CODE SMART

THE LARAVEL FRAMEWORK *VERSION 5* FOR BEGINNERS

DAYLE REES

Laravel: Code Smart (ES)

The Laravel Framework Version 5 for Beginners

Dayle Rees y Antonio Laguna

Este libro está a la venta en http://leanpub.com/codesmart-es

Esta versión se publicó en 2016-11-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 Dayle Rees y Antonio Laguna

Índice general

Reconocimientos

Antes que nada, me gustaría agradecer a mi novia Emma, no solo por animarme con todas mis frikis aventuras, ¡si no también por hacer esas increíbles fotos a los pandas rojos para ambos libros! ¡Te amo Emma!

Gracias a mis padres, que me han apoyado durante más o menos treinta y dos años. También gracias por comprar un billón de copias de los primeros libros para la familia.

Taylor Otwell, el camino con Laravel ha sido increíble. Gracias por darme la oportunidad de ser parte del equipo y por tu continuada amistad. Gracias por hacer un framework que es un placer usar, hace que nuestro código se lea como la poesía y por poner tanto tiempo y pasión en su desarrollo.

Gracias a todo el que haya comprado mis libros anteriores, Code Happy, y a toda la comunidad de Laravel. Sin vuestro soporte, nuevos títulos no hubieran sido posibles.

Errata

Este puede ser mi cuarto libro y mi escritura puede haber mejorado desde la última vez, pero te aseguro que habrá muchos, muchos errores. No tengo editora, equipo de revisión o una filología en Español. Hago lo mejor que puedo para ayudar a otros para aprender sobre Laravel. Te pido que tengas paciencia con mis errores. Puedes mejorar el libro enviando un correo con cualquier error que hayas encontrado a sombragriselros@gmail.com[1] junto con el título de la sección.

Los errores serán corregidos conforme vayan siendo descubiertos. Las correcciones serán lanzadas con ediciones futuras del libro.

[1] mailto:sombragriselros@gmail.com

Feedback

De la misma forma, puedes enviarme cualquier feedback que tengas sobre el contenido del libro o lo que quieras, enviando un correo a sombragriselros@gmail.com[2] o un tweet a @belelros[3]. Me esforzaré en responder a todo correo que reciba.

[2] mailto:sombragriselros@gmail.com

[3] https://twitter.com/belelros

Traducciones

Si quieres traducir Code Smart a tu idioma, por favor envíame un correo a me@daylerees.com[4] con tus intenciones. Ofreceré un 50/50 de los beneficios de la copia traducida, que tendrán el mismo precio que la copia en Inglés.

El libro está escrito en formato markdown.

[4] mailto:me@daylerees.com

Presentación

¡Hola! Encantado de conocerte.

Mi nombre es Dayle Rees y soy un programador. Acabo de empezar mi cuarta década en este planeta y te saludo desde la agradable tierra de Gales. Hace muchos años, fui uno de los primeros en usar un nuevo framework para el lenguaje de programación PHP llamado Laravel. Desde que leí la documentación por primera vez, supe que tenía que comenzar a crear aplicaciones con él. Era limpio, conciso y bonito. Vi mucho potencial en el framework, ¡y creo que no me equivoqué!

Desde entonces he sido un colaborador principal, he dado charlas, consultor y autor de varios libros basados en el framework de Laravel. Los libros han recibido unas reacciones asombrosas y tienen miles de fans en todo el mundo. ¡Ha sido una experiencia increíble! Si has leído alguno de mis libros antes, ¡te lo agradezco! Si no, bueno, déjame que te explique cómo funciona esto.

Como verás, no soy un autor tradicional. No me gusta complicar las cosas. No me gusta usar palabras pomposas para aparentar ser inteligente y escribo mis libros como si estuviéramos compartiendo una aventura de descubrimientos. Como he dicho antes, y lo diré de nuevo; me gusta escribir los libros como si fuéramos dos amigos sentados en un bar, compartiendo una bebida y conversación. ¡Al menos hasta que tú pagues una ronda!

Este es el tercer título en la serie de Laravel y cubre la versión 5.x del framework. He aprendido algunas cosas sobre escribir libros técnicos con cada uno de los libros lanzados, y por tanto, espero que encuentres agradable la experiencia de lectura de este libro. Si hay cualquier cosa que no te gusta del libro o un capítulo o concepto que no quede claro, ¡envíame un correo! Para que este libro sea tan perfecto como sea posible, voy a necesitar tu feedback. ¡Escribamos esta obra maestra juntos!

