tablas, gracias a Eloquent. Es decir, si definimos correctamente los modelos, poco nos importará que estas informaciones se encuentren en tablas separadas.

## Migración a la hora de definir una relación de 1 a 1

En la definición de las tablas, programada en las correspondientes migraciones, debemos comenzar a definir las relaciones, ya que Laravel toma esa información para poder configurar los modelos.

En el ejemplo que nos ocupa tenemos dos tablas: users y profiles. La migración de la tabla users nos la dan hecha en Laravel, puesto que el propio framework ya implementa unos [mecanismos para poder registrar y loguear usuarios](#). Por tanto, solamente necesitaremos definir la migración de la tabla Profile.

En la migración colocaremos los campos de la tabla perfil, para almacenar todos los datos que necesitemos y la definición de la clave foránea.

```
Schema::create('profiles', function (Blueprint $table) {
    $table->integer('user_id')->unsigned();
    $table->primary('user_id');
    $table->text('bio');
    $table->string('company', 150);
    $table->string('technologies', 200);
});
```

```
$table->timestamps();
$table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
});
```

Es interesante apreciar que nuestra tabla tiene el campo "user\_id". En la última línea de la definición de esta tabla, verás definida la clave foránea como referencia al campo "id" de la tabla "users". Además, para evitar que se puedan borrar usuarios y se queden "colgados" sus perfiles sin referencia, hemos definido un "on delete cascade". Eso provocará que los perfiles se borren automáticamente cuando se borre un usuario, algo muy práctico para ahorrarnos realizar esa operación doble de borrado y sobre todo importante para que debido a un olvido estemos borrando usuarios sin borrar sus perfiles.

**Nota:** He definido también como clave primaria user\_id. Esto no sería absolutamente necesario, pero realmente desde el punto de vista de una clave primaria, user\_id cumple con los requisitos, pues no se deben producir dos perfiles para un mismo usuario. Esto ayudará a respetar la integridad de los datos, evitando que se puedan insertar dos perfiles para el mismo usuario por error. Pero para ese mismo cometido podríamos también haber definido un índice único. Esto es más un criterio de diseño de base de datos que de Laravel, pues el framework no necesitaría que se definiese ese campo como clave primaria para que la relación de 1 a 1 se pueda configurar.

## Declaración de las relaciones para la configuración de los modelos

En Laravel, la declaración de relaciones se realiza mediante una función, que se suele colocar en ambos modelos pertenecientes a ambas tablas relacionadas. No obstante, no es una necesidad realizar estas funciones, puesto que solo harán falta si realmente necesitas acceder desde un modelo a su información relacionada. Es decir, si entre mis operaciones de acceso a la información de un usuario necesito acceder al perfil, entonces configuraré la relación en el modelo del usuario. Pero si nunca voy a necesitar, dado un perfil, acceder a los datos del usuario al que pertenece, no necesitaría configurar esa relación en el modelo del perfil.

Todas las funciones responden a un patrón similar de declaración, aunque usaremos diversos métodos ayudantes dependiendo del tipo de relación. Y esto vale para todos los tipos de relaciones, de 1 a muchos, de muchos a muchos, etc. De momento nos centramos en las relaciones de 1 a 1, pero mucho de lo que aprenderás ahora se puede aplicar de manera muy similar a otros tipos de relaciones. Lo verás más claro con un poco de código.

## Definir la relación desde el usuario hacia el perfil

Nuestro modelo User debe informar que tiene un perfil asociado. Esta relación la querrás configurar siempre, porque realmente es muy necesario que dado un usuario puedas acceder a su perfil.

Es tan sencillo como agregar este método a la clase User, definida en app/user.php

```
public function profile() {
    return $this->hasOne('App\Profile');
}
```

Puedes ver el patrón que usaremos en esta y otros tipos de relaciones. Usamos un método con el nombre de aquel modelo que estamos relacionando. (function profile() para relacionar con el modelo Profile). Como código de la función colocamos un return sobre lo que te devuelve el método hasOne(), indicando a este método el modelo con el que queremos relacionar con su namespace.

**Nota:** En esta ocasión el tipo de relación está definida por el método hasOne(), pero en otros casos usaremos otros métodos como hasMany() o belongsTo().

Obviamente, para que esto funcione, aparte del modelo User, que te lo dan hecho, tendrás que crear el modelo Profile, que sería nuevo. En el artículo de [crear modelos Eloquent](#) está detallado el proceso.

