El *framework* Laravel

PID_00289849

Samuel González Rodríguez

Tiempo mínimo de dedicación recomendado: 5 horas

Universitat Oberta de Catalunya

Samuel González Rodríguez

Ingeniero Superior de Telecomunicación por la Universidad Politécnica de Cataluña (UPC). Profesor titular de ciclos formativos de la familia profesional Informática y Comunicaciones, tarea que compagina como profesor colaborador en la Universitat Oberta de Catalunya (UOC) en el máster universitario de Desarrollo de Sitios y Aplicaciones Web.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Javier Luis Cánovas Izquierdo

Primera edición: septiembre 2022

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Samuel González Rodríguez
Producción: FUOC

Índice

Introducción

El presente material introduce al *framework* Laravel de una forma práctica. No se trata de un manual de uso exhaustivo ni tampoco de una guía de referencia técnica. El objetivo que persigue es proporcionar los aspectos más importantes de este *framework* y servir como guía de aprendizaje, ofreciendo otros recursos vinculados y elegidos expresamente que ya existen en Internet para poder ampliar conocimientos.

El documento se estructura principalmente en cuatro bloques:

1) Introducción a Laravel

2) Instalación y estructura de directorios y archivos de Laravel

3) Conceptos básicos: rutas, controladores, vistas, bases de datos, migraciones y modelos

4) Autenticación, API, extensiones y ecosistema de Laravel

Durante la exposición de los conceptos básicos se recomienda practicar con los ejemplos propuestos. Esto es especialmente importante, ya que la mejor manera de aprender Laravel (y cualquier *framework*) consiste en dedicarle el tiempo suficiente como para comprender su funcionamiento.

Nota

Los ejemplos de este material están inspirados principalmente en la documentación oficial de Laravel 9: https://laravel.com/docs/9.x

Este módulo está basado en la versión 9 de Laravel, que es la última versión de Laravel en el momento de la redacción de este documento.

16/1/26 Laravel 12 https://laravel.com/docs/12.x/releases

Laravel, creado por Taylor Otwell en 2011, es un *framework* basado en el lenguaje de programación PHP. De esta forma, para poder sacar el máximo provecho, es necesario tener unos conocimientos mínimos de este lenguaje de programación.

Figura 1. Logotipo de Laravel 9
Fuente: <https://laravelnews.imgix.net/images/laravel9.png?ixlib=php-3.3.1>

Lecturas y enlaces recomendados

Formación para adquirir conocimientos básicos sobre PHP:

Jeffrey Way. «The PHP Practitioner». <https://laracasts.com/series/php-for-beginners>

Publicaciones de referencia sobre PHP:

Daniel Julià (septiembre 2022). *Introducción a PHP* (primera edición). Barcelona: UOC.

Matt Zandstra (abril 2021). *PHP 8 Objects, Patterns, and Practice: Mastering OO Enhancements, Design Patterns, and Essential Development Tools* (6.ª edición). Apress. <https://learning.oreilly.com/library/view/ php-8-objects/9781484267912/>

Amplia biblioteca con documentación, guías y tutoriales sobre Laravel en formato video: https://laracasts.com/

Formación sobre Laravel 8, aunque la mayoría de funcionalidad tratada es compatible con la versión 9:

Jeffrey Way. «Laravel 8 From Scratch». <https://laracasts.com/series/laravel-8-from-scratch>

Diferencias entre las versiones 9 y 8 de Laravel:

Jeffrey Way. «What's New in Laravel 9». <https://laracasts.com/series/whats-new-in-laravel-9>

Publicación de referencia sobre Laravel:

Matt Stauffer (2019). *Laravel: Up & Running, 2nd Edition*. O'Reilly Media. <https://learning.oreilly.com/library/view/laravel-up/9781492041207/>

El lema de Laravel es «el *framework* PHP para artesanos web» y tiene como objetivo ser un *framework* de una sintaxis elegante y simple, evitando el «código espagueti».[1] Tiene gran influencia de reconocidos *frameworks* como Ruby on Rails y Symfony, y entre sus principales características destaca que:

(1) ¿Qué es el código espagueti?: https://es.wikipedia.org/wiki/C%C3%B3digo_espagueti

- Es un *framework* escrito en PHP.
- Emplea el paradigma de programación orientada a objetos.
- Es de código abierto (licencia MIT).
- Cuenta con una amplia comunidad de desarrolladores.
- Cuenta con una curva de aprendizaje que, comparada con *frameworks* similares como Symfony, es relativamente asequible.
- Ofrece una buena escalabilidad, que lo hace ideal para proyectos complejos.
- Emplea la arquitectura modelo-vista-controlador, que facilita:
  - el trabajo en equipo
  - la claridad del proyecto
  - el mantenimiento
  - la reutilización de código

Además, incorpora:

- **Artisan**, que es una interfaz de línea de comandos que permite realizar múltiples tareas durante el proceso de desarrollo o despliegue a producción.
- **Eloquent**, que es un ORM que permite manejar de una forma fácil y sencilla la comunicación con las bases de datos.

ORM (*Object-Relational Mapping*)

Un ORM es una herramienta que facilita el manejo de los registros de la base de datos. Esto se consigue representando los datos como objetos y trabajando como capa de abstracción sobre el motor de la base de datos.

- **Sistema de plantillas Blade**, que es un gestor de vistas simple y a su vez potente.

- Control de versiones de la base de datos, gracias al **sistema de migraciones**.

También ofrece características avanzadas para desarrolladores sénior, como son la inyección de dependencias, las pruebas unitarias, las colas y los eventos en tiempo real, entre otras funcionalidades.

Por último, cabe destacar que la cuota de mercado de Laravel es relativamente considerable.[2]

(2)Ved las estadísticas de Laravel en https://trends.builtwith.com/framework/Laravel

Sitios web que emplean Laravel

El propio Laracasts, el portal de *e-learning* MyRank o el servicio CheckPeople.

1. Instalación

Laravel 9 se puede instalar en una máquina local siguiendo varios procedimientos. El más sencillo consiste en utilizar la herramienta Composer, que es un gestor de dependencias para proyectos implementados en PHP. Está inspirado en npm (gestor de paquetes para JS) y en Bundler (Ruby). Un gestor de dependencias se encarga de evaluar la compatibilidad, de descargar, instalar, actualizar y desinstalar paquetes de software dentro de un proyecto. De este modo, Composer permite instalar librerías o paquetes de terceros de una forma ordenada, ágil y que garantiza el mantenimiento del proyecto.

Enlace recomendado

Instalación de Composer: https://getcomposer.org/download/

Figura 2. Logo de Composer
Fuente: <https://getcomposer.org/>

Para instalar la última versión de Composer en el momento de redactar este documento, es necesario tener instalado previamente la versión 8 de PHP.

Guías de instalación de PHP v8

En Windows: https://php.tutorials24x7.com/blog/how-to-install-php-8-on-windows

En Linux: https://ubunlog.com/php-8-0-instalar-lenguaje-en-ubuntu/

En MacOS: https://postsrc.com/posts/how-to-install-php-8-on-macos-big-sur-using-homebrew

Una vez instalado Composer, se recomienda verificar que el directorio `bin` de Composer se ha añadido en el PATH del sistema operativo local. Si no fuese el caso, se puede añadir de la siguiente manera:

En macOS: $HOME/.composer/vendor/bin

En Windows: %USERPROFILE%\AppData\Roaming\Composer\vendor\bin0

En GNU/Linux: $HOME/.config/composer/vendor/bin, o bien $HOME/.composer/vendor/bin

Para comenzar a utilizar Composer en un proyecto web cualquiera es necesario crear previamente un archivo llamado `composer.json` en la raíz del proyecto. En el caso de Laravel, una vez instalado, el archivo `composer.json` ya se encuentra por defecto en su raíz. En este archivo se podrá consultar y modificar la versión de Laravel instalada, la versión de PHP o las librerías instaladas, entre otra información.

Enlaces recomendados

Cómo funciona Composer:

Miguel Ángel Álvarez (2020). «Composer, gestor de dependencias para PHP». *Desarrolloweb.com*. <https://desarrolloweb.com/articulos/composer-gestor-dependencias-para-php.html>

Bruno Lorente (2021). «¿Qué es Composer y qué puede hacer por ti?». *Novadevs*. <https://novadevs.com/publicaciones/que-es-composer-y-que-puede-hacer-por-ti/>

Laravel también se puede instalar mediante la herramienta Sail, que requiere de la plataforma Docker. Se recomienda para usuarios con conocimientos más avanzados en sistemas.

Guías de instalación de Sail y Docker

Sail: https://laravel.com/docs/9.x/sail

Docker: https://www.docker.com/

1.1. Instalación de Laravel con Composer

Para crear un proyecto nuevo en Laravel con Composer simplemente basta con escribir la siguiente instrucción por línea de comandos:

```
composer create-project laravel/laravel proyecto-prueba-uoc
```

Esta instrucción crea un directorio llamado **`proyecto-prueba-uoc`** y allí se descarga la estructura del *framework* Laravel.

Nota

Para conocer la versión instalada de Laravel: `php artisan --version`

Recuerda: Artisan es una interfaz de línea de comandos que permite desarrollar múltiples tareas que facilitan el desarrollo y despliegue a producción.

Ved también

Artisan se desarrolla más adelante, en el apartado 5 de este módulo.

Una vez descargado el *framework*, entraremos en la carpeta correspondiente con:

```
cd proyecto-prueba-uoc
```

y se podrá probar simplemente iniciando el servidor web local que integra Laravel, con este comando:

```
php artisan serve
```

Este comando inicia el servidor web local en el puerto 8000 (por defecto). Si este puerto estuviese ocupado, lo haría en el 8001 y así sucesivamente.

¡Recordatorio!

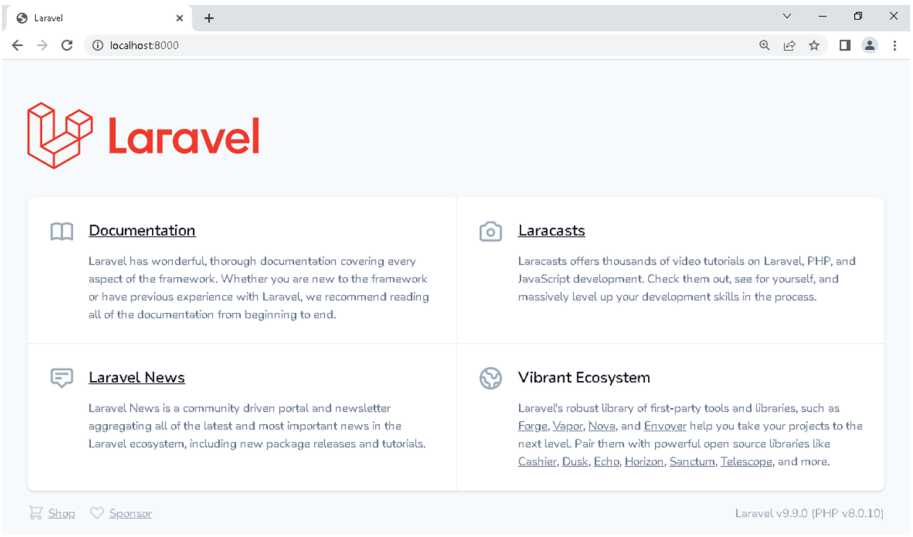
En Linux, si detenemos el servidor web con Ctrl+Z, el puerto quedará ocupado (proceso no finalizado). Para detenerlo correctamente se deberá emplear Ctrl+C, igual que en Windows.

Cuando se haya iniciado el servidor web local, se podrá acceder a la aplicación por medio de un navegador web en la URL `http://localhost:8000`

Alternativa

Laravel 9 también puede instalarse como una dependencia global de Composer: https://laravel.com/docs/9.x/installation#the-laravel-installer

Figura 3. Página de inicio de Laravel v9.9



Fuente: elaboración propia.