Probablemente estés deseando empezar, ¿no? Es un framework grande pero te prometo que si vamos paso a paso, estarás creando grandes aplicaciones antes de que te des cuenta. ¿Estás preparado? Adelante, pasa la página.

Instalación

Estoy seguro de que estás deseando comenzar con Laravel. Dado que es un framework para web, es importante crear un entorno con un servidor web. En un entorno de producción nos llevaría mucho rato el configurarlo y necesitaríamos una plataforma basada en Unix así que vamos a crear uno casero.

> ¡No hay tiempo para hacer cosas caseras!

¡Ja! Puede que tengas razón pero no es lo que quería decir exactamente. Quería decir que vamos a usar un entorno virtualizado usando una software llamado Vagrant. Nos permitirá comenzar a desarrollar en Laravel en un santiamén.

Instalar las dependencias de software

Antes de que podamos empezar, tenemos que instalar unas pocas dependencias. He aquí la lista completa.

- PHP http://www.php.net/[5]
- Git https://git-scm.com/[6]
- Composer https://getcomposer.org/[7]
- Virtualbox https://www.virtualbox.org/[8]
- Vagrant https://www.vagrantup.com/[9]

No voy a dar instrucciones específicas de instalación ya que cambian de semana en semana. En vez de eso, ve a las URLs arriba especificadas y encontrarás secciones en cada sitio sobre cómo descargar e instalar para tu sistema operativo.

Echemos un vistazo a lo que nos ofrece cada pieza de software.

Vamos a necesitar **PHP**. Es el lenguaje usado por el framework de Laravel. No hay manera de quitarnos esto de encima. Descarga la última versión que esté disponible.

[5] http://www.php.net/
[6] https://git-scm.com/
[7] https://getcomposer.org/
[8] https://www.virtualbox.org/
[9] https://www.vagrantup.com/

Usaremos **Git** para el control de versiones. También es una buena forma de descargarnos una copia base de Laravel. Merece la pena tenerlo dado que forma parte del día a día de un programador experimentado.

Usaremos **Composer** para gestionar todas las librerías que usemos con PHP. No dejes de instalarlo. ¡Es uno de mis programas favoritos!

Luego, tenemos **VirtualBox**. Es un programa que te permitirá virtualizar sistemas operativos. En esencia, te permite ejecutar ordenadores *de mentira* en tu equipo principal. Dado que el desarrollo web va mejor en entornos Unix, usaremos un entorno virtual de unix para poder desarrollar en cualquier plataforma.

Por último, tenemos **Vagrant**. Vagrant es usado para aprovisionar entornos virtuales y para configurarlos como queramos. Es un programa de línea de comandos así que lo usaremos en un terminal. Vagrant hará uso de Virtualbox para crear entornos de proyectos.

Una vez que tengamos las cinco piezas instaladas, podemos pasar a la siguiente sección.

Crea un proyecto de Laravel

Antes de poder crear nuestro entorno, necesitamos tener un proyecto de Laravel preparado.

Primero tenemos que decidir una ubicación para nuestros proyectos. Voy a ser creativo y voy a crear un directorio `Proyectos` en mi carpeta de usuario. Aquí es donde pondré todos los proyectos de Laravel.

Vamos a nuestra carpeta de proyectos, ¿no?

Ejemplo 01: Navega al directorio de proyectos.

```
1 cd ~/Proyectos
```

No ha sido difícil, ¿no? Ahora podemos usar **Composer** para crear un nuevo proyecto de Laravel. Vamos a escribir el siguiente comando y veamos lo que pasa. Puedes cambiar la palabra 'ejemplo' con el nombre de tu proyecto si es que prefieres algo diferente.

Ejemplo 02: Crea un proyecto de Laravel

```
1 composer create-project laravel/laravel ejemplo
```

Verás un montón de cosas en la pantalla. Composer se está descargando todas las librerías que Laravel necesita. Las llamaremos dependencias del paquete. Una vez terminado, habrás instalado Laravel. Ahora vamos a instalar Homestead.

Instala Homestead

Primero, naveguemos al directorio en el que hemos creado nuestro proyecto de Laravel.

