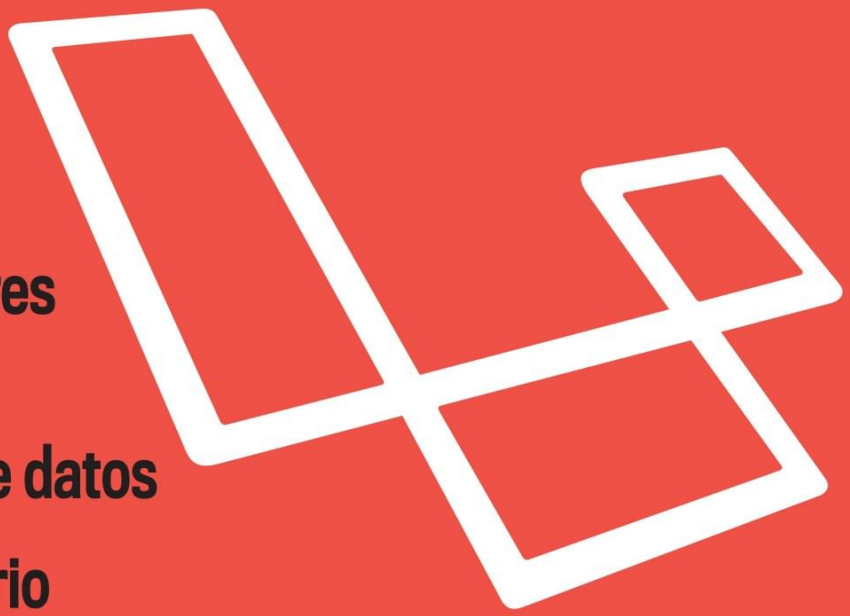
USERS

EL FRAMEWORK PHP MÁS UTILIZADO

Introducción a Laravel

APLICACIONES ROBUSTAS Y A GRAN ESCALA



- Características e instalación
- Vistas, controladores y rutas
- Modelos y bases de datos
- Interfaces de usuario
- Pruebas automatizadas

¡APRENDA DESDE CERO!

RU

por MARCELO CICERI

EL LIBRO DE UN VISTAZO

Este libro está enfocado en los programadores de PHP que nunca hayan trabajado con un framework y que deseen dar sus primeros pasos utilizando Laravel.
En cada uno de los capítulos se analizará un aspecto fundamental del framework, y se incluirá un segmento práctico en el cual se desarrollarán módulos que permitirán construir un blog de noticias.

01
CARACTERÍSTICAS E INSTALACIÓN
Es importante saber distinguir un framework de algo que no lo es. En este capítulo, además de acentuar estas definiciones, instalaremos el ambiente de desarrollo necesario para programar en Laravel.

02
PRIMEROS PASOS
Laravel es una herramienta muy compleja, por lo que antes de comenzar a escribir código, es necesario tener una noción acerca de sus principales características, la estructura de carpetas y archivos, la configuración y la arquitectura en general que posee el framework.

03
RUTAS
Las rutas son el punto de entrada de una aplicación Laravel. En este capítulo estudiaremos cómo construirlas y, a la vez, haremos un repaso del protocolo HTTP para entender bien su relación con el sistema de rutas del framework.

04
CONTROLADORES
En esta sección estudiaremos la manera en la que podemos ir segmentando la lógica de nuestra aplicación en diferentes controladores, que serán el espacio donde haremos converger gran parte de la lógica del blog.

05
VISTAS
En este segmento estudiaremos Blade, el sistema de plantillas que ofrece Laravel, y analizaremos sus principales características, como la herencia, la implementación de estructuras de código y las subvistas.

06
BASE DE DATOS
Este capítulo está dedicado a las herramientas que provee Laravel para poder crear, manipular, versionar y poblar una base de datos, haciendo un análisis de las principales características de la arquitectura implementada para operar con diversos motores.

07
MODELOS
En principio, realizaremos un breve repaso sobre la programación orientada a objetos, luego estudiaremos cómo interviene este paradigma en el patrón de arquitectura MVC, para luego poder definir nuestros propios modelos en Laravel.

08
TRABAJAR CON MODELOS
Con los modelos que hemos definido en el capítulo anterior, en este segmento realizaremos distintas operaciones que nos permitirán filtrar datos y ejecutar consultas hacia la base utilizando los modelos.

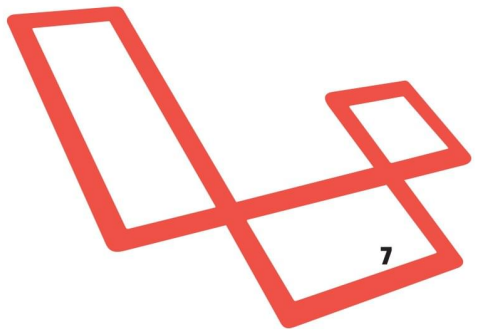
09
RELACIONES ENTRE MODELOS
Existen diferentes tipos de relaciones que podemos abordar en un sistema orientado a objetos, las cuales estudiaremos e implementaremos en este capítulo haciendo uso de los modelos de Laravel.

10
INTERFACES DE USUARIO
En esta sección profundizaremos más en las herramientas complementarias a las vistas que ofrece el framework para poder construir interfaces de usuario, tales como Laravel Mix, NPN, Webpack y Bootstrap.

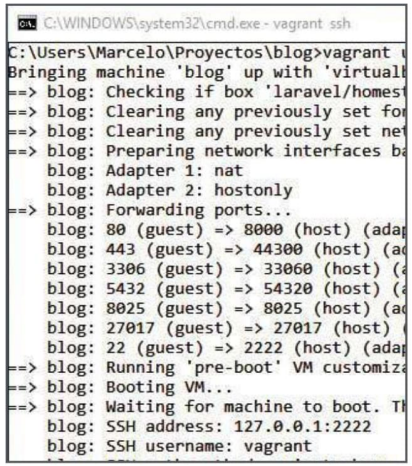
11
FORMULARIOS
En este capítulo estudiaremos la forma en la que podemos construir formularios con Collective y procesarlos haciendo uso de las validaciones que trae incorporadas el framework. También personalizaremos los mensajes y vincularemos formularios con modelos analizando sus ventajas.

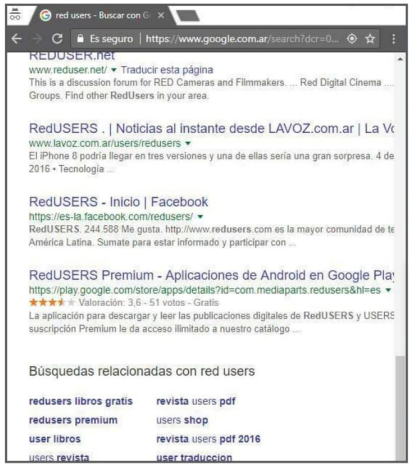
12
USUARIOS
Laravel ofrece herramientas estándar que permiten personalizar la experiencia del usuario. En este capítulo estudiaremos la internacionalización para poder implementar más de un idioma, pasando por los middlewares y, luego, la autenticación y el sistema de notificaciones que provee el framework.

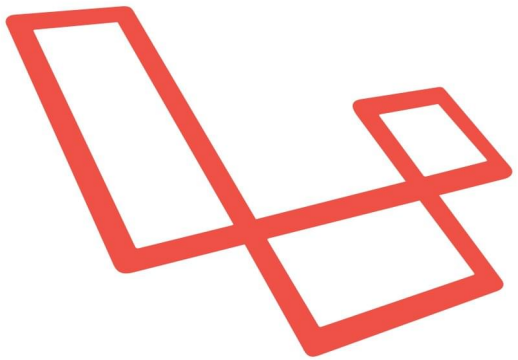
13
TESTING
Para poder comprobar nuestros desarrollos, Laravel ofrece dos herramientas: PHPUnit y Laravel Dusk. En este capítulo, además de estudiarlas, abordaremos los principales conceptos básicos que engloban a las actividades de comprobación de software.



CONTENIDO







INTRODUCCIÓN

La programación es una actividad que, por lo general, demanda realizar sus primeros pasos en soledad. Si bien existen contextos en los que se nos invita a trabajar en grupo cuando estamos aprendiendo, llega un momento en el que es necesario sentarse y empezar a diagramar algoritmos para resolver la parte del problema que nos toca.

Desarrollar software estando solos y luego empezar a trabajar en grupo requiere de un gran salto, tanto técnico como metodológico, y este libro tiene como objetivo darle herramientas al lector para ayudarlo a realizar mejor esa transición.

Es usual ver a diseñadores y maquetadores web devenidos en programadores PHP. Del mismo modo, podemos encontrar ese tipo de desarrolladores que establecen su carrera alrededor de sistemas empaquetados como CMS u otro tipo preparados en este lenguaje. Sin embargo, no es tan frecuente encontrar programadores que puedan adaptarse rápidamente a entornos de trabajo que contengan procesos de desarrollo para construir software de mediana o gran escala, y éste es un perfil muy requerido en el mercado.

Al principio de la obra, estableceremos los conceptos principales que componen un framework y veremos de qué manera se distinguen de otro tipo de sistemas. Luego estudiaremos Laravel, el cual, además de ser el proyecto PHP de mayor tendencia según Github, presenta amplias ventajas para el desarrollo de aplicaciones web de calidad.

En la actualidad, Laravel es uno de los frameworks PHP más completos, y realizar un libro que permita abordar todos sus temas merece más de un volumen. Por eso, esta obra apunta a que aquellos que nunca han trabajado en un framework puedan introducirse en uno haciendo uso de Laravel; no obstante, también hace referencia a varios lugares en los cuales el lector podrá profundizar los conocimientos sobre el framework.

Durante cada capítulo se analizará un aspecto del framework por medio de la construcción de módulos para un blog, por lo que al finalizar la obra, el lector podrá contar con un proyecto completo realizado con Laravel.

Es importante tener conocimientos básicos en PHP para poder abordar este texto, los cuales pueden adquirirse a través del libro PHP + MySQL, de esta misma editorial. También es fundamental contar con conocimientos básicos del paradigma de programación orientada a objetos.

01 Características e instalación

En este capítulo conoceremos el concepto de framework, analizaremos las principales características de Laravel e instalaremos todos los componentes necesarios para el ambiente de desarrollo. Por último, trabajaremos con Composer, el sistema de manejo de librerías más utilizado por los frameworks PHP.

¿QUÉ ES UN FRAMEWORK?

Podemos asociar el término framework a un subsistema y/o conjunto de librerías que proveen funcionalidades estándar a cualquier sistema; sin embargo, esta definición es incompleta, ya que además nos brinda:

Una **estructura** de carpetas y archivos para organizar el código.

Una **arquitectura** para desarrollar un proyecto.

Seguridad, ya que los frameworks son actualizados frecuentemente para poder implementar medidas contra nuevas amenazas.

Robustez, porque los frameworks son utilizados por muchos programadores en diversos proyectos; en consecuencia, cada framework está expuesto a un alcance mucho mayor al que podemos lograr escribiendo nuestro propio código.

Soporte, ya que al ser utilizado por otros programadores, es muy fácil encontrar a alguien que haya tenido el mismo problema que podamos tener nosotros y que no logremos resolver.

Un conjunto de **buenas prácticas** de programación para tener nuestro código lo más legible posible, de manera tal que podamos entender el código escrito por otro programador y, a la vez, hacer que el nuestro sea más entendible para los demás.

En síntesis, una definición más completa de framework es la siguiente: un conjunto de estructuras y componentes de software predefinidos e interconectados que sirven de base para la organización y el desarrollo de sistemas con propósitos generales.

¿Qué no es un framework?

Si realizamos una búsqueda en Internet con los términos **Framework PHP**, podemos encontrar una gran cantidad de sistemas que, muchas veces, no califican técnicamente como framework. En la **Tabla 1** conoceremos los distintos tipos de sistemas existentes, de modo de poder diferenciarlos con respecto a los frameworks.

CLASIFICACIÓN DE SISTEMAS BASADOS EN SU ORIENTACIÓN

Tipo de sistema	Orientación	Ejemplos
Framework	No están construidos en torno a una orientación específica, se los denomina de propósitos generales.	▶ Laravel ▶ Symfony ▶ Zend Framework ▶ Cake PHP
CMS	Del inglés Content Management System, es decir, Sistema de administración de contenido.	▶ Wordpress ▶ Drupal ▶ Joomla
e-commerce	Sistemas dedicados al comercio electrónico.	▶ Magento ▶ OpenCart
VLE	Del inglés Virtual Learning Environment, es decir, Entorno virtual de aprendizaje.	▶ Moodle ▶ Dokeos
Wiki	Permiten crear sitios web cuyas páginas pueden ser editadas directamente desde el navegador.	▶ DokuWiki ▶ MediaWiki ▶ PHPWiki

Tabla 1. Sistemas PHP que permiten crear sitios web.

Aplicaciones de un framework

Los **frameworks** se caracterizan por que pueden servir para crear cualquier tipo de sistema, ya que al estar orientados a propósitos generales, no tienen conceptos de una orientación en particular. Por lo tanto, podemos usar un framework para hacer **CMS**, **e-commerce**, **VLE** o **Wikis** y muchas cosas más. Sin embargo, esta propiedad no es reversible, ya que cambiar un sistema de una orientación a otra puede llegar a tener un costo demasiado alto en términos de tiempos de desarrollo, performance y robustez.

La elección correcta del sistema para construir un sitio web depende mucho del problema que queramos atacar.

Si el problema es muy acotado, con una proyección de cambios muy baja y en el cual un sistema con orientación particular satisfaga las necesidades generales, la mejor alternativa podría no ser un framework.

Sin embargo, cuando las necesidades del problema se vuelven muy particulares, con reglas de negocio muy complejas, entonces un framework es la mejor elección.

Cuando nos convencemos de que es conveniente utilizar un framework para llevar adelante un proyecto, surge la siguiente pregunta: ¿qué framework nos conviene elegir?

PHP ofrece muchos frameworks, algunos de los cuales se encuentran desde hace mucho tiempo en el mercado, mientras que otros son más recientes. En la **Tabla 2** podemos ver algunos de los más reconocidos.

▶ FRAMEWORK PHP

Nombre	Primera versión	Descripción	URL
Laravel	Junio 2011	Es el framework PHP más utilizado del momento. Su filosofía es crear código simple y elegante.	https://laravel.com
Symfony	Octubre 2005	Este framework puede ser utilizado para crear proyectos web y como un conjunto de componentes reutilizables; de hecho, Laravel reutiliza muchos componentes de Symfony.	http://symfony.com
Zend	Marzo 2006	Tiene una implementación orientada 100% a la programación orientada a objetos.	https://framework.zend.com
CodeIgniter	Febrero 2006	Está compuesto por un kit de herramientas simples y elegantes para crear aplicaciones web.	https://codeigniter.com
Phalcon	Noviembre 2012	Es conocido por funcionar como una extensión en Zephir/C, de manera tal que el código generado es compilado.	https://phalconphp.com/es/

Tabla 2. Los frameworks PHP que permiten crear distintas aplicaciones y sitios web.

LARAVEL

Como ya sabemos, existen muchos frameworks PHP disponibles en el mercado, y Laravel es relativamente nuevo comparado con los principales que fueron presentados en la **Tabla 2**. Veamos un poco acerca de su historia.

Historia de Laravel

En el año 2011, uno de los frameworks PHP más populares era CodeIgniter. Sin embargo, muchas funcionalidades fundamentales para la creación de aplicaciones web, como la autenticación, no estaban incorporadas en él, motivo por el cual **Taylor Otwell**, un programador web, decidió crear un framework que las incluyera.

En un principio, Laravel no fue creado con el patrón de arquitectura MVC, y su foco estaba puesto principalmente en resolver problemas de autenticación. No obstante, la primera versión incorporaba funcionalidades que fueron bien recibidas, y de forma rápida, por la comunidad de desarrolladores.

La segunda versión tardó menos de seis meses en salir al mercado. Laravel terminó de adoptar el patrón MVC para su arquitectura e incorporó el siguiente slogan:

> Liberándote del **código espagueti**, Laravel te ayuda a crear aplicaciones maravillosas usando una sintaxis simple y expresiva. El desarrollo debe ser una experiencia creativa que disfrutes, no algo que sea doloroso. Disfruta del aire fresco.

Sistemas basados en Laravel

Es muy importante dedicar tiempo a estudiar bien el problema que debemos resolver y analizar las herramientas disponibles en el mercado antes de elegir una para crear nuestra aplicación. Existen también sistemas basados en Laravel que están orientados a propósitos particulares, por ejemplo, **Statamic, https://statamic.com**, y **OctoberCMS, https://octobercms.com**, que son CMS basados en Laravel.

Es de destacar que, a partir de junio de 2015, Laravel comenzó a distribuir versiones **LTS** (*Long Term Support*).

¿Por qué es importante esto? Las versiones LTS están diseñadas para ser soportadas durante un período más largo de lo normal, proveen corrección de errores durante dos años, y la aplicación de medidas de seguridad durante tres años. Considerando que hoy en día la tecnología avanza muy rápidamente, es importante contar con herramientas que garanticen un período de continuidad, sobre todo, teniendo en cuenta la creación de proyectos que se pueden desarrollar a mediano o largo plazo.

Laravel en la actualidad

El slogan actual de Laravel es el siguiente:

> ¿Amas el código hermoso? Nosotros también.
> Laravel, el framework para los artesanos de la Web.

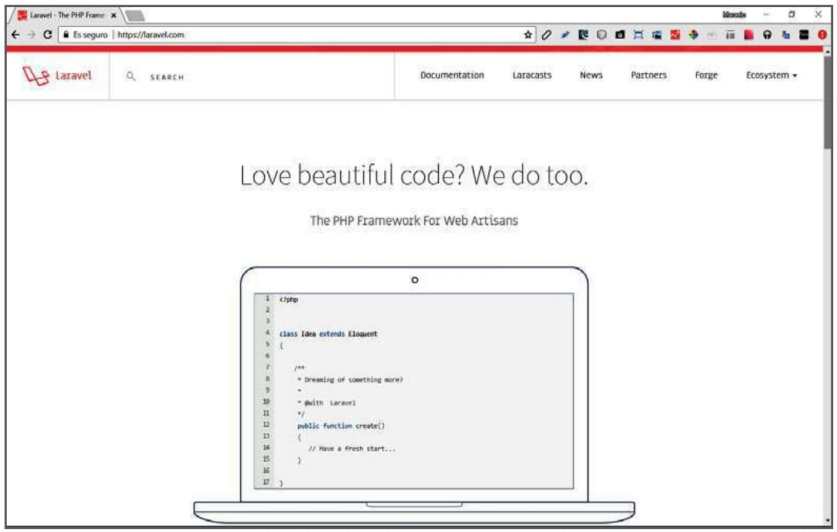


Figura 1. Laravel todavía mantiene la idea de construir proyectos con código simple y expresivo.

Algunas de las características que podemos destacar de Laravel son las siguientes:

- Posee una buena curva de aprendizaje.
- Tiene una **documentación** muy completa, simple y gratuita.
- Cuenta con una gran comunidad de desarrolladores, **https://laravel.io**.
- Es el proyecto más **popular** en github desarrollado con PHP.
- Utiliza muchos componentes de Symfony, el cual es el segundo proyecto PHP más popular en github.
- Brinda un conjunto de servicios y herramientas de infraestructura que facilitan su puesta en funcionamiento en diferentes entornos, tales como Forge y Homestead.
- Ofrece versiones **LTS** (*Long Term Support*). Su última versión LTS, la 5.5, fue lanzada en agosto de 2018 y tendrá actualizaciones de seguridad hasta 2020, lo cual la hace ideal para llevar a cabo proyectos de amplia longevidad.
- Sigue el patrón de arquitectura **Modelo-Vista-Controlador** (MVC).
- Provee un poderoso **ORM**, Eloquent, que está basado en el patrón active record.
- Utiliza un sistema de **plantillas** con un sistema de caché que permite mejorar la performance de los sitios desarrollados con Laravel.

Frameworks propios

En una primera impresión, aprender a utilizar un framework nuevo de un desconocido puede parecer más difícil que crear uno con nuestras manos. Sin embargo, hay que considerar que, al utilizar un framework abierto, contamos con soporte de una comunidad y, en muchos casos, con una buena documentación, lo cual permite que sea más sencillo incorporar nuevos programadores a nuestro proyecto.

El ecosistema Laravel

Debido a su popularidad, se crearon muchas herramientas para facilitar el desarrollo de sistemas con Laravel, entre las cuales podemos destacar las siguientes:

Homestead	Es la máquina virtual oficial utilizada como ambiente de desarrollo para proyectos Laravel. Brinda todo lo necesario para ejecutar el framework junto con herramientas útiles para el desarrollo.
Lumen	Es una versión reducida de Laravel pensada para el desarrollo de servicios web. Al tener ese foco, hace que sea mucho más rápida y liviana que la versión tradicional de Laravel.
Laravel Cashier	Es una interfaz que puede integrarse a servicios de administración de suscripciones, como **Stripe** y **Braintree**.
Statamic	Es un CMS construido con Laravel.

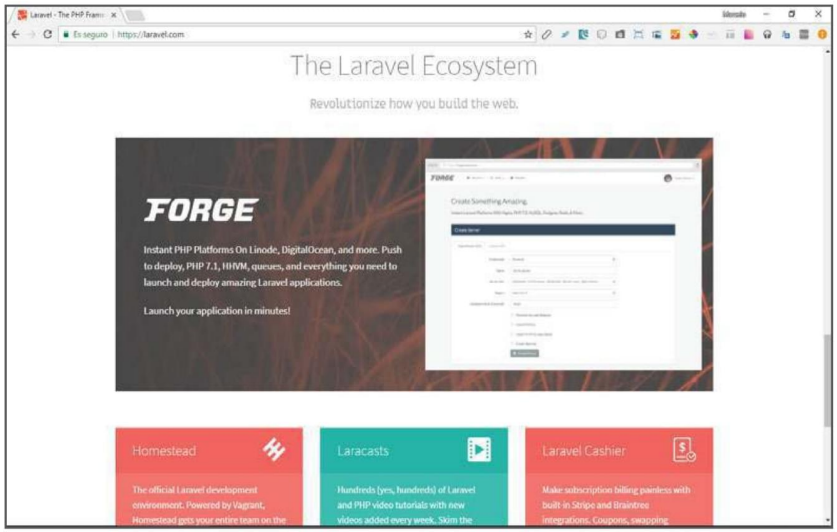


Figura 2. El ecosistema de Laravel permite optimizar tiempos y procesos de desarrollo.

La comunidad detrás de Laravel

Al haber sido adoptado por un gran número de desarrolladores, existen muchos portales con información, documentación, librerías, videos y tutoriales en los cuales podemos buscar apoyo a la hora de trabajar con Laravel. La mayoría de los portales oficiales se encuentran en inglés, pero también podemos encontrar mucha información extraoficial en español. Entre los portales, destacamos los siguientes:

- **https://laravel.io**: portal oficial de la comunidad de desarrolladores Laravel, el cual integra los demás portales de la comunidad.
- **https://laravel-news.com**: sitio web dedicado a noticias sobre Laravel.
- **https://laracasts.com**: portal con videotutoriales.
- **http://laravel-tricks.com**: este portal contiene pequeñas porciones de código fuente para realizar trucos sencillos y útiles con Laravel.
- **https://larachat.co**: a través de Slack, un sistema de chat con diversas funcionalidades, se administran diferentes espacios para conversar sobre el framework.
- **https://www.meetup.com/es/topics/laravel**: permite encontrar grupos de personas que trabajen con Laravel a partir de una ubicación geográfica.
- **https://larajobs.com**: último en la lista, pero no por eso menos importante, LaraJobs es el sistema de clasificados online dedicado por completo a la búsqueda de desarrolladores Laravel.

Código espagueti

El código espagueti hace referencia a los sistemas que tienen una estructura compleja e incomprensible, una práctica muy mal vista en el mercado. Es común de ver en proyectos PHP, ya que el lenguaje, en sus orígenes, carecía de características que permitiesen crear estructuras simples, sólidas y elegantes. Gracias a la evolución de PHP, podemos contar con un framework como Laravel, con código más legible y expresivo.

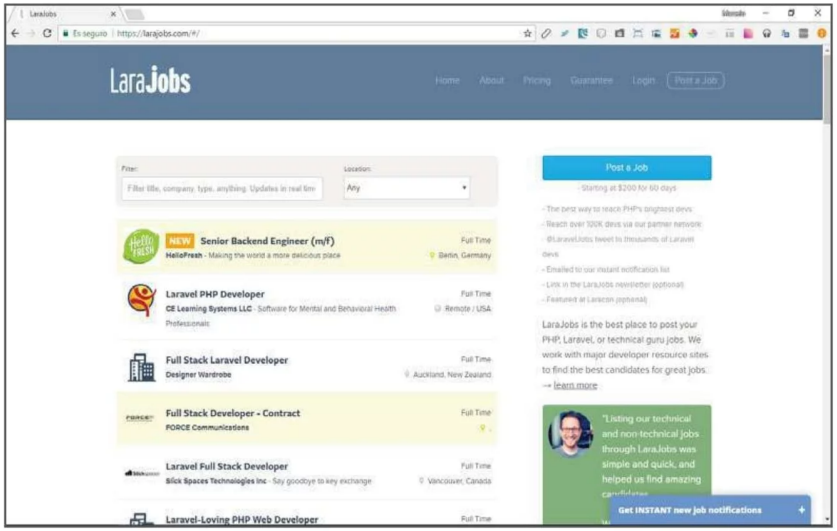


Figura 3. **LaraJobs** publica avisos desde todo el mundo y permite filtrar los puestos buscados para trabajar de forma remota.

AMBIENTE DE DESARROLLO

Existen dos formas de instalar el ambiente de desarrollo para hacer aplicaciones con Laravel. Una de ellas es instalando todos los componentes necesarios en nuestra máquina, y la otra es utilizando una máquina virtual.

Mantener el código simple

Como ya mencionamos, Laravel mantiene como objetivo generar código simple y elegante. Taylor Otwell, el creador de Laravel, publicó recientemente un artículo en **https://medium.com/@taylorotwell/measuring-code-complexity-64356da605f9**, donde brinda datos que nos permiten comparar la complejidad del código fuente de diferentes frameworks.

Instalar el ambiente en nuestra máquina

Laravel nos permite crear tanto aplicaciones que puedan ser ejecutadas por una consola como también aplicaciones web, siendo este último el caso más utilizado.

Para el desarrollo de aplicaciones web con Laravel es necesario contar con lo siguiente:

Un **sistema operativo** que será el contenedor principal de todas las herramientas tecnológicas. Laravel funciona con varios sistemas operativos, aunque la documentación oficial y el soporte de la comunidad están orientados a sistemas basados en GNU/Linux.

Un **servidor web** que funcionará para exponer nuestro sitio web, tal como Apache o Nginx.

El motor de **PHP**. La versión 5.5 de Laravel establece la versión necesaria en https://github.com/laravel/laravel/blob/master/composer.json#L8.

Un motor de **base de datos** en la cual almacenemos la información de nuestra aplicación, como MySQL o PostgreSQL.

A su vez, debemos contar con las siguientes librerías habilitadas:

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

Es importante destacar que de todos los elementos mencionados anteriormente el único que puede faltar es la base de datos, aunque es algo poco común en el contexto de aplicaciones web. Por otra parte, en el contexto de las aplicaciones de escritorio, el servidor web no es necesario.

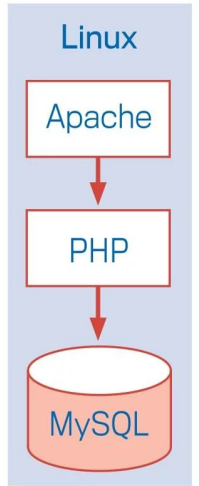


Figura 4. Laravel funciona en el stack LAMP (Linux, Apache, MySQL, PHP).

Utilizar una máquina virtual

Las máquinas virtuales son programas que nos permiten crear computadoras virtuales utilizando parte de los componentes de hardware de nuestra máquina física.

Homestead es una máquina virtual provista de manera oficial por los desarrolladores de Laravel, que incluye todo el ambiente de desarrollo junto con otros elementos de utilidad. Homestead está creada utilizando los siguientes componentes:

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

A lo largo del libro estudiaremos y trabajaremos con estas herramientas, pero para utilizar Homestead primero debemos instalar algunos programas en el sistema.

VirtualBox

Homestead es distribuida a través de diferentes sistemas de virtualización, entre ellos, VirtualBox. Con VirtualBox podremos no sólo instalar Homestead, sino también crear diferentes máquinas virtuales.

Stack de tecnologías

El diagrama de la Figura 4 tiene una forma parecida a una pila, por eso a los conjuntos de tecnologías se los suele denominar **stack**, que significa **pila** en inglés. En algunos libros también aparece como **bundle**, que significa **manojo**. Un stack muy conocido es **LAMP** (Linux, Apache, MySQL o MariaDB, PHP), que varía a **WAMP** al cambiar el sistema operativo Linux por Windows. Otro que está emergiendo es el stack **MEAN** (Mongo, Express, Angular, NodeJs).

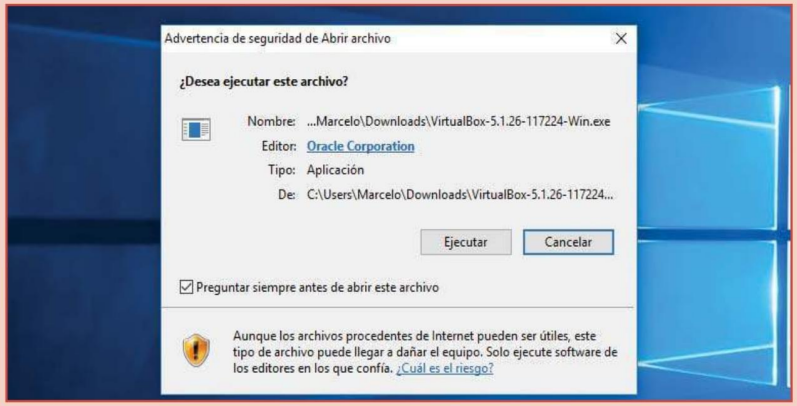
INSTALAR VIRTUALBOX EN WINDOWS

01 Ingrese en la dirección **www.virtualbox.org/wiki/Downloads**. Descargue el archivo que corresponda al sistema operativo instalado en su máquina física. En este caso es Windows 10, por lo que se utilizará el archivo disponible en el enlace Windows hosts para la arquitectura de 64 bits.

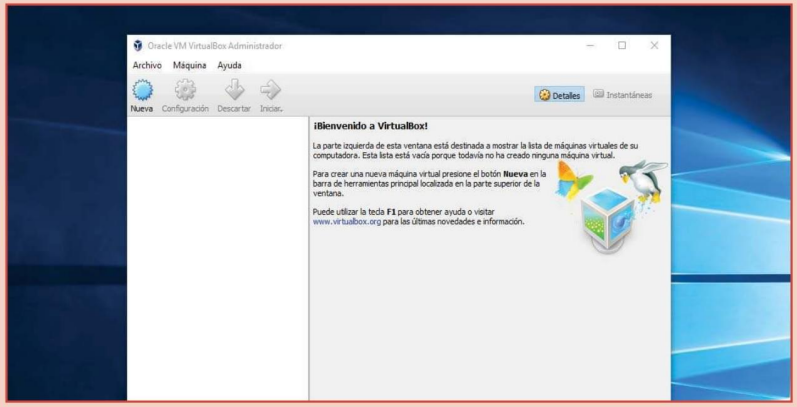
Agilidad

Es muy recomendable utilizar una máquina virtual para el desarrollo de software, ya que si varios programadores trabajan en el mismo proyecto, se evita que cada uno tenga que instalar todos los componentes de software, dado que las máquinas virtuales son sumamente portables. A su vez, al utilizar todos el mismo ambiente de trabajo, se reduce la posibilidad de que existan problemas en el entorno de un programador en particular.

02 Una vez que la descarga haya finalizado, haga doble clic sobre el archivo adecuado y espere mientras se inicia el asistente de instalación; será necesario contar con privilegios de Administrador. Haga clic en **Siguiente** en cada ventana, manteniendo las opciones predeterminadas.



03 Al finalizar, podrá ejecutar Virtualbox para administrar sus máquinas virtuales. Hay dos formas de crear máquinas con VirtualBox: una es utilizando esta interfaz, y otra más sencilla es a través de Vagrant.



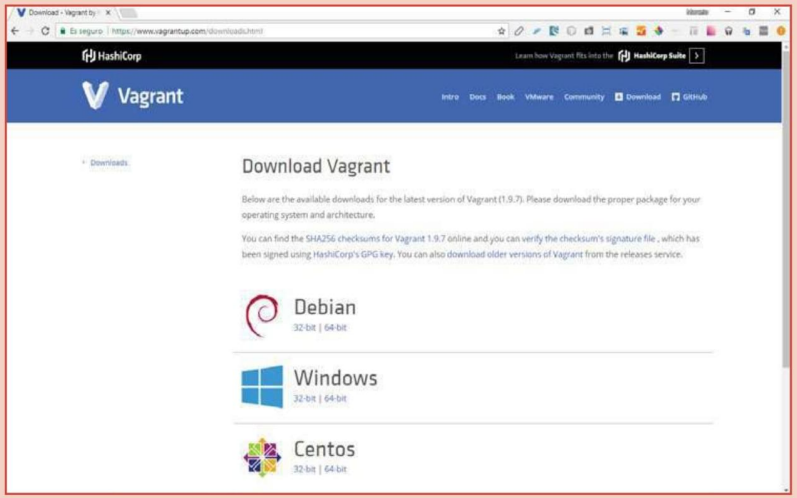
Vagrant

Vagrant es una herramienta que facilita la creación y configuración de máquinas virtuales estableciendo los parámetros de éstas de forma declarativa y en un único archivo. Este archivo puede formar parte del código fuente de nuestro proyecto; de esta manera, si trabajamos en grupo con varios programadores, lograremos que todos utilicemos la misma máquina con las mismas configuraciones.

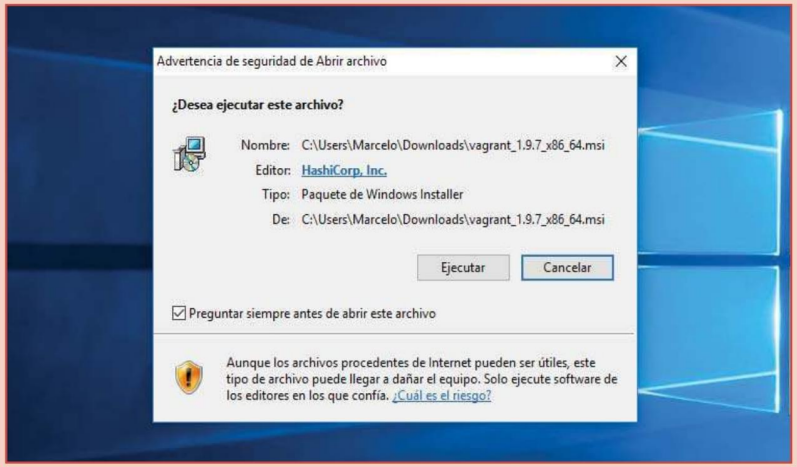
Virtualbox y **Vagrant** son una gran combinación, pero también es posible utilizar otras herramientas, como **Docker (www.docker.com)**, para generar el ambiente de desarrollo necesario y distribuirlo como parte del proyecto.

INSTALAR VAGRANT EN WINDOWS

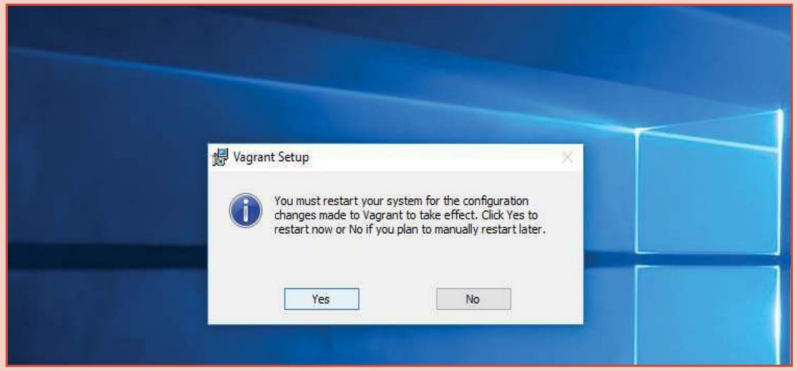
01 Ingrese en la dirección **www.vagrantup.com/downloads.html**. Debe descargar el archivo que corresponda al sistema operativo instalado en su máquina física.



02 Una vez que la descarga haya finalizado, haga doble clic sobre el archivo adecuado y espere mientras se inicia el asistente de instalación; será necesario contar con privilegios de Administrador. Haga clic en **Siguiente** en cada ventana, manteniendo las opciones establecidas.



03 Al finalizar el asistente, reinicie el equipo.



04 Para comprobar que Vagrant se haya instalado correctamente, abra una interfaz de línea de comando y ejecute **vagrant**.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Versión 10.0.15063]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Marcelo>vagrant
Usage: vagrant [options] <command> [<args>]

    -v, --version                    Print the version and exit.
    -h, --help                       Print this help.

Common commands:
     box             manages boxes: installation, removal, etc.
     connect         connect to a remotely shared Vagrant environment
     destroy         stops and deletes all traces of the vagrant machine
     global-status   outputs status Vagrant environments for this user
     halt            stops the vagrant machine
     help            shows the help for a subcommand
     init            initializes a new Vagrant environment by creating a Vagrantfile
     login           log in to HashiCorp's Vagrant Cloud
     package         packages a running vagrant environment into a box
     plugin          manages plugins: install, uninstall, update, etc.
     port            displays information about guest port mappings
     powershell      connects to machine via powershell remoting
     provision       provisions the vagrant machine
     push            deploys code in this environment to a configured destination
     rdp             connects to machine via RDP
     reload          restarts vagrant machine, loads new Vagrantfile configuration
     resume          resume a suspended vagrant machine
     share           share your Vagrant environment with anyone in the world
     snapshot        manages snapshots: saving, restoring, etc.
     ssh             connects to machine via SSH
     ssh-config      outputs OpenSSH valid configuration to connect to the machine
     status          outputs status of the vagrant machine
     suspend         suspends the machine
     up              starts and provisions the vagrant environment
     validate        validates the Vagrantfile
     version         prints current and latest Vagrant version

For help on any individual command run `vagrant COMMAND -h`

Additional subcommands are available, but are either more advanced
or not commonly used. To see all subcommands, run the command
`vagrant list-commands`.

C:\Users\Marcelo>_
```

Configuración como parte del código

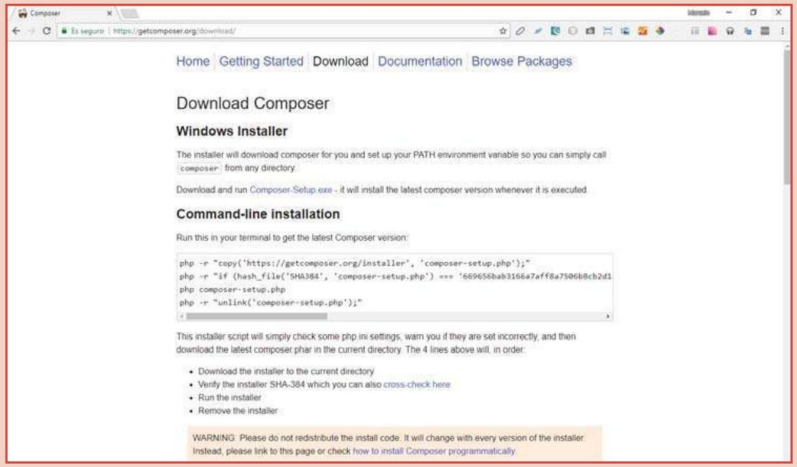
Es muy útil, y también una práctica bastante común en la actualidad, incluir, en el repositorio de código de nuestro proyecto, los archivos necesarios para generar el ambiente de desarrollo. Esto permite mantener sincronizado entre todos los desarrolladores cualquier cambio requerido para ejecutar el ambiente y, por otra parte, también sirve como punto de partida para la documentación y generación de la infraestructura del proyecto.

Composer

Laravel está compuesto por muchos paquetes de código provenientes de diferentes fuentes y repositorios. Para no buscar y descargar los fuentes desde todos estos lugares, contamos con una herramienta que lo hace por nosotros, denominada **Composer**; es necesario tener instalado PHP previamente.

INSTALAR COMPOSER EN WINDOWS

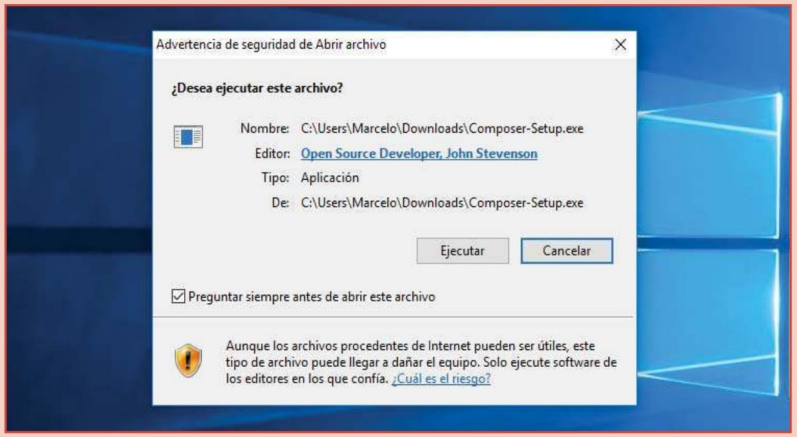
01 Ingrese en la dirección **https://getcomposer.org/download** y descargue el archivo **Composer-Setup.exe**.



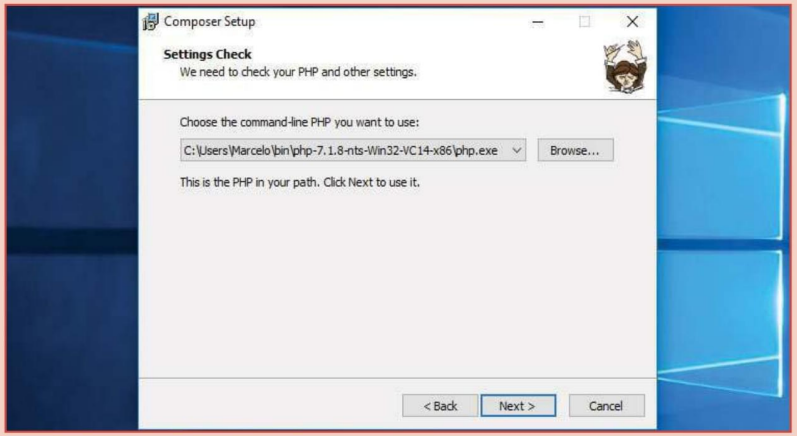
Herramientas para el desarrollo

Existen herramientas que nos facilitan tareas que a veces tienen que ver con el código generado, con el soporte necesario para que nuestro software se ejecute o con la administración de los requerimientos. Su elección es importante, sobre todo cuando trabajamos en equipo, pues permiten ahorrar tiempo y optimizar el trabajo.

02 Una vez que la descarga haya finalizado, haga doble clic sobre el archivo adecuado y espere mientras se inicia el asistente de instalación. Será necesario contar con privilegios de Administrador.



03 Durante la instalación, indique la ruta donde tenga descargado PHP, que deberá estar instalado en el sistema operativo junto con la librería php_openssl.



04 Para comprobar que Composer se haya instalado correctamente, abra una nueva interfaz de línea de comando y ejecute `composer`.

```
C:\Users\Marcelo\Proyectos>composer create-project --ignore-platform-reqs --prefer-dist laravel/laravel blog
Installing laravel/laravel (v5.4.30)
  - Installing laravel/laravel (v5.4.30): Downloading (100%)
Created project in blog
> php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 59 installs, 0 updates, 0 removals
  - Installing symfony/css-selector (v3.3.6): Downloading (100%)
  - Installing tijsverkoyen/css-to-inline-styles (2.2.0): Downloading (100%)
  - Installing doctrine/inflector (v1.1.0): Downloading (100%)
  - Installing symfony/polyfill-mbstring (v1.4.0): Downloading (100%)
  - Installing symfony/var-dumper (v3.3.6): Downloading (100%)
  - Installing jakub-onderka/php-console-color (0.1): Downloading (100%)
  - Installing jakub-onderka/php-console-highlighter (v0.3.2): Downloading (100%)
  - Installing dnoegel/php-xdg-base-dir (0.1): Downloading (100%)
  - Installing nikic/php-parser (v3.1.0): Downloading (100%)
  - Installing psr/log (1.0.2): Downloading (100%)
  - Installing symfony/debug (v3.3.6): Downloading (100%)
  - Installing symfony/console (v3.3.6): Downloading (100%)
  - Installing psy/psysh (v0.8.11): Downloading (100%)
  - Installing vlucas/phpdotenv (v2.4.0): Downloading (100%)
  - Installing symfony/routing (v3.3.6): Downloading (100%)
  - Installing symfony/process (v3.3.6): Downloading (100%)
  - Installing symfony/http-foundation (v3.3.6): Downloading (100%)
  - Installing symfony/event-dispatcher (v3.3.6): Downloading (100%)
  - Installing symfony/http-kernel (v3.3.6): Downloading (100%)
```

El comando **composer** brinda un listado completo de todos los comandos que ofrece la herramienta. En la **Tabla 3** podemos observar una breve descripción de sus principales funciones.

▶ PRINCIPALES COMANDOS DE COMPOSER

Comando	Descripción
composer install	Busca un archivo denominado **composer.json** donde se declaran todas las **dependencias** (paquetes de código fuente) necesarias para el proyecto, las descarga y las almacena en una carpeta llamada **vendor**.
composer require	Recibe como parámetro el nombre de un paquete al cual busca, descarga en la carpeta **vendor** y actualiza el archivo **composer.json** para agregarlo como una nueva dependencia del proyecto.
composer update	Lee todas las dependencias declaradas en **composer.json**, comprueba si hay nuevas versiones de ellas y las instala.
composer create-project	Permite crear nuevos proyectos PHP. Tiene varias opciones, entre ellas, una que nos permite especificar si queremos utilizar algún framework en particular.

Tabla 3. Composer nos permite ahorrar mucho tiempo automatizando tareas.

Crear un proyecto Laravel desde Composer

En esta ocasión crearemos un proyecto denominado **blog**, con la última versión disponible de Laravel, para lo cual descargaremos todas las librerías necesarias. Para lograrlo, abrimos una interfaz de comandos, nos ubicamos en la carpeta donde crearemos el proyecto y escribimos el siguiente comando: **composer create-project --ignore-platformreqs --prefer-dist laravel/laravel blog 5.5.***

```
C:\Users\Marcelo\Proyectos>composer create-project --ignore-platform-reqs --prefer-dist laravel/laravel blog
Installing laravel/laravel (v5.4.30)
  - Installing laravel/laravel (v5.4.30): Downloading (100%)
Created project in blog
> php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 59 installs, 0 updates, 0 removals
  - Installing symfony/css-selector (v3.3.6): Downloading (100%)
  - Installing tijsverkoyen/css-to-inline-styles (2.2.0): Downloading (100%)
  - Installing doctrine/inflector (v1.1.0): Downloading (100%)
  - Installing symfony/polyfill-mbstring (v1.4.0): Downloading (100%)
  - Installing symfony/var-dumper (v3.3.6): Downloading (100%)
  - Installing jakub-onderka/php-console-color (0.1): Downloading (100%)
  - Installing jakub-onderka/php-console-highlighter (v0.3.2): Downloading (100%)
  - Installing dnoegel/php-xdg-base-dir (0.1): Downloading (100%)
  - Installing nikic/php-parser (v3.1.0): Downloading (100%)
  - Installing psr/log (1.0.2): Downloading (100%)
  - Installing symfony/debug (v3.3.6): Downloading (100%)
  - Installing symfony/console (v3.3.6): Downloading (100%)
  - Installing psy/psysh (v0.8.11): Downloading (100%)
  - Installing vlucas/phpdotenv (v2.4.0): Downloading (100%)
  - Installing symfony/routing (v3.3.6): Downloading (100%)
  - Installing symfony/process (v3.3.6): Downloading (100%)
  - Installing symfony/http-foundation (v3.3.6): Downloading (100%)
  - Installing symfony/event-dispatcher (v3.3.6): Downloading (100%)
  - Installing symfony/http-kernel (v3.3.6): Downloading (100%)
  - Installing symfony/finder (v3.3.6): Downloading (100%)
  - Installing swiftmailer/swiftmailer (v5.4.8): Downloading (100%)
  - Installing paragonie/random_compat (v2.0.10): Downloading (100%)
  - Installing ramsey/uuid (3.7.0): Downloading (100%)
  - Installing symfony/translation (v3.3.6): Downloading (100%)
  - Installing nesbot/carbon (1.22.1): Downloading (100%)
```

Figura 5. La creación de un proyecto puede demorar varios minutos dependiendo de nuestra conexión a Internet.

Composer fuera de Laravel

Composer es una librería para administrar dependencias en proyectos PHP; no necesariamente tienen que ser proyectos realizados con Laravel. También nos brinda la posibilidad de crear nuestras propias dependencias y publicarlas a través del sitio web **https://packagist.org**. Es muy recomendable leer la guía **https://getcomposer.org/doc/01-basic-usage.md** para entender cómo funciona Composer y de qué manera podemos incluirlo en nuestros proyectos.

Si analizamos el comando ejecutado, vemos que **--ignore-platform-reqs** indica a Composer que no realice ninguna validación en nuestra máquina local para ver si ésta es compatible con Laravel. Esta validación no será necesaria si utilizamos Homestead. En el caso contrario, Composer analiza la máquina y nos indica si cumple con todos los requisitos necesarios. De no ser así, deberemos adaptarla hasta cumplir los criterios.

La opción **--prefer-dist laravel/laravel** indica a Composer que el proyecto **blog** que crearemos será desarrollado con Laravel, luego de introducir el nombre del proyecto **blog** indicamos que usaremos la última versión LTS disponible. Si el comando se ejecutó de manera exitosa, al ingresar en la carpeta **blog** veremos que se ha creado la estructura de archivos y carpetas correspondiente.

Figura 6. Estructura de carpetas para organizar el código fuente.

Instalar Homestead mediante Composer

Como sabemos, Vagrant nos permite declarar la configuración de una máquina virtual mediante un archivo. Para el caso de Homestead, dicho archivo se encuentra **empaquetado** en la nube, y la forma más sencilla de acceder a él es mediante Composer.

Ingresamos en una interfaz de línea de comando a la carpeta donde creamos el proyecto **blog**, y ejecutamos el comando **composer require laravel/homestead --dev –ignore-platform-reqs**.

Figura 7. **Descarga de los archivos Vagrant de Homestead mediante Composer.**

Este paquete nos provee de un script que analizará nuestra máquina física y establecerá los parámetros de configuración en un archivo denominado **Homestead.yaml**. Para ejecutar dicho script en Windows, debemos utilizar el comando **vendor\\bin\\homestead make**.

Descarga de librerías de desarrollo

Mediante el parámetro **--dev** indicamos a **Composer** que la librería en cuestión será utilizada para construir nuestro proyecto. De esta forma, podemos diferenciar las librerías necesarias para ejecutarlo, de aquellas que son requeridas para construirlo. Ver más información en **https://getcomposer.org/doc/04-schema.md#require-dev**.

Para finalizar, ejecutamos el comando **vagrant up**, que descargará e iniciará la máquina virtual Homestead. La primera vez el comando podría solicitar cambios en su configuración de sistema operativo y/o firewall, ya que genera una interfaz de red para conectar la máquina virtual con la física.

En caso de tener problemas con las private key, debemos comentar la línea que contiene **keys: y - ~/.ssh/id_rsa** utilizando el carácter #, y la anterior a la misma, de manera que quede de la siguiente forma:

```
  ip: 192.168.10.10
memory: 2048
cpus: 1
provider: virtualbox
authorize: ~/.ssh/id_rsa.pub
#keys:
#    - ~/.ssh/id_rsa
folders:
    -
        map: 'C:\Users\desar\proyectos\blog'
        to: /home/vagrant/blog
sites:
    -
        map: blog.app
        to: /home/vagrant/blog/public
databases:
    - homestead
name: blog
hostname: blog
```

Es normal que la primera ejecución de **vagrant up** demore varios minutos, ya que realiza la descarga de la máquina virtual.

Será necesario ejecutar este comando cada vez que deseemos iniciar la máquina virtual, ya sea por haberla apagado previamente con el comando **vagrant halt** o por haber reiniciado la máquina física. Nuestro sitio web responderá en la IP establecida en el archivo **Homestead.yaml**.

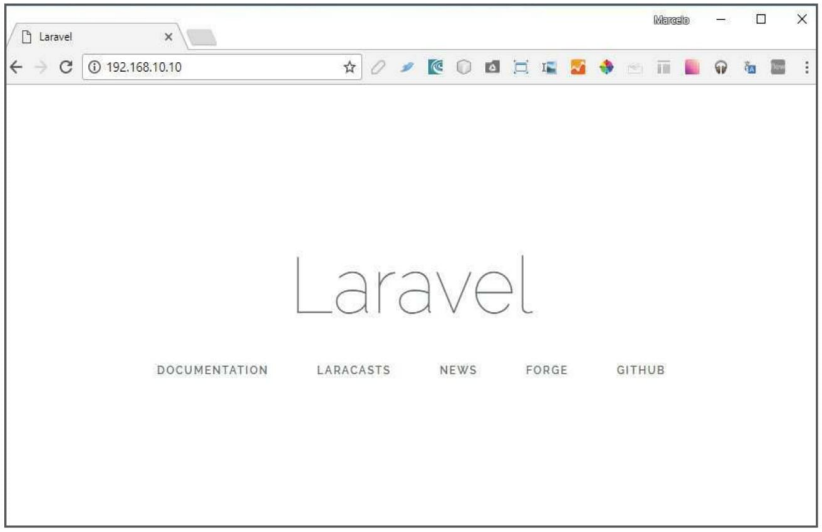


Figura 8. Pantalla de inicio de Laravel en una máquina virtual Homestead.

La instalación de Homestead puede parecer algo compleja, pero antes de llegar a esta conclusión, repasemos todos los componentes que trae consigo Homestead y consideremos el tiempo que demandaría instalar cada uno por separado junto con un nuevo sistema operativo.

Tengamos en cuenta también que, al trabajar con máquinas virtuales, podemos disponer de diferentes ambientes de trabajo sin tener que alterar nuestra máquina física para cada ocasión.

Resumen Capítulo 01

En este capítulo conocimos qué es un framework, aprendimos a diferenciarlo de otros sistemas y vimos las aplicaciones que puede tener. Luego repasamos la historia de Laravel y las ventajas que trae consigo, y también analizamos sus características principales. Más adelante, instalamos los componentes necesarios para crear nuestro ambiente de desarrollo y generamos un proyecto blog mediante Composer. Para terminar, vimos cómo trabajar con Homestead, la máquina virtual de Laravel, e instalamos todo lo necesario para su ejecución.

ACTIVIDADES

Test de Autoevaluación

1. ¿Qué es un framework?
2. ¿Cuál es la diferencia entre un framework orientado a propósitos generales y uno orientado a propósitos particulares? ¿Cuál es la principal ventaja de un framework con propósitos generales?
3. ¿Cuáles son los requerimientos necesarios para instalar Laravel?
4. ¿Qué es una máquina virtual?
5. ¿Qué componentes brinda Homestead?
6. ¿Qué es VirtualBox? ¿Lo necesitamos para desarrollar con Laravel?
7. ¿Qué es Vagrant? ¿Lo necesitamos para desarrollar con Laravel?
8. ¿Qué es una dependencia? ¿Cómo administra Laravel sus dependencias?
9. ¿Podemos utilizar Composer por fuera de Laravel?
10. ¿Qué contiene el archivo **Homestead.yaml**?

Ejercicios prácticos

1. Inicie VirtualBox y abra un administrador de tareas. Luego abra una interfaz de línea de comando e inicie Homestead mediante el comando **vagrant up**. Observe los cambios en VirtualBox y en el Administrador de tareas. Apague Homestead mediante el comando **vagrant halt** y siga observando los cambios.

2. Abra una interfaz de línea de comando y posiciónese en una carpeta nueva vacía, fuera del proyecto **blog**. Ejecute el comando **composer init**. Observe los parámetros requeridos y el archivo **composer.json** generado.

3. En la misma carpeta del paso anterior, ejecute **composer require fzaninotto/faker**. ¿Qué cambios se produjeron en el archivo **composer.json**? ¿Qué carpetas y/o archivos nuevos se generaron?

02 Primeros pasos

En este capítulo vamos a iniciar nuestro recorrido por el código fuente del framework. Analizaremos la estructura de archivos y carpetas, estudiaremos la inicialización del sistema y veremos las diferentes estrategias de configuración. También instalaremos componentes adicionales y abriremos un espacio para hablar de la arquitectura del framework.

TRABAJAR CON HOMESTEAD

En el **Capítulo 1** vimos la manera de instalar Homestead, la máquina virtual oficial de Laravel, que funciona como entorno de desarrollo. En esta sección vamos a analizar los aspectos principales que se deben tener en cuenta para trabajar con una máquina virtual.

Iniciar sesión

Siempre que iniciemos nuestra jornada, luego de encender la computadora física, debemos iniciar la máquina virtual, lo cual se realiza mediante el comando **vagrant up** desde una terminal. Una vez iniciada la máquina virtual, debemos ejecutar el comando **vagrant ssh** para poder iniciar una terminal en ella. Al ejecutar este comando, veremos que la sesión en la terminal pasó a ser **vagrant@blog:~$**.

Recordemos que Homestead es creada con un sistema operativo Ubuntu, por lo cual a partir de este momento, estaremos trabajando con un sistema operativo basado en GNU/LINUX. La primera parte antes del carácter **@** indica el usuario con el cual estamos conectados, y la parte posterior, el nombre que tiene la máquina virtual.

En este caso, la terminal y el proyecto se llaman **blog**; sin embargo, esta característica se estableció en el archivo **Homestead.yaml**, el cual se generó cuando ejecutamos el comando **vendor\\bin\\homestead make**.

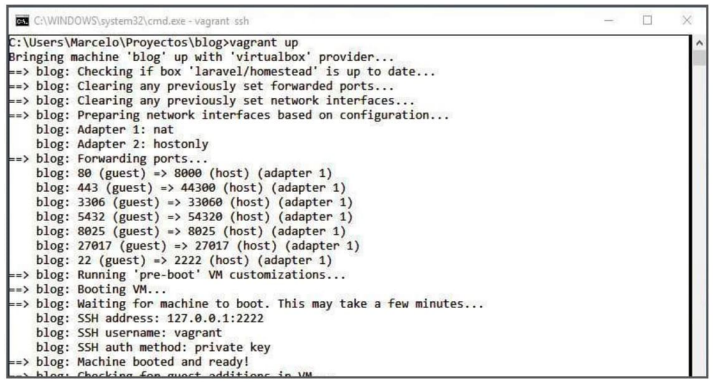


Figura 1. Recordemos que el comando **homestead make** analiza nuestra máquina para establecer los parámetros de **Homestead.yaml**.

Carpetas compartidas

Si listamos los archivos donde acabamos de iniciar sesión, lo cual puede hacerse a través del comando **ls -al**, veremos carpetas y archivos correspondientes al usuario **vagrant** de nuestra máquina **blog**.

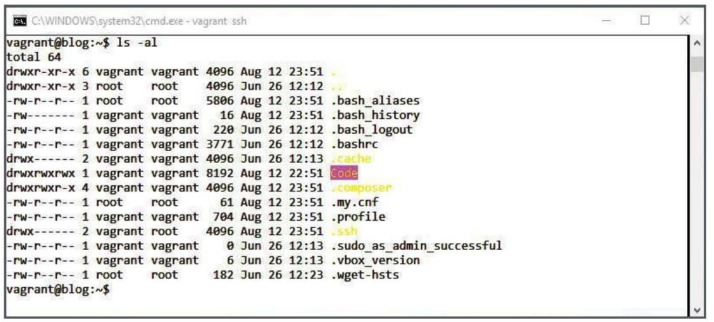


Figura 2. Las sesiones mediante el comando **vagrant ssh** siempre se inician en la carpeta **/home/vagrant**.

La primera impresión es que este listado no muestra ninguno de los archivos de nuestro proyecto, pero vemos que hay una carpeta denominada **Code**. Si ingresamos en ella y la listamos, veremos una estructura de carpetas y archivos como la de un proyecto Laravel.

Virtualbox nos permite ejecutar máquinas virtuales en sistemas operativos Windows, Linux y también Mac OSX, entre otros.

Iniciando Homestead

Es recomendable seguir el log que se genera a medida que se va levantando la máquina virtual, ya que éste brinda información clave sobre la configuración de la misma; por ejemplo, muestra si hay cambios en la máquina que estamos utilizando, nos indica qué puertos desde la máquina virtual se redireccionan hacia qué puertos de la máquina física, e indica el método de conexión que se va a utilizar.

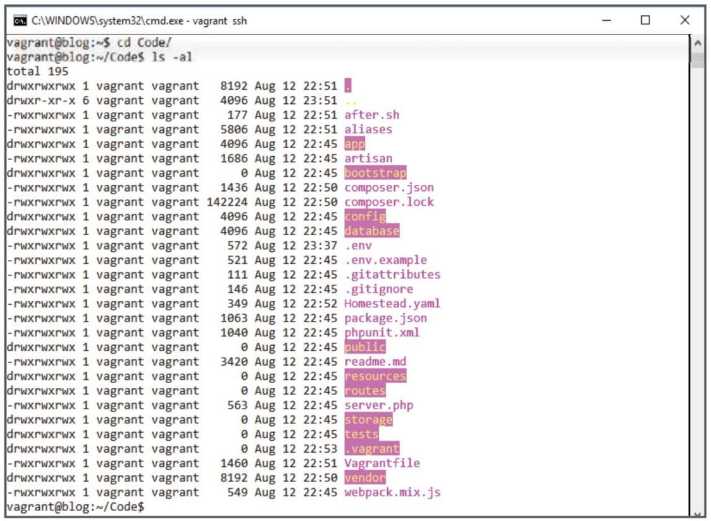


Figura 3. Al trabajar con máquinas virtuales, es necesario configurar carpetas compartidas entre la máquina virtual y la física.

Observemos el archivo **Homestead.yaml** en las siguientes líneas:

```
folders:
    -
        map: 'C:\Users\Marcelo\Proyectos\blog'
        to: /home/vagrant/Code
```

Lo que determina esta sección del archivo es el punto de contacto que va a existir entre nuestras máquinas, la virtual y la física. En la sentencia **map** aparece la ruta de nuestra máquina física en la que se encuentra nuestro proyecto.

En la sentencia **to** aparece la ruta de nuestra máquina virtual en la cual se montará esta carpeta.

Estas carpetas están permanentemente sincronizadas, pero como es bueno ver para creer, realicemos una prueba.

Ingresemos en nuestra máquina física en la carpeta **public** y creemos un archivo **hola.php** con el siguiente contenido:


```
<?php

echo "Hola Homestead";

phpinfo();
```

Luego ingresemos en nuestro navegador en la dirección **http://192.168.10.10/hola.php**.

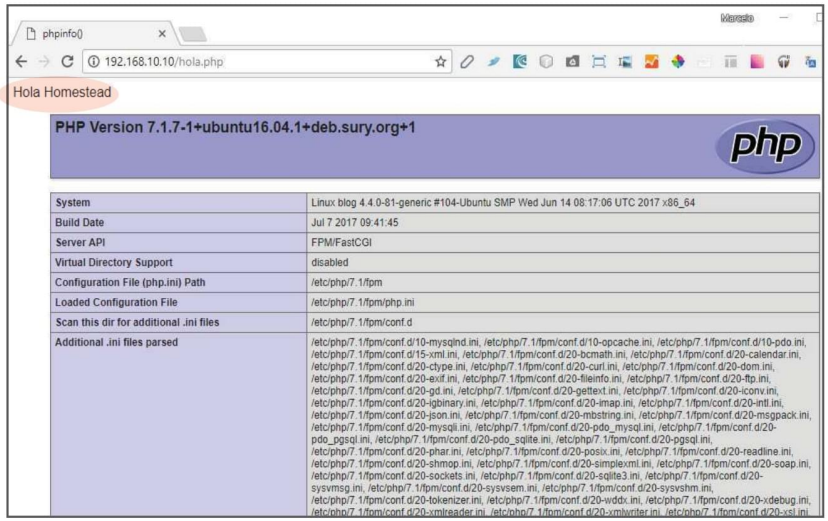


Figura 4. A través del servidor web de nuestra máquina virtual, accedemos a un archivo creado desde nuestra máquina física.

En la **Figura 4** vemos que algunos puertos de la máquina virtual se redireccionan a la máquina física; entre ellos, el puerto 80 de la virtual, que redirecciona al 8000 de la física. Si cambiamos la ruta utilizada por **http://localhost:8000**, veremos el mismo resultado, y esto se debe a esta propiedad, la cual hace posible acceder a los servicios de la máquina virtual con herramientas de la física.

Si ingresamos en la carpeta **Code/Laravel/public** en nuestra máquina virtual y listamos los archivos mediante **ls -al** también veremos **hola.php**.

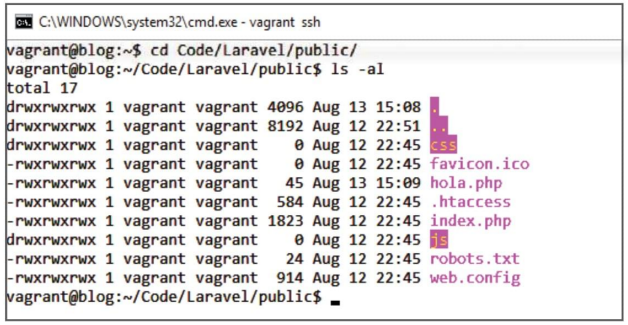


Figura 5. La sincronización funciona en ambas vías, es decir, también podemos crear archivos desde la máquina virtual y verlos en la física.

Utilizando la terminal

Recordemos que PHP puede ser ejecutado a través tanto de un servidor web (en el caso de Homestead se utiliza **Nginx**), como de una interfaz de línea de comando, también conocida como **CLI** (*Command Line Interface*) por sus siglas en inglés.

Laravel brinda una interfaz de línea de comando denominada **Artisan**, la cual nos permite realizar diversas tareas. Para utilizarla, debemos abrir una terminal y posicionarnos en la carpeta de nuestro proyecto. En el caso de Homestead, primero es necesario iniciar sesión en la terminal de la máquina virtual, ejecutando el comando **vagrant ssh**.

Una vez iniciada la sesión, nos posicionamos en la carpeta de nuestro proyecto y ejecutamos el comando **php artisan**, el cual listará todos los comandos de Artisan disponibles.

Trabajar con máquinas virtuales

Es posible acceder a Artisan por una interfaz de línea de comando **(CLI)** tanto desde el sistema operativo de nuestra máquina física como desde la máquina virtual. Lo recomendable es siempre acceder desde el sistema donde se ejecute el entorno, ya que puede diferir la configuración establecida para php-cli e, incluso, pueden llegar a tener distintas versiones de PHP.

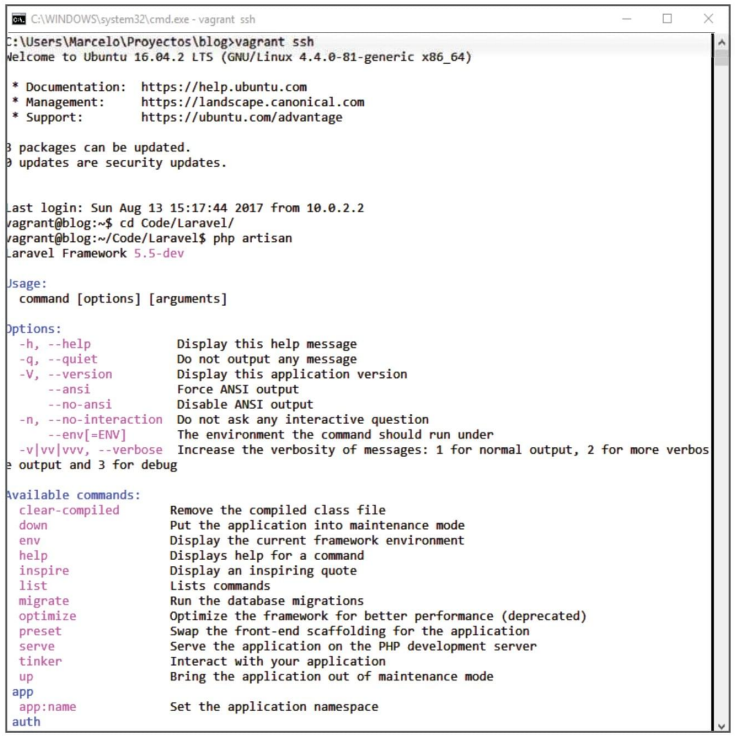


Figura 6. Recordemos que Laravel es el framework para **Web Artisans**, de allí el nombre de su CLI.

CLI

Una de las ventajas que nos brinda la CLI es que podemos combinar comandos con otras herramientas del sistema operativo, por ejemplo, generar tareas programadas que puedan ser ejecutadas en horarios de menor carga. También, ejecutar los comandos de Artisan por fuera de la CLI, **https://laravel.com/docs/5.5/artisan#programmatically-executing-commands.**

Ejecutemos nuestro primer comando de Laravel escribiendo **php artisan down** y luego intentemos acceder nuevamente a nuestro sitio web.

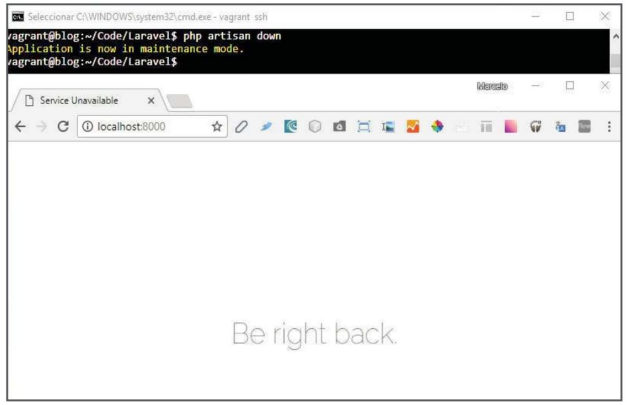


Figura 7. Cuando la aplicación se encuentra en modo mantenimiento, devolverá siempre esta pantalla.

Para volver a levantar nuestro sitio, ejecutamos el comando **php artisan up**.

En el capítulo dedicado a Artisan analizaremos los principales comandos y crearemos los nuestros.

ESTRUCTURA DE ARCHIVOS Y CARPETAS

Como ya lo mencionamos, un framework nos brinda algo más que código, también nos da una manera de organizarlo.

Laravel fue modificando la estructura de carpetas a lo largo de sus distintas versiones y actualmente nos presenta una estructura bastante simple de entender a primera vista. Analicemos la estructura presentada para la versión 5.5.

app	Es la carpeta donde generaremos el código fuente vinculado a la lógica de nuestra aplicación.
bootstrap	Es donde se encuentran los archivos para iniciar el framework. Contiene la lógica para cargar todos los componentes propios y, a la vez, los que se instalan mediante Composer. Esta carpeta no tiene relación con el conocido framework de diseño de sitios web.
config	Es donde se almacenan los archivos de configuración de la aplicación.
database	En esta carpeta vamos a encontrar todo lo necesario para generar la estructura de nuestra base de datos y poblarla. La configuración a la base no se encuentra aquí sino en la carpeta **config**.
public	Es la carpeta raíz de nuestro sitio web, la que debe configurarse como punto de entrada del sitio. Todo lo que se agregue aquí será accesible desde el navegador.
resources	Es la carpeta que contiene la lógica para la construcción de las interfaces visuales.
routes	Aquí se encuentran las rutas, es decir, los puntos de entrada que exponemos en el sitio web.
storage	Es una carpeta de almacenamiento de datos, caché y logs que genera el framework.
tests	Es donde se concentra la lógica de las pruebas automatizadas de nuestro sistema.
vendor	Como lo mencionamos en el **Capítulo 1**, es una carpeta creada por Composer en donde se almacenan todas las dependencias de nuestro proyecto.

Como sabemos, una de las ventajas que nos brinda un framework es la de darnos una estructura que nos permita organizar y dividir mejor el trabajo. Si observamos con detalle la estructura de carpetas, visualizaremos esta organización.

En general, en el mercado de desarrollo de sitios web, suelen realizarse tres distinciones de desarrolladores: programadores **frontend**, **backend** y **full stack**.

Es importante considerar que cada tipo de programador que el mercado requiere, debe poseer un conjunto de habilidades específicas. Por lo tanto, es de suma importancia especializarnos para adquirir estas habilidades.

Los programadores frontend se encargan de todo lo relacionado con la generación de interfaces de usuario; en Laravel, trabajarían principalmente en la carpeta **resources**. Los programadores backend trabajan en el procesamiento final y la persistencia de datos; en Laravel lo harían principalmente en la carpeta **app**. Los programadores full stack se desenvuelven como programadores backend y frontend.

A través de esta estructura de carpetas podemos ver que Laravel se adapta a las necesidades del mercado y nos brinda diferentes espacios para que el trabajo en grupo sea más sencillo.

En cuanto a los archivos, es importante destacar los siguientes grupos:

Archivos de variables de entorno	Son los archivos **.env** y **.env.example** (provienen del inglés environment). Veremos más al respecto en la sección **Configuración**.
Archivos de Git	Git es uno de los sistemas de versionado de código más utilizados del mercado y se emplea en el proyecto Laravel de Github. Estos archivos contienen información relevante para Git, y son **.gitignore** y **.gitattributes**.
Archivos de dependencias	En el **Capítulo 1** vimos el uso de **composer.json** y **composer.lock**, que contiene información sobre la versión exacta de las librerías instaladas en nuestro proyecto. El archivo **package.json** tiene información sobre las dependencias del repositorio de **npm**, que estudiaremos en el **Capítulo 10**.
Archivos de Homestead	Se utilizan para levantar la máquina virtual con una configuración determinada. Son los archivos **after.sh**, **Homestead.yaml** y **Vagrantfile**.

CONFIGURACIÓN

La configuración es clave en el funcionamiento del sistema, de allí la importancia de los archivos que nos permiten modificar los parámetros que cambian en relación con el entorno.

Los **archivos de configuración** de un sistema permiten establecer parámetros que pueden cambiar en relación con el entorno en el cual se ejecuta una aplicación. Por ejemplo, no es lo mismo ejecutar una aplicación en nuestra máquina local mientras estamos desarrollando el sistema, que hacerlo en un servidor en producción, al cual acceden los usuarios finales.

Para establecer dichos parámetros existen diferentes estrategias. Pero antes analicemos un poco los archivos que se encuentran en la carpeta **config** de nuestro proyecto **blog**.