1.2. Estructura de directorios y archivos de un proyecto Laravel

El *framework* Laravel se compone de múltiples directorios y ficheros. A continuación, se describen los más significativos:

Tabla 1. Directorios y ficheros respecto a la raíz del proyecto

`.env`	Muchas de las opciones de configuración de Laravel varían según el entorno en el que se ejecutan (máquina local o servidor). En este fichero se almacena la configuración principal, como son el nombre de la base de datos, el usuario y la contraseña de acceso, la IP de la máquina o el sistema gestor de bases de datos empleado, entre otros. Más información en: <https://laravel.com/docs/9.x/configuration#environment-configuration> ¡Atención! En caso de subirse el proyecto Laravel a Git, por motivos de seguridad habría que asegurarse de que este fichero se encuentra en el `.gitignore`
`composer.json`	Contiene la información para instalar Laravel mediante el gestor de paquetes Composer. Por defecto se especifica la instalación de un solo paquete, el propio *framework* de Laravel, pero también se puede especificar la instalación de otras librerías o paquetes externos que añaden funcionalidad a Laravel.
`app/`	Contiene el código principal de la aplicación: controladores (**`/Http/Controllers`**), modelos (**`app/Models`**), *middlewares* (filtros), etc.
`bootstrap/`	Contiene el archivo `app.php`, encargado de poner en marcha el *framework*. Contiene un directorio llamado *caché* para optimizar el funcionamiento de Laravel (archivos caché de rutas y servicios). ¡Atención! No confundir con la librería de *front-end* desarrollada por Twitter que lleva el mismo nombre.

Ampliación en <https://laravel.com/docs/9.x/structure>

config/	Archivos de configuración (base de datos, caché, mail, sesiones, etc.). Por defecto, Laravel emplea los valores definidos en el archivo .env. En caso de que no encuentre algún parámetro, emplea los valores definidos en el fichero **/database.php**. Ejemplo: host => *env('DB_HOST' , localhost)* Laravel buscará el valor de *DB_HOST* en el archivo .env. En caso de no encontrarlo, emplearía el valor por defecto, que es el que se especifica en la segunda posición (*localhost*, en este caso).
database/	Definición de la base de datos, migraciones (**/migrations**) y modelo *factory* (**/factories**).
lang/	Almacena los ficheros relacionados con el idioma del proyecto.
public/	El único directorio que deberá ser visible en el servidor web; contiene el fichero `index.php` (donde se inicia todo el proceso de ejecución del *framework*); contiene también los archivos CSS, JS, imágenes y todos aquellos que quieren hacerse públicos.
resources/	Recursos de la aplicación, como las vistas (**/views**), el soporte de los diferentes idiomas (**/lang**) y *assets* de preprocesadores CSS (less, sass).
routes/	Archivos de definición de rutas (**web.php**) y apis (**api.php**). ¡Atención! En versiones antiguas de Laravel el fichero de rutas se encontraba en `app/Http/routes.php`.
storage/	*Logs* del sistema (**/logs**) y carpetas en las que se guarda toda la información interna necesaria para ejecutar la aplicación web (archivos de sesión y compilación de vistas, entre otros).
tests/	Se emplea para guardar los archivos con las pruebas automatizadas (Laravel facilita el uso de PHP Unit).
vendor/	Se alojan todas las librerías y dependencias del *framework*. La herramienta Composer se encarga de gestionar su contenido. A modo de ejemplo, por defecto se incluyen las librerías Symfony, Doctrine (ORM de PHP), PHPUnit y el mismo Composer, entre otros.

Ampliación en <https://laravel.com/docs/9.x/structure>

2. Enrutamiento y controladores

La función principal de un *framework* web es recibir peticiones de un usuario (cliente) y entregar respuestas, generalmente con el protocolo HTTP (o HTTPS). Esto se lleva a cabo mediante una acción denominada *enrutamiento*, que ha de ser definido en primer lugar al trabajar con Laravel.

De forma general, las rutas se vinculan a controladores, los cuales sirven de enlace entre los datos y la respuesta al cliente. Esta forma de gestionar peticiones y respuestas corresponde al patrón modelo-vista-controlador.

2.1. Modelo-vista-controlador

Laravel se basa en la arquitectura de software modelo-vista-controlador (MVC), que se fundamenta en la separación del código en las tres capas siguientes:

Tabla 2. Capas de código

Modelo	Representa una tabla de una base de datos (o un registro de una tabla). Por ejemplo, una tabla «noticia», o el campo «autor» de esta.
Vista	Se ocupa de la «lógica de presentación», esto es, representa la plantilla que envía datos al navegador del usuario. Por ejemplo, una plantilla de una página de inicio de sesión de usuario (*login*), creada con HTML, CSS y JavaScript.
Controlador	Se encarga de la «lógica de negocio», es decir, de gestionar las solicitudes HTTP recibidas, obtener los datos de la base de datos (modelo) y enviar una respuesta al usuario (vista).

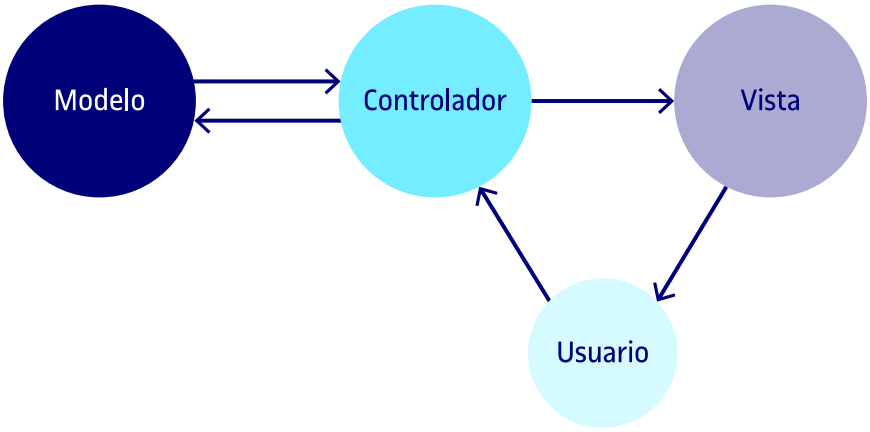
En la siguiente figura se representa un esquema en el que:

1) En primer lugar, el usuario escribe en el navegador una URL que se traduce en una solicitud HTTP al servidor web remoto.

2) En segundo lugar, el controlador, en respuesta a esa solicitud, puede escribir datos o extraerlos del modelo (base de datos).

3) Finalmente, lo habitual es que el controlador envíe datos a una vista, que se devolverá al usuario final para que la muestre en su navegador.

Figura 4. Modelo-vista-controlador



Fuente: Stauffer, Matt (2019). *Laravel: Up & Running, 2nd Edition*. Capítulo 3. O'Reilly Media.

Enlaces recomendados

Matt Stauffer (2019). *Laravel: Up & Running, 2nd Edition*. Capítulo 3. O'Reilly Media. <https://learning.oreilly.com/library/view/ laravel-up/9781492041207/>

Miguel Ángel Álvarez (2020). «Qué es MVC». *Desarrolloweb.com*. <https://desarrolloweb.com/articulos/que-es-mvc.html>

2.2. Enrutamiento

En aplicaciones web, existen dos tipos de rutas: rutas tipo web y rutas tipo API.

Las rutas web son aquellas que visitan los usuarios finales. En Laravel, este tipo de rutas se definen en el fichero `routes/web.php` y cuentan con herramientas como protección CSRF y sesiones (proporcionadas por el *middleware* web).

Sesión

En aplicaciones web, una sesión es un mecanismo que permite conservar información sobre un usuario (estado) entre peticiones HTTP (protocolo sin estado) a un mismo sitio web.

CSRF

El CSRF (*cross-site request forgery*) es una vulnerabilidad que consiste en enviar solicitudes no deseadas por un usuario mediante el mismo usuario.

Middleware

Se denomina *middleware* generalmente a aquellos artefactos software que ofrecen servicios y funciones comunes a otras aplicaciones.

Ved también

Los *middlewares* se desarrollan más adelante, en el subapartado 2.2.6 de este módulo.

Por otra parte, las rutas API se utilizan para gestionar las peticiones de tipo API REST. En Laravel este tipo de rutas se definen en el fichero `routes/api.php` y, a diferencia de las de tipo web, no tienen estado (sesiones). Además, pertenecen al *middleware* API, en lugar de al *middleware* web.

En este apartado nos centraremos en las rutas web.

Ved también

Las rutas de tipo API las desarrollaremos más adelante, en el apartado 7 de este módulo.

2.2.1. Los verbos HTTP

Enlace recomendado

Definición de métodos HTTP en la sección 9 del RFC 2616: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

El enrutamiento permite responder a cualquier petición HTTP. Estas peticiones también se conocen como verbos HTTP:

Tabla 3. Verbos HTTP

Verbo HTTP	Descripción
GET	Solicita un recurso (o una lista de recursos). Las peticiones de tipo GET únicamente han de recuperar datos, sin causar ningún tipo de efecto secundario en el servidor (por ejemplo, no producir nuevos registros ni modificar los existentes). Esta propiedad se conoce con el nombre de *idempotencia* (una acción ejecutada un número indefinido de veces produce siempre el mismo resultado).
HEAD	Solicita una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
POST	Se utiliza para enviar una entidad a un recurso en concreto, causando a menudo un cambio en el estado o efectos secundarios en el servidor (crear nuevos recursos).
PUT	Reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
PATCH	Se emplea para aplicar modificaciones parciales a un recurso.
DELETE	Borra un recurso en específico.
OPTIONS	Se utiliza para describir las opciones de comunicación para el recurso de destino.

Los verbos más empleados en desarrollo web son GET y POST, seguidos de PUT y DELETE.

2.2.2. Enrutamiento básico

La forma más sencilla de definir una ruta en Laravel es hacer coincidir una ruta, por ejemplo **`/greeting`**, con una función anónima o clausura:

```
// Fichero routes/web.php

use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
   return 'Hello, world!';
});
```

El comando `use` se emplea para importar clases. En el ejemplo, se importa la clase cuyo *namespace* (directorio para clases) es `Illuminate\Support\Facades\Route`.

Illuminate

Es el nombre elegido por Laravel para guardar todas las utilidades que proporciona. Para Laravel 9, se pueden consultar en: https://laravel.com/api/9.x/Illuminate.html

Facade

Facade (o fachada) es un tipo de patrón de diseño empleado en la programación orientada a objetos. Consiste en dividir un entorno complejo en subsistemas, con el objetivo de reducir su complejidad.

En Laravel se emplean múltiples *facades*. En el fichero config/app.php (sección *aliases*) se pueden consultar todas las *facades*.

Más información: https://laravel.com/docs/9.x/facades

En PHP, una clausura o *closure* es una función anónima (función sin nombre) que captura el ámbito actual y proporciona acceso a ese ámbito cuando se invoca la función.

En el ejemplo, si se visita la ruta `/greeting` se ejecutará la función anónima (definida en la misma ruta) y se devolverá la frase "Hello, world!".

Nótese que el contenido se devuelve con un `return` en lugar de mostrarlo directamente con un `echo` o `print()`. Utilizar `return` tiene una ventaja: antes de mostrarlo al usuario, se puede tratar el contenido devuelto en la subcapa *middleware*.