También es importante mencionar que para que esta declaración de relación salga correctamente, deben darse varios patrones de nombrado de modelos, tablas, claves, etc. Eloquent asume que la clave foránea de la tabla que relacionamos debe corresponder con la clave primaria "id" de la tabla de la que partimos (en este caso clave foránea en la tabla profiles "user\_id" debe corresponder con la el campo "id" de la tabla users, o aquel campo definido como \$primaryKey). Si no es este el caso, podemos definir la relación indicando parámetros adicionales:

```
return $this->hasOne('App\Profile', 'clave_foranea', 'clave_local_a_relacionar');
```

### Definir la relación inversa, del perfil hacia el usuario

Si lo consideras necesario, también podrías definir en el modelo Profile la relación hacia el usuario al que pertenece.

**Nota:** Esto lo hemos mencionado antes, pero insistimos en que la decisión de definir o no esta relación depende de tus necesidades. En concreto en este caso esa configuración no siempre será necesaria, puesto que generalmente necesitarás acceder a un perfil de un usuario dado y no a un usuario de un perfil dado.

Esta relación de pertenencia (un perfil pertenece a un usuario) se define mediante el método belongsTo().

```
public function user()
{
    return $this->belongsTo('App\User');
}
```

El esquema de trabajo es muy similar. Usamos un método user, porque es el usuario el que se querrá relacionar al perfil. Luego usamos belongsTo() indicando el modelo con el que estamos relacionando. Del mismo modo, se deben dar una serie de patrones asumidos por Laravel en nombres de identificadores, claves, etc. Si no es así podemos invocar al método belongsTo() con parámetros adicionales. Esto lo hemos explicado para la relación anterior (usuario a perfil), pero te sugerimos leer la documentación para mayores

informaciones.

## Usar modelos con tablas relacionadas

Ya introdujimos en un artículo anterior que, una vez definidos los modelos correctamente, para dar soporte a las relaciones deseadas, podremos [acceder a los datos de las tablas relacionadas como si fueran datos del propio modelo](#). Es decir, una vez realizados los pasos descritos anteriormente, ya solo nos queda disfrutar de las facilidades que nos aporta el ORM.

De todos modos, como colofón a este artículo vamos a presentar varios códigos que realizan operaciones típicas que queremos realizar con este tipo de relaciones.

Para empezar, si vas a hacer uso de ambos modelos, lo primero que tendrás que hacer es declarar los correspondientes "use" de sus espacios de nombres.

```
use App\Profile;  
use App\User;
```

**Nota:** Recuerda que el namespace de los modelos en Laravel está en App, ya que los modelos se encuentran generalmente sueltos en la carpeta "app", pero nosotros podríamos decidir otro tipo de organización si lo consideramos así.

**Crear un perfil y asociarlo a un usuario:** Esta operativa la realizas creando un perfil mediante el modelo y luego asignando ese perfil a un modelo de usuario. No te olvides de salvar el usuario luego.

```
$perfil = new Profile;  
$perfil->bio = 'Esta es la biografía de un usuario';  
$perfil->company = 'EscuelaIT';  
$perfil->technologies = 'PHP, Javascript, Apache, HTML';  
$user = User::find(1);  
$user->profile()->save($perfil);
```

**Borrar un usuario y su perfil:** Realmente, si hemos definido el "delete cascade" no necesitamos más que borrar el usuario y el perfil se borrará automáticamente.

```
$usuario = User::find(1);  
$usuario->delete();
```

**Acceder al perfil de un usuario y comprobar su existencia:** El acceso a las tablas relacionadas, una vez definida la correspondiente relación en el modelo (hasOne / belongsTo), se hace como si fueran propiedades del propio modelo.

```
$usuario = User::find(2);
```

```
$perfil = $usuario->profile;
if($perfil) {
    echo 'tengo perfil ' . json_encode($perfil);
} else {
    return 'no tiene profile';
}
```

En esta operación cabe señalar dos detalles:

1. El dato del perfil es "lazy load", a no ser que especifiquemos lo contrario. Veremos más adelante cómo.
2. Una vez hemos accedido a la propiedad profile del usuario (\$usuario->profile) en el objeto \$usuario ya se encontrarán los datos del perfil. Por ello, las siguientes veces que se acceda a esa propiedad Laravel no necesitará irse de nuevo a la base de datos para buscar el perfil.