```
C:\WINDOWS\system32\cmd.exe
C:\Users\Marcelo\Proyectos\blog>cd config

C:\Users\Marcelo\Proyectos\blog\config>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 3ABB-7BC6

 Directorio de C:\Users\Marcelo\Proyectos\blog\config

12/08/2017  19:45    <DIR>          .
12/08/2017  19:45    <DIR>          ..
12/08/2017  19:45             9.163 app.php
12/08/2017  19:45             3.251 auth.php
12/08/2017  19:45             1.530 broadcasting.php
12/08/2017  19:45             2.598 cache.php
12/08/2017  19:45             3.951 database.php
12/08/2017  19:45             2.059 filesystems.php
12/08/2017  19:45             4.214 mail.php
12/08/2017  19:45             2.481 queue.php
12/08/2017  19:45               980 services.php
12/08/2017  19:45             6.850 session.php
12/08/2017  19:45             1.004 view.php
              11 archivos         38.081 bytes
               2 dirs  66.783.154.176 bytes libres
```

Figura 8. Las distribuciones en varios archivos de configuración contribuyen al orden del proyecto.

Como podemos ver, los archivos están nombrados a partir del tipo de configuración que almacenan, e iremos trabajando con ellos a lo largo de todo el libro.

Antes de comenzar a trabajar con los archivos de configuración, instalaremos el componente **Debugbar**, que nos ayudará a entender de mejor forma los **entornos**.

Debugbar

Debugbar es un componente que brinda información sobre lo que estamos viendo en pantalla, para lo cual incorpora una barra debajo de nuestro sitio web. Es fundamental en un entorno de desarrollo, dado que entrega información sobre lo que está produciendo el código.

Instalar este componente en Laravel 5.5 es tan simple como posicionarse en la carpeta raíz de nuestro proyecto y ejecutar **composer require barryvdh/laravel-debugbar:dev-master**. Luego de haber instalado el componente, accedemos a nuestro sitio web nuevamente y visualizaremos debajo una barra que brinda información.

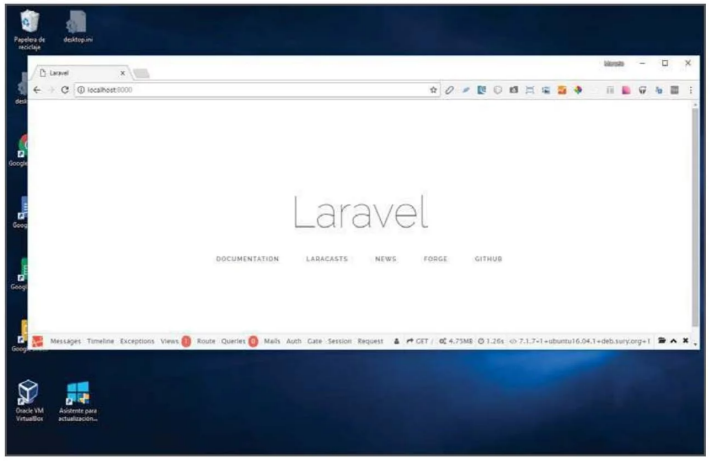


Figura 9. La información aparece segmentada en categorías, por lo que recurriremos a Debugbar durante todo el desarrollo.

Package Auto-Discovery

En versiones anteriores a Laravel 5.5 es necesario modificar el archivo **config/app.php** para agregar los servicios que incorpora el componente y los alias de las funciones. En la nueva versión se incorpora el descubrimiento automático de servicios y alias, por lo cual no tendremos que hacer ninguna modificación; Composer lo hace por nosotros.

Cambiar la configuración

Como ya dijimos, Debugbar es fundamental en un entorno de desarrollo, cuando un desarrollador se encuentra trabajando en los cambios y necesita ver información sobre lo que está generando el código. Sin embargo, en un entorno productivo, Debugbar sería contraproducente, puesto que brindaría información que a los usuarios finales no les interesa y, además, éstos también estarían visualizando un componente que probablemente no se alinee con la estética del sitio, lo cual los induciría a creer que el sitio tiene errores.

Para quitar la Debugbar abrimos el archivo que se encuentra en **config/app.php** con nuestro editor favorito. En este caso, utilizaremos **Atom**, que se puede descargar e instalar de forma gratuita desde **https://atom.io/**. Buscamos la línea:

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

Debemos modificarla de la siguiente forma:

```
'debug' => false,
```

Si accedemos otra vez a nuestro sitio, veremos que la Debugbar desapareció. Esto no quiere decir que se haya desinstalado, sino que la misma sólo se encuentra visible si la aplicación está en modo **debug**, una propiedad de configuración que modificamos cuando introdujimos el cambio.

Interfaz de entorno de desarrollo (IDE)

Existen diversos IDE para PHP. Al ser un lenguaje interpretado (es decir que no se convierte a otro lenguaje de más bajo nivel para que pueda ser ejecutado), PHP no necesita un IDE en absoluto, sino que es posible utilizar cualquier editor de texto, como **Sublime**, **Gedit**, **Atom**, **Nano** o **Vim**. Estos dos últimos tienen la ventaja de que no necesitan una interfaz gráfica en el sistema operativo, la cual muchas veces está ausente en servidores productivos.

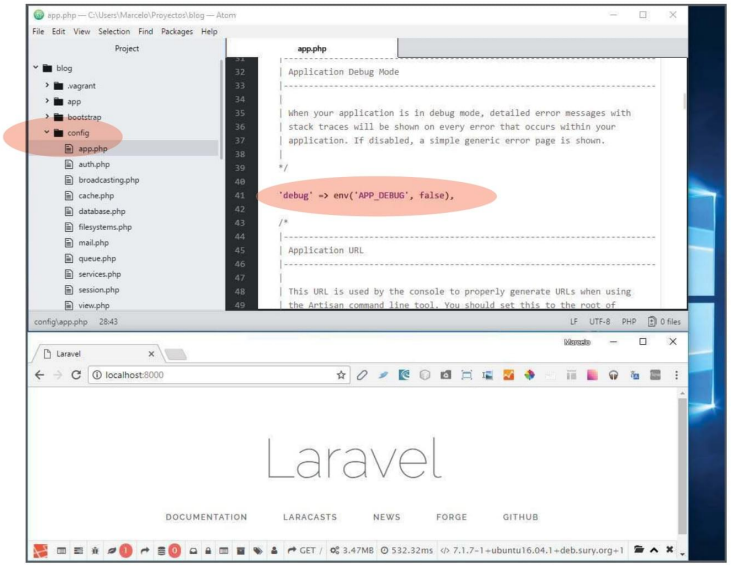


Figura 10. Los archivos contienen documentación relevante para cada parámetro que se establece.

Dotenv

Volvamos a establecer la configuración de **debug** al valor original y analicemos esa línea por un momento:

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

Podemos observar que en ningún lugar aparece el valor **true** para activar el modo **debug** de nuestra aplicación. Sin embargo, si accedemos otra vez a nuestro sitio por el navegador, veremos la Debugbar activada. En consecuencia, ¿dónde se establece el valor **true**?

En el parámetro **debug** encontramos una función que se denomina **env**. Esta función proviene de la librería **PHP Dotenv https://github.**

com/vlucas/phpdotenv y se encarga de levantar variables de entorno desde archivos **.env** a **getenv()**, **$_ENV** y **$_SERVER** y cargar los parámetros establecidos en ellos en nuestro sistema.

Si trabajamos con Homestead, vamos a encontrar que ya existe un archivo **.env** en la carpeta raíz de nuestro proyecto blog.

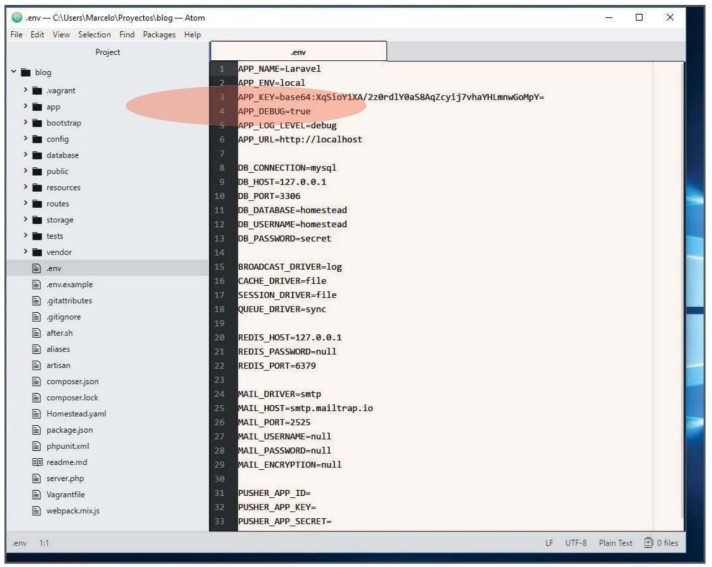


Figura 11. En este punto se establece en **true** al modo debug de nuestra aplicación.

Si establecemos en **false** el parámetro **APP_DEBUG** en el archivo **.env** y accedemos nuevamente al sitio, veremos que Debugbar se encuentra otra vez desactivado.

Con esta información ya podemos saber que la función **env** toma dos parámetros. El primero es una cadena de texto, y lo que hace es buscar en las variables de entorno del sistema el valor correspondiente a dicha cadena; en este caso, la cadena sería **APP_DEBUG**. El segundo parámetro establece el valor que se asignará en caso de no encontrar la cadena **APP_DEBUG** en la configuración.

La ventaja de utilizar archivos **.env** es que todos los parámetros de configuración vinculados al entorno se abstraen del código fuente. Esto significa no tener que modificar el código para aplicar cambios en el entorno de ejecución. Por eso, los archivos **.env** no deben cargarse en Git o el sistema de control de versiones que se utilice para el proyecto.

Acceder a la configuración

Para poder acceder a los parámetros de configuración debemos utilizar el método **config**. Veamos algunos ejemplos de uso:

```
$value = config('app.timezone');
$value = config('auth.defaults.guard');
$value = config('cache.prefix');
$value = config('session.driver');
```

Por omisión, Dotenv no se encarga de sobreescribir las variables de entorno que corresponden al sistema.

Si observamos los archivos de la carpeta **config**, podemos detectar un patrón. El parámetro del método **config** es una cadena separada por puntos. El primer punto determina el archivo que se buscará, siempre dentro de la carpeta **config**; el resto de los puntos son los valores de los índices del array asociativo que se utilizan en los archivos de configuración.

Siguiendo esta misma nomenclatura, podemos tanto agregar nuevos parámetros a los archivos ya existentes como también crear nuestros propios archivos.

Dotenv

Si bien no debemos incluir los archivos **.env** en el repositorio de código fuente de nuestro proyecto, podemos considerar como una buena práctica incluir un archivo **.env.example** en el repositorio; de esta forma, es posible indicar qué variables hay que configurar para que se ejecute nuestro sistema. Debemos saber que Dotenv también permite definir variables requeridas; **https://github.com/vlucas/phpdotenv#requiring-variables-to-be-set.**

ARQUITECTURA

Laravel basa parte de su arquitectura en un patrón de diseño muy conocido en el mundo del desarrollo web, denominado Patrón **Modelo-Vista-Controlador** o también **patrón MVC**.

Este patrón propone separar la lógica de un proyecto en tres partes:

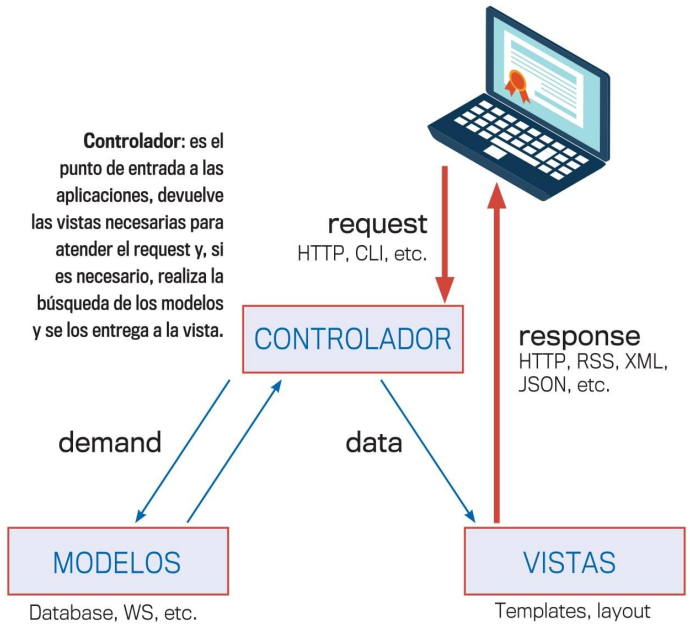


Figura 12. En una aplicación web, los requests son recibidos por los controladores, y éstos retornan responses con vistas.

Vistas en Laravel

Cuando listamos la arquitectura de carpetas que presenta el framework en la sección Estructura de archivos y carpetas, vimos que todo lo relacionado a la construcción de interfaces visuales se encontraba en la carpeta **resources**.

Cuando pensamos en interfaces visuales en web, lo primero que nos viene a la mente es código HTML, y es lo que podemos encontrar en Laravel en la carpeta **resources/views**. Pero desde hace ya un tiempo, las interfaces web se volvieron mucho más complejas, y existen diferentes técnicas para generar una interfaz web. Por eso, y para mantenernos organizados, Laravel presenta dos subcarpetas más.

La carpeta **lang** contiene toda la lógica vinculada a la visualización de contenido en diferentes idiomas, y la carpeta **assets** contiene tanto lógica como recursos que se utilizan en las vistas, por ejemplo, archivos javascript, css e imágenes.

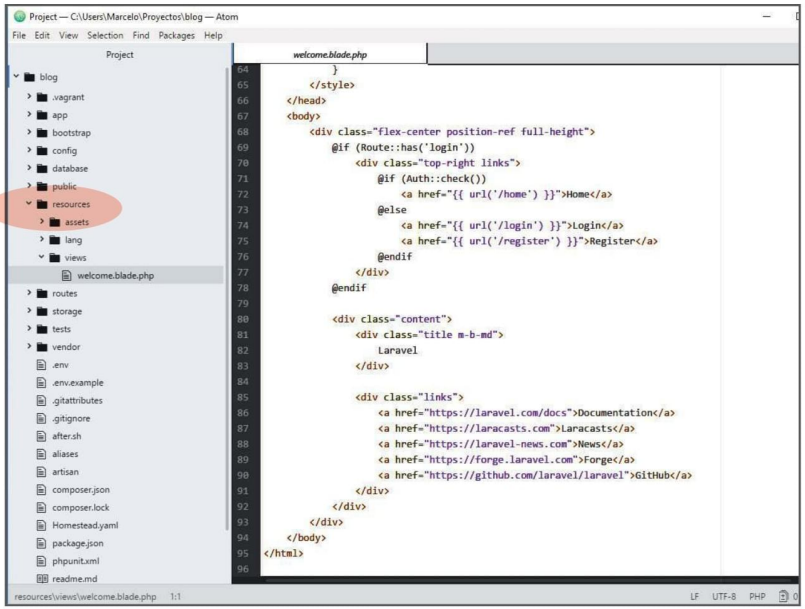


Figura 13. La vista **welcome.blade.php** es la que vemos cada vez que accedemos a un sitio recientemente creado con Laravel.

Controladores en Laravel

Recordemos que Laravel puede servir tanto para aplicaciones web como para aplicaciones ejecutables por CLI. Es por eso que si observamos la carpeta **app**, encontraremos diferentes subcarpetas.

Dado que los controladores interactúan con un servidor web recibiendo peticiones (también conocidas como requests) y devolviendo respuestas (también conocidas como responses), y todo esto lo hacen utilizando el famoso protocolo de la Web, el lugar donde encontraremos los controladores es **app/Http/Controllers**.

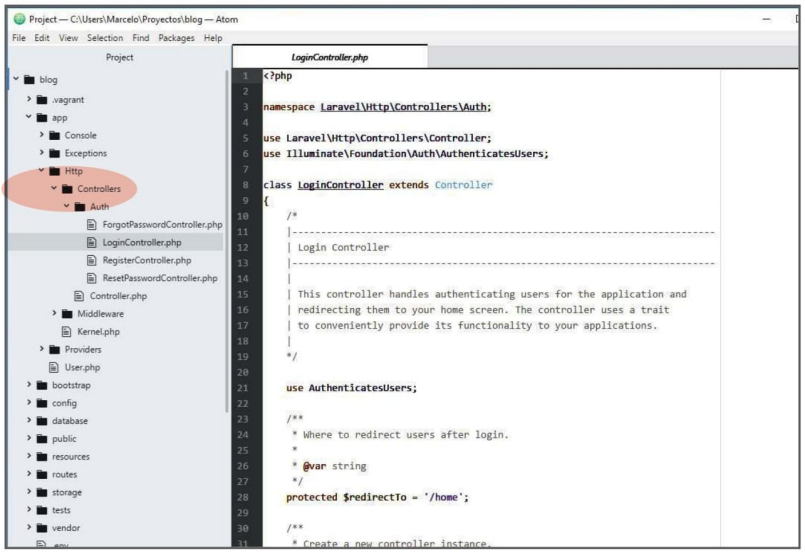


Figura 14. En una instalación nueva de Laravel encontraremos los controladores del componente de autenticación.

Modelos en Laravel

Es importante tener en cuenta que un modelo es la representación de información de nuestro sistema. Muchas veces suele transmitirse que los modelos contienen la lógica de base de datos, pero esta definición es incompleta, porque hoy en día las aplicaciones pueden obtener los datos desde otros lugares, por ejemplo, desde un servicio web.

Por todas las características que hemos conocido y algunas opciones que analizaremos en los próximos capítulos, podemos decir que Laravel es más que un framework MVC.

Si trabajamos con una base de datos, Laravel cuenta con una librería que permite generar modelos para interactuar con las tablas donde se almacena esa información. A este tipo de librerías se las conoce como ORM, por sus siglas en inglés de **Object-Relational-Mapping**; el **ORM** de Laravel es **Eloquent**.

Hay incluso quienes separan los modelos en dos partes: una de repositorio de la información o clases repository, y otra con la representación de la información propiamente dicha y la lógica de negocios. Profundizaremos más respecto de este tema en la sección de modelos.

Laravel almacena los modelos en archivos PHP en la carpeta **app**.

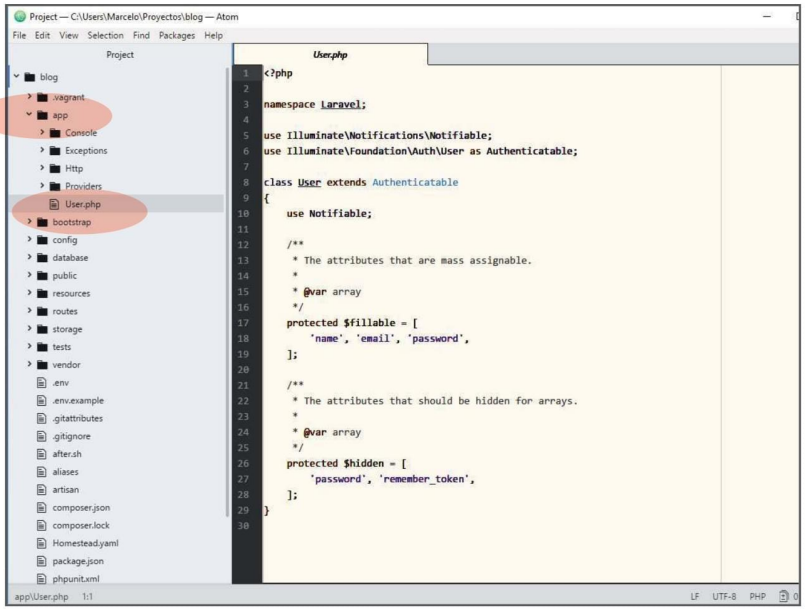


Figura 15. El modelo **user.php** es parte del componente de autenticación y puede extenderse para adaptarse a nuestras necesidades.

Extendiendo MVC

Como ya mencionamos, Laravel basa parte de su arquitectura en este patrón, pero esto no significa que esté constituido únicamente por vistas, modelos y controladores. Veamos los principales componentes de arquitectura del framework.

Rutas

Desde que surgió la Web y, con ella, los motores de búsqueda, las URL de los sitios fueron evolucionando de manera tal que la construcción de URL amigables puede tornarse una tarea compleja. A su vez, recordemos que Laravel puede ser ejecutado desde una aplicación web y, también, de otras maneras. Para poder organizar todo esto dispondremos de la carpeta **routes** en la raíz de nuestro proyecto.

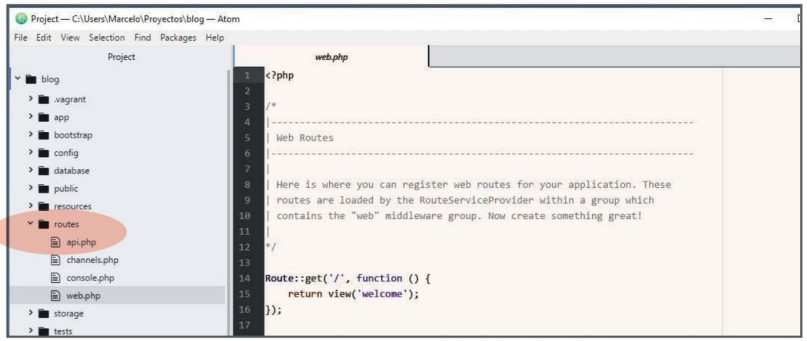


Figura 16. La ruta **/** es el home de nuestro sitio. En una instalación mostrará la vista welcome que presentamos en la **Figura 14**.

ORMs en PHP

Existen diversos ORM para PHP. Además de **Eloquent**, uno de los más utilizados es **Doctrine**, el cual es empleado por los frameworks **Symfony** y **Zend Framework**, entre otros. También es importante saber que podemos utilizar un ORM fuera del contexto de un framework, es decir, cargándolo como un componente de una aplicación. La mayoría de ellos pueden ser instalados mediante Composer.

Servicios

Los servicios son el núcleo de conexión entre los diferentes componentes de nuestro framework.

Es muy probable que cada vez que agreguemos un nuevo componente al framework, ya sea mediante Composer o escribiendo nuestro propio componente, estemos creando un **Service Provider** y un **Service Container**, y siempre debemos cargar los providers en **config/app.php**.

Los Service Containers se encargan de administrar las dependencias que son necesarias a la hora de utilizar una clase. Los Service Providers son los que ponen a disposición del framework aquellas clases que fueron creadas en los containers.

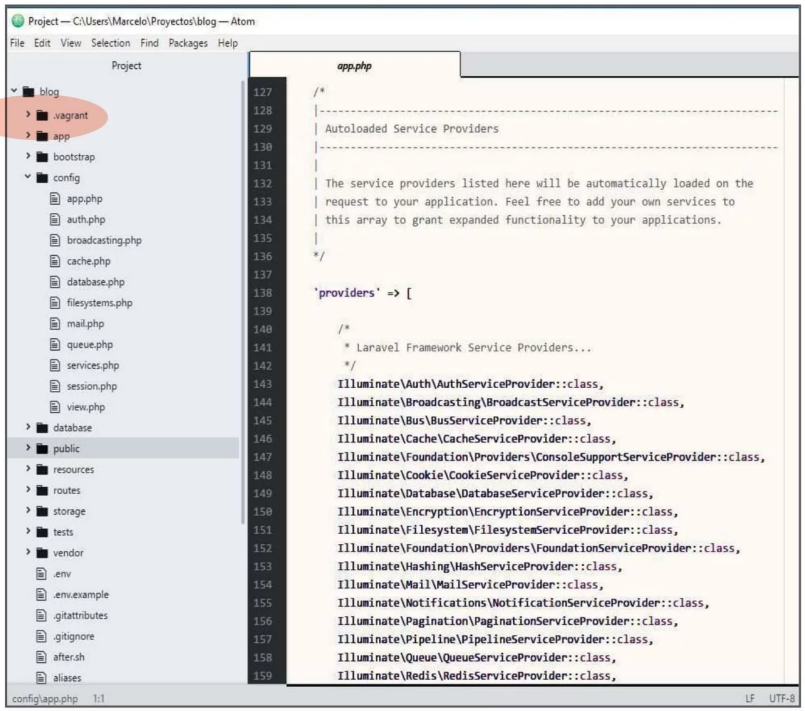


Figura 17. A partir de la versión 5.5, los servicios se cargan automáticamente cuando instalamos componentes mediante Composer.

Kernels

Como ya mencionamos, los Service Providers son los encargados de poner a disposición todas las herramientas que provee el framework, pero ¿quién carga a los Service Providers? Es allí donde intervienen los kernel.

Si trabajamos en una aplicación web, Laravel está pensado para que el **documentroot** de nuestra aplicación sea la carpeta **public**. En ella vamos a ver un archivo **index.php** que será el punto de entrada de todas las peticiones que reciba nuestro sitio; en él se realiza la carga del **kernel http**. Si trabajamos en una aplicación CLI, estaremos invocando al archivo **artisan.php** en todas las peticiones, y en él es donde se realiza la carga del **kernel Console**.

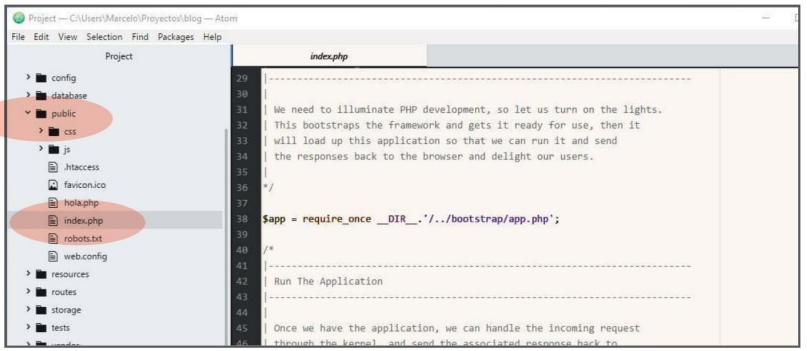


Figura 18. La invocación al Kernel Http resume la simpleza de un llamado web donde se recibe un **request** y se retorna una **response**.

Resumen Capítulo 02

En este capítulo comenzamos a interactuar con Artisan, la interfaz de línea de comando que provee Laravel. Luego analizamos la estructura de archivos y carpetas, introdujimos el concepto de ambiente y empezamos a trabajar en la configuración del framework utilizando Dotenv. Más adelante, vimos cómo instalar nuevos componentes e instalamos Debugbar. Por último, analizamos los aspectos principales de la arquitectura MVC, su implementación en Laravel y los principales componentes adicionales que utiliza.

ACTIVIDADES

Test de Autoevaluación

1. ¿En qué consiste el uso de las carpetas compartidas?
2. ¿Qué es Artisan? ¿Cómo podemos obtener un listado de sus comandos?
3. ¿Cuáles son las ventajas que brinda la estructura de archivos y carpetas de Laravel? ¿Cómo se relacionan con los perfiles del mercado?
4. ¿Qué es la configuración de un sistema?
5. ¿Para qué sirve la Debugbar? ¿Cómo se activa/desactiva una vez instalada?
6. ¿Cuáles son las ventajas de utilizar Dotenv?
7. ¿Cómo se accede a los parámetros de configuración?
8. ¿En qué consiste el patrón de arquitectura MVC?
9. ¿Dónde están en Laravel los modelos, las vistas y los controladores?
10. ¿Qué otros componentes, además de los del patrón MVC, incorpora Laravel en su arquitectura?

Ejercicios prácticos

1. Inicie la máquina virtual con **vagrant up**, conéctese mediante **vagrant ssh** y liste los archivos de la carpeta **Code**. Luego, modifique el archivo **Homestead.yaml** en la sección **folders** e introduzca otra ruta de su computadora. Salga de la máquina virtual con el comando **exit** y ejecute **vagrant reload**. Liste nuevamente los archivos de la carpeta **Code**. Observe los cambios en la carpeta e intente ingresar otra vez al sitio web a través del navegador.
2. Elimine la carpeta **vendor**, intente acceder al sitio desde el navegador, luego ejecute el comando **composer install** y acceda nuevamente.
3. Active la Debugbar y ubique en ella la información relacionada a las vistas. Identifique en la estructura el archivo que se utiliza como vista y modifíquelo de manera tal que se muestre el nombre Blog en lugar de Laravel.

03 Rutas

En este capítulo comenzaremos a generar puntos de entrada a nuestro sistema utilizando rutas. Analizaremos las principales características del protocolo HTTP y su relación con el framework. Veremos cómo Laravel maneja las peticiones y respuestas y, a la vez, las estudiaremos también en el navegador web.

RUTAS Y HTTP

Las rutas nos permiten generar las URL de nuestro sitio web; en otras palabras, son el punto de acceso a nuestro sistema. Recordemos que las URL son una cadena de caracteres que identifican a los recursos de una red de forma unívoca.

Las rutas en Laravel se organizan en la carpeta **routes**, donde encontraremos cuatro archivos: **api.php**, **channels.php**, **console.php** y **web.php**. Comencemos a trabajar en el archivo **web.php** y analicemos su código.

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

Route es un servicio de Laravel que permite construir nuestras rutas. Los métodos del servicio se relacionan con los métodos de petición del protocolo HTTP, también conocidos como **verbos HTTP**. Para continuar repasemos los principales:

GET	Es el método más conocido y, probablemente, el más utilizado. Cada vez que ingresamos una URL en la barra de direcciones del navegador, estamos ejecutando este método. Su función es recuperar datos sobre un recurso disponible en la Web.
HEAD	Es igual al método **GET** excepto que sólo devuelve el encabezado de la petición sin el cuerpo. Recordemos que todas las respuestas a un servidor HTTP están compuestas por un encabezado **(header)** y un cuerpo **(body)**. El encabezado contiene información sobre la respuesta que se recibirá en el cuerpo.

Recursos de una red

Siempre que pensemos en Web debemos tener en cuenta que nuestro sistema formará parte de una red, ya sea Internet o una intranet. Sin embargo, tenemos que pensar en grande y considerar que no siempre estaremos trabajando con páginas web, ya que nuestro sistema podría trabajar también como un servicio web. En síntesis, un recurso puede ser tanto una página en un sitio web como datos sobre una entidad en un servicio web.

POST	Este método se utiliza para enviar datos al servidor solicitando que éstos sean procesados por la URL.
PUT	Solicita la creación o el reemplazo de un recurso por el contenido enviado en la petición. Un pedido **PUT** con un determinado contenido en una URL realizado de forma exitosa sugiere que luego podamos hacer un método **GET** a la misma URL y obtendremos el mismo contenido, aunque no hay garantías de que esto sea así. Hay que tener en cuenta que este método no suele estar habilitado en hostings compartidos y que versiones antiguas de los navegadores no lo reconocen.
DELETE	Este método solicita eliminar el recurso de una determinada URL.
OPTIONS	Permite conocer los métodos de comunicación disponibles en el servidor para un determinado recurso sin implicar una acción sobre el recurso.
PATCH	Permite modificar un recurso existente en una URL. A diferencia de **PUT**, este método asume que el recurso ya existe y que debe generarse una nueva versión del mismo, mientras que **PUT** implica la creación del recurso en caso de que no exista.

En resumen, podemos decir que los métodos disponibles del servicio **route** son los siguientes:

```
Route::get($url, $callback);
Route::post($url, $callback);
Route::put($url, $callback);
```

Diferencias entre POST y PUT

La diferencia fundamental entre **POST** y **PUT** es que, si bien ambos permiten enviar datos al servidor, los dos métodos tienen intenciones diferentes. La intención en un método **POST** es que el contenido enviado en la petición sea **interpretado y procesado** por la URL en cuestión, mientras que en el método **PUT** el objetivo es que el contenido sea **insertado o reemplazado** por la URL en cuestión.

```
Route::patch($url, $callback);
Route::delete($url, $callback);
Route::options($url, $callback);
```

Todos siguen la misma estructura: el primer parámetro es la URL que estableceremos para nuestro recurso, y el segundo, el callback, es decir, la lógica que atenderá este pedido y devolverá la respuesta.

La ruta **/** que tenemos definida en **web.php** tiene como callback devolver la vista **welcome.blade.php**.

Requests y Responses

Agreguemos esta nueva ruta en el archivo **web.php** e ingresémosla a través del navegador:

```
Route::get('/hola-mundo', function () {
    return "hola mundo!";
});
```

Laravel convierte automáticamente las cadenas de texto de las rutas y los controladores en objetos del tipo **Response**, que son clases que representan la respuesta HTTP que se está enviando.

Tengamos en cuenta que las representaciones de las peticiones se presentan en objetos del tipo **Request**.

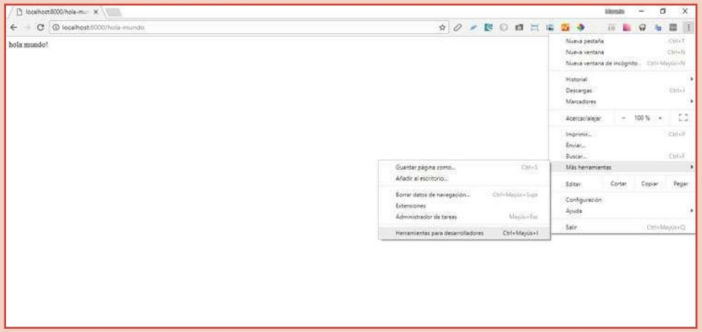
Como ya dijimos, todas las peticiones y respuestas están compuestas de un encabezado y un cuerpo. Una manera de ver esta información es utilizando las herramientas para desarrollador de nuestro navegador.

Divide y reinarás

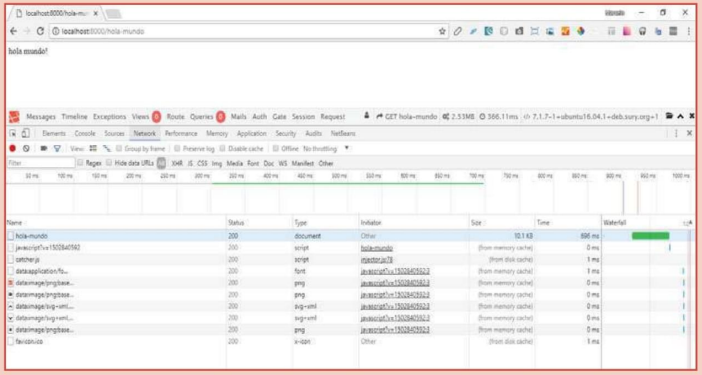
Recordemos que Laravel basa parte de su arquitectura en el patrón MVC. En este capítulo estamos desarrollando sin hacer uso ni de Modelos, ni de Vistas ni de Controladores, y podremos notar que el resultado es un poco caótico. La mejor manera de obtener código legible es dividiendo la lógica, para lo cual deberemos incorporar MVC y otros componentes más en nuestro sistema.

VISUALIZAR PEDIDOS Y RESPUESTAS HTTP EN CHROME

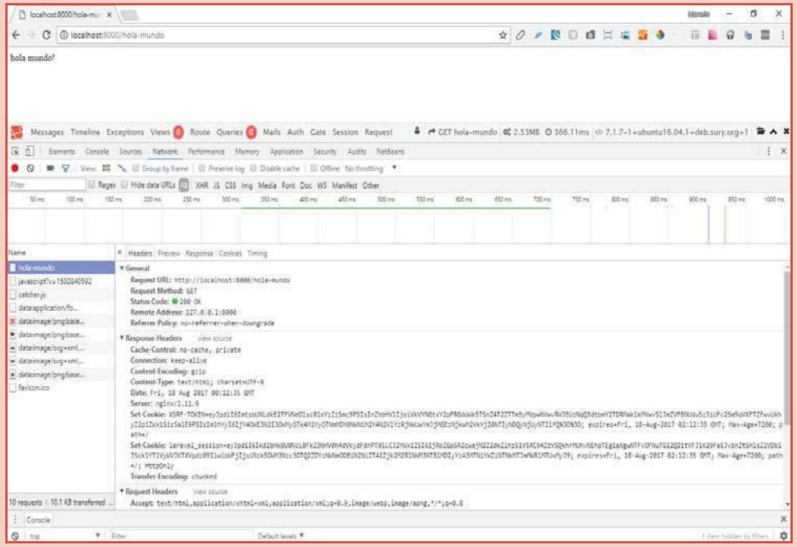
01 Ingrese en Chrome a la ruta **/hola-mundo** y seleccione el menú **Más herramientas/Herramientas Para desarrolladores.**



02 A continuación seleccione la pestaña **Network**. Si ésta aparece vacía presione [F5]. Esta pestaña muestra todos los requests generados para cargar esta página. Además del request **/hola-mundo** vemos también algunos necesarios para la Debugbar. Seleccione el request **/hola-mundo**.



03 En esta pantalla vemos toda la información sobre la petición. Además de los encabezados, podemos ver también el código de respuesta, la URL y la IP.



Antes de salir de las herramientas del desarrollador, busquemos en los **Response Headers** el atributo **Content-Type**. Este atributo nos indica el tipo de contenido que nos devolverá el servidor. Podemos observar que para la URL **/hola-mundo** su valor es **text/html**.

Códigos HTTP

El protocolo http, además de brindarnos métodos para realizar peticiones, brinda códigos que nos indican el estado. Los códigos se separan en grupos que nos dan un primer indicio de la respuesta obtenida. Los verbos HTTP y los códigos de estado del protocolo son muy importantes a la hora de desarrollar servicios web.

Agreguemos la siguiente ruta y busquemos el mismo atributo en el encabezado, de la siguiente forma:

```
Route::get('/hola-json', function () {
    return [1, 2, 3];
});
```

En esta oportunidad, veremos que el valor del **Content-Type** cambió por **application/json**. Esto significa que Laravel automáticamente reconoce en las respuestas de las rutas a las cadenas de texto y las devuelve como código HTML, y a los arrays los devuelve como código JSON. En ambos casos convierte estas respuestas en objetos del tipo **Response** que brindan toda la información que vemos en el navegador.

También podemos construir nuestras propias responses estableciendo todos los valores de los headers, creando nuestros propios headers e, incluso, agregando cookies como en el siguiente ejemplo:

```
Route::get('/home', function () {
    return response('Esta es nuestra home', 200)
                ->header('Content-Type', 'text/plain')
                ->header('Mi-Header-Personalizado', 'Mi
Valor Personalizado')
                ->cookie('hola-cookies', 'mi-valor-en-
criptado', 60);
});
```

Content-Type

El sitio web para desarrolladores de Mozilla es uno de los más completos sobre el desarrollo web en español. En el enlace **https://developer.mozilla.org/es/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Lista_completa_de_tipos_MIME** podemos ver todos los tipos de Content-Type y en **https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/JSON** podemos ver más acerca del tipo application/json.

Ahora que ya tenemos nuestra ruta **/home**, redireccionemos la ruta **/** de manera tal que **/home** se convierta en nuestra pantalla de inicio.

```
Route::redirect('/', '/home');
```

Parámetros en las rutas

Los parámetros son segmentos de la ruta que podemos recuperar mediante variables y que funcionan como punto de entrada a nuestro sistema. Agreguemos la siguiente ruta:

```
Route::get('/hola/{persona}', function ($persona) {
    return "hola " . $persona;
});
```

Al encerrar una parte de la ruta entre llaves, estamos indicando que ese parámetro será variable y recuperaremos su información a partir de, valga la redundancia, una variable que se denominará con el mismo nombre, en este caso, **persona**.

Veamos qué sucede si agregamos una ruta similar pero con otra variable:

```
Route::get('/hola/{usuario}', function ($usuario) {
    return "bienvenido " . $usuario;
});
```

En esta oportunidad, podemos ver que las dos rutas tienen la misma estructura:

- **'/hola/{persona}'**
- **'/hola/{usuario}'**

En estos casos, Laravel sólo identificará la que se encuentre definida primero en el archivo **routes.php**, mientras que la segunda quedará obsoleta puesto que no será posible de acceder.

Parámetros opcionales

Probemos ahora a ingresar en la ruta sin agregar el parámetro variable, es decir, ingresar en **http://localhost:8000/hola/**. Veremos que el sitio web arroja un **error 404**, es decir, página no encontrada, y esto es porque la ruta ingresada no coincide con ningún patrón. Por más que el parámetro establecido sea variable, se encuentra definido de manera obligatoria, y en consecuencia, no es posible vincular la ruta ingresada con una ruta válida. Realicemos la siguiente modificación:

```
  Route::get('/hola/{persona?}', function ($persona = "humano") {
      return "hola " . $persona;
  });
```

Si intentamos ingresar nuevamente en **http://localhost:8000/hola/**, en esta oportunidad tendremos éxito y esto se debe a que acabamos de convertir el parámetro **$persona** en opcional. Para hacerlo, es necesario agregar el carácter **?** y, a su vez, establecer un valor por default a la variable **$persona**. Ahora podremos ingresar en la ruta, estableciendo el parámetro o no.

Restricción de parámetros

Veamos qué sucede si ingresamos en **http://localhost:8000/hola/1234**. Si nos resulta raro que estemos saludando a 1234 y no a una cadena de caracteres, no nos preocupemos, Laravel tiene una solución para esto y es la **restricción de parámetros** a partir de expresiones regulares.

Modifiquemos el código e intentemos otra vez:

```
  Route::get('/hola/{persona?}', function ($persona = "humano") {     return "hola " . $persona;
  })->where('persona', '[A-Za-z]+');
```

Ahora volvemos a obtener un error 404. Con esta modificación,

introdujimos una validación en el parámetro **$persona** de la ruta a través de la expresión regular **[A-Za-z]+**, estableciendo que sólo admita valores alfabéticos entre mayúsculas y minúsculas. Por lo tanto, si ingresamos un número o un carácter especial, la ruta no será válida y el sistema devolverá un error 404.

Para continuar agreguemos una nueva ruta que también utilice un parámetro **persona**:

```
  Route::get('/hola/{persona?}', function ($persona = "humano") {
      return "hola " . $persona;
  })->where('persona', '[A-Za-z]+');

  Route::get('/bienvenido/{persona?}', function ($persona = "humano") {
      return "bienvenido " . $persona;
  })->where('persona', '[A-Za-z]+');
```

Si observamos el código, podemos ver que tenemos que definir la restricción del parámetro **persona** por cada vez que lo utilicemos. Si nuestro sistema tuviese cientos de rutas y el día de mañana nos indican que a partir de ese momento sólo serán admisibles los caracteres en letras minúsculas, perderemos algo de tiempo modificando todas las ocurrencias del archivo. En estos casos lo que nos conviene hacer es establecer una restricción de parámetros de forma **global**. Para hacerlo, debemos modificar el archivo **app/Providers/ RouteServiceProvider** y cambiar la función **boot** de la siguiente manera:

Pensar en grande

Como programadores, es fundamental ser capaces de detectar patrones. Esta práctica nos permite atacar problemas encapsulándolos en código que podemos reutilizar. En esta oportunidad vemos que el parámetro **$persona** sólo se repite una vez, pero siempre debemos pensar que nuestro proyecto podría formar parte de un sistema más grande y, entonces, nos conviene analizar el costo/beneficio de hacer una implementación más general.


```
  public function boot()
  {
      Route::pattern('persona', '[a-z]+');
      parent::boot();
  }
```

Ahora, quitemos la restricción particular establecida en las rutas **/hola/{persona?}** y **/bienvenido/{persona?}**, ya que no son necesarias debido a que tenemos una restricción global establecida para el parámetro **persona**.

```
  Route::get('/hola/{persona?}', function ($persona = "humano") {
      return "hola " . $persona;
  });
  Route::get('/bienvenido/{persona?}', function ($persona = "humano") {
      return "bienvenido " . $persona;
  });
```

A partir de este momento, cada vez que necesitemos realizar cambios sobre el parámetro, podremos hacerlos en la función **boot** del **RouteServiceProvider**.

```
C:\WINDOWS\system32\cmd.exe - vagrant ssh
vagrant@blog:~/Code$ php artisan route:list
+--------+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------------------------------------------------------+
| Domain | Method                            | URI                               | Name                  | Action                                                                            |
+--------+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------------------------------------------------------+
|        | GET|HEAD|POST|PUT|PATCH|DELETE | /                                 |                       | \Illuminate\Routing\RedirectController                                            |
|        | GET|HEAD                          | _debugbar/assets/javascript       | debugbar.assets.js    | Barryvdh\Debugbar\Controllers\AssetController@js                                  |
|        | GET|HEAD                          | _debugbar/assets/stylesheets      | debugbar.assets.css   | Barryvdh\Debugbar\Controllers\AssetController@css                                 |
|        | GET|HEAD                          | _debugbar/clockwork/{id}          | debugbar.clockwork    | Barryvdh\Debugbar\Controllers\OpenHandlerController@cl                            |
|        | GET|HEAD                          | _debugbar/open                    | debugbar.openhandler  | Barryvdh\Debugbar\Controllers\OpenHandlerController@ha                            |
|        | GET|HEAD                          | api/user                          |                       | Closure                                                                           |
|        | GET|HEAD                          | bienvenido/{persona?}             |                       | Closure                                                                           |
|        | GET|HEAD                          | hola-json                         |                       | Closure                                                                           |
|        | GET|HEAD                          | hola-mundo                        |                       | Closure                                                                           |
|        | GET|HEAD                          | hola/{persona?}                   |                       | Closure                                                                           |
|        | GET|HEAD                          | home                              |                       | Closure                                                                           |
+--------+-----------------------------------+-----------------------------------+-----------------------+-----------------------------------------------------------------------------------+
vagrant@blog:~/Code$
```

Figura 1. A través del comando **php artisan route:list** de **Artisan** podemos listar todas las rutas de nuestro sistema.

Podemos ver que la segunda columna se titula **name** y en todas las rutas que hemos creado aparece vacía; esto es porque no hemos definido nombres para las rutas.

Si bien los nombres no son necesarios, resultan muy útiles ya que, bien empleados, pueden darnos información declarativa sobre la ruta. Modifiquemos nuestras rutas de la siguiente manera:

```
  Route::get('/hola/{persona?}', function ($persona = "humano") {
      return "hola " . $persona;
  })->name('decir.hola');
  Route::get('/bienvenido/{blogger?}', function ($blogger = "Blogger") {
      return "bienvenido " . $blogger;
  })->name('decir.bienvenido');
```

Al ejecutar nuevamente el comando, veremos que aparecen los nombres que les dimos a las rutas. Ahora que hemos definido nombres, podemos usar la función **route**, la cual recibe como parámetro el nombre de una ruta y devuelve la URL que le corresponde. Veamos un ejemplo modificando la ruta **decir.hola**:

```
  Route::get('/hola/{persona?}', function ($persona = "humano") {
      return "hola " . $persona .
      "<br /> <a href='" . route('decir.bienvenido', ["blogger" => $persona]) . "'>Decir Bienvenido</a>";
  })->name('decir.hola');
```

En este código utilizamos la salida de la función **route** para poder establecer el atributo **href** de un elemento **anchor**, generando un enlace hacia la ruta **decir.bienvenido**.

Dado que la ruta **decir.bienvenido** recibe un parámetro, es necesario pasar como segundo parámetro en la función **route** un array asociativo, donde los valores de los índices deberán coincidir con el nombre del parámetro que se espera en la ruta.

En este caso, el parámetro en la ruta es **{blogger?}**, el valor del índice es **"blogger"** y el nombre de la variable en la ruta **decir.bienvenido** es **$blogger**.

Probar otros métodos HTTP

Hasta ahora hemos ingresado solamente métodos **GET** y ya sabemos que con sólo ingresar la URL en nuestro navegador, podremos acceder sin problemas.

Pensemos ahora en nuestro blog. Es muy probable que debamos tener una ruta en la cual podamos ingresar los valores de una noticia, y podemos considerar que, como mínimo, tendrán un título y un cuerpo.

Ingresar el cuerpo de una noticia como parámetro a través de un navegador será una tarea bastante incómoda para nuestros usuarios; lo apropiado para este caso sería utilizar un método **POST**. Añadamos las rutas para agregar y ver una noticia:

```
  //Recuperamos el objeto $request que representa la petición
  Route::post('/noticia', function (Illuminate\Http\Request $request) {

      //Generamos un valor random que utilizaremos como ID
      $id = substr(md5(microtime()), 0, 4);

      //Los datos enviados por post los debemos obtener del objeto request. No son parámetros de la ruta
      $titulo = $request->input('titulo');
      $cuerpo = $request->input('cuerpo');

      //Generamos un array con los datos de la noticia
      $entrada = array("id" => $id, "titulo" => $titulo, "cuerpo" => $cuerpo);

      //Guardamos el array en la sesión utilizando el servicio session del framework
      session([$id => $entrada]);

      //Devolvemos como respuesta la ruta hacia la noticia creada
      return route('noticia.show', ["id" => $id]);
```

```
})->name('noticia.store');

Route::get('/noticia/{id}', function ($id) {
    //Recuperamos los datos de la noticia
    $noticia = session()->get($id);
    $titulo = $noticia["titulo"];
    $cuerpo = $noticia["cuerpo"];

    //Mostramos la noticia
    return "<h1>" . $titulo . "</h1><br><p>" . $cuerpo .
"</p>";
})->name('noticia.show');
```

Si ingresamos en nuestro navegador la URL **http://localhost:8000/noticia**, obtendremos una excepción del tipo **MethodNotAllowedHttpException** por estar utilizado un método (**GET**) en una ruta definida para ser invocada mediante **POST**.

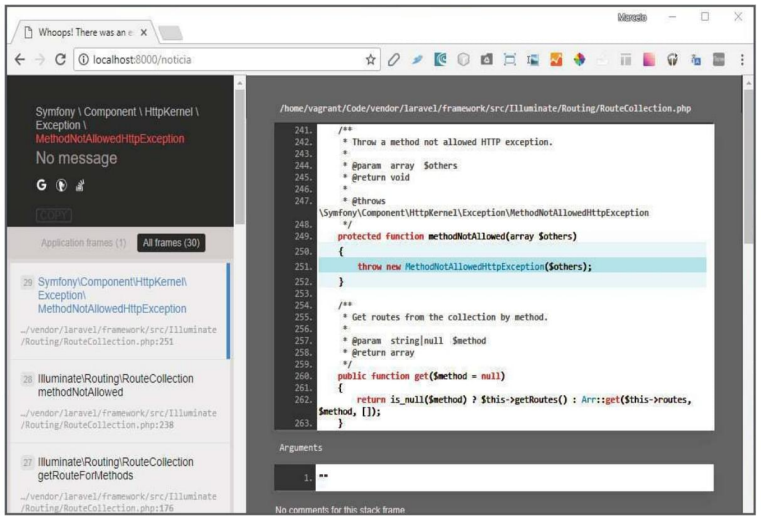


Figura 2. Con el modo debug activado podemos ver información muy útil al momento de producirse excepciones.

Lo normal sería invocar a la ruta **noticia.store** a través de un formulario web, pero si nuestro foco está puesto en este momento en ver cómo almacenar los datos, no querremos dedicar tiempo a construir un formulario con todo lo que ello implica. En estos casos es conveniente utilizar **Postman**.

Postman es una aplicación que podemos descargar de manera gratuita desde **https://www.getpostman.com** y que nos permite realizar peticiones de la manera en la que lo haría un navegador, brindándonos una interfaz gráfica para poder ingresar los valores necesarios.

Antes de utilizar Postman, editemos el atributo **$except** en la clase **app/Http/Middleware/VerifyCsrfToken.php** para que se vea de la siguiente manera:

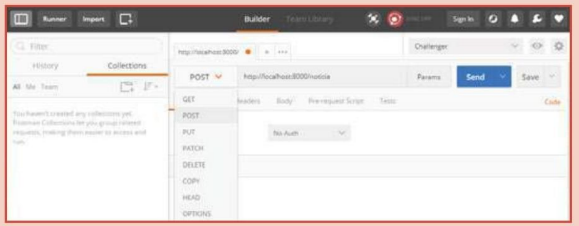
```
protected $except = [
    '/noticia*',
];
```

Luego de ejecutar Postman, hablaremos sobre esta última modificación que acabamos de realizar.

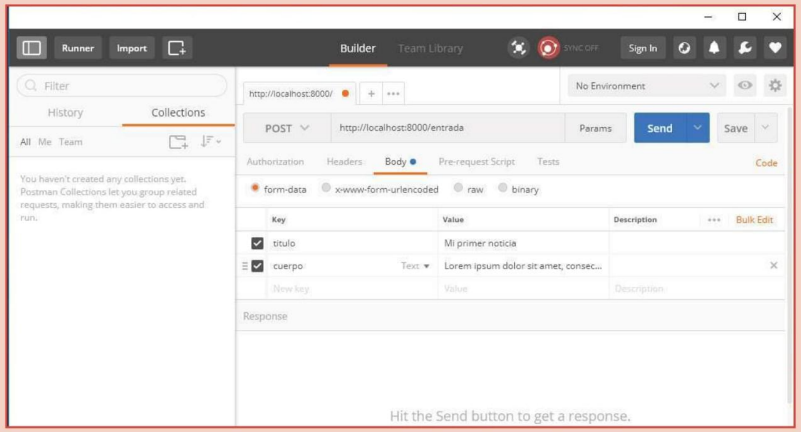
El siguiente procedimiento nos muestra cómo probar las rutas que acabamos de crear utilizando Postman.

EJECUTAR MÉTODOS POST UTILIZANDO POSTMAN

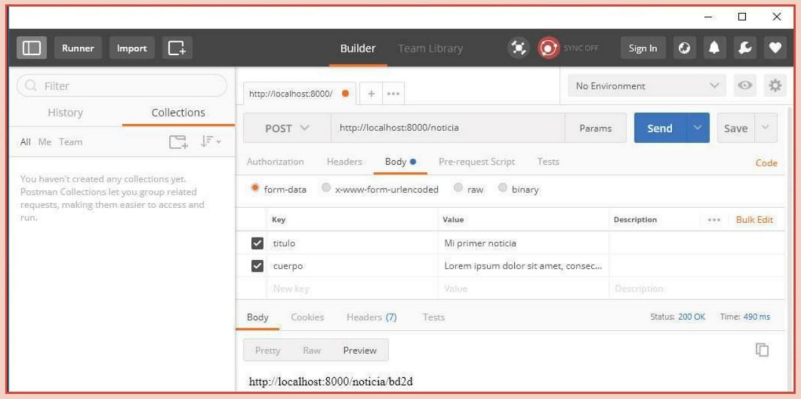
01 Ingrese en Postman la URL **http://localhost:8000/entrada** y presione en **GET** para obtener un listado completo de los verbos http. Seleccione en el listado el método **POST**.



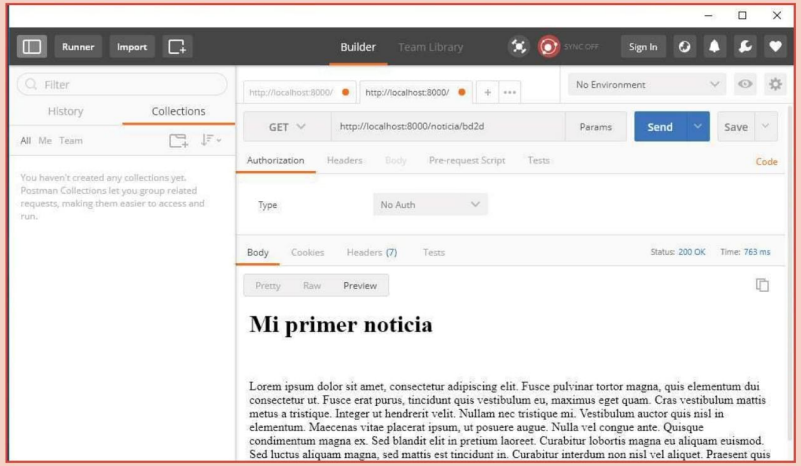
02 A continuación, seleccione la pestaña **Body** y luego **form-data**. En las columnas **keys** introduzca los nombres de las variables, y en las columnas **value**, los valores.



03 Presione el botón **Send** y luego la pestaña **preview** que aparece debajo. En este punto ya ha ejecutado la petición **POST**. Ejecute ahora el método **GET** para obtener la noticia.



04 Copie la URL brindada en la respuesta del paso anterior, abra una nueva pestaña en Postman y pegue la URL. Ejecute otra vez el botón **Send** y diríjase nuevamente a **Preview** para ver la respuesta en formato HTML.



Como podemos observar, Postman nos permite ver las respuestas en un formato lindo (**Pretty**), Crudo (**Raw**) y con Previsualización (**Preview**). También nos permite almacenar **requests** complejos para reutilizar o compartir con otros programadores e incluso, mediante code, podemos generar la petición en otros lenguajes de programación.

Sesiones

Si copiamos la URL generada en Postman y la pegamos en Chrome, no visualizaremos nada. Esto se debe a que estamos guardando los datos en la sesión, y recordemos que las sesiones se establecen por cliente. En este caso, Postman y Chrome actúan como clientes diferentes de nuestro servidor web, por lo tanto, tienen sesiones distintas. Más info en **http://php.net/manual/es/book.session.php**.

Protección CSRF

La protección **CSRF**, del inglés Cross Site Request Forgery, es decir, falsificación de peticiones cruzadas entre sitios web, es una práctica maliciosa que consiste en realizar peticiones no autorizadas a un sitio web.

Laravel incorpora un mecanismo de protección para estos ataques que consiste en generar un **hash**, es decir, una cadena de caracteres encriptada, para cada sesión, por lo cual, ante cada pedido al servidor web, se envía siempre este valor.

Si se realiza una petición sin este valor, el cual se almacena como **_token** en la sesión, o si el **_token** no coincide con el de la sesión del navegador, Laravel arrojará una excepción.

Figura 3. Si abrimos en Debugbar la pestaña **Sessions**, veremos el parámetro **_token** y el valor establecido para nuestra sesión.

Este mecanismo viene activado para todas las peticiones registradas en las rutas del archivo **routes/web.php**. Cuando modificamos la clase **app/Http/Middleware/VerifyCsrfToken.php** en la sección anterior, lo que hicimos fue desactivar este mecanismo para todas las rutas que comiencen con **/noticia**.

Si eliminamos este valor, lo cual sería lo correcto en un contexto productivo, las peticiones que hagamos mediante Postman fallarán porque no estaremos enviando el valor **_token**.

¿Dónde quedó el código hermoso?

Como vimos en el **Capítulo 1**, Laravel ama el código hermoso, y en este capítulo incluimos varios ejemplos que están lejos de apreciarse de esa manera.

Los frameworks, además de darnos estructura y orden, nos invitan a seguir buenas prácticas de programación. Recordemos que el principio del patrón MVC era **dividir** los diferentes tipos de lógicas, y en este

capítulo hemos **mezclado** la entrada, el procesamiento y la salida de nuestro sistema en una única función, e **integrado** dos lenguajes diferentes de programación (**PHP** y **HTML**), todo esto en un **único archivo routes.php**. A partir de esto podemos concluir que:

- El solo hecho de usar un framework no es indicador de hacer uso de las buenas prácticas de programación.
- Encerrar toda la lógica en un único lugar hace que el código sea menos legible.
- Al ser menos legible el código y estar toda la lógica mezclada, es más difícil rastrear los errores en nuestro sistema.
- Tener toda la lógica concentrada hace más difícil el trabajo en grupo, ya que varios programadores tendrán que modificar un mismo archivo.

Por eso, para que nuestro proyecto no se nos vaya de las manos debemos **controlarlo** y la manera de hacerlo es utilizar **Controladores**, que veremos en el próximo capítulo.

Resumen Capítulo 03

En este capítulo conocimos las rutas y vimos su vínculo con el protocolo HTTP. También analizamos la manera en la que se reciben las peticiones Request, cómo se devuelven las respuestas Responses, y la manera de apoyarnos en las Herramientas del desarrollador de Chrome. Luego fuimos agregando complejidad a nuestras rutas implementando parámetros, haciéndolos optativos y, más adelante, agregando restricciones. Por último, vimos el modo de probar métodos **POST** utilizando Postman, y aprendimos el concepto de CSRF junto con la manera en que Laravel atiende este problema.

ACTIVIDADES

Test de Autoevaluación

1. ¿Qué es una ruta?
2. ¿Cuáles son los métodos HTTP?
3. ¿Qué representan los objetos **$request** y **$response**?
4. ¿De qué manera podemos ver los encabezados de una petición y la respuesta a un servicio web?
5. ¿Qué es un parámetro de una ruta?
6. ¿En qué consiste que un parámetro sea opcional? ¿Qué cambios son necesarios para convertir un parámetro obligatorio en uno opcional?
7. ¿Cómo se pueden restringir los parámetros?
8. ¿Para qué sirve la aplicación Postman?
9. ¿En qué consiste la protección CSRF?
10. ¿En qué casos debemos desactivar la protección?

Ejercicios prácticos

1. Cree todas las rutas necesarias para poder listar, editar y eliminar una noticia. Considere todos los verbos HTTP disponibles y establezca nombres para cada ruta.
2. Identifique los parámetros en común generados en las rutas del paso anterior y cree restricciones para ellos. Considere si deben ser globales o no.
3. Realice peticiones mediante Postman para crear todas las rutas. Tenga en cuenta agregar las excepciones correspondientes a la protección CSRF.

04 Controladores

En este capítulo comenzaremos a organizar mejor nuestro código. En primer lugar, repasaremos el concepto de namespaces, veremos su relación con los controladores y exploraremos algunas ideas para agruparlos. También analizaremos el vínculo que tienen con el servicio de Rutas de Laravel.

CONTROLADORES EN LARAVEL

En el **Capítulo 3** trabajamos la lógica en el archivo **routes.php** y pudimos darnos cuenta de que si seguimos por este camino, nuestro sistema se saldrá de control. Para manejarlo usamos **controladores**.

Los controladores son una **interfaz lógica** que recibe peticiones, interactúa con los componentes del sistema para procesar la petición y luego devuelve una respuesta. Recordemos que las peticiones se representan en objetos **Request**, y las respuestas, en objetos **Response**. En la **Figura 1** vemos un controlador **SaludarController.php** generado a través del comando de Artisan **php artisan make:controller**.

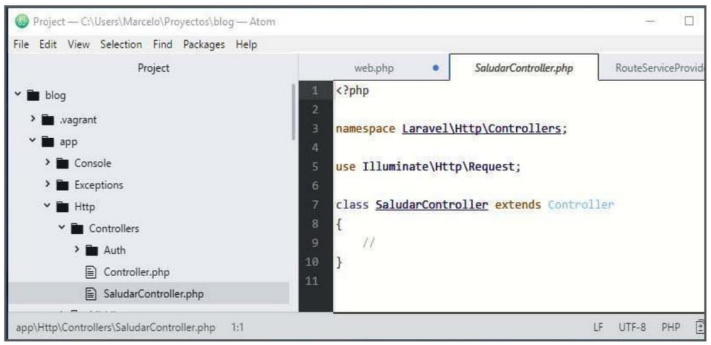


Figura 1. Los controladores deben terminar con **Controller** al final y se almacenan en la carpeta **app/Http/Controllers**.

La primera línea define el **espacio de nombres** o **namespace** al cual pertenece el controlador. En este caso, el primer valor es Laravel, ya que éste es el nombre que se le asigna por default a una aplicación creada con el framework. Lo correcto sería que nuestra aplicación se llamara Blog, por lo tanto, ejecutaremos el comando **php artisan app:name Blog**.

Si abrimos nuevamente el archivo **SaludarController.php**, veremos que nuestro namespace ha cambiado. Observemos también que todos los controladores deben heredar de la clase **Controller** y se inician declarando que se utiliza la clase **Illuminate\Http\Request**.

Vamos a migrar la lógica que creamos en el capítulo anterior en la ruta **decir.hola** al controlador **SaludarController**. Para hacerlo, crearemos una función en nuestro controlador de la siguiente manera:

```
public function decirHola($persona = 'humano'){
   return "hola " . $persona . "<br /> <a href='"
   .route('decir.bienvenido', ["blogger" => $persona]).
   "'>Decir Bienvenido</a>";

}
```

A continuación, quitamos esta lógica del archivo **routes.php** y asociamos la ruta con este método del controlador. Para lograrlo, cambiamos el callback: en lugar de introducir una función, ingresaremos entre comillas el nombre del controlador, luego el carácter @ y por último el nombre del método del controlador. Deberá quedar de la siguiente forma:

```
Route::get('/hola/{persona?}', 'SaludarController@
decirHola')->name('decir.hola');
```

Si accedemos por el navegador a esta ruta, vamos a encontrar que nada ha variado. El cambio más significativo está en nuestra lógica, ya que implementando controladores, podemos mejorar la legibilidad, el orden y la organización de nuestras rutas.

Para apreciar mejor este punto implementemos tres controladores: **SaludarController.php**, **HomeController.php** y **NoticiaController.php**. Nuestro archivo **routes/web.php** quedará de la siguiente manera:

Illuminate y namespaces

Vamos a encontrar que muchos componentes de Laravel pertenecen al namespace **Illuminate**, debido a que ése es el nombre clave de la versión 4 del framework. Los namespaces se incorporaron en la versión 5.3 de PHP, y el mejor lugar para consultar más información sobre el tema es en la documentación oficial de PHP, la cual podemos encontrar en español en **http://php.net/manual/es/language.namespaces.php**.