Nótese también que la ruta se ha definido con `Route::get()`. Dicho de otro modo, Laravel ejecutará la función anónima cuando la petición HTTP sea de tipo GET. Para otros verbos HTTP también se podrán definir rutas:

```
Route::post('/greeting', function () {
   // Tratar la respuesta en caso de que alguien envíe una petición POST a esta ruta
});


Route::put('/greeting', function () {
   // Tratar la respuesta en caso de que alguien envíe una petición PUT a esta ruta
});


Route::patch('/greeting', function () {
   // Tratar la respuesta en caso de que alguien envíe una petición PATCH a esta ruta
});


Route::delete('/greeting', function () {
   // Tratar la respuesta en caso de que alguien envíe una petición DELETE a esta ruta
});
```

También se puede dar una respuesta común a múltiples verbos HTTP con el método `match`:

```
Route::match(['get', 'post'], '/greeting', function () {
   // Tratar la respuesta en caso de que alguien envíe una petición GET o POST a esta ruta
});
```

O incluso se puede definir una respuesta a cualquier verbo HTTP, con el método `any`:

```
Route::any('/greeting', function () {
   // Tratar la respuesta en caso de recibir cualquier tipo de verbo HTTP en esta ruta
});
```

Ruta Fallback

Con el método `Route::fallback` se puede definir una ruta que se ejecutará cuando ninguna otra ruta definida previamente coincida con la petición entrante. Esto permite evitar y personalizar el mensaje 404 que por defecto Laravel devuelve en estos casos.

Veámoslo en un ejemplo:

```
Route::fallback(function () {
   // Escribir aquí el código de la función
});
```

Redirección de rutas

En Laravel también se pueden redirigir rutas con el método `Route::redirect`.

```
// Redirigir la ruta /greeting a la ruta /home
Route::redirect('/greeting', '/home');
```

Importante

Cuando se definen múltiples respuestas a un mismo URI (en el ejemplo, el URI `/greeting`), la prioridad se establece mediante el orden en el que están definidas las rutas. Por

este motivo, los métodos `get`, `post`, `put`, `patch`, `delete` y `options` se han de definir antes que los métodos `match`, `any` y `redirect`, mientras que la ruta para *fallback* se tendrá que definir en último lugar.

Enrutamiento a vistas

Laravel también permite enrutar directamente a la capa vista por medio del método `Route::view`.

```
// La ruta /greeting devuelve la vista greeting
Route::view('/greeting', 'greeting');

// La ruta saludo devuelve la vista saludo, a la que se le pasa un parámetro
Route::view('/greeting', 'greeting', ['name' => 'Taylor']);
```

2.2.3. Listado de rutas

El comando `route:list` de Artisan devuelve las rutas definidas de una forma rápida y sencilla. Además, en la versión 9 de Laravel se ha mejorado la visualización de la lista:

Figura 5. Visualización de la lista

```
// Laravel 9
$ php artisan route:list

GET|HEAD   posts ..................................................... posts.index › PostController@index
POST       posts ..................................................... posts.store › PostController@store
GET|HEAD   posts/create ........................................... posts.create › PostController@create
GET|HEAD   posts/{post} ............................................ posts.show › PostController@show
PATCH      posts/{post} ........................................... posts.update › PostController@update
GET|HEAD   posts/{post}/edit ....................................... posts.edit › PostController@edit
POST       posts/{post}/pin ......................................... posts.pin › PinPostController
POST       posts/{post}/publish ................................ posts.publish › PublishPostController
POST       posts/{post}/unpin ..................................... posts.unpin › UnpinPostController
```

Fuente: elaboración propia.

El listado del comando `route:list` de Artisan muestra los siguientes datos de cada ruta:

- El verbo HTTP (`GET|HEAD`, `POST`, etc.).
- El URI (`post`, `post/create`, etc.). Los valores entre claves son parámetros de ruta.
- El nombre de la ruta (`posts.index`, `posts.store`, `posts.create`, etc.).
- La acción asociada a la ruta. Por ejemplo, `PostController@store` indica que se llamará al método *store* de la clase `PostController`.
- *Middlewares* asociados a la ruta. Este campo es opcional y se muestra con `php artisan route:list -v`.

2.2.4. Parámetros de ruta

En ocasiones habrá que leer segmentos de la URL, como por ejemplo un identificador de una noticia. Esto se consigue definiendo ciertos parámetros en las rutas:

```
Route::get('/news/{id}', function ($id) {
   return 'Se ha accedido a la noticia de identificador '.$id;
});
```

Se pueden definir tantos parámetros como sean necesarios:

```
Route::get('/news/{newsId}/remarks/{remarkId}', function ($newsId, $remarkId)
{
   // Acciones que realizar
});
```

También se puede indicar que ciertos parámetros son opcionales simplemente añadiendo el símbolo `?` al parámetro:

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

En caso de querer inyectar dependencias en una ruta, los parámetros se añadirán después de las dependencias:

```
use Illuminate\Http\Request;

Route::get('/news/{id}', function (Request $request, $id) {
   // Acciones que realizar
});
```

Inyección de dependencias

Es una técnica utilizada para implementar el patrón de diseño orientado a objetos conocido como inversión de dependencias. Más información:

https://desarrolloweb.com/articulos/patron-diseno-contenedor-dependencias.html

Expresiones regulares en parámetros de rutas

Laravel 9 también permite emplear expresiones regulares que limiten el formato de los parámetros:

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

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

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

Algunas de las expresiones regulares más empleadas cuentan con métodos `helpers` que facilitan su uso:

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

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

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

Route::get('/category/{category}', function ($category) {
   //
})->whereIn('category', ['economy', 'culture', 'sports']);
```

Se mostrará un error 404 si una petición no encaja en ningún patrón.

Helper

Es una función de ayuda que realiza una tarea o procedimiento específico, y que puede emplearse más de una vez a lo largo del proyecto. Más información: https://laravel.com/docs/9.x/helpers

Cabe destacar el *helper* `dd()` (*dump and die*), que muestra el contenido de la variable que se le pasa como argumento y finaliza la ejecución de la aplicación. Más información: https://laravel.com/docs/9.x/helpers#method-dd

Laravel también permite definir restricciones fuera del fichero de rutas con el método `pattern`.

Enlace recomendado

Sobre el método `pattern`: https://laravel.com/docs/9.x/routing#parameters-glo bal-constraints

2.2.5. Rutas con nombre

Por practicidad, en Laravel 9 se recomienda asociar un nombre a cada ruta. A continuación, vemos un par de ejemplos:

```
Route::get('/contact', function () {
   // Código de la función
})->name('contact');
```

```
Route::get(
      '/contact',
       [ContactController::class, 'show']
})->name('contact');
```

Los nombres de ruta se utilizan para generar URL o redirecciones sin necesidad de escribir la URL real.

Nota

Los nombres de ruta han de ser únicos para evitar conflictos de nombres.

```
// Generar URL a la ruta de nombre contact:
$url = route('contact');


// Generar redirección a la ruta de nombre contact:
return redirect()->route('contact');
```

Además, si en un futuro la URL real cambia (por ejemplo, `/contacto` en lugar de `/contact`), este cambio no afectará a las rutas con nombre.

Las rutas con nombre también aceptan parámetros. En el siguiente ejemplo se muestra cómo pasarlos:

```
// Archivo de rutas:
Route::get('/news/{id}/category', function ($id) {
   // Código de la función
})->name('category');
```

```
// Generar URL a la ruta de nombre "category" y paso de un parámetro:
$url_1 = route('category', ['id' => 1]);
  // En este caso, $url_1 será igual a /news/1/category

// Generar URL a la ruta de nombre "category" y paso de dos parámetros
$url_2 = route('category', ['id' => 1, 'shortened' => 'no']);
  // En este caso, $url_2 será igual a /news/1/category?shortened=no
```

Más adelante veremos que los nombres de ruta también son útiles para referenciar rutas en las vistas sin necesidad de emplear URL.

2.2.6. *Middleware* en rutas

En Laravel, un *middleware* es una clase que permite filtrar rutas. Un ejemplo clásico consiste en añadir a una ruta (o grupo de rutas) un *middleware* para restringir el acceso a estas si un usuario no está autentificado en la aplicación. En Laravel esto se consigue con el *middleware* `authenticate`, que viene instalado por defecto.

En el siguiente ejemplo, solo se podrá acceder al URI `administration` si el usuario previamente se ha registrado. En caso contrario, se devolverá el URI `/login`.

Enlace recomendado

Ampliación sobre middlewares: https://laravel.com/docs/9.x/middleware

```
Route::get('administration', [UserController::class, 'show'])->middleware('auth');
```

2.2.7. Grupos de ruta

En Laravel se pueden crear grupos de rutas para especificar acciones comunes a todas ellas. Por ejemplo, aplicar un *middleware*, un controlador, un subdominio, un prefijo o un espacio de nombres.

Enlace recomendado

En la documentación oficial de Laravel 9 se puede encontrar más información y ejemplos: https://laravel.com/docs/9.x/routing#route-groups

2.2.8. *Model binding* en rutas

Uno de los patrones de rutas más empleados consiste en buscar un recurso a partir de un identificador. En estos casos habrá que consultar la base de datos para gestionar las acciones de la ruta de forma adecuada. Por ejemplo, si queremos devolver el correo electrónico de un usuario guardado en la base de datos, podríamos escribir el siguiente código:

```
use App\Models\User;
```

```
Route::get('/users/{user}', function (User $user) {

   // En caso de no encontrar el usuario en la base de datos, Laravel devolvería

   // de forma automática un mensaje de error 404.

   return $user->email;
});
```

Este ejemplo responde al patrón *implicit model binding* en ruta, o enlazado de datos de tipo implícito.

Laravel permite enlazar datos de formas algo más avanzadas.

Enlace recomendado

Para ampliar este apartado, se puede consultar la documentación oficial: https://laravel.com/docs/9.x/routing#route-model-binding

2.3. Controladores

Hasta ahora hemos visto cómo llamar a funciones directamente desde el archivo de rutas. Sin embargo, en la práctica resultará mejor gestionar la lógica de negocio mediante clases, que se situarán en la capa controlador del patrón MVC.

Enlace recomendado

Formación sobre clases en PHP: https://laracasts.com/series/php-for-beginners/episodes/12

En este apartado estudiaremos el funcionamiento de un controlador básico, un controlador de recursos y el uso de *middlewares* en controladores.

2.3.1. Controladores de escritura

Controlador básico

En Laravel se puede crear un controlador con Artisan de la siguiente forma:

```
php artisan make:controller UserController
```

Este comando generará el fichero `UserController.php` en la carpeta en la que se almacenan los controladores, `app/Http/Controllers`:

```
// controlador creado en app/Http/Controllers

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
```

```
class UserController extends Controller
{
   /**
    * Se escribirá aquí el código del controlador
    *
    */
}
```

Como se puede observar, el controlador generado consta de una clase, que a su vez extiende la clase `Controller`, de la que heredará métodos y propiedades.

A partir de este controlador base, se podrían aplicar los siguientes cambios para implementar un controlador básico en Laravel:

```
// controlador creado en app/Http/Controllers

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;

class UserController extends Controller
{
   /**
    * Mostrar información del perfil del usuario seleccionado
    *
    * @param  int  $id
    * @return \Illuminate\View\View
    */
   public function show($id)
   {
      /**
      * Este método consulta el perfil seleccionado en la base de datos
      * y devuelve la vista user.profile, a la que se le pasa el perfil
      * del usuario como parámetro
      *
      */
       return view('user.profile', [
           'user' => User::findOrFail($id)
       ]);
   }
}
```

Nótese que en este controlador se define un espacio de nombres (*namespace*) del controlador base de Laravel, `App\Http\Controllers`. Los espacios de nombres en PHP permiten crear aplicaciones complejas de forma organizada, flexible, mejorando la legibilidad del código y evitando problemas de conflictos entre clases.

Enlace recomendado

Ampliación sobre *namespaces* en PHP: https://laracasts.com/series/php-for-beginners/episodes/24

Una forma de crear una ruta al método **`show()`** del controlador anterior es:

```
// fichero de rutas: routes/web.php

use App\Http\Controllers\UserController;


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

En este ejemplo, cuando la aplicación reciba una solicitud que coincida con el URI especificado `(/user/1)` se llamará al método `show()` de la clase `App\Http\Controllers\UserController` y a este método se le pasarán los parámetros de la ruta (por ejemplo, para el URI `/user/1`, el parámetro sería `1`).