**Desde un perfil, acceder al usuario al que pertenece:** Si hemos definido también la relación desde el perfil al usuario (belongsToMany) podremos acceder desde un perfil al usuario al que le pertenece.

```
$perfil = Profile::find(1);
$user = $perfil->user;
```

## Conclusión

Hemos aprendido muchas cosas nuevas sobre Eloquent y las relaciones. Tendrás que practicar con todo esto para asimilar realmente toda la información.

En el futuro trataremos con otros tipos de relaciones, más habituales como es el caso de uno a muchos o de muchos a muchos, pero verás que todo es bastante similar a lo que hemos explicado en este artículo. En resumen, si has podido entender este tipo de relación, las siguientes serán mucho más fáciles de asimilar.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 01/07/2016  
Disponible online en <http://desarrolloweb.com/articulos/relaciones-1-laravel-eloquent.html>

## Servicios en Laravel

Servicios integrados dentro del propio framework Laravel, por medio de librerías mantenidas por el propio equipo de desarrollo. Están enfocados en el desarrollo con partes importantes de una aplicación. Estos servicios, aunque son de uso habitual, no forman parte del core de Laravel, por lo que unas aplicaciones pueden requerir usarlos y otras no.

### Instalar y modificar sistema de autenticación de usuarios en Laravel

**Cómo crear el sistema de autenticación de usuarios en Laravel 5.2, instalando el sistema predeterminado del framework para que se puedan logear en la aplicación con usuario y contraseña.**

En Laravel 5.2, igual que ocurre con la versión 5.1, no se encuentra disponible el sistema de usuarios en la instalación inicial, por lo que tenemos que instalarlo a mano. Es un proceso bastante sencillo, porque realmente está casi todo el trabajo pesado ya realizado de antemano.

En este artículo explicaremos cómo disponibilizar de nuestro sistema de autenticación, paso por paso. Verás que la mayoría de los pasos consisten en restaurar algunos componentes, como las tablas mediante el sistema de migraciones. Tendremos además que crear algunas vistas con los correspondientes formularios, rutas y poco más. Insistimos, aunque pueda parecer mucho trabajo, mediante varios comandos Artisan es una tarea que se puede realizar en minutos.



Resumimos el procedimiento de la documentación oficial en <https://laravel.com/docs/5.2/authentication>. Explicaremos paso por paso el procedimiento. Sin embargo, conviene estudiar el anterior enlace para encontrar más detalles.

**Nota:** En nuestro ejemplo partimos de una instalación de Laravel realizada mediante la máquina virtual [Homestead](#). En este caso está ya preparada la información de conexión a la base de datos, así como otras cosas que ya se han comentado en el [Manual de Laravel](#).

### Comando para generar las rutas y vistas

Comenzamos generando lo que sería un "scaffolding" de aplicación, es decir, una estructura básica para que nuestra instalación de Laravel comience a parecerse a una aplicación capaz de autenticar usuarios. Consiste en una serie de rutas y vistas, las cuales son necesarias para hacer funcionar el sistema de autenticación.

Esto se resume en un único comando que nos ofrece Artisan. Lo ideal es ejecutar el comando en una instalación de Laravel recién hecha.

```
php artisan make:auth
```

Observarás, una vez ejecutado el comando, que también se han generado varias vistas en `resources/views/auth`. Además hay un archivo "layout" que contiene la plantilla general de la aplicación. Lo encuentras en la ruta `resources/views/layouts`. Si lo abres verás que usa el framework de diseño Bootstrap.

**Nota:** Quizás tengas que hacer algo de limpieza para eliminar Bootstrap o sustituirlo por tu librería preferida. Es una decisión personal mantenerlo o no y generalmente depende de gustos y costumbres. En general las tareas de eliminar o personalizar Bootstrap no son muy complicadas pero de momento quedan fuera de este manual.

También se han generado varios controladores. Por una parte tenemos el `Auth/AuthController`, que nos implementa mecanismos de logueo, `Auth/PasswordController` para los password reset y un `HomeController` que nos provee del código para implementar el "dashboard" (La traducción sería escritorio, generalmente usado como punto de inicio de un panel de control de usuarios) en la ruta `/home`.