```
  Route::redirect('/', '/home', 301);
  Route::get('/hola-mundo', 'SaludarController@holaMundo');
  Route::get('/hola-json', 'SaludarController@holaJson');
  Route::get('/home', 'HomeController@index');
  Route::redirect('/', '/home');
  Route::get('/hola/{persona?}', 'SaludarController@
decirHola')->name('decir.hola');
  Route::get('/bienvenido/{blogger?}', 'HomeController@
bienvenido')->name('decir.bienvenido');
  Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
  Route::get('/noticia/{id}', 'NoticiaController@show')-
>name('noticia.show');
```

Ahora podemos notar una mejora significativa; sin embargo, todavía podemos optimizarlo aún más.

Agrupar controladores

Como ya sabemos, el comando **php artisan make:controller** genera archivos en la carpeta **app/Http/Controllers**, pero tal vez nos sea conveniente agrupar los controladores en subcarpetas conforme a nuestras necesidades.

Para nuestro blog, en principio vamos a crear tres subsistemas:

backend	Es donde ingresarán los administradores del sitio web para crear y administrar noticias.
frontend	Es donde ingresarán los visitantes del sitio web a ver las noticias.
services	Será nuestra interfaz de programación de aplicaciones, es decir, la API, que podrá ser consumida para desarrollar, por ejemplo, una aplicación móvil destinada a ver noticias.

Una de las entidades principales de nuestro blog será **Noticia**. Si repasamos la descripción de cada subsistema, podemos ver que todos trabajarán con noticias; por lo tanto, vamos a crear un **NoticiaController.php** en cada subcarpeta.

En esta oportunidad no utilizaremos el comando de Artisan, sino que simplemente copiaremos y pegaremos en cada subcarpeta el archivo **NoticiaController.php** que tenemos en **app/Http/Controllers**. La estructura debería quedar como muestra la **Figura 2**.

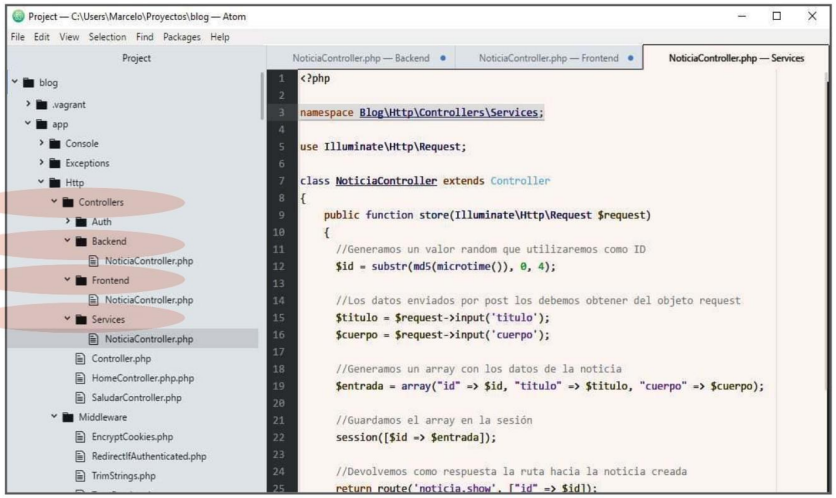


Figura 2. Muchas aplicaciones modernas están compuestas de, al menos, backend, frontend y servicios web.

Dado que **NoticiaController** hereda de la clase **Controller.php** que se encuentra en **app/Http/Controllers/Controller.php**, será necesario realizar los mismos pasos con este archivo, es decir, copiarlos en cada subcarpeta y establecer su namespace conforme al subsistema al que pertenecen.

Programación orientada a objetos

En este punto podemos notar la importancia de la programación orientada a objetos en el framework. Es fundamental para poder entender el framework y aprovechar al máximo sus ventajas, tener un buen conocimiento de este paradigma y conocer la manera en la que se implementa en PHP. Por eso, es recomendable consultar el enlace **http://php.net/manual/es/language.oop5.php**.

Al hacer esto, ya tenemos un espacio estratégico para implementar soluciones transversales a los controladores de cada subsistema gracias a la herencia de clases.

Namespaces

Ahora que tenemos tres **NoticiasController.php**, ¿cómo haremos para distinguir cada uno de ellos? Es en este punto donde los namespaces empiezan a cobrar más sentido. Será necesario establecer en el namespace de cada controlador el subsistema al cual pertenecen, y para hacerlo, vamos a modificar los archivos de la siguiente manera:

app/Http/Controllers/Backend/NoticiaController.php

```
namespace Blog\Http\Controllers\Backend;
```

app/Http/Controllers/Backend/NoticiaController.php

```
namespace Blog\Http\Controllers\Frontend;
```

app/Http/Controllers/Backend/NoticiaController.php

```
namespace Blog\Http\Controllers\Services;
```

Rutas

Dado que ahora tenemos tres **NoticiasController.php**, si intentamos acceder por Postman, obtendremos una excepción del tipo **ReflectionException**.

Si queremos que esta ruta funcione correctamente, debemos indicar en ella el namespace del controlador al cual queremos referirnos; en este caso será el Backend:

routes/web.php

```
Route::post('/noticia', 'Backend\NoticiaController@
store')->name('noticia.store');
```

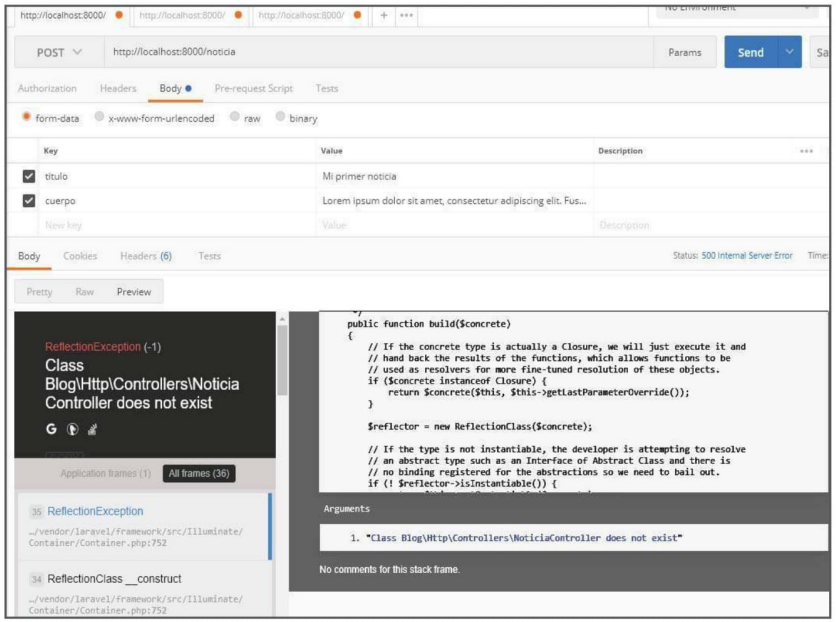



Figura 3. La aplicación busca la clase **Blog\Http\Controllers\NoticiaController**, la cual ya no existe más.

Otra manera de hacer esto es agrupando las rutas e indicando el namespace en el que operarán las rutas del grupo. Modifiquemos el archivo **routes/web.php**:

```
Route::namespace('Backend')->group(function () {
    // Controlladores dentro del namespace "App\Http\Con-
trollers\Backend"
    // Eliminamos \Backend ya que lo hemos indicado en el
grupo
    Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
    Route::get('/noticia/{id}', 'NoticiaController@
show')->name('noticia.show');
});
```

```
  Route::namespace('Frontend')->group(function () {
      // Controlladores dentro del namespace "App\Http\Con-
trollers\Frontend"
      Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
      Route::get('/noticia/{id}', 'NoticiaController@
show')->name('noticia.show');
  });
```

Ahora tenemos dos grupos de rutas; sin embargo, los nombres de cada ruta no nos permiten distinguir fácilmente si pertenecen al backend o al frontend.

Sería bueno adoptar la siguiente nomenclatura para definir los nombres de las rutas: **subsistema.controlador.metodo**. Para no tener que modificar cada una, podemos aprovechar las ventajas de tenerlas agrupadas y asignar un nombre al grupo, el cual se concatenará luego al nombre de cada ruta del grupo.

```
  Route::namespace('Backend')->name('backend.')-
>group(function () {
      Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
      Route::get('/noticia/{id}', 'NoticiaController@
show')->name('noticia.show');
  });
  Route::namespace('Frontend')->name('frontend.')-
>group(function () {
      Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
      Route::get('/noticia/{id}', 'NoticiaController@
show')->name('noticia.show');
  });
```

Si intentamos realizar la petición, obtendremos un error como el que indica la **Figura 4**.

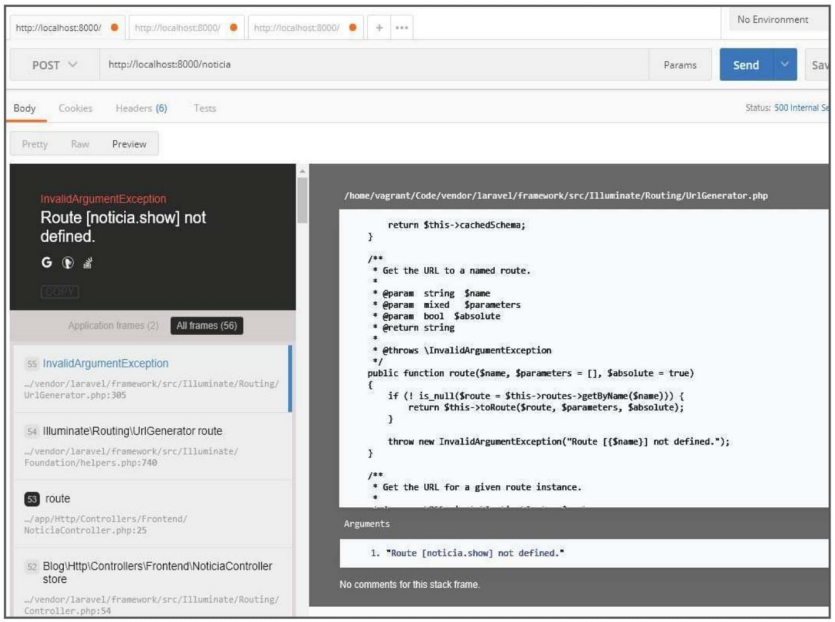


Figura 4. En este punto, la ruta **noticia.show** no existe más; en cambio, sí tenemos **backend.noticia.show** y **frontend.noticia.show**.

Modifiquemos los métodos **store** de los controladores de backend y frontend para que cada uno asigne correctamente la ruta del subsistema al que pertenece:

Blog\Http\Controllers\Backend\NoticiaController.php@store deberá utilizar:

```
return route(backend.noticia.show', ["id" => $id]);
```

Blog\Http\Controllers\Frontend\NoticiaController.php@store deberá utilizar:

```
return route('frontend.noticia.show', ["id" => $id]);
```

Con estos cambios parecería ser que todo funciona de maravillas. Sin embargo, por más que las rutas se llamen de manera diferente,

están utilizando la misma URL y, en consecuencia, la que aparece segunda, es decir la de frontend, se encuentra inaccesible.

Podemos usar URLs diferentes en cada ruta o, al igual que hicimos con los nombres, podemos aprovechar las ventajas de que las rutas se encuentren agrupadas. Por lo tanto, vamos a utilizar la función **prefix** en la generación de grupos para agregar un nuevo segmento en todas las rutas de backend, de manera que el grupo se definirá así:

```
  Route::namespace('Backend')->name('backend.')->prefix('/
backend')->group(function () {
      Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
      Route::get('/noticia/{id}', 'NoticiaController@
show')->name('noticia.show');
  });
```

De esta manera, todas las rutas del grupo backend deberán antecederse ingresando **/backend**. En consecuencia, si queremos llamar desde Postman a este método deberemos, en primera instancia, exceptuar esta nueva ruta de la protección CSRF:

Blog\Http\Middleware\VerifyCsrfToken

```
protected $except = [
        '/noticia*',
        '/backend/noticia*'
    ];
```

Depende de nosotros

Si bien el framework nos da elementos para organizarnos, en un punto la organización pasa a depender de nosotros. Vamos a organizar los controladores por subsistema, dado que cada sistema atiende un grupo diferente de usuarios; de esta manera, podremos implementar soluciones transversales entre subsistemas. No obstante, cada proyecto es un mundo y depende de nosotros definir el criterio para cada caso.

La URL que se utilizará en Postman será **http://localhost:8000/backend/noticia**.

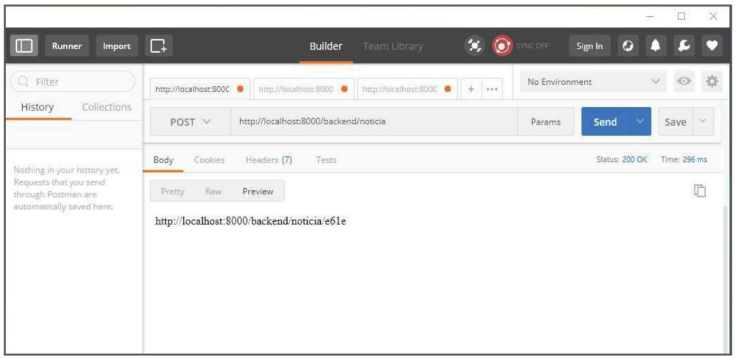


Figura 5. Laravel detectó que la ruta **backend.noticia.show** pertenece al grupo **backend** y, por lo tanto, ya agrega el prefijo **/backend**.

Servicios web

Hasta este punto hemos visto cómo organizarnos con el subsistema backend y frontend, pero no vimos nada acerca de los servicios.

En la carpeta **routes** encontraremos un archivo especial para establecer las rutas de los servicios web, denominado **api.php**. Las rutas definidas en este archivo tienen algunas particularidades. Veamos qué tiene preparado el servicio de rutas de Laravel en **app/Providers/RouteServiceProvider.php** y analicemos el método **mapApiRoutes**:

```
    protected function mapApiRoutes()
    {
        Route::prefix('api')
             ->middleware('api')
             ->namespace($this->namespace)
             ->group(base_path('routes/api.php'));
    }
```

Podemos ver que el servicio hace lo siguiente:

- Establece el prefijo api.
- Asigna el middleware api (veremos más al respecto en el capítulo dedicado a los Middlewares).
- Asigna un namespace.
- Agrupa todas las rutas del archivo **routes/api.php**.

Debido a que decidimos crear un namespace particular para los servicios, vamos a modificar sólo esa línea:

```
    ->namespace($this->namespace . '\Services')
```

Como las rutas ya tienen un prefijo y un namespace asignado, solamente agruparemos las rutas de nuestra API para asignar nombres:

```
  Route::name('services.')->group(function () {
      Route::post('/noticia', 'NoticiaController@store')->name('noticia.store');
      Route::get('/noticia/{id}', 'NoticiaController@show')->name('noticia.show');
  });
```

Si queremos invocar el método por Postman, deberemos hacerlo a **http://localhost:8000/api/noticia**. Tengamos en cuenta dos aspectos importantes:

1 Para que funcione correctamente, debemos actualizar la ruta que se genera en la salida del método **store** y en **app/Http/Controlles/Services/NoticiaController.php**. En este caso no fue necesario agregar la excepción a la protección CSRF. Esto se debe a que estas rutas son **stateless**, es decir, sin estado: cada petición se atiende de manera independiente y no tienen, entre otras cosas, sesión ni protección CSRF.

2 Al no tener sesión, la URL devuelta en la respuesta no funcionará, y deberá ser necesario utilizar otro mecanismo de persistencia en estos casos, por ejemplo, una base de datos.

Resources

Recordemos la definición de URL expuesta en el **Capítulo 3**: es una cadena de caracteres que identifica los recursos de una red de manera unívoca. En las aplicaciones web es normal que estos recursos puedan transformarse y, como ya sabemos, disponemos de un protocolo con verbos que nos brindan un punto de partida.

Laravel se apoya en esta premisa para crear controladores del tipo **resource**, es decir, controladores que ya tienen creados los métodos necesarios para operar en la transformación de recursos. Creemos un controlador de recursos para las categorías de nuestras noticias en el backend con el siguiente comando: **php artisan make:controller CategoriaController --resource**.

Este comando, además de generar el controlador, también armará, a través de un template, una clase con todos los métodos y sus firmas, como se muestra en la **Figura 6**.

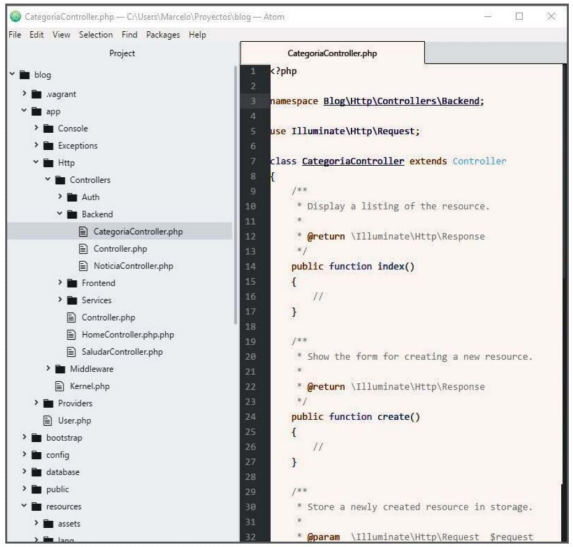


Figura 6. Recordemos que Artisan genera los controladores en **app/Http/Controllers**. Debemos moverlo a **backend** y actualizar el **namespace**.

Los métodos que genera un controlador del tipo **resource** son los necesarios para construir lo que se conoce como **ABM**. El término proviene de Altas, Bajas y Modificaciones, y hace referencia a las operaciones básicas para administrar un recurso en un sistema web. Se suelen omitir en la sigla, aunque no en los sistemas, los listados, los cuales se consideran como fundamentales por ser el punto de acceso principal a los recursos.

En inglés, el término se conoce como CRUD, por Create, Read, Update y Delete, que significa crear, leer, actualizar y borrar, respectivamente.

Además de haber creado el controlador del **resource**, debemos crear una ruta del tipo **resource**:

```
Route::namespace('Backend')->name('backend.')->prefix('/
backend')->group(function () {
      Route::post('/noticia', 'NoticiaController@store')-
>name('noticia.store');
      Route::get('/noticia/{id}', 'NoticiaController@
show')->name('noticia.show');

      Route::resource('categorias', 'CategoriaController');
  });
```

Habiendo realizado estos cambios, las rutas y los métodos del controlador quedarán relacionados de la siguiente forma:

Tener en cuenta Lumen

En este proyecto vamos a elaborar tres subsistemas, de modo que nos conviene utilizar Laravel porque generaremos lógica transversal a la aplicación que reutilizaremos en cada subsistema. Sin embargo, una tendencia actual es crear solamente servicios web que luego serán consumidos por frameworks JavaScript, como AngularJS. En esos casos, debemos considerar el uso de Lumen, ya que está pensado especialmente para generar APIs.

MÉTODOS Y RUTAS DE UN RESOURCE

Verbo	URL	Método	Ruta
GET	Categorias	index	categorias.index
POST	categorías/create	create	categorias.create
POST	Categorias	store	categorias.store
GET	categorias/{categoria}	show	categorias.show
GET	categorias/{categoria}/edit	edit	categorias.edit
PUT PATCH	categorias/{categoria}	update	categorias.update
DELETE	categorias/{categoria}	destroy	categorias.destroy

Tabla 1. Recordemos que, al estar en el grupo backend, las URLs llevarán el prefijo establecido para el grupo. Lo mismo sucederá con los nombres de las rutas.

Dado que las rutas de noticias que creamos siguen esta nomenclatura, podemos reemplazarlas por un resource:

```
Route::namespace('Backend')->name('backend.')->prefix('/
backend')->group(function () {
      Route::resource('noticia', 'NoticiaController');
      Route::resource('categorias', 'CategoriaController');
  });
```

Resumen Capítulo 04

En este capítulo conocimos los controladores, creamos namespaces para los subsistemas de nuestra aplicación y armamos grupos de controladores a partir de ellos. También vimos el vínculo entre los controladores y las rutas, así como las ventajas de agrupar rutas e implementar funcionalidades transversales en cada grupo. Por último, observamos las particularidades de las rutas para las APIs y los controladores que intervienen en ellas. Gracias a esto podemos empezar a organizar más nuestro código, lo cual nos permitirá luego analizarlo y depurarlo de una mejor forma.

ACTIVIDADES

Test de Autoevaluación

1. ¿Qué es un controlador?
2. ¿Para qué sirven los namespaces en los controladores?
3. ¿Por qué cuando creamos un controlador se genera la línea **use Illuminate\Http\Request;**?
4. ¿Cómo se vincula un controlador a una ruta?
5. ¿Qué son los prefijos?
6. ¿Qué sucede con los nombres de las rutas de un grupo que también tiene su propio nombre?
7. ¿En qué se diferencian las rutas definidas en **routes/web.php** de las de **routes/api.php**?
8. ¿Por qué no es necesario agregar la excepción a la protección CSRF para los controladores que responden en las rutas declaradas en **routes/api.php**?
9. ¿Qué significa que una petición sea **stateless**?
10. ¿Qué cambios es necesario hacer en el método store del controlador **Services\NoticiaController.php** para que guarde información?

Ejercicios prácticos

1. Identifique los métodos que utilizaría en los subsistemas backend, frontend y services.
2. Cree las rutas y los controladores necesarios para los métodos del punto anterior.
3. Separe las rutas de backend y frontend en dos archivos diferentes utilizando la misma estrategia de organización que las APIs.
4. Modifique las respuestas de los servicios web de manera tal que respondan content-types application/json.
5. Genere una collection de Postman para invocar a los servicios del paso anterior.

05 Vistas

En este capítulo comenzaremos con la construcción de interfaces para nuestro blog, a partir de las vistas de Laravel. Analizaremos las principales características de Blade, el poderoso motor de plantillas que trae Laravel. Veremos cómo construir mocks con datos más parecidos a un contenido real y estudiaremos el vínculo entre las estructuras de Blade y las del lenguaje PHP.

¿QUÉ SON LAS VISTAS?

PHP es uno de los primeros lenguajes de programación del lado del servidor que se podía incorporar directamente en el documento HTML en lugar de llamar a un archivo externo que procesara los datos. En el año 1994 esto era una excelente idea; tengamos en cuenta que HTML no tenía las mismas características que tiene ahora.

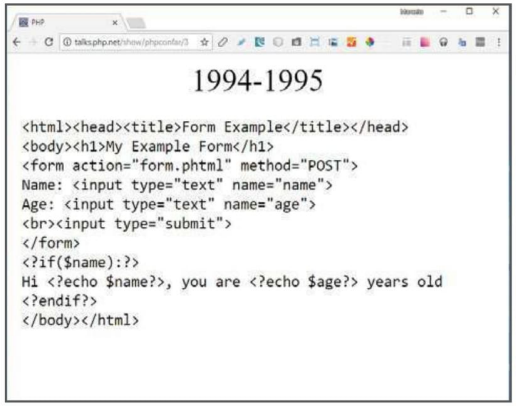


Figura 1. El lanzamiento de HTML fue en 1991. Las páginas web eran mucho más simples que las que vemos hoy en día.

En la actualidad, se considera una muy mala práctica combinar código PHP y HTML. La mayoría de los frameworks modernos que vimos en el **Capítulo 1** separan esta lógica y dedican la parte de código HTML en archivos denominados **vistas**.

Historia de PHP

Entender la historia del lenguaje nos permite comprender algunas prácticas que se utilizaron en los inicios y que se mantienen hoy en día. Es importante tener en cuenta que el lenguaje evolucionó, y con él, las prácticas de desarrollo, y si queremos mantenernos vigentes en el mercado, nosotros también debemos evolucionar como programadores. La historia de PHP está disponible en http://php.net/manual/es/history.php.php.

Hoy las vistas han evolucionado mucho debido a que HTML también lo ha hecho, y es por eso que también se las conoce como lógica de presentación o **Web UI**, del inglés User Interface, es decir, interfaz de usuario.

Las vistas ya dejaron de ser etiquetas que brindan un formato. Hoy en día, gracias a los avances de JavaScript es posible construir interfaces muy ricas en cuanto a funcionalidades y formatos.

Blade

Para poder mantener separada la lógica de PHP de la lógica de presentación, los frameworks modernos utilizan sistemas de plantillas. Las plantillas son archivos que no introducen lógica PHP y se apoyan en etiquetas y/o formatos particulares para establecer estructuras de lógica.

En Laravel, el sistema de plantillas utilizado es **Blade** y es por esto que todas las vistas terminan en la extensión **.blade.php**.

Blade presenta como ventaja que todas las plantillas se compilan y se almacenan en caché, lo cual hace que nuestra aplicación sea más rápida al no tener que interpretar las plantillas en tiempo de ejecución.

Al enfrentar un proyecto de desarrollo es importante considerar que la división de lógicas es clave para llevar a cabo el trabajo en equipo de forma correcta.

Como vimos en el **Capítulo 2**, Laravel organiza todo lo necesario para construir las interfaces de usuario en la carpeta **resources**, y las vistas se encuentran en la carpeta **views**. Las vistas pueden retornarse en una función callback desde una ruta o devolverse desde un controlador, siendo esta última opción la más adecuada.

Dado que decidimos crear subsistemas para **backend** y para **frontend**, para seguir manteniendo el mismo orden debemos crear subcarpetas en **resources/views**.

Es recomendable también agrupar las vistas conforme a los controladores que las utilizarán. Por lo tanto, crearemos una carpeta **noticia** en cada subsistema. Vamos a empezar creando las vistas para mostrar una noticia; la estructura deberá quedar como muestra la **Figura 2**.

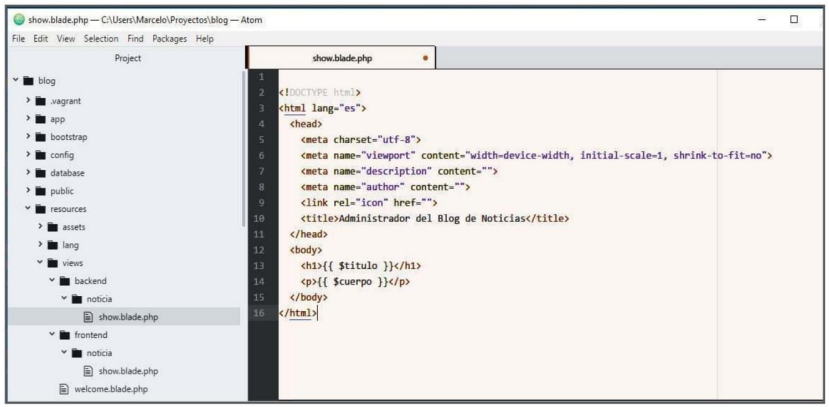


Figura 2. Es recomendable asociar el nombre del archivo de la vista al nombre del método del controlador que la utilizará.

Pasar información a la vista

En el código HTML de la **Figura 2** podemos apreciar unas etiquetas especiales:

```
<h1>{{ $titulo }}</h1>
<p>{{ $cuerpo }}</p>
```

Estas etiquetas que inician con dos caracteres de llave abierta {{ y cierran con dos caracteres de llave cerrada }} imprimen el contenido de las variables **$titulo** y **$cuerpo**. La asignación de valores a estas variables debe realizarse en otro lugar, generalmente en un

Sistemas de plantillas

Al igual que muchos componentes del framework, hay diversos sistemas de vistas que pueden implementarse en proyectos de manera independiente al framework. Además de Blade, otro sistema muy conocido es **Twig**, el cual utiliza el framework Symfony. Fuera del ambiente de los frameworks, uno de los sistemas pioneros de este tipo es el conocido **Smarty**.

controlador, dado que no es responsabilidad de la vista ir a buscar y/o procesar información, sino mostrar la información de una determinada manera.

Vamos a retornar esta vista y asignarle valores desde el controlador **app/Http/Controllers/backend/NoticiaController.php** en el método **show**:

```
      public function show($id){
        $titulo = "Titulo de la noticia";
        $cuerpo = "Cuerpo de la noticia";
        return view('backend.noticia.show', ['titulo' =>
$titulo, 'cuerpo' => $cuerpo]);
      }
```

La función **view** recibe un primer parámetro con la ubicación del archivo de la vista en la carpeta **resources/views** y usa como separador de carpetas el punto. Al llamar a la vista no es necesario agregar la extensión **.blade.php**. El segundo parámetro sólo es necesario si la vista contiene variables y es un array asociativo donde los nombres de los índices deben coincidir con los nombres de las variables del template.

El proceso mediante el cual se compilan los datos de las vistas y se devuelve el HTML final es conocido como **renderizar**, término que significa retratar. La vista también puede recibir objetos:

```
      <h1>{{ $noticia->titulo }}</h1>
      <p>{{ $noticia->cuerpo }}</p>
```

En el controlador debemos pasar el objeto como una variable más:

```
      public function show($id){
        $noticia = (object) array(
          'titulo' => 'Titulo de la noticia',
          'cuerpo' => 'Cuerpo de la noticia',
          'id' => $id);
        return view('backend.noticia.show', ['noticia' =>
$noticia]);
      }
```

Herencia en las vistas

Identificar patrones en nuestro código nos permitirá ahorrar mucho tiempo. Si creamos la vista de **Blog\Http\Controllers\Backend\NoticiaController@store**, podemos apreciar que el **<head></head>** será el mismo que utilizamos para el método **show**. Para no tener que copiar este elemento en todas las vistas, podemos crear una que luego sea heredada por otras; a esta vista la denominaremos **Layout**.

En la herencia en Blade, lo que ocurre es que la vista **hija** está contenida dentro de la vista **padre**, es decir, dentro del layout.

Vamos a crear el layout para el backend generando un archivo en **resources/views/backends/layouts/main.blade.php**, de la siguiente forma:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="">
    <title>Administrador del Blog - @yield('title')</
title>
  </head>
  <body>
    @yield('content')
  </body>
</html>
```

Mocks

Los mocks son objetos simulados cuyos datos y comportamiento imitan un determinado contexto de una forma controlada. Sirven para analizar, trabajar o testear una parte de un sistema sin querer introducirse en otra. En este caso nos interesa estudiar las vistas; por lo tanto, ignoraremos la manera en la que se obtendrá la información del objeto **Noticia.**

Una característica de los layouts son las directivas **yield**, las cuales consisten en definir un espacio que será utilizado por otra vista. Los yield son el punto de contacto entre las vistas padre y las vistas hijas.

Modifiquemos la vista **show.blade.php**:

```
@extends('backend.layouts.main')

@section('title', 'Noticia: ' . $noticia->titulo)

@section('content')
  <h1>{{ $noticia->titulo }}</h1>
  <p>{{ $noticia->cuerpo }}</p>
@endsection
```

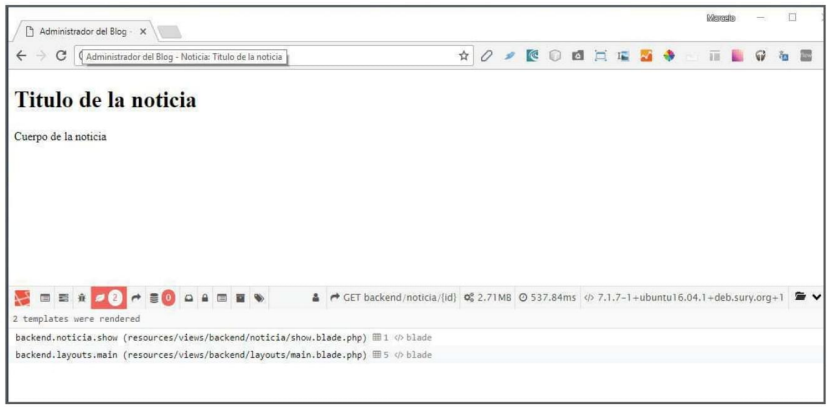


Figura 3. Debugbar nos permite ver todas las vistas que se renderizan en una página y las variables de cada una.

A partir de la directiva **extends** establecemos que la vista utilizará como layout el definido en **backend.layouts.main**. A partir de la directiva **section** establecemos el contenido que será insertado en el espacio definido previamente por la directiva **yield** en el layout.

Podemos decir que el controlador esta "mockeando" una noticia.

También es posible iniciar una sección en una vista y luego continuarla en otra, de manera tal que los aspectos generales queden cubiertos por el layout, y los particulares, por la vista que va a determinar el contexto de la página que estemos visualizando en ese momento. Veamos esto con un ejemplo; modifiquemos el body del layout:

```
<body>
 <ul>
   <li><b>Menu Principal</b></li>
   <li><a href="#">Noticias</a></li>
   <li><a href="#">Usuarios</a></li>
 </ul>
 <ul>
   @section('menu-contextual')
     <li><b>Menu Contextual</b></li>
   @show
 </ul>
 @yield('content')
</body>
```

Observemos que definimos una sección **menu-contextual** que hemos finalizado con **@show** en vez de con **@endsection**. Esto es porque la sección continuará en la vista de la siguiente forma:

```
@extends('backend.layouts.main')

@section('title', 'Noticia: ' . $noticia->titulo)
```

Layout

El término layout proviene del inglés y tiene varias traducciones, aunque para nuestro caso la que mejor se adapta es **distribución**, un término utilizado ampliamente en desarrollo web. Esto se debe a que por lo general la responsabilidad de un layout consiste en definir las estructuras de una página. Sin embargo, el contenido será establecido luego por otras vistas, componentes o slots.


```
  @section('menu-contextual')
    @parent
    <li><a href="{{ route('frontend.noticia.show', ["id" =>
$noticia->id]) }}">Ver en frontend</a></li>
    <li><a href="#">Editar</a></li>
    <li><a href="#">Eliminar</a></li>
  @endsection

  @section('content')
    <h1>{{ $noticia->titulo }}</h1>
    <p>{{ $noticia->cuerpo }}</p>
  @endsection
```

Mediante **@parent** indicamos que el contenido de esta sección es una continuación a la establecida en el layout.

Una forma de ver cómo se une el código es agregando comentarios HTML al inicio y cierre de cada sección. De esta manera, si abrimos esta pantalla en el navegador y visualizamos el código HTML resultante, veremos lo que se muestra en la **Figura 4**.

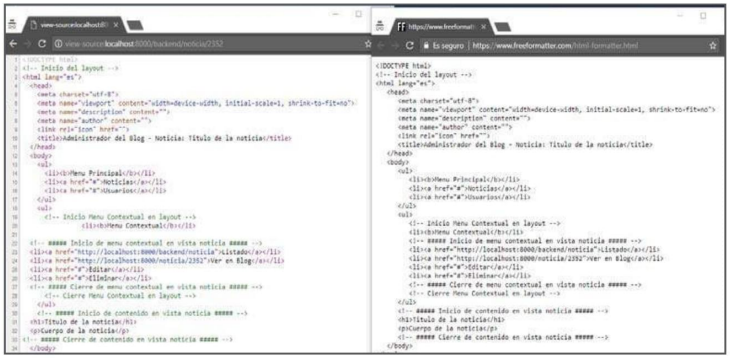


Figura 4. Las dos ventanas muestran el mismo código HTML. La de la derecha tiene el código formateado, para mayor legibilidad.

Siguiendo esta estrategia, nuestros menús contextuales para el recurso **noticia** quedarían como se observa en la **Figura 5**.

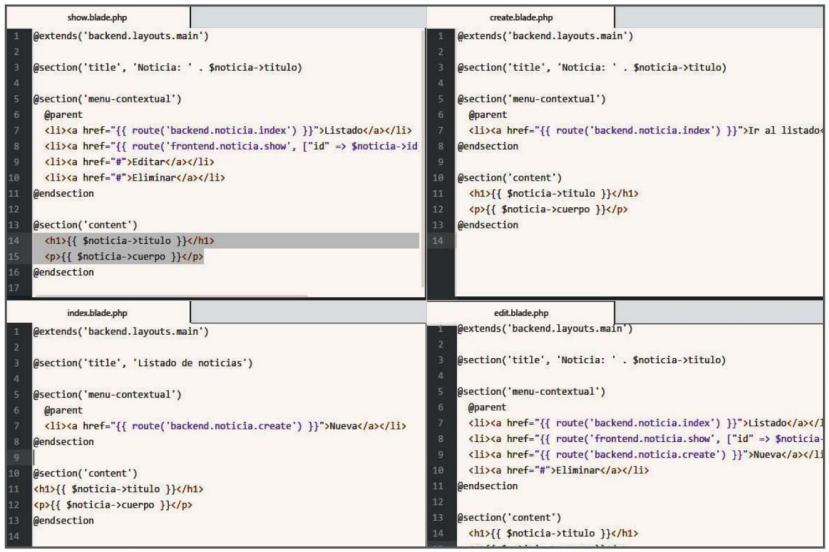


Figura 5. El contenido de un menú contextual debe variar dependiendo de la visualización que se esté presentando en pantalla.

Estructuras en las vistas

En el **Capítulo 4** establecimos que los listados son un punto fundamental en los ABM. Vamos a crear un mock en nuestro controlador de noticias para empezar a trabajar con los listados en las vistas. En esta oportunidad emplearemos un mock un poco más sofisticado utilizando Faker.

Faker

Faker es una librería que genera datos a partir de determinadas características dadas. Es muy fácil de instalar mediante Composer a través del comando **composer require fzaninotto/faker**, aunque a partir de la versión 5.1 ya forma parte del framework.

Para mantener nuestro código ordenado, vamos a crear una clase para este propósito, la cual ubicaremos en **app/Factories/NoticiaFactory.php** con el siguiente código:


```php
<?php

namespace Blog\Factories;
use Faker;

class NoticiaFactory
{

  public static function generarNoticias($cantidad){
    $mocks = array();
    for($i = 0; $i < $cantidad; $i++){
      $mocks[] = NoticiaFactory::hacerNoticia();
    }
    return $mocks;
  }

  public static function hacerNoticia(){
    $faker = Faker\Factory::create();
    return (object) [
      'id' => $faker->randomNumber(4),
      'titulo' => $faker->sentence,
      'cuerpo' => $faker->paragraphs(3, true),
      'imagen' => $faker->imageUrl(80, 80),
      'autor' => $faker->name,
    ];
  }
}
```

Factories

Las clases **Factory** suelen referenciar a aquellas que se relacionan directamente con un modelo y permiten generar registros para una base de datos; se encuentran en la carpeta **/database/factories/**. En este caso, no estamos utilizando un modelo, y por eso usamos una ubicación diferente. No obstante, bautizamos la clase como **Factory** puesto que, conceptualmente, hace lo mismo, sólo que persiste en memoria en lugar de en una base de datos.

El método **hacerNoticia** crea objetos con datos obtenidos de Faker, para lo cual utiliza diferentes métodos que provee la librería con el fin de generar distintos datos. Es posible obtener un listado completo en **https://github.com/fzaninotto/Faker**.

Bucles

Vamos a utilizar nuestra Factory en el método **index** del controlador **Backend\NoticiasController.php** para enviar un listado de noticias a la vista:

```
  public function index(){
    $listado = NoticiaFactory::generarNoticias(20);
    return view('backend.noticia.index', ['listado' =>
$listado]);
  }
```

Recordemos agregar la instrucción **use Blog\Factories\NoticiaFactory;** debajo de la definición del **namespace**.

Es importante tener en cuenta que con esta modificación, la variable **$noticia** ya no existe, de manera que debemos modificar el **section title** para que funcione.

Si modificamos la vista insertando una etiqueta **{{ $noticias }}** en la **section content**, obtendremos un error del tipo **htmlspecialchars() expects parameter 1 to be string, array given**. Como ya sabemos, cuando en Blade encerramos una variable entre llaves, nos imprime el contenido de ésta, y para hacerlo, utiliza la función de PHP **htmlspecialchars()**, la cual espera recibir un valor del tipo string. Sin embargo, nosotros estamos pasando un array. Éste es un ejemplo claro de las operaciones que

Utilizar bootstrap

Sabemos que el formato HTML crudo puede ser algo desagradable de ver. Por suerte existe una forma simple y rápida de mejorar esto, gracias a la implementación de Bootstrap, el cual puede conseguirse en forma gratuita visitando la dirección web **http://getbootstrap.com**. Más adelante, en esta obra, profundizaremos sobre las distintas maneras que podemos utilizar para incluir **assets** para las vistas.

Blade hace al momento de compilar y, debido a que las vistas que se visualizan están en caché, vemos un archivo diferente al que acabamos de escribir en el error.

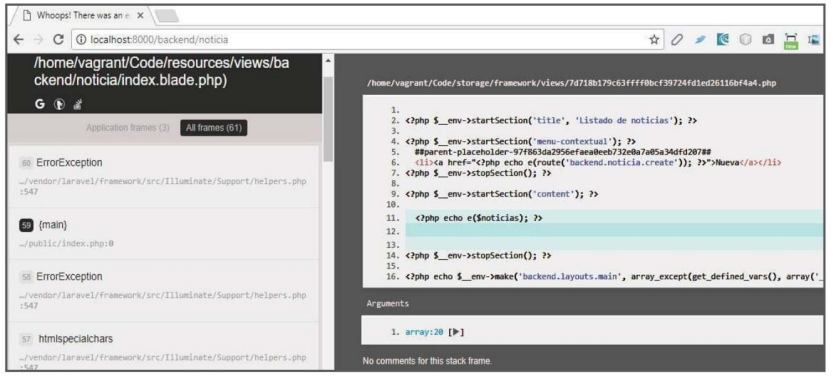


Figura 6. Así es como se ve una vista almacenada en caché; esto ahorra tiempo de procesamiento en la ejecución de las peticiones.

Blade nos proporciona etiquetas que permiten trabajar de una manera clara y limpia las estructuras básicas de programación; los bucles son una de ellas.

Modifiquemos la vista **backend.noticia.index** e implementemos algunas de estas etiquetas en la **section content**:

```
@section('content')
  @foreach ($noticias as $noticia)
    <p><b>Titulo:</b> {{ $noticia->titulo }}</p>
  @endforeach
@endsection
```

En este caso hemos utilizado la estructura **@foreach**, la cual es igual a la que ya conocemos de PHP. Sin embargo, mediante esta etiqueta hemos evitado tener que abrir **<?php** y cerrar en cada línea **/>**. Por lo tanto, resulta más cómodo para programar y, además, el código es más legible.

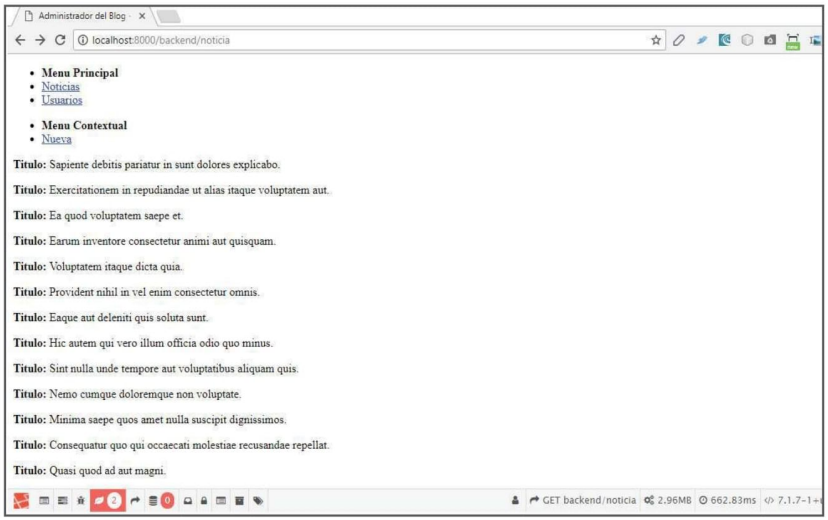


Figura 7. Debugbar presenta las vistas que se están mostrando en esta pantalla.

Además, Blade ofrece algunas variaciones y agregados en los bucles que pueden ser muy útiles:

```
@section('content')
  @forelse ($noticias as $noticia)
      <p><b>Titulo:</b> {{ $noticia->titulo }}</p>
  @empty
      <p>No hay noticias para mostrar</p>
  @endforelse
@endsection
```

@forelse introduce una estructura **@empty** que podemos utilizar cuando esperamos recibir elementos de un array y está vacío. Para probarla, pasemos un array vacío en **Backend\NoticiasController.php**:

```
    public function index(){
      $noticias = array();
```



```
        return view('backend.noticia.index', ['noticias' =>
$noticias]);
      }
```

Otro agregado funcional muy importante cuando trabajamos con bucles es la variable **$loop**, la cual está presente en todas las iteraciones y nos permite obtener información importante sobre la iteración. La **Tabla 1** nos muestra sus propiedades:

PROPIEDADES DE LA VARIABLE $LOOP

Propiedad	Descripción
$loop->index	Índice de la iteración actual del bucle. Inicia en 0.
$loop->iteration	Iteración actual del bucle. Inicia en 1.
$loop->remaining	Cantidad de iteraciones restantes del bucle.
$loop->count	Número total de elementos que serán iterados.
$loop->first	Indica si es la primera iteración del bucle.
$loop->last	Indica si es la última iteración del bucle.
$loop->depth	Indica el nivel de profundidad de la iteración actual cuando se anidan varios bucles.
$loop->parent	En un bucle anidado, es una referencia al bucle padre.

Tabla 1. Éste es un sólido ejemplo de las ventajas que tiene utilizar las estructuras de Blade en lugar de las de PHP.

Modifiquemos nuestra vista para utilizarla:

```
@section('content')
  @forelse ($noticias as $noticia)
    @if($loop->first)
      <h1>Mostrando {{ $loop->count }} noticias.</h1>
    @endif
```

```
    <p><b>Titulo:</b> {{ $noticia->titulo }}</p>
  @empty
      <p>No hay noticias para mostrar</p>
  @endforelse
@endsection
```

Observemos que, además de haber utilizado la variable **$loop**, también usamos la estructura **@if** y **@endif**, la cual es idéntica a su contraparte en PHP. En lo que respecta a los condicionales, también contamos con **@empty** y **@isset**.

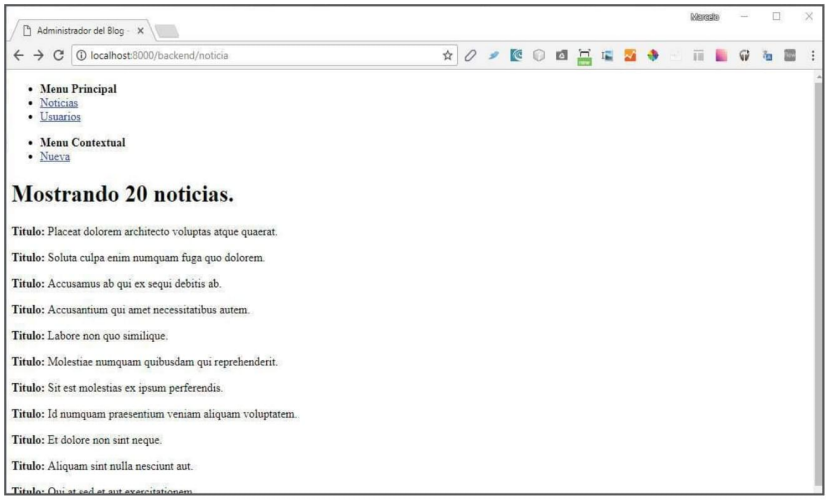


Figura 8. A través de la estructura condicional, establecemos el encabezado (h1) de la pantalla.

Subvistas

Otra manera de reutilizar código es insertando una vista en otra, lo cual podemos hacer a través de **@include**.

Por ejemplo, podemos separar la manera en la que queramos visualizar la noticia en una vista y, luego, incluirla en nuestro loop, de manera que el código quede más ordenado. Creemos la vista **resources/views/backend.noticia/ítem.blade.php**:


```
  <div>
    <h3>{{ $noticia->titulo }}</h3>
    <p>{{ str_limit($noticia->cuerpo, 100) }}</p>
    <a href="{{ route('backend.noticia.show', ['noticia' =>
$noticia->id]) }}">Ver</a>
  </div>
```

Observemos que estamos utilizando un **helper** de Laravel llamado **str_limit** para tomar sólo una determinada cantidad de caracteres, en este caso 100, para mostrar una parte del cuerpo de la noticia.

Existen muchos helpers disponibles para trabajar con diferentes elementos; podemos consultar el listado completo en **https://laravel.com/docs/5.5/helpers**. Modifiquemos **resources/views/backend.noticia/index.blade.php** para incluir la vista **backend.noticia.item**:

```
  @section('content')
    @forelse ($noticias as $noticia)
      @if($loop->first)
        <h1>Mostrando {{ $loop->count }} noticias.</h1>
      @endif
      @include('backend.noticia.item')
    @empty
        <p>No hay noticias para mostrar</p>
    @endforelse
  @endsection
```

Obtendremos un resultado similar simplificando el código de la siguiente forma:

```
  @section('content')
    @if(count($noticias) > 0)
      <h1>Mostrando {{ count($noticias) }} noticias.</h1>
    @endif
    @each('backend.noticia.item', $noticias, 'noticia',
'backend.noticia.empty')
  @endsection
```

La etiqueta **@each** nos permite realizar un **include** sobre cada elemento del array. El primer parámetro es la vista que se utilizará, el segundo es el array y el tercero, cómo se llamará cada elemento del array dentro de la subvista.

Es importante tener en cuenta que en las subvistas se heredan las propiedades de las vistas padre. No obstante, en el caso de **@each** esto no ocurre, por lo que no podemos acceder, por ejemplo, a la variable **$loop**. En esos casos, estamos obligados a utilizar un bucle.

El cuarto parámetro es opcional y es el nombre de la vista a utilizar cuando el array esté vacío. Lo crearemos con el siguiente contenido:

```
  <div>
    <p>No hay noticias para mostrar</p>
    <p>Para crear una noticia ingrese en <a href="{{
route('backend.noticia.create') }}">este enlace</a>.</p>
  </div>
```

Nuestro listado se verá como muestra la **Figura 9**.

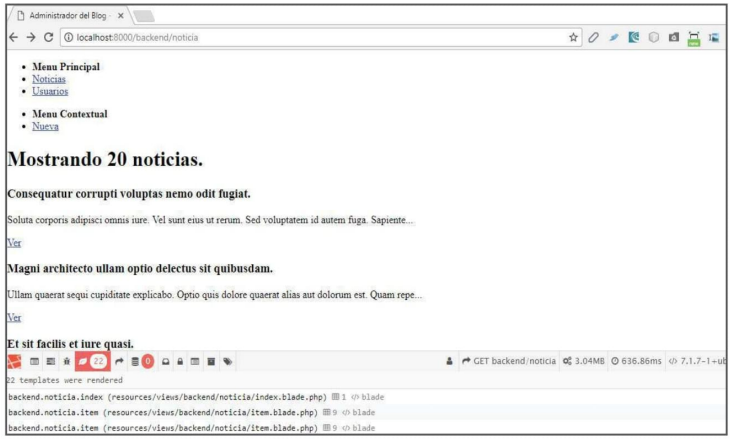


Figura 9. Debugbar muestra 22 vistas, **1 layout.blade.php**, **1 index.blade.php** y **20 item.blade.php**, es decir, uno por cada noticia.

Por último, generemos un error en la vista **index.blade.php** para poder ver la vista compilada de Blade. Para esto, basta con agregar **{{ $variableQueNoExista }}** al final de la vista.

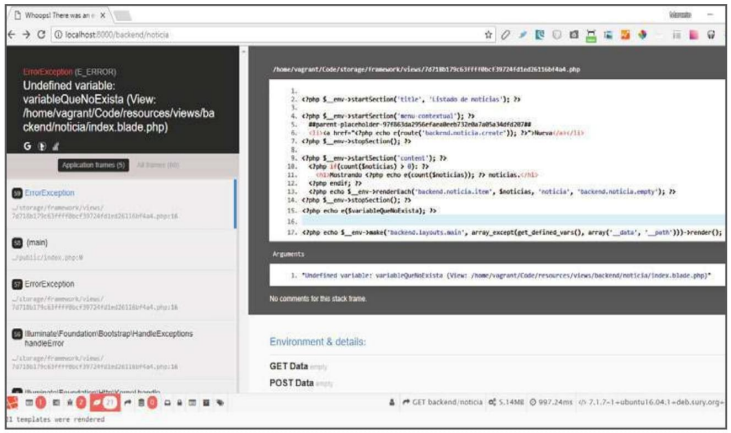


Figura 10. La vista compilada nos permite ver la relación directa que existe entre las estructuras de Blade y las del lenguaje PHP.

Resumen Capítulo 05

En este capítulo repasamos el concepto de vista e introdujimos el de motor de plantillas. Aprendimos a utilizar Blade, el motor que usa Laravel, y vimos diversas maneras de reutilizar el código de las vistas. También creamos mocks más avanzados que nos permitieron tener información más precisa para poder trabajar y, luego, analizamos las ventajas de utilizar las estructuras de Blade en vez de las de PHP. Por último, mejoramos todo lo realizado organizando el código en distintas subvistas.

ACTIVIDADES

Test de Autoevaluación

1. ¿Qué es una vista?
2. ¿Para qué sirve un motor de plantillas?
3. ¿Dónde se almacenan las vistas en Laravel?
4. ¿De qué manera se envía información a una vista?
5. ¿Para qué sirve la herencia en las vistas?
6. ¿Cómo se relacionan las directivas **yield** y **section**?
7. ¿Para qué sirve la librería Faker?
8. ¿Cuáles son las ventajas de utilizar las estructuras de Blade en lugar de las de PHP?
9. ¿Dónde se puede utilizar la variable **$loop**? ¿Qué información contiene?
10. ¿Qué es una subvista? ¿Mediante qué directiva es posible implementarlas?

Ejercicios prácticos

1. Cree un Layout para el frontend y todas las vistas necesarias para los métodos **index** y **show** del frontend.
2. Cree una Factory para las categorías.
3. Cree las vistas necesarias para todos los métodos del resource **noticia** y **categoria**.

Bases de datos

06

En este capítulo analizaremos la arquitectura que utiliza Laravel para trabajar con bases de datos. Crearemos la estructura para guardar las noticias de nuestro blog usando migraciones y estudiaremos las herramientas que provee el framework para modificar la estructura de la base y para crear datos. Por último, utilizaremos fábricas de objetos para generar registros en la base.

ARQUITECTURA

Los frameworks PHP modernos utilizan una capa de **abstracción** para conectarse a las bases de datos, lo cual significa que no hacen uso de las funciones específicas que proveen los drivers de determinados motores. Por ejemplo, para trabajar con MySQL contamos con un conjunto de funciones específicas que nos permiten interactuar con una base de datos de ese tipo; no obstante, y valga la redundancia, sólo sirve para bases MySQL.

A su vez, los frameworks suelen utilizar librerías que permiten generar clases a partir de las tablas de una base de datos. Se las conoce como **ORM**, del inglés *Object-Relational Mapping*, es decir, mapeo objeto-relacional.

La mayoría de los ORM utilizan como capa de abstracción la librería PHP PDO, **http://php.net/manual/es/class.pdo.php**. En síntesis, la arquitectura para operar con una base de datos suele estar definida como se muestra en la **Figura 1**.

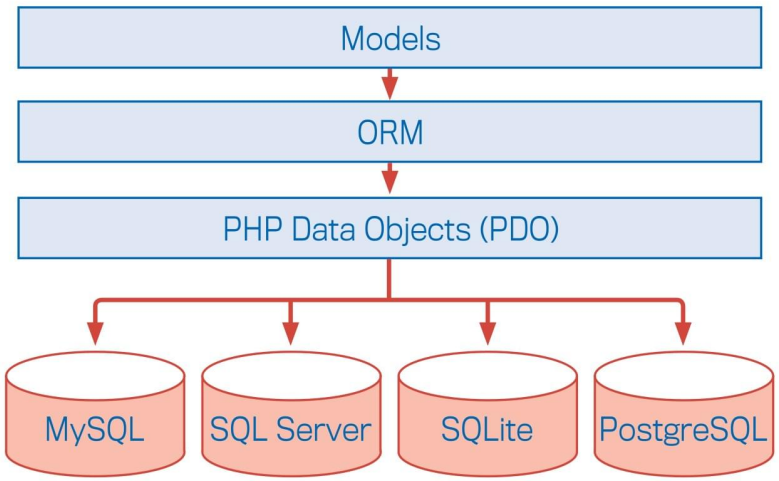


Figura 1. Éstos son los cuatro motores de base de datos que soporta el framework actualmente.

Configuración

Debido a la capacidad de trabajar con múltiples motores, lo primero que encontramos en la configuración de base de datos en **config/database.php** es la conexión por default a la base:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

En Laravel podemos tener conexiones tanto a un mismo motor como a diferentes motores; el único requisito es asignar un nombre a cada conexión y, a la vez, establecer la que utilizaremos por default.

Los nombres de las conexiones se almacenan en índices de un array denominado **$connections**, que se encuentra en el archivo de configuración. Veamos un fragmento de este código:

```
'connections' => [

        'sqlite' => [
            'driver' => 'sqlite',
            'database' => env('DB_DATABASE', database_
path('database.sqlite')),
            'prefix' => '',
        ],

        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],
```

En este segmento tenemos definidas dos conexiones, denominadas **sqlite** y **mysql**. El nombre debe ser declarativo de la fuente de datos que nos brindará esta conexión; por lo tanto, modifiquemos **mysql** por **blog** y establezcamos dicha conexión como default.