Ejemplo 03: Navega al directorio del proyecto.

```
cd ~/Proyectos/ejemplo
```

Ahora, vamos a instalar el paquete Homestead para poder crear un entorno virtual. No te preocupes: es sencillo. Tan solo, escribe el siguiente comando:

Ejemplo 04: Añade homestead a las dependencias de composer.

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

Verás más cosas aparecer en pantalla. No te preocupes. ¡Es lo que queremos!

Ahora ejecutaremos el comando `make`, para configurar nuestro proyecto para homestead. He aquí el comando completo.

Ejemplo 05: Instalar homestead.

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

Bueno, ¡al menos este fue rápido!

Ahora nos queda el más largo. Vamos a usar el siguiente comando para levantar una máquina virtual. Si es la primera vez que ejecutas el comando, tardará un rato en descargarse todo lo necesario. Son unos 600mb o así. ¡Hazte un café o un te!

Ejemplo 06: Arrancando la máquina virtual.

```
vagrant up
```

Una vez que la máquina virtual esté operativa, hay una última cosa que tienes que hacer. Vamos a añadir el dominio de la aplicación de homestead a nuestro archivo de hosts para que podamos acceder a nuestra aplicación web en el navegador.

Usa tu editor de texto favorito con el archivo `/etc/hosts`. Vas a necesitar privilegios de admin/root para acceder al archivo. Necesitas añadir la siguiente línea.

Ejemplo 07: Añade una entrada local al registro DNS.

```
192.168.10.10  homestead.app
```

Para comprobar que todo funcione, visita `http://homestead.app` en tu navegador. Deberías ver el texto 'Laravel 5'. ¡Esta es nuestra aplicación de Laravel!

Aprendiendo Vagrant

Tu entorno de desarrollo web existe en una máquina virtual y esa máquina drena recursos de tu máquina. Usará la RAM disponible y tiempo de procesador.

De tiempo en tiempo puede resultar útil detener y restaurar el entorno para guardar recursos del sistema. Merece la pena destacar que cuando reiniciemos nuestro equipo, el entorno virtual **no se iniciará automáticamente**.

Por este motivo, vamos a ver algunos comandos básicos para mantener nuestros entornos Vagrant. Usaremos estos comenados en el directorio en que se encuentre el archivo `Vagrantfile`.

El primer comando es usado para iniciar la máquina virtual. Ya lo has usado. Aquí lo tienes.

Ejemplo 08: Arranca la máquina virtual.

```
1 vagrant up
```

El segundo comando es el opuesto. Se usa para detener la máquina virtual. Ya no usará recursos del sistema. Este es el comando:

Ejemplo 09: Para la máquina virtual.

```
1 vagrant halt
```

Si decides cambiar la configuración de tu máquina Vagrant, actualizando el archivo `Vagrantfile`, necesitas usar `provision` para aplicar los cambios a la máquina virtual.

Ejemplo 10: Actualiza los ajustes de la máquina virtual.

```
1 vagrant provision
```

En algunas circunstancias, puede que quieras ejecutar comandos en la máquina virtual. Se puede acceder fácilmente a la máquina usando el comando SSH de Vagrant. Esto te permitirá acceso sudo a la máquina. Por ejemplo.

Ejemplo 11: Accede por SSH a la máquina virtual.

```
1 vagrant ssh
```

Finalmente, si ya hemos terminado con nuestro proyecto y no queremos que la máquina virtual ocupe espacio en nuestro disco duro, podemos eliminarla. Usa el comando `destroy`.

Ejemplo 12: Destruye la máquina virtual.

```
vagrant destroy
```

Ya lo tenemos todo para comenzar a trabajar con Laravel. Parece que ha llegado la hora de pasar a algo nuevo. En el próximo episodio, vamos a echar un vistazo al ciclo de vida de una aplicación PHP moderna. Haz tu mejor pose de super-héroe, ¡y pasa de página!

Ciclo de vida

Si no has usado un framework de PHP antes, es probable que estés acostumbrado a tener un montón de archivos PHP diferentes en tu directorio web. La persona que use tu aplicación web, requerirá cada script de manera individual.

Laravel usa una combinación de controlador frontal y de router. Esto significa que solo hay un archivo PHP en tu raíz y todas las peticiones a tu aplicación serán realizadas a través de dicho script. Este archivo es llamado `index.php` y lo encontrarás en la carpeta `public`, que es el único directorio que debería ser accesible en la web.