En `Auth/AuthController` hay una variable que nos permite decidir hacia donde se redirigen a los usuarios que acaban de loguearse. Inicialmente están redireccionados hacia la raíz del sitio web, pero podrías cambiarlo para que se redireccionen hacia el dashboard.

```
protected $redirectTo = '/home';
```

## Guards

Existen diversos sistemas para mantener el estado de autenticación de un usuario. Laravel ya tiene implementados dos sistemas que son los más habituales en aplicaciones web y en el desarrollo de APIs. Las sesiones son habitualmente los mecanismos para usuarios que se autentican personalmente en la aplicación. Mientras que si se desarrolla un API lo común es usar los "Token".

Generalmente no tendrás que tocar nada, pero se administran desde `config/auth.php`.

## Comando para las migraciones

Estas migraciones están disponibles en el sistema. Al generar el proceso de Login las migraciones se ejecutan, para incorporar al sistema las tablas de usuarios y de resets de clave. Recuerda que si tuvieras que ejecutar las migraciones tienes disponible el comando.

```
php artisan migrate
```

**Nota:** Si hemos instalado Laravel con Homestead lo lógico es que ejecutes ese comando desde la máquina virtual, a la que te has conectado con ssh (vagrant up y luego vagrant ssh). Lo harás desde la raíz del proyecto y no debería fallar, porque está todo ya configurado gracias a Homestead).

Deberíamos obtener la siguiente salida:

```
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
```

### Modificar el formulario de registro para agregar campos

Es una práctica habitual tener un campo "apellidos" ("last\_name" llamaremos a este campo) en el formulario de registro, para poder separar el nombre y los apellidos del usuario. A continuación te explicamos los pasos para conseguirlo.

**Migraciones:** Por ello podrías hacer una migración para incluir el campo en la tabla de usuarios. Comienzas con el comando:

```
php artisan make:migration user_apellidos_migration
```

Luego tienes que generar los cambios deseados en la tabla "users".

```
public function up()
{
    // Creo un campo apellidos "last_name"
    Schema::table('users', function($table){
        $table->string('last_name', 250);
    });
}

public function down()
{
    // Elimino el campo last_name
    Schema::table('users', function ($table) {
        $table->dropColumn('last_name');
    });
}
```

Ejecutamos esta migración con el comando de artisan:

```
php artisan migrate
```

**Nota:** En pasados artículos podrás encontrar [más información sobre el mecanismo de migraciones](#).

**Cambiar la vista:** El segundo paso consiste en cambiar la vista del formulario de registro, que encuentras en la ruta: `resources/views/auth/register.blade.php`

Básicamente agregamos un bloque de formulario para mostrar el campo de texto para los apellidos. Obviamente no solo el campo INPUT, sino con todo lo que hay alrededor para mostrar errores de validación, etc. Quedará más o menos así:

```
<div class="form-group{{ $errors->has('last_name') ? ' has-error' : '' }}">
  <label class="col-md-4 control-label">Last name</label>

  <div class="col-md-6">
    <input type="text" class="form-control" name="last_name" value="{{ old('last_name') }}">

    @if ($errors->has('last_name'))
      <span class="help-block">
        <strong>{{ $errors->first('last_name') }}</strong>
      </span>
    @endif
  </div>
</div>
```

**Cambiar el controlador:** Nuestro controlador hace también las validaciones y manda los datos del formulario al modelo. Cambiarlo para agregar este nuevo campo es bien sencillo.

Por un lado modificamos el método `validator()`:

```
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'last_name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|min:6|confirmed',
    ]);
}

Luego tenemos el método que invoca al modelo con los datos.

protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
```

```
'last_name' => $data['last_name'],  
'email' => $data['email'],  
'password' => bcrypt($data['password']),  
]);  
}
```

**Cambio en el modelo:** Por último modificamos el modelo para que se pueda rellenar el campo "last\_name" desde el método create(), que es el que usa el controlador para guardar la información del usuario.

```
protected $fillable = [  
    'name', 'email', 'password', 'last_name'  
];
```

Con esto ya sabes lo básico del sistema de registro y autenticación de usuarios.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 18/04/2016  
Disponible online en <http://desarrolloweb.com/articulos/instalar-modificar-sistema-autenticacion-usuarios-laravel.html>

## Laravel Elixir

**Conjunto de herramientas NodeJS relacionadas con la parte frontend de una aplicación desarrollada con Laravel, enfocadas en la optimización y despliegue del proyecto.**

Laravel Elixir no es un desarrollo con PHP, sino un desarrollo NodeJS. No está pensado para la parte backend, como Laravel en general, sino para la parte frontend. Por todo eso puede parecer un poco raro que un framework PHP incluya algo como Elixir, aunque la verdad es que sirve de mucho y se agradece que esté disponible, porque nos va a ahorrar mucho tiempo de configuraciones en nuestro proyecto.