Lo que determina qué tipo de motor se estará utilizando en esa conexión es el parámetro **driver**.

Podemos ver que, para estas dos conexiones que utilizamos, en un caso es **sqlite** y en otro es **mysql**.

Una vez definido el **driver**, el resto de los parámetros pueden variar, debido a que si bien existe una abstracción para la operación de las bases, es posible que algunos motores demanden más parámetros de configuración que otros para poder conectarse.

Por último, tengamos en cuenta que es posible tanto definir los valores en estos parámetros como utilizar valores de las variables de entorno, tal como vimos en el **Capítulo 2** bajo el título DotEnv. Si utilizamos Homestead, dichos parámetros los encontraremos en el archivo **.env** y deberán verse de la siguiente manera:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

Recordemos que al iniciar Homestead se realiza un redireccionamiento de puertos, como se observa en la **Figura 2**.

Plural o singular

Es una buena práctica establecer la manera en la que se trabajará con los nombres de las diferentes entidades. Siguiendo la convención de Laravel, los nombres de las tablas se declaran en plural, y los de los modelos y los controladores, en singular. Esto puede parecer absurdo, pero tengamos en cuenta el tiempo que debemos estar consultando nuestros archivos para analizar cómo los declaramos a la hora de referenciarlos.

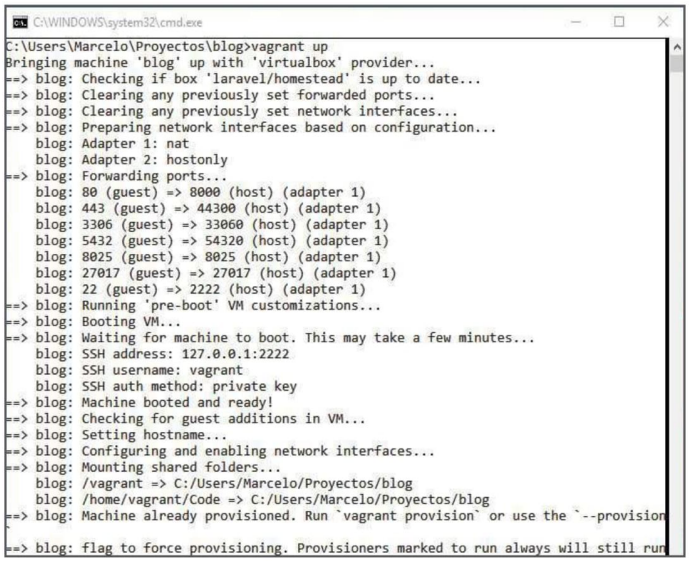


Figura 2. Entre los puertos que se redireccionan podemos ver el 3306, el cual es el que utiliza el motor MySQL por default para sus conexiones.

Por lo tanto, por medio de los parámetros del archivo **.env**, podemos usar cualquier herramienta de administración de base de datos para conectarnos a la base de datos que se encuentra en Homestead. Sólo debemos acordarnos de utilizar el puerto **33060** en vez del 3306.

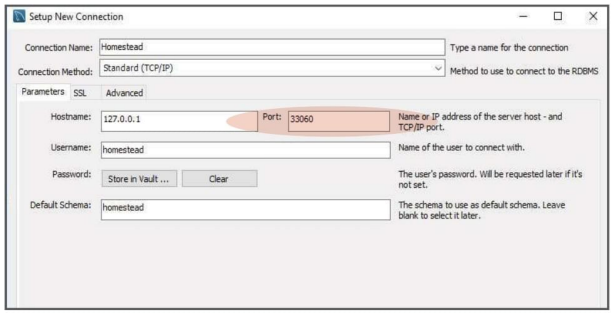


Figura 3. MySQL Workbench es una herramienta visual que permite administrar bases MySQL, disponible en **www.mysql.com/products/workbench**.

Si queremos evitar instalar un cliente en nuestra máquina física, también podemos conectarnos por ssh a través de la interfaz de línea de comando y, desde allí, utilizar el cliente MySQL de Homestead.

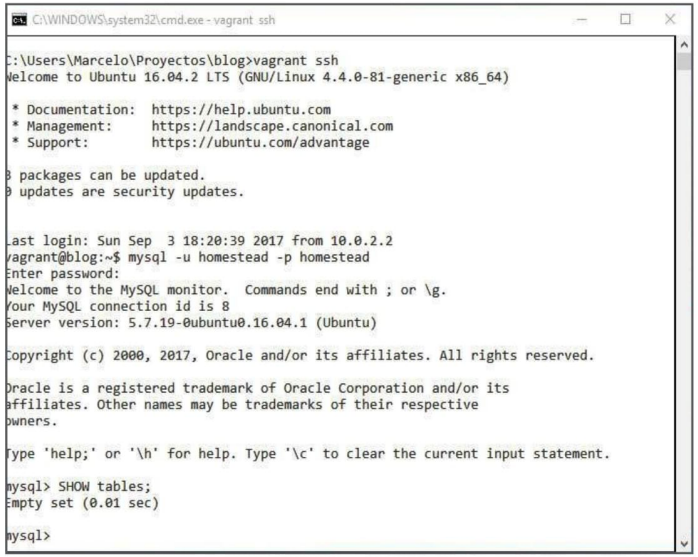


Figura 4. La sintaxis del comando es **mysql -u usuario -p basededatos**; al solicitar la clave debemos ingresar **secret**.

MIGRACIONES

Como sabemos, Laravel nos brinda organización y estructura, y las bases de datos no están exentas.

Cuando trabajamos en grupo, es muy común pasarse scripts de base de datos para armar la estructura, tanto al inicio como durante el desarrollo. Esto puede tornarse caótico en un proyecto donde intervengan varios programadores en la base de datos.

La forma que tiene Laravel de atacar este problema es a través de las migraciones, que permiten versionar una base de datos en archivos PHP e incorporarlos como parte del código fuente

del proyecto. De esta manera, se pueden compartir entre todos los programadores.

En síntesis, una migración es una herramienta que permite establecer estructuras de base de datos, a través de una lógica declarativa. Estos archivos generarán los scripts necesarios para producir y/o actualizar la estructura de la base.

Las migraciones se encuentran en la carpeta **database/migrations**. Si buscamos esa carpeta, veremos dos migraciones, que son las que permiten generar la estructura de datos para crear las tablas **users** y **password_resets**. Éstas forman parte del sistema de autenticación que trae el framework.

En un principio podemos pensar que utilizar migraciones es un trabajo que aparenta ser más laborioso, pero con el tiempo nos daremos cuenta de que versionar la base hará más simple el trabajo en equipo.

Al observar las migraciones que ya están creadas, notaremos que ambas tienen dos métodos, **up** y **down**. El método **up** es el que se lee al momento de ejecutar una migración para realizar cambios en la base, mientras que el método **down** es aquel que se lee al momento de revertir los cambios producidos en la ejecución.

Decimos que una migración se **ejecuta** cuando se lee el archivo de la migración, se generan los scripts **sql** y se impactan los cambios en la base de datos.

Antes de ejecutar la migración, ejecutamos el comando **php artisan migrate:install**, el cual genera una tabla en la base de datos que se utiliza para mantener un registro de las migraciones ejecutadas en esa base.

Comandos para las migraciones

En Artisan todos los comandos se agrupan por namespaces. Gracias a ello es posible obtener un listado completo de todas las operaciones que se pueden realizar en las migraciones ejecutando el comando **php artisan**, y buscar el namespace **migrate**. Todos los comandos disponen de un parámetro **–help**, que nos brinda una descripción completa de lo que realiza, y los parámetros mandatorios y opcionales.

```
mysql> quit
Bye
vagrant@blog:~/Code$ php artisan migrate:install
Migration table created successfully.
vagrant@blog:~/Code$ mysql -u homestead -p homestead
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
+--------------------+
| Tables_in_homestead |
+--------------------+
| migrations         |
+--------------------+
1 row in set (0.00 sec)

mysql>
```

Figura 5. El nombre de esta tabla se define en el archivo de configuración de base de datos, en **config/database.php**.

Podemos consultar dicho registro a partir del comando **php artisan migrate:status**. Para ejecutar las migraciones pendientes, ejecutemos **php artisan migrate**.

```
C:\WINDOWS\system32\cmd.exe - vagrant ssh
vagrant@blog:~/Code$ php artisan migrate:status
+------+-----------------------------------------------------+
| Ran? | Migration                                           |
+------+-----------------------------------------------------+
| N    | 2014_10_12_000000_create_users_table                |
| N    | 2014_10_12_100000_create_password_resets_table      |
+------+-----------------------------------------------------+
vagrant@blog:~/Code$ php artisan migrate
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
vagrant@blog:~/Code$ php artisan migrate:status
+------+-----------------------------------------------------+
| Ran? | Migration                                           |
+------+-----------------------------------------------------+
| Y    | 2014_10_12_000000_create_users_table                |
| Y    | 2014_10_12_100000_create_password_resets_table      |
+------+-----------------------------------------------------+
vagrant@blog:~/Code$
```

Figura 6. El comando **migrate:status** lee los archivos de la carpeta **database/migrations** y busca en la tabla **migrations** si existen o no.

Esquemas

Comencemos a crear una migración para la estructura donde almacenaremos la información de las noticias de nuestro blog. Para hacerlo, vamos a ejecutar el comando **php artisan make:migration create_noticias_table**.

```
C:\WINDOWS\system32\cmd.exe - vagrant ssh
vagrant@blog:~/Code$ php artisan make:migration create_noticias_table
Created Migration: 2017_09_03_191128_create_noticias_table
vagrant@blog:~/Code$ cat database/migrations/2017_09_03_191128_create_noticias_table.
php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateNoticiasTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('noticias', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('noticias');
    }
}
vagrant@blog:~/Code$
```

Figura 7. El parámetro **create_noticias_table** del comando define el nombre que le daremos a la migración.

En la **Figura 7** podemos ver el archivo que acabamos de generar, y notar que todos comienzan con una fecha y hora. Esto le permite al framework saber el orden en el que se deben ejecutar.

Mediante el comando **php artisan migrate --pretend**, vamos a ver el script que se ejecutará en la base sin realizar los cambios. Ejecutémoslo y analicemos el método **up** de la clase **CreateNoticiasTable** de nuestra migración:

```
public function up()
{
```

```
        Schema::create('noticias', function (Blueprint
$table) {
            $table->increments('id');
            $table->timestamps();
        });
    }
```

Observemos que este script generará una tabla con tres campos, lo cual se logra a través del servicio **Schema**. Éste es un servicio que permite manipular el esquema de la base de datos. En nuestra migración, el método **create** recibe dos parámetros: el primero es el nombre de la tabla que creará, y el segundo, la función **callback** que se ejecutará para completarla.

En la función **callback** recibimos un objeto **Blueprint**, el cual permite crear columnas en una tabla. El método **increments** generará un campo autoincremental y lo establecerá como clave primaria. El método **timestamps** generará dos campos, **created_at** y **updated_at**, en los cuales almacenaremos las fechas en las que se creen y actualicen noticias.

Antes de ejecutar la migración definitiva, agreguemos algunos campos más:

```
    public function up()
    {
        Schema::create('noticias', function (Blueprint
$table) {
            $table->increments('id');
```

Métodos Blueprint

Existen muchos métodos en esta clase debido a que están relacionados a los tipos de datos que se pueden almacenar en una base de datos. También hay métodos para establecer modificadores, cambiar columnas existentes e, incluso, eliminar columnas. El mejor lugar para consultar todos los métodos es la documentación oficial de Laravel, la cual se encuentra disponible en inglés en **https://laravel.com/docs/5.5/migrations#columns.**


```
            $table->string('titulo', 255);
            $table->text('cuerpo');
            $table->string('imagen', 255)->nullable();
            $table->timestamps();
        });
    }
```

Ahora sí, ejecutemos nuestra nueva migración con **php artisan migrate**.

Relaciones

Como ya sabemos, Laravel provee un componente para operar con usuarios. Vinculemos a los autores de las noticias con los usuarios del sistema generando relaciones entre las tablas.

Podemos crear una nueva migración en la cual, en vez de utilizar el método **create**, utilizaremos **table** en **Schema**; o mejor aún, podemos hacer un rollback, es decir, deshacer los cambios de la última migración, modificarla y ejecutarla otra vez.

Para hacerlo, primero ejecutemos **php artisan migrate:rollback** y, luego, modifiquemos la migración de la siguiente manera:

```
    public function up()
    {
        Schema::create('noticias', function (Blueprint
$table) {
            $table->increments('id');
            $table->string('titulo', 255);
            $table->text('cuerpo');
            $table->string('imagen', 255)->nullable();
            $table->timestamps();
            $table->unique('titulo');
            $table->integer('autor')->unsigned();
            $table->foreign('autor')
              ->references('id')
              ->on('users')
              ->onDelete('cascade')
```

```
                ->onUpdate('cascade');
        });
    }
```

Observemos que agregamos un índice único para los títulos de las noticias y, luego, agregamos un campo **autor**, el cual debemos definir de la misma manera que se definió el campo **id** en la tabla **users**. Por último, creamos la clave mediante el método **foreign**, y en el método **on** establecimos la tabla con la que relacionaremos ese campo. Con los modificadores **onDelete** y **onUpdate** establecemos la opción deseada para cuando se eliminen o actualicen los registros de la tabla **users**.

Si hacemos ingeniería inversa con alguna herramienta de administración de base de datos, obtendremos el siguiente diagrama de entidad-relación:

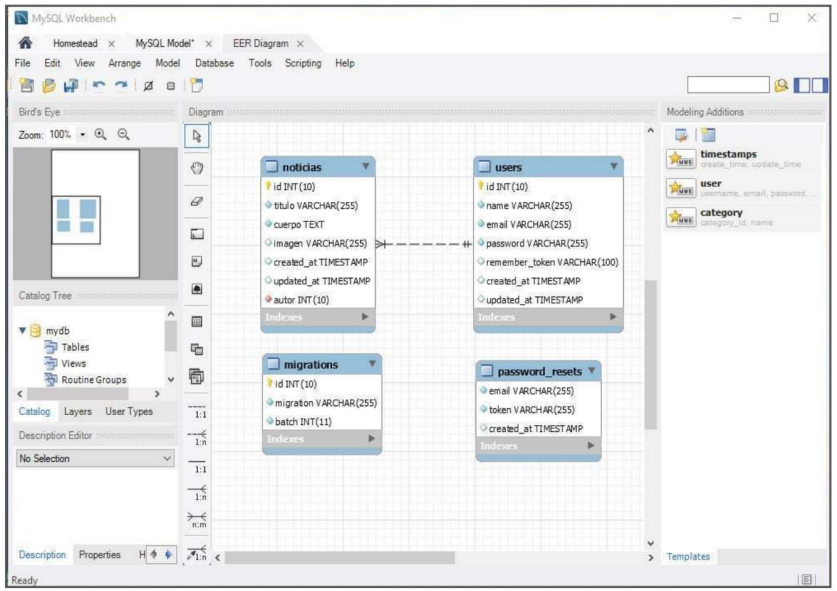


Figura 8. En el diagrama es posible visualizar de manera gráfica la relación que acabamos de crear.

SEEDERS

Ahora que tenemos una estructura para nuestra base estamos en condiciones de empezar a trabajar con los datos, y es aquí donde intervienen los seeders.

Los **seeders** son clases que nos permiten crear registros en las tablas de nuestra base. Su nombre proviene del inglés seed, que significa sembrar. En Laravel los encontramos en la carpeta **database/seeders**.

El archivo **DatabaseSeeder.php** es el único que se ejecuta mediante el comando **php artisan db:seed**. No obstante, este seeder lo único que hace es invocar a otros seeders; en sí mismo no inserta registros en una base, salvo que agreguemos lógica en su método **run**.

Creemos el seeder para agregar usuarios mediante el comando **php artisan make:seed UsersTableSeeder**. El parámetro **UsersTableSeeder** es el nombre que asignaremos a la clase que se creará.

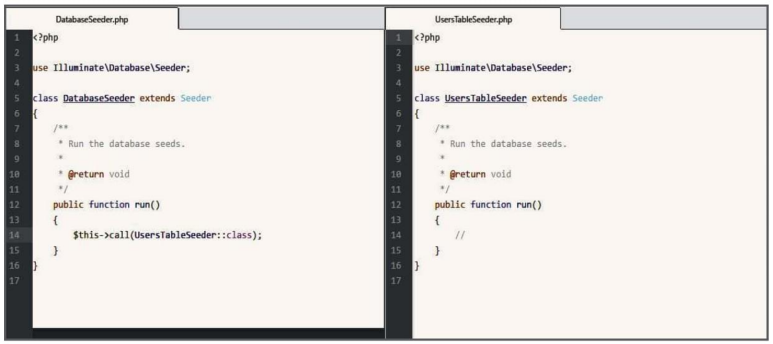


Figura 9. El método **run** de **DatabaseSeeder** ya contiene un llamado a **UsersTableSeeder**, aunque el mismo está comentado.

Ingeniería inversa

La ingeniería inversa consiste en obtener un diseño a partir de un producto dado. En nuestro caso, el producto es la base de datos, y el diseño es el diagrama de entidad-relación que obtenemos a partir del análisis que hacemos, o mejor dicho, que MySQL Workbench hace sobre la base. Es un término muy utilizado en ingeniería de software y aplicable en muchos ámbitos.

En este punto, si ejecutamos **php artisan db:seed** veremos que **UsersTableSeeder** se ha ejecutado de manera exitosa. Sin embargo, el método **run**, que es donde debemos introducir la lógica, se encuentra vacío. Vamos a modificarlo de la siguiente manera:

```
public function run()
{
  DB::table('users')->insert([
       'name' => 'marcelo',
       'email' => 'marcelo@email.com',
       'password' => bcrypt('secret'),
  ]);
}
```

Ejecutar consultas

DB es un servicio que nos permite ejecutar consultas en la base de datos. Dispone de diversos métodos para realizar diferentes operaciones. Además de **insert**, que es el que acabamos de implementar, también contamos con **select**, **update** y **delete**. Todos los métodos que escriben en la base devuelven como resultado la cantidad de registros afectados por la consulta.

También disponemos de **transaction**, **beginTransaction** y **rollback**, los cuales son imprescindibles para trabajar con transacciones de base de datos. Podemos encontrar ejemplos de todos estos métodos en la documentación oficial de Laravel disponible, en inglés, en la dirección **https://laravel.com/docs/5.5/database**.

Si ejecutamos nuevamente el comando **php artisan db:seed**, obtendremos el mismo resultado que la vez anterior, pero si ingresamos en la base de datos y listamos los registros de la tabla **users**, notaremos que hay un nuevo registro con los datos que creamos en el seeder.

Es posible utilizar muchas técnicas para "sembrar" la base de datos, pero cuanto más varíen los datos, mejor serán las pruebas que realicemos, y éste es un punto importante, sobre todo, en las interfaces de usuario.
En consecuencia, obtendremos un sistema más robusto.

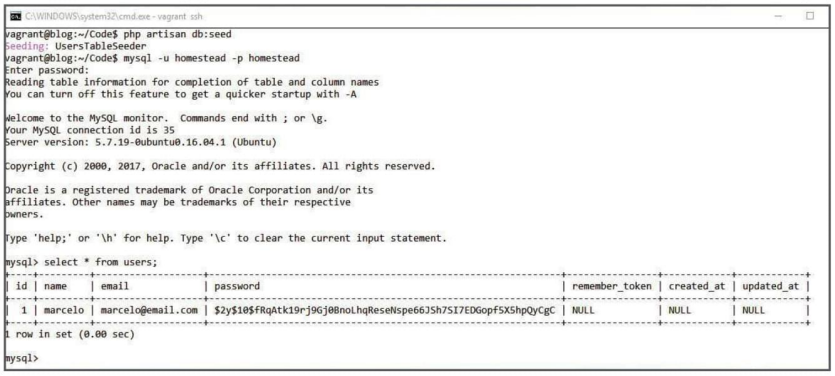


Figura 10. El campo **password** aparece encriptado debido a que utilizamos el helper **bcrypt**; observemos que los campos **timestamps** están todavía vacíos.

Podemos ejecutar otra vez **php artisan db:seed** y, en esta oportunidad, detectaremos un error.

Es importante tener en cuenta que los seeders deben respetar la estructura definida en la base de datos; esto significa que los campos deben llamarse igual y tener la misma longitud, entre otras cosas. En nuestro caso en particular, el error se produce porque la columna **email** está definida con un **índice único**, lo que hace que no puedan existir dos registros en la base de datos con el mismo valor.

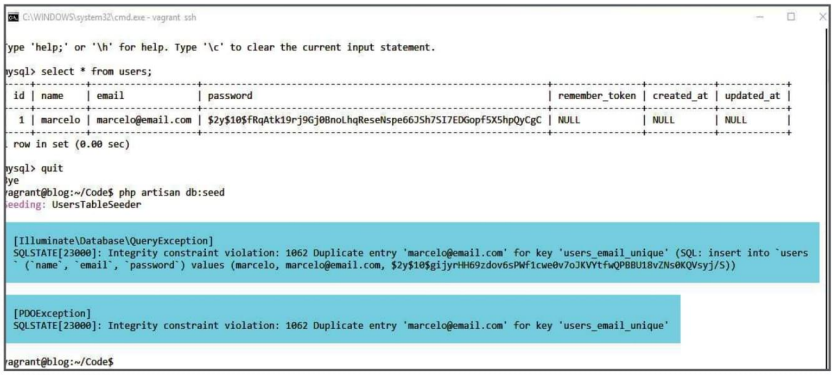
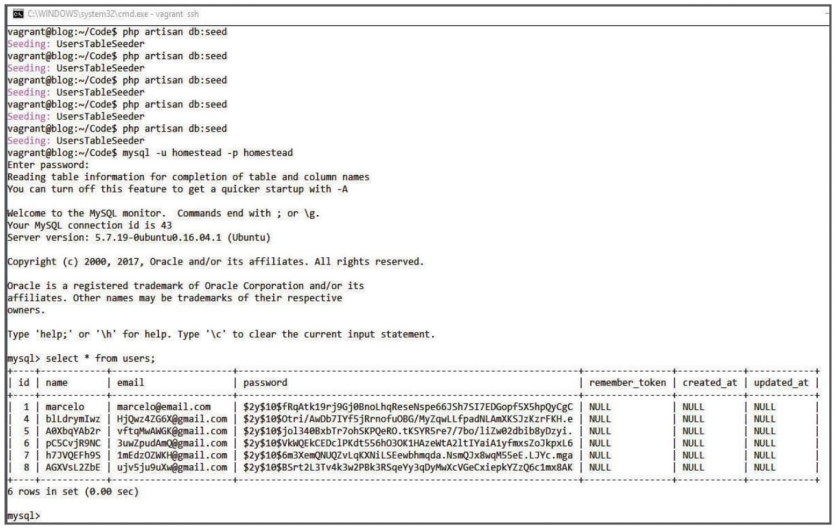


Figura 11. El mensaje de error que muestra Artisan es el mismo que obtendríamos con cualquier cliente MySQL.

Modifiquemos el método **run** para poder utilizarlo cuantas veces queramos:

```
public function run()
{
  DB::table('users')->insert([
       'name' => str_random(10),
       'email' => str_random(10).'@gmail.com',
       'password' => bcrypt('secret'),
  ]);
}
```

A partir de estos cambios, los correos y los nombres se generarán con cadenas de caracteres aleatorias, de modo que podemos ejecutar el método varias veces.



```
C:\WINDOWS\system32\cmd.exe - vagrant ssh
vagrant@blog:~/Code$ php artisan db:seed
Seeding: UsersTableSeeder
vagrant@blog:~/Code$ php artisan db:seed
Seeding: UsersTableSeeder
vagrant@blog:~/Code$ php artisan db:seed
Seeding: UsersTableSeeder
vagrant@blog:~/Code$ php artisan db:seed
Seeding: UsersTableSeeder
vagrant@blog:~/Code$ php artisan db:seed
Seeding: UsersTableSeeder
vagrant@blog:~/Code$ mysql -u homestead -p homestead
Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 43
Server version: 5.7.19-0ubuntu0.16.04.1 (Ubuntu)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select * from users;
+----+-------------+-------------------------+--------------------------------------------------------------+----------------+------------+------------+
| id | name        | email                   | password                                                     | remember_token | created_at | updated_at |
+----+-------------+-------------------------+--------------------------------------------------------------+----------------+------------+------------+
|  1 | marcelo     | marcelo@email.com       | $2y$10$fRqAtk19rj9Gj0BnoLhqReseNspe66JSh7SI7EDGopf5X5hpQyCgC | NULL           | NULL       | NULL       |
|  4 | blLdrymIwz  | HjQwz4ZG6X@gmail.com    | $2y$10$Otri/AwDb7IYf5jRrnofuOBG/MyZqwLLfpadNLAmXKSJzKzrFKH.e | NULL           | NULL       | NULL       |
|  5 | A0XbqYAb2r  | vftqMwAWGK@gmail.com    | $2y$10$jol340BxbIr7ohSKPQeRO.tKSYRSre7/7bo/liIw0dbib8yDzyi. | NULL           | NULL       | NULL       |
|  6 | pC5CvjR9NC  | 3uwZpudAmQ@gmail.com    | $2y$10$VkWQEkCEDc1PKdtS56hO3OK1HAzeWtA2ltIYaIAtyfmxsZoJkpxL6 | NULL           | NULL       | NULL       |
|  7 | h7JVQEFh9S  | 1mEdzOZWKH@gmail.com    | $2y$10$6m3XemQNUQZvLqKXNiLSEeubhmqda.NsmQJx8wqMS5eE.LJYc.mga | NULL           | NULL       | NULL       |
|  8 | AGXVsL2ZbE  | ujv5ju9uXw@gmail.com    | $2y$10$BSrt2L3Tv4k3w2PBk3RSqeYy3qDyMwXcVGeCxiepkYZzQ6c1mx8AK | NULL           | NULL       | NULL       |
+----+-------------+-------------------------+--------------------------------------------------------------+----------------+------------+------------+
6 rows in set (0.00 sec)

mysql>
```

Figura 12. A la hora de operar con seeders, siempre es necesario corroborar que la ejecución haya surtido efecto en la base de datos.

Fábricas

Las fábricas de objetos, del inglés *Factory*, permiten generar instancias de objetos con datos simulados. En el capítulo anterior creamos una clase **Factory** que persistía sus datos en memoria; ahora vamos a crear una Factory basada en una clase **Modelo**.

En el capítulo siguiente veremos cómo generar nuestros propios modelos, pero por el momento, utilizaremos el que trae Laravel para su componente de usuarios.

Las factory se almacenan en **database/factories**, y Laravel ya nos provee una para usuarios, la cual contiene el siguiente código:

```
$factory->define(App\User::class, function (Faker
$faker) {
    static $password;

    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'password' => $password ?: $password =
bcrypt('secret'),
        'remember_token' => str_random(10),
    ];
});
```

Como podemos observar, el método **define** de la instancia recibe como primer parámetro el nombre de una clase **Model**, y como segundo, una función **callback** que tiene una instancia de **Faker** como parámetro.

Nombres, ¿en inglés o en español?

Es común que nos surja esta pregunta durante el desarrollo. Por el momento estamos trabajando con nombres en español y manteniendo en inglés todo lo que el framework provee. No obstante, es una buena práctica mantener todo el código en inglés para conservar la consistencia general con lo ya establecido por el framework. Además, en español contamos con caracteres especiales, como la letra ñ, la cual puede provocar comportamientos no deseados.

Debemos cambiar el namespace **App\User** debido a que, al establecer el nombre de la aplicación en **Blog**, el namespace **App** ya no existe más; por lo tanto, la llamada al modelo debe ser **Blog\User::class**.

Para poder utilizar esta factory en nuestro seeder, basta con modificar el método **run** de la siguiente manera:

```
public function run()
{
  factory(Blog\User::class, 5)->create();
}
```

El helper **factory** recibe como primer parámetro el nombre del modelo que queramos instanciar, y como segundo parámetro, la cantidad de elementos. Por último, a través del método **create** indicamos que deseamos crear y guardar esos objetos en nuestra base de datos.

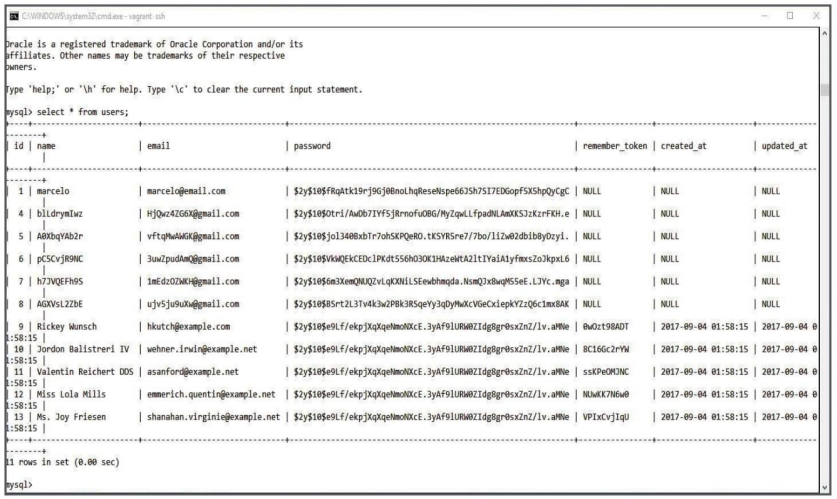


Figura 13. Observemos que, en este caso, los campos **timestamps** fueron completados.

Hay una diferencia fundamental entre crear registros mediante el método **insert** del servicio **DB** y hacerlo mediante **$factory**.

En el primer caso, estamos directamente insertando un registro en la base de datos. En el segundo, estamos insertándolo de manera indirecta, ya que trabajamos en base a un modelo, y los modelos utilizan **Eloquent**, el ORM de Laravel que trae consigo mucha funcionalidad, entre ella, la capacidad de detectar campos **timestamps** y establecer sus valores al momento de almacenar un registro en la base. Es por eso que los campos **created_at** y **updated_at** que se ingresaron en la última ejecución poseen datos, mientras que los que insertamos al principio no.

Resumen Capítulo 06

En este capítulo estudiamos la estrategia del framework para operar con las bases de datos y, luego, analizamos los archivos de configuración y la manera de conectarnos al motor MySQL que provee Homestead. A continuación, creamos una migración para generar la tabla donde almacenaremos las noticias de nuestro blog, vinculándola con la tabla de usuarios que ya brinda el framework. Por último, implementamos y comparamos dos estrategias diferentes para generar registros, una mediante el servicio **DB** y otra a través de **$factory**.

ACTIVIDADES

Test de Autoevaluación

1. ¿Cuál es la relación entre los modelos, los ORM y PDO?
2. ¿Qué parámetro define el motor de base de datos que se utilizará en una conexión de base de datos?
3. ¿Cómo podemos conectarnos al motor de base de datos que provee Homestead?
4. ¿Qué es una migración?
5. ¿Qué significa que una migración haya sido ejecutada?
6. ¿Cómo se llama el servicio que permite generar estructuras de base de datos en una migración?
7. ¿De qué forma podemos establecer relaciones entre dos tablas?
8. ¿Qué es un seeder?
9. ¿En qué consiste una fábrica de objetos?
10. ¿Cuál es la diferencia que existe entre cargar registros mediante el servicio **DB** y hacerlo mediante **$factory**?

Ejercicios prácticos

1. Cree las migraciones y los seeders utilizando **DB** para la tabla **categoría**. Vincule esta tabla con la tabla **noticias**.
2. Cree una migración para agregar el campo **copete** en la tabla **noticia**. Deberá tener un máximo de 255 caracteres y no ser obligatorio.
3. Analice los diferentes comandos de Artisan relacionados a las migraciones, especialmente el comando **php artisan refresh --seed**.

07 Modelos

El concepto de modelo puede resultar complejo. Esto se debe a que se utiliza tanto en la programación orientada a objetos como en el patrón de arquitectura MVC, y a su vez, es posible, algorítmicamente hablando, fusionar estos conceptos o separarlos. Por lo tanto, antes de comenzar a hablar de modelos es necesario hacer un breve repaso sobre la programación orientada a objetos y analizarla desde un punto de vista conceptual para, luego, compararla con los modelos que propone el patrón MVC.

PROGRAMACIÓN ORIENTADA A OBJETOS

En la década del 80 comenzaron a salir al mercado las primeras computadoras personales que incluían interfaces gráficas de usuario, y con ellas, empezó a surgir la idea de tener carpetas, archivos y ventanas alrededor de un escritorio. A partir de estas ideas, el paradigma de la programación orientada a objetos fue tomando vital importancia, ya que su propósito es crear objetos que interactúen en un contexto determinado. Incluso, el primer lenguaje de programación orientada a objetos se llamaba Simula 67 y fue creado con la idea de simular la actividad de aeronaves.

La metáfora del escritorio surgida en la década del 80 nos acompaña hasta el día de hoy. Tengamos en cuenta que su punto de partida fue una situación que se da en el mundo real, es decir, la de tener un escritorio con elementos que interactúan entre sí y nos permiten realizar diferentes trabajos. Luego, a partir de esa situación se crearon **modelos** de software que lograron trasladar ese mundo real a uno virtual.

El paradigma de la programación orientada a objetos se adapta muy fácilmente a estos escenarios, porque logra representar estos modelos de software en clases. Una **clase** es un modelo o plantilla que permite declarar las características que tendrán los objetos del sistema. Estas características se componen, principalmente, de atributos y métodos.

Los **atributos** son datos que permiten establecer el estado de un objeto, mientras que los **métodos** definen el comportamiento que éste podrá tener, tanto para sí mismo como para con otros objetos del sistema.

A su vez, cada objeto debe tener una propiedad que lo diferencie del resto, es decir, una **identidad**.

Modelos en MVC

Partiendo de la idea anterior, podemos crear modelos de software para simular diferentes situaciones que nos permitan resolver un problema en particular, estableciendo la forma en la que se representa cada entidad de un sistema y cómo se relacionan unas con otras. En otras palabras, estaremos construyendo la **lógica de negocio** de nuestro sistema.

El patrón MVC complementa esta idea e incorpora la gestión de la administración de la información de estas entidades. Todo lo vinculado

a la persistencia, es decir, el almacenamiento de la información, está también involucrado en una clase denominada **Modelo**.

En consecuencia, si pensamos en los modelos en el patrón MVC, estamos hablando de clases que contienen tanto la lógica de dominio como la de la persistencia en la información de estas clases. Éste es el enfoque que estudiaremos en este capítulo, ya que es el mismo que presenta el framework Laravel. A continuación, presentamos un diagrama de clases que modela el escenario que construiremos para nuestro blog.

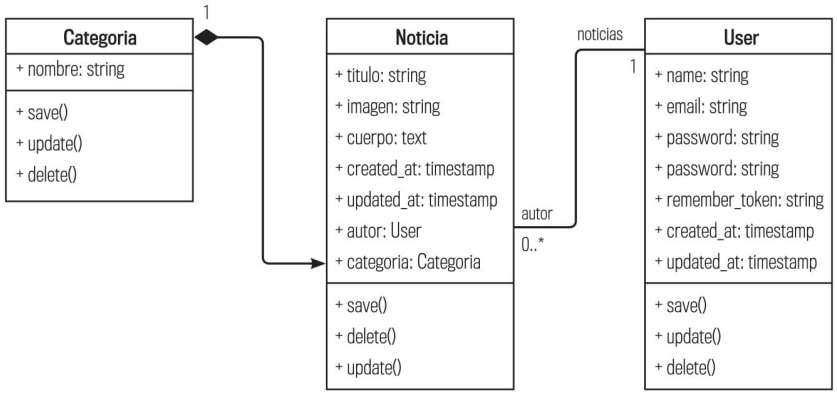


Figura 1. Es muy recomendable diagramar las clases del sistema porque nos darán un mayor conocimiento del problema que debemos dominar.

Patrón repository

Cuando hablamos de persistencia de la información, lo primero que nos viene a la mente son las bases de datos relacionales. Sin embargo, en la actualidad también contamos con servicios web que permiten persistir información y, sobre todo, con bases de datos no relacionales, las cuales están abarcando cada vez más mercado. Debido a esto, en algunos sistemas se separa la lógica de negocios, de la lógica de persistencia de la información, creando dos conjuntos de clases: por un lado, los modelos, y por otro, clases tipo **repository**, que serán implementadas por interfaces que permitan gestionar la información de los modelos, pero abstrayendo a éstos de la necesidad de generar la lógica de conexiones, comandos y estructuras internas. Esta estrategia obedece a un patrón de diseño denominado **patrón de repositorios** o *repository pattern*.

Patrón Active Record

En el capítulo anterior introdujimos el concepto de ORM. Ahora profundizaremos en la forma en la que Laravel implementa su ORM denominado **Eloquent**.

Como sabemos, existen métodos para guardar (**save**), actualizar (**update**) y eliminar (**delete**) la información de cada clase. Esto responde a la idea de ABM y, también, al patrón de diseño **Active Record**, el cual establece que los atributos de un objeto deben relacionarse directamente con las columnas de una tabla de base de datos.

Eloquent utiliza la herencia para poder generar los modelos; por lo tanto, cada clase que queramos convertir en un modelo deberá heredar de la clase **Model** provista por el framework.

Recordemos que la herencia es una característica de la programación orientada a objetos en la cual podemos establecer una relación de jerarquía entre clases, creando una clase padre y una o varias clases hijas.

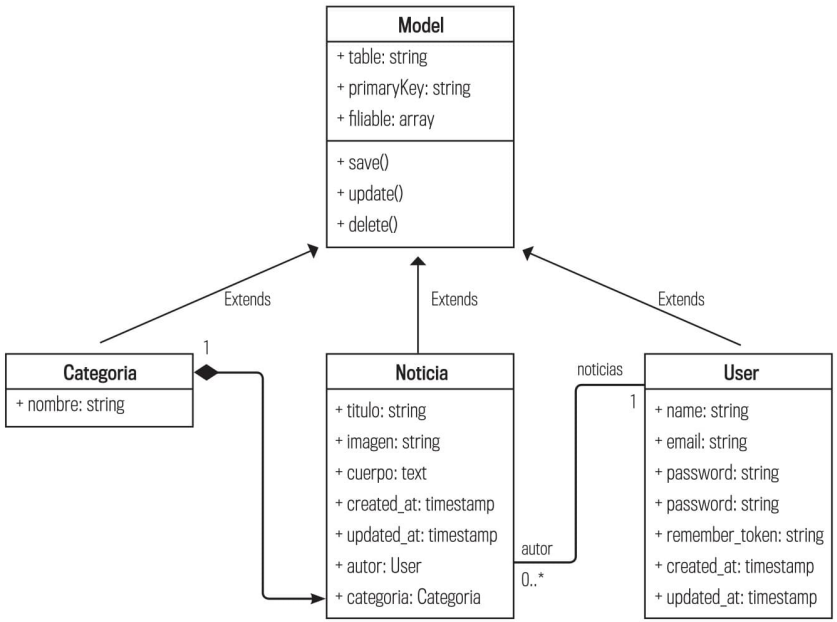


Figura 2. En la herencia, los métodos establecidos en la clase padre (**Model**) son heredados por las clases hijas (**Categoria**, **Noticia**, **User**).

A partir de las modificaciones generadas, una de las grandes ventajas de Laravel es la de enfocarnos directamente en nuestra lógica de negocio, dado que todo lo relacionado con la persistencia de la información ya lo ha resuelto por nosotros.

Eloquent hace esto posible, en parte, siguiendo una determinada nomenclatura, y en parte, realizando consultas a los metadatos de la estructura de la base de datos. Pero para empezar a analizar este tema con profundidad comencemos a crear nuestros primeros modelos.

MODELOS EN LARAVEL

Como ya sabemos, tenemos muchos comandos en Artisan para generar código, y los modelos no son una excepción. Generemos el modelo de nuestra clase **Noticia** con el comando **php artisan make:model Noticia**.

Este comando generará un archivo **Noticia.php** en la carpeta **app**. Hemos observado que es conveniente agrupar los archivos conforme a los subsistemas que crearemos en nuestra aplicación. Sin embargo, los modelos son componentes transversales a todos los sistemas. Tengamos en cuenta que cuando guardemos una noticia en una base de datos, lo haremos siempre de la misma manera, sin importar el subsistema que invoque a la ejecución del método **save** del modelo.

Es necesario considerar que el patrón denominado Active Record es un término conceptual implementado por muchos ORM, por lo tanto, nos sirve más allá del lenguaje de programación PHP.

Documentación de Laravel

Además de la documentación oficial, siempre es bueno tener como referencia la documentación generada a partir del código fuente del framework, la cual puede consultarse en **https://laravel.com/api/5.5/index.html**. Por ejemplo, si observamos la clase **model**, vamos a encontrar muchas más funcionalidades que heredamos además de la generada para el patrón **Active Record**. También es posible acceder a esta información si utilizamos un IDE como Netbeans.

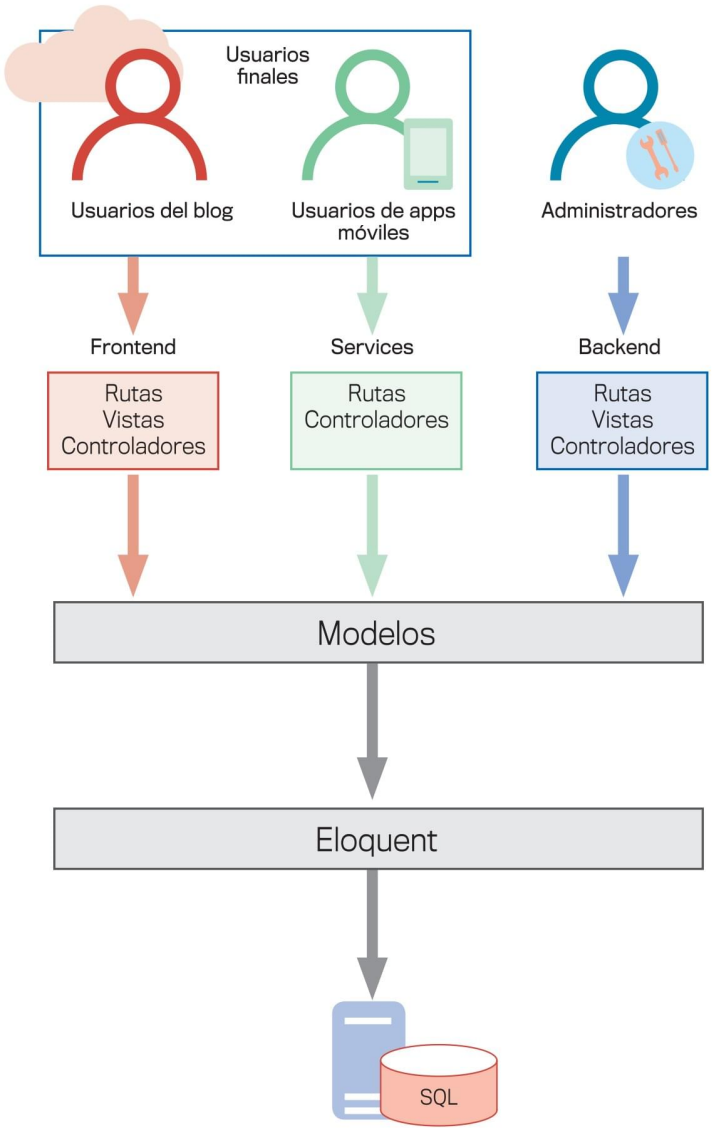


Figura 3. Cada subsistema está pensado para interactuar con diferentes grupos de usuarios.

La ventaja de esta arquitectura es que podemos implementar diferentes estrategias en cada subsistema para acondicionar la información y las estructuras de datos a cada grupo de usuarios.

Si bien no es necesario agrupar los modelos por subsistema, vamos a organizarlos dentro de un namespace para tenerlos todos agrupados en una carpeta. Para hacerlo, debemos:

- Crear la carpeta **app/Models**.
- Mover el archivo **app/Noticia.php** a la carpeta **app/models**.
- Actualizar el namespace a **Blog\Models**.

Podemos hacer lo mismo con el modelo **User.php**, pero debemos actualizar todas las referencias donde ya estamos utilizando este modelo.

Características de los modelos

Los modelos tienen una relación directa con la base de datos; esto quiere decir que, para que funcionen correctamente, tenemos dos aproximaciones posibles:

1	**Adaptar la estructura de la base de datos siguiendo características y nomenclaturas que los modelos esperan de la base.**
2	**Adaptar los modelos para establecer aquellas características que el modelo utilizará para operar con la base de datos.**

Vamos a analizar estas aproximaciones creando un seeder para nuestras noticias con el comando **php artisan make:seeder NoticiasTableSeeder** y le asignaremos la siguiente lógica:

```
<?php

use Illuminate\Database\Seeder;
use Blog\Models\Noticia;

class NoticiasTableSeeder extends Seeder
```

```
    {
        public function run()
        {
            $n = new Noticia();
            $n->titulo = "Titulo de prueba";
            $n->cuerpo = "Cuerpo de prueba";
            //Este valor debe corresponder al ID de un regis-
tro de la tabla users
            $n->autor = 1;
            $n->save();
        }
    }
```

Podemos ejecutar únicamente este seeder mediante el comando **php artisan db:seed --class=NoticiasTableSeeder.**

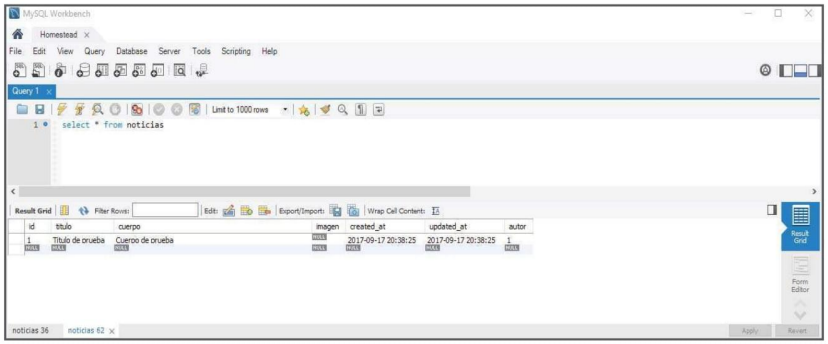


Figura 4. Siempre es necesario verificar que el seeder haya logrado el resultado deseado.

En este seeder hemos inaugurado el modelo **noticia**. Si observamos con detalle el código fuente y las columnas de la tabla, podemos identificar que existe una relación directa entre los nombres de los atributos de la clase y los nombres de las columnas.

Intentemos asignar a nuestro objeto **noticia** un atributo que no exista en la base de datos agregando antes de **$n->save();** lo siguiente: **$n->noExiste = "no existe";**. Luego ejecutemos el seeder otra vez.

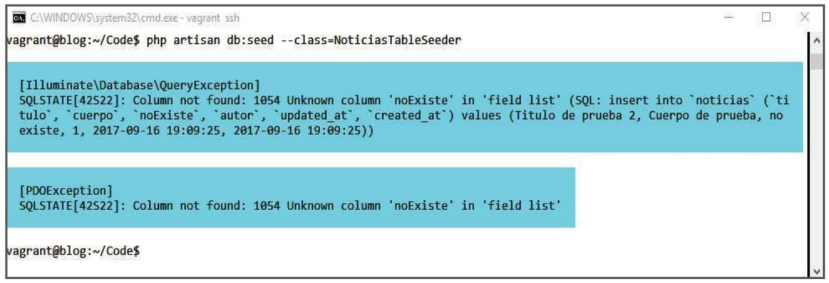


Figura 5. Recordemos que Eloquent detecta los campos **timestamps** y los asigna correctamente, lo que nos ahorra tener que hacer la asignación.

La **Figura 5** ilustra la relación que existe entre los nombres de los atributos y de las columnas de la base de datos. Eloquent parece haber acertado al nombre de la tabla para armar la query, pero esto no es una casualidad. Por convención, Eloquent utilizará el nombre en plural de la clase. Como la clase se llama **Noticia**, intentará insertar los datos en la tabla **noticias**, a menos que especifiquemos lo contrario en el modelo, lo cual podemos hacer asignando a la clase el atributo de Eloquent **$tableName**. Modifiquemos el modelo **Noticia** para asignar otro nombre:

```
<?php

namespace Blog\Models;

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

Aprovechar las ventajas del lenguaje

PHP es un lenguaje no tipado, esto quiere decir que podemos establecer valores para los atributos de un objeto sin especificar su tipo y, a la vez, tampoco es necesario declarar los atributos en la clase para poder asignarlos a un objeto. Esto lo convierte en un lenguaje muy dinámico, pero este dinamismo, que es muy útil en los tiempos de desarrollo, se penaliza en los tiempos de ejecución debido a que el intérprete debe procesar todas estas características.

```
class Noticia extends Model
{
    protected $table = 'noticias_del_blog';
}
```

Eliminemos el atributo **noExiste** del seeder y ejecutémoslo nuevamente. Se intentará insertar el registro en una tabla que no existe en la base de datos. Para que nuestro modelo vuelva a funcionar, vamos a eliminar el atributo **$table**. La **Tabla 1** muestra los principales atributos de Eloquent que podemos utilizar en nuestros modelos.

PRINCIPALES ATRIBUTOS DE LA CLASE MODEL DE ELOQUENT

ATRIBUTO	DESCRIPCIÓN
$table	Establece el nombre de la tabla de la base de datos donde se almacenarán los datos.
$primaryKey	Establece el nombre de la clave primaria de la tabla. Si no se define, utilizará el campo **id**.
$incrementing	Debe establecerse en false cuando no deseamos utilizar un campo autoincremental para la clave primaria de la tabla.
$connection	Recordemos que Laravel permite tener varias conexiones de base de datos; si no se establece este parámetro, cada modelo intentará utilizar la base definida por default.
$timestamps	Debemos establecerlo en false si no queremos utilizar los campos **created_at y updated_at**, los cuales espera encontrar por default en todas las tablas.
$dates	Es un array que indica los atributos del modelo que deben ser tratados como fechas, para lo cual Laravel crea objetos de la librería Carbon **https://github.com/briannesbitt/Carbon**.

Tabla 1. Podemos encontrar la descripción completa en **https://laravel.com/api/5.5/Illuminate/Database/Eloquent/Model.html**.

Persistir modelos

Como ya sabemos, el método **save** nos permite insertar un modelo, pero el mismo método nos da la posibilidad de actualizar un registro ya existente. Previamente, debemos ir a buscar dicho registro utilizando el método **find**. Vamos a modificar el seeder de la siguiente manera y luego lo ejecutamos:


```
    public function run()
    {
      $n = Noticia::find(1);
      $n->titulo = "Titulo de prueba Modificado";
      $n->save();
    }
```

El método **find** buscará en la tabla **noticias** a través de su clave primaria y retornará un objeto **Noticia**. Dado que el objeto tiene establecido el campo **id**, Eloquent realizará un **update** en vez de un **insert**.

Otra manera de aprovechar la inteligencia de Laravel es utilizando el método **firstOrNew**, el cual utilizará los atributos del objeto para buscarlo en la base de datos; en caso de no encontrarlo, retornará un objeto nuevo.

Probémoslo modificando el seeder de la siguiente manera:

```
    public function run()
    {
      //Este objeto existe en la base
      $n = Noticia::firstOrNew(['titulo' => "Titulo de
prueba Modificado"]);
      $n->titulo = "Titulo de prueba Modificado Nueva-
mente";
      $n->save();
      //Este objeto será creado
      $n = Noticia::firstOrNew(['titulo' => "Titulo de
prueba Nuevo"]);
```

Pluralidad de los modelos

La convención de nombres de modelos en singular y nombres de las tablas en plural se realiza teniendo en cuenta las reglas del idioma inglés. En el caso de **Noticia** no tenemos inconvenientes, pero si tuviésemos un modelo **Ciudad**, sería un problema, debido a que Eloquent no intentará buscar la tabla **ciudades**. Éste es otro motivo por el cual es conveniente utilizar este idioma para asignar nombres en todo el sistema.

```
        $n->cuerpo = "Como es nuevo, hay que asignar los
valores not null";
        $n->autor = 1;
        $n->save();
    }
```

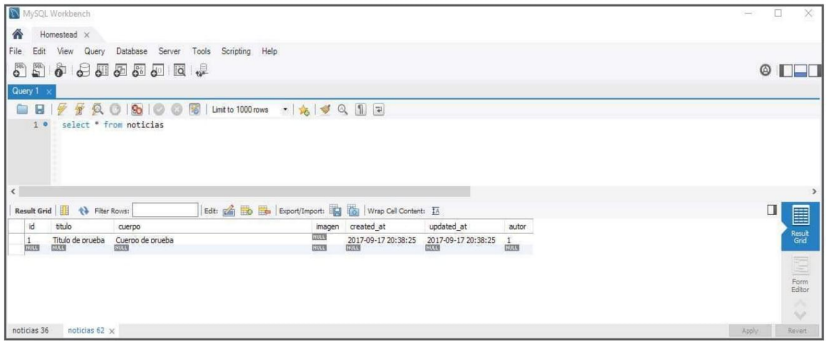


Figura 6. El método **firstOrNew** recibe un array asociativo donde los índices representan los atributos por los cuales se buscará el objeto.

Alternativamente, también contamos con el método **firstOrCreate**, el cual funciona de la misma manera que **firstOrNew** con la salvedad de que creará el objeto en la base de datos en caso de que no exista al realizar la búsqueda, mientras que **firstOrNew** lo crea en caso de que no exista al invocar al método **save**.

Eliminar modelos

Para eliminar un modelo, podemos usar el método **delete**, pero primero tenemos que buscarlo con **find**. Hagamos la prueba modificando el seeder:

```
    public function run()
    {
      $n = Noticia::find(1);
      $n->delete();
    }
```


Podemos ahorrarnos la búsqueda si conocemos el id del registro a eliminar, utilizando el método **destroy**:

```
    public function run()
    {
      $n = Noticia::destroy(2);
    }
```

En caso de enviar un id que no exista, el método no retornará un error. Éste es otro motivo más por el cual es conveniente revisar que los cambios deseados se hayan impactado en la base cuando ejecutamos un seeder.

Soft Deletes

El término **soft delete**, el cual traducido del inglés significa eliminación blanda, consiste en realizar una eliminación lógica del objeto. Esto quiere decir que se implementa una estrategia para marcar el objeto como eliminado, pero se mantiene el registro en la base de datos.

Vamos a convertir nuestras noticias en soft deletes creando una nueva migración con el comando **php artisan make:migration --table noticias add_soft_delete_noticias**. El parámetro **--table** inicializará la función **up** con el método **table** de **Schema**, el cual nos permitirá obtener una referencia a la tabla lista para trabajar. Lo modificamos de la siguiente manera:

```
    public function up()
    {
        Schema::table('noticias', function (Blueprint
$table) {
            $table->softDeletes();
        });
    }
```

Antes de ejecutar la migración, veamos los cambios que realizará en la base de datos con **php artisan migrate –pretend**:

```
  vagrant@blog:~/Code$ php artisan migrate --pretend
  AddSoftDeleteNoticias: alter table `noticias` add `de-
leted_at` timestamp null
```

La estrategia que utilizará Eloquent para los modelos **soft delete** es crear un campo **deleted_at**, el cual, al estar establecido, dejará registrada la fecha en la que el registro fue marcado como eliminado. Es por ello que el método **down** de la migración deberá definirse de la siguiente forma:

```
      public function down()
      {
          Schema::table('noticias', function (Blueprint
$table) {
              $table->dropColumn('deleted_at');
          });
      }
```

Ejecutemos la migración con **php artisan migrate** y luego modifiquemos el método **run** del seeder de noticias del siguiente modo:

```
      public function run()
      {
        $n = Noticia::firstOrNew(['titulo' => "Noticia para
eliminar"]);
        $n->cuerpo = "Como es nuevo, hay que asignar los
valores not null";
        $n->autor = 1;
        $n->save();
        $n->delete();
      }
```

Al ejecutar el seeder, podemos observar que el registro aún continúa eliminándose de la base de datos. Esto se debe a que, además de indicar en la migración que utilizaremos **soft deletes**, también debemos hacerlo en el modelo implementando el trait **softDeletes**, lo cual hacemos agregando la sentencia **use SoftDeletes;**

Modifiquemos el modelo de la siguiente forma:

```
<?php

namespace Blog\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Noticia extends Model
{
    use SoftDeletes;
    protected $dates = ['deleted_at'];
}
```

Si ejecutamos otra vez el seeder, veremos que el registro se creó en la base de datos, pero con el campo **deleted_at** establecido.

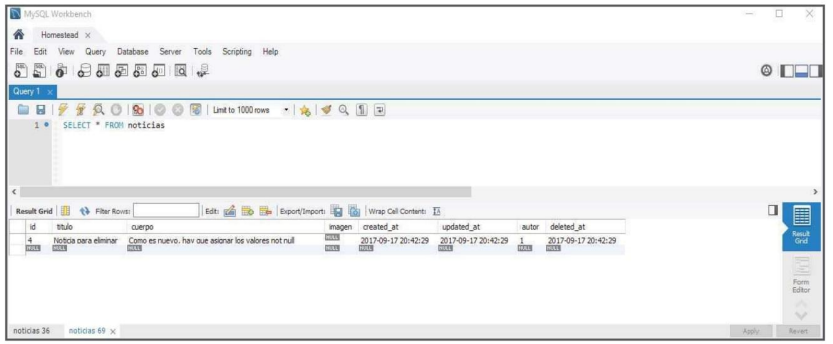


Figura 7. Eloquent filtrará por default todos los registros que tengan el campo **deleted_at**.

Traits

Los traits son una estrategia de reutilización de código muy empleada en Laravel y disponible en PHP desde la versión 5.4. Consiste en poder utilizar métodos y propiedades sobre varias clases independientes y pertenecientes a clases jerárquicas distintas. Es recomendable leer la documentación oficial de PHP, disponible en español en **http://php.net/manual/es/language.oop5.traits.php.**

A partir de este punto, sólo podremos acceder a los registros eliminados especificando su inclusión en los métodos de búsqueda de los objetos.

Probando modelos

Hasta ahora fuimos probando los modelos utilizando seeders, pero recordemos que éstos están para poder cargar las tablas de las bases de datos. Una herramienta más apropiada para probar los modelos es Tinker.

Tinker es una consola de programación que permite leer las entradas que se ingresan en la consola, evaluarlas e imprimirlas. Es parte de los sistemas denominados REPL, por su singla en inglés *Read, Eval, Print Loop*. Fue incorporado a Laravel en la versión 5.4, y en versiones anteriores puede instalarse mediante Composer. Para iniciar sesión en Tinker debemos ejecutar el comando **php artisan tinker**.

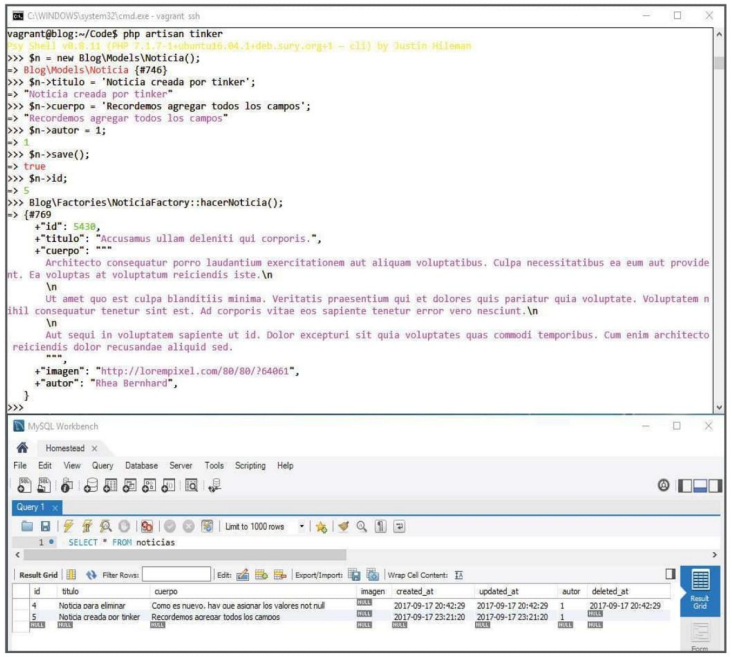


Figura 8. Tinker nos permite acceder a todas las clases de nuestra aplicación, incluidos los modelos.

En otras palabras, Tinker convierte nuestra consola en una interfaz para interactuar con toda la lógica de nuestra aplicación, sin necesidad de crear interfaces gráficas o comandos. Recordemos que, además de toda la lógica de base de datos, en los modelos debemos introducir también la lógica de negocio.

Supongamos que, a partir de nuestro negocio, debemos identificar todas las noticias creadas o actualizadas del último mes como nuevas. Agreguemos el siguiente método en **Blog\Models\Noticia.php**:

```
    public function esNueva()
    {
      $lastMonth = \Carbon\Carbon::now()->subDays(30);
      return $this->created_at->greaterThan($lastMonth)
|| $this->updated_at->greaterThan($lastMonth);
    }
```

Si intentamos acceder al método desde la sesión de Tinker que ya tenemos abierta, habrá un error, dado que dicho método no existía al momento de iniciar esa sesión. Por lo tanto, es necesario salir de Tinker e iniciar sesión nuevamente.

Resumen Capítulo 07

En este capítulo realizamos un breve repaso del paradigma de la programación orientada a objetos, para luego profundizar en diferentes acepciones del concepto de modelo. A continuación, estudiamos los patrones de diseño repository y Active Record para analizar su relación con la implementación de modelos que realiza Laravel. Más adelante, vimos las ventajas de la estrategia de dividir el proyecto en subsistemas, creamos el modelo de Noticia y examinamos sus principales características. Por último, probamos las diferentes formas de crear, modificar y eliminar modelos con la estrategia de eliminaciones blandas.

ACTIVIDADES

Test de Autoevaluación

1. ¿Cuál es la relación entre los modelos, los ORM y PDO?
2. ¿Qué parámetro define el motor de base de datos que se utilizará en una conexión de base de datos?
3. ¿Cómo podemos conectarnos al motor de base de datos que provee Homestead?
4. ¿Qué es una migración?
5. ¿Qué significa que una migración haya sido ejecutada?
6. ¿Cómo se llama el servicio que permite generar estructuras de base de datos en una migración?
7. ¿De qué forma podemos establecer relaciones entre dos tablas?
8. ¿Qué es un seeder?
9. ¿En qué consiste una fábrica de objetos?
10. ¿Cuál es la diferencia que existe entre cargar registros mediante el servicio **DB** y hacerlo mediante **$factory**?

Ejercicios prácticos

1. Cree el modelo de la clase **Categoria**.
2. Cree un seeder para generar diferentes categorías utilizando modelos.
3. Implemente la estrategia soft delete en el modelo **Categoria**.

Trabajar con modelos

08

En el capítulo anterior hemos empezado a construir modelos para relacionarlos con las tablas de la base de datos de nuestro sistema. En este capítulo comenzaremos a realizar operaciones con los modelos de manera tal que podamos aprovechar al máximo las ventajas de Eloquent.

RECUPERAR MODELOS

Es hora de reemplazar los mocks que estamos utilizando en los controladores y empezar a utilizar modelos.

Antes de hacerlo, nos será conveniente tener un mayor volumen de datos para operar.

Para esto, crearemos una fábrica con el comando **php artisan make:factory NoticiasFactory --model Models\\Noticia** y seguiremos la misma lógica establecida para la clase **UserFactory**:

```
<?php

use Faker\Generator as Faker;

$factory->define(Blog\Models\Noticia::class, function
(Faker $faker) {
    $created = $faker->dateTimeBetween('-5 years');
    return [
        'titulo' => $faker->sentence,
        'cuerpo' => $faker->text,
        'imagen' => $faker->optional()->imageUrl,
        'created_at' => $created,
        'updated_at' => $faker->dateTimeBetween($created),
        'autor' => 1
    ];
});
```

Por último, actualicemos el seeder de noticias:

```
<?php

use Illuminate\Database\Seeder;
use Blog\Models\Noticia;

class NoticiasTableSeeder extends Seeder
{
```



```
    public function run()
    {
        factory(Blog\Models\Noticia::class, 100)->cre-
ate();
    }
}
```

De esta manera, si ejecutamos el seeder nuevamente, tendremos varias noticias para poder trabajar.

Modifiquemos el controlador **Blog\Http\Controllers\Backend\NoticiaController.php**, primero agregando use **Blog\Models\Noticia;** debajo del **namespace** y luego el método **index** para que quede de la siguiente forma:

```
    public function index(){
      $noticias = Noticia::all();
      return view('backend.noticia.index', ['noticias' =>
$noticias]);
    }
```

No es necesario hacer ningún otro cambio; si ingresamos en la URL **http://localhost:8000/backend/noticia**, veremos las noticias que ya no son más un mock, sino que se recuperan desde la base de datos.

Novedades de Laravel 5.5

El comando **make:factory** se encuentra disponible a partir de la versión 5.5 y nos permite separar nuestras factory en diferentes archivos. En las versiones previas, todas las factory debían estar en un único archivo, lo cual, además de ir en contra de la idea de mantener el código limpio y elegante, podía llegar a generar un archivo muy largo, difícil de mantener y de desarrollar en un contexto colaborativo.

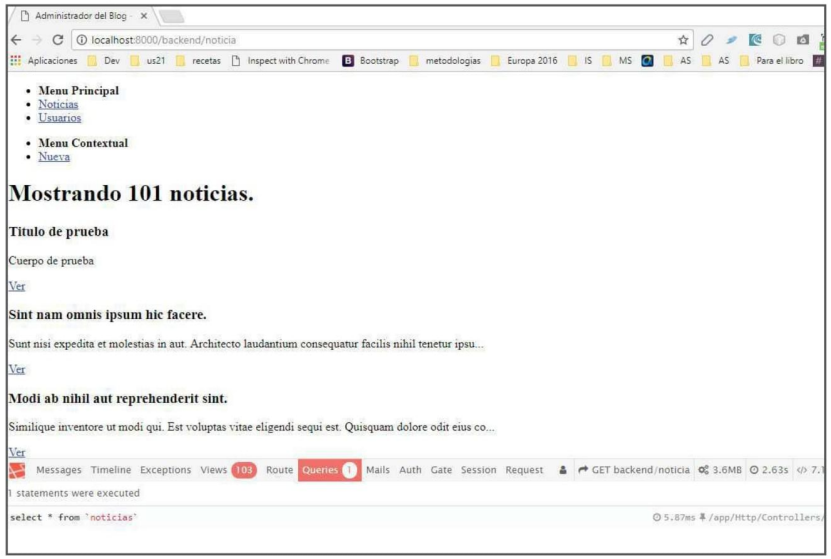


Figura 1. Debugbar nos presenta las consultas que se ejecutan en la base de datos para mostrar esta pantalla.

Esto es posible gracias al método **all**, el cual busca todas las noticias y devuelve una **Collection**.

Colecciones

Cuando trabajamos fuera de un ORM y fuera de PDO, por lo general, creamos una conexión a la base de datos, luego ejecutamos una consulta con esa conexión, y ésta nos devuelve un resultado o **result set**, el cual debemos procesar, por ejemplo, en el caso de MySQL, con funciones del tipo **mysql_fetch_array**.

Eloquent hace todo este proceso por nosotros, y como resultado tenemos un objeto **Collection** que, además de contener los resultados de la consulta, también contiene una serie de características muy útiles.

Recordemos que en la vista estamos iterando cada noticia mediante la sentencia **@each('backend.noticia.item', $noticias,'noticia','backend.noticia.empty')**. Por lo tanto, podemos descifrar que las Collections son

iterables, esto quiere decir que es posible generar bucles para recorrer cada uno de los elementos que contiene la Collection.

El método **all** es la manera más básica de recuperar los modelos de la base de datos, sin embargo, tenemos varias opciones.

Vamos a modificar la vista **resources\views\backend\noticia\item.blade.php** de la siguiente forma:

```
  <div>
    <h3>{{ $noticia->titulo }}</h3>
    @if(isset($noticia->imagen))
      <img src="{{ $noticia->imagen }}" style="max-height:
100px;" />
    @else
      <h4>No hay imagen disponible</h4>
    @endif
    <p>{{ str_limit($noticia->cuerpo, 100) }}</p>
    <a href="{{ route('backend.noticia.show', ['noticia' =>
$noticia->id]) }}">Ver</a>
  </div>
```

Recordemos que las imágenes son opcionales y el Factory de noticias que creamos contiene el modificador **optional**, el cual hace que en algunos casos inserte un valor y en otros no.

El cambio que acabamos de introducir hace que, al renderizar cada ítem, es decir, cada noticia, en caso de que contenga imagen, se aplique el tag **img** para mostrarla, y en caso de que no la tenga, muestra la leyenda No hay imagen disponible.

Lorempixel

Lorempixel es un sitio web gratuito disponible en **http://lorempixel.com** que ofrece URLs con imágenes de prueba para utilizar en la etapa de desarrollo. Es muy útil ya que provee diferentes opciones en relación a tamaños y categorías. Faker usa este servicio para generar URLs que nos permiten acceder a imágenes; podemos ver todas sus opciones en **https://github.com/fzaninotto/Faker#fakerproviderimage.**

Vamos a crear un grupo para filtrar noticias dentro del grupo de rutas del backend cuyo resultado sea el siguiente:

```
  Route::namespace('Backend')->name('backend.')->prefix('/
backend')->group(function () {
      Route::resource('noticia', 'NoticiaController');
      Route::prefix('noticias/filtradas')->group(function (){
          Route::get('con-imagenes', 'NoticiaController@
conImagenes');
      });
      Route::resource('categorias', 'CategoriaController');
  });
```

Por último, crearemos el método **conImagenes** en el controlador:

```
      public function conImagenes(){
        $noticias = Noticia::whereNotNull('imagen')->get();
        return view('backend.noticia.index', ['noticias' =>
$noticias]);
      }
```

Figura 2. Observemos que, por default, Eloquent trae únicamente todos los registros que tengan el campo **deleted_at** en **null**.

Observemos que, a la hora de buscar noticias, hemos realizado una variación que introduce dos nuevos métodos:

whereNotNull	En este caso estamos implementando un método de la clase **Database** de Laravel, también conocida como **QueryBuilder**, que traducido sería constructor de consultas. Todos los modelos de Laravel son QueryBuilders, por lo tanto, podemos utilizar en los modelos todos los métodos de la clase **Database**.
get	QueryBuilder nos permite ir construyendo consultas parciales, de manera tal que la misma se ejecutará en la base de datos cuando ejecutemos el método **get**, que nos retornará una Collection. También se ejecutará si utilizamos el método **first**, el cual devolverá un modelo.

En este caso, estamos buscando todas las noticias que tengan imágenes, es decir, aquellas cuyo campo **imagen** sea **not null**.

En esta última pieza de código hay una observación importante que debemos hacer. En el capítulo anterior mencionamos las ventajas de introducir lógica de negocios en los modelos para que pueda ser reutilizada en los demás subsistemas. Habiendo resuelto la búsqueda de noticias con imágenes de esta manera, este método no estará disponible para los demás subsistemas, por lo cual nos será conveniente crear un método en el modelo para introducir esta lógica. De esta forma, el filtro estará disponible para ser utilizado en otras partes del sistema.

Para hacerlo, introduzcamos en el modelo **Blog\Models\Noticia.php** el siguiente método:

```
      public static function conImagenes(){
        return Noticia::whereNotNull('imagen')->get();
      }
```

Y modifiquemos el controlador para hacer uso de este nuevo método:

```
      public function conImagenes(){
        $noticias = Noticia::conImagenes();
```

```
        return view('backend.noticia.index', ['noticias' =>
$noticias]);
      }
```

Si bien el resultado es el mismo, ahora tendremos el método **conImagenes** disponible para usar en los demás controladores. Hay una manera de mejorar incluso más esta estrategia por medio de QueryScopes, para lo cual dedicaremos una sección particular sobre ese tema en este mismo capítulo.

QueryBuilder

Eloquent dispone de una clase denominada **Builder**, a la cual podemos acceder mediante su facade **DB**; se la conoce más por su alias **QueryBuilder**.

Los facades son una manera de acceder a los servicios de Laravel utilizando una sintaxis simple de recordar, y nos permiten mantener el código simple y más legible. Por ejemplo, es mucho más sencillo recordar e interpretar este código:

```
  DB::table('noticias')->where('titulo', = 'Noticia para
eliminar');
```

Que éste:

```
  $c = new Illuminate\Database\Connection();
  $c->table('noticias')->where('titulo', = 'Noticia para
eliminar');
```

Todos los modelos de Eloquent son QueryBuilders, pero a su vez, podemos generar instancias no asociadas a modelos mediante los métodos **DB::table** y **DB::query**.

Esta clase permite ejecutar consultas a la base de datos, para lo cual nos ofrece una serie de métodos muy convenientes que nos permiten construirlas.

Es muy probable que, si estamos acostumbrados a trabajar de manera directa con una base de datos, utilizar métodos de una clase para realizar consultas nos resulte absurdo e, inclusive, complejo. Sin embargo, toma muy poco tiempo empezar a aprovechar las ventajas que ofrece esta estructura, entre las cuales podemos destacar:

Código más legible

El código que acabamos de escribir es autodescriptivo, es decir, el método se llama **conImagenes** y lo invocamos escribiendo **Noticia::conImagenes**. Como resultado, valga la redundancia, obtenemos únicamente noticias que contengan imágenes.

Consultas parciales

Al utilizar los métodos **Database**, obtenemos como resultado un objeto **QueryBuilder**, el cual podemos manipular en tiempo de ejecución utilizando métodos de una clase. Esto evita tener que manipular cadenas de texto para armar consultas.

Prevenir la inyección SQL

Recordemos que Eloquent utiliza PDO; en consecuencia, para asociar parámetros a una consulta ya se están aplicando los filtros necesarios para evitar que se inyecte código SQL en ella.

Abstracción parcial de las características particulares de cada motor

Si queremos obtener diez noticias en MySQL, podemos lograrlo con **select * from noticias limit 10**, mientras que en SQL Server debemos hacerlo con **select top 10 from noticias**. Utilizando **QueryBuilder**, con sólo emplear el método **->limit(10)**, nos abstraemos de tener que diferenciar consultas para cada motor, ya que Eloquent hace ese esfuerzo por nosotros.

Considerar el contexto

Algunos motores de base de datos proveen funciones muy útiles que pueden ahorrarnos muchos problemas. No obstante, desde el momento en que decidimos utilizar este tipo de funciones en nuestro sistema, lo hacemos dependiente del motor. La mejor manera de utilizar estas funciones en Laravel es a través del método **raw**; sin embargo, en caso de que exista una migración de motor y/o una actualización de la versión, debemos recordar adaptar el uso de estas funciones para mantener la compatibilidad del sistema.

De todas maneras, si no estamos del todo convencidos de utilizar estos métodos, **QueryBuilder** nos permite realizar una ejecución directa a la base de datos mediante el método **raw**.

La **Tabla 1** muestra los principales métodos de la clase **Database** que podemos utilizar para construir consultas.

▶ PRINCIPALES MÉTODOS DE QUERYBUILDER

Método	Ejemplos
select	DB::table('noticias')->select('titulo');
join	DB::table('noticias') ▶ join('users', 'users.id', '=', 'noticias.autor') ▶ select('noticias.*', 'autor.name');
where	DB::table('noticias')->where('titulo', = 'Noticia para eliminar'); DB::table('noticias') ▶ where(id, '>', 100) ▶ orWhere('autor', 1);
orderBy	DB::table('noticias') ▶ orderBy('titulo', 'desc');
groupBy	DB::table('noticias') ▶ groupBy('autor');

Tabla 1. Como podemos observar, existe un vínculo muy fuerte entre los métodos de la clase y el lenguaje SQL.

La documentación oficial disponible en **https://laravel.com/docs/5.5/queries** presenta muchos ejemplos de uso, y es muy recomendable analizar los distintos casos para tener una idea clara acerca de todas las posibilidades que ofrece la clase **QueryBuilder**.

Query Scopes

Siguiendo con la idea de ofrecer un código simple y elegante, Eloquent nos brinda la posibilidad de utilizar scopes. La palabra scope significa alcance, y es muy útil cuando queremos acotar una selección de elementos.

Nuestro método **conImagenes** ya es bastante simple. Sin embargo, a través de los scopes podemos mejorarlo aún más, simplemente reemplazando el método actual por el siguiente:

```
    public function scopeConImagenes($query){
      $query->whereNotNull('imagen');
    }
```

También vamos a reemplazar su uso en el controlador:

```
    public function conImagenes(){
      $noticias = Noticia::conImagenes()->get();
      return view('backend.noticia.index', ['noticias' =>
$noticias]);
    }
```

Para convertir un método en scope, sólo debemos iniciarlo con la palabra **scope** y utilizar como parámetro **$query**, ya que en esta variable residirá el **QueryBuilder** con el cual estaremos operando. Si generamos varios métodos scope, podemos combinarlos o utilizarlos por separado.

Agreguemos el siguiente scope en **Blog\Models\Noticia.php**:

```
  public function scopeRecientes(){
    $query->where('created_at', '>=', \Carbon\
Carbon::now()->startOfMonth())->get();
  }
```

Importancia del inglés

Saber inglés es importante no sólo por una cuestión de mercado, también lo es pues nos permitirá entender de manera más simple el código que logramos a través del uso de librerías abiertas. En el ejemplo que hemos revisado en las páginas anteriores podemos ver que, para obtener las noticias recientes, se buscan aquellas cuya fecha de creación sea mayor o igual a la fecha del mes en curso.

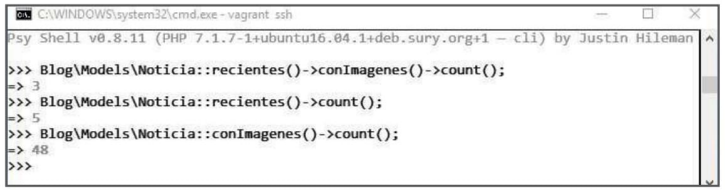


Figura 3. Los scopes pueden utilizarse por separado o combinarse, en esto consiste la idea de consultas parciales.

Ejecución a la base

Observemos que hemos mudado a la función **get**. Cuando operamos con **QueryBuilder**, la consulta a la base de datos no se ejecuta hasta que hagamos uso de alguno de los métodos que muestra la **Tabla 2**.

Método	Descripción	Ejemplos
get	Es el método más utilizado; ejecutará lo que se encuentre establecido en ese momento y retornará siempre una **Collection** sin importar si la consulta devuelve uno o varios registros.	**DB::table('noticias')->get();** **Noticia::get();**
all	Se utiliza sólo en los modelos y devuelve todos los registros de la tabla asociada al mismo en una **Collection**.	**Noticia::all();**
first	Devuelve una única instancia de un **Model** o una clase **stdClass** cuando se invoca desde **DB**.	**DB::table('noticias')->get();** **Noticia::first();**
value	Devuelve el valor del primer registro de la columna especificada, aun cuando la consulta arroje varios registros. Si el valor es una fecha, devuelve una instancia de **Carbon**.	**DB::table('noticias')->value('titulo');** **Noticia::where('id','>','4')->value('titulo');**

Método	Descripción	Ejemplos
pluck	Devuelve una **Collection** pero únicamente establece los valores de la columna especificada.	**DB::table('noticias')->pluck('titulo');** **Noticia:: pluck('titulo');**
chunk	Recorre toda una tabla por partes, donde en el primer parámetro se especifica el tamaño de cada parte, y en el segundo se establece una función cuyo parámetro son los resultados de la consulta. Cuando la función devuelve false, se dejará de recorrer la tabla.	**DB::table('noticias')->orderBy('id')->chunk(100, function ($noticias) {** **foreach ($noticia as $n) {** **//** **}** **});**
count, max, min, avg	Se relacionan con las funciones de grupo de las bases de datos. Devuelven un único valor por los resultados del grupo de la consulta.	**DB::table('users')->count();** **Noticia:: count();**

Tabla 2. Métodos de QueryBuilder que ejecutan consultas a la base.

Paginador

En programación web es importante tener en cuenta la cantidad de información que enviamos al navegador debido a que, si enviamos mucha información, éste podría tardar en procesarla y en renderizar la vista de manera adecuada.

Procesamiento de grandes volúmenes

Chunk es muy útil para procesar grandes volúmenes de información. Es muy conveniente utilizarlo en seeders en los cuales busquemos realizar una migración de datos, ya sea entre distintas tablas o entre distintas bases. También es muy útil utilizarlo en comandos de Artisan. No obstante, no es aconsejable hacerlo en sistemas web, dado que el servidor no arrojará una respuesta hasta no haber procesado todos los registros, y podrá producirse un error de timeout.

Como consecuencia, habrá una mala experiencia de usuario.

A su vez, si contamos con una conexión no muy rápida, es probable que nuestra vista con imágenes tome un tiempo en cargar. Esto se debe, en principio, a la cantidad de noticias que estamos enviando y, en particular, porque se debe buscar un conjunto importante de imágenes en Lorempixel. Esta problemática general se resuelve implementando paginadores.

Siempre que naveguemos por sitios web que provean listados, encontraremos la información organizada en páginas.

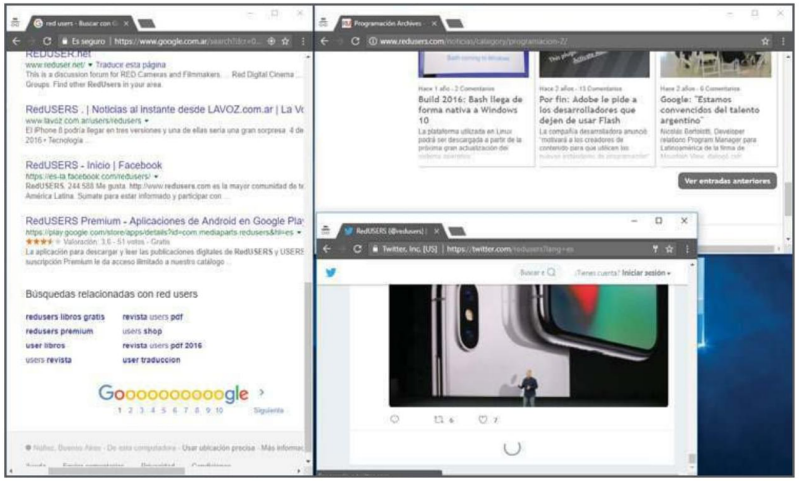


Figura 4. En algunos casos la división en páginas es muy evidente, y en otros viene camuflada en la interfaz.

Problemas generales y particulares

Es importante poder dedicar un tiempo adecuado al problema que intentamos resolver en nuestro sistema, separar bien todas las partes y discernir si son problemas particulares propios de nuestra lógica de negocios o generales. Este factor nos permite decidir la búsqueda de soluciones en la Web que puedan acoplarse a nuestro desarrollo para así ahorrarnos tiempo.

Debido a que Laravel convierte los problemas particulares en propios, contamos con una herramienta para cambiar nuestros listados actuales en páginas de forma muy simple.

Modifiquemos el controlador de la siguiente manera:

```
    public function conImagenes(){
      $noticias = Noticia::conImagenes()->paginate(10);
      return view('backend.noticia.index', ['noticias' =>
$noticias]);
    }
```

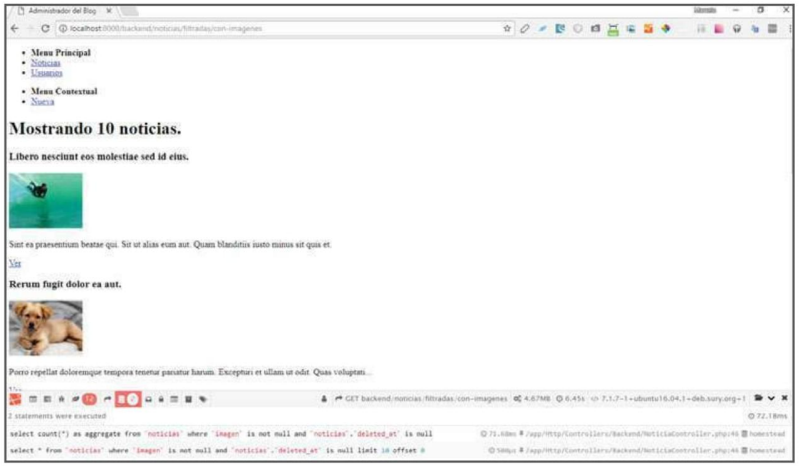


Figura 5. Recordemos siempre utilizar **Debugbar** para analizar el impacto de los cambios a medida que los realizamos.

¿Quién debe ejecutar la consulta?

Eloquent ofrece diversas posibilidades para la construcción de consultas a la base de datos y la ejecución. Es posible realizar la ejecución tanto en el modelo como en el controlador; el lugar donde corresponda hacerlo, es una decisión de cada desarrollador. Sin embargo, es conveniente concentrar la lógica de cómo realizar la consulta en el modelo, y la ejecución, en el controlador, ya que este último es quien sabe cómo devolver la información en la respuesta.

El método **paginate** debe realizar dos consultas a la base de datos: una para calcular la cantidad de páginas que ofrecerá y otra para obtener los registros de la página en cuestión.

Para renderizar las páginas debemos hacer uso de la vista del método **links**, lo cual podemos lograr mediante la etiqueta **{{ $noticias->links() }}**.

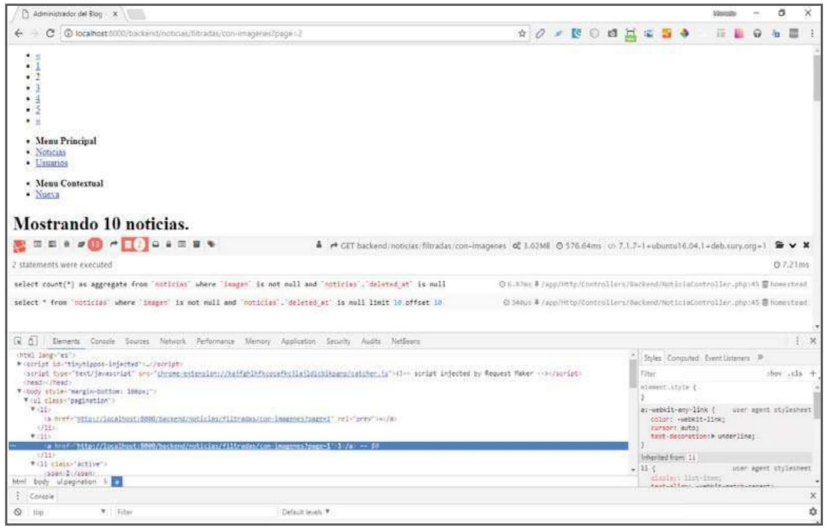


Figura 6. El método **links** renderiza una estructura HTML compatible con un componente de Bootstrap, **http://getbootstrap.com**.

Experiencia de usuario

La experiencia de usuario es una práctica que consiste en estudiar el conjunto de factores y elementos relativos a la interacción de un usuario con un producto. En desarrollo web, muchas veces suele asociarse al desarrollo de interfaces, pero es algo que debe tenerse en cuenta en todas las etapas del desarrollo, incluyendo la manera en la que se hacen las consultas, dado que éstas impactan significativamente en los tiempos y en la información que es capaz de manipular la interfaz.

Si inspeccionamos el código HTML con el navegador, veremos que **paginate** crea todas las páginas necesarias en función de los registros que devuelve la consulta de la clase **QueryBuilder**.

Cada página tiene la forma de un enlace con el parámetro **page**, el cual Laravel puede procesar para buscar los registros que correspondan. Esto puede verse a partir de la consulta que figura en Debugbar.

Para conocer todas las opciones que ofrece el paginador es recomendable visitar la documentación oficial disponible en **https://laravel.com/docs/5.5/pagination**.

Resumen Capítulo 08

En este capítulo comenzamos a reemplazar los mocks de nuestros controladores por modelos. Ejecutamos las primeras consultas a la base de datos y aprendimos a trabajar con colecciones. Luego estudiamos todos los métodos que ofrece **QueryBuilder**, junto con las ventajas de que los modelos funcionen también de esa manera. Además de ver las ventajas de realizar consultas parciales con el ORM, también vimos cómo acotar los resultados de búsqueda utilizando **QueryScopes**. Por último, introdujimos el paginador de Laravel, que nos permite segmentar la información para lograr una mejor experiencia de usuario.

ACTIVIDADES

Test de Autoevaluación

1. ¿Cuál es el resultado de ejecutar el método **all** en un modelo?
2. ¿En qué consiste una **Collection**?
3. ¿Qué significa que una **Collection** sea iterable?
4. ¿En qué consiste un **QueryBuilder**?
5. ¿Cuál es la relación entre **QueryBuilder** y los modelos?
6. ¿Qué es un scope?
7. ¿En qué consiste la idea de la creación de consultas parciales?
8. ¿De qué manera se ejecutan las consultas a la base de datos?
9. ¿Por qué es importante paginar los resultados?
10. ¿Por qué al implementar un paginador se realizan más consultas a la base?

Ejercicios prácticos

1. Cree una página donde puedan verse todas las categorías.
2. Cree una página donde sólo puedan verse las categorías que contengan noticias.
3. Modifique ambos listados de manera tal que el nombre de cada categoría sea un enlace que redireccione a una página que liste las noticias de esa categoría.

Relaciones entre modelos

09

Si hablamos de bases de datos relacionales, no podemos dejar de estudiar las relaciones. De la misma manera que los ORM mapean tablas en objetos, también pueden convertir las relaciones entre tablas, en relaciones entre objetos, y Eloquent nos brinda varias herramientas al respecto.

TIPOS DE RELACIONES

Existen diferentes relaciones entre tablas, y para cada una de ellas Laravel tiene una estrategia.

Uno a muchos

Actualmente tenemos una migración que relaciona las noticias con los usuarios mediante el campo **autor** de la tabla **noticias**. Esto quiere decir que para cada noticia tenemos un autor, y para cada autor tenemos muchas noticias.

Modifiquemos el modelo **Blog\Models\Noticia.php** para establecer esta relación incorporando el siguiente método:

```
public function creadaPor(){
    return $this->belongsTo('Blog\User', 'autor');
}
```

Este método nos permite acceder al objeto **User** vinculado al objeto **Noticia**, de manera tal que, por ejemplo, podemos agregar en la vista **backend.noticia.item** debajo del título la siguiente información:

```
<h4>Creada por {{ $noticia->creadaPor->name }}</h4>
```

El método **belongsTo** es el encargado de llevar la relación existente en la base de datos a los objetos de nuestro sistema. El primer parámetro es la clase con la que se relaciona. El segundo indica la clave foránea donde se encuentra el identificador de la tabla relacionada. En nuestro caso es obligatorio establecerlo, pero esto no siempre es necesario, ya que si el nombre de la clave terminase con el sufijo **_id**, Eloquent utilizaría el nombre del método de la relación para efectuar la consulta. Es decir, si nuestra columna se llamase **autor_id**, podríamos implementar la misma relación de la siguiente manera:

Es fundamental tener conocimientos sólidos sobre la normalización de bases de datos; de esta forma, seremos capaces de comprender y aplicar correctamente las relaciones.


```
public function autor(){
    return $this->belongsTo('Blog\User');
}
```

Existe un tercer parámetro que debemos utilizar únicamente cuando el nombre de la clave primaria de la tabla relacionada no es **id**.

Optimizar las consultas

Analicemos otra vez el impacto de estos cambios en la vista **backend.noticia.item**.

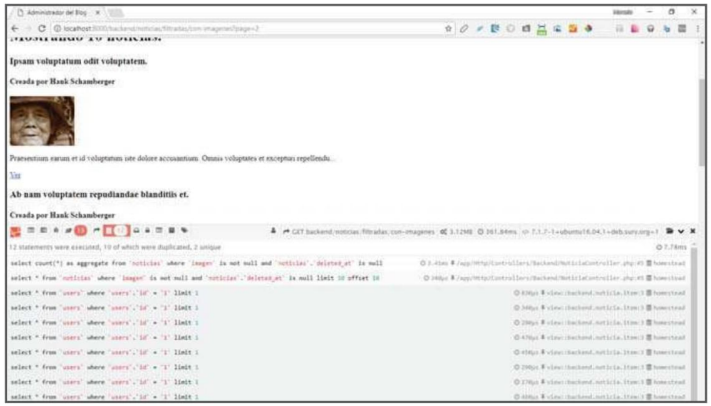


Figura 1. Observemos que la cantidad de consultas creció exponencialmente con este cambio.

El crecimiento de consultas se debe a que, para cada noticia, Eloquent ejecuta una consulta a la base de datos con el fin de buscar su autor. Una manera de reducir la cantidad de consultas es indicando a QueryBuilder que debemos traer los elementos relacionados, lo cual podemos hacer modificando el controlador de la siguiente forma:

```
$noticias = Noticia::with('creadaPor')->conImagenes()->paginate(10);
```

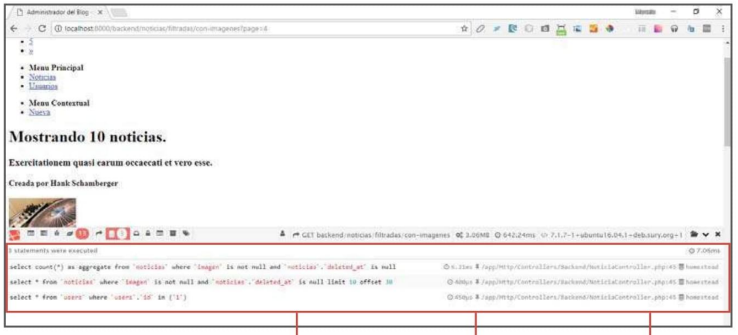


Figura 2. Con este cambio, la cantidad de consultas a la base de datos se redujo a tan sólo tres.

En este caso, todas las noticias están relacionadas con un solo autor. Si modificamos el Factory **NoticiaFactory** reemplazando la asignación del campo autor, la cual actualmente está hardcodeada en 1 por **Blog\User::inRandomOrder()->first()->id**, obtendremos un escenario más parecido al de un contexto productivo y veremos que la consulta cambiará nuevamente.

Hardcodear

El término hardcodear es un latiguillo popular entre los programadores y está relacionado con la práctica de hacer asignaciones de datos duros en lugares que deberían ser variables. En nuestro caso, teníamos **hardcodeado** el autor en 1, cuando lo correcto sería realizar una asignación aleatoria del autor. El hardcodeo es considerado una mala práctica ya que elimina la variabilidad que puede existir en un contexto.

Si modificamos **NoticiaFactory** e incluimos al seeder de Noticias en **DatabaseSeeder**, podemos regenerar toda la base con un simple comando **php artisan migrate:refresh --seed**.

Muchos a uno

A partir de una noticia, ya podemos obtener su autor, pero aún no podemos realizar la acción inversa, es decir, recuperar todas las noticias de un determinado autor. Para hacerlo, debemos agregar en el modelo **Blog\User** el siguiente método:

```
  public function noticias(){
      return $this->hasMany('Blog\Models\Noticia', 'au-
tor');
  }
```

A partir de este cambio, intentemos recuperar las noticias de un autor en Tinker.

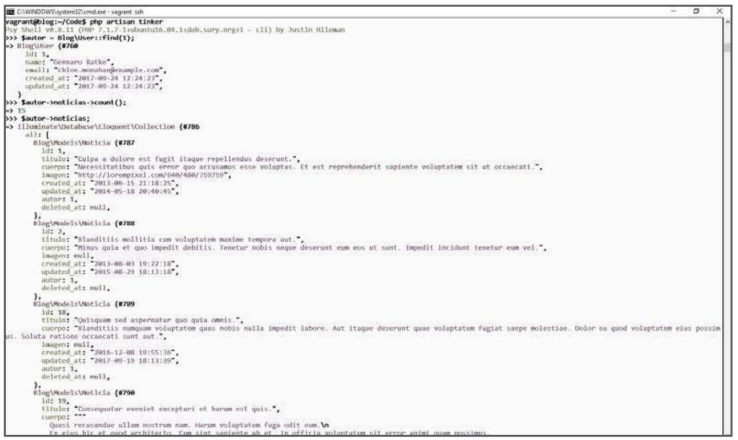


Figura 3. Recordemos que no siempre es necesario construir interfaces para analizar nuestros modelos.

Al igual que en el método **belongsTo**, en nuestro caso nos vemos obligados a establecer el segundo parámetro indicando el nombre de la columna que establece la relación. Si la columna se llamase **user_id** en lugar de autor, el segundo parámetro no sería necesario.

Es importante observar que todas las relaciones funcionan como QueryBuilders. Por lo tanto, hemos utilizado el método **count** para obtener la cantidad de noticias de un autor, lo cual podemos apreciar en Tinker en la imagen anterior.

Uno a uno

Vamos a crear una tabla para poder almacenar los avatars de nuestros usuarios. Cada usuario podrá tener un avatar, mientras que cada avatar pertenecerá a un usuario; en esto consiste la relación uno a uno.

Primero, generemos el modelo y la migración de la nueva entidad Avatar con el comando **php artisan make:model Blog\\Models\\Avatar --migration**. El método **up** de la migración quedará determinado de la siguiente forma:

```
Schema::create('avatars', function (Blueprint $table) {
    $table->increments('id');
    $table->string('img_location', 255);
    $table->boolean('active')->default(true);
    $table->integer('user_id')->unsigned();
    $table->foreign('user_id')
        ->references('id')
        ->on('users')
        ->onDelete('cascade')
        ->onUpdate('cascade');
    $table->timestamps();
});
```

El método **down** será mucho más simple:

```
Schema::dropIfExists('avatars');
```

En este caso, utilizaremos la relación **hasOne** en la clase **Blog\User** del siguiente modo:


```
public function avatar(){
  return $this->hasOne('Blog\Models\Avatar');
}
```

Por otro lado, en la clase **Blog\Models\Avatar** debemos indicar la relación con un método **belongsTo**, de la misma manera en que lo hacemos con las noticias y los usuarios:

```
public function user(){
  return $this->belongsTo('Blog\User');
}
```

```
C:\WINDOWS\system32\cmd.exe - vagrant ssh
vagrant@blog:~/Code$ php artisan tinker
Psy Shell v0.8.11 (PHP 7.1.7-1+ubuntu16.04.1+deb.sury.org+1 — cli) by Justin Hileman
>>> $a = new Blog\Models\Avatar;
=> Blog\Models\Avatar {#746}
>>> $a->img_location = 'http://lorempixel.com/400/200/people/7/';
=> "http://lorempixel.com/400/200/people/7/"
>>> $a->user_id=1;
=> 1
>>> $a->save();
=> true
>>> $u = Blog\User::find(1);
=> Blog\User {#761
     id: 1,
     name: "Prof. Jesus Walter",
     email: "irwin.nader@example.com",
     created_at: "2017-09-24 19:15:24",
     updated_at: "2017-09-29 22:28:22",
   }
>>> $u->avatar->img_location;
=> "http://lorempixel.com/400/200/people/7/"
>>>
```

Figura 4. Debido a que seguimos la estructura de base de datos que utiliza el framework, no fue necesario establecer parámetros adicionales.

¿Es necesario establecer siempre las relaciones de ambos lados?

La respuesta es no; sin embargo, es una buena práctica establecer las relaciones de ambos lados de las clases. Si trabajamos en un proyecto de mediana o gran escala con varios programadores, es posible que, por más que estemos implementando una relación en una clase para resolver una situación particular, otros programadores necesiten e, incluso, esperen que la relación inversa se encuentre establecida.

Muchos a muchos

Hasta el momento tenemos noticias que puedan organizarse por categorías. Las categorías ofrecen una manera jerárquica de clasificación, mientras que, si queremos catalogar las noticias de una forma horizontal, nos será conveniente utilizar etiquetas.

Una etiqueta podrá estar vinculada a muchas noticias, mientras que una noticia podrá tener muchas etiquetas. Esto difiere de la categoría debido a que en nuestro modelo hemos establecido que cada noticia pertenece sólo a una categoría.

En síntesis, nuestro diagrama de entidad relación quedará definido de la siguiente forma:

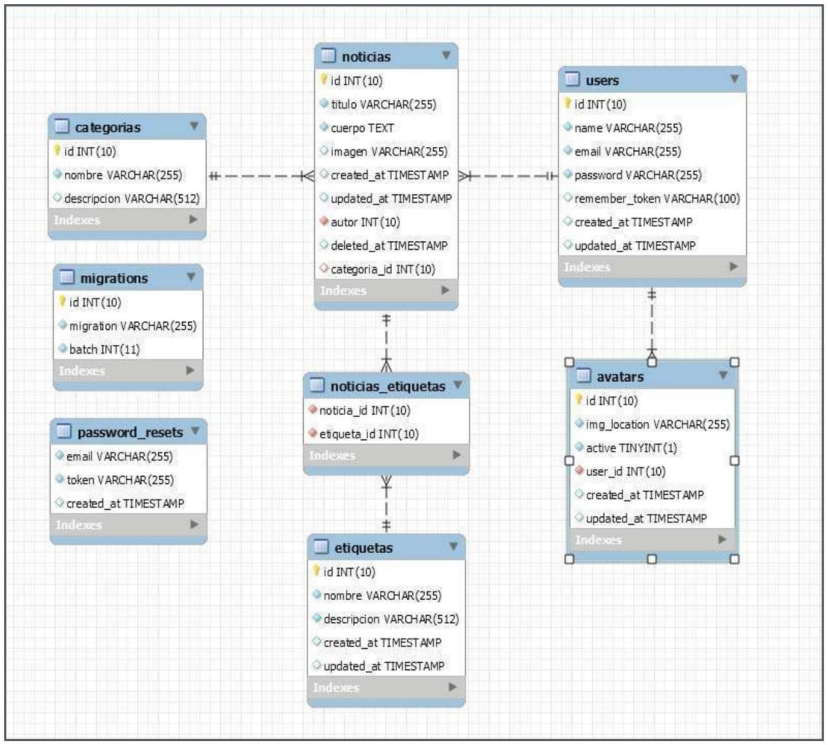


Figura 5. Recordemos que las relaciones de muchos a muchos demandan la creación de una tabla intermedia.

Primero, generemos el modelo, la migración, la fábrica y el controlador de la nueva entidad **Etiqueta** con el comando **php artisan make:model Blog\\Models\\Etiqueta --all**.

Los métodos **up** y **down** de la migración quedarán definidos tal como se muestra a continuación:

```
  public function up(){
      //Creamos la tabla etiquetas
      Schema::create('etiquetas', function (Blueprint $ta-
ble) {
          $table->increments('id');
          $table->string('nombre', 255);
          $table->string('descripcion', 512);
          $table->timestamps();
      });
      //Creamos la tabla para la relación muchos a muchos
      Schema::create('noticias_etiquetas', function (Blue-
print $table) {
        $table->integer('noticia_id')->unsigned();
        //Creamos clave foranea con la tabla noticias
        $table->foreign('noticia_id')
          ->references('id')
          ->on('noticias')
          ->onDelete('restrict')
          ->onUpdate('restrict');
        //Creamos clave foranea con la tabla etiquetas
        $table->integer('etiqueta_id')->unsigned();
        $table->foreign('etiqueta_id')
          ->references('id')
          ->on('etiquetas')
          ->onDelete('restrict')
          ->onUpdate('restrict');
      });
  }
  public function down(){
      Schema::dropIfExists('noticias_etiquetas');
      Schema::dropIfExists('etiquetas');
  }
```

Para reflejar esta relación haremos uso del método **belongsToMany()** tanto en el modelo **Blog\Models\Noticia** como en **Blog\Models\Etiqueta**.

En **Blog\Models\Noticia** lo implementaremos de esta forma:

```
  public function etiquetas(){
    return $this->belongsToMany('Blog\Models\Etiqueta',
'noticias_etiquetas');
    }
```

En **Blog\Models\Etiqueta** lo haremos del siguiente modo:

```
  public function noticias(){
    return $this->belongsToMany('Blog\Models\Noticia', 'no-
ticias_etiquetas');
  }
```

Nuevamente, debido a nuestra estructura, debemos establecer el segundo parámetro indicando la tabla donde se almacena la relación, también conocida como tabla **pivot**. Para no tener que utilizar este segundo parámetro, vamos a nombrar la tabla pivot con el nombre de las tablas de los modelos relacionados en orden alfabético; en nuestro caso debería llamarse **etiquetas_noticias**.

También contamos con un tercer parámetro para indicar el nombre de la columna del modelo desde el cual estamos creando la relación, y un cuarto parámetro para indicar el nombre de la columna hacia la cual estamos relacionando el modelo. En nuestro

¿Quién debe ejecutar la consulta?

Conocer el idioma inglés es una gran ventaja en general para la programación y, en especial, para el lenguaje SQL. Laravel simplifica aún más las cosas utilizando un idioma coloquial para establecer las relaciones. **hasOne** significa "tiene un"; **hasMany**, "tiene muchos"; **belongsTo**, "pertenece a un"; y **belongsToMany**, "pertenece a muchos". A partir de estos nombres, es muy sencillo comprender conceptualmente las relaciones que existen entre los objetos.

caso no es necesario definirlos, ya que, por default, intentará utilizar el nombre de la clase con el sufijo **_id** tanto para la clave local como para la foránea, y debido a que nuestros campos se llaman **noticia_id** y **etiqueta_id**, esa asignación no es necesaria.

Almacenar objetos relacionados

Vamos a crear un seeder para las etiquetas con el siguiente contenido:

```
  //Generamos las etiquetas
  factory(Blog\Models\Etiqueta::class, 20)->create();

     Blog\Models\Noticia::chunk(2, function($noticias){
        foreach($noticias as $n){
          //Buscamos entre 1 y 3 etiquetas cada dos noti-
cias
          $etiquetasRandom = Blog\Models\
Etiqueta::inRandomOrder()->limit(rand(1, 3))->get();
          //Asignamos las etiquetas a las noticias
          $n->etiquetas()->saveMany($etiquetasRandom);
     }
  });
```

Observemos que en el seeder ya estamos haciendo uso de la relación que establecimos entre etiquetas y noticias.

Cuando tenemos relaciones entre objetos, podemos utilizar diversos métodos con el fin de guardar los objetos relacionados; en este caso, estamos utilizando el método **saveMany**.

Sin embargo, hay que tener en cuenta que estos métodos se obtienen a partir de una instancia de otro objeto, el cual mantiene la relación; en nuestro caso el objeto es **BelongsToMany**. Para obtenerlo, debemos invocar al método donde fijamos la relación, y lo hacemos con **$n->etiquetas()**.

Es importante hacer la distinción de que podemos utilizar la relación tanto como una **propiedad**, como lo hicimos en la vista **backend.noticia.item** para la relación entre noticias y usuarios; o como un **método**, de la manera en la que lo estamos haciendo en este seeder.

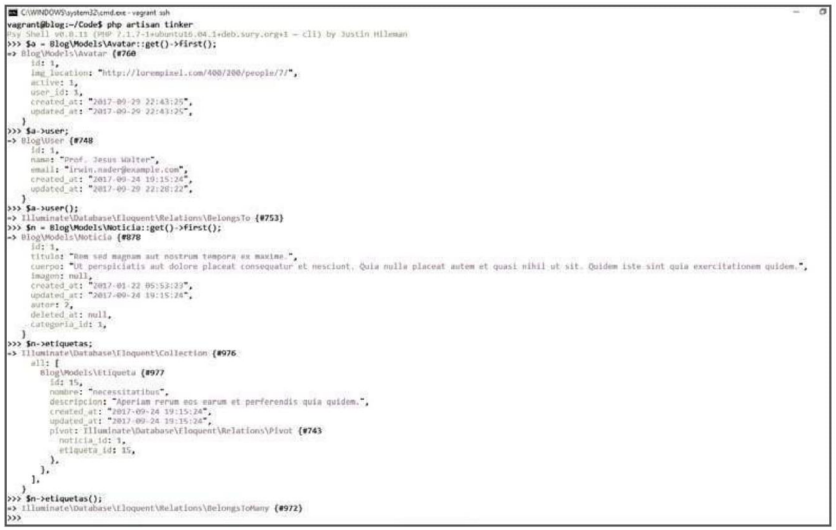


Figura 6. Dependiendo de la relación establecida entre los objetos, se retornarán diferentes instancias de relaciones.

```
C:\WINDOWS\system32\cmd.exe - vagrant ssh
>>> $a = new Blog\Models\Avatar();
=> Blog\Models\Avatar {#759}
>>> $a->img_location = 'https://es.gravatar.com';
=> "https://es.gravatar.com"
>>> $u = Blog\User::inRandomOrder()->get()->first();
=> Blog\User {#774
     id: 4,
     name: "Ms. Amelie Koss",
     email: "gustave.schuppe@example.net",
     created_at: "2017-09-24 19:15:24",
     updated_at: "2017-09-24 19:15:24",
   }
>>> $u->name = "Usuario con gravatar";
=> "Usuario con gravatar"
>>> $u->avatar()->save($a);
=> Blog\Models\Avatar {#759
     img_location: "https://es.gravatar.com",
     user_id: 4,
     updated_at: "2017-09-29 23:48:21",
     created_at: "2017-09-29 23:48:21",
     id: 5,
   }
>>> $a->id;
=> 5
>>> $a->user_id;
=> 4
>>> $u;
=> Blog\User {#774
     id: 4,
     name: "Usuario con gravatar",
     email: "gustave.schuppe@example.net",
     created_at: "2017-09-24 19:15:24",
     updated_at: "2017-09-24 19:15:24",
   }
>>>
```

Para relaciones del tipo **hasOne** podemos, simplemente, utilizar el método **save**.

Además de realizar la comprobación por Tinker, como muestra la **Figura 7**, es recomendable observar el impacto de los cambios en la base de datos y comprobar que las referencias en los registros almacenados se generaron de manera correcta.

Figura 7. Al obtener la instancia de la relación y utilizar **save**, debemos pasar el objeto que vincularemos.

Observemos que al ejecutar **save** hemos:

- Asociado al **Avatar** con el objeto **User**.
- Insertado el objeto **Avatar**, junto con su referencia.
- Actualizado el objeto **User**.

Es decir, Eloquent detecta que **Blog\Models\Avatar** ha sido recientemente creado en memoria, pero aún no existe en la base de datos, por lo tanto, agrega la referencia con **User** y luego inserta el registro en la tabla **avatars**. Por último, ejecuta un update en **users** para almacenar el cambio efectuado.

En caso de utilizar una relación **belongsTo**, debemos almacenar los objetos con **associate()**.

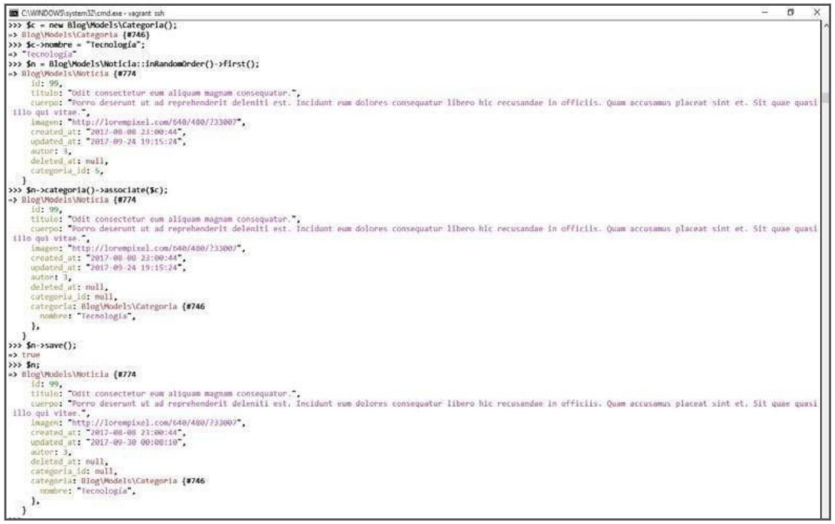


Figura 8. En este caso, Eloquent realiza la asociación en memoria, pero no guarda el objeto **Categoria**.

En el caso del método **associate**, Eloquent sólo almacenará la relación con **Blog\Models\Categoria** si ésta ya posee un **id**, es decir, si fue almacenada previamente.

Tablas pivot

Como ya sabemos, una tabla pivot se crea para generar relaciones de muchos a muchos entre dos tablas. En nuestro blog tenemos la tabla pivot **noticias_etiquetas**, que únicamente vincula las noticias con las etiquetas. En las tablas pivot, también podemos agregar información que sea útil para la relación que mantiene esa tabla.

Vamos a modificar la tabla pivot generando una nueva migración con el comando **php artisan make:migration add_ user_column _noticias_ etiquetas** para agregar el usuario que vincula la noticia con la etiqueta, y agregaremos campos timestamps.

El método **up** quedará establecido de la siguiente forma:

```
  //Por default asignamos al usuario con id 1
  Schema::table('noticias_etiquetas', function (Blueprint
$table) {
        $table->integer('user_id')->unsigned()->default(1);
        $table->foreign('user_id')
          ->references('id')
          ->on('users')
          ->onDelete('restrict')
          ->onUpdate('restrict');
        $table->timestamps();
  });
```

Luego de ejecutar la migración, modificamos la relación para que se mantengan actualizados los campos timestamps incorporando, al final del método de la relación, el método **withTimestamps()**.

Por ejemplo, en **Blog\Models\Etiqueta.php** el método quedará de la siguiente manera:

Más sobre QueryBuilder

Todas las relaciones funcionan también como **QueryBuilders**, por lo que podemos realizar consultas a través de ellas, como por ejemplo **Noticia::find(1)->etiquetas()->where('etiqueta_id','=',1);**. También podemos contar los registros relacionados invocando al método **withCount**, por ejemplo, **$n = Noticia::find(1)->withCount('etiquetas')->get();**.


```
  public function noticias(){
    return $this->belongsToMany('Blog\Models\Noticia', 'no-
ticias_etiquetas')->withTimestamps();
  }
```

A partir de este cambio, podremos acceder a los datos de la tabla pivot desde cualquier objeto de la clase **Blog\Models\Etiqueta** o **Blog\ Models\Noticia**.

Los campos timestamps serán inicializados en **null** debido a que ejecutamos **EtiquetasTableSeeder** antes de realizar el cambio. Por lo tanto, podemos o bien eliminar los registros de **noticias_etiquetas** y ejecutar **EtiquetasTableSeeder** nuevamente, o debido a que nos encontramos en un contexto de desarrollo, podemos ejecutar **php artisan migrate:refresh --seed**, lo cual regenerará la base de datos y ejecutará todos los seeds establecidos en **DatabaseSeeder.php**.

Figura 9. Observemos que, en la propiedad **pivot**, sólo se recuperan los campos **id** y los timestamps.

Para poder recuperar otros campos de la tabla **pivot** debemos especificarlos en la relación, utilizando el método **withPivot()**. Por ejemplo, en el modelo **Blog\Models\Noticia** la relación quedará establecida así:

```
  public function etiquetas(){
    return $this->belongsToMany('Blog\Models\Etiqueta',
'noticias_etiquetas')
      ->withPivot('user_id')
      ->withTimestamps();
  }
```

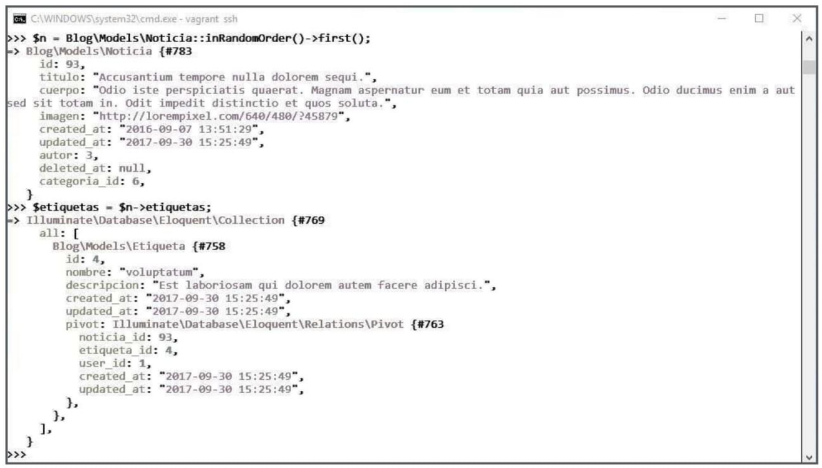


Figura 10. Recordemos que debemos hacer lo mismo del otro lado de la relación, es decir, en el modelo **Blog\Models\Etiqueta**.

Ahora podemos obtener la referencia, es decir, el **user_id** que pertenece al usuario que asignó la etiqueta. No obstante, no podemos obtener el objeto **Blog\User**, lo cual sería más apropiado en el contexto que estamos estudiando. Para poder obtener una relación desde una tabla intermedia, no nos quedará más remedio que crear un modelo para la clase **pivot**, lo cual haremos a través del comando **php artisan make:model Blog\\Models\\NoticiaEtiqueta**.

Es importante tener en cuenta que los modelos pivot no heredarán desde **Illuminate\Database\Eloquent\Model** sino que lo harán desde **Illuminate\Database\Eloquent\Relations\Pivot**, por lo tanto, **Blog\Models\NoticiaEtiqueta** quedará conformado del siguiente modo:

```
  namespace Blog\Models;

  use Illuminate\Database\Eloquent\Relations\Pivot;

  class NoticiaEtiqueta extends Pivot
  {
      public function user(){
        return $this->belongsTo('Blog\User');
      }
  }
```

Habiendo generado esta clase, debemos indicar en las tablas adyacentes que usaremos este nuevo modelo pivot, lo cual haremos modificando **Blog\Models\Noticia**:

```
  public function etiquetas(){
    return $this->belongsToMany('Blog\Models\Etiqueta',
'noticias_etiquetas')
      ->withPivot('user_id')
      ->withTimestamps()
      ->using('Blog\Models\NoticiaEtiqueta');
  }
```

Y lo mismo debemos hacer en **Blog\Models\Etiqueta**:

```
  public function noticias(){
    return $this->belongsToMany('Blog\Models\Noticia', 'no-
ticias_etiquetas')
      ->withPivot('user_id')
      ->withTimestamps()
      ->using('Blog\Models\NoticiaEtiqueta');
  }
```

A partir de este cambio, podremos obtener la instancia de la clase **Blog\User**, es decir, el usuario que asignó una etiqueta a una noticia.

Relaciones distantes

Las relaciones distantes son aquellas relaciones indirectas que existen entre entidades y para las cuales Eloquent nos permite establecer atajos utilizando el método **hasManyThrough()**.

Si observamos nuestro diagrama de entidad-relación, veremos que existe una relación indirecta entre las categorías y los usuarios. Ésta se da a partir de las noticias, debido a que todas las noticias contienen un autor, el cual es un usuario, y a su vez, todas las noticias pertenecen a una categoría.

En consecuencia, podemos decir que las categorías tienen usuarios a través de las noticias.

Para establecer esta relación debemos agregar el siguiente método en **Blog\Model\Categoria.php**:

```
public function users(){
  return $this->hasManyThrough(
    'Blog\User',
    'Blog\Models\Noticia',
    'categoria_id',
    'id',
    'id',
    'autor'
  );
}
```

El método recibe los siguientes parámetros:

- El nombre de la clase destino con la que se generará la relación.
- El nombre de la clase de la entidad intermedia.
- La clave de la entidad origen en la tabla intermedia.
- La clave de la entidad intermedia en la tabla destino.
- La clave de la tabla de la clase de origen.
- La clave de la tabla destino en la tabla intermedia.

Figura 11. Dado que cada categoría pueda tener muchas noticias, entonces cada categoría tendrá muchos usuarios.

Resumen Capítulo 09

En este capítulo estudiamos los principales tipos de relaciones entre clases que pueden implementarse con Eloquent. Partimos estableciendo relaciones uno a muchos que ya existían en nuestro modelo de datos y luego implementamos la relación inversa. Más adelante mejoramos la base de datos para que las noticias del blog pudieran tener etiquetas y con esto instauramos relaciones muchos a muchos. Luego profundizamos en las tablas pivot y las diferentes estrategias que podemos adoptar con ellas. Por último, generamos una relación distante entre las categorías y los usuarios del blog.

ACTIVIDADES

Test de Autoevaluación

1. ¿En qué consiste una relación entre clases?
2. ¿Cómo se representan las relaciones entre clases en Eloquent?
3. ¿Es necesario tener una estructura y/o nomenclatura específica para implementar las relaciones?
4. ¿Qué sucede si, al establecer una relación, no se optimizan las consultas?
5. ¿De qué manera es posible almacenar objetos que se encuentran vinculados?
6. ¿En qué consiste una tabla pivot?
7. ¿Es necesario crear un modelo pivot para establecer relaciones muchos a muchos?
8. ¿Siempre se obtienen todos los valores de una tabla pivot?
9. ¿En qué consisten las relaciones distantes?
10. ¿Cuáles son las principales relaciones que es posible mapear en Eloquent?

Ejercicios prácticos

1. Cree una tabla para almacenar comentarios que podrán realizar los usuarios en las noticias.
2. Establezca las relaciones necesarias entre las clases **Comentario**, **Noticia** y **Usuario**.
3. Identifique y cree relaciones distantes generadas a partir de esta nueva entidad.

10 Interfaces de usuario

Desde la concepción de la idea de la Web 2.0, las tecnologías y los lenguajes fueron evolucionando para incorporar elementos tendientes a mejorar y enriquecer la experiencia del usuario a partir de las interfaces. Laravel no está exento de esta situación, y en este capítulo estudiaremos cómo trabajar con el framework para generar interfaces ricas, elegantes y amigables.

RECURSOS

Las tecnologías se reconstruyen todo el tiempo para mejorar y enriquecer las interfaces gráficas, de manera tal que se pueda ofrecer a los usuarios una experiencia limpia, uniforme, simple y agradable.

Junto con las tecnologías, el mercado va aumentando las exigencias para encontrar perfiles capaces de adaptarse a los cambios tecnológicos y a las nuevas tendencias. En este aspecto, trabajar con Laravel presenta varias ventajas, ya que constantemente se va adaptando a las nuevas tendencias.

Para construir interfaces de usuario ricas tanto gráficamente como en funcionalidad, HTML no es suficiente, y es necesario complementar Laravel con otros recursos web.

Para agruparlos, contamos con una carpeta **resources**, y es allí mismo donde guardaremos el resto de los recursos que vayamos a usar. Recordemos que también almacenamos en esta carpeta las vistas, por lo tanto, todo lo vinculado a las interfaces se encontrará en el mismo lugar.

LARAVEL MIX

Antes de comenzar a trabajar con Laravel Mix, es necesario comprender un poco acerca de JavaScript. Se trata de un lenguaje de programación que se ejecuta del lado del cliente, es decir, en el navegador. Muchos de los recursos que utilizaremos para construir interfaces estarán vinculados a este lenguaje, el cual ha

GitHut

En el sitio web **http://githut.info** podemos encontrar estadísticas relacionadas con los repositorios de Github y los lenguajes de programación que utilizan. Es un muy buen lugar para analizar la historia de los lenguajes y la popularidad conforme a los avances de cada año. También es muy recomendable visitar **https://octoverse.github.com**, que nos dará estadísticas sobre la actividad del último año en Github.

incrementado su popularidad en los últimos años para llegar a convertirse en el más usado según Github.

Si bien nació para ser ejecutado en los navegadores web, hoy existen diferentes frameworks que permiten utilizar JavaScript del lado del servidor, entre los cuales cabe destacar NodeJS.

Dado su exponencial crecimiento, el mercado actual cuenta con una amplia gama de frameworks y librerías en JavaScript para diferentes propósitos. Para trabajar con ellas en nuestras aplicaciones Laravel contamos con **Laravel Mix**, una capa intermedia entre otras herramientas JavaScript que también debemos aprender a utilizar: ellas son NPM y WebPack.

Introducción de JavaScript

La manera más simple y rápida de introducir código JavaScript es insertando etiquetas **inline** en el código HTML. Vamos a generar una función JavaScript que será la base para crear una funcionalidad que permita a los usuarios guardar sus noticias favoritas. Modifiquemos primero el archivo **routes/web.php** para introducir el Resource **NoticiaController**, eliminando todos los métodos de escritura, de la siguiente manera:

```
  Route::namespace('Frontend')->name('frontend.')-
>group(function () {
      Route::resource('noticia', 'NoticiaController', ['ex-
cept' => [
      'create', 'store', 'update', 'destroy'
      ]]);
  });
```

Luego, debemos modificar **Blog\Http\Controllers\Frontend\NoticiaController.php** agregando el método **index**. Recordemos añadir **use Blog\Models\Noticia;** para acceder al modelo **Noticia**:

```
  public function index(){
    $noticias = Noticia::paginate();
```

```
    return view('frontend.noticia.index', ['noticias' =>
$noticias]);
  }
```

También crearemos un layout en **resources\views\frontend\layouts\main.blade.php**:

```
  <!DOCTYPE html>
  <html lang="es">
    <head>
      <meta charset="utf-8">
      <title>Blog con Laravel - @yield('title')</title>
    </head>
    <body>
      @yield('content')
    </body>
  </html>
```

Y por último, crearemos una vista para listar las noticias de nuestro blog:

```
  @extends('frontend.layouts.main')
  @section('title', 'Últimas noticias')
  @section('content')
    @foreach($noticias as $noticia)
      <div>
        <h3>{{ $noticia->titulo }}</h3>
        @if(isset($noticia->imagen))
          <img src="{{$noticia->imagen}}" />
        @endif
        <p>{{ $noticia->cuerpo }}</p>
        <button class="butFav" data-noticia-id="{{ $noti-
cia->id }}">*</button>
      </div>
    @endforeach
```



```
    <!-- Inicio de código javascript -->
    <script type="text/javascript">
      var buttons = document.getElementsByClassName("butF
av");
      for (i = 0; i < buttons.length; i++){
        buttons[i].onclick = function(noticiaId){
          alert('Noticia agregada en favoritos');
        }
      }
    </script>
    <!-- Fin de código javascript -->
  {{ $noticias->links() }}
  @endsection
```

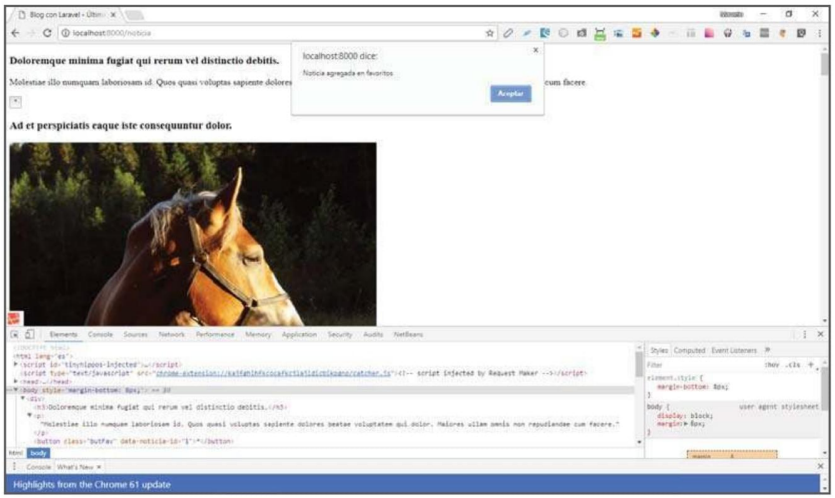


Figura 1. Al presionar el botón, veremos una ventana del tipo alert, propia del navegador, que es disparada por el código JavaScript que acabamos de introducir.

El código JavaScript inline está considerado una muy mala práctica. Esto se debe, en parte, a que tenemos como resultado archivos donde se mezclan etiquetas HTML, etiquetas de Blade y código JavaScript, lo cual podemos apreciar en este breve ejemplo.

Tengamos en cuenta que si nuestra aplicación escala y seguimos implementando código inline, lo más probable es que tengamos como resultado archivos con código irreconocible.

Una estrategia para mitigar este problema es separar la lógica JavaScript en un archivo e incluirlo con una etiqueta **<script>**. Para hacerlo, vamos a crear en la carpeta **public/js/** el archivo **favoritos.js** con el mismo código inline que hemos introducido previamente en **index.blade.php**, pero sin las etiquetas scripts. Luego, en el lugar de la vista reemplazamos las etiquetas **<script>** de la siguiente forma:

<script type="text/javascript" src="js/favoritos.js"></script>. Si refrescamos la página, obtendremos los mismos resultados, pero con un código más legible y ordenado.

Si queremos que esta funcionalidad esté disponible en todas las páginas del sistema y no sólo en esta vista en particular, será conveniente implementar la etiqueta **<script>** en **resources/views/frontends/layouts/main.blade.php** en vez de hacerlo en la vista.

NPM

De la misma manera en la que hemos escrito esta pequeña función, existen muchas librerías disponibles en la Web que nos permiten incorporar funcionalidad en nuestras interfaces. Buscar, descargar e instalar todas estas librerías puede tornarse una tarea compleja; no obstante, teniendo NPM, esto se vuelve mucho más fácil.

NPM es el equivalente a Composer, pero para librerías desarrolladas con JavaScript. Si utilizamos Homestead, no debemos instalarlo; de lo contrario, lo descargamos e instalamos **NodeJs** desde **https://nodejs.org/en/download**. La instalación de NodeJs incluye NPM, ya que

JavaScript nativo

JavaScript es un lenguaje que adquirió popularidad en un lapso de tiempo muy breve, lo que produjo que cada navegador lo interprete de una manera diferente. Utilizar JavaScript sin ningún framework es arriesgado, ya que hay que conocer muy bien las características de cada navegador para poder generar código que funcione. Por esta razón, resulta conveniente recurrir a algún framework; uno de los más utilizados es jQuery, https://jquery.com.

es el manejador de paquetes de NodeJs, y si bien no haremos uso de NodeJs, aprovecharemos todas las ventajas que trae su administrador de dependencias. Una vez que instalamos NodeJs, podemos ejecutar los comandos **node -v** y **npm -v** para asegurarnos de que ambos estén accesibles desde el entorno.

Habiendo asegurado la instalación de NPM, simplemente ejecutando **npm install** se instalarán los paquetes JavaScript necesarios para utilizar Laravel Mix. A partir de este comando, se creará una carpeta denominada **node_modules** con todas las librerías JavaScript del proyecto.

Figura 2. Al igual que con **composer install**, es normal que **npm install** demore algunos minutos la primera vez que se ejecuta.

De la misma manera que en Composer definimos los paquetes a utilizar en el archivo **composer.json**, NPM utiliza un archivo **package.json** con los mismos propósitos; al igual que con Composer, dicho archivo forma parte del framework.

WebPack

Ahora que tenemos una gran cantidad de recursos JavaScript disponibles en la carpeta **node_modules**, podemos empezar a incluirlos en la interfaz. Siguiendo la misma estrategia con la cual incluimos el primer archivo JavaScript, tendríamos que copiar los recursos desde **node_modules** hacia **public** y, luego, incluir todas las etiquetas **<script>** correspondientes. Esta aproximación no es muy conveniente, por cuestiones tanto de tiempo como de performance.

Cada vez que utilicemos un recurso JavaScript para el desarrollo de una interfaz, será preciso evaluar el impacto que tendrá su implementación en la performance.

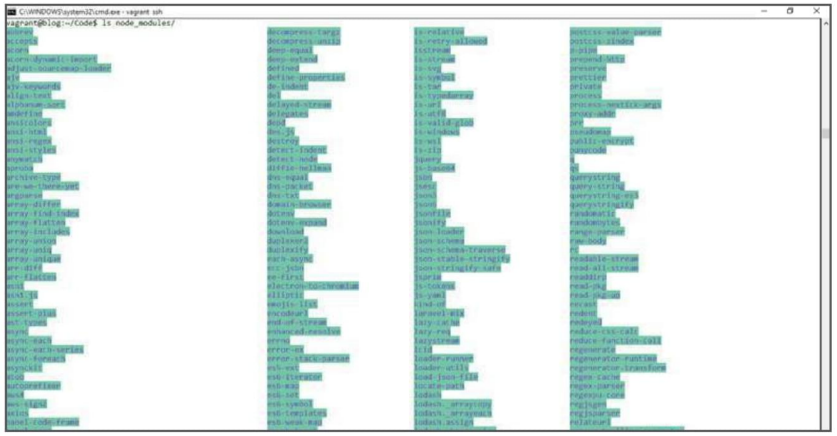


Figura 3. En la carpeta **node_modules** encontraremos una gran cantidad de librerías, pero no todas son necesarias para las interfaces.

Homestead y Windows hosts

NPM utiliza enlaces simbólicos para la instalación de varios paquetes. Los enlaces simbólicos son una manera de referenciar carpetas en el sistema de archivos, y para que funcione correctamente en Windows es necesario iniciar la interfaz de línea de comando con permisos de Administrador e iniciar la máquina virtual con dicha consola. Una vez levantada la máquina, iniciando **vagrant ssh** es posible ejecutar el comando **npm install** sin problemas; en caso contrario, habrá errores en la creación de enlaces simbólicos.

Cada vez que cargamos una página web con múltiples recursos, estamos generando peticiones entre el navegador y el servidor. Por ejemplo, si tenemos 20 tags del tipo **<script type="text/javascript"> src="miarchivo.js"/>**, estaremos realizando 20 peticiones para cargar la página.

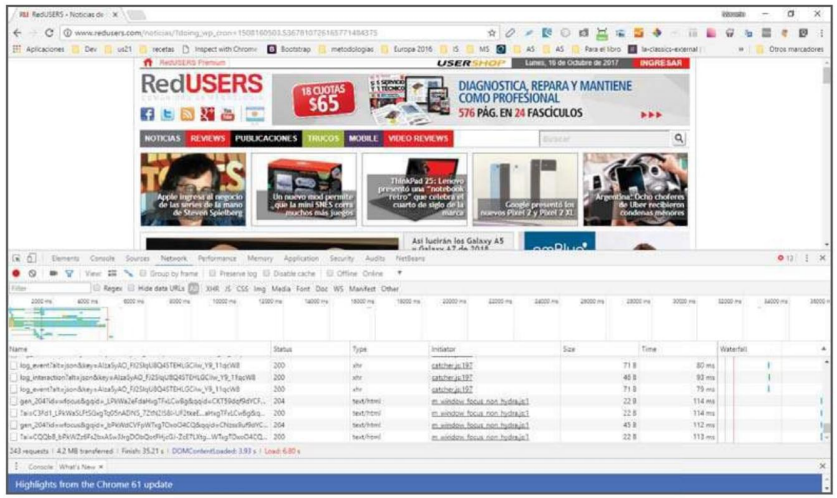


Figura 4. Las herramientas para desarrolladores que ofrecen los navegadores permiten ver todas las peticiones que se realizan para cargar una página web.

WebPack permite compilar todas las librerías que utilicemos en el proyecto en uno o varios archivos, según la forma en la que lo establezcamos. Por default, todos los recursos de un mismo tipo

Documentación oficial

La documentación oficial de Laravel nos ofrece una sección completa que contiene todas las funcionalidades que proporciona **mix**, con varios ejemplos. Esta sección se encuentra disponible en la dirección web **https://laravel.com/docs/5.5/mix**. Debemos considerar que, en las versiones anteriores a Laravel 5.4 se utilizaba otra herramienta, aunque muy similar, denominada **elixir**, cuya documentación se encuentra en la dirección web **https://laravel.com/docs/5.3/elixir**.

siempre se combinan en un único archivo, y esto se logra mediante el comando **npm run development**.

Figura 5. El comando **npm run development** generó los archivos **js/app.js** y **css/app.css** y copió archivos de fuentes.

Incluir los archivos en el código también es muy simple, ya que Laravel Mix nos brinda una herramienta para hacerlo: simplemente, modificamos el archivo **resources\views\frontend\layouts\main.blade.php** de la siguiente forma:

```
  <!DOCTYPE html>
  <html lang="es">
    <head>
      <meta charset="utf-8">
      <title>Blog con Laravel - @yield('title')</title>
      <!-- Incluimos todos los recursos css -->
      <link rel="stylesheet" href="{{ mix('/css/app.css')
}}">
    </head>
    <body>
      @yield('content')
    </body>
    <!-- Incluimos todos los recursos javascript -->
```



```
    <script type="text/javascript" src="{{ mix('/js/app.
js') }}"></script>
  </html>
```

Cuando recargamos la página, vemos que se ha incorporado un nuevo diseño, y esto se debe a que, si inspeccionamos el archivo **package.json**, observaremos que Laravel incorpora el framework Bootstrap, de manera tal que WebPack también lo incluye. Profundizaremos más en Bootstrap en la sección dedicada a este tema.

De esta manera, hemos mejorado la performance, en parte, haciendo el código HTML más liviano, y en parte, logrando que el navegador realice un único request para todos los JavaScript y CSS.

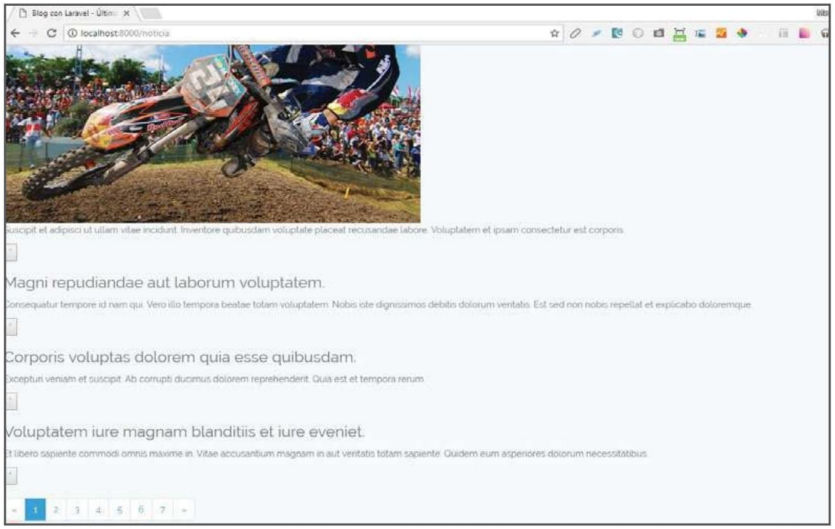


Figura 6. Recordemos que el paginador de Laravel es compatible con Bootstrap, por eso se genera este componente de manera automática.

WebPack es muy importante debido a que Laravel Mix consta de una capa de configuración por encima de él e incluye varias utilidades que facilitan su implementación. Entre ellas, está la función **mix**, que devuelve la ruta hacia el archivo de recursos compilado.

Configuración de WebPack

Toda la configuración de WebPack se encuentra en el archivo **webpack.mix.js**, en la carpeta raíz del proyecto. Si inspeccionamos el archivo, encontraremos que primero se incluye la incorporación de mix mediante el comando **let mix = require('laravel-mix');**, el cual forma parte de la sintaxis de Ecmascript 2015, es decir, de la última versión disponible de JavaScript. Luego aparece el siguiente código:

```
mix.js(`resources/assets/js/app.js', `public/js')
   .sass(`resources/assets/sass/app.scss', `public/css');
```

Éstas son dos tareas por separado: por un lado, la compilación de archivos JavaScript a partir de lo establecido en **resources/assets/js/app.js**, y por otro, la compilación de los recursos sass en css.

Vamos a agregar a la compilación el archivo **public/js/favoritos.js**. Para hacerlo, primero vamos a moverlo a la ubicación **resources/assets/js/favoritos.js** y, luego, modificaremos **resources/assets/js/app.js** agregando la línea **require('./favoritos');** debajo de **require('./bootstrap');**.

Si aún no lo hemos hecho, eliminamos la etiqueta **<script type="text/javascript" src="js/favoritos.js"></script>** de la vista y recargamos la página.

Podemos observar que, cuando intentamos presionar un botón, no se activa el funcionamiento, porque debemos volver a generar el archivo **js/app.js** ante cada cambio.

Si volvemos a ejecutar **npm run development**, veremos otra vez el cambio aplicado.

WebPack

WebPack es una herramienta muy completa. Es recomendable visitar su sitio web en **https://webpack.js.org** y, en particular, la sección dedicada a las guías, **https://webpack.js.org/guides**, que presenta tutoriales paso a paso para implementar diferentes estrategias destinadas a compactar y distribuir los recursos necesarios para las interfaces de usuario.

Comandos de mix

Generar el archivo **js/app.js** puede resultar incómodo si nos encontramos desarrollando en JavaScript. Para evitar esto, podemos utilizar el comando **npm run watch-poll**, que irá observando los cambios realizados en los recursos y regenerando el archivo **js/app.js** en cada modificación.

Vamos a comprobarlo iniciando el comando y luego modificando el código de favoritos por medio del framework JQuery, ya que éste nos brindará una mayor adaptación a diversos navegadores:

```
$('.butFav').click(function(item){
  alert('Noticia ' + $(this).attr("data-noticia-id") + ' agregada en favoritos');
});
```

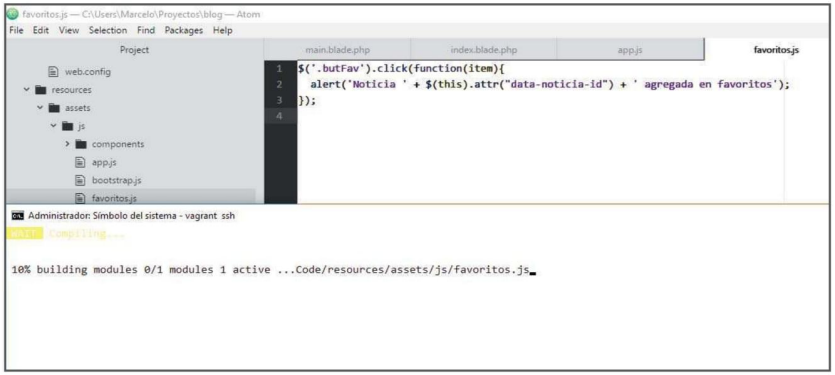


Figura 7. La línea de comando muestra que se detectó el cambio en el recurso **favoritos.js**; por eso se genera nuevamente **app.js**.

Podemos comprobar los cambios recargando la página y presionando el botón otra vez. En este caso, hemos realizado una leve modificación, ya que recuperamos el atributo **noticia-data-id** que hemos asignado a cada botón y lo mostramos en el componente **alert**.

No obstante, si queremos ver cambios en el código fuente, una manera muy útil es hacer uso de las herramientas para desarrolladores que ofrecen los navegadores. Buscando la pestaña **sources** podemos encontrar el archivo **app.js**.

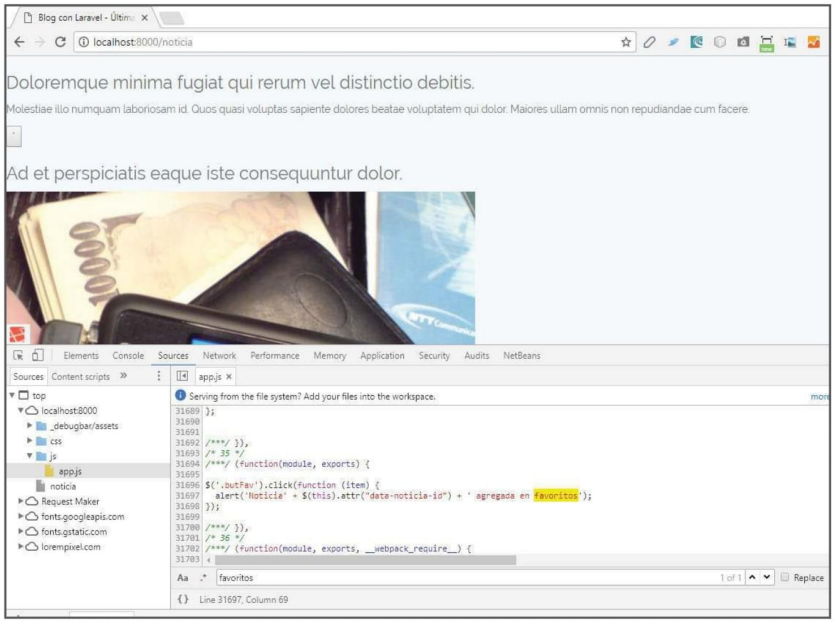


Figura 8. Resulta difícil encontrar nuestro código debido a que en el mismo archivo también está el de todos los demás resources JavaScript.

Para buscar nuestro código de manera más simple, podemos utilizar la función **sourceMaps().** Ésta hace que WebPack introduzca información adicional en las herramientas para desarrolladores de los navegadores según la ubicación de los archivos utilizados para generar **js/app.js**. Dicha función tiene un costo en performance y no se recomienda para contextos productivos; no obstante, es muy útil en un entorno de desarrollo.

Para implementarla, modificamos el archivo **webpack.mix.js** de la siguiente forma:

```
  mix.js('resources/assets/js/app.js', 'public/js').source-
Maps();
  mix.sass('resources/assets/sass/app.scss', 'public/css');
```

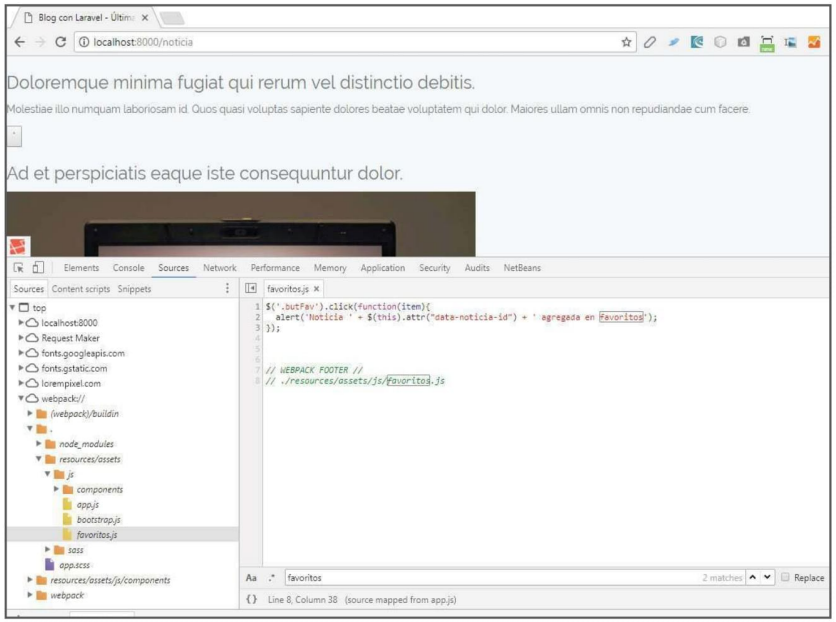



Figura 9. Con esta nueva carpeta, podemos ver todos los recursos que estamos utilizando para generar **js/app.js**.

SourceMaps sólo trabaja en la interfaz del navegador. Sin embargo, en la ejecución, nuestro código sigue integrado al de los demás proveedores (vendors). Si queremos aislarlo, podemos agregar una línea separada en **webpack.mix.js** para que todos los archivos JavaScript generados en el blog se almacenen en una carpeta separada:

```
  mix.js('resources/assets/js/app.js', 'public/js').
sourceMaps();
  mix.scripts('resources/assets/js/favoritos.js', 'public/
js/blog.js').sourceMaps();
  mix.sass('resources/assets/sass/app.scss', 'public/
css');
```

En este punto, debemos remover **require('./favoritos');** de **resources\assets\js\app.js** e incluir también el archivo **js/blog.js** en nuestro layout para no repetir código y que todo funcione correctamente. Es importante tener en cuenta el orden, debido a que el código de **js/blog.js** hace uso de jQuery; por lo tanto, si no introducimos primero el framework disponible en **js/app.js**, éste no funcionará.

```
    <script type="text/javascript" src="{{ mix('/js/app.
js') }}"></script>
    <script type="text/javascript" src="{{ mix('/js/blog.
js') }}"></script>
```

Figura 10. Observemos que **npm run development** muestra que acaba de generar el archivo **js/blog.js**.

Compilación para producción

En el resultado del comando **npm run development** observamos que en los archivos **js/app.js** y **css/app.css** aparece la leyenda **[big]**, debido al tamaño de éstos. Pero podemos generar archivos más pequeños si comprimimos el código. La compresión consiste en tomar cada archivo y eliminar todos los caracteres innecesarios, por ejemplo,

Compilar varios archivos

En este caso, en el primer parámetro de la función **scripts** estamos usando la ruta hacia un archivo; no obstante, también podemos enviar un array con las rutas de varios archivos para que WebPack los una a todos en **js/blog.js**. Lo mismo podemos hacer para compilar varios archivos CSS, sólo que debemos utilizar la función **styles** en vez de la de scripts.

los espacios de tabulación que son útiles para tener un código legible, pero que no producen ningún efecto en la ejecución.

Si ejecutamos el comando **npm run production**, obtendremos los mismos archivos, pero con un tamaño inferior, debido a que ahora están comprimidos.

Es importante considerar que la compresión no genera impacto en el tiempo de procesamiento.

Figura 11. Observemos que, mediante la compresión de recursos, el tamaño de los archivos se ha reducido de manera considerable.

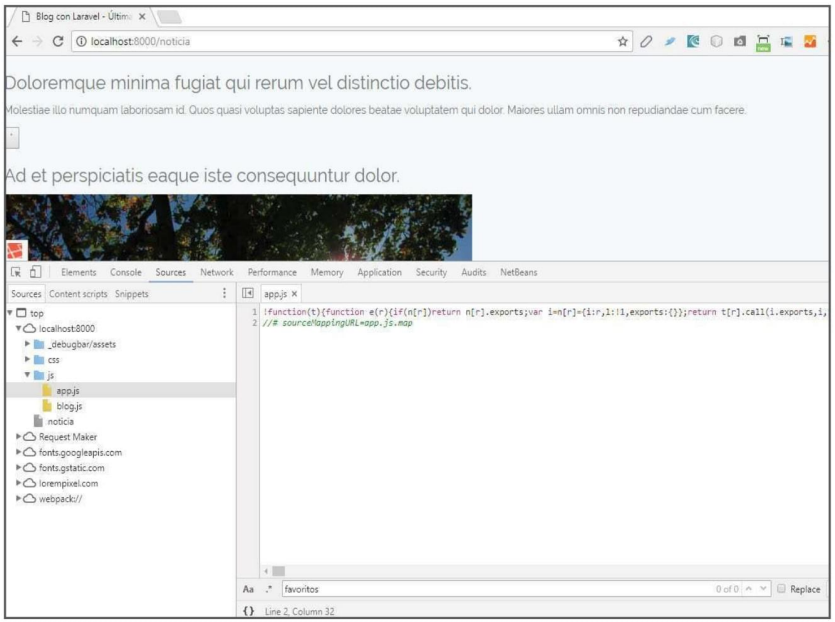


Figura 12. Al estar comprimido, todo aparece en una sola línea. En Chrome, mediante el botón {} podemos visualizarlo sin compresión.

Otro aspecto importante que debemos tener en cuenta al compilar para producción se relaciona con la caché. Los navegadores actuales utilizan estrategias de caché para no recargar un mismo archivo en cada petición, pero esto puede hacer que, si introducimos cambios en los recursos, los mismos no se reflejen en la página web producto de estar accediendo a la versión **cacheada** del navegador del archivo **js/app.js**, en vez de a la versión que existe en el servidor.

Una estrategia para resolver este problema es versionar los archivos compilados, es decir, adjuntar una estampilla de tiempo para invocar a cada archivo. De esta manera, el navegador nunca encontrará dicho archivo en su caché, por lo que siempre irá a buscarlo al servidor.

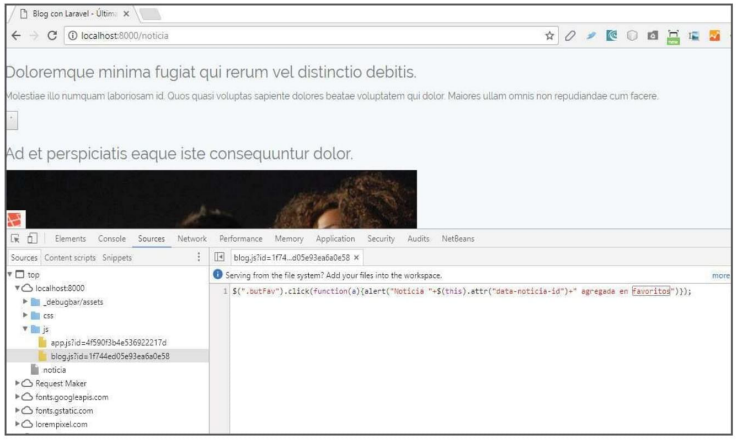


Figura 13. La función **mix** utilizada en el layout se encarga de generar la versión correspondiente para cada archivo.

Bootstrap

Bootstrap es un conjunto de herramientas que incluyen componentes en JavaScript, HTML y CSS destinadas a construir interfaces gráficas adaptativas.

La traducción del término **bootstrap** se refiere al pedazo de cuero u otro material fuerte en la parte posterior de una bota que

utilizamos para poder colocárnosla. Debido a este concepto tan popular, este término se encuentra disperso en muchas librerías, frameworks y sistemas.

Si inspeccionamos el archivo **package.json**, veremos que ya se encuentra la inclusión del framework Bootstrap y es por eso que lo hemos incluido al compilar todos los recursos para las interfaces.

La ventaja más importante que brinda Bootstrap es contar con una variedad de componentes que podemos apreciar en la Web de su sitio oficial, disponible en **http://getbootstrap.com**, junto con varios ejemplos para implementar.

Al momento de escribir este libro, existe una versión Alpha de su nuevo release 4.0.0, aunque la versión que se instala con Laravel es la 3.7.7, cuya documentación está disponible en **https://getbootstrap.com/docs/3.3**.

Vamos a comenzar utilizando un componente simple; reemplacemos el código del botón para guardar la noticia favorita que agregamos en el archivo **resources\views\frontend\noticia\index.blade.php**, por el siguiente código:

```
<button class="butFav btn btn-primary" data-noticia-id="{{ $noticia->id }}"><span class="glyphicon glyphicon-star" aria-hidden="true"></span></button>
```

Si prestamos atención a los cambios, hemos agregado un estilo **btn**, luego **btn-primary** y por último una etiqueta **span** para agregar una fuente que será utilizada en un ícono para ilustrar al botón. Ésta es una implementación muy similar a la obtenida del ejemplo de la

Twitter Bootstrap

Bootstrap es un framework para diseño de sitios y aplicaciones web. Fue desarrollado por Twitter y lanzado en 2011, y hoy es uno de los más utilizados del mercado. Presenta como ventaja una excelente curva de aprendizaje que permite construir interfaces agradables en poco tiempo y con poco esfuerzo. Es recomendable leer el libro de esta editorial sobre el tema, disponible en **www.redusers.com/noticias/publicaciones/bootstrap**.

documentación oficial disponible en **https://getbootstrap.com/docs/3.3/components/#glyphicons-examples**.

Otra mejora que podemos incorporar es el uso de paneles para las noticias, **https://getbootstrap.com/docs/3.3/components/#panels**, lo cual hacemos modificando **resources\views\frontend\noticia\index.blade.php** de la siguiente forma:

```
  @extends('frontend.layouts.main')
  @section('title', 'Últimas noticias')
  @section('content')
  <div class="container">
    <div class="page-header">
      <h1>Blog con Laravel <small>Últimas noticias</
small></h1>
    </div>
    <div class="row">
      @foreach($noticias as $noticia)
          <div class="panel panel-primary">
            <div class="panel-heading">
              <h3 class="panel-title">{{ $noticia->titulo
}}</h3>
            </div>
            <div class="panel-body">
            @if(isset($noticia->imagen))
              <img class="img img-responsive"
src="{{$noticia->imagen}}" />
            @endif
              <p>{{ $noticia->cuerpo }}</p>
              <button class="butFav btn btn-primary" da-
ta-noticia-id="{{ $noticia->id }}"><span class="glyphicon
glyphicon-star" aria-hidden="true"></span></button>
            </div>
          </div>
      @endforeach
      </div>
    {{ $noticias->links() }}
  </div>
  @endsection
```


Observemos que, además de incorporar paneles, también hemos incluido un ** ** para organizar todas las noticias en un ** **.

Figura 14. Hemos incorporado en las imágenes la clase **img-responsive**, pero ésta sólo actúa en caso de que el contenedor sea más pequeño.

Una ventaja de los componentes de Bootstrap es que podemos extenderlos para que se adapten a nuestras necesidades. Vamos a incorporar un nuevo archivo de estilos para agregar la capacidad de que los paneles se agrupen en fila aprovechando toda la pantalla. Para hacerlo, creamos el archivo **resources\assets\css\masonry.css** con el siguiente contenido:

```
*, *:before, *:after {box-sizing:  border-box !important;}

.row {
  -moz-column-width: 25em;
  -webkit-column-width: 25em;
  -moz-column-gap: .5em;
```

```
  -webkit-column-gap: .5em;
}

.panel {
  display: inline-block;
  margin:  .5em;
  padding:  0;
  width:98%;
}
```

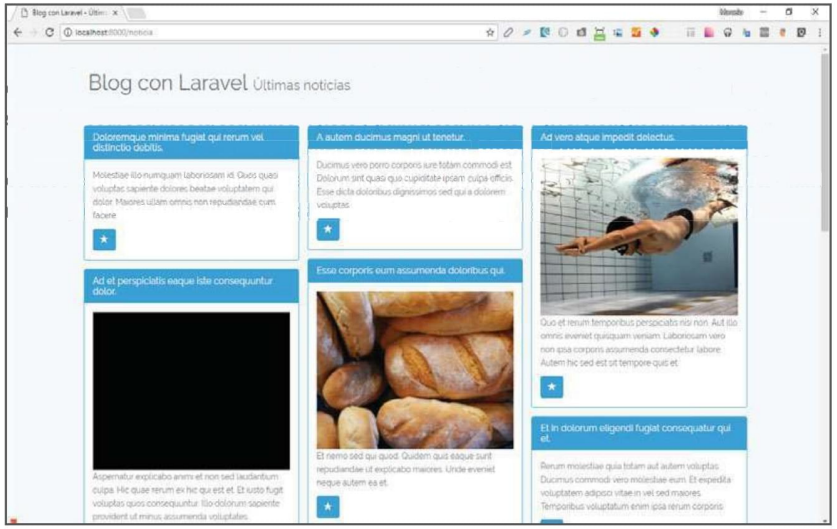


Figura 15. Con esta modificación, podemos ver que las imágenes se adaptan a su contenedor.

Ventajas de Bootstrap

Utilizar herramientas Open Source tienen como ventaja el soporte de una gran comunidad. En el caso de Bootstrap, existen sitios web desde donde podemos tomar interfaces ya desarrolladas, por ejemplo **https://wrapbootstrap.com** y **https://startbootstrap.com.**

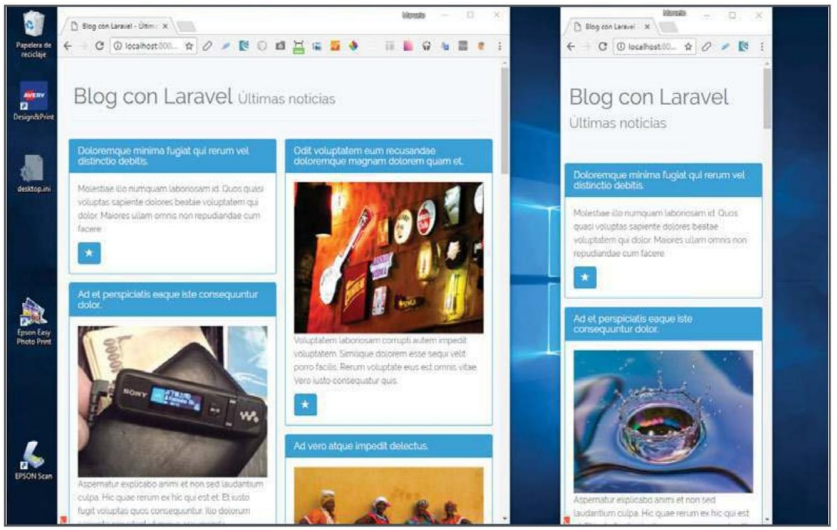


Figura 16. Si redimensionamos la ventana del navegador, apreciaremos las capacidades adaptativas que ofrece Bootstrap.

Dado que el último código que introdujimos es sólo para el frontend, en el layout del backend podemos omitirlo y, de ser necesario, compilar un archivo para los recursos de este último. Por ahora, en el layout sólo incluiremos los recursos de librerías externas, las cuales podemos reutilizar sin problemas. El código del archivo **resources\views\backend\layouts\main.blade.php** quedará de la siguiente forma:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">
    <link rel="icon" href="">
    <title>Administrador del Blog - @yield('title')</
title>
    <link rel="stylesheet" href="{{ mix('/css/app.css')
```

```
}}">
    </head>
    <body>
      <ul>
        <li><b>Menu Principal</b></li>
        <li><a href="#">Noticias</a></li>
        <li><a href="#">Usuarios</a></li>
      </ul>
      <ul>
        @section('menu-contextual')
          <li><b>Menu Contextual</b></li>
        @show
      </ul>
      @yield('content')
    </body>
    <script type="text/javascript" src="{{ mix('/js/app.
js') }}"></script>
  </html>
```

Si recargamos la página, veremos que los estilos de Bootstrap empiezan a darle forma a la página. Vamos a reemplazar el menú que hemos realizado, por el componente navbar de Bootstrap. Debajo de la etiqueta **<body>** introducimos el siguiente código:

```
      <nav class="navbar navbar-default">
        <div class="container-fluid">
```

Alternativa a Boostrap

Una alternativa que podemos utilizar como framework CSS, si no queremos construir nuestras propias interfaces desde cero, es **Foundation**, el cual también es gratuito y open source: **https://foundation.zurb.com**. Podemos encontrar una comparación entre ambos, en idioma español, en la dirección **https://loogic.com/bootstrap-vs-foundation-por-que-usar-bootstrap-para-un-sitio-web**.


```
          <div class="navbar-header">
            <button type="button" class="navbar-toggle col-
lapsed" data-toggle="collapse" data-target="#bs-example-
navbar-collapse-1" aria-expanded="false">
              <span class="sr-only">Toggle navigation</
span>
              <span class="icon-bar"></span>
              <span class="icon-bar"></span>
              <span class="icon-bar"></span>
            </button>
          </div>
          <!-- Collect the nav links, forms, and other con-
tent for toggling -->
          <div class="collapse navbar-collapse" id="bs-ex-
ample-navbar-collapse-1">
            <ul class="nav navbar-nav">
              <li class="dropdown">
                <a href="#" class="dropdown-toggle" data-
toggle="dropdown" role="button" aria-haspopup="true" aria-
expanded="false">Menu Principal <span class="caret"></
span></a>
                <ul class="dropdown-menu">
                  <li><a href="#">Noticias</a></li>
                  <li><a href="#">Usuarios</a></li>
                </ul>
              </li>
            </ul>
```

Perfiles del futuro

Uno de los perfiles profesionales más buscados en la actualidad es el de Analista de experiencia de usuario, también conocido como analista UX. Se trata de una persona dedicada a analizar las interacciones que realiza una persona con un determinado producto. En desarrollo web, esto involucra en gran parte a las definiciones de las interfaces, dado que ellas son el principal punto de contacto entre el sistema y el usuario.

```
        </div><!-- /.navbar-collapse -->
      </div><!-- /.container-fluid -->
    </nav>
```

Para aprovechar las ventajas del sistema de grillas de Bootstrap, cuya documentación se encuentra en **https://getbootstrap.com/docs/3.3/css/#grid**, vamos a cambiar el layout, de manera tal que el menú contextual quede volcado siempre a la izquierda de la pantalla. Entonces, debajo de la barra de navegación que acabamos de introducir, agregamos el siguiente código:

```
<div class="container-fluid">
    <div class="row">
      <div class="col-md-2">
        <ul class="nav nav-pills nav-stacked">
          @section('menu-contextual')
            <li><b>Menu Contextual</b></li>
          @show
        </ul>
      </div>
      <div class="col-md-10">
        @yield('content')
      </div>
    </div>
  </div>
```

Gulp

Las versiones de Laravel que utilizan **elixir** operan con otro manejador de tareas JavaScript, denominado **GulpJS**, cuya documentación puede consultarse en **https://gulpjs.com**. En la dirección **https://da-14.com/blog/gulp-vs-grunt-vs-webpack-comparison-build-tools-task-runners** podemos encontrar un completo análisis comparativo entre Gulp, WebPack y Grunt, otro ejecutor de tareas JavaScript conocido.

Al haber introducido este cambio en el layout, todas las interfaces del backend seguirán el mismo patrón.

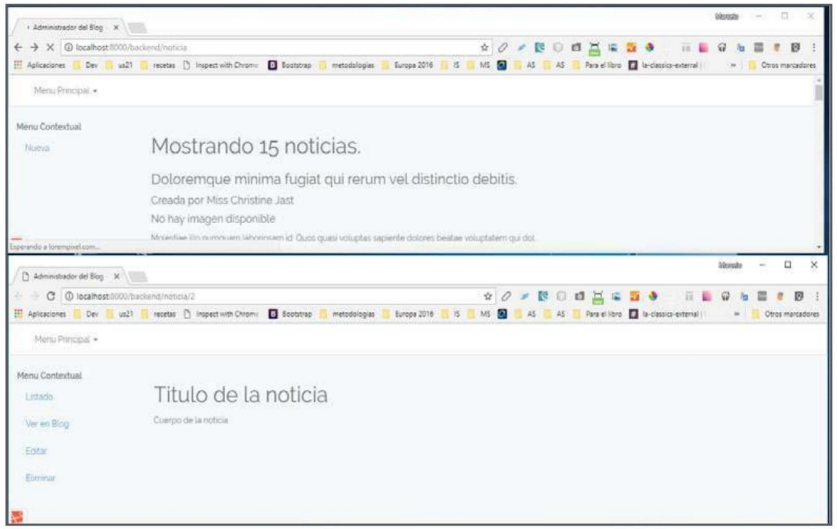


Figura 17. Salvo que utilicemos otro layout, todas las interfaces del backend tendrán en la columna izquierda el menú contextual, y en la otra, el contenido.

Resumen Capítulo 10

En este capítulo realizamos un breve repaso de JavaScript y analizamos lo que el framework Laravel entiende por recurso. Luego, fuimos introduciéndonos en las herramientas NPM para la descarga de paquetes JavaScript y WebPack para la compilación de recursos. También vimos cómo Laravel Mix se apoya en ambas y simplifica la integración de todos los recursos necesarios para construir interfaces. A continuación estudiamos los principales comandos de Laravel Mix y los aspectos que debemos considerar para compilar recursos en un ambiente productivo. Por último, realizamos una breve introducción a Bootstrap e implementamos algunos componentes para nuestro blog de noticias, teniendo como resultado una interfaz rápida y elegante.

ACTIVIDADES

Test de Autoevaluación

1. ¿En qué consiste un recurso?
2. ¿Por qué es necesario entender NPM y WebPack para utilizar Laravel Mix?
3. ¿Para qué sirve NPM?
4. ¿Para qué sirve WebPack?
5. ¿Qué ventaja presenta compilar todo en un archivo, frente a incluir todos los necesarios en el código HTML?
6. ¿Cuál es la diferencia entre el comando **npm run development** y **npm run production**?
7. ¿Para qué se utiliza el versionado de archivos compilados?
8. ¿Es importante el orden en el cual se cargan los archivos js o css?
9. ¿En qué consiste Bootstrap?
10. ¿Es posible modificar y/o extender componentes de Bootstrap?

Ejercicios prácticos

1. Cree una interfaz para visualizar la noticia completa en el frontend con todos los datos que considere pertinentes.
2. Modifique y vincule la interfaz generada en el punto anterior con el listado de noticias del frontend.
3. Reemplace el componente **alert** por el componente de ventana modal de Bootstrap.

11 Formularios

Los formularios son un elemento clave en cualquier aplicación web debido a que son el principal punto de ingreso de información por parte del usuario. Los malos formularios pueden generar un efecto negativo derivado en dropoff, es decir, hacer que el usuario no ingrese más en el sistema. Por el contrario, un buen formulario, junto con una performance adecuada, invitan al usuario a quedarse e interactuar con las diferentes alternativas que podemos ofrecerle.

FORMULARIOS EN LARAVEL

Para la generación de interfaces, es posible utilizar HTML plano, pero existe una librería denominada **Collective**, que facilita el vínculo entre la interfaz y los datos. Dichos datos deben ser validados, y para esto, Laravel posee un mecanismo de validación que forma parte del mismo framework. Estas dos herramientas permiten crear formularios de manera simple y rápida.

Collective

Laravel Collective es una librería que representa un conjunto de componentes que permiten, entre otras cosas, generar formularios y código HTML. Es muy simple de instalar mediante el comando **composer require laravelcollective/html**.

Recordemos que, gracias a auto-discovery, en la version 5.5 de Laravel no debemos realizar ninguna modificación adicional, mientras que en versiones anteriores es necesario alterar **config/app.php** agregando el provider:

```
    'providers' => [
      // ...
      Collective\Html\HtmlServiceProvider::class,
      // ...
    ],
```

Y también debemos modificar el array de aliases en el mismo archivo más abajo:

Laravel Collective

En un principio, formaba parte de Laravel, pero luego Taylor Otwell decidió quitarla ya que no todos los sistemas utilizan formularios HTML y hay muchos que prefieren usar directamente código HTML. Además, se buscaba hacer el framework más liviano. La documentación oficial se encuentra disponible en **https://laravelcollective.com**.


```
    'aliases' => [
      // ...
        'Form' => Collective\Html\FormFacade::class,
        'Html' => Collective\Html\HtmlFacade::class,
      // ...
    ],
```

Vamos a crear el formulario para poder generar noticias. Primero creamos el método **create** en **Blog\Http\Controllers\Backend\NoticiaController** con el siguiente código:

```
  public function create(){
    return view('backend.noticia.create');
  }
```

Simplemente, se renderiza una vista, la cual tendrá el siguiente contenido:

```
  @extends('backend.layouts.main')

  @section('title', 'Nueva noticia')

  @section('menu-contextual')
    @parent
    <li><a href="{{ route('backend.noticia.index') }}">Ir
al listado</a></li>
  @endsection

  @section('content')
    <form method="post" action="{{ route('backend.noticia.
store') }}">
      <div class="form-group">
        <label  class="control-label" for="titulo">Titulo</
label>
        <input type="text" class="form-control"
```

```
placeholder="Titulo">
      </div>
      <div class="form-group">
        <label  class="control-label" for="cuerpo">Cuerpo</
label>
        <input type="text" class="form-control"
placeholder="Cuerpo">
      </div>
      <button type="submit" class="btn btn-
default">Enviar</button>
    </form>
  @endsection
```

A partir de este punto, nos concentraremos en la sección **<form>**. Observemos que para la generación del atributo **action** hemos utilizado el helper para construir rutas de Laravel.

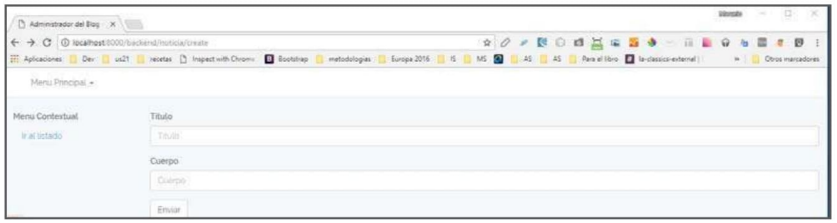


Figura 1. En este punto, obtendremos el principio de un formulario elegante gracias a los estilos de Bootstrap.

Hasta ahora estamos utilizando HTML plano y aún no hemos sacado ninguna ventaja de Collective. La primera ventaja que podremos apreciar es al introducir el **<select>** para seleccionar la categoría de la noticia.

Primero, nos aseguramos de enviar esa información a la vista desde el controlador, para lo cual hay que modificar el método **create** de la siguiente forma:

```
public function create(){
  $categorias = Categoria::pluck('nombre', 'id');
```



```
    return view('backend.noticia.create', ['categorias' =>
$categorias]);
  }
```

Notemos que estamos utilizando el método **pluck** de **collection**, que nos devuelve un array. En este caso, al utilizar dos parámetros, el primero indicará qué atributo será usado para los valores del array, y el segundo es el que será utilizado para los índices. Recordemos que, al utilizar el modelo **Categoria**, tenemos que introducir el namespace al inicio del controlador. Con este array podemos generar el **select** en la vista de la siguiente forma:

```
<select id="categoria" class="form-control">
  <option value="">Seleccione una categoría...</option>
  @foreach($categorias as $id => $nombre)
    <option value="{{ $id }}">{{ $nombre }}</option>
  @endforeach
</select>
```

Primero dejamos un valor vacío para forzar al usuario a que seleccione uno; luego generamos una etiqueta **<option>** para cada categoría.

Si utilizamos Collective, podemos reemplazar este código tal como se muestra a continuación:

```
{{ Form::select('categoria', $categorias, null, ['place-
holder' => 'Seleccione una categoría', 'class' => 'form-
control']) }}
```

Divide y triunfarás

Es una estrategia muy común y útil segmentar lo máximo posible los formularios a través de sus campos. Esto permite distribuir el trabajo entre varias personas y a su vez dirigir el foco de cada programador según el criterio establecido. Otra ventaja es que el código también estará segmentado y será más legible.

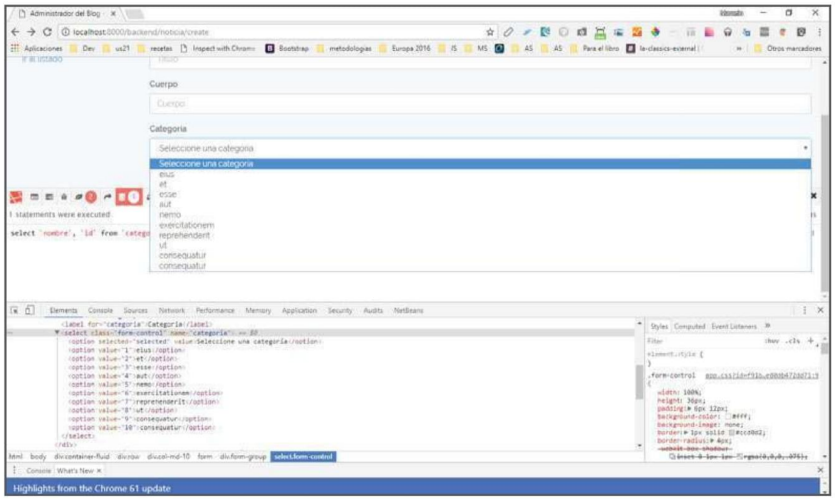


Figura 2. Con esta única línea podemos generar el mismo código HTML que logramos con el código anterior.

Reemplacemos los componentes creados inicialmente con métodos de Collective, de la siguiente forma:

```
  <div class="form-group">
    <label class="control-label" for="titulo">Titulo</la-
bel>
    {{ Form::text('titulo', null, ['class' => 'form-con-
trol', 'placeholder' => 'Titulo']) }}
```

Formularios HTML

El sitio web para desarrolladores de Mozilla contiene un artículo muy interesante con buenas prácticas para estructurar un formulario, contemplando aspectos como la usabilidad y la accesibilidad. Podemos consultarlo en español en el siguiente enlace: **https://developer.mozilla.org/es/docs/Learn/HTML/Forms/How_to_structure_an_HTML_form.**


```
  </div>
  <div class="form-group">
    <label class="control-label" for="cuerpo">Cuerpo</la-
bel>
    {{ Form::textarea('cuerpo', null, ['class' => 'form-
control', 'placeholder' => 'Cuerpo']) }}
  </div>
```

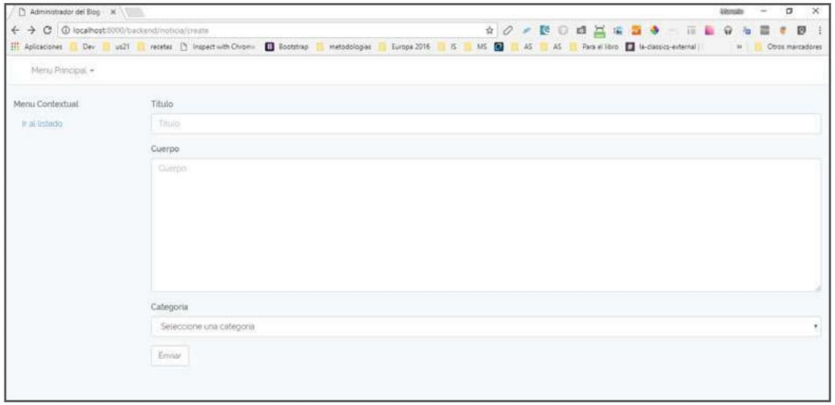


Figura 3. Hemos reemplazado el **input text** del copete por un **textarea**, cuya interfaz es más apropiada para este dato.

Si observamos con detalle, podemos apreciar que no existe una gran ventaja en reemplazar los **input** de HTML por **Form::text**; no obstante, aún falta vincular el formulario con el modelo que estaremos utilizando. Al realizar este vínculo, los campos se cargarán con la información del modelo. Antes de poder apreciar esto, es necesario agregar la lógica para procesar el formulario.

Procesar formularios

Hay varios aspectos que debemos considerar al procesar formularios. Vamos a empezar por el peor ejemplo, y luego lo iremos mejorando. Modifiquemos el método **store** de **Blog\Http\Controllers\Backend\NoticiaController** con el siguiente contenido:

```
public function store(Request $request)
{
  $noticia = new Noticia();
  $noticia->titulo = $request->input('titulo');
  $noticia->cuerpo = $request->input('cuerpo');
  $noticia->categoria_id = $request->input('categoria');
  $noticia->autor = 1;
  $noticia->save();
}
```

En esta lógica estamos creando un objeto **Noticia** y recuperando la información del formulario desde el objeto **Request**, que recibe el método utilizando **input()**. El parámetro debe coincidir con el atributo **name** de la etiqueta **<input>** introducida en el **<form>** en la vista.

Si no ingresamos ningún dato en el formulario y presionamos el botón enviar, obtendremos un error como el siguiente:

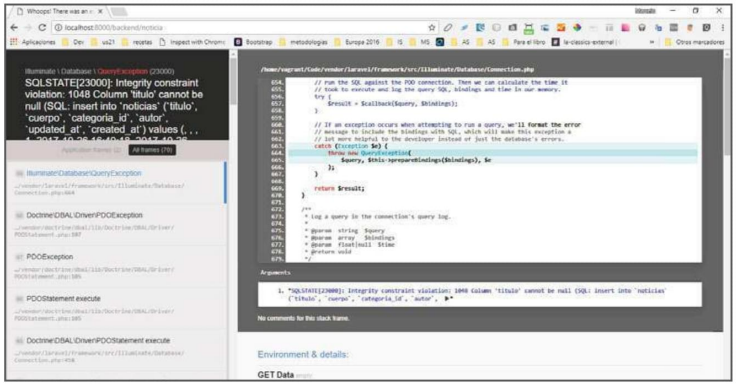


Figura 4. El error se produce debido a que **titulo** es un campo obligatorio, porque lo establecimos como **NOT NULL** cuando generamos la tabla.

Existen diferentes estrategias para manejar esta situación. La peor de todas es capturar el error producido por el motor de base de datos y actuar en consecuencia.

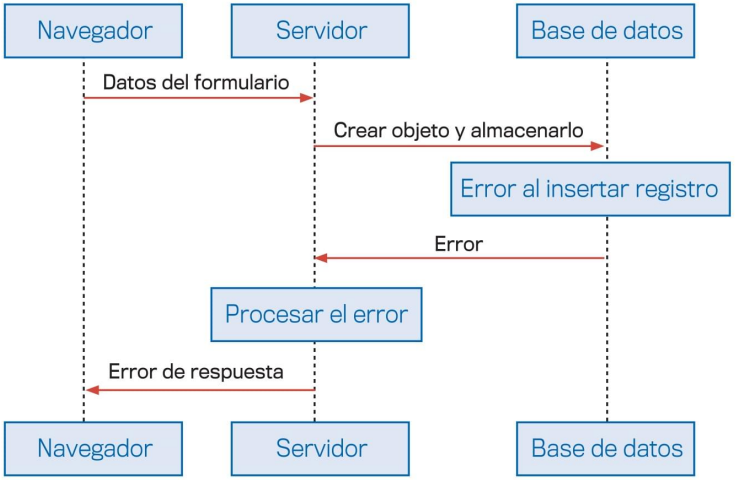


Figura 5. Con esta aproximación, estamos realizando procesamientos tanto en el servidor web como en el motor de base de datos.

Validar requests

Lo que debemos hacer es validar la información que proviene del **request**. Dado que éste contiene la información que se ingresó mediante el formulario, lo que estaremos haciendo en última instancia en este caso es validar los datos del formulario.

Para esto, Laravel posee herramientas que facilitan el trabajo. Agreguemos al principio del método **store** el siguiente código:

```
$validatedData = $request->validate([
  'titulo' => 'required',
  'cuerpo' => 'required',
  'categoria' => 'required',
]);
```

El método **validate** recibe un array cuyos índices son los nombres de los campos que deben validarse, y los valores son las reglas que deberán

aplicarse para comprobar el dato. En este caso estamos utilizando para todos los campos la regla **required**, la cual implica que el dato es requerido.

Cuando una de estas reglas de validación no se cumple, el controlador retorna a la vista un array con errores que nos permitirán brindar información al usuario respecto del problema; dicho array lo encontraremos en la variable **$errors**.

Agreguemos el siguiente código en la vista al inicio de **@section('content')** para poder visualizar los errores:

```
<h3>Nueva noticia</h3>
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

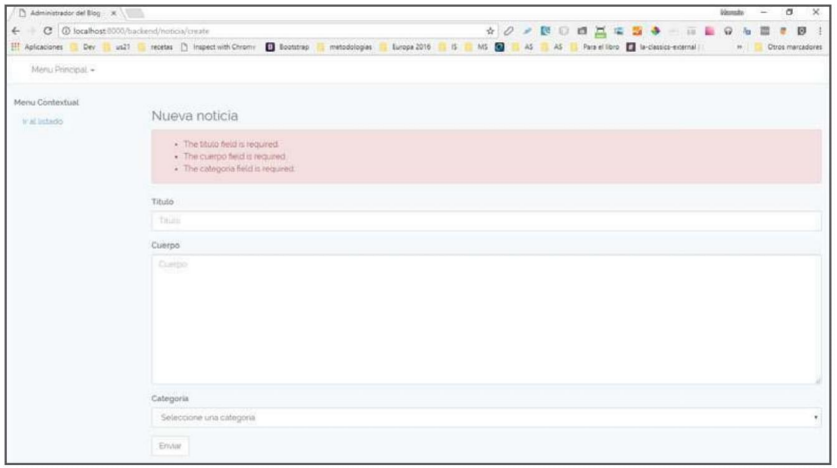


Figura 6. Por default, Laravel ya posee mensajes de error para enviar al usuario, los cuales muestra en inglés.

En este punto ya podemos aprovechar algunas de las ventajas de Collective. Observemos que, cuando se produce el error y se retorna el formulario, los datos que el usuario ha completado ya aparecen cargados, de manera tal que no tenemos que introducir lógica para que esto suceda y, a la vez, el usuario logra una experiencia mucho más agradable. Tengamos en cuenta que algunos formularios pueden contener muchos campos, y el hecho de tener que cargar todos nuevamente aumenta las probabilidades de que se produzca un dropoff.

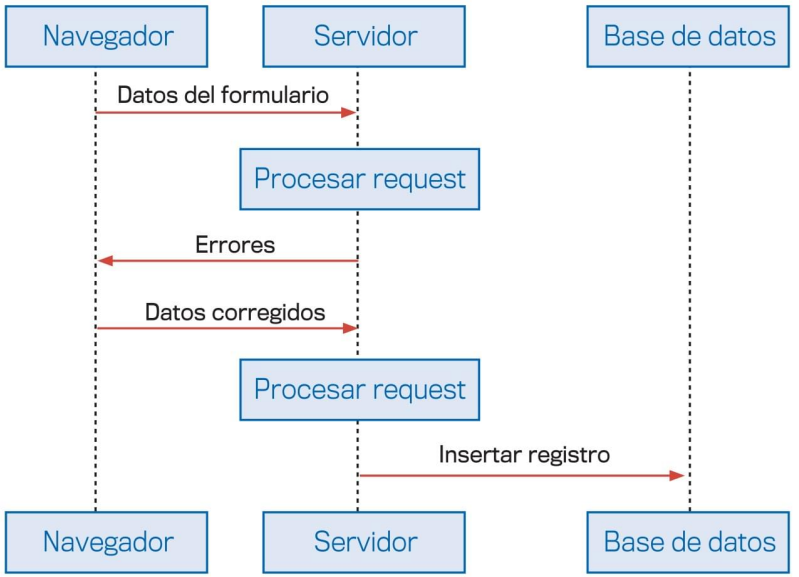


Figura 7. Con esta aproximación evitamos que el motor procese datos inválidos y mejoramos los tiempos de respuesta al usuario.

Antes de profundizar el tema de la visualización de errores, analicemos las principales características de las reglas de validación.

Si observamos el campo **titulo** de la tabla, veremos que éste ha sido establecido con un máximo de 255 caracteres.

Observemos qué sucede si cargamos un título que supere esa cantidad, y también cargamos un copete y una categoría.

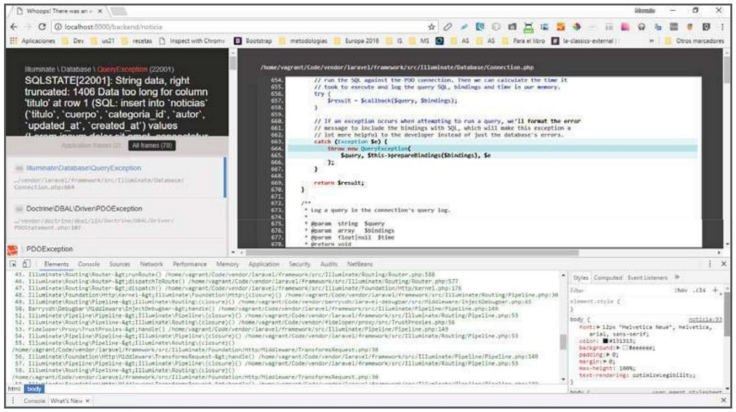


Figura 8. En este caso obtenemos otra vez un error desde la base de datos, por no considerar el caso de que se exceda la cantidad de caracteres permitidos.

Vamos a modificar el código de la validación para introducir esta nueva regla:

```
$validatedData = $request->validate([
  'titulo' => 'required|max:255',
  'cuerpo' => 'required',
  'categoria' => 'required',
]);
```

Si observamos con atención, el cambio que hemos introducido fue agregar la regla **max:255** para el campo **titulo**. Cuando queremos aplicar más de una regla sobre un campo debemos utilizar el carácter |, también conocido como **pipe**, que significa tubo.

En la regla **max** es necesario especificar el largo máximo permitido, lo cual hacemos introduciendo en el medio el carácter **dos puntos**. Vale aclarar que no es necesario que coincida con el largo del campo establecido en la base de datos.

El listado completo de reglas de validación puede consultarse en el enlace **https://laravel.com/docs/5.5/validation#available-validation-rules**.

Recordemos que la columna **titulo** posee un índice único, de modo que si intentamos agregar el mismo título dos veces, la base de datos enviará un error.

La regla **unique** realiza esa validación consultando a la base de datos la existencia de ese campo o no. Agreguemos un nuevo pipe para cargar esta regla sobre el campo **titulo**, de manera que quede de la siguiente forma:

```
'titulo' => 'required|max:255|unique:noticias,titulo',
```

Luego de la regla, introducimos dos puntos para indicar la tabla y, separado por una coma, la columna donde deberá consultarse la regla de unicidad del campo en cuestión. Hasta ahora hemos introducido las reglas de validación en el controlador. Sin embargo, es muy probable que utilicemos las mismas reglas si precisamos validar un **request** proveniente del frontend o de los servicios web. Por lo tanto, será conveniente trasladar las reglas de validación al modelo, para lo cual debemos introducir el siguiente método estático en **Blog\Models\Noticia**:

```
  public static function getRules($id = 0){
    return [
        'titulo' => 'required|max:255|unique:noticias,titu
lo,' . $id,
        'cuerpo' => 'required',
        'categoria_id' => 'required',
        'imagen' => 'image|max:2048'
    ];
  }
```

Observemos que hemos incluido un parámetro opcional en el método. Lo que hace es concatenarse a la regla **unique** para ignorar su validación en un determinado **id**. Haremos uso de este atributo al momento de actualizar el modelo.

Luego debemos modificar el controlador para buscar las reglas de validación en el modelo:

```
$validatedData = $request->validate(Noticia::getRules());
```

Validación CSRF

Recordemos que, en el **Capítulo 3**, quitamos todas las validaciones CSRF para todas las rutas que comiencen con **backend/noticia**, lo cual logramos modificando el archivo **app\Http\Middleware\VerifyCsrfToken.php**.

Si reactivamos esta validación, veremos que el método **$request->validate()** también se encarga de realizarla. Para lograrlo, simplemente dejamos vacío el array **$except** de dicha clase y enviamos un formulario con todos los datos cargados correctamente. La validación hará que visualicemos una pantalla indicando que la sesión ha expirado.

Podemos agregar el token para la protección CSRF utilizando las funciones de Collective para abrir y cerrar los formularios. Reemplacemos la línea **<form method="post" action="{{ route('backend.noticia.store') }}">** por **{{ Form::open(['action' => 'backend.noticia.store']) }}** y el cierre que hacemos con **</form>** por esta otra línea: **{!! Form::close() !!}**.

Observemos que Collective también ha introducido el atributo **method="post"** por default al abrir el formulario.

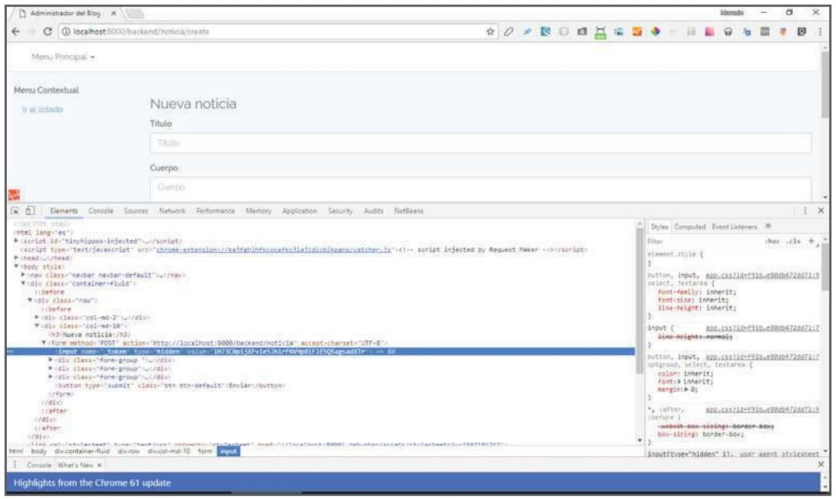


Figura 9. Si inspeccionamos el código HTML producido, podremos observar el **token** generado como un **input hidden**.

Mostrar errores

La forma actual de mostrar errores no es muy amigable. Pensemos que, si estamos en un formulario largo y/o en una pantalla muy pequeña, tenemos que hacer scroll hasta la parte superior de la pantalla y recordar todo lo que debemos modificar. Lo recomendable es mostrar el error en cada campo, junto con un texto de ayuda que le permita al usuario comprender de qué manera subsanarlo.

Para lograrlo, vamos a modificar el código del control **titulo** del formulario tal como se muestra a continuación:

```
  <div class="form-group @if($errors->has('titulo')) has-
error has-feedback @endif">
    <label class="control-label" for="titulo">Titulo</la-
bel>
    {{ Form::text('titulo', null, ['class' => 'form-con-
trol', 'placeholder' => 'Titulo']) }}
    @if($errors->has('titulo'))
        <span id="helpBlock" class="help-block">{{ $er-
rors->first('titulo') }}</span>
        <span class="glyphicon glyphicon-remove form-con-
trol-feedback" aria-hidden="true"></span>
    @endif
  </div>
```

Con este cambio, utilizamos el método **has** de **$errors** para saber si hay errores vinculados al campo **titulo** o no; en caso de haberlos, se agregan los estilos **has-error** y **has-feedback** de Bootstrap.

Debajo del formulario volvemos a realizar la misma comprobación, en este caso, para agregar primero una etiqueta **** con la

Validaciones JavaScript

También es posible introducir validaciones en el navegador con JavaScript, lo cual evitará que el servidor web deba procesar formularios con datos erróneos y brindará una respuesta aún más rápida al usuario. Sin embargo, las validaciones de este tipo pueden pasarse por alto, por lo que es recomendable siempre introducir validaciones del lado del servidor web.

descripción del error y luego otra del mismo tipo que incorporará un ícono al final del **<input>**. Todos estos elementos harán que el campo resalte del formulario, de manera que la interfaz le estará indicando al usuario dónde poner el foco para determinar rápidamente el error.

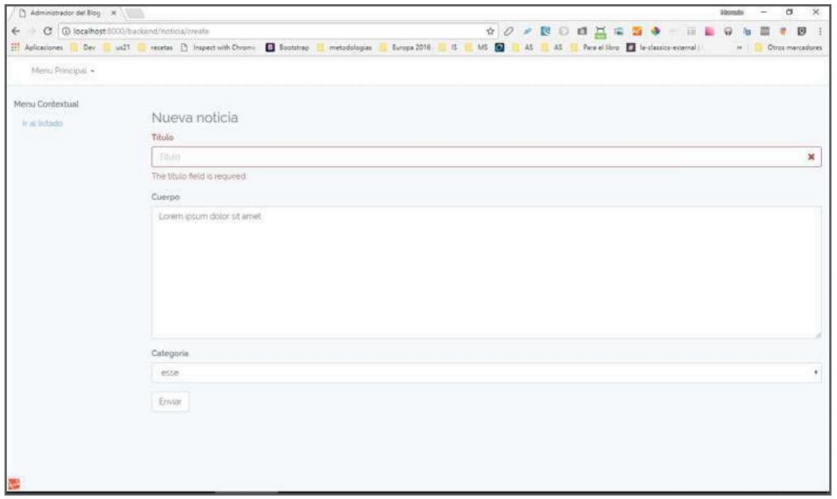


Figura 10. Hasta el momento no hemos introducido ningún estilo propio; **has-error** y **has-feedback** pertenecen a Bootstrap.

Asignación en masa

El método **store** crea un objeto y utiliza todos los valores del **request** para asignar sus atributos. Otra posibilidad es instanciar el objeto utilizando todos los datos del objeto **request**, lo que logramos con el siguiente código:

```
  public function store(Request $request)
  {
    $validatedData = $request-
>validate(Noticia::getRules());
    $noticia = new Noticia($validatedData);
    //Provisoriamente, hardcodearemos el id del autor
    $noticia->autor = 1;
    $noticia->save();
  }
```

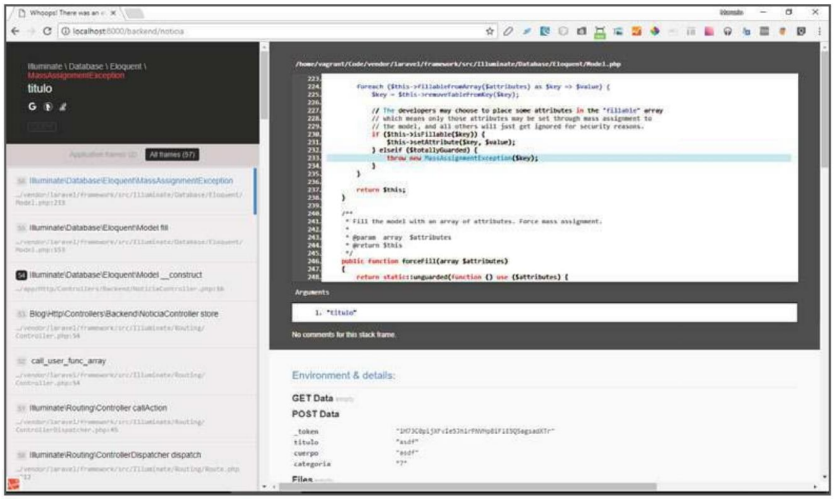



Figura 11. Si enviamos el formulario y ejecutamos esta lógica, obtendremos el error que muestra la imagen.

La asignación en masa, conocida en inglés como *mass assignment*, permite enviar un array en el constructor de un modelo. Los índices del array que coincidan con el nombre de un atributo serán asignados al objeto. Sin embargo, primero debemos definir en el modelo qué atributos podrán ser asignables masivamente, lo cual se realiza introduciendo el siguiente atributo en el modelo **Noticia**:

```
  protected $fillable = ['titulo', 'cuerpo', 'categoria_
id'];
```

Una vez implementado el atributo **$fillable**, no deberá aparecer el error de la imagen salvo que estemos intentando asignar un campo que no se encuentre declarado allí.

Observemos que, debido a que en la **Noticia** el atributo donde guardamos la referencia se llama **categoria_id**, es preciso introducir ese nombre, por lo tanto, también tenemos que cambiarlo en la vista.

Flash messages

En este punto, nuestro formulario de alta estará funcionando correctamente, pero estaremos visualizando una pantalla en blanco. Podemos comprobar que funciona consultando los registros de la tabla **noticias**.

Es recomendable redireccionar al listado de noticias una vez que se haya ejecutado el almacenamiento del objeto. Para esto es fundamental enviar un mensaje al usuario indicando que el proceso se realizó de manera correcta.

Una opción es utilizar la sesión para almacenar el mensaje que queremos mostrar. Sin embargo, debemos eliminar dicho mensaje luego del siguiente **request**, debido a que, de no hacerlo, éste permanecerá hasta que expire la sesión. Afortunadamente, Laravel tiene un método que hace esto por nosotros y se denomina **flash**. Si lo implementamos en el método **store**, éste quedará del siguiente modo:

```
  public function store(Request $request){
    $validatedData = $request-
>validate(Noticia::getRules());
    $noticia = new Noticia($validatedData);
    //Provisoriamente, hardcodearemos el id del autor
    $noticia->autor = 1;
    $noticia->save();
    $request->session()->flash('status', 'Se guardó la noti-
cia ' . $noticia->titulo);
    return redirect()->route('backend.noticia.index');
  }
```

Ahora tenemos que introducir la lógica para utilizar este dato y mostrar la información al usuario. Como va a ser algo que utilizaremos todo el tiempo, lo introduciremos en el layout **resources\views\backend\layouts\main.blade.php**, en la columna donde mostramos el contenido principal:


```
<div class="col-md-10">
  @if(Session::has('status'))
    <div class="alert alert-info">
      {{ Session('status') }}
    </div>
  @endif
  @yield('content')
</div>
```

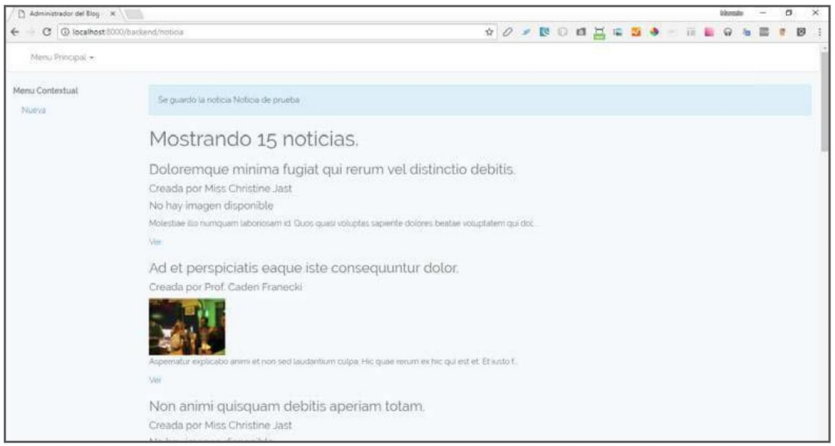


Figura 12. Con este cambio podremos enviar mensajes **status** desde cualquier controlador del backend.

Si refrescamos la página o presionamos en otro enlace, veremos que el mensaje ya no aparece más.

Sesiones

Laravel provee varios drivers para almacenar las sesiones; por default está activado el driver file, que almacena las sesiones en **storage/framework/sessions.** Es recomendable utilizar memcached o redis, ya que al persistir ambos en la memoria del servidor, se vuelven muy rápidos y escalables. Para obtener más información, se recomienda consultar la documentación oficial disponible en **https://laravel.com/docs/5.5/session.**

Procesar archivos

Hasta ahora tenemos un formulario que funciona correctamente pero que está incompleto, dado que aún no le dimos al usuario la posibilidad de cargar la imagen de la noticia. Agreguemos el siguiente componente en el formulario para lograr este propósito:

```
  <div class="form-group @if($errors->has('imagen')) has-
error has-feedback @endif">
    <label class="control-label" for="titulo">Imagen</la-
bel>
    {{ Form::file('imagen') }}
    @if($errors->has('imagen'))
        <span id="helpBlock" class="help-block">{{ $errors-
>first('imagen') }}</span>
        <span class="glyphicon glyphicon-remove form-con-
trol-feedback" aria-hidden="true"></span>
    @endif
  </div>
```

Con este cambio hemos implementado el control **Form::file** de Collective. Sin embargo, es importante activar la opción **files** en la apertura del formulario para poder enviar archivos; eso se hace aplicando el siguiente código:

```
  {{ Form::open(['route' => 'backend.noticia.store', 'files'
=> true]) }}
```

Documentación

La documentación oficial de Laravel sobre las validaciones puede consultarse en la dirección web **https://laravel.com/docs/5.5/validation**. Contiene diversos ejemplos, entre los que se incluyen la personalización de mensajes de error, la creación de reglas personalizadas de validación y el listado completo de todas las reglas de validación que brinda el framework.

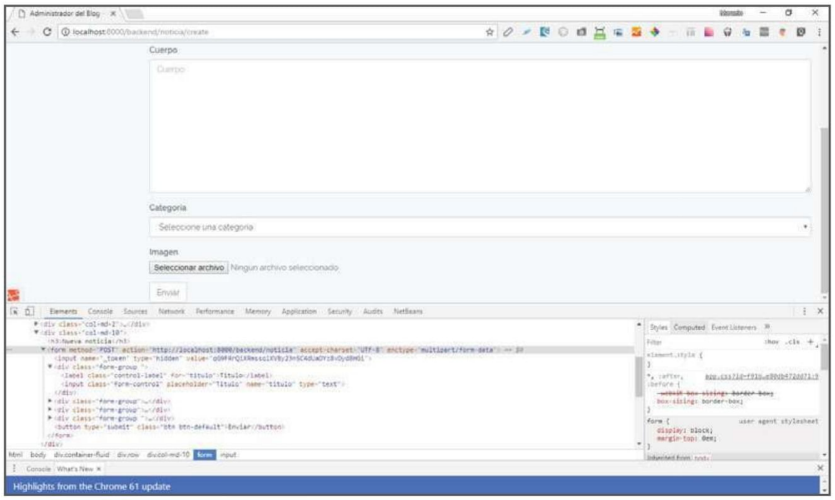


Figura 13. La activación de la opción **files** introduce el atributo **enctype="multipart/form-data"**.

Para la validación de archivos contamos con dos reglas: **image** comprueba que el archivo enviado sea del tipo **jpeg**, **png**, **bmp**, **gif** o **svg**; y **max** verifica que el tamaño del archivo sea menor o igual a la cantidad de Kb que indiquemos en la regla. Con estas dos reglas generaremos la siguiente validación:

```
  'imagen' => 'image|max:2048'
```

Observemos que, en este caso, no estamos utilizando la regla **required**, debido a que este campo no es obligatorio, pero en caso de ingresar un archivo, se aplicarán las validaciones establecidas.

Una vez aprobada la validación, debemos almacenar la imagen en algún lugar. Afortunadamente, Laravel brinda algunas herramientas para facilitarnos esta tarea. Modifiquemos el método **store** de la siguiente forma:

```
  public function store(Request $request)
  {
```

```
    $validatedData = $request-
>validate(Noticia::getRules());
    $noticia = new Noticia($validatedData);
    //Provisoriamente, hardcodearemos el id del autor
    $noticia->autor = 1;
    $noticia->save();
    //Dado a que no es un dato obligatorio, consultamos si
hay imagen
    if ($request->hasFile('imagen')) {
      //Obtenemos una instancia de la clase Illuminate\
Http\UploadedFile
      $archivoImagen = $request->file('imagen');
      //Guardamos el archivo en el storage, en la carpeta
noticias/id_noticia
      //Este método retorna la ruta donde se almacena
      $path = $archivoImagen->store('noticias/' . $noticia->id);
      //Asignamos la ruta a el campo del objeto
      $noticia->imagen = $path;
      //Actualizamos la base de datos para almacenar la ruta
      $noticia->save();
    }

    $request->session()->flash('status', 'Se guardó la noti-
cia ' . $noticia->titulo);
    return redirect()->route('backend.noticia.index');
  }
```

Procesamiento de archivos

La persistencia de archivos y la del registro donde almacenamos esa información son eventos separados. Una posibilidad es procesar primero el archivo y, en caso de un error, no continuar almacenando el registro. En este caso hacemos lo contrario, porque deseamos tener una referencia entre la carpeta donde almacenamos los archivos y el id del registro, lo cual hace que el registro se almacene dos veces. La manera indicada de hacerlo depende directamente de nuestra lógica de negocio.

Lo más importante para tener en cuenta es que podemos obtener una instancia de la clase **UploadedFile**, la cual provee varios métodos que nos ayudan a procesar los archivos. Podemos realizar consultas en **https://laravel.com/api/5.5/Illuminate/Http/UploadedFile.html**.

Figura 14. El método **store** genera un nuevo nombre de archivo aleatorio para asegurar que no se repita con otro archivo existente.

Espacios de almacenamiento

El código que acabamos de introducir presenta el problema de que los archivos no son accesibles por el navegador porque se encuentran en la ruta privada del sistema. En consecuencia, no podremos generar

Almacenamiento de archivos

Además de permitir el almacenamiento de archivos en el servidor, el driver de almacenamiento de Laravel brinda la posibilidad de utilizar servicios como Amazon Web Services y Rackspace.
A su vez, contiene métodos que facilitan la administración y persistencia de archivos.
Es recomendable leer la documentación oficial al respecto disponible en **https://laravel.com/docs/5.5/filesystem**.

interfaces que permitan visualizar las imágenes. Recordemos que la ruta pública comienza en la carpeta **public**.

Para evadir este problema, empecemos por ejecutar el comando **php artisan storage:link**.

```
Administrador: Símbolo del sistema - vagrant ssh
vagrant@blog:~/Code$ php artisan storage:link
The [public/storage] directory has been linked.
vagrant@blog:~/Code$ ls -al public/
total 22
drwxrwxrwx 1 vagrant vagrant 4096 Oct 27  2017 .
drwxrwxrwx 1 vagrant vagrant 8192 Oct 22 22:46 ..
drwxrwxrwx 1 vagrant vagrant    0 Oct 23 02:09 css
-rwxrwxrwx 1 vagrant vagrant    0 Aug 12 22:45 favicon.ico
drwxrwxrwx 1 vagrant vagrant    0 Oct 22 16:47 fonts
-rwxrwxrwx 1 vagrant vagrant   45 Aug 13 15:09 hola.php
-rwxrwxrwx 1 vagrant vagrant  584 Aug 12 22:45 .htaccess
-rwxrwxrwx 1 vagrant vagrant 1823 Aug 12 22:45 index.php
drwxrwxrwx 1 vagrant vagrant 4096 Oct 22 23:27 js
-rwxrwxrwx 1 vagrant vagrant  370 Oct 23 02:09 mix-manifest.json
-rwxrwxrwx 1 vagrant vagrant   24 Aug 12 22:45 robots.txt
lrwxrwxrwx 1 vagrant vagrant    0 Oct 27  2017 storage -> /home/vagrant/Code/storage/app/public
-rwxrwxrwx 1 vagrant vagrant  914 Aug 12 22:45 web.config
vagrant@blog:~/Code$
```

Figura 15. Este comando generará un enlace simbólico entre la carpeta **public** y **storage/app/public**.

Luego, debemos hacer un cambio al momento de almacenar el archivo:

```
$path = $archivoImagen->storeAs('public/noticias/' . $noticia->id, $archivoImagen->getClientOriginalName());
```

El método **store** genera un nombre de archivo aleatorio, lo cual es muy útil para evitar que se pisen archivos con un mismo nombre. Sin embargo, nuestra lógica almacena un archivo por noticia y creamos una carpeta por noticia, en consecuencia, no es posible que se repitan archivos con un mismo nombre. Debido a esto y para mantener el nombre original del archivo subido, utilizamos el método **storeAs** en lugar de **store** e invocamos al método **getClientOriginalName()** para obtener el nombre original del archivo.

Observemos también que, debido al enlace simbólico creado, estamos almacenando en la carpeta **public**. Sin embargo, cuando queremos referenciar al archivo desde el navegador, debemos ignorar esa carpeta porque es la raíz de los archivos accesibles del navegador. En consecuencia, deberemos guardar la ruta de la siguiente forma:


```
$savedPath = str_replace("public/", "", $path);
//Asignamos la ruta a el campo del objeto
$noticia->imagen = $savedPath;
```

Por último, para obtener una referencia al archivo almacenado, podemos usar el helper **asset**. Vamos a modificar la vista **resources\views\backend\noticia\show.blade.php** para comprobarlo:

```
@section('content')
<!-- #### Inicio de contenido en vista noticia #### -->
  <h1>{{ $noticia->titulo }}</h1>
  <p>{{ $noticia->cuerpo }}</p>
  <img src="{{ asset('storage/' . $noticia->imagen) }}"
/>
<!-- #### Cierre de contenido en vista noticia #### -->
@endsection
```

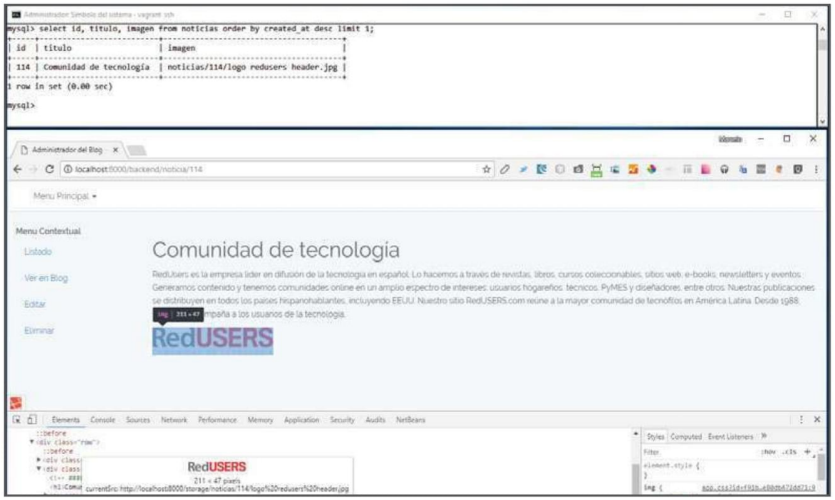


Figura 16. El helper **asset** genera un enlace que puede ser invocado por la Web.

Observemos que hemos almacenado el archivo con espacios, y **asset** los encodea generando la cadena url necesaria para invocar al recurso.

Model binding

Bind significa "pegar" en inglés. Model binding es la manera en la que podemos vincular un formulario con un modelo, lo que permitirá que, al momento de cargar la pantalla para editar un modelo, todos los datos existentes de él aparezcan cargados.

El formulario que utilizaremos para editar será el mismo que hemos creado para dar de alta noticias, con la salvedad de que la apertura se hará de manera diferente, dado que en la edición debemos realizar el model binding.

Primero vamos a crear el archivo **resources\views\backend\noticia\form.blade.php** para almacenar todos los componentes del formulario. Copiamos todo el contenido que se encuentra entre **{{ Form::open(['method' => 'post', 'route' => 'backend.noticia.store', 'files' => true]) }} y {!! Form::close() !!}** en **resources\views\backend\noticia\create.blade.php** y lo pegamos en el archivo recientemente creado. El contenido final de la sección **content** del archivo **resources\views\backend\noticia\create.blade.php** deberá ser el siguiente:

```
  @section('content')
    <h3>Nueva noticia</h3>
    {{ Form::open(['route' => 'backend.noticia.store',
'files' => true]) }}
      @include('backend.noticia.form')
    {!! Form::close() !!}
  @endsection
```

Observemos que, con este cambio, el formulario de creación de noticias se deberá mantener y funcionar de la misma manera. Ahora editamos la sección content del archivo para la edición **resources\views\backend\noticia\edit.blade.php**:

```
  @section('content')
    <h3>Editando noticia {{ $noticia->titulo }}</h3>
    {{ Form::model($noticia, [ 'method' => 'put', 'route'
=> ['backend.noticia.update', $noticia->id], 'files' =>
true]) }}
      @include('backend.noticia.form')
    {!! Form::close() !!}
  @endsection
```

En este caso, no estamos utilizando **Form::open** sino **Form::model** y el primer parámetro es la instancia del objeto **Noticia** que estamos editando. Tengamos en cuenta también que estamos asignado el método **put**, el cual es el apropiado para esta ruta. Pero como los formularios HTTP no soportan este método, se enviará un método **POST** y Collective se encargará de introducir un campo oculto indicando que nuestra intención es enviar un método **PUT**. Laravel tiene la capacidad de procesar ese campo y asociarlo a la ruta correspondiente.

Agreguemos también el código para cargar la vista para editar noticias en **Blog\Http\Controllers\Backend\NoticiaController**:

```
  public function edit($id){
    $noticia = Noticia::findOrFail($id);
    $categorias = Categoria::pluck('nombre', 'id');
    return view('backend.noticia.edit', ['noticia' => $no-
ticia, 'categorias' => $categorias]);
  }
```

Recordemos que es necesario obtener las categorías para cargar el **select**.

Validaciones con HTML5

HTML5 incorpora un conjunto de atributos en las etiquetas **<input>** que permiten realizar validaciones desde el navegador, las cuales podemos consultar en **www.w3schools.com/html/html_form_attributes.asp.** También encontraremos atributos relacionados con las validaciones que se pueden aplicar sobre las etiquetas **<form>.**

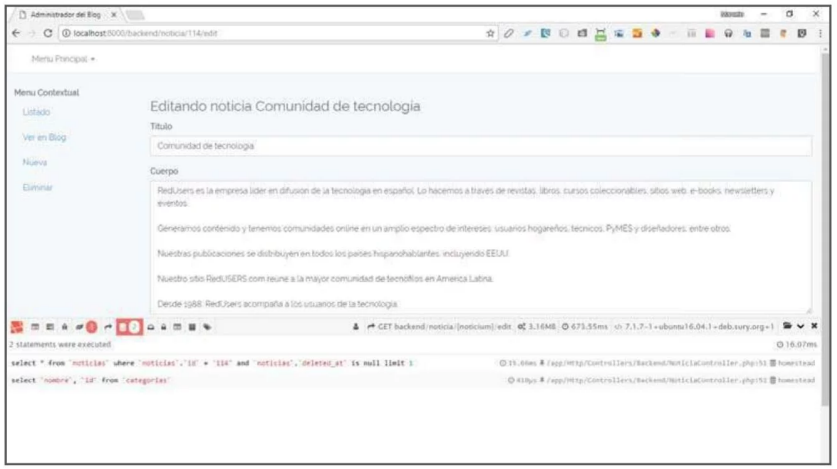


Figura 17. Collective posee la lógica para capturar la información y asignarla a cada uno de los componentes del formulario.

Por último, vamos a agregar el método **update** con la lógica para actualizar el registro:

```
  public function update(Request $request, $id){
    $noticia = Noticia::findOrFail($id);
    $validatedData = $request->validate(Noticia::getRules(
$id));
    $this->procesarImagen($request, $noticia);
    $noticia->update($validatedData);
    $noticia->save();
    $request->session()->flash('status', 'Noticia actualizada');
    return redirect()->route('backend.noticia.edit', $noti-
cia->id);
  }
```

Prestemos atención a que hemos separado la lógica de procesamiento de la imagen, dado que será la misma tanto para la creación como para la edición. De esta manera, podemos reutilizarla en ambos lugares:


```
  private function procesarImagen(Request $request, Noticia
$noticia){
    //Dado a que no es un dato obligatorio, consultamos si
hay imagen
    if ($request->hasFile('imagen')) {
      //Obtenemos una instancia de la clase Illuminate\
Http\UploadedFile
      $archivoImagen = $request->file('imagen');
      //Guardamos el archivo en el storage, en la carpeta
noticias/id_noticia
      //Este método retorna la ruta donde se almacena
      $path = $archivoImagen->storeAs('public/noticias/' .
$noticia->id, $archivoImagen->getClientOriginalName());

      $savedPath = str_replace("public/", "", $path);
      //Asignamos la ruta a el campo del objeto
      $noticia->imagen = $savedPath;
      //Actualizamos la base de datos para almacenar la
ruta
      $noticia->save();
    }
  }
```

Resumen Capítulo 11

En este capítulo comenzamos generando formularios con HTML, luego instalamos la librería Collective y fuimos implementándola analizando sus ventajas. Una vez generados los formularios, creamos la lógica para su procesamiento, primero realizando las validaciones con las herramientas que el framework nos ofrece para este fin. Luego continuamos con la visualización de errores y aplicamos mass assignment para la creación de objetos. También analizamos cómo procesar los archivos que se envían mediante formularios y las herramientas de Laravel relacionadas con el almacenamiento. Más adelante empleamos model binding para la edición, para lo cual también reutilizamos parte de la lógica desarrollada en la creación de noticias.

ACTIVIDADES

Test de Autoevaluación

1. ¿Por qué son importantes los formularios en las aplicaciones web?
2. ¿En qué consiste Collective?
3. ¿Cuál es la ventaja de implementar validaciones del lado del servidor?
4. ¿En qué consisten las reglas de validación de Laravel?
5. ¿Dónde es conveniente establecer las reglas de validación?
6. ¿En qué consiste la variable **$errors**?
7. ¿Para qué se utiliza la función **flash**?
8. ¿Es conveniente usar la función **move_uploaded_file** de PHP para procesar archivos en una aplicación Laravel?
9. ¿Qué son los espacios de almacenamiento?
10. ¿Para qué es útil el model binding?

Ejercicios prácticos

1. Cree todos los formularios para tener ABMS de todas las entidades del blog.
2. Cree un formulario de eliminación, tenga en cuenta enviar el método HTTP adecuado.
3. Modifique el modelo de datos para que las noticias puedan soportar más de un archivo y luego modifique el formulario para enviar más de una imagen por noticia.

12 Usuarios

Es muy difícil pensar en una aplicación web que no contemple a los usuarios como una entidad. Es por eso que Laravel, desde su instalación, provee migraciones y modelos para cubrir esta necesidad. El framework también brinda herramientas muy útiles para personalizar la experiencia de usuario, como la internacionalización, la autenticación y las notificaciones.

INTERNACIONALIZACIÓN

En el capítulo anterior implementamos validaciones en los formularios, que envían una respuesta al usuario indicando el motivo por el cual no se pueden procesar los datos enviados. Pero hasta ahora vemos todos los mensajes de error en inglés, debido a que no hemos configurado las opciones de internacionalización ni establecido los mensajes para el idioma español.

Primero, debemos modificar el archivo **config\app.php** y fijar el parámetro **locale** en **es**, de la siguiente manera: **'locale' => 'es',**. Este parámetro establecerá el código de idioma que utilizará el sistema; **es** para español, **en** para inglés. Los archivos donde se establecen los mensajes de error que aparecen actualmente se encuentran en **resources/lang**.

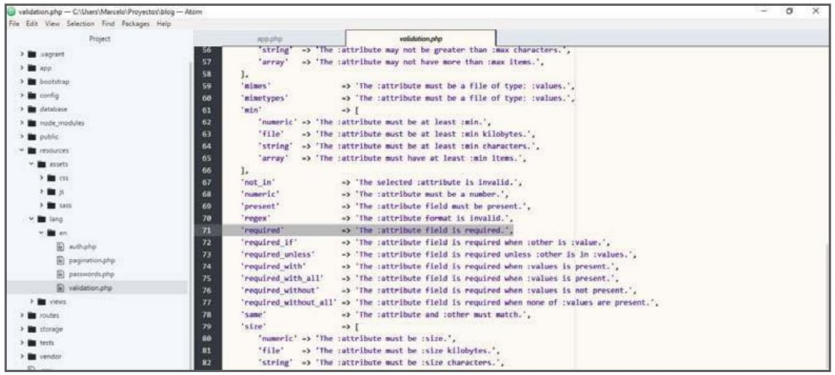


Figura 1. Debemos tener una carpeta por cada idioma que queramos incluir, el nombre debe concordar con lo establecido en la propiedad **locale**.

Códigos de idioma

Los códigos de idioma se toman a partir del estándar **ISO 639-1**, el cual ofrece un conjunto de lenguas y un código para cada una. Este código suele confundirse con otro estándar ISO, el **3166-1**, que brinda un código para cada país. Entonces, son dos códigos separados; por ejemplo, en la Argentina, el código de país es **ar**, mientras que el idioma **ar** corresponde al idioma árabe.

Podemos copiar la carpeta **en** en otra denominada **es** y cambiar manualmente cada mensaje. Sin embargo, recordemos que una de las ventajas de Laravel es que tiene una gran comunidad detrás, y uno de sus miembros generó un paquete, disponible en **https://github.com/Laraveles/spanish**, que podemos instalar vía Composer con **composer require laraveles/spanish**, que tiene las traducciones en español. Luego de haber descargado el paquete, ejecutamos **php artisan vendor:publish --tag=lang** para generar la carpeta **es** con las traducciones.

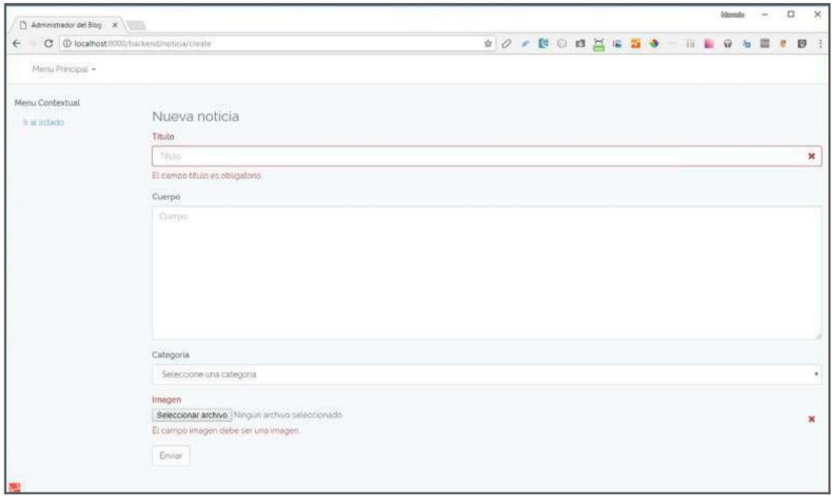


Figura 2. Con estos simples cambios, ya contamos con varios mensajes de error disponibles en español.

Esto no significa que nuestra aplicación funcione en ambos idiomas, ya que toda la interfaz se encuentra únicamente en español. Sólo los mensajes de error, cuya lógica es parte del framework, están internacionalizados.

Para lograr que la aplicación funcione también en inglés debemos realizar cambios en la interfaz. Tomemos como ejemplo el caso de **resources\views\backend\noticia\index.blade.php**, donde tenemos **@section('title','Listado de noticias')**.

Para convertir el texto **Listado de noticias** a diferentes idiomas, recurrimos a una función cuyo nombre es el doble guión bajo, es decir,

@section('title', __('noticias.listado')). Esta función buscará en la carpeta del idioma el archivo **noticias.php**. Éste deberá ser un array asociativo, en el cual establecemos el índice listado.

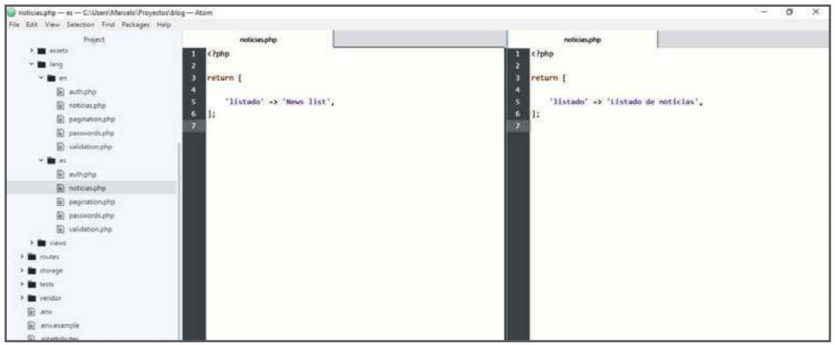


Figura 3. Cada vez que creemos un archivo nuevo para los recursos de idioma, debemos generarlo para todos los idiomas.

Si cambiamos **locale** en **config/app.php**, veremos las modificaciones correspondientes en la etiqueta **title**.

Observemos qué sucede con el caso de **Mostrando 15 noticias**. Con lo que sabemos hasta ahora, podemos utilizar dos cadenas y concatenarlas, pero sería mejor crear un mensaje que reciba un parámetro, lo cual podemos hacer con el carácter dos puntos.

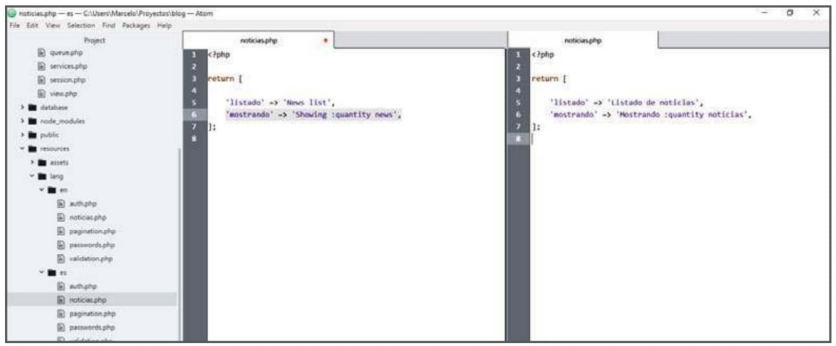


Figura 4. Los índices del array que se envía como parámetro deben coincidir con el nombre establecido en el mensaje.

Con esta modificación, el segmento donde mostramos la cantidad de noticias deberá quedar de la siguiente forma:

```
<h1>{{  __('noticias.mostrando', ['quantity' =>
count($noticias)]) }}</h1>
```

Recordemos que, al actualizar una noticia, enviamos un mensaje al usuario desde el controlador; allí también podemos hacer uso de las variables de internacionalización. Vamos a probarlo modificando el mensaje que enviamos en la edición:

```
$request->session()->flash('status', __('noticias.actual-
izada'));
```

Modificar el idioma

Para permitir que el usuario cambie el idioma, podemos utilizar la función **App::setLocale($locale);**, donde **$locale** será el código del idioma que deseemos establecer.

Es importante tener en cuenta la propiedad **fallback_locale**, que podemos configurar en **config/app.php**. Ésta establece qué idioma se utilizará cuando no se encuentre una traducción en particular.

Agreguemos una opción en el menú para cambiar el idioma. Para hacerlo, añadimos el siguiente contenido debajo de la etiqueta de cierre **
** de **<li class="dropdown">** que utilizamos para crear el menú principal:

Idioma por default

Cuando Laravel no encuentre una traducción en el idioma establecido ni en el de fallback_locale, se mostrará el código que enviamos en la función __(). Por lo tanto, si bien es necesario tener cargadas todas las traducciones, es recomendable asegurarse de tener completo el idioma establecido en fallback_locale. También podemos hacer uso de Lang::has('noticia.listado') para garantizar la existencia de una traducción y actuar en consecuencia.

```
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-
toggle="dropdown" role="button" aria-haspopup="true"
aria-expanded="false">{{ __('idioma.idioma') }} <span
class="caret"></span></a>
    <ul class="dropdown-menu">
      <li><a href="{{ route('backend.idioma.cambiar', ['lo-
cale' => 'en']) }}">English</a></li>
      <li><a href="{{ route('backend.idioma.cambiar', ['lo-
cale' => 'es']) }}">Español</a></li>
    </ul>
  </li>
```

Luego agregamos la ruta con el código para atender el request. Lo mejor sería agregar un controlador; no obstante, aplicamos este escenario para ahorrar pasos.

```
  Route::name('idioma.cambiar')->prefix('idioma')-
>get('cambiar/{locale}', function($locale){
    App::setLocale($locale);
    session()->flash('status', __('idioma.actualizado'));
    return redirect()->back();
  });
```

Es conveniente utilizar la internacionalización, aun teniendo un solo idioma, ya que permite concentrar la dialéctica del sistema en un único lugar, lo cual hará más fácil la tarea del corrector de comunicación.

Con este cambio podemos apreciar que el mensaje enviado por flash cambia en función del idioma que hayamos presionado, mientras que el resto de la interfaz se mantiene intacta debido a que **App::setLocale** no es persistente. Esto quiere decir que el idioma sólo se aplica en el **request** que se está ejecutando; es por eso que cuando se ejecuta **redirect()->back()**, seguimos viendo el idioma aplicado en la configuración. Para hacer esta lógica persistente es conveniente utilizar un **middleware**.

MIDDLEWARE

Los middlewares son componentes que podemos asociar a una o varias rutas y que se pueden ejecutar antes o después de la lógica asociada a la ruta.

Laravel dispone de varios middlewares e, incluso, hemos modificado uno para habilitar y deshabilitar la protección CSRF en el capítulo dedicado a los formularios.

Vamos a crear un middleware para establecer correctamente el idioma. Para hacerlo, disponemos del comando de Artisan **make:middleware**, el cual ejecutaremos de la siguiente forma: **php artisan make:middleware LanguageMiddleware**.

Todos los middlewares se encuentran en **app/Http/Middleware**; en esa carpeta podemos apreciar, además del que acabamos de generar, otros que provee el framework.

La lógica del middleware se ejecuta en un método denominado **handle**; iniciemos la del que acabamos de crear de la siguiente forma:

```
<?php

namespace Blog\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\Facades\App;

class LanguageMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
      if (Session::has('locale')) {
```

```
            App::setLocale(Session::get('locale'));
        }
        return $next($request);
    }
}
```

Podemos notar que nuestro middleware primero analizará los datos de sesión del **request** y, en caso de encontrar la variable **locale**, utilizará la función **App::setLocale**. De esta manera, lograremos la persistencia deseada dado que el middleware se ejecutará en todos los requests.

Luego de implementar este cambio, debemos establecer la variable de sesión al momento de cambiar el idioma. Entonces, antes de retornar la respuesta, agregamos la línea **session()->put('locale', $locale);** en la función que hemos creado para cambiar el idioma.

Figura 5. En este punto podemos ver que la sesión se establece, pero el idioma de la vista sigue sin modificarse.

Una vez que hemos definido la lógica del middleware, debemos registrarlo, para lo cual agregamos en el atributo **$middleware** de **app\Http\Kernel.php** la siguiente línea:

```
\Blog\Http\middleware\LanguageMiddleware::class
```

Una vez que tenemos el middleware registrado, vamos a definir los requests que queremos asociar. Debido a que esta funcionalidad será

útil en backend y en frontend, pero no en los servicios web, que no cuentan con sesión, será conveniente crear la asociación como parte de los middlewares que se ejecutan en el grupo web de **$middlewareGroups** en el archivo **app\Http\Kernel.php**.

Figura 6. Ahora podemos cambiar de idioma y mantener de manera persistente la opción seleccionada por el usuario.

Otra manera de asociar middlewares a requests es utilizando la función middleware en la generación de la ruta. Esto puede aplicarse tanto a un grupo como a una ruta individual; no obstante, primero es necesario registrar un alias en el atributo **$routeMiddleware** de **app\Http\Kernel.php** agregando la siguiente línea:

```
    'lang' => \Blog\Http\middleware\
LanguageMiddleware::class,
```

Documentación oficial

Como ya sabemos, los middlewares son componentes clave en la arquitectura del framework e, incluso, podemos ver que éste ya provee algunos para implementar. En el enlace **https://laravel.com/docs/5.5/middleware** encontraremos la documentación oficial sobre middlewares, además de algunos ejemplos e información sobre la administración de parámetros.

Luego, en la definición de la ruta podemos aplicar la función de la siguiente manera:

```
Route::resource('noticia', 'NoticiaController')->middleware('lang');
```

O incluso podemos hacerlo en la generación del grupo:

```
Route::namespace('Backend')->name('backend.')->middleware('lang');
```

AUTENTICACIÓN

Implementar el sistema de autenticación de Laravel es muy simple: primero debemos ejecutar **php artisan make:auth** y, luego, asegurarnos de contar con las tablas necesarias: **users** y **password_resets**. Nosotros las hemos creado previamente al ejecutar **php artisan migrate**.

Repasemos bien los principales componentes del sistema de autenticación de Laravel:

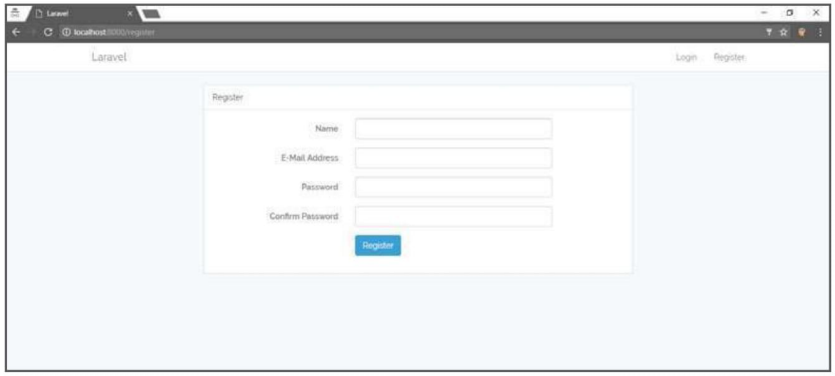


Figura 7. Con dos simples comandos, podemos obtener el formulario para registrar y loguear usuarios.

Rutas	Se generan con **Auth::routes();** y podemos ver esa instrucción en **routes\web.php.**
Vistas	Son cuatro y están en **resources\views\auth.** Hay una para el registro, una para el login, una para solicitar el restablecimiento de la clave y otra para resetear la clave. Todas utilizan un layout que se encuentra en **resources\views\layouts\app.blade.php.**
Controladores	También son cuatro y se corresponden con las mismas funcionalidades de las vistas.
Modelo	El único modelo que se utiliza es el que se encuentra en **app\User.php.**
Middlewares	Hay tres, y sus alias son **auth, auth.basic** y **guest.**

Una vez ejecutado el comando, asignamos las secciones de nuestra aplicación que requerirán autenticación. Esto se realiza a través del middleware **auth**, el cual, como ya sabemos, podemos aplicar en un grupo de rutas; en nuestro caso lo haremos con el grupo backend:

```
Route::namespace('Backend')->name('backend.')->prefix('/backend')->middleware('auth')->group(…
```

Con este cambio, cada vez que intentemos acceder a una ruta del backend, la aplicación retornará el formulario de login, que al estar basado en Bootstrap, mantiene una estética coherente con lo que hemos desarrollado hasta el momento. Si nos registramos y/o logueamos, el sistema nos redireccionará a la ruta **/home**.

Scaffolding

La técnica de scaffolding consiste en especificar una base de datos y luego, mediante diferentes técnicas de generación de código fuente, obtener un conjunto de elementos que nos permitan realizar todas las operaciones típicas de un ABM sobre las tablas del sistema. Esta técnica se hizo popular con el framework Ruby on Rails y más tarde fue adaptada por diferentes frameworks MVC, entre ellos, Laravel, para lograr la autenticación.

Personalizar la autenticación

El hecho de que Laravel nos dé todo resuelto no significa que tengamos que utilizarlo de la forma en la que nos lo brinda; podemos hacer modificaciones en el sistema de autenticación para adaptarlo a nuestras necesidades.

Comencemos por cambiar los redireccionamientos. Si utilizamos cualquiera de las funciones implementadas por la autenticación, observaremos que, luego de realizada la acción, se produce un redireccionamiento a la ruta **/home**. Esto se encuentra definido en los controladores mediante el atributo **$redirectTo**. Modifiquemos los controladores de manera tal que redireccionen al listado de noticias:

```
protected $redirectTo = '/backend/noticia';
```

Figura 8. En la pestaña Auth de Debugbar podemos ver la información del usuario autenticado.

En caso de querer introducir lógica al momento de realizar el redireccionamiento, podemos declarar un método **redirectTo**, que deberá retornar una cadena con la ruta a la cual redireccionar.

Las vistas que tenemos no ameritan cambios, sin embargo, podemos tomar algunos elementos presentes en **resources\views\layouts\app.blade.**

php y trasladarlos a nuestro layout. Analicemos el siguiente segmento del layout **app.blade.php**:

```
  <ul class="nav navbar-nav navbar-right">
    <!-- Authentication Links -->
    @guest
      <li><a href="{{ route('login') }}">Login</a></li>
      <li><a href="{{ route('register') }}">Register</a></li>
    @else
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-
toggle="dropdown" role="button" aria-expanded="false">
            {{ Auth::user()->name }} <span class="caret"></
span>
        </a>
        <ul class="dropdown-menu" role="menu">
          <li>
            <!-- Logout -->
            <a href="{{ route('logout') }}"
                onclick="event.preventDefault();
                         document.getElementById('logout-
form').submit();">
                Logout
            </a>
            <form id="logout-form" action="{{
route('logout') }}" method="POST" style="display: none;">
                {{ csrf_field() }}
            </form>
          </li>
        </ul>
      </li>
    @endguest
  </ul>
```

Vemos que mediante la instrucción **@guest** detecta si el usuario que accedió se encuentra autenticado o no. Nosotros no haremos este chequeo porque no daremos acceso a ninguna pantalla del backend sin autenticación. Sin embargo, podemos tomar parte de

código que empieza en **@else** e incorporarla a nuestro layout, ya que posee la lógica para recuperar el usuario autenticado y mostrar su nombre **{{ Auth::user()->name }}**, y también tenemos el código para armar el logout.

Observemos que esta lógica está dentro de un elemento **<ul class="nav navbar-nav navbar-right">**, que nos dará una opción de menú en la parte derecha de la barra de navegación y debemos agregarla debajo de nuestro actual **<ul class="nav navbar-nav">**.

Figura 9. Con este cambio podemos visualizar datos del usuario y permitirle desloguearse.

Ahora que sabemos cómo recuperar la información del usuario autenticado, podemos modificar el método **store** de **Blog\Http\Controllers\Backend\NoticiaController**, ya que habíamos dejado lógica provisoria para asignar al autor de las noticias. Con este cambio, es posible almacenar directamente al usuario logueado:

```
$noticia->autor = Auth::user()->id;
```

Debemos asegurarnos de agregar previamente en el controlador la declaración del Facade de autenticación que estaremos utilizando mediante **use Illuminate\Support\Facades\Auth;**.

Agregar campos en la registración

El formulario de registro solicita cuatro campos: nombre, email, password y confirmación del password. Recordemos que nuestro blog también permite a los usuarios cargar un avatar, por lo tanto, vamos a aplicar los cambios correspondientes para que puedan hacerlo al momento de registrarse.

Primero, modificamos la vista para darle la posibilidad de subir el archivo en el formulario, agregando lo siguiente debajo del div de la repetición del password:

```
  <div class="form-group{{ $errors->has('avatar') ? ' has-
error' : '' }}">
      <label for="avatar" class="col-md-4 control-
label">Avatar</label>
      <div class="col-md-6">
          <input id="avatar" type="file" class="form-con-
trol" name="avatar">
          @if($errors->has('avatar'))
              <span class="help-block">
                  <strong>{{ $errors->first('avatar') }}</
strong>
              </span>
          @endif
      </div>
  </div>
```

Recordemos que debemos agregar el atributo **enctype="multipart/form-data"** en el formulario porque, de no hacerlo, no se estará enviando el archivo adjunto.

En esta oportunidad no implementamos Collective para mantener consistencia con el resto del código presente en el archivo. Ahora debemos modificar la lógica para validar y persistir este nuevo dato.

Modificamos la función **validator** de **Blog\Http\Controllers\Auth\RegisterController** del siguiente modo:

```
  protected function validator(array $data)
  {
```

```
        return Validator::make($data, [
            'name' => 'required|string|max:255',
            'email' => 'required|string|email|max:255|unique
:users',
            'password' => 'required|string|min:6|confirmed',
            'avatar' => 'image|max:2048'
        ]);
    }
```

Luego, cambiamos la función **create** para procesar la imagen, crear la instancia de Avatar y asociarla al usuario:

```
    protected function create(array $data)
    {
        $user = User::create([
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => bcrypt($data['password']),
        ]);

        //Obtenemos la instancia de $request
        $request = request();

        if ($request->hasFile('avatar')) {
            //Procesamos el archivo de la misma manera que en las
noticias
            $archivoImagen = $request->file('avatar');
            $path = $archivoImagen->storeAs('public/avatar/' .
$user->id, $archivoImagen->getClientOriginalName());
            $savedPath = str_replace("public/", "", $path);
            //Creamos la instanacia de Avatar y la asociamos al
usuario
            $avatar = new Avatar();
            $avatar->user_id=$user->id;
            $avatar->img_location = $savedPath;
            $avatar->save();
```



```
        }
        //Retornamos el usuario creado
        return $user;
    }
```

Recordemos declarar al inicio del controlador el modelo Avatar con **use Blog\Models\Avatar;**.

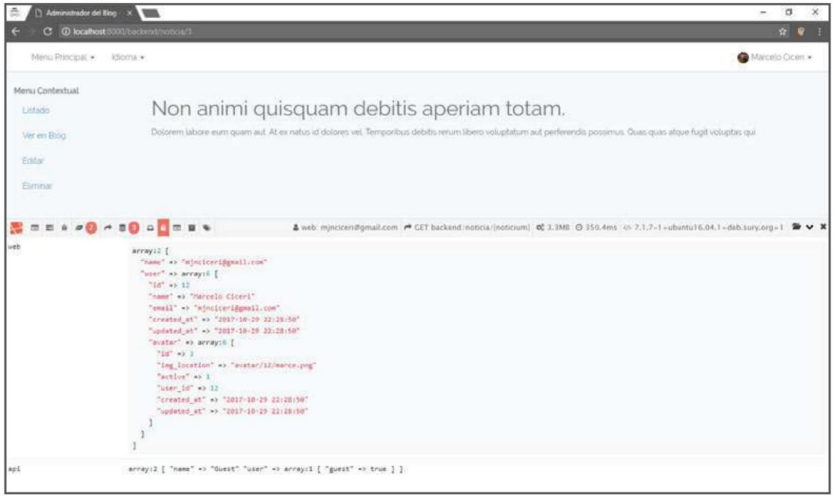


Figura 10. Ahora que contamos con el avatar, podemos utilizarlo para acompañar el nombre de usuario en el layout.

Podemos incluir el avatar en el layout mediante el siguiente código:

Documentación oficial

La documentación oficial brinda más información sobre diferentes formas de personalización, que incluyen, entre otras cosas, métodos de persistencia personalizados mediante los **Guard** y también acerca de otras formas de autenticación que pueden emplearse. Podemos consultarla en **https://laravel.com/docs/5.5/authentication.**

```
  <a href="#" class="dropdown-toggle" data-
toggle="dropdown" role="button" aria-expanded="false">
    @if(Auth::user()->avatar)
      <img src="{{ asset('storage/' . Auth::user()->avatar-
>img_location) }}" class="img img-reponsive img-circle"
style="max-height: 20px" />
    @endif
    {{ Auth::user()->name }} <span class="caret"></span>
  </a>
```

Enviar correos

Ya tenemos nuestro formulario de registro y autenticación funcionando; sin embargo, si probamos la funcionalidad de restablecer contraseña, obtendremos el siguiente error:

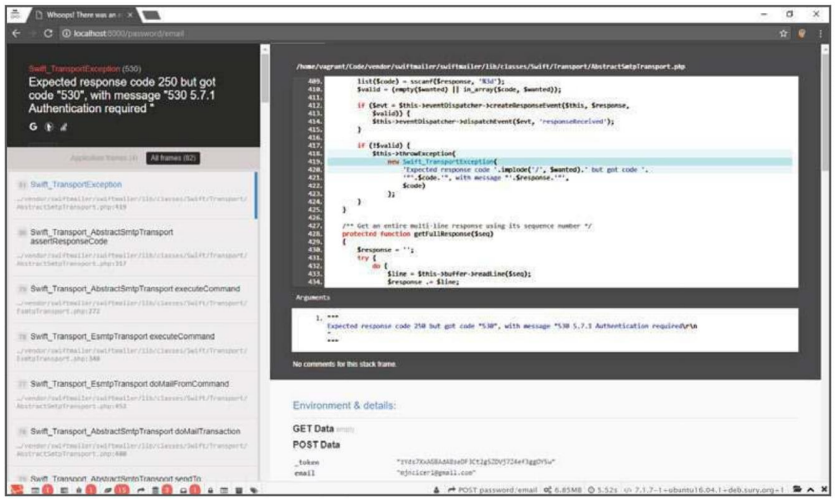


Figura 11. Se produce un error al intentar enviar el correo electrónico con el enlace para restablecer la clave.

El servidor de correo se configura en **config\mail.php**. No obstante, veremos que la mayoría de las opciones se recupera desde variables de entorno.

La principal variable de configuración se denomina **driver**, ya que establece cuál será el servicio que utilizaremos para enviar correos. Si usamos Homestead, encontraremos las variables de entorno establecidas para el servicio **Mailtrap**.

Mailtrap, **https://mailtrap.io**, es un servicio que provee un servidor SMTP de pruebas muy útil para ambientes de desarrollo. Su función es crear, visualizar y enviar correos previniendo el envío a usuarios reales durante la fase de desarrollo y/o pruebas.

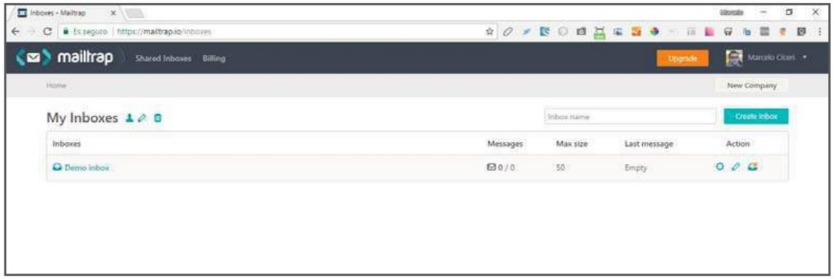


Figura 12. Mailtrap nos provee una casilla de correo en las cuentas gratuitas, en la cual se recibirán todos los mensajes enviados por el SMTP.

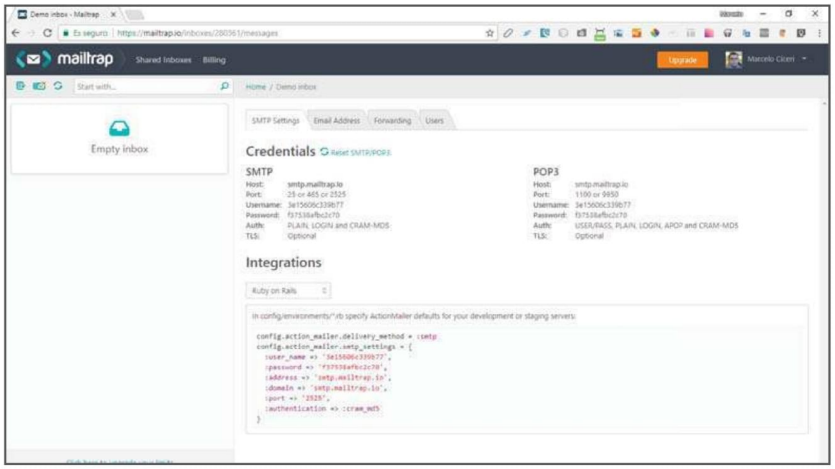


Figura 13. En el detalle de la casilla obtendremos las credenciales del servidor SMTP que debemos aplicar en nuestro sistema.

Básicamente, Mailtrap captura todos los correos que se envían desde el SMTP y permite visualizarlos en esta casilla, sin importar a cuál haya sido disparado. Pero como emula las funciones de un SMTP real, sí es necesario que el correo sea válido.

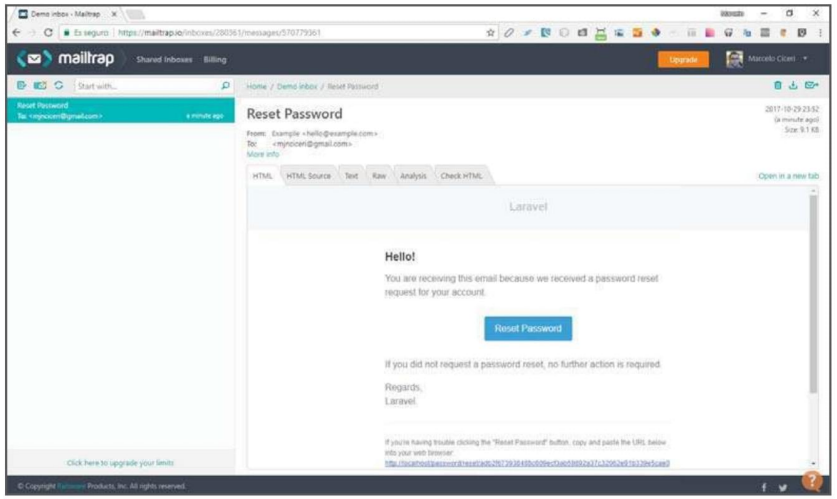


Figura 14. Podemos observar varios detalles incluyendo las direcciones de los correos desde y hacia donde se realizó el envío.

Si queremos personalizar el correo electrónico que se envía, debemos generar una notificación. Podemos consultar la manera de hacerlo en la sección correspondiente de este mismo capítulo.

Socialite

Es muy común encontrar sitios web que permiten autenticarnos con las credenciales de otros sistemas. Esto se logra gracias a un protocolo de autenticación denominado oAuth 2.0, que consiste en un mecanismo para autenticar usuarios entre sistemas sin compartir las credenciales.

Socialite es un paquete oficial de Laravel que contiene adaptadores de este protocolo para permitir la autenticación con diferentes sistemas, entre ellos, Facebook, Twitter, Google, LinkedIn, GitHub y Bitbucket. Su instalación es muy sencilla debido a que podemos hacerlo

mediante Composer con **composer require laravel/socialite**. Sin embargo, la configuración depende mucho de la implementación que deseemos hacer y del proveedor que utilicemos.

Vamos a realizar un ejemplo implementando la autenticación mediante Facebook.

Una vez instalado Socialite, incorporamos el siguiente método en **Blog\Http\Controllers\Auth\LoginController**:

```
public function redirectToProvider(){
  return Socialite::driver('facebook')->redirect();
}
```

Este código realizará el redireccionamiento al proveedor de autenticación, que en este caso será Facebook, para lo cual utilizamos una función **redirect** de Socialite. Luego, debemos implementar en el mismo archivo el siguiente método:

```
public function handleProviderCallback(){
    $user = Socialite::driver('facebook')->user();
    //$user nos dará una clase con los datos que obtengamos del proveedor.
     var_dump($user); die();
}
```

Esta lógica es la que se ejecutará en el callback; una traducción posible de este término sería **llamada de regreso** o **llamada de**

Documentación oficial

Laravel propone en su documentación oficial un conjunto de paquetes oficiales. En varios de ellos podemos apreciar que Taylor Otwell, el creador de Laravel, contribuye activamente, como en el caso de **https://github.com/laravel/socialite**. No obstante, podemos construir nuestros propios paquetes; en caso de querer hacerlo, es recomendable seguir la guía disponible en **https://laravel.com/docs/5.5/packages**.

vuelta. Es necesario considerar el flujo de navegación que tendrá el usuario, el cual se describe a continuación:

1	Ingresará en la aplicación, que le solicitará autenticarse.
2	Presionará el botón **Login con Facebook** en nuestra aplicación y será redireccionado a Facebook; esto lo hace el método **redirectToProvider**.
3	En Facebook hará la autenticación y le aparecerá un cuadro de diálogo indicando que sus datos serán enviados a Blog con Laravel. Una vez que acepte los datos, Facebook deberá redireccionar al usuario otra vez a nuestra aplicación, para lo cual tendrá que saber qué URL invocar (de ahí surge el término callback). La URL será la de la ruta que creemos y asociemos al método **handleProviderCallback**.
4	El usuario finalmente volverá a estar en nuestra aplicación y autenticado con sus datos de Facebook.

Naturalmente, debemos crear las rutas para estos nuevos métodos:

```
Route::get('login/facebook', 'Auth\LoginController@redirectToProvider');
Route::get('login/facebook/callback', 'Auth\LoginController@handleProviderCallback');
```

Es importante tener en cuenta que éstas no deben estar en el grupo Backend, sino accesibles sin ninguna restricción. Una vez realizados los cambios en la aplicación, vamos a crear y configurar una aplicación en Facebook, para lo cual debemos realizar los siguientes pasos:

Usuarios finales

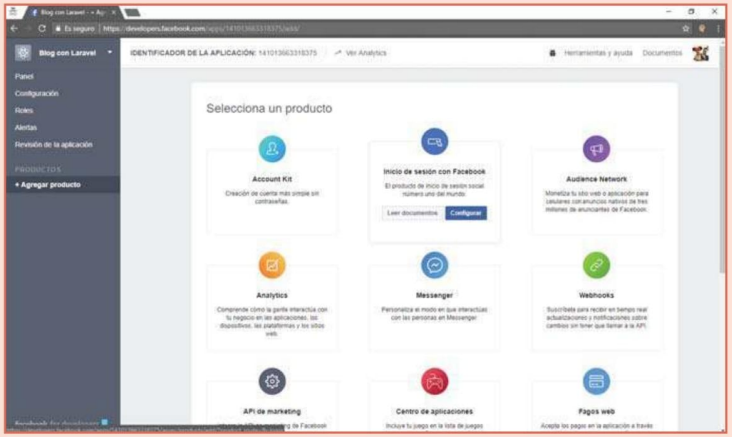
El término usuario suele emplearse de diversas maneras y en distintas circunstancias, lo cual lo hace un vocablo muy amplio, sobre el que se tiende a realizar clasificaciones. Una de las más conocidas es la del usuario final, que suele usarse para separar a las personas que utilizarán el sistema, de los programadores que lo desarrollan.

CONFIGURAR INICIO DE SESIÓN CON FACEBOOK

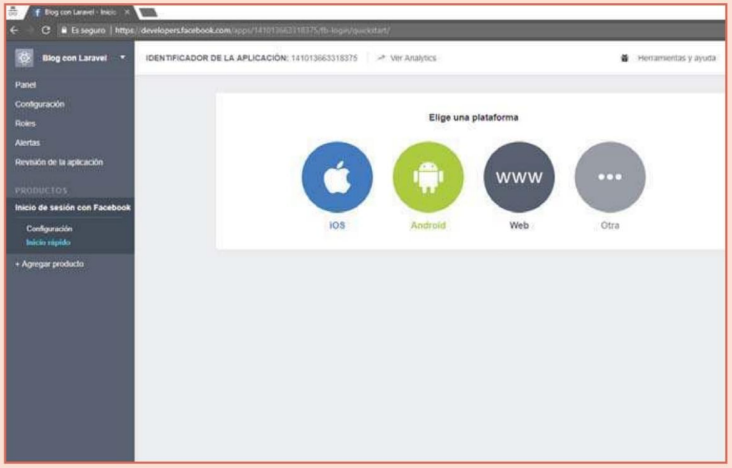
01 Para comenzar, ingrese en la dirección **https://developers.facebook.com**. Inicie sesión con su cuenta de Facebook y, en caso de no contar con una, regístrese. Luego, en el menú superior derecho, seleccione la opción **crear agregar una aplicación.**

02 En el cuadro **para crear un nuevo identificador de la aplicación** ingrese primero el nombre de su aplicación Laravel. Tenga en cuenta que éste será el nombre que Facebook utilizará para solicitar la confirmación al usuario al momento de compartir los datos.

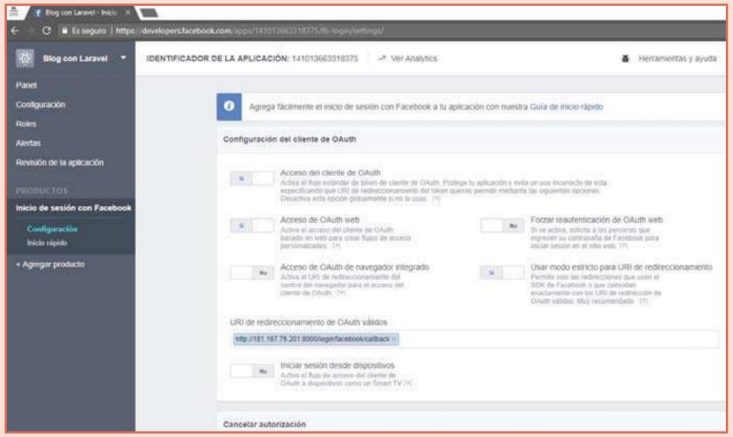
03 En la pantalla **selecciona producto** presione el botón **configurar** dentro de la opción **Inicio de sesión con Facebook**.



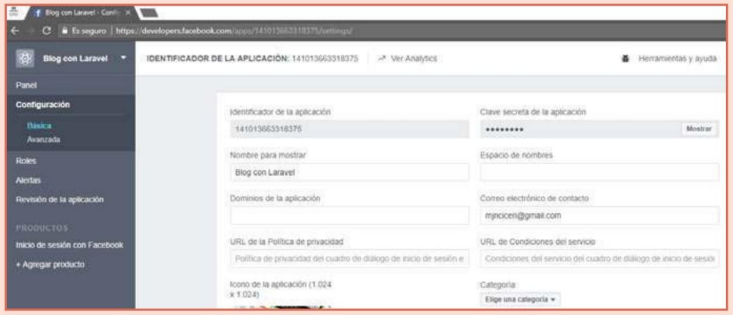
04 En la pantalla siguiente seleccione en el menú lateral izquierdo la opción **Configuración** del menú **Inicio de sesión con Facebook**.



05 En la próxima pantalla agregue la URL de la ruta que funcionará como **callback** en su aplicación Laravel. Tenga en cuenta que tiene que ser posible acceder a dicha URL desde Internet, y debe formar parte de los parámetros de autenticación, por lo que tiene que coincidir exactamente. Si utiliza Homestead incluya el :8000. Presione **guardar cambios**.



06 En el menú lateral izquierdo seleccione la opción **básica** dentro de **configuración**, copie el identificador de la aplicación y la clave secreta de la aplicación. Es recomendable introducir una imagen porque ésta aparecerá en el cuadro de diálogo de confirmación de envío de datos. Presione **guardar cambios**.



No olvidemos introducir el botón Login con Facebook en la vista, con la ruta hacia el método correspondiente:

```
<a href="{{ action('Auth\LoginController@redirectToProvider') }}" class="btn btn-primary">
    Login con Facebook
</a>
```

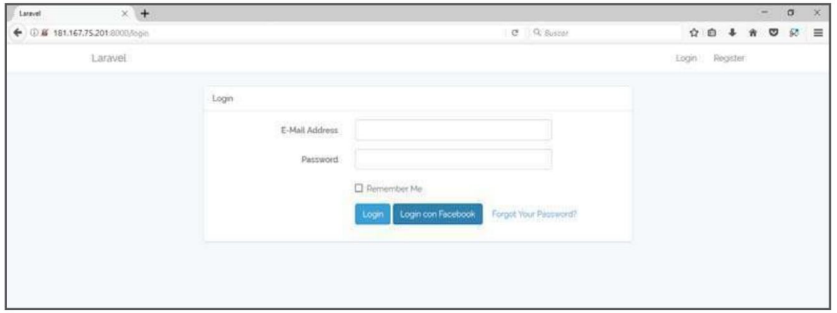


Figura 15. Recordemos que es necesario ingresar con la IP o dominio que hemos configurado en la aplicación creada en Facebook.

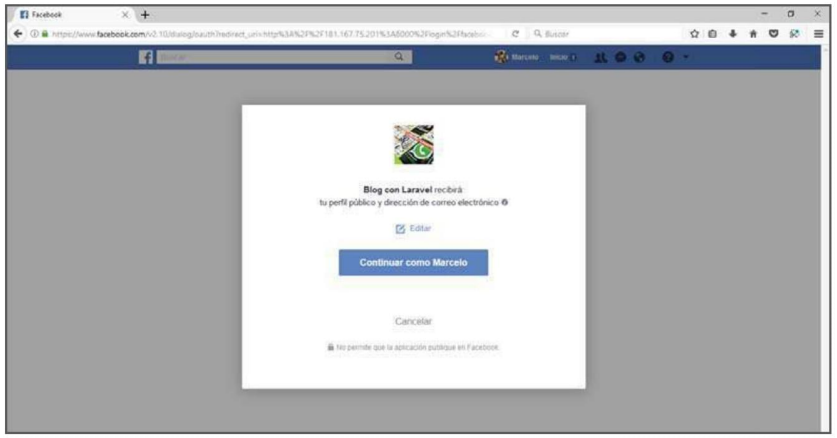


Figura 16. Es importante configurar bien la aplicación en Facebook para que el usuario entienda que estará enviando sus datos a nuestro blog.

Figura 17. Con la función **var_dump** de PHP podemos ver los atributos del objeto de la clase **Laravel\Socialite\Two\User**.

Al finalizar el proceso, obtendremos una instancia con los datos del proveedor. En este punto, dependerá de la lógica de negocios de nuestra aplicación decidir qué hacer con esos datos. En nuestro blog podemos optar por realizar lo siguiente:

```
public function handleProviderCallback(){
    $socialiteUser = Socialite::driver('facebook')->user();

    //Revisamos si el usuario ya existe
    $blogUser = BlogUser::where('email', $socialiteUser->getEmail())->first();

    if($blogUser){
      //Si el usuario ya existe, lo autenticamos manualmente
      Auth::login($blogUser);
      return redirect($this->redirectTo);
    }else{
      //Creamos un nuevo usuario
      $blogUser = new BlogUser();
      $blogUser->email = $socialiteUser->getEmail();
      $blogUser->name = $socialiteUser->getName();
      //Guardamos un password random para que la DB no arroje error
```

```
        $blogUser->password = substr(md5(mt_rand()), 0, 10);
        $blogUser->save();
        Auth::login($blogUser);
        return redirect($this->redirectTo);
    }
}
```

NOTIFICACIONES

En la actualidad existen diversos canales de comunicación: ya no utilizamos sólo el e-mail, sino que es muy normal recibir un mismo mensaje también por SMS o por la mensajería de alguna red social. Laravel asume esta problemática desde un espacio transversal que permite separar el mensaje, del canal de comunicación. Vamos a analizarlo mediante la creación de un ejemplo. Incorporaremos una notificación que informe al usuario que su cuenta fue creada. Empecemos por ejecutar el comando **php artisan make:notification Bienvenido**. Éste generará un archivo en **app\Notifications\Bienvenido.php** cuyo contenido será una clase de nombre **Bienvenido**, la cual hereda de **Notification**.

Como hemos visto, Laravel ya se encargó por nosotros de realizar la integración con Mailtrap y otros servicios de envío de correo electrónico para que podamos concentrarnos en la generación del correo que queremos enviar.

Observemos cada uno de los métodos que presenta esta clase:

via	Define los canales a través de los cuales se enviará la notificación, estableciendo el mail por default.
toMail	En este método estará la lógica para enviar la notificación por correo electrónico. La nomenclatura sigue la regla **to + channel**; por ejemplo, si queremos introducir un método para el envío de notificaciones mediante Facebook, deberíamos declararlo **toFacebook**.
toArray	En este método introducimos la lógica para mandar una notificación a la base de datos, para así tener un historial de todas las notificaciones enviadas.

Notificaciones por e-mail

Modifiquemos el método **toMail** de la siguiente forma:

```
public function toMail($notifiable){
  return (new MailMessage)
    ->subject('Alta de usuario')
    ->greeting('¡Bienvenido al blog de Laravel!')
    ->line('Para comenzar a cargar tus noticias ingresá en ')
    ->action('Blog', url('/backend/noticia'))
    ->line('Gracias por haberte registrado!')
    ->salutation('Blog con Laravel');
}
```

Observemos que el método espera recibir el parámetro **$notifiable**, el cual puede ser cualquier clase que implemente el trait **Illuminate\Notifications\Notifiable;**. Notemos que ya está implementado en **Blog\User**.