> ¡No puedo hacer una aplicación con una página!

No te preocupes. Tenemos una solución para esto. Como verás, Laravel usa algunas técnicas diferentes para servir diferentes contenidos basados en el estado de la petición del navegador. De hecho, he aquí un sencillo diagrama para mostrar el ciclo de vida de una petición a una aplicación con el Framework de Laravel.

```
1  Petición > Servicios > Enrutado > Lógica > Respuesta
```

En cierto sentido, una petición a un servidor web es un bucle de entrada/salida. Siendo sinceros, hay unos pocos pasos más involucrados, pero no es el momento de hablar de ellos. Vamos a ir sobre cada una de las secciones del proceso, ¿te parece?

Petición

Cada petición realizada por un navegador web a tu aplicación tiene un amplio rango de información asociada a ella. Por ejemplo, la URL, el método HTTP, los datos de la petición, las cabeceras de la petición e información sobre la fuente de la petición.

Ahora le toca a Laravel y a tu aplicación interrogar la información dentro de la petición para decidir qué acción llevar a cabo. Usando Laravel, la información para la petición actual se guarda en una instancia de la clase `Illuminate\Http\Request`, extiende de la clase del framework de Symfony `Symfony\Component\HttpFoundation\Request`.

Symfony tiene una fantástica implementación del protocolo HTTP en su paquete HTTP Foundation. Laravel hace uso de este paquete para evitar re-inventar la rueda.

Ahora tenemos una petición y tenemos toda la información que necesitamos para nuestra aplicación para decidir una acción apropiada. Así que, ¿qué toca ahora? Echemos un vistazo.

Servicios

El siguiente paso del proceso es inicializar el framework. Laravel viene con un montón de servicios y características que hace que nuestras vidas como programadores web sean considerablemente más fáciles. Antes de que podamos hacer uso de esos servicios, necesitan ser inicializados.

El framework cargará todos los servicios definidos y configuración y se asegura de que tengan todo lo que necesitan para apoyar a nuestro código. Echaremos un vistazo a cómo se cargan los servicios en el capítulo de Proveedores de Servicios.

El framework está ahora listo para nuestro código, pero ¿qué pieza de código debería ejecutar? ¡Descubrámoslo!

Enrutado

Como ya hemos descubierto anteriormente, hay un único script que está disponible al usar Laravel. ¿Cómo mostramos páginas diferentes o realizamos acciones diferentes? He aquí donde el router hace su aparición.

El único propósito del router es hacer coincidir una petición con la pieza de código apropiada para ejecutar. Por ejemplo, el router sabrá que debería ejecutar el código para mostrar el perfil de un usuario cuando la petición incluya un verbo HTTP de GET y una URI de `/usuario/perfil`.

Laravel tiene una bonita forma de definir estas rutas y las veremos enseguida . Por ahora, echemos un vistazo a lo que ocurre después.

Lógica

Ahora, tenemos nuestro segmento de lógica. Esta sección queda mejor descrita como *tu* código. Es donde estarás hablando con la base de datos, validando formularios o mostrando páginas.

En Laravel, tenemos diferentes formas de definir tu lógica pero lo veremos en un capítulo posterior. Por ahora, vamos a ver la última sección, ¿te parece? Sé que los usuarios de nuestra aplicación tendrán muchas ganas de esta.

Respuesta

La respuesta es creada al final de nuestra lógica. Puede ser una plantilla HTML. Puede ser datos JSON; puede ser una cadena. O puede no ser nada. En algunas tristes circunstancias, podría ser una pantalla de error, como una página 404. Es bueno tener opciones, ¿no te parece?

La respuesta es lo que los usuarios de tu aplicación verán. Es la parte que están esperando.

Resumamos.

El navegador manda una petición. Laravel inicializa sus servicios, interroga la petición y lleva a cabo operaciones de enrutado. Nuestro código es ejecutado y entrega una respuesta al usuario. Es una operación sencilla, ¿no?

Suena a proceso realmente simple, pero te prometo que mantener estos *flujos* en tu mente, te harán un mejor desarrollador web.

Subamos el volumen y mantengamos la fiesta en al próximo capítulo. Vamos a ver el patrón de arquitectura de software MVC (Modelo Vista Controlador). Es usado por un montón de aplicaciones de Laravel, ¡así que supongo que merece la pena aprender sobre ello!.