Elixir se usa durante la etapa de desarrollo, facilitando la tarea de preprocesamiento entre lenguajes como Sass o Less y CSS, o entre Coffee Script y Javascript. También la parte de la transpilación de un hipotético proyecto realizado con ES6. Para todo ello se apoya en un gestor de tareas popular, como es Gulp. En la etapa de despliegue nos permite asimismo optimizar los archivos de la aplicación, compactando CSS o Javascript.



El resultado de ello es mayores prestaciones a lo largo del desarrollo y un sitio más optimizado una vez publicado. Son las mismas ventajas que obtendríamos si usamos Gulp directamente. Si ya estamos acostumbrados a usar Gulp o Grunt en nuestros proyectos podemos prescindir de Elixir, que es simplemente una capa por encima.

Si tenemos un dominio del tooling de NodeJS podríamos personalizar nuestras tareas y adaptarlas a otra serie de necesidades diferentes, pero lo cierto es que hay mucho trabajo realizado detrás de Elixir y muchas configuraciones que seguramente nos vengan bien. Sobre todo Elixir ya está listo para realizar muchas cosas y configurado para adaptarse a una estructura de carpetas definida, la estructura del framework Laravel.

Los desarrolladores de Laravel seguramente disfruten de Elixir y les permita centrarse en la parte que mejor dominan, el desarrollo Backend, sin necesidad de perder prestaciones habituales entre los mejores frontend. Un desarrollador PHP es habitual que tenga dudas con la configuración de la parte frontend de un proyecto y para él será especialmente útil Elixir.

## Instalación de Elixir

En un proyecto recién instalado de Laravel todavía no se encuentra disponible Elixir. Hay que instalarlo aparte, aunque es una tarea que se resume en un sencillo comando. El único detalle es que necesitamos tener disponible NodeJS.

Si hemos instalado Laravel en una [máquina Homestead](#), ya tendremos NodeJS configurado. Si lo hemos instalado en otro ordenador quizás tengamos Node ya disponible. Si no es así, [tendrás que instalarlo con el proceso habitual para tu sistema operativo](#). En general podemos saber si tenemos NodeJS instalado con el comando:

```
node -v
```

Como segundo paso instalaremos Gulp, que es un programa realizado con NodeJS que es un automatizador de tareas, sobre el que está construido Elixir. Lo haces con el siguiente comando.

```
npm install --global gulp
```

Una vez sabemos que está Node y Gulp en nuestro sistema hay que instalar Elixir y todas sus dependencias. Esto es algo sencillo gracias a [npm](#), que es el gestor de paquetes de Node, igual que [Composer](#) es el gestor de paquetes para PHP.

En resumen, npm tiene un archivo de configuración de las dependencias de un proyecto, que se llama package.json. Para instalar todas las dependencias definidas en ese archivo ejecutamos un comando, desde el directorio raíz de tu proyecto Laravel.

```
npm install
```

**Nota:** Comandos adicionales y alternativas de instalación diferentes los podremos encontrar en la

[documentación de Laravel Elixir.](#)

## Configurar Elixir

Dependiendo de las tareas que queramos realizar en la parte frontend, nuestras herramientas habituales y el grado de optimización que busquemos, tendremos que configurar Elixir de una manera u otra.

Esta parte ya es más dependiente del propio NodeJS que de PHP, así que la realizaremos con Javascript. No obstante, es muy sencillo como verás a continuación, por lo que si no sabes Javascript o Node no tienes por qué asustarte.

Cada cosa que queramos hacer, de entre las que nos ofrece Elixir, tenemos que definirla en el archivo `gulpfile.js`, que tienes en la carpeta raíz de tu proyecto.

**Nota:** `gulpfile.js` es un archivo de Gulp donde se definen todas las tareas que se desean automatizar con Gulp. Por tanto ese archivo no es de Laravel propiamente dicho, sino que nos viene del hecho de Elixir estar desarrollado encima de Gulp. Como hemos dicho, contendrá código Javascript.