Nota

Los controladores no están obligados a extender la clase base `Controller`. Sin embargo, si se hace (Artisan por defecto lo hace), se tendrá acceso a funciones muy prácticas, como son los métodos `middleware` (filtros) y `authorize`.

Más información: https://github.com/laravel/framework/blob/5.8/src/Illuminate/Routing/Controller.php

https://laravel.com/api/9.x/Illuminate/Routing/Controller.html

Controladores de acción única

En general, si las acciones que realiza un controlador son más bien complejas, se recomienda escribir una clase controlador dedicada. A su vez, si un controlador solo ha de dar servicio a una ruta (controlador de acción única), no será necesario definir métodos nuevos. Se puede utilizar el método mágico `__invoke`:

Enlace recomendado

Ampliación sobre métodos mágicos en PHP: https://www.php.net/manual/es/language.oop5.magic.php

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
```

```
class ProvisionServer extends Controller
{
   /**
    * Proporciona un nuevo web server
    *
    * @return \Illuminate\Http\Response
    */
   public function __invoke()
   {
       // Código de la función
   }
}
```

Las rutas a controladores de acción única no necesitan especificar ningún método de la clase controlador. En la ruta simplemente se especificará el nombre del controlador. Siguiendo el ejemplo anterior:

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

Con Artisan se puede generar un controlador de tipo `__invoke` de forma sencilla con el comando:

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

2.3.2. *Middleware* en controladores

En la sección enrutamiento hemos visto qué es un *middleware* y cómo emplearlos en el fichero de rutas. Sin embargo, en la práctica puede resultar más directo emplear *middlewares* (sobre rutas) en la capa controlador. Bastaría con añadir los *middlewares* en el constructor del controlador. A modo de ejemplo:

```
class UserController extends Controller
{
   /**
    * El constructor instancia un nuevo controlador
    *
    * @return void
    */
   public function __construct()
```

```
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

También se pueden emplear *middlewares* dentro de controladores por medio de funciones clausura:

```
$this->middleware(function ($request, $next) {
    return $next($request);
});
```

2.3.3. Controladores de tipo *resource*

Laravel incorpora un tipo de controlador especial llamado *controlador de recursos* (*resource controller*). Este tipo de controlador asigna automáticamente las rutas típicas de creación, lectura, actualización y eliminación (CRUD) a un controlador. Para crearlas, basta con escribir una sola línea de código en el archivo de rutas que emplee `Route::resource`.

```
use App\Http\Controllers\PhotoController;

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

Las rutas que genera este controlador de recursos son:

Figura 6. Respuesta a `php artisan route:list`

```
GET|HEAD        photos ..................................................... photos.index › PhotoController@index
POST            photos ..................................................... photos.store › PhotoController@store
GET|HEAD        photos/create .............................................. photos.create › PhotoController@create
GET|HEAD        photos/{photo} ............................................. photos.show › PhotoController@show
PUT|PATCH       photos/{photo} ............................................. photos.update › PhotoController@update
DELETE          photos/{photo} ........................................... photos.destroy › PhotoController@destroy
GET|HEAD        photos/{photo}/edit ........................................ photos.edit › PhotoController@edit
```

Fuente: elaboración propia.

En Laravel también se pueden definir múltiples controladores de recursos de la siguiente forma:

```
Route::resources([
    'photos' => PhotoController::class,
    'opinions' => OpinionController::class,
]);
```

Resumiendo, un controlador de recursos gestionará el siguiente enrutamiento:

Tabla 4. Enrutamiento del controlador de recursos

Verbo HTTP	URI	Acción	Nombre de la ruta
GET/HEAD	/photos	index	photos.index
GET/HEAD	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET/HEAD	/photos/{photo}	show	photos.show
GET/HEAD	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Se puede crear un controlador de recursos de forma sencilla con el siguiente comando de Artisan:

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

Esta instrucción creará un nuevo controlador llamado `PhotoController.php` en el directorio `app/Http/Controllers/`. Además, este controlador incorporará métodos (vacíos, por defecto) para cada una de las acciones asociadas a los verbos HTTP mencionados anteriormente (`index`, `create`, `store`, `show`, `edit`, `update` y `destroy`).

Personalización de la respuesta HTTP 404

Como hemos visto, Laravel devolverá un error 404 si no se encuentra el recurso solicitado. Sin embargo, este comportamiento se puede personalizar mediante el método `missing`. Veamos un ejemplo en el que, si no se encuentra la sección photos, se redirigirá a la sección opinions:

```
use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
       ->missing(function (Request $request) {
           return Redirect::route('opinions.index');
       });
```

Enlaces recomendados

Ampliación: Inyección de dependencias en controladores.

Inyección de dependencias en el constructor: https://laravel.com/docs/9.x/controllers#constructor-injection

Inyección de dependencias en métodos: https://laravel.com/docs/9.x/controllers#method-injection

3. Vistas y plantillas Blade

Como hemos visto en la sección sobre enrutamiento y controladores, una vista se puede devolver directamente desde una ruta o desde un controlador. Sin embargo, no resulta práctico devolver código HTML directamente desde rutas y controladores. Gracias a las vistas se puede guardar HTML en archivos dedicados y a la vez separar la lógica de negocio de la lógica de presentación.

3.1. Visualización de datos

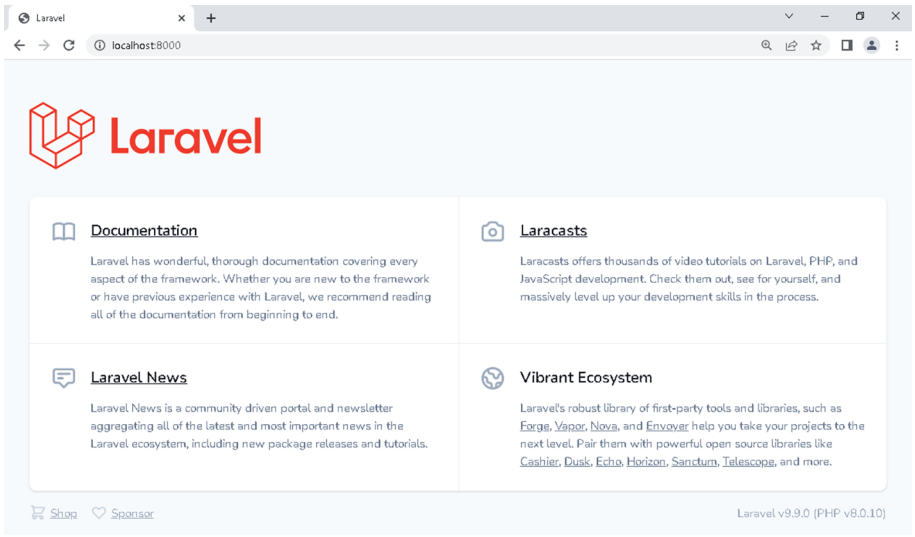
En la sección de enrutamiento también hemos visto ejemplos de llamadas a vistas mediante el *helper* **`view`**, así como del paso de datos a la capa de presentación (vista) por medio del segundo argumento. En este nuevo ejemplo:

```
/* Ejemplo de vista devuelta desde una ruta */

Route::get('/greeting', function () {
   return view('greeting', ['VariableFromRoute' => 'Taylor']);
});
```

`greeting` es el nombre de una vista que se tendrá que guardar en el directorio `resources/views` con el nombre `greeting.blade.php`.

De hecho, cuando instalamos Laravel, por defecto ya integra una vista que muestra un mensaje de bienvenida. Esta vista se llama `welcome.blade.php` y se encuentra en la ruta de vistas mencionada.

Figura 7. Página de inicio de Laravel



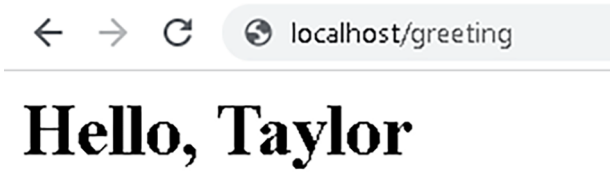
Fuente: elaboración propia.

Siguiendo el ejemplo anterior de la aplicación que devuelve un saludo, una vista sencilla podría ser:

```
<!-- Ejemplo de vista guardada como resources/views/greeting.blade.php -->

<html>
   <body>
       <h1>Hello, {{ $VariableFromRoute }}</h1>
   </body>
</html>
```

Este ejemplo mostrará «Hello, Taylor» cuando en el navegador se escriba el URI `/greeting`:

Figura 8. Saludo



Fuente: elaboración propia.

En esta vista llama la atención el uso de la variable `$VariableFromRoute` entre los símbolos `{{ }}` para emplear la función *echo* de PHP. Esto se consigue gracias al motor de plantillas Blade.

Eliminar el segmento

En Laravel, por defecto la aplicación se mostrará en el URI `/public` (en el ejemplo sería `localhost/public/greeting`). Sin embargo, este segmento del URI se puede eliminar siguiendo las indicaciones de este recurso: https://www.tutsmake.com/how-to-remove-public-from-url-in-laravel/

Un motor de plantillas en PHP es una solución tecnológica que permite añadir PHP en un documento HTML sin necesidad de emplear la sintaxis propia de PHP. Se puede ver como una capa de abstracción de PHP que facilita su uso.

Laravel incorpora el motor de plantillas llamado Blade. A diferencia de otros motores de plantillas, Blade también permite utilizar directamente la sintaxis de PHP nativa, aunque no es lo más recomendable.

Blade apenas afecta al rendimiento de la aplicación, porque las plantillas de Blade se compilan en código PHP y se almacenan en caché.

Como hemos visto, los archivos de plantilla Blade se almacenan en el directorio `resources/views` y utilizan la extensión `.blade.php`. Esta extensión informa al *framework* de que el archivo contiene una plantilla Blade. Las vistas también se pueden guardar con extensión `.php`, pero en este caso Laravel no compilará las instrucciones Blade.

Las plantillas Blade contienen, además de HTML, directivas que permiten mostrar variables de forma sencilla y crear estructuras de control, como condiciones y bucles, entre otras funciones.

3.2. Directorios de vistas anidadas

Las vistas también se pueden anidar dentro de subdirectorios en **`resources/views`**. En este caso se emplea una notación de «puntos» para acceder a ellas. Por ejemplo, si una vista está guardada en `resources/views/admin/profile.blade.php`, se podrá devolver desde rutas (o controladores) de la siguiente manera:

```
return view('admin.profile', $data);
```

Para evitar conflictos, los nombres de los directorios no han de contener el símbolo «`.`».

3.3. Paso de datos a vistas

Tal como hemos visto en ejemplos anteriores, a una vista se le pueden pasar datos para que estén disponibles en la capa de presentación. Recuperemos uno de los ejemplos:

```
/* Ejemplo de vista devuelta desde una ruta o bien desde un controlador */

return view('greeting', ['name' => 'Marina']);
```

Si se le pasa información de esta manera, los datos se convertirán en un array con pares clave y valor. En el ejemplo, a la vista se le pasará el array `$name`.

En la vista se podrá acceder a estos datos de dos formas distintas:

1) Con comandos Blade. En la vista simplemente bastará con utilizar `{{ }}` y escribir:

```
<!-- Ejemplo de vista guardada como resources/views/greeting.blade.php -->

<html>
   <body>
       <h1>Hello, {{ $name }}</h1>
   </body>
</html>
```

2) Aunque de forma general no es la opción recomendada, también se puede utilizar código PHP nativo:

```
<!-- Ejemplo de vista guardada como resources/views/saludo.blade.php -->

<html>
   <body>
      <?php
   echo "<h1>Hello, $name</h1>";
   ?>
   </body>
</html>
```

En ambos casos el resultado final será el mismo:

Figura 9. Saludo

Fuente: elaboración propia.

3.3.1. El método *with*

El método `with` ofrece una alternativa al paso de datos con array:

```
/* Ejemplo de vista devuelta desde una ruta o bien desde un controlador */

return view('greeting')
           ->with('name', 'Marina')
           ->with('occupation', 'study at the UOC');
```

En este caso, la vista podrá consultar las variables `$name` y `$occupation`, cuyos valores se han asignado en la llamada a la vista:

```
<!-- Ejemplo de vista guardada como resources/views/greeting.blade.php -->

<html>
   <body>
       <h1>{{ $name }}'s occupation is {{ $occupation }}</h1>
   </body>
</html>
```

Y el resultado en el navegador será:

Figura 10. Saludo

Marina's occupation is study at the UOC

Fuente: elaboración propia.

3.4. Optimización de vistas

Por defecto, las plantillas Blade se compilan cuando se llaman. Cuando esto sucede, Laravel consulta si existe una versión compilada de esta. En caso afirmativo, Laravel averigua si la vista sin compilar se ha modificado más recientemente que la compilada. En caso de que la vista compilada no exista, o se hayan realizado cambios en la no compilada, Laravel volverá a compilarla.

Esta manera de actuar puede perjudicar ligeramente el rendimiento de la aplicación. Para solucionarlo, Laravel incorpora el comando `view:cache` de Artisan, que precompila todas las vistas utilizadas. Para mejorar el rendimiento, se recomienda añadir este comando dentro del proceso de implementación:

```
php artisan view:cache
```

También se pueden borrar las cachés de las vistas con el comando `view:clear` de Artisan:

```
php artisan view:clear
```

3.5. Directivas Blade

Hemos visto que en las vistas de Blade podemos escribir código HTML y también mostrar el valor de variables que provienen de rutas o controladores. Sin embargo, Blade es un sistema bastante sofisticado que ofrece más posibilidades, como por ejemplo definir estructuras de control, heredar plantillas y mejorar el uso de formularios. Esto se consigue con etiquetas propias llamadas directivas.

Las directivas de Blade emplean el prefijo @. A continuación, se muestran ejemplos de algunas de las directivas más empleadas:

1) Estructura condicional `@if`:

```
{{-- Ejemplo de comentario en Blade. No se mostrará en HTML --}}

@if ($hour < 12)
   <p>Good morning!</p>
@elseif ($hour < 17)
   <p>Good afternoon!</p>
@else
   <p>Good evening!</p>
@endif
```

2) Condiciones `@isset` y `@empty`:

```
@isset($hour)
   // la variable $hour se ha definido y, por tanto, no es null
```

```
@endisset

@empty($hour)
   // la variable $hour no se ha definido previamente (es null)
@endempty
```

3) Conmutador `@switch`:

```
@switch($day)
   @case(0)
       <p>Today is Monday</p>
       @break

   @case(1)
       <p>Today is Tuesday</p>
       @break

   {{-- etc --}}

   @default
       <p>Day of the week either not defined or incorrectly defined</p>
@endswitch
```

4) Bucles: `@for`, `@foreach`, `@while` y variable `$loop`:

```
@for ($i = 0; $i < 10; $i++)
   <p>Iteration {{ $i }}</p>
@endfor

@foreach ($users as $user)
   @if ($loop->first)
       <p>This is the first user.</p>
   @endif

   @if ($loop->last)
       <p>This is the last user.</p>
   @endif

   <p>Dealing with the user: {{ $user->id }}</p>
@endforeach

@while (true)
   <p>We are in a infinite loop!</p>
```

```
@endwhile
```

5) Autenticación: `@auth` y `@guest`:

```
@auth   {{-- o bien @auth('admin') para verificar la guarda --}}
   <p>User authenticated</p>
@endauth



@guest   {{-- o bien @guest('admin') para verificar la guarda --}}
   <p>User <b>not</b> authenticated</p>
@endguest
```

Guarda de Laravel

En Laravel, una guarda (*guard*) es una manera de proporcionar la lógica que se utiliza para identificar a los usuarios autenticados. Laravel proporciona diferentes tipos de guarda, como son las sesiones y los *tokens*. La guarda de sesión mantiene el estado del usuario en cada solicitud mediante *cookies*, y la guarda de *token* determina si un usuario está autenticado mediante un *token*.

6) Formularios: `@csrf`, `@method`, `@error`:

```
<!-- /resources/views/post/form.blade.php -->

<form method="POST" action="{{ route('register' }}">
   <!-- Se deberá añadir el siguiente campo oculto para proteger
        formularios creados en Laravel de posibles ataques
        CSRF:
   -->
    @csrf

   <!-- Laravel permite añadir un método PUT, PATCH o DELETE en un
        formulario (técnica no soportada por HTML) mediante
        el siguiente campo oculto:
   -->
   @method('PUT')

</form>
```

```
<!-- /resources/views/post/form.blade.php -->

<form method="POST" action="{{ route('register' }}">
    @csrf
```

```
    <label for="title">Name and surname: </label>

    <!-- La directiva «error» permitirá comprobar si existen errores
         de validación
    -->
    <input id="title"
       type="text"
       class="@error('title') is-invalid @enderror">

    @error('title')
       <div class="alert alert-danger">{{ $message }}</div>
    @enderror

    <label for="email">E-mail: </label>

    <input id="email"
       type="email"
       class="@error('email', 'login') is-invalid @else is-valid @enderror">

    <!-- Se puede especificar el tipo de error en el segundo parámetro -->
    @error('email', 'login')
       <div class="alert alert-danger">{{ $message }}</div>
    @enderror

    <input type="submit" value="Submit">

</form>
```

Form requests de Laravel

En formularios de Laravel, los *form requests* se pueden utilizar para crear reglas de validación complejas. Más información en:

https://laravel.com/docs/9.x/validation#form-request-validation

https://www.youtube.com/watch?v=Ze-Sg2BT3mc

Nombres de ruta

Como adelantamos en la sección de enrutamiento, los nombres de ruta también son útiles para referenciar rutas en las vistas sin necesidad de emplear URL. En el siguiente tutorial se ilustra con ejemplos:

https://aprendible.com/series/fundamentos-de-laravel-9/lecciones/rutas-con-nombre-en-laravel-9

https://aprendible.com/series/fundamentos-de-laravel-9/lecciones/blade-el-motor-de-plantillas-de-laravel-9

Enlace recomendado

Ampliación sobre directivas Blade: https://laravel.com/docs/9.x/blade#blade-directives

3.6. Diseño de *layouts*

Laravel permite aplicar un mismo diseño HTML en varias páginas. Esta propiedad es muy interesante porque permite unificar el diseño, reutilizar código y, en consecuencia, ahorrar trabajo y minimizar errores.

Recordatorio

Después de actualizar una plantilla Blade, se recomienda eliminar todas las vistas Blade almacenadas en caché con el comando: `php artisan view:clear`.

3.6.1. *Layouts* a partir de componentes

Imaginemos que estamos creando una aplicación para gestionar tareas. Podríamos definir un componente Blade de tipo *layout* como el siguiente:

```
<!-- resources/views/components/layout.blade.php -->

<html>
   <head>
       <title>Example of Layout using Component</title>
   </head>
   <body>
       <h1>{{ $title ?? 'Title by default: Pending tasks' }}</h1>
       <hr/>
       {{ $slot }}
   </body>
</html>
```

Cabe destacar que los componentes Blade se han de guardar en la ruta `resources/views/components`.

Componentes y *slots* de Blade

Los componentes y los *slots* de Blade son técnicas relativamente nuevas en Laravel que evitan duplicar código en las vistas Laravel.

Más información: https://laravel.com/docs/9.x/blade#components

Nótese que en la vista anterior se ha utilizado una variable llamada `$slot`. Se trata de una variable predefinida que se utiliza para inyectar contenido en un componente de Blade.

Este componente también mostrará el contenido de `$title` siempre y cuando se le haya asignado un valor en la llamada. En caso contrario, se mostrará el título que se ha definido por defecto. Esta condición se implementa con los símbolos `??`. Más adelante veremos su aplicación con un ejemplo.

Enlace recomendado

Se puede ampliar esta información en https://laravel.com/docs/9.x/blade#slots

A partir del componente (*layout*) anterior ya podríamos definir una vista que lo utilice. En el siguiente ejemplo se crea una vista que mediante un bucle muestra el contenido de la lista de tareas:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
```

```
    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

Como en esta vista no se ha personalizado el título, se mostrará el título por defecto que hemos definido en el componente. En caso de querer personalizar el título podríamos escribir:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    <x-slot:title>
        Custom title to be defined here
    </x-slot>

    @foreach ($tasks as $task)
        <p> {{ $task }} </p>
    @endforeach
</x-layout>
```

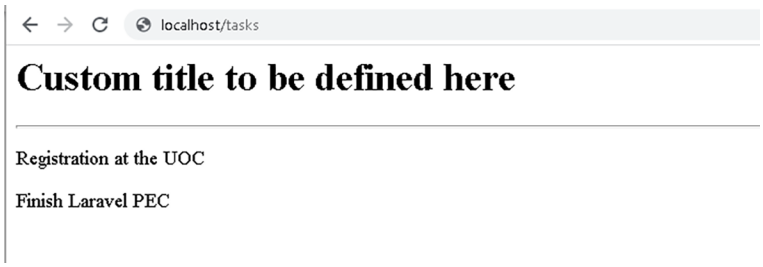
Y finalmente podríamos definir una ruta para mostrar la vista `tasks` como sigue:

```
<!-- routes/web.php -->

use App\Models\Task;

Route::get('/tasks', function () {
    return view('tasks', ['tasks' => Task::all()]);
});
```

Como veremos más adelante en la sección dedicada a las bases de datos, la instrucción `Task::all()` devolverá todas las tareas que se hayan añadido al modelo tarea.

El resultado final con el título que se ha personalizado en `tasks.blade.php` sería:

Figura 11. Resultado con título personalizado



Fuente: elaboración propia.

En caso contrario, se mostrará el título que se ha definido por defecto:

Figura 12. Título por defecto

Fuente: elaboración propia.

3.6.2. *Layouts* con herencia de plantillas

Además del diseño de *layouts* a partir de componentes y los *slots*, Laravel 9 también soporta la técnica conocida como herencia de plantillas. Veamos cómo funciona a partir de un ejemplo.

En primer lugar, crearemos una vista plantilla. Esta vista definirá una estructura común que podremos aplicar en las páginas de la aplicación que consideremos:

```
<!-- resources/views/layouts/app.blade.php -->

<html>
   <head>
       <title>Application name: @yield('title')</title>
   </head>
   <body>
       @section('sidebar')
           Layout sidebar
       @show

       <div class="container">
           @yield('content')
       </div>
   </body>
```

```
</html>
```

Como se puede observar, en esta vista tenemos, además de etiquetas HTML, dos directivas Blade, que son `@section` y `@yield`. La primera se utiliza para definir una sección de contenido y la segunda se emplea para inyectar contenido en la sección correspondiente.