Configuración

La configuración es una parte importante de cualquier aplicación. No siempre querremos que nuestra aplicación tenga los mismos ajustes en cada entorno. Si decidimos entregar la aplicación a otros usuarios, puede que quieran configurarla por ellos mismos. ¿Qué pasa si usan un servidor de base de datos diferente al nuestro?

Echemos un vistazo a cómo Laravel nos permite especificar nuestros propios valores de configuración.

Archivos de configuración

Comenzaremos echando un vistazo al directorio `config` en la raíz de nuestro proyecto:

Ejemplo 01: Archivos de configuración

```
app.php
auth.php
broadcasting.php
cache.php
compile.php
database.php
filesystems.php
mail.php
queue.php
services.php
session.php
view.php
```

Verás un porrón de archivos PHP. Muchos de ellos tienen nombres de componentes del framework. Son los archivos de configuración por defecto que vienen en una instalación nueva de Laravel. Escoge un archivo. Cualquiera. Escoge tu favorito y ábrelo en tu editor.

¿Qué hay dentro? ¡Matrices de PHP! Como puedes ver, Laravel usa matrices PHP para su configuración. Las claves representan el nombre de la configuración y los valores son, por supuesto, los valores en sí. Con Laravel, podemos acceder a esos ajustes usando la fachada `config()`. He aquí un ejemplo:

Ejemplo 02: Accediendo a los valores de configuración

```
// Usando una fachada
$debug = Config::get('app.debug', false);

// Usando una función de ayuda
$debug = config('app.debug', false);
```

La firma de ambas técnicas es idéntica. El primer parámetro es el nombre de la clave que queremos obtener. La clave está compuesta por el nombre del archivo de configuración que nos gustaría revisar, seguida por un punto, y luego la clave de la matriz del ajusto, usando una notación de *punto*. Aquí estamos solicitando el valor que tiene por clave `debug` en el archivo `app.php`.

No te olvides de añadir `use Config;` en lo alto de la clase si tiene espacio de nombre. No te has olvidado ya del capítulo sobre espacios de nombre, ¿no?

El segundo parámetro de ambas funciones es el valor por defecto. En caso de que el valor falte, obtendremos el valor por defecto.

En un capítulo posterior, aprenderemos cómo inyectar el componente de configuración en nuestras clases para evitar tener que depender de funciones globales ni fachadas.

Si quieres crear tus propios archivos de configuración, ponlos en el directorio de `config`. Laravel los cargará de manera automática y te permitirá usarlos de la misma forma que sus propios archivos de configuración. A la hora de escribir aplicaciones para otros, asegúrate de añadir tanta configuración como te sea posible. Es genial cuando puedes hacer que una aplicación funcione *a tu manera*.

Variables de entorno

Si has estado bicheando en los archivos de configuración, puede que hayas descubierto una pequeña función llamada `env()`. La función `env()` es una nueva forma de tener diferentes configuraciones disponibles en diferentes entornos. Por ejemplo, no querrás que tu aplicación de desarrollo use la misma base de datos que la aplicación final.

Desde su versión 5, Laravel no usa subcarpetas basadas en el entorno para la configuración. En vez de esto, hace uso de la librería DotEnv para permitir valores de configuración que dependan de variables de entorno. Establecer variables de entorno puede ser un poco farragoso. Por ejemplo, no querremos estar haciendo esto todo el rato:

Ejemplo 03: Establece una variable de entorno

```
export DB_HOST="127.0.0.1"
```

Confía en mi; es incluso peor en Windows. Las variables de entornos son el método de configuración preferido por administradores de sistemas ya que es sencillo de transferir a nuevos servidores web. No obstante, si estamos trabajando con valores de configuración a diario, probablemente no queramos exportarlos cada vez.

En vez de ello, tenemos un archivo `.env` en la raíz de nuestro proyecto. Echémosle un vistazo:

Ejemplo 04: Nuestro archivo .env.

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=base64:a0hRMNEdGdWW6EUVCn3vIu8VFQiJc113CMciFkJ+pcw=
APP_URL=http://localhost

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

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

El archivo `.env` es muy simple. Claves en la izquierda (normalmente en mayúsculas) y valores en la derecha. Aquí vemos un montón de valores por defecto que son usados por el Framework. Puedes

ver las bases de datos, correos y ajustes del servidor de cache. Si queremos que nuestros valores de configuración estén basados en nuestro entorno, lo mejor es añadirlos a este fichero.