Elixir nos abstrae de toda la complejidad de crear tareas para Gulp. Ahora ya depende de nuestras tareas el código que debemos crear. Encontramos, entre las posibilidades de Elixir, las siguientes tareas:

Preprocesado de archivos Less o Sass  
Preprocesado de archivos Coffee Script  
Compactado de CSS o Javascript  
Unificación de varios archivos CSS o Javascript en un único archivo, para reducir las solicitudes http al servidor.  
Copiar archivos "assets" a nuestra carpeta de proyecto public.  
Transpilación de ES6 con Babel o Browserify  
Sincronización del navegador, con live reload cuando cambian archivos de la aplicación

La configuración de todas estas cosas es muy sencilla, porque realmente Elixir nos hace todo el trabajo pesado. Simplemente hay que declarar cuáles de estas cosas necesitamos hacer. Vamos a ver un par de ejemplos.

**Nota:** Observa en `gulpfile.js` que se comienza por la línea `var elixir = require('laravel-elixir');` Esto nos sirve para traernos el código del módulo de Elixir. Es gracias a ese módulo que se produce la capa de Elixir encima de Gulp.

**Preprocesado de archivos Sass** Esta opción es interesante de comentar porque ya está configurada en `gulpfile.js` y la podemos tomar de referencia.

Si abres ese archivo "`gulpfile.js`" verás el siguiente código:

```
elixir(function(mix) {  
  mix.sass('app.scss');  
});
```

Eso es lo único que necesitamos hacer para configurar una tarea Gulp aprovechando la capa que Elixir nos pone por encima.

Cuando se ejecute esta tarea Elixir irá a la carpeta `resources/assets/sass` y dentro de ella tomará el archivo `app.scss` y lo procesará, convirtiendo el código Sass en CSS. El archivo CSS resultante lo colocará en `public`, en la carpeta `public/css`. Es en esa carpeta donde tenemos que enlazar el CSS para generar los estilos de nuestro proyecto.

**Unión de varios archivos Javascript** Si estamos usando en nuestra aplicación varios archivos Javascript distintos, con librerías y plugins diversos, podemos unirlos en un solo archivo gracias a Elixir.

La tarea se escribe más o menos así:

```
elixir(function(mix) {
  mix.scripts([
    'app.js',
    'prism.js'
  ]);
});
```

Esto irá a buscar los archivos en `resources/assets/js`. Tomará los archivos Javascript "app.js" y "prism.js". Los unirá en un único fichero llamado `all.js` que colocará en la ruta `public/js`.

En nuestro HTML, en lugar de incluir todos los scripts por separado, enlazaremos con ese único fichero de código Javascript.

Explicaremos otros procesos más adelante. De todos modos, los tienes documentados en la página de Elixir y generalmente es todo muy parecido a lo que has visto. Además es posible tener tareas Gulp, creadas sin Elixir, y ejecutarlas igualmente.

## Ejecutar Elixir

Tenemos un par de mecanismos para ejecutar las tareas de gulp configuradas en el proyecto Laravel.

1. Ejecutar explícitamente las tareas programadas en `gulpfile`
2. Mantener abierto un "watcher" que ejecutará las tareas automáticamente cuando detecte que se deben ejecutar, dado que hayan cambiado los archivos fuente

1) Ejecutamos **Elixir de manera explícita** con el comando:

```
gulp
```

Eso ejecutará todas las tareas de `gulpfile.js` y producirá el resultado. Pero si lo deseas, puedes alterar este comando indicando que también se compacten los archivos resultantes.

```
gulp --production
```

Esto generará los mismos archivos pero con el código compactado, sin comentarios, etc.

2) Mantenemos un **watcher Gulp** activo con el comando:

```
gulp watch
```

Esto estará escuchando cualquier cambio en los assets para procesar las tareas cuando es necesario, sin que tengamos que estar todo el tiempo pidiéndolo explícitamente a Gulp. Los resultados son los mismos, solo que el watcher no nos permite indicar que se desea compactar el resultado, algo que no es problema porque el watcher lo usarás solo para la etapa de desarrollo.

## Conclusión

De momento es todo. Con esto tendrás un flujo de trabajo ágil para gestión de los componentes frontend de tus aplicaciones Laravel. Además, cuando publiques tu proyecto te garantiza una buena optimización de los assets, lo que también será muy de agradecer.