Para poder probar esta plantilla, crearemos una vista de ejemplo que heredará la plantilla en cuestión:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Website title')

@section('sidebar')
   @parent

   <p>By using «parent» directive, this content is added to the default content of the
   layout sidebar.</p>

@endsection

@section('content')
   <p>Body content.</p>
@endsection
```

La directiva `@extends` se utiliza para heredar una plantilla (recordar que un punto en un nombre de vista indica vista anidada en subdirectorio). Por otra parte, la directiva `@parent` permite añadir contenido (en lugar de sobrescribir) al que ya se ha definido por defecto en la barra lateral `(sidebar)`.

Enlace recomendado

Más ejemplos de *layouts* a partir de componentes y con herencia de plantillas en Laracasts: https://laracasts.com/series/laravel-8-from-scratch/episodes/15

4. Bases de datos, migraciones y modelos

En general, las aplicaciones web requieren interactuar con bases de datos para persistir y consultar datos de la aplicación (por ejemplo, usuarios registrados de la aplicación, productos de una tienda en línea, etc.). El *framework* Laravel facilita esta tarea principalmente gracias al ORM Eloquent (visto en la introducción) y a la herramienta Query Builder. En este apartado veremos estas dos herramientas junto con otras funcionalidades relacionadas con bases de datos en Laravel.

4.1. Configuración de la base de datos

En el momento de publicar estos materiales, Laravel 9 es compatible con hasta cinco tipos distintos de bases de datos:

1) MariaDB 10.2+

2) MySQL 5.7+

3) PostgreSQL 10.0+

4) SQLite 3.8.8+

5) SQL Server 2017+

Aunque el archivo principal que contiene la configuración de la base de datos es `config/database.php`, los datos de configuración se deberán guardar en el archivo `.env`. Veámoslo en un ejemplo:

```
# Fichero .env:
#    Solo se muestran las variables de entorno de relacionadas
#    con la base de datos
#
# Ejemplo:
#    Configuración típica con XAMPP (root por defecto no tiene contraseña)

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

El archivo `.env` por defecto está ubicado en la raíz de la aplicación. Este fichero guarda variables de entorno de la base de datos, así como de otras funcionalidades de Laravel (habilitar modo *debug*, URL de la aplicación, etc.).

Nota

Si se trabaja con Git, habrá que asegurarse de que el archivo `.gitignore` contiene el fichero `.env` excluido.

La herramienta Tinker (que desarrollamos en la sección correspondiente) permite probar de forma sencilla la conexión a la base de datos. En primer lugar, abriremos Tinker:

```
php artisan tinker
```

Y escribiremos:

Enlace recomendado

Más información sobre el archivo .env: https://laravel.com/docs/9.x/configuration#environment - configuration

```
DB::connection()->getPDO();
```

Se devolverá el objeto de datos de PHP (*PHP data object*, PDO, en inglés) si se ha podido conectar a la base de datos. En caso contrario, se devolverá un error y por tanto habrá que revisar la configuración en el fichero `.env`.

Enlaces recomendados

Qué es PDO: https://www.php.net/manual/es/intro.pdo.php

Screencast sobre configuración de bases de datos en Laravel: https://laracasts.com/series/laravel-8-from-scratch/episodes/17

4.2. Modelos y Eloquent

Como hemos visto, Laravel integra el ORM Eloquent. Esta herramienta permite hacer peticiones y consultas complejas a la base de datos sin necesidad de emplear código SQL. Eloquent se puede ver como una capa de abstracción de SQL mediante programación orientada a objetos.

Un ORM generalmente implementa el patrón de diseño *active record*, que permite trabajar con las tablas de la base de datos como si fuesen clases. En un ORM, estas clases se denominan *modelos*.

Enlace recomendado

Más información sobre *active record*: https://researchhubs.com/post/ computing/web-applica tion/ the-active-record-de sign-pattern.html

En Laravel, para generar un nuevo modelo se puede emplear Artisan:

```
php artisan make:model Noticia
```

En este caso se creará el modelo `Noticia.php` y se guardará en la carpeta `app/Models`. El contenido por defecto del modelo será:

```
<?php
```

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Noticia extends Model
{
    //
}
```

Como se puede observar, la clase Noticia extiende la clase Model, de la cual hereda métodos y propiedades.

En este ejemplo, el ORM asume que tenemos una tabla de nombre `noticias` en la base de datos. Nótese que en el modelo se emplea el singular y la primera letra en mayúscula, mientras que en la tabla se utiliza el plural y minúsculas. Se trata de una regla del ORM.

Además, el ORM Eloquent por defecto sigue la convención *snake case* (convención por defecto en bases de datos). Veamos otro ejemplo. Si declaramos un modelo de nombre `ReservasDeCliente` (nótese «cliente» en singular), el ORM asumirá que la tabla se llama `reservas_de_clientes` («clientes» en plural).

Sin embargo, Eloquent es flexible y también permite utilizar tablas que no sigan esta nomenclatura. En este caso, habrá que definir la tabla como propiedad dentro de la clase del modelo:

Enlaces recomendados

Qué es *snake case*: https://www.theserverside.com/definition/ Snake-case

Información general sobre convenciones de nombres: https://juniortoexpert.com/en/naming-convention/

En Laravel: https://laravel.com/docs/9.x/eloquent#table-names

Enlaces recomendados

Ampliación sobre Eloquent:

Screencast: https://laracasts.com/series/laravel-8-from-scratch/episodes/19

Documentación oficial: https://laravel.com/docs/9.x/eloquent

```
class Noticia extends Model
{
   /**
    * Asociar tabla concreta "my_news" al modelo Noticia
    *
    * @var string
    */
   protected $table = 'my_news';
}
```

4.2.1. Query Builder

Una vez creado el modelo y asociado a una tabla, ya podemos realizar consultas por medio del constructor de consultas llamado *Query Builder*. Vamos a obtener, por ejemplo, todas las noticias que estén publicadas y las ordenaremos por título:

```
$news = Noticia::where('published', 1)
                        ->orderBy('title')
                        ->get();
```

Esta consulta devuelve un objeto cuyas propiedades son las columnas y los registros de la tabla. Como puede verse, la estructura de las consultas es muy parecida a las consultas en lenguajes para bases de datos relacionales, como SQL.

Otro ejemplo sería el siguiente, que permite obtener el título de una noticia empleando `$news->title`:

```
use App\Models\Noticia;

foreach (Noticia::all() as $news) {
   echo $news->title;
}
```

No es el objetivo de estos materiales presentar en profundidad Query Builder. Se puede consultar más información en la documentación oficial.

Enlace recomendado

Ampliación sobre Query Builder: https://laravel.com/docs/9.x/queries

4.2.2. Consultas con SQL nativo

Aunque en general no es lo más recomendable, en Laravel también se pueden hacer consultas con SQL nativo mediante el *facade* DB. Veámoslo en un ejemplo:

Enlace recomendado

Más ejemplos de consultas SQL: https://laravel.com/docs/9.x/database# running-queries

```
use Illuminate\Support\Facades\DB;
    /* … */
    $users = DB::select('select * from users');
```

4.3. Migraciones

Las migraciones de Laravel son un sistema de control de versiones orientado a bases de datos. Esto permite que un equipo de desarrolladores pueda modificar sobre una misma base de datos. Además, evita tener que realizar manualmente cambios en esta y, a su vez, mantiene un histórico de todos los cambios aplicados.

Las migraciones generan ficheros PHP que incluyen una descripción de las tablas o los atributos que crear. Si en un futuro se modifica la estructura de la base de datos, las migraciones generarán un nuevo fichero PHP con los cambios por hacer.

4.3.1. Creación, estructura y ejecución de migraciones

Si ya hemos creado manualmente la base de datos y configurado el fichero `.env`, ya estamos en disposición de crear una tabla en la base de datos mediante Artisan:

Nota

como hemos visto en el subapartado 4.2, por convención los nombres de las tablas se han de definir en plural, a pesar de que en este ejemplo *news* no se distinga del singular en inglés.

```
php artisan make:migration create_news_table
```

Este comando crea la primera migración en la ruta `database/migrations`. Se trata de un fichero PHP de nombre `TIME_STAMP_create_news_table.php`. La marca temporal (*time stamp*) permite ordenar las migraciones por orden de ejecución.

Además, este fichero de migraciones define una clase que está preparada para crear o modificar los campos de la tabla con dos métodos:

1) `up()`: es el método que se ejecuta cuando se lanza una migración. En este método se especificará cómo crear o modificar la tabla.

2) `down()`: se especificará cómo deshacer los cambios que se ejecutan en el `up()`, esto es, se añadirá la operación inversa. Este método permitirá revertir cambios, propio de un control de versiones.

Ampliación

Laravel también permite generar directamente una migración cuando se genera un modelo mediante: `php artisan make:model Noticia --migration`.

El fichero de migración `TIME_STAMP_create_news_table.php` contendrá por defecto:

```
<?php

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

return new class extends Migration
{
    /**
     * Ejecutar la migración.
     *
     * @return void
     */
```

```
    public function up()
    {
        Schema::create('news', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Deshacer la migración.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('news');
    }
};
```

Tabla vacía

Por ejemplo, si queremos añadir un campo llamado `title` a la tabla `news` (noticias), el método `up()` podría ser:

```
public function up()
{
    Schema::create('news', function (Blueprint $table) {
        $table->id();
        $table->string('title'); // Nuevo campo
        $table->timestamps();
    });
}
```

Añadir una migración

Si ya existe una tabla y queremos añadir una migración que modifique los campos de esa tabla, podremos emplear: php artisan make:migration add_title_to_news_table –table=news.

El último paso consistirá en lanzar la migración con:

```
php artisan migrate
```

Esta instrucción creará varias tablas en la base de datos:

- Tabla `news`, con las tres columnas que hemos definido.

- Tabla `migrations`, en la que se guardará el historial de cambios de la base de datos.
- Tablas relacionadas con la autenticación (`users`, `personal_access_tokens`, `password_resets`).
- Tabla `failed_jobs`, en caso de emplear colas de trabajo.

Si se quiere volver al estado inicial de la base de datos (*rollback*) se puede conseguir con el comando:

```
php artisan migrate:rollback
```

Enlaces recomendados

Ampliación sobre migraciones en Laravel: https://laravel.com/docs/9.x/migrations

Screencast, crear un modelo y la migración: https://laracasts.com/series/laravel-8-from-scratch/episodes/20

Ampliación sobre bases de datos en Laravel:

Crear tablas: https://laravel.com/docs/9.x/migrations#tables

Crear columnas: https://laravel.com/docs/9.x/migrations#columns

Crear índices: https://laravel.com/docs/9.x/migrations#indexes

Tipos de índice disponibles: https://laravel.com/docs/9.x/migrations#available-index-types

Claves primarias: https://laravel.com/docs/9.x/eloquent#primary-keys

Restricciones de clave externa: https://laravel.com/docs/9.x/migrations#foreign-key-constraints

Relación de muchos a muchos:

https://laravel.com/docs/9.x/eloquent-relationships#many-to-many

https://laravel.com/docs/9.x/eloquent-relationships#many-to-many-polymorphic-relations

Pivot tables:

https://medium.com/@0510winnie/laravel-eloquent-relationships-many-to-many-ce3e83ef8ebe

https://styde.net/pivot-tables-con-eloquent-en-laravel/

Ejemplos: https://appdividend.com/2022/01/21/laravel-many-to-many-relationship/

Paginación de resultados: https://laravel.com/docs/9.x/pagination

4.4. *Seeding*

Hasta ahora hemos visto cómo crear la estructura de la base de datos mediante las migraciones de Laravel. Para gestionar los registros, Laravel incorpora los `seeders`. Un *seeder* es una herramienta que permite de forma sencilla cargar información en la base de datos.

Los *seeders* funcionan de una forma similar a las migraciones, a excepción de que no se hace un control de versiones.

Veamos cómo funciona siguiendo el ejemplo de la tabla de noticias. Laravel almacena los *seeders* en la carpeta `database/seeders`. Allí por defecto existe un fichero llamado `DatabaseSeeder.php` que deberemos adaptar antes de lanzar el `seeder`:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
   /**
    * Ejecutar el seeder de la base de datos.
    *
    * @return void
    */
   public function run()
   {
       DB::table('news')->insert([
           'title' => Str::random(10),
       ]);
   }
}
```

Como se puede observar, en el método `run()` gestionaremos los registros de la tabla que seleccionemos. En este caso se han utilizado comandos de Query Builder.