La función `env()` puede ser usada en nuestra configuración PHP para obtener valores de nuestro archivo `.env`. Esto convierte a nuestra capa de configuración en un proceso de dos pasos pero nos permite usar diferentes `.env` en múltiples entornos. La función `env()` acepta los mismos parámetros que la función `config()`. He aquí un ejemplo:

Ejemplo 05: La función env()

```
1 $host = env('DB_HOST', '127.0.0.1')
```

Aquí estamos solicitando un ajuste de la base de datos, pero ofreciendo un valor por defecto para cuando no esté presente en nuestro archivo `.env` o variables de entorno. Merece la pena destacar que las variables de entorno tienen prioridad sobre cualquier valor en el archivo `.env`.

> ¿Entonces porqué no usar `env()` simplemente en vez de usar `config()`?

¡Gran pregunta y demuestra que estás escuchando con atención! Si quisieras, podrías usar `env()` para todo, no obstante, de esta forma no puedes hacer uso de la funcionalidad de cache que tiene la capa de configuración de Laravel. Como verás, Laravel puede almacenar los ficheros en la cache para hacer que el framework sea más rápido. No obstante, no puede cachear valores de variables de entorno, por lo que si decides usar `env()` fuera de los archivos de configuración, puede que acabes teniendo problemas de cache. No queremos eso, ¿verdad?

> Supongo que no.

¡Genial! La regla de oro a usar es que `env()` solo debería ser vista en el directorio `config`. No es difícil, ¿no?

Si un paquete de terceros viene con sus propios archivos de configuración, simplemente usa `php artisan vendor:publish` para copiarlos a tu directorio de configuración.

Caché de configuración

Como mencionamos antes, Laravel puede cachear nuestros valores de configuración para permitir que carguen mucho más rápido. Es una característica que es mejor usarla en un entorno de producción.

Para hacer uso de la cache en la configuración tenemos que ir a la raíz del proyecto (esto es en `/vagrant/` en tu máquina Homestead) y ejecutar un comando de Artisan.

```
php artisan config:cache
```

Nuestra configuración ahora será almacenada en la cache y los archivos de configuración dejarán de ser usados. Si añadiéramos nueva configuración y queremos limpiar la cache, solo tenemos que ejecutar el comando `config:clear`.

```
php artisan config:clear
```

Probablemente no necesites esto para tu desarrollo en local y no mientras aprendes a usar el framework, pero es bueno saber que está ahí si lo necesitamos, ¿no?

En el próximo capítulo comenzaremos a escribir algo de código. ¿Estás emocionado? Pasa la página para aprender sobre el enrutado.

Enrutado básico

Echemos un ojo a la petición que se hace al framework de Laravel.

Ejemplo 01: A URL

```
http://homestead.app/mi/pagina
```

En este ejemplo, estamos usando el protocolo HTTP (usado por la mayoría de los navegadores web) para acceder a tu aplicación Laravel alojada en `homestead.app`. La porción `mi/pagina` de la URL es lo que usaremos para enrutar las peticiones web a la lógica apropiada.

Iré más allá y te mostraré el camino. Las rutas son definidas en el archivo `app/Http/routes.php`, así que vamos allá y creemos una ruta que escuche la petición que hemos mencionado arriba.

Definiendo rutas

Ejemplo 02: Nuestra primera ruta.

```php
<?php

// app/Http/routes.php

Route::get('mi/pagina', function () {
    return '¡Hola mundo!';
});
```

Ahora, introduce `http://homestead.app/mi/pagina` en tu navegador web, cambiando `homestead.app` with por la dirección de tu aplicación Laravel.

Si todo ha sido configurado correctamente, ¡verás ahora las palabras `¡Hola mundo!` con Times New Roman! Porqué no lo echamos un ojo más atentamente a la declaración de la ruta para ver cómo funciona.

Las rutas están siempre declaradas usando la clase `Route`. Eso es lo que tenemos al principio, antes de `::`. La parte `get` es el método que usamos para 'capturar' las peticiones que son realizadas usando el verbo 'GET' de HTTP hacia una URL concreta.