```
<?php

namespace Blog;

use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;
```

Versiones 5.3 y anteriores

La implementación de Socialite descripta funcionará en versiones 5.4 en adelante. Para las anteriores es necesario utilizar Socialite 2.0, cuya documentación oficial podemos consultar en **https://github.com/laravel/socialite/tree/2.0**. Tengamos presente también que, si no contamos con auto-discovery, deberemos modificar el archivo **config/app.php** tal como lo establece la documentación oficial.

De esta manera, con sólo instanciar la clase **Bienvenido** que acabamos de crear y utilizando el método **notify** que nos brinda el trait, podemos enviar nuestra notificación. Vamos a modificar en **Blog\Http\Controllers\Auth\RegisterController** e introducir la siguiente línea antes de retornar el usuario creado:

```
        $user->notify(new Bienvenido());

        return $user;
    }
```

No olvidemos declarar al comienzo el namespace use **Blog\Notifications\Bienvenido;**.

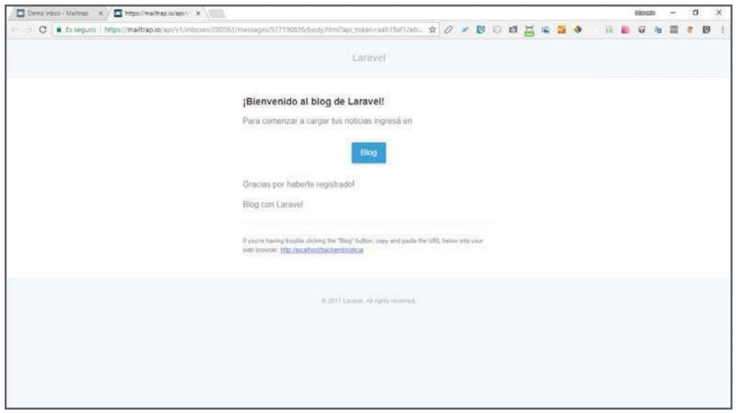


Figura 18. Si registramos un nuevo usuario, podremos ver el correo enviado en Mailtrap.

Observemos que el correo ya tiene un formato específico, cuyo contenido puede manipularse haciendo uso de los métodos que ofrece **MailMessage**.

Podemos consultarlos en **https://laravel.com/api/5.5/Illuminate/Notifications/Messages/MailMessage.html**.

Personalizar el formato

Como ya sabemos, Laravel siempre brinda una solución fácil y rápida para situaciones comunes, y también nos permite modificar dicha solución para mejorarla y/o adaptarla a nuestras necesidades.

Mediante el comando **php artisan vendor:publish --tag=laravel-notifications** se genera la carpeta **resources/views/vendor/notification**, en la cual podremos acceder al template utilizado para generar el correo, para modificar todos los aspectos que deseemos, tanto en la estructura HTML como en los estilos.

Otra opción que podemos utilizar, en lugar de hacer uso del formato preestablecido por Laravel, es manejar una vista. Esto se realiza de la siguiente forma:

```
public function toMail($notifiable){
  return (new MailMessage)->view(
      'emails.backend.', ['invoice' => $this->invoice]
  );
}
```

A su vez, Laravel ofrece el uso de varios templates que pueden generarse mediante Markdown. Este formato tiene la ventaja de que el framework se encarga de procesar y generar correos que se adaptan a distintos formatos y que, mediante el uso de componentes, permite escribir correos personalizados a diferentes contextos.

Vamos a crear un correo del tipo MarkDown para confirmar la creación de una nueva noticia al autor. Empecemos por ejecutar el comando **php artisan make:notification NoticiaNueva --markdown=mail.noticia.nueva**. Observemos que éste genera una nueva clase que también hereda

Versiones 5.2 y anteriores

Esta implementación funcionará en versiones 5.4 en adelante; las anteriores no implementan una capa transversal de notificaciones. No obstante, para los correos electrónicos podemos utilizar en esas versiones el Facade **Mail**, cuyo funcionamiento está disponible en **https://laravel.com/docs/5.2/mail**.

de **Notification** e implementa un método **toMail**; no obstante, hace uso de un método **markdown**:

```
  public function toMail($notifiable){
    return (new MailMessage)->markdown('mail.noticia.nue-
va');
  }
```

Este comando también ha generado una vista en **resources\views\mail\noticia\nueva.blade.php**, pero si la observamos con atención, no parece ser una vista HTML:

```
@component('mail::message')
# Introduction

The body of your message.

@component('mail::button', ['url' => ''])
Button Text
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

Podemos analizar el formato iniciando sesión en Tinker y enviando la notificación que acabamos de crear.

Markdown

El formato Markdown tiene como objetivo lograr una sintaxis simple de leer y de escribir que, a partir de texto plano, pueda convertirse en formato HTML. Inicialmente, esta conversión se lograba con el lenguaje Perl, y luego, varios lenguajes (entre ellos, PHP) se adaptaron. Podemos encontrar más información al respecto en la documentación oficial disponible en **https://daringfireball.net/projects/markdown**.

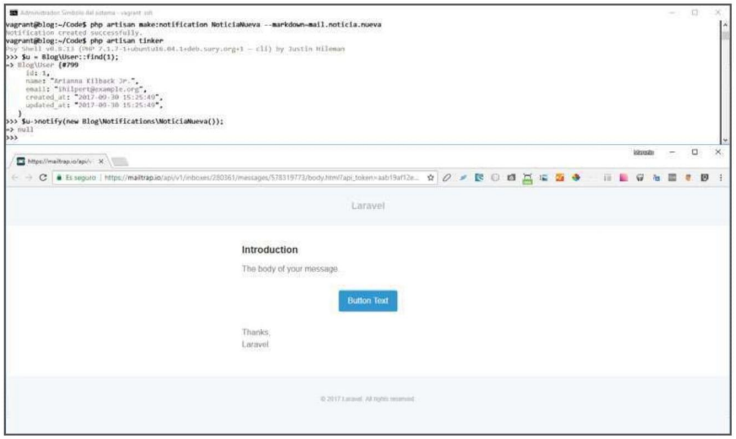


Figura 19. La vista genera un e-mail en formato HTML mediante el uso de componentes.

A través de Markdown, además de la sintaxis propia de este lenguaje, disponemos de algunos componentes que podemos utilizar para mejorar el formato de nuestras notificaciones.

Entre ellos destacamos Panel y Table, cuyo formato podemos consultar en **https://laravel.com/docs/5.5/mail#markdown-mailables**.

Modifiquemos el formato actual de **resources\views\mail\noticia\nueva.blade.php** por lo siguiente:

```
  @component('mail::message')

  {{ $noticia->creadaPor->name }} creó una nueva noticia el
**{{ $noticia->created_at }}**:

  # {{ $noticia->titulo }}

  @component('mail::panel')
  {{ $noticia->cuerpo }}
  @endcomponent
```

```
@component('mail::button', ['url' => $url])
Ver noticia en el Blog
@endcomponent

Saludos,<br>
{{ config('app.name') }}
@endcomponent
```

En este formato podemos ver que hacemos uso de la variable **$noticia** como en cualquier otro template de Blade, por lo que debemos asegurarnos de enviar ese parámetro a la vista:

```
public function toMail($notifiable){
  return (new MailMessage)
    ->markdown('mail.noticia.nueva',
      [
       'noticia' => $this->noticia,
       'url' => route('frontend.noticia.show', ['id' =>
$this->noticia->id])
      ]
    );
}
```

Vemos que estamos utilizando **$this->noticia**, debido a que, en caso de que necesitemos datos para una notificación, tenemos que establecerlo en el constructor de la misma.

Notificaciones masivas

Es importante tener en cuenta que, si queremos realizar una notificación en forma masiva, lo ideal es encolar el envío, sobre todo si el canal depende de un servicio externo, ya que, de no hacerlo de esta manera, podríamos generar un cuello de botella en el sistema. Es recomendable consultar **https://laravel.com/docs/5.5/notifications#queueing-notifications** y también **https://laravel.com/docs/5.5/queues.**


```
public function __construct(Noticia $noticia){
    $this->noticia = $noticia;
}
```

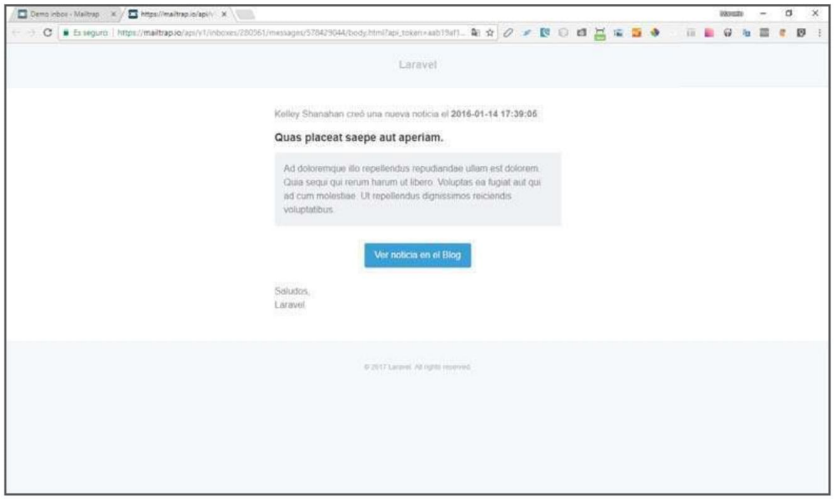


Figura 20. Éste es el formato final que obtendremos; notemos que el componente **Panel** permite destacar el cuerpo de la noticia.

Por último, vamos a modificar el método **store** de **Blog\Http\Controllers\Backend\NoticiasController** agregando la siguiente línea antes de realizar el redireccionamiento:

```
Auth::user()->notify(new NoticiaNueva($noticia));
```

Base de datos

Contar con un historial de notificaciones es muy útil porque es una forma sencilla de ver y analizar las interacciones que el usuario tiene con el sistema. Para comenzar, ejecutamos el comando **php artisan notifications:table**, que generará un archivo con la migración para almacenar la información en la base de datos. Luego ejecutamos **php artisan migrate**.

A continuación, debemos implementar en el método **toArray** de la notificación aquellos campos que son necesarios para el envío de ésta; en el caso de **Bienvenido** podemos dejarlo vacío, mientras que en **NoticiaNueva** quedaría de la siguiente manera:

```
  public function toArray($notifiable){
    return [
      'noticia' => $this->noticia,
      'url' => route('frontend.noticia.show', ['id' =>
$this->noticia->id])
    ];
  }
```

Por último, recordemos modificar el método **via** de ambas notificaciones para añadir el nuevo canal:

```
  public function via($notifiable){
    return ['mail', 'database'];
  }
```

Figura 21. **Illuminate\Notifications\Notifiable** ofrece métodos para recuperar y manipular las notificaciones.

Observemos que en el atributo **type** tenemos la clase utilizada para enviar la notificación y, a su vez, hay un campo **read_at** marcado en **null**. Cada notificación cuenta con un método **marsAsRead();** que establece dicho campo. Si queremos recuperar todas las notificaciones no leídas, podemos hacerlo con **$user->unreadNotifications**.

Tengamos presente que en estos ejemplos no hemos utilizado la internacionalización para el contenido debido a que el propósito era mantener el foco en los cambios que estamos buscando con relación a las notificaciones. Un aspecto importante en los sistemas y muy poco presente en los programadores es el de la comunicación con el usuario.

Es relevante definir y consensuar un estilo de comunicación y transmitirlo a todo el equipo de trabajo, de manera que exista una experiencia de usuario uniforme. Puede resultar incómodo para un usuario usar diferentes voces o formas y latiguillos propios de una región. A su vez, hay otros elementos que acompañan a la experiencia de usuario; podemos encontrar una guía muy completa en la dirección **https://blog.hubspot.es/marketing/branding-e-identidad-corporativa-guias-de-estilo**.

Resumen Capítulo 12

En este capítulo estudiamos las principales herramientas que brinda el framework para personalizar la experiencia de usuario. Comenzamos con la internacionalización, es decir, la capacidad de añadir más de un idioma, y mediante los middlewares vimos cómo mantener persistente su selección. Luego implementamos la autenticación de Laravel, modificamos algunos elementos para adaptarlos a nuestro blog e introdujimos también la autenticación por Facebook mediante Socialite. Para finalizar, trabajamos con notificaciones implementando el canal de e-mail básico de Laravel, más tarde con Markdowns y, por último, almacenando dichas notificaciones en la base de datos, para luego poder recuperarlas y manipularlas.

ACTIVIDADES

Test de Autoevaluación

1. ¿En qué consiste la internacionalización?
2. ¿Por qué al establecer el idioma con la función **App::setLocale** la selección no se mantiene?
3. ¿Cuál es la relación entre un middleware y las rutas?
4. ¿Cuáles son los principales componentes que brinda el sistema de autenticación de Laravel?
5. ¿Es posible agregar más campos al usuario que provee Laravel?
6. ¿Para qué se utiliza Socialite?
7. ¿Cuáles son los principales proveedores de identidad que podemos utilizar mediante Socialite?
8. ¿Es posible implementar Socialite si nuestro equipo no puede ser accedido por Internet?
9. ¿En qué consisten las notificaciones?
10. ¿En qué consiste un canal?

Ejercicios prácticos

1. Implemente la internacionalización de la aplicación en al menos dos idiomas.
2. Implemente la autenticación mediante algún proveedor de identidad diferente de Facebook.
3. Implemente notificaciones para la edición y eliminación de una noticia, tanto por e-mail como por base de datos.

13 Testing

Como ya sabemos, Laravel se apoya en muchas herramientas existentes para implementar gran parte de sus funcionalidades, y el testeo no es la excepción. En este capítulo estudiaremos los principales conceptos que envuelven a las técnicas de pruebas automatizadas y, también, PHPUnit junto a Laravel Dusk, las dos herramientas en las cuales el framework se basa para permitirnos comprobar nuestros desarrollos.

CONCEPTOS BÁSICOS DE TESTING

La informática es el estudio de las técnicas de procesamiento de datos para generar información. En otras palabras, cuando hablamos de sistemas también nos referimos a datos de **entrada**, **procesamiento** e información de **salida**.

Al diseñar y programar sistemas, tenemos conocimiento de las entradas y salidas, de manera que si logramos aislar segmentos de nuestro sistema, podemos ejecutar pequeñas piezas de código con diferentes entradas y **asegurar** que la salida sea la que corresponda para ese caso. En esto consisten las pruebas automatizadas: básicamente, establecemos una serie de datos que son procesados por el código del sistema y, luego, comparamos los resultados obtenidos con los esperados. Antes de analizar en profundidad este tema y poner manos a la obra, es necesario familiarizarse con los siguientes conceptos:

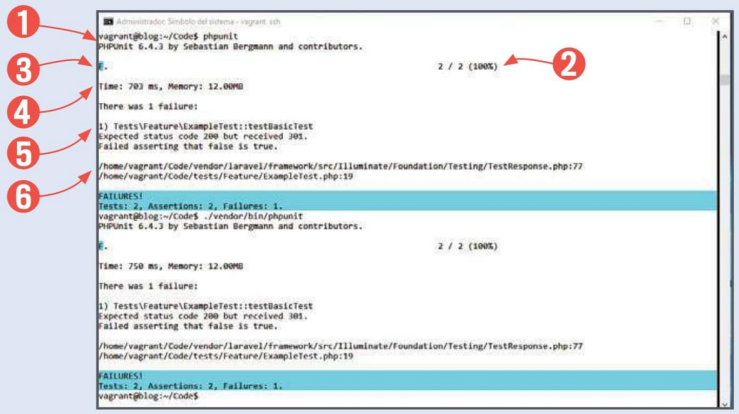
Prueba unitaria	Implica probar una unidad de software del sistema, pero aislada del resto, lo cual nos garantiza que dicha unidad tendrá un correcto funcionamiento operando independientemente del resto del sistema.
Prueba de integración	Suele ejecutarse luego de una prueba unitaria y consiste en realizar pruebas en grupo entre varias unidades de software.
Caso de prueba	Describe el escenario en el cual se deberá ejecutar una prueba, introduciendo las entradas y las salidas esperadas.
Plan de pruebas	Es la ejecución de varios casos de prueba, que pueden diseñarse de acuerdo con distintos focos, por ejemplo, la actividad de un usuario o los estados de las noticias.
Regresión	Significa realizar pruebas que hayan sido ejecutadas previamente con éxito y que, ante algún cambio en el sistema, se mantengan de forma satisfactoria.
Mock	En ocasiones, resulta muy complejo generar el contexto adecuado para la ejecución de una prueba, por lo que se crean objetos que simulan los datos y/o comportamientos necesarios para las entradas y servicios del contexto.

PHPUnit

PHPUnit es un framework para realizar pruebas automatizadas perteneciente a la familia de xUnit, la cual está compuesta por diversos frameworks para diferentes lenguajes de programación; todos siguen la misma estructura del original sUnit, que estaba preparado para el lenguaje SmallTalk. Como muchos de los componentes que forman parte de Laravel, se puede utilizar tanto dentro como fuera del framework y es instalable vía Composer mediante el comando **composer require --dev phpunit/phpunit**. Recordemos que la opción **--dev** indica que dicha librería es necesaria únicamente para desarrollo.

PHPUnit también puede instalarse de manera global en toda la máquina, como sucede en Homestead, lo cual nos permite ejecutar el comando **phpunit** desde cualquier lugar de la consola.

RESULTADO DEL COMANDO PHPUNIT



1 En la línea **PHPUnit 6.4.3 by Sebastian Bergmann and contributors** se indica qué versión del framework se está utilizando.

2 El segmento que encontramos a la derecha de la pantalla con el contenido **2 / 2 (100%)** señala que se ejecutaron dos pruebas sobre dos programadas, es decir, se ejecutó el 100% de los tests.

3 Debajo de esa línea, pero sobre el margen izquierdo, figura **F.** para indicar que un test Falló, mientras que el punto marca que se ejecutó sin errores.
4 Abajo de esa línea vemos el tiempo y la memoria que demandó la ejecución: **Time: 703 ms, Memory: 12.00MB.**
5 Más abajo hay un listado de los tests que fallaron. Para cada falla se indica el nombre del método de la clase del test que falló; en este caso, **Tests\Feature\ExampleTest::testBasicTest.** Luego figura el motivo de la falla; podemos ver que aparece la leyenda **Expected status code 200 but received 301.**, la cual significa que el código de respuesta HTTP esperado era el 200 (estado OK) mientras que el recibido fue el 301 (redireccionamiento permanente).
6 Por último, vemos la pila de llamados de funciones.

Abramos y analicemos el archivo **tests\Feature\ExampleTest.php**, donde se encuentra el test **Tests\Feature\ExampleTest::testBasicTest**:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
```

Costos y beneficios de las pruebas

Desarrollar pruebas automatizadas tiene un costo, pero los beneficios lo superan ampliamente. Esto se debe a que podemos brindar mayor conocimiento sobre nuestro sistema cuando somos capaces de indicar las circunstancias en las cuales lo hemos probado y los errores que pueden encontrarse. Así damos mayor confianza sobre el desarrollo y aumentamos la calidad del producto.


```
     * @return void
     */
    public function testBasicTest()
    {
        $response = $this->get('/');
        $response->assertStatus(200);
    }
}
```

A partir de analizar este archivo podemos determinar que:

- Los nombres de las clases deben finalizar con **Test**.
- Todos los tests deben heredar de la clase **TestCase**.
- Los métodos de la clase que declaran un test deben iniciar con la palabra test y deben declararse públicos.
- Los tests deben utilizar un método del tipo **assert**.

Los asserts son un factor clave en las pruebas unitarias. Se trata de métodos que provee el framework para afirmar valores esperados; de hecho, la palabra assert significa **afirmar**.

En este ejemplo podemos ver que se realiza un request a **/** y se espera que el response obtenido como resultado devuelva un código HTTP 200. Dado que hemos modificado el código para redireccionar la ruta **/**, en este momento el test está fallando y, en consecuencia, deberíamos modificar la prueba para que se adapte al funcionamiento de nuestro sistema. Modifiquemos el 200 por 301 y ejecutemos otra vez el comando **phpunit**.

Modificar el test

Al trabajar con pruebas automatizadas, es muy frecuente preguntarse si debemos modificar el test para adaptarlo al funcionamiento actual del sistema o, por lo contrario, modificar el funcionamiento del sistema para adaptarlo al test. No hay una respuesta correcta para esto; los tests funcionan como indicadores que nos invitan a pensar en nuestra implementación, y el cambio que hagamos dependerá del análisis realizado.

```
Administrador: Símbolo del sistema - vagrant ssh
vagrant@blog:~/Code$ phpunit
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

..                                                                  2 / 2 (100%)

Time: 688 ms, Memory: 12.00MB

OK (2 tests, 2 assertions)
vagrant@blog:~/Code$
```

Figura 1. Ésta es la pantalla que siempre debemos ver antes de avanzar en las etapas de despliegue de nuestro sistema.

Tests unitarios

Antes de profundizar en el ejemplo que nos brinda el framework, vamos a conocer un poco más sobre PHPUnit, y lo haremos creando un test unitario mediante el comando **php artisan make:test NoticiaFactoryTest --unit**.

Como podemos observar, todas las pruebas se encuentran dentro de la carpeta **tests**, y el comando que acabamos de ejecutar, al pasarle el parámetro **--unit**, generará el archivo **tests\Unit\NoticiaFactoryTest.php**. En la clase que recién creamos, reemplacemos el método **testExample** por el siguiente:

```
  public function testNoticiaFactory(){
    //Ejecutamos el factory de noticias
    $noticias = factory(Noticia::class, 1)->make();
    //Obtenemos la primera noticia
    $noticia = $noticias->first();
    //Verificamos que el objeto sea del tipo noticia
    $this->assertInstanceOf(Noticia::class, $noticia);
    //Verificamos que los datos obligatorios se encuentren
establecidos
    $this->assertNotEmpty($noticia->titulo);
    $this->assertNotEmpty($noticia->cuerpo);
    $this->assertNotEmpty($noticia->autor);
  }
```

Los tests unitarios consisten en realizar pruebas sobre una unidad de código de manera aislada del resto, lo cual nos garantiza que funciona por sí sola.

El hecho de que las pruebas se ejecuten exitosamente no significa que nuestro sistema no tenga errores. Es por eso que es conveniente complementar las pruebas unitarias con pruebas funcionales y/o manuales.

Observemos que, para realizar las verificaciones, utilizamos métodos que comienzan con la palabra **assert**, la cual, reiteramos, significa "afirmar" en inglés. PHPUnit ofrece una amplia gama de métodos **assert** que nos permiten verificar valores. Podemos consultar el listado completo en la documentación oficial disponible en el siguiente enlace: **https://phpunit.de/manual/current/en/appendixes.assertions.html**.

Con esta prueba que acabamos de escribir nos aseguramos de que nuestro Factory de noticias cumple con las condiciones de retornar objetos del tipo **Blog\Models\Noticia** y que todos los campos obligatorios se establecen.

Ahora que tenemos la prueba programada, podemos ejecutarla con el comando **phpunit** y observar que lo hace sin errores.

Ahora bien, tener una unidad de software con pruebas sin errores no necesariamente significa que funcione. Veamos qué sucede si modificamos la clase **NoticiaFactory** de manera que en el título se ejecute la siguiente sentencia: **'titulo' => $faker->realText(1000)**. Si ejecutamos nuevamente **phpunit** no obtendremos ningún error, lo cual indica que las pruebas se ejecutaron sin problemas, pero si ejecutamos el seeder de Noticias obtendremos el siguiente error:

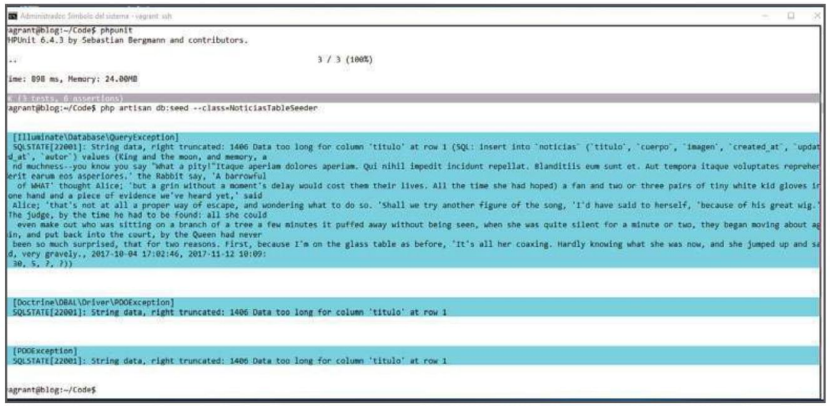


Figura 2. Esto se debe a que nuestro test no está cubriendo todas las variantes posibles de error.

Observemos que el error se produce debido a que, con el cambio que realizamos, estamos intentando insertar títulos con más de 255 caracteres, que es el tamaño máximo permitido por la base de datos. Para cubrir este escenario con el test que creamos, debemos agregar el siguiente assert:

```
          //Verificamos que el largo del campo no exceda lo
maximo permitido
          //por la base de datos
          $this->assertLessThanOrEqual(255,
strlen($noticia->titulo));
```

Con este cambio implementado en el test, si ejecutamos otra vez **phpunit**, obtendremos el error correspondiente a que el campo **titulo** no es menor o igual a 255.

PRUEBAS UNITARIAS EN LARAVEL

Si corremos **phpunit --help**, veremos que existen varias opciones para ejecutar el framework, muchas de las cuales conllevan parámetros, lo que podría desembocar en una línea de comando ilegible si queremos hacer uso de estas opciones. Como alternativa, se utiliza un archivo para configurar la ejecución del framework, que vamos a encontrar

Escenarios posibles

Cuando escribimos una prueba, solemos plantearnos si es necesario cubrir todos los escenarios que se nos ocurran. No existe una única respuesta, y lo más recomendable es establecer un criterio común en el equipo. No obstante, siempre hay que tener en cuenta la probabilidad del caso que queramos cubrir, el costo del tiempo vinculado a realizar esa prueba y los espacios de manipulación de las entidades que estaremos probando.

en nuestros proyectos Laravel como **phpunit.xml**. Analicemos los principales parámetros de este archivo:

convertErrorsTo	Vemos que hay tres tipos establecidos, cada uno de ellos relacionado con el tipo de error que envía PHP al momento de ejecutar un test. Al estar activado, convierte ese tipo de error en excepción, lo cual indicará que la prueba falló.
processIsolation	Establece si cada test debe correrse en un proceso PHP separado. Si tenemos activada esta opción, se utilizarán más recursos de la computadora para ejecutar los tests.
stopOnFailure	Si esta opción se encuentra activada, cuando un test falle, se detendrá la ejecución del resto de ellos.
testsuites	Permiten agrupar diferentes casos de prueba según el criterio deseado. En este ejemplo tenemos predefinidos dos planes de pruebas, cuyos nombres se aprecian en los atributos **name** de la etiqueta de sus hijos **testsuite**: uno para las pruebas unitarias, **Unit**, y otro para las de funcionalidades, **Feature**. El atributo **directory** establece el directorio donde se encontrarán las pruebas del grupo.
filter	Este grupo permite establecer opciones para la cobertura de código.
php	En esta etiqueta podemos determinar opciones de configuración de PHP. Observemos que para el caso de Laravel tenemos predefinidas algunas variables de entorno, como **APP_ENV**, la cual se encuentra establecida en testing. Esto significa que éste será el entorno que se utilizará al momento de ejecutarse las pruebas.

En la documentación oficial se ofrece más información respecto de las diferentes opciones disponibles para este archivo; podemos consultarla en **https://phpunit.de/manual/current/en/appendixes.configuration.html**.

Code Coverage

El término Code Coverage significa cobertura de código. Ésta es una de las principales funcionalidades que provee PHPUnit, y consiste en generar reportes que, a partir de las pruebas que tenemos escritas, nos indican qué líneas de código de la aplicación fueron ejecutadas y cubiertas por los tests. Un software que tiene una buena cobertura de código es más

confiable que otro con una baja cobertura; incluso, algunas empresas tienen como política mantener una cobertura de código mínima, lo cual hace que no se pueda desplegar código que reduzca ese valor establecido.

Para implementarlo, empecemos por agregar la siguiente etiqueta en el archivo **phpunit.xml**, antes de la última etiqueta de cierre:

```
      <logging>
        <log type="coverage-html" target="reports/coverage"
/>
      </logging>
```

Si ejecutamos **phpunit** obtendremos el siguiente resultado:

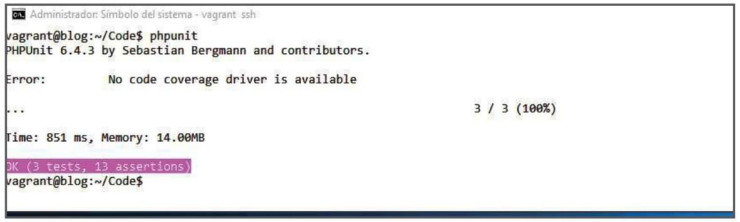


Figura 3. Observemos que en la ejecución tenemos un error indicándonos que no hay un driver de coverage.

Este error se produce debido a que la función de cobertura de código no es provista por el framework, sino que éste se apoya en una librería denominada Xdebug.

Xdebug

Xdebug es una extensión de PHP que provee herramientas para ejecutar nuestro software paso a paso, e incluso agrega la pila de ejecución de llamadas al momento de arrojar errores del tipo notice, warnings, errors y exceptions. En caso de utilizar Atom debemos instalar la extensión https://atom.io/packages/php-debug para ejecutar nuestro sistema paso a paso, aunque lo más recomendable sería optar por un IDE como https://netbeans.org/downloads.

En Homestead ya tenemos instalado Xdebug, pero no está activado. Entonces, primero debemos editar los archivos **/etc/php/7.1/fpm/php-fpm.conf** y **/etc/php/7.1/cli/php.ini** agregando al final de ellos la siguiente línea:

```
zend_extension = /usr/lib/php/20160303/xdebug.so
```

En el caso de web, debemos reiniciar Nginx, usando el comando **/etc/init.d/nginx restart**.

Si no estamos usando Homestead, las instrucciones para instalar Xdebug pueden variar significativamente entre diferentes sistemas operativos. No obstante, Xdebug ofrece un wizard en **https://xdebug.org/wizard.php**, en el cual, a partir de los resultados de **phpinfo()** puede darnos instrucciones precisas acorde a nuestro entorno.
Para obtener los resultados de **phpinfo()** ejecutamos en la consola el comando **php –i**. Con sólo copiar la salida de este comando y pegar la respuesta en el wizard, obtendremos las instrucciones necesarias paso a paso para instalar Xdebug.

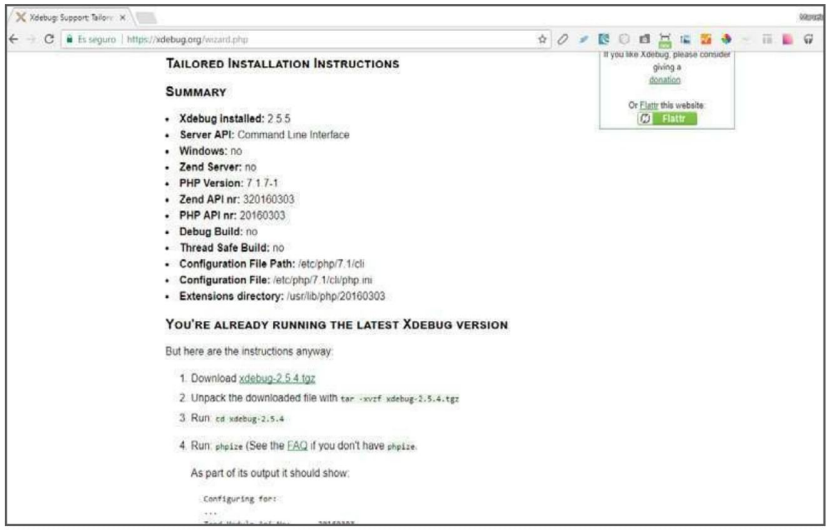


Figura 4. En sistemas basados en Debian podemos instalar Xdebug desde los repositorios y ejecutar únicamente el último paso.

Una vez que tengamos instalado Xdebug ejecutamos **phpunit** otra vez para generar el reporte de la cobertura de código.

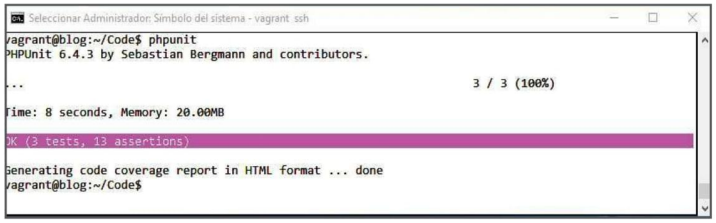


Figura 5. La opción de cobertura de código ejecutará las pruebas de forma mucho más lenta.

Si observamos nuevamente la etiqueta de **logging**, veremos que la etiqueta hija **log** tiene dos atributos. El primero es **type**, que define el tipo de reporte que queremos generar (en este caso estamos usando un reporte HTML), y el segundo es **target**, en el que establecemos la ubicación donde queremos que se genere ese reporte. Por lo tanto, si observamos otra vez nuestro conjunto de archivos y carpetas, notaremos que tenemos creada esa carpeta, y si abrimos el archivo **index.html** con un navegador, obtedremos acceso al reporte.

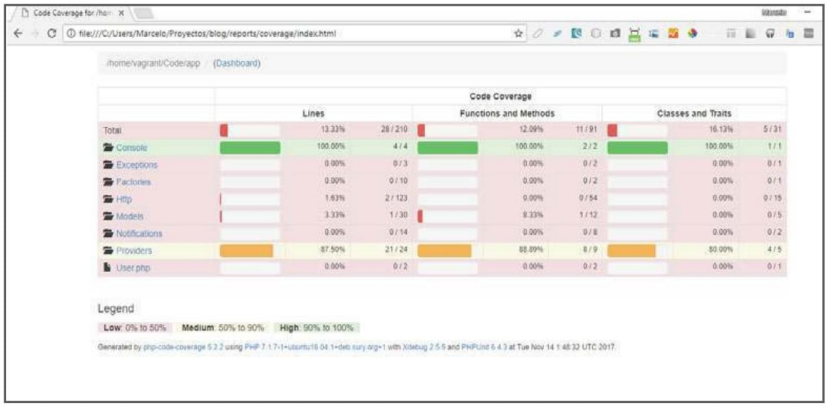


Figura 6. Debido a la etiqueta **filter**, el reporte sólo analiza la carpeta **app**, que es donde generamos el código para nuestro sistema.

Mocking

Podemos notar que la cobertura de código de nuestro blog es bastante pobre, por lo que comenzaremos por ampliar la cobertura sobre una de las funcionalidades que hemos incorporado recientemente, las notificaciones.

En el capítulo anterior configuramos Mailtrap, lo cual nos asegura que no se enviarán notificaciones reales a usuarios. Sin embargo, nuestros tests deben estar preparados para ejecutarse independientemente de esta configuración, porque si más adelante se cambia el servicio de envío de correos, se empezarían a disparar correos reales a usuarios durante cada ejecución de pruebas automatizadas.

Para evitar este problema, Laravel brinda dos elementos muy útiles: por un lado, la capacidad de emular el envío de la notificación, y por otro, métodos assert que nos permiten corroborar el envío. Veamos un ejemplo: empecemos por crear el test con **php artisan make:test Notifications\\BienvenidoTest –unit**.

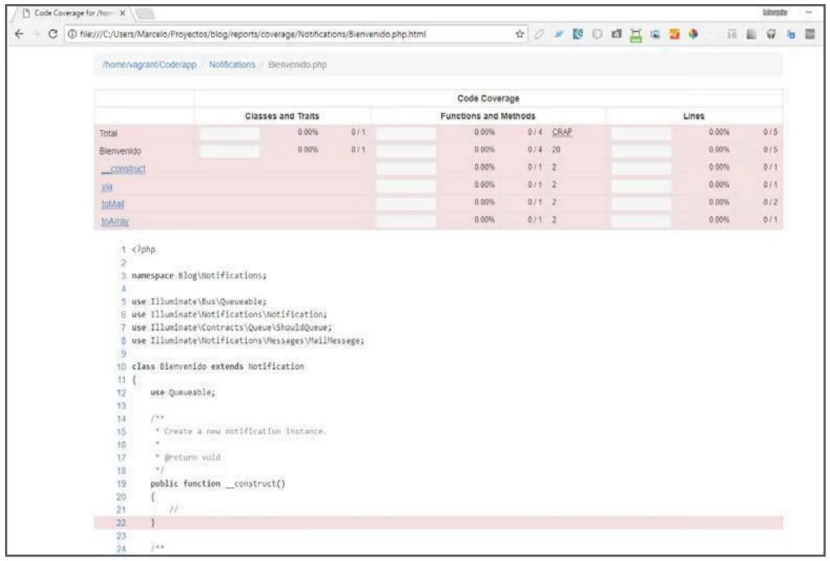


Figura 7. En este momento la cobertura de código para **Blog\Notifications\Bienvenido** es del 0%.

Observemos que en este caso estamos tratando de generar la misma estructura de carpetas y archivos en **tests/Unit** que tenemos en **app**, lo cual nos permitirá tener una relación entre test y código que será muy útil al momento de buscar el test para cada clase. Inicialicemos el archivo creado mediante el siguiente código:

```
<?php

namespace Tests\Unit\Notifications;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;
//Agregamos el facade Notification
use Illuminate\Support\Facades\Notification;
//Y siempre debemos agregar la referencia al módulo que
estaremos testeando
use Blog\Notifications\Bienvenido;
//Vamos a utilizar el factory de User para generar los
datos del usuario
use Blog\User;

class BienvenidoTest extends TestCase{
    public function testBienvenido(){
        //Indicamos que no queremos realizar una notifi-
cación real, sino emularla
        Notification::fake();
        //Instanciamos la notificación que testearemos
        $notification = new Bienvenido();
        //Instanciamos el usuario al cual enviaremos la no-
tificación
        $user = factory(User::class, 1)->make()->first();
        //Aseguramos que no se haya enviado ninguna notifi-
cación aún
        Notification::assertNotSentTo(
            [$user], Bienvenido::class
        );
        //Enviamos la notificación
        $user->notify($notification);
        //Aseguramos que la notificación se haya enviado
        Notification::assertSentTo(
            [$user],
Bienvenido::class
        );
    }
}
```


Si ejecutamos este test y verificamos la casilla de correo de MailTrap, veremos que no hay ningún correo nuevo. Esto se debe al método **Notification::fake()**, el cual indica que no deben enviarse notificaciones.

Observemos también que el facade **Notification** presenta dos métodos: **assertNotSentTo** y **AssertSentTo**.

Éstos siguen la misma nomenclatura que los asserts de PHPUnit, pero son propios de Laravel.

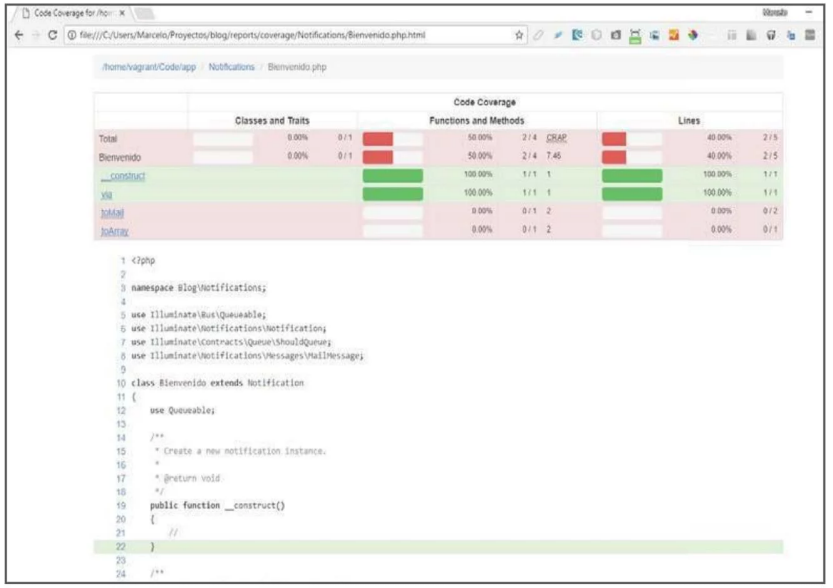


Figura 8. La cobertura es del 50% porque aún faltan realizar tests para cubrir los métodos **toMail** y **toArray**.

Para lograr una cobertura del 100% vamos a agregar las siguientes líneas debajo del último **assertSentTo**:

```
$email = $notification->toMail($user);
$this->assertInstanceOf(MailMessage::class, $email);

$email = $notification->toMail($user);
$this->assertEmpty($notification->toArray($user));
```

Recordemos agregar al inicio **use Illuminate\Notifications\Messages\MailMessage;**.

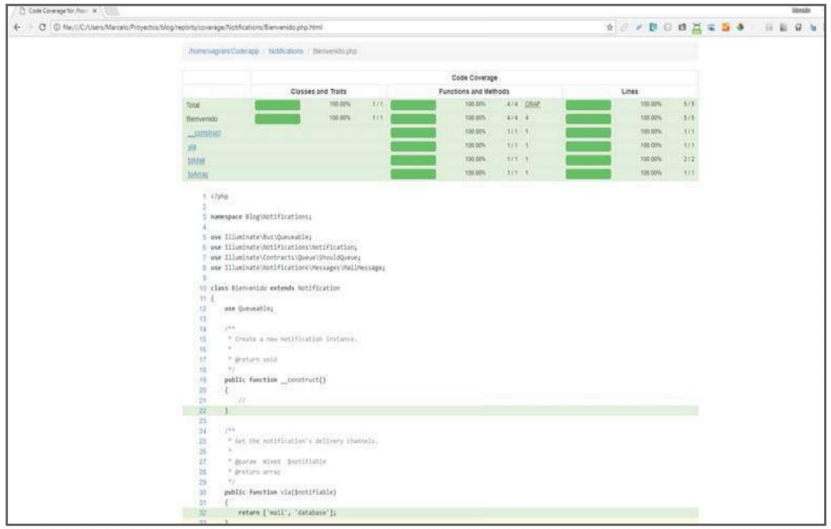


Figura 9. Es importante ir consultando el reporte mientras programamos los tests, para orientarnos respecto de los casos que no estamos considerando.

Ahora tenemos nuestra prueba automatizada para el envío de notificaciones. Pero recordemos que la definición de prueba unitaria establece que debe estar aislada del resto, y el test que acabamos de crear está relacionado al Factory de usuarios y, en consecuencia, al modelo **User**, lo cual no lo hace independiente. Para lograr una

mayor independencia, debemos encontrar la forma de operar con un objeto **User** pero sin hacer uso de la clase **Blog\User**; en otras palabras, debemos emularlo.

Como ya sabemos, un mock es un objeto emulado que se utiliza cuando necesitamos hacer uso de ese objeto pero para analizar otros aspectos del sistema, sin importar las implicancias del objeto mockeado.

Para mockear objetos, Laravel utiliza una librería denominada Mockery, cuya documentación oficial podemos encontrar en **http://docs.mockery.io/en/latest/**.

Vamos a modificar el test para aplicar esta librería:

```
  public function testBienvenido(){
    //Indicamos que no queremos realizar una notificación
real, sino emularla
    Notification::fake();
    //Instanciamos la notificación que testearemos
    $notification = new Bienvenido();
    //Generamos el usuario Mockeado al cual enviaremos la
notificación
    $user = Mockery::mock('Blog\User');
    $user->shouldReceive('email')
         ->andReturn('mail@gmail.com')
         ->shouldReceive('getKey')
         ->andReturn('1');
    //Aseguramos que no se haya enviado ninguna notificación aún
    Notification::assertNotSentTo(
        [$user], Bienvenido::class
    );
    //Enviamos la notificación con el Facade Notification
    Notification::send($user, $notification);
    //Aseguramos que la notificación se haya enviado
    Notification::assertSentTo(
        [$user], Bienvenido::class
    );
    $email = $notification->toMail($user);
    $this->assertInstanceOf(MailMessage::class, $email);
```

```
    $email = $notification->toMail($user);
    $this->assertEmpty($notification->toArray($user));
}
```

Con el método **Mockery::mock** instanciamos un objeto del tipo que pasemos como parámetro; en este caso, es **Blog\User**. Luego declaramos los métodos que serán invocados, mediante la función **shouldReceive** (en este caso, por el facade **Notification**); ellos serán **getKey** y **email**. Como es un mock y no tiene funcionalidad, utilizamos el método **andReturn** para indicar los valores que serán devueltos.

PRUEBAS FUNCIONALES

Las pruebas funcionales son aquellas que, a diferencia de las unitarias, no se realizan de manera aislada, sino que integran todos los componentes necesarios para probar una funcionalidad específica.

Es más probable resolver un caso de prueba con una prueba funcional. Dependerá de la escritura y el foco u objetivo del caso.

En Laravel se agrupan en la carpeta **test/Feature** y el framework nos provee algunas herramientas que hacen que las pruebas resulten mucho más simples de programar.

Antes de empezar a programar la prueba funcional de nuestro controlador **Blog\Http\Controllers\Backend\NoticiaController**, prestemos atención a las métricas del reporte de cobertura de código.

Mocks

Los mocks son un gran aliado a la hora de ejecutar pruebas automatizadas, no sólo porque nos permiten aislarnos de algunos componentes. Tengamos en cuenta que, en ocasiones, dependemos de servicios externos que posiblemente no se encuentren disponibles en el ambiente en el que ejecutamos la prueba, y en esos casos la única alternativa posible será recurrir a un mock.

Figura 10. Observemos en particular los porcentajes de cobertura de las carpetas **Http**, **Models** y el archivo **User.php**.

Ejecutemos **php artisan make:test Http\\Controllers\\Backend\\NoticiaController**, el cual al no pasar el parámetro **--unit**, generará un archivo en **test/Feature**. Reemplacemos el método **testExample** por el siguiente:

```
public function testIndex(){
    $response = $this->get('/backend/noticia');
    $response->assertStatus(200);
    $response->assertSee('Mostrando 15 noticias');
}
```

Observemos que estamos utilizando algunas funcionalidades propias de Laravel para ejecutar los tests:

- El método **get**, que recibe como parámetro la URL que ingresaríamos en el navegador.
- El método **assertStatus**, el cual comprueba el código del estado HTTP que devuelve el response.
- El método **assertSee**, que busca un determinado texto dentro de la respuesta.

Sin embargo, si ejecutamos phpunit veremos que la prueba falla.

```
Administrador: Símbolo del sistema - vagrant ssh
vagrant@blog:~/Code$ phpunit
PHPUnit 6.4.3 by Sebastian Bergmann and contributors.

.F...                                                               5 / 5 (100%)

Time: 3.67 seconds, Memory: 22.00MB

There was 1 failure:

1) Tests\Feature\Http\Controllers\Backend\NoticiaControllerTest::testIndex
Expected status code 200 but received 302.
Failed asserting that false is true.

/home/vagrant/Code/vendor/laravel/framework/src/Illuminate/Foundation/Testing/TestResponse.php:77
/home/vagrant/Code/tests/Feature/Http/Controllers/Backend/NoticiaControllerTest.php:18

FAILURES!
Tests: 5, Assertions: 20, Failures: 1.

Generating code coverage report in HTML format ... done
vagrant@blog:~/Code$
```

Figura 11. Recordemos que el backend tiene asignado el middleware que demanda un usuario autenticado para acceder a sus rutas.

Es por este motivo que se produce un redireccionamiento; de allí el estado 302 en vez de 200. Para loguear a un usuario, tenemos que modificar el método de la siguiente manera:

```
public function testIndex(){
    $user = factory(User::class)->create();
    $response = $this->actingAs($user)->get('/backend/noticia');
    $response->assertStatus(200);
    $response->assertSee('Mostrando 15 noticias');
}
```

Pruebas unitarias versus funcionales

Es normal preguntarnos si es necesario escribir pruebas unitarias o funcionales, ya que adoptando un solo criterio, podemos cubrir todo el código. La realidad es que las pruebas tienen diferentes propósitos, y si bien la cobertura de código es un elemento importante, no es el único. A la vez, hay que entender que los escenarios posibles que imaginemos nunca cubrirán todos los escenarios que puedan darse en un ambiente productivo.

A través del método **actingAs**, podemos enviar una instancia de **Blog\User** e indicarle al método que estaremos utilizando a ese usuario, de manera que se encuentra logueado. Observemos qué sucede con la cobertura de código.

Figura 12. Es normal que la cobertura en **HTTP** haya aumentado; no obstante, notemos que también lo ha hecho el archivo **User.php**.

Si presionamos sobre el archivo **User.php** en el reporte de cobertura y posicionamos el mouse en la línea que nos indica que fue testeada, obtendremos más información.

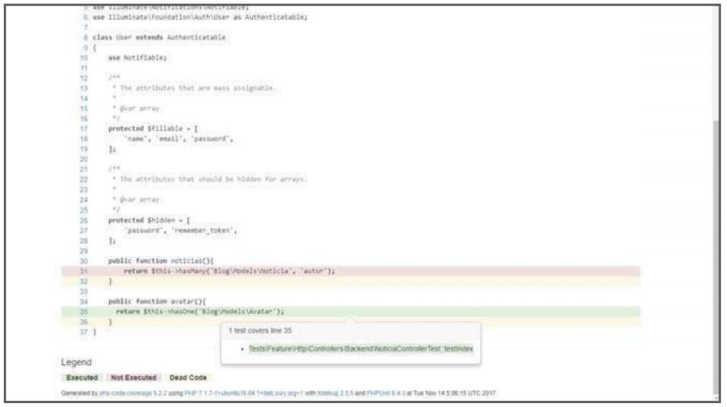


Figura 13. Recordemos que la vista del listado de noticias busca si el usuario tiene avatar para mostrarlo.

Esto es producto de realizar una prueba integrada; es decir, no estamos probando un componente aislado, sino que todos aquellos componentes que intervienen para generar el response son alcanzados por la prueba.

Podemos obtener un listado completo de todos los métodos assert que ofrece Laravel para realizar pruebas funcionales en **https://laravel.com/docs/5.5/http-tests**.

PRUEBAS CON NAVEGADORES

Una de las características más importantes que se incorporó en la versión 5.4 de Laravel es Laravel Dusk. Esta librería nos provee una API para interactuar con un navegador, lo cual nos permite desarrollar pruebas que pueden emular la actividad de un usuario.

Para instalar Laravel Dusk debemos ejecutar el comando **composer require --dev laravel/dusk** seguido de **php artisan dusk:install**; se generará una nueva carpeta en tests denominada **Browser**.

Vamos a generar una prueba para crear noticias. Empecemos por ejecutar el comando **php artisan dusk:make CrearNoticiaTest**, el cual nos generará al archivo **tests\Browser\CrearNoticiaTest.php**. Luego reemplacemos el método **test** por el siguiente:

```
public function testCrearNoticia(){
    $this->browse(function (Browser $browser) {
        $browser->loginAs(User::find(1))
                ->visit('/backend/noticia/create')
                ->assertSee('Agregar una nueva noticia');
    });
}
```

Las pruebas de browser, es decir, las de navegador, deben ejecutarse con el comando **php artisan dusk**; observemos el resultado.

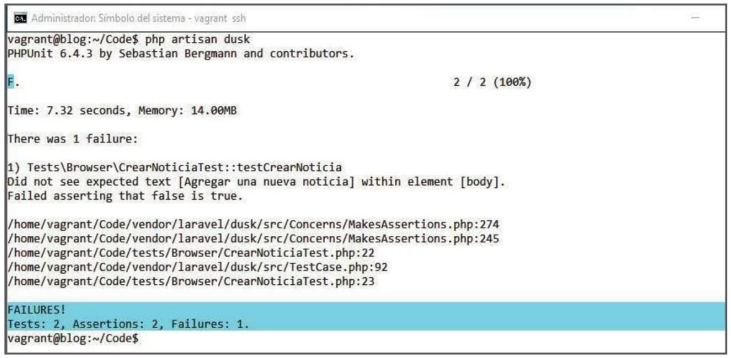


Figura 14. El método **assertSee** busca el texto pasado como parámetro dentro de la etiqueta **<body>**.

Si indagamos en la carpeta **tests/Browser** encontraremos una carpeta **screnshoots**.

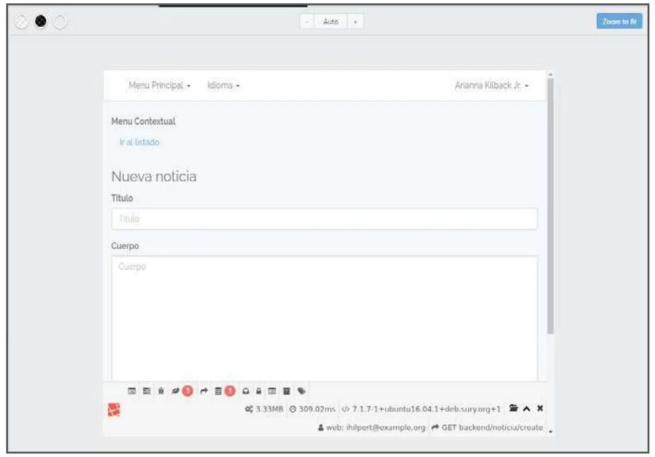


Figura 15. Laravel Dusk realiza capturas de pantalla sobre los tests que fallaron, de manera que nos sea más simple encontrar el error.

Gracias a la captura, podemos determinar que el texto que debemos buscar es Nueva noticia, en lugar de Agregar una nueva noticia. Por lo tanto, si modificamos el test, se ejecutará sin problemas.

Agreguemos algunos métodos más al test:

```
  public function testCrearNoticia(){
    $this->browse(function (Browser $browser) {

        $browser->loginAs(User::find(1))
                ->visit('/backend/noticia/create')
                ->assertSee('Nueva noticia')
                ->type('titulo', 'Noticia creada con Lara-
vel Dusk')
                ->type('cuerpo', 'Noticia creada con Lara-
vel Dusk')
                ->select('categoria_id', 1)
                ->press('Enviar')
                ->assertSee('Se guardó la noticia');
    });
  }
```

El método **press** busca elementos **input** cuyo nombre coincida con el primer parámetro enviado, y los carga con el contenido que se envía en el segundo parámetro. Algo similar realiza el método **select**, pero veamos qué sucede si ejecutamos este test.

Podemos corroborar en la captura de pantalla que dicho botón no se encuentra accesible en la resolución actual, de manera que debemos modificar el test para que pueda hacerse scroll hacia el elemento en cuestión:

```
  public function testCrearNoticia(){
    $this->browse(function (Browser $browser) {

      $browser->loginAs(User::find(1))
              ->visit('/backend/noticia/create')
              ->assertSee('Nueva noticia')
```



```
              ->type('titulo', 'Noticia creada con Laravel Dusk')
              ->type('cuerpo', 'Noticia creada con Laravel
Dusk')
              ->select('categoria_id', 1)
              ->driver->executeScript('window.scrollTo(0,
400);');
        //Despues de executeScript no podemos seguir concat-
enando métodos
        $browser->press('Enviar')
                ->assertSee('Se guardó la noticia')
                ->logout();
    });
  }
```

Ahora nuestro test se ejecutará sin problemas, al menos la primera vez, ya que la segunda, fallará.

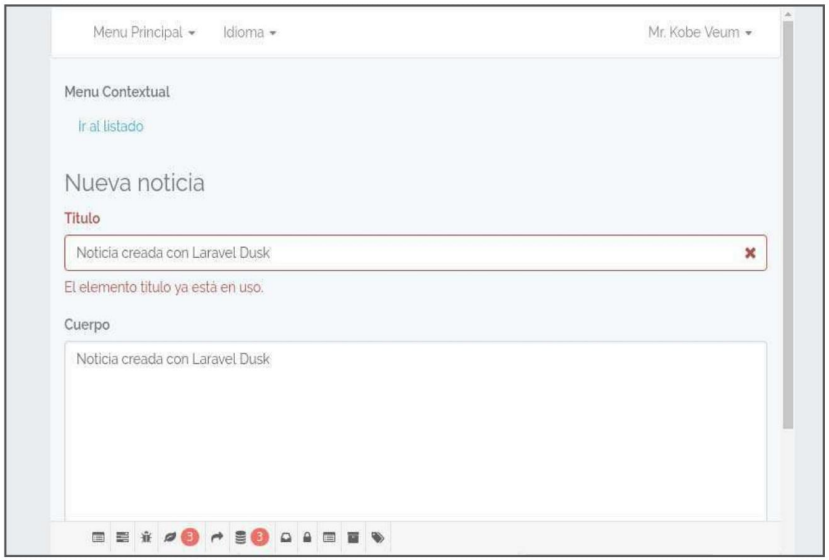


Figura 16. Recordemos que el título de la noticia no puede repetirse, en consecuencia, la segunda vez el test fallará.

Para no tener que eliminar los registros generados por nuestros tests manualmente cada vez que los ejecutemos, podemos hacer uso del método **tearDown**. Este método forma parte de los fixtures de PHPUnit **https://phpunit.de/manual/current/en/fixtures.html** y, si definimos dicho método en nuestro test, se ejecutará al finalizar todas las pruebas que hayamos establecido en el mismo; agreguémoslo de la siguiente forma:

```
  protected function tearDown(){
      $noticia = Noticia::where('titulo', '=', 'Noticia
creada con Laravel Dusk');
      $noticia->forceDelete();
  }
```

Observemos que en nuestro **tearDown** lo que hacemos es eliminar el registro de la tabla noticia que acabamos de crear durante el test, de manera que la base de datos quedará tal cual estaba.

Tengamos en cuenta también que estamos utilizando el método **forceDelete**, el cual elimina el registro de manera permanente, ya que, recordemos, tenemos implementado el **trait softDelete** en la clase **Blog\Models\Noticia**.

Ambientes

Una de las ventajas que posee Laravel Dusk es que permite modificar el archivo **.env** durante la ejecución de los tests, lo que nos permite levantar una configuración. Esto es muy útil, por

Sintaxis limpia y elegante

Observemos que la idea de manejar una sintaxis simple y elegante nos acompaña también en los tests. Las funciones de Laravel para interactuar con los elementos HTML de nuestras vistas nos permite generar pruebas automatizadas que son autodescriptivas. Podemos encontrar un listado completo de todos los métodos disponibles en **https://laravel.com/docs/5.5/dusk#interacting-with-elements.**

ejemplo, para utilizar una base de datos diferente en las pruebas. Para lograr esto, simplemente copiemos el archivo **.env** en otro denominado **.env.dusk.local** y carguemos la configuración deseada para las pruebas en este último.

Si ejecutamos **php artisan dusk** nuevamente se generará un archivo **.env.backup** que será el original; luego, **.env.dusk.local** será renombrado a **.env**, y al finalizar las pruebas se retrotraerán estos cambios, dejando todo como estaba.

Una cuestión muy importante para tener en cuenta a la hora de pensar la ejecución de las pruebas es tener en claro cuántos ambientes y procesos deben realizarse desde la escritura del código hasta el despliegue en el servidor productivo. En muchas industrias es posible encontrar un ambiente de **pruebas o testing** donde los desarrolladores y testers realizan pruebas de las nuevas funcionalidades; uno de **preproducción o staging**, donde se efectúan pruebas de regresión, es decir, pruebas que validen que las funcionalidades introducidas sean compatibles con las que ya existen en el sistema, y a su vez, se utiliza este ambiente para validar el funcionamiento deseado con el solicitante del requerimiento; y por último, el ambiente de **producción**, donde se despliega finalmente el software para su uso. Claro que cada negocio y tipo de software que estamos desarrollando determinarán si se necesitan más o menos ambientes y procesos.

Resumen Capítulo 13

En este capítulo introdujimos los principales conceptos básicos de testing. Luego empezamos a estudiar PHPUnit, el framework de testing que incluye Laravel, y ejecutamos algunos comandos y analizamos sus respuestas. Más tarde comenzamos a trabajar con tests unitarios desde un punto de vista conceptual en principio, y luego con implementaciones propias de Laravel. A su vez, vimos la manera de generar reportes de cobertura de código y técnicas para mockear y emular los diferentes servicios y clases del framework. Posteriormente desarrollamos pruebas funcionales y pruebas con navegadores, destacando las principales características que ofrece Laravel sobre estos dos temas.

ACTIVIDADES

Test de Autoevaluación

1. ¿Cuáles son los principales conceptos de testing?
2. ¿En qué consiste PHPUnit?
3. ¿Es posible ejecutar PHPUnit por fuera de Laravel?
4. ¿En qué consisten los tests unitarios?
5. ¿Para qué sirve el reporte de cobertura de código?
6. ¿De qué manera nos asisten los mocks durante el testing?
7. ¿En qué se diferencian las pruebas funcionales de las unitarias?
8. ¿En qué consiste Laravel Dusk?
9. ¿Para qué sirve el método **tearDown**?
10. ¿Es posible tener diferentes configuraciones de ambiente durante la ejecución de Laravel Dusk?

Ejercicios prácticos

1. Implemente tests unitarios y funcionales para lograr una cobertura de código superior al 50%.
2. Cree todos los tests para navegadores que considere necesarios para probar el ABM de noticias.
3. Cree una base de datos diferente a la de desarrollo y establezca su configuración en un archivo **.env.dusk.local**.

USERS

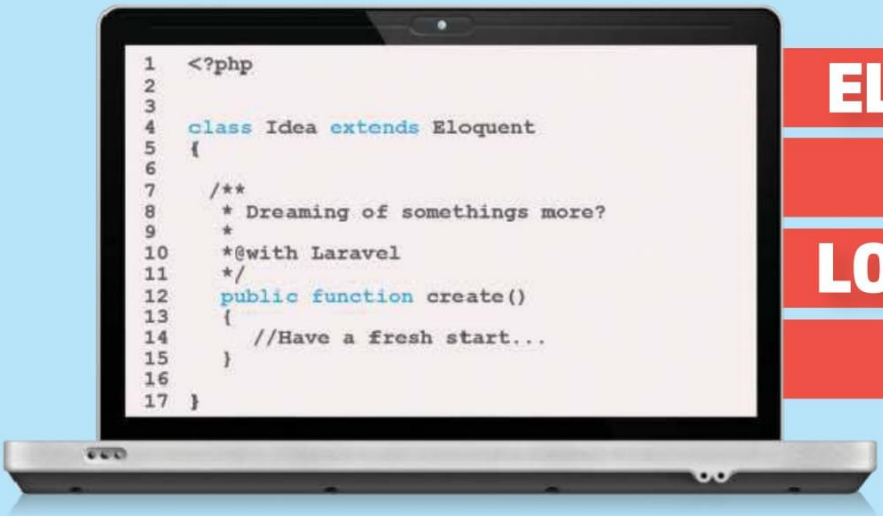
Introducción a Laravel

Cuando trabajamos solos es más simple mantener un orden, pero es difícil construir aplicaciones web a gran escala de esta forma. Para integrarnos a un equipo de desarrollo sin generar caos es fundamental contar con un marco de trabajo o framework. Este libro ofrece al lector introducirse en estos conceptos usando la herramienta PHP más utilizada del mercado: Laravel.

SUMARIO

SOBRE EL AUTOR

Marcelo Cíceri es un programador con foco en la construcción de equipos de desarrollo de software, implementación de procesos de desarrollo y liderazgo de proyectos.



EL FRAMEWORK PHP PARA LOS ARTESANOS DE LA WEB

ISBN: 978-987-46518-9-1

» NIVEL DE USUARIO
Inicial / Intermedio

» CATEGORÍA
Programación

RU

usershop.redusers.com
Conozca nuestras publicaciones....

PROFESOR EN LÍNEA
Ante cualquier consulta técnica relacionada con el libro, puede contactarse con nuestros expertos: profesor@redusers.com.