Ahora se trata de examinar un poco más de cerca la documentación de Laravel y ver cómo se programan y configuran las tareas Gulp con Elixir.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en *04/05/2016*  
Disponible online en <http://desarrolloweb.com/articulos/laravel-elixir.html>

# Paquetes de terceros para extender Laravel con funcionalidades extra

En los siguientes capítulos veremos paquetes externos realizados por otros desarrolladores, que podemos integrar dentro de Laravel para implementar diversas funcionalidades útiles.

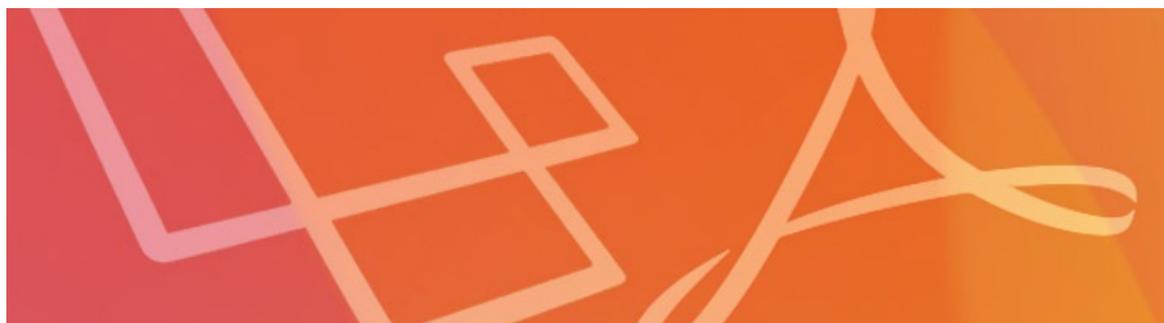
## Generar PDF en Laravel con DomPDF

**Cómo crear documentos en formato PDF desde Laravel, usando la librería DomPDF, a partir de código HTML.**

La librería DomPDF es una sencilla alternativa para la construcción de PDF en PHP. Nos ofrece la posibilidad de crear los documentos PDF a partir de código HTML, que puede residir en un archivo, una cadena de texto, etc. A partir de ese código HTML crea un documento PDF que se puede descargar, almacenar en el servidor o cualquier otra operación que se desee.

Alternativas para crear PDF en PHP hay varias, pero esta es interesante por no depender de otros programas de compleja instalación. Genera el PDF con bastante fidelidad a partir del código HTML y soporta casi todo el CSS de nivel 2 y algo de CSS 3. De todos modos, si no es la que te interesa, puedes consultar otros artículos de DesarrolloWeb.com donde ya hemos tratado la [creación de PDF por ejemplo con FPDF](#).

Yo la seleccioné porque necesitaba una herramienta que me permitiese crear documentos con cabecera y pie de página, que se repitieran a lo largo de todas las páginas del PDF. Eso se consigue a partir de un código HTML sencillo que luego veremos.



La mayor precaución que se debe tomar para usar la librería DomPDF con mejores resultados es tener un HTML de origen bien formado. Se lleva mal con el HTML incorrecto, por lo que no es mala idea contar con alguna otra librería como Tidy o HTML Purify que pueden ayudar al filtrar cosas y saber si el HTML está correcto.

## Instalación de DomPDF para PHP

La instalación se realiza en Laravel por medio de un “wrapper” o envoltura, que nos ofrece directamente el “service provider” que necesitamos para integrarlo en el framework. Lo podemos instalar usando Composer, que ya sabemos es el gestor de paquetes para PHP.

En el composer.json agregamos esta dependencia en el objeto “require”

```
"barryvdh/laravel-dompdf": "0.6.*"
```

Luego ejecutamos en la terminal, desde la raíz del proyecto, el comando:

```
composer update
```

Eso nos descargará el propio wrapper, pero también todas sus dependencias que no tengamos todavía como la librería DomPDF.