Como verás, todas las peticiones realizadas por un navegador web contienen un verbo. La mayoría de las veces, el verbo será `GET`, que es usado para solicitar una página web. Se envía una petición `GET` cada vez que escribes una nueva dirección web en tu navegador.

Aunque no es la única petición. También está `POST`, que es usada para hacer una petición y ofrecer algunos datos. Normalmente se usa para enviar un formulario en la que se necesita enviar los datos sin mostrarlo en la URL.

Hay otros verbos HTTP disponibles. He aquí algunos de los métodos que la clase de enrutado tiene disponible para ti:

Ejemplo 03: Métodos de enrutado

```php
<?php

// app/Http/routes.php

Route::get();
Route::post();
Route::put();
Route::delete();
Route::any();
```

Todos esos métodos aceptan los mismo parámetros, por lo que puedes usar cualquier método HTTP que sea apropiado para la situación. Esto es conocido como `enrutado REST`. Hablaremos sobre esto con más detalle luego. Por ahora, todo lo que tienes que saber es que se usa `GET` para hacer peticiones, y `POST` cuando tienes que mandar datos adicionales con la petición.

El método `Route::any()` es usado para hacerlo coincidir con cualquier verbo HTTP. No obstante, te recomendaría que usaras el verbo correcto para la situación en la que estás para que la aplicación sea más transparente.

Volvamos al ejemplo. He aquí para refrescar tu memoria:

Ejemplo 04: Nuestra primera ruta, de nuevo

```php
<?php

// app/Http/routes.php

Route::get('mi/pagina', function () {
    return '¡Hola mundo!';
});
```

La siguiente porción del código es el primer parámetro del método `get()` (o cualquier otro verbo). Este parámetro define la URI con la que quieres hacer coincidir la URL. En este caso estamos haciendo coincidir `mi/pagina`.

El parámetro final es usado para ofrecer lógica para gestionar la petición. Aquí estamos usando una `Closure`, que también es conocida como una `función anónima`. Las closures son simplemente funciones sin nombre que pueden ser asignadas a variables, como lo haríamos con cualquier valor.

Por ejemplo, el fragmento de arriba podría también ser escrito así:

Ejemplo 05: Closures separadas

```php
<?php

// app/Http/routes.php

$logica = function () {
    return '¡Hola mundo!';
};

Route::get('mi/pagina', $logica);
```

Aquí estamos guardando la `Closure` en la variable `$logic` y luego pasándosela al método `Route::get()`.

En esta ocasión, Laravel ejecutará la Closure solo cuando la petición actual esté usando el verbo `GET` de HTTP y la URI coincida con `mi/pagina`. Bajo esas condiciones, la sentencia `return` será procesada y se le pasará la cadena "¡Hola mundo!" al navegador.

Puedes definir tantas rutas como quieras. Por ejemplo:

Ejemplo 06: Múltiples rutas

```php
<?php

// app/Http/routes.php

Route::get('primera/pagina', function () {
    return '¡Primera!';
});

Route::get('segunda/pagina', function () {
    return '¡Segunda!';
});

Route::get('tercera/pagina', function () {
    return '¡Patata!';
});
```

Intenta navegar a las siguientes URLs para ver cómo se comporta nuestra aplicación.

Ejemplo 07: Múltiples URLs

```
http://homestead.app/primera/pagina
http://homestead.app/segunda/pagina
http://homestead.app/tercera/pagina
```

Seguramente quieras asociar la raíz de tu aplicación web. Por ejemplo…

Ejemplo 08: Sin ruta

```
http://homestead.app
```

Normalmente, esta será usada para la página de inicio de tu aplicación. Creemos una ruta que coincida con esto.

Ejemplo 09: Route with no path.

```
<?php

// app/Http/routes.php

Route::get('/', function () {
```

return ‘En la Rusia Soviética, la función te define a ti.’; });

> Ey,¡espera un segundo! ¿¡No tenemos que poner una barra al final de la URI?!

¡Buen ojo! Aunque no tienes que preocuparte por ello.

Como ves, una ruta que contiene únicamente una barra invertida, coincidirá con la URL del sitio web, tenga o no tenga una barra al final. La ruta de arriba responderá a cualquiera de estas URLs.

