Laravel

Introducción a Laravel

¿Qué es Laravel?

Laravel es un framework de código abierto diseñado para el desarrollo de aplicaciones y servicios web utilizando el lenguaje de programación PHP. Creado por Taylor Otwell en 2011, Laravel se ha convertido en una herramienta popular entre los desarrolladores web debido a su elegante sintaxis, su arquitectura MVC (Modelo-Vista-Controlador), y una amplia gama de características que facilitan el desarrollo eficiente y la creación de aplicaciones web robustas.

Sus características principales son:

- **Arquitectura MVC:** Adopta el patrón de diseño Modelo-Vista-Controlador para una estructura organizada del código.
- **Sistema de Empaquetado Modular:** Facilita la gestión de dependencias y la reutilización de código.
- **Artisan CLI:** Ofrece una interfaz de línea de comandos (CLI) potente para la automatización de tareas.
- **Eloquent ORM:** Permite interactuar con la base de datos de manera intuitiva mediante objetos y relaciones.
- **Autenticación y Autorización:** Incorpora sistemas de autenticación y autorización para gestionar la seguridad.

Laravel ha evolucionado continuamente, manteniéndose actualizado con las mejores prácticas de desarrollo web y proporcionando a los desarrolladores una herramienta eficiente y elegante para construir aplicaciones web modernas.

Instalación

1. Instalar XAMPP

XAMPP es una solución de servidor web gratuita y de código abierto que incluye PHP, MySQL y Apache. Para instalar XAMPP en Windows, sigue estos pasos:

Visita la página de descargas de XAMPP y descarga el instalador adecuado para tu versión de Windows.

Ejecuta el instalador y sigue las instrucciones en pantalla. Durante el proceso de instalación, puedes elegir qué componentes instalar. Asegúrate de seleccionar PHP y MySQL.

Una vez instalados, ejecuta el **Panel de Control de XAMPP** e inicia los servicios Apache y MySQL.

2. Instalar Composer

Composer es una herramienta de gestión de dependencias para PHP que es necesaria para instalar Laravel. Para instalar Composer en Windows, sigue estos pasos:

Visita la página de descarga de Composer y descarga el archivo **Composer-Setup.exe**.

Ejecuta el archivo **Composer-Setup.exe** y sigue las instrucciones que aparecen en pantalla.
También se te pedirá que selecciones el modo de instalación. Asegúrate de elegir **Instalar para todos los usuarios (recomendado)**.

Asegúrate de seleccionar el ejecutable PHP correcto durante el proceso de instalación (normalmente se encuentra en la carpeta de instalación de XAMPP en xampp/php/php.exe).
Haz clic en **Siguiente** para seguir las instrucciones en pantalla y luego en **Instalar**.

Una vez completada la instalación, haz clic en **Finalizar**.

3. Verificar la Instalación de Composer

Para verificar que Composer se ha instalado correctamente, abre el símbolo del sistema y ejecuta el siguiente comando:

composer —version

Si la instalación se ha realizado correctamente, deberías ver la versión de Composer mostrada.

4. Instalar Laravel Utilizando Composer

Puedes utilizar Composer, que ya está instalado, para instalar Laravel globalmente en tu sistema. Para ello, abre el símbolo del sistema y ejecuta el comando indicado:

composer create-project laravel/laravel app-name

Esto descargará automáticamente todos los archivos relevantes de Laravel para crear un nuevo proyecto.

5. Verificar la Instalación de Laravel

Para verificar que Laravel se ha instalado correctamente, abre el símbolo del sistema y ejecuta el siguiente comando:

laravel —version

Tras una instalación correcta, podrás ver la versión de Laravel