Finalmente, a partir de este código, ejecutaremos el *seeder* con:

```
php artisan db:seed
```

Laravel también permite ejecutar *seeders* concretos. Bastaría con crear un *seeder* nuevo con el comando:

```
php artisan make:seeder NewsSeeder
```

A continuación, editaríamos el fichero generado, en este caso `NewsSeeder.php`, que se guardaría igualmente en la carpeta `database/seeders`. Y finalmente ejecutaríamos este *seeder* en concreto mediante:

Enlace recomendado

Ampliación sobre *seeders*: https://laravel.com/docs/9.x/seeding

```
php artisan db:seed --class=NewsSeeder
```

4.4.1. Modelo *factory*

En algunas ocasiones nos interesará cargar nuestra base de datos con registros de información de contenido aleatorio para realizar pruebas. En Laravel esto es posible gracias a la clase *factory* (factoría).

Factory permite crear instancias fácilmente de la siguiente manera:

```
use App\Models\User;

/**
* Run the database seeders.
*
* @return void
*/
public function run()
{
   /* En este ejemplo se generarían 20 noticias
      de contenido aleatorio */
   $news = Noticia::factory()->count(20)->create();

}
```

Enlaces recomendados

Ampliación sobre *factory*:

Documentación oficial:

https://laravel.com/docs/9.x/seeding#using-model-factories

https://laravel.com/docs/9.x/database-testing#defining-model-factories

Screencast: https://laracasts.com/series/laravel-8-from-scratch/episodes/28

Tutorial: https://www.tutsmake.com/laravel-8-factory-generate-dummy-data-tutorial/

Faker

En este contexto resulta muy interesante el uso de la librería `Faker`, que permite personalizar el tipo de datos aleatorios generados para ajustarlos a las necesidades de la aplicación.

Enlaces recomendados

Librería Faker:
https://laravel.com/docs/9.x/database-testing#concept-overview
https://laravel.com/docs/9.x/database-testing#defining-relationships-within-factories
Repositorio oficial de Faker:
https://github.com/fzaninotto/Faker

5. Artisan y Tinker

Como se ha ido viendo a lo largo del documento, Artisan y Tinker son dos herramientas integradas en Laravel que permiten llevar a cabo tareas de desarrollo y testeo de una forma sencilla.

5.1. Artisan

Artisan es una interfaz de línea de comandos que viene integrada en Laravel. Artisan se encuentra por defecto en la raíz de una aplicación Laravel, como *script* de PHP, y proporciona una serie de comandos que facilitan el desarrollo de una aplicación Laravel.

El siguiente comando muestra una lista de todas las funcionalidades disponibles con Artisan:

```
php artisan list
```

También se puede obtener ayuda de Artisan sobre funcionalidades concretas, por ejemplo, ayuda relativa a las migraciones:

```
php artisan help migrate
```

Como hemos ido viendo, con Artisan se pueden generar controladores, modelos, realizar migraciones, consultar el enrutamiento o gestionar la caché de las vistas, entre otras muchas funciones.

5.2. Tinker

Es habitual que un desarrollador web quiera probar código PHP o JavaScript mientras desarrolla una aplicación con el propósito de testearlo. Con JavaScript es relativamente fácil hacerlo, basta con abrir la consola del navegador y hacer pruebas con el código. En PHP, con *scripts* sencillos también es suficiente con emplear alguna técnica del tipo:

- Crear un archivo PHP y ejecutarlo desde la terminal.

- Acceder a un intérprete en línea de PHP y probar el código. Por ejemplo, con el de w3schools: https://www.w3schools.com/php/php_compiler.asp

Sin embargo, este proceso se dificulta cuando trabajamos con entornos más complejos basados en PHP, como el *framework* Laravel o el mismo WordPress, que incluyen más archivos de configuración y procesamiento de datos para llevar a cabo una tarea.

Tinker es un sofisticado REPL (*Read-Eval-Print Loop*) integrado en Laravel que soluciona este problema. Se trata de una consola de comandos con la que se puede interactuar con todas las clases y métodos de la aplicación. Por ejemplo, permite interactuar con las bases de datos mediante Eloquent y hacer acciones como comprobar las relaciones existentes entre las tablas o acciones CRUD. Esto se consigue sin necesidad de modificar el código del proyecto, es decir, sin crear nuevas rutas o formularios para introducir datos, entre otras técnicas.

Enlace recomendado

Qué es un REPL: https://www.digitalocean.com/ community/tutorials/ what-is-repl

Tinker está instalado en Laravel por defecto. Sin embargo, si se ha desinstalado previamente, se puede volver a instalar con:

```
composer require laravel/tinker
```

Para poner en marcha Tinker basta con escribir:

```
php artisan tinker
```

Enlaces recomendados

En el siguiente *screencast* hay ejemplos de acciones ejecutadas en Tinker sobre Laravel 9: https://laracasts.com/series/laravel-8-from-scratch/episodes/24?modal=contact-support

Más información sobre Tinker en la documentación oficial: https://laravel.com/docs/9.x/artisan#tinker

Más ejemplos sobre Tinker en Laravel 8 (también válido para Laravel 9): https://www.youtube.com/watch?v=O2HyI543oAg

5.2.1. Tinkerwell

Tinkerwell se creó para ampliar la funcionalidad de Tinker. Tinkerwell es un editor de código para Laravel que permite escribir código PHP, ejecutarlo y obtener el resultado, tanto en local como en conexiones remotas.

Nota

Tinkerwell originalmente se creó para Laravel, pero con el tiempo se han ido desarrollando nuevas versiones para trabajar con otras plataformas, como Drupal o WordPress, entre otros.

Enlaces recomendados

Ampliación sobre Tinkerwell: https://tinkerwell.app/Más información sobre Tinkerwell y WordPress: https://roelmagdaleno.com/tinkerwell-para-desarrollo-en-wordpress/

6. Autenticación de usuarios

En general, implementar un sistema de autenticación de usuarios en una aplicación web es una tarea compleja y con riesgos potenciales de seguridad. Laravel, sin embargo, permite crear un sistema de autenticación en relativamente pocos pasos y de forma rápida, segura y sencilla.

Kits de inicio

Laravel 9, como en versiones anteriores, proporciona lo que se conoce como kits de inicio o *starter kits* (https://laravel.com/docs/9.x/starter-kits). Gracias a estas herramientas se puede crear toda la estructura de un sistema de autenticación (controladores, rutas y vistas) con la ejecución de un solo comando.

6.1. Laravel Breeze

Laravel Breeze es una de las herramientas incluida en el kit de inicio de Laravel. Esta herramienta permite crear un sistema de autenticación de forma rápida y sencilla. En la práctica, proporcionará las siguientes funcionalidades: inicio de sesión, registro, restablecimiento de contraseña, verificación de correo electrónico, confirmación de contraseña y cierre de sesión.

Laravel Breeze se puede incorporar al *framework* Laravel mediante Composer:

```
composer require laravel/breeze --dev
```

A continuación, instalaremos el sistema de autenticación Breeze con Artisan, compilaremos los ficheros necesarios de la parte del *frontend* con npm (gestor de paquetes de Node.js) y finalmente lanzaremos una migración, que creará las tablas necesarias en la base de datos para gestionar usuarios:

¡Atención!

Antes de instalar Breeze se recomienda hacer una copia del proyecto porque podría sobrescribir ficheros.

```
php artisan breeze:install

npm install
npm run dev
php artisan migrate
```

Con estos sencillos pasos ya tendremos generado todo el sistema de autenticación.

Finalmente, resulta interesante fijarse en los cambios hechos por Breeze en los siguientes ficheros:

1) Rutas:

- `routes/web.php`: se habrá creado una nueva ruta, `/dashboard`, que permite acceder a un panel de control, y un *include* de un nuevo fichero de rutas: `auth.php`.

- `routes/auth.php`: contiene todas las rutas que hacen referencia a la gestión de usuarios (`/login`, `/register`, `/forgot-password`, etc.).

2) Vistas:

- `resources/views/auth`: contiene todas las vistas que ha generado Breeze (`login.blade.php`, `register.blade.php`, etc.).

3) Controladores:

- `app/Http/Controllers/Auth`: se habrán generado tantos controladores como funciones necesarias para el sistema de autenticación (`NewPasswordController.php`, `VerifyEmailController.php`, etc.).

6.2. Utilización del sistema de autenticación

A continuación, veremos ejemplos de cómo utilizar Breeze para desarrollar tareas relacionadas con el sistema de autenticación en Laravel.

Enlaces recomendados

Ampliación:
https://laravel.com/docs/9.x/authentication
https://www.danielprimo.io/blog/laravel-series-vi-instalando-un-sistema-de-autenticacion

1) Obtener información de un usuario autenticado:

```
use Illuminate\Support\Facades\Auth;

// Obtiene el usuario autenticado actual:
$user = Auth::user();

// Obtiene el identificador del usuario autenticado actual:
$id = Auth::id();
```

2) Acceder a los datos de usuario:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class NewsController extends Controller
{
```

```
    /**
     * Actualizar la información de una noticia
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        // $request->user()
    }
}
```

3) Determinar si el usuario actual está autenticado:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // El usuario sí está autenticado
}
```

4) Proteger rutas (también con guardas):

```
Route::get('/news', function () {
    // Solo usuarios autenticados podrán acceder a esta ruta
})->middleware('auth'); // Ejemplo de uso con guarda: middleware('auth:admin')
```

5) Autenticación manual de usuarios:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Gestión de un intento de autenticación
     *
     * @param  \Illuminate\Http\Request  $request
```

```
     * @return \Illuminate\Http\Response
     */
    public function authenticate(Request $request)
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required'],
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();

            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our records.',
        ])->onlyInput('email');
    }
}
```

7. API

Laravel es un *framework* muy interesante también para implementar API de *backend*. Una aplicación típica de estas soluciones *backend* consiste en ofrecer almacenamiento de datos de aplicaciones o autenticación que luego se podrían consumir desde un *framework frontend*, como por ejemplo Vue.js o Next.js.

Ved también

Sobre el almacenamiento de datos de aplicaciones o autenticación ved Laravel Sanctum, en el apartado 8 de este módulo.

Enlaces recomendados

Tutorial: Cómo crear y testear una API Restful en Laravel: https://www.toptal.com/laravel/restful-laravel-api-tutorial

Características avanzadas que ofrece Laravel:

Servicios de correos electrónicos: https://laravel.com/docs/9.x/mail

Servicios de colas: https://laravel.com/docs/9.x/queues

Qué es una API REST

Una API REST (o API RESTful) es una interfaz que permite interactuar con servicios web siguiendo el patrón de diseño REST.

Más información: https://www.redhat.com/es/topics/api/what-is-a-rest-api

7.1. Creación de rutas y controladores de API

En la sección de enrutamiento hemos visto que las rutas de tipo API se han de definir en el fichero `routes/api.php`. A estas rutas Laravel les añadirá automáticamente el segmento `/api` en el URI.

El método `apiresource` permite crear el enrutamiento de una API con los verbos HTTP que más se utilizan. Lo ilustraremos con un ejemplo. Si en el fichero de rutas `routes/api.php` escribimos:

```
use App\Http\Controllers\PhotoController;

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

Se crearán las siguientes rutas y verbos HTTP:

Figura 13. Respuesta a: `php artisan route:list`

```
GET|HEAD        api/photos ..................................... photos.index › PhotoController@index
POST            api/photos ..................................... photos.store › PhotoController@store
GET|HEAD        api/photos/{photo} ............................. photos.show › PhotoController@show
PUT|PATCH       api/photos/{photo} ......................... photos.update › PhotoController@update
DELETE          api/photos/{photo} ....................... photos.destroy › PhotoController@destroy
```

Fuente: elaboración propia

Además, con Artisan se puede crear de forma sencilla un controlador para la API:

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

Este controlador incluirá métodos (por defecto vacíos) para las acciones `index()`, `store()`, `show()`, `update()` y `destroy()`.