Ahora nos queda asociar este service provider en nuestro archivo config/app.js. Nos dirigimos a la lista de todos los proveedores de servicio y en el array 'providers' agregamos:

```
Barryvdh\DomPDF\ServiceProvider::class,
```

Además opcionalmente esta envoltura nos proporciona un “facade”, ya sabes, una fachada para ahorrar un poco de código a la hora de usar esta clase por medio de funciones. Esta la tienes que configurar en el array ‘aliases’, también sobre el mismo app.js.

```
PDF' => Barryvdh\DomPDF\Facade::class,
```

## Usar DomPDF desde Laravel

A partir de ahora ya estamos en posición de usar DomPDF desde Laravel. Dependiendo de cómo esté tu aplicación trabajarás con la librería desde un controlador o desde otro lugar. Vamos a ir a lo simple, usando la librería directamente desde el sistema de rutas, pero asumimos que no será lo más habitual, como has aprendido en el [Manual de Laravel](#).

El HTML de origen para la generación del PDF también puede llegar desde varios sitios, un archivo, una vista, o una simple cadena de texto que tengas en una variable. En este caso vamos a traernos el HTML que nos genera una vista dentro de Laravel. Además, como estarás comprobando en el código siguiente, estamos usando DomPDF a través de su fachada.

```
Route::get('/test/', function () {  
    $pdf = PDF::loadView('pruebapapdf');  
    return $pdf->download('pruebapdf.pdf');  
});
```

En la primera línea generamos el PDF, gracias al método `PDF::loadView()`, indicando el nombre de la vista que hemos usado.

En la segunda línea tenemos una de las varias opciones que podemos implementar para la distribución de este PDF, por medio del método `download()`, indicando el nombre del archivo que se generará para descarga. El resultado es que nuestro navegador descargará el archivo y tendrá el nombre `'pruebapdf.pdf'`.

## HTML para generar un PDF

Como decía, lo que me pareció interesante de esta librería es la posibilidad de generar cabecera y pie de página para todas las páginas del documento. Eso se consigue simplemente con un HTML y un poco de CSS para posicionamiento de los elementos que van a servir como contenedores del pie y cabecera de la página.

Como verás en el ejemplo, se puede generar hasta la numeración de las páginas, algo muy útil cuando tienes informes largos que generar en PDF.

**Nota:** En este ejemplo que reproduzco a continuación creo la cabecera y pie de página con un HEADER y un FOOTER, etiquetas de sobra conocidas de HTML5. Pero podrías usar cualquier otro elemento HTML de tu preferencia. Lo importante aquí son los estilos definidos en el CSS. Por cierto, el ejemplo está directamente inspirado en la página de [demo de DomPDF](#).

```
<html>
<head>
<style>
  body{
    font-family: sans-serif;
  }
  @page {
    margin: 160px 50px;
  }
  header { position: fixed;
    left: 0px;
    top: -160px;
    right: 0px;
    height: 100px;
    background-color: #ddd;
    text-align: center;
  }
  header h1{
    margin: 10px 0;
  }
  header h2{
    margin: 0 0 10px 0;
  }
  footer {
    position: fixed;
    left: 0px;
    bottom: -50px;
```

```

    right: 0px;
    height: 40px;
    border-bottom: 2px solid #ddd;
  }
  footer .page:after {
    content: counter(page);
  }
  footer table {
    width: 100%;
  }
  footer p {
    text-align: right;
  }
  footer .izq {
    text-align: left;
  }
</style>
<body>
  <header>
    <h1>Cabecera de mi documento</h1>
    <h2>DesarrolloWeb.com</h2>
  </header>
  <footer>
    <table>
      <tr>
        <td>
          <p class="izq">
            Desarrolloweb.com
          </p>
        </td>
        <td>
          <p class="page">
            Página
          </p>
        </td>
      </tr>
    </table>
  </footer>
  <div id="content">
    <p>
      Lorem ipsum dolor sit...
    </p><p>
      Vestibulum pharetra fermentum fringilla...
    </p>
    <p style="page-break-before: always;">
      Podemos romper la página en cualquier momento...</p>
    </p><p>
      Praesent pharetra enim sit amet...
    </p>
  </div>
</body>
</html>

```

Eso es todo. Es así de simple. Puedes incluir imágenes y otros tipos de contenido, aplicando estilos CSS que

podrías incluso tener en archivos aparte. Ahora es simplemente documentarse, aunque la verdad es que no se encuentra mucha información (y eso es un punto flaco) a no ser la [página del repositorio de github de DomPDF](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en *04/04/2016*  
Disponible online en <http://desarrolloweb.com/articulos/generar-pdf-laravel-dompdf.html>