Ejemplo 10: Con o sin barra

```
http://homestead.app
http://homestead.app/
```

Las URLs pueden tener tantos segmentos (partes entre las barras) como quieras. Puedes usarlo para construir una jerarquía en el sitio.

Considera la siguiente estructura:

Ejemplo 11: Web imaginaria

```
/
/libros
    /ficcion
    /ciencia
    /romance
/revistas
    /celebridades
    /tecnologia
```

Vale, es un sitio bastante sencillo pero es un gran ejemplo de estructura que a menudo encontrarás en la web. Vamos a recrearlo con rutas de Laravel.

Por claridad, he eliminado el contenido de cada Closure.

Ejemplo 12: Rutas de nuestro sitio imaginario

```php
<?php

// app/Http/routes.php

// home page
Route::get('/', function () {});


// routes for the books section
Route::get('/libros', function () {});
Route::get('/libros/ficcion', function () {});
Route::get('/libros/ciencia', function () {});
Route::get('/libros/romance', function () {});

// routes for the magazines section
Route::get('/revistas', function () {});
Route::get('/revistas/celebridades', function () {});
Route::get('/revistas/tecnologia', function () {});
```

Con esta colección de rutas, hemos creado fácilmente una jerarquía del sitio. Puede que que te hayas dado cuenta de que hay cierta repetición. Vamos a buscar una forma de minimizar esta repetición y así, dejar de repetirnos.

Parámetros de las rutas

Los parámetros de las rutas pueden ser utilizados para introducir valores de relleno en tus definiciones de ruta. Esto creará un patrón sobre el cual podamos recoger segmentos de la URI y pasarlos al gestor de la lógica de la aplicación.

Esto puede sonar un poco confuso, pero cuando lo veas en acción todo tendrá sentido. Allá vamos.

Ejemplo 13: Parámetros de ruta

```
<?php

// app/Http/routes.php

// routes for the books section
Route::get('/libros', function () {
    return 'Índice de libros.';
});

Route::get('/libros/{genero}', function ($genero) {
    return "Libros en la categoría {$genero}.";
});
```

En este ejemplo, hemos eliminado la necesidad de tener todas las rutas por género, incluyendo una variable en la ruta. La variable `{genero}` sacará todo lo que esté detrás de la URI `/libros/`. Esto pasará su valor al parámetro `$genero` de la Closure, que nos permitirá usar la información en nuestra parte de lógica.

Por ejemplo, si quisieras visitar la siguiente URL:

Ejemplo 14: Parámetro en la URL

```
http://homestead.app/libros/crimen
```

Serías recibido con esta respuesta de texto:

Ejemplo 15: Output.

```
Libros en la categoría crimen.
```

Podríamos eliminar también el requisito de ese parámetro usando uno opcional. Un parámetro puede ser convertido en opcional añadiendo un signo de interrogación (?) al final de su nombre. Por ejemplo:

Ejemplo 16: Parámetros opcionales

```php
<?php

// app/Http/routes.php

// Rutas para la sección libros
Route::get('/libros/{genero?}', function ($genero = null) {
    if ($genero == null) {
         return 'Índice de libros.';
    }
    return "Libros en la categoría {$genero}.";
});
```

Si no se facilita un género en la URL, el valor de `$genero` será igual a `null` y se mostrará el mensaje 'Índice de libros.'.

Si no queremos que el valor del parámetro de una ruta sea `null` por defecto, podemos especificar una alternativa usando una asignación. Por ejemplo:

Ejemplo 17: Valores de parámetros por defecto

```php
<?php

// app/Http/routes.php

// Rutas para la sección libros
Route::get('/libros/{genero?}', function ($genero = 'Crimen') {
    return "Libros en la categoría {$genero}.";
});
```

Ahora, si visitamos la siguiente URL:

Ejemplo 18: URL a la que le faltan parámetros

```
http://homestead.app/libros
```

Recibiremos esta respuesta:

Ejemplo 19: Salida.

```
Libros en la categoría Crimen.
```

Espero que estés empezando a ver cómo se usan las rutas para dirigir tus peticiones en tu sitio y que son un 'pegamento' usado para mantener tu aplicación unida.

Hay mucho más sobre rutas. Antes de que volvamos sobre ellas, vamos a cubrir más sobre lo básico. En el próximo capítulo, echaremos un ojo a los tipos de respuesta que Laravel tiene que ofrecer.