En la práctica resulta conveniente proteger las API con autenticación.

Enlace recomendado

Material de consulta: https://laravel.com/docs/9.x/authentication#laravels-api-authentication-services

7.2. Testear API

Existen varias herramientas para testear las API. Estas herramientas facilitan las pruebas gracias a que permiten enviar peticiones de cualquier tipo a un *endpoint* (URL) de la API, especificar las cabeceras, parámetros o guardar las peticiones, entre otras funcionalidades.

Herramientas para testear API

Algunos ejemplos de herramientas para testear API son:

Postman:

https://www.postman.com/use-cases/api-testing-automation/

https://desarrolloweb.com/articulos/como-usar-postman-probar-api

SoapUI: https://www.soapui.org/learn/functional-testing/api-testing-101/

También se pueden testear por línea de comandos con el comando *curl*:

https://ajgallego.gitbook.io/laravel-5/capitulo_5/capitulo_5_curl

Enlaces recomendados

Ampliación:

Testear API (documentación oficial): https://laravel.com/docs/9.x/http-tests

Testear API Laravel con PHPUnit: https://auth0.com/blog/testing-laravel-apis-with-phpunit/

7.2.1. Códigos de estado HTTP habituales

Al implementar una API se deberá definir correctamente la respuesta a una petición. Los códigos que habitualmente se devolverán son:

1) 2xx: Utilizado para respuestas con éxito

- 200 (*OK*). Es el código de éxito por defecto.
- 201 (*Created*). Indica que la petición se ha procesado y se ha creado un nuevo objeto como resultado.
- 204 (*No content*). Sin contenido. Cuando una acción se ejecutó con éxito, pero no hay contenido para devolver.
- 206 (*Partial content*). Contenido parcial. Útil cuando se tiene que devolver una lista paginada de recursos.

2) 3xx: Empleado para redirecciones

- 302 (*Found*). El recurso solicitado ha sido trasladado temporalmente a una URL diferente.

3) 4xx: Indican errores del cliente (en la petición)

- 400 (*Bad request*). Petición incorrecta. Se devuelve en solicitudes que no pasan la validación (parseado de petición incorrecto, por ejemplo).
- 401 (*Unauthorized*). No autorizado. El usuario necesita estar autenticado.
- 403 (*Forbidden*). Prohibido. El usuario está autenticado, pero no tiene los permisos para realizar una acción.
- 404 (*Not found*). No encontrado. Laravel devolverá este código de error cuando no encuentre el recurso solicitado.

4) 5xx: Usados cuando hay algún problema en el servidor

- 500 (*Internal server error*). Error interno del servidor.
- 502 (*Bad gateway*). Puerta de enlace incorrecta. Indica que el servidor trabaja como puerta de enlace y no ha recibido una respuesta válida de los servidores consultados.
- 503 (*Service unavailable*). Servicio no disponible. Como su nombre indica, el servidor no está disponible para gestionar la petición.
- 504 (*Gateway timeout*). Significa que el servidor no ha recibido la respuesta que esperaba de otro servidor intermedio al intentar acceder a un sitio web o completar otra solicitud.

Enlaces recomendados

Ampliación sobre códigos de estado:

https://datatracker.ietf.org/doc/html/rfc7231

https://restfulapi.net/http-status-codes/

8. Ecosistema Laravel y extensiones

Una de las claves del éxito de Laravel es que, a medida que ha ido creciendo, se han ido creando un conjunto de herramientas que simplifican tareas y facilitan el trabajo de los desarrolladores.

Gran parte de estas herramientas se han integrado en el núcleo de Laravel y son gratuitas. Sin embargo, existen otros proyectos dentro del ecosistema de Laravel que ofrecen ciertas funcionalidades de pago y que en parte se utilizan para financiar el desarrollo del *framework*.

Algunas de las herramientas más empleadas y útiles son las siguientes:

Tabla 5. Herramientas de Laravel

Nombre de la extensión y temática	Descripción y URL
Laravel Passport (autenticación)	Proporciona un sistema de autenticación basado en el estándar OAuth2. OAuth2 es un estándar abierto para autorizar API, que permite compartir información entre sitios sin tener que compartir la identidad. Se trata de un mecanismo de autenticación utilizado hoy por grandes compañías como Google, Facebook, Microsoft, Twitter, GitHub o LinkedIn, entre otras muchas. Más información sobre OAuth2: <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2> Instalación, configuración y más información sobre Laravel Passport: <https://laravel.com/docs/9.x/passport>
Laravel Sanctum (autenticación)	Permite llevar a cabo autenticación de forma segura utilizando *cookies* y API *tokens*. A diferencia de Laravel Passsport, Laravel Sanctum no soporta el estándar OAuth2. Se suele emplear en sistemas SPA (aplicaciones de una sola página –*single page applications*–), aplicaciones móviles y API sencillas basadas en *tokens*. Instalación, configuración y más información: <https://laravel.com/docs/9.x/sanctum>
Laravel Socialite (autenticación)	Además de la autenticación típica basada en alta en formularios, Laravel también permite autenticarse con proveedores de OAuth utilizando la librería Laravel Socialite. Laravel Socialite actualmente admite autenticación mediante cuentas de Facebook, Twitter, LinkedIn, Google, GitHub, GitLab y Bitbucket. Instalación, configuración y más información: <https://laravel.com/docs/9.x/socialite>
Laravel Cashier (Stripe) (facturación)	Proporciona una interfaz para servicios de pago con Stripe. Además de la administración básica de suscripciones, Cashier permite gestionar cupones de descuento, cancelar periodos de gracia e incluso generar facturas en PDF. Instalación, configuración y más información: <https://laravel.com/docs/9.x/billing>

Packagist

Es el principal repositorio de Composer. Se emplea para instalar librerías de PHP. Más información: https://packagist.org/

Nombre de la extensión y temática	Descripción y URL
Laravel Breeze (autenticación)	Tal como hemos visto en la sección de autenticación de usuarios, Laravel Breeze es una herramienta que permite crear un sistema de autenticación de forma rápida y sencilla. La capa de vista predeterminada de Laravel Breeze se compone de plantillas Blade simples diseñadas con Tailwind CSS. Además, Breeze permite gestionar las plantillas Blade con Laravel Livewire. Instalación, configuración y más información: <https://laravel.com/docs/9.x/starter-kits#laravel-breeze>
Laravel Dusk (testeo)	Dusk se trata de un componente para realizar pruebas automatizadas en Laravel, que nos permitirá probar aplicaciones desde la perspectiva de un usuario, incluso aquellas que hagan uso de JavaScript en el navegador. Por defecto, Laravel Dusk utiliza ChromeDriver, una herramienta de código abierto para realizar pruebas automatizadas de aplicaciones web. Instalación, configuración y más información: <https://laravel.com/docs/9.x/dusk>
Homestead (entorno de desarrollo)	Laravel Homestead es una *box* oficial de Vagrant. Ofrece un entorno de desarrollo en una máquina local sin necesidad de instalar PHP. Homestead se ejecuta en cualquier sistema operativo Windows, macOS o Linux, e incluye herramientas como Nginx, PHP, MySQL, PostgreSQL, Redis, Memcached, Node y todo el software que se necesita para desarrollar aplicaciones de Laravel. Instalación, configuración y más información: <https://laravel.com/docs/9.x/homestead>
Laravel Horizon (gestión de procesos)	En ocasiones es necesario crear procesos complejos en ejecución y, a la vez, que sean transparentes para el usuario. Por ejemplo, el procesamiento de vídeo, procesos en cascada o envío de múltiples correos electrónicos, entre otros. Para solucionar estas necesidades lo mejor es gestionar estos procesos mediante colas que se ejecutan en *background*. Laravel por defecto ofrece un sistema de colas (*queues*) que se ejecutan de forma síncrona. Afortunadamente, se puede configurar el sistema de colas en Laravel para que se ejecuten en segundo plano, por ejemplo, empleando colas de Redis. Laravel Horizon permite gestionar colas de Redis y además ofrece un panel de control que muestra toda la información del sistema de colas. Instalación, configuración y más información: <https://laravel.com/docs/9.x/horizon> <https://www.cursosdesarrolloweb.es/blog/laravel-horizon-colas-en-laravel-con-redis>
Laravel Mix (*frontend*: CSS y JS)	Laravel Mix proporciona una potente y versátil API que permite definir de forma rápida y sencilla el preprocesado de CSS y JavaScript, entre otras utilidades. Laravel Mix hace que sea muy fácil compilar y minimizar los archivos CSS y JavaScript de una aplicación. Instalación, configuración y más información: <https://laravel.com/docs/9.x/mix>
Inertia.js	Inertia.js permite crear aplicaciones de una sola página sin necesidad de construir una API. No está ligado en sí a Laravel o a Vue.js, pero ofrece un adaptador para estos dos *frameworks*. Instalación, configuración y más información: <https://inertiajs.com/>

Nombre de la extensión y temática	Descripción y URL
Envoyer (despliegue)	Laravel Envoyer es una aplicación que permite desplegar PHP, en general, y Laravel, en particular. Se puede integrar con GitHub, Bitbucket o GitLab. Instalación, configuración y más información: <https://envoyer.io/>
Forge (administración)	Laravel Forge facilita las tareas de administrar servidores, crear dominios, instalar Laravel y desplegar aplicaciones en producción, entre otras utilidades. Instalación, configuración y más información: <https://forge.laravel.com/>
Laravel Vapor (administración)	Laravel Vapor permite desplegar aplicaciones sin necesidad de contar con un servidor (*serverless*) con la tecnología de AWS. Vapor permite administrar varios servicios de AWS mediante API tales como bases de datos, cachés, *networks*, certificados y todo lo que se necesita para desplegar una aplicación desarrollada en Laravel. Instalación, configuración y más información: <https://vapor.laravel.com/>
Laravel Nova (gestión de contenidos)	Laravel Nova es una herramienta enfocada a la gestión de contenidos. Ofrece un panel de administración que se integra dentro del ecosistema de Laravel y que está pensado para que sea sencillo de utilizar y modificar. El *frontend* de Nova es una SPA (*single page application*) que funciona con Vue.js, Vue Router y Tailwind. Instalación, configuración y más información: <https://nova.laravel.com/>
Laravel Scout (búsquedas en modelos)	Laravel Scout es una biblioteca que permite añadir búsquedas de texto completo a los modelos de Eloquent. Principalmente maneja la manipulación de los índices de búsqueda y los sincroniza cada vez que hay un cambio en los datos del modelo. Instalación, configuración y más información: <https://laravel.com/docs/9.x/scout>
Laravel Spark (gestión de usuarios)	Laravel Spark permite crear un proyecto con todo lo necesario para gestionar usuarios, autenticar de dos factores, equipos, pagos en línea, suscripciones y mucho más de forma relativamente sencilla y rápida. Instalación, configuración y más información: <https://spark.laravel.com/>
Laravel Telescope (depuración)	Telescope es un asistente de depuración para Laravel. Proporciona información sobre las solicitudes que se realizan, excepciones, entradas de registro, consultas de bases de datos, trabajos en cola, correo, notificaciones, operaciones de caché, tareas programadas y volcados de variables, entre otras utilidades. Instalación, configuración y más información: <https://laravel.com/docs/9.x/telescope>
Laravel Valet (entorno de desarrollo)	Laravel Valet es un entorno de desarrollo minimalista y muy rápido para **macOS** que además requiere muy pocos recursos. Instalación, configuración y más información: <https://laravel.com/docs/9.x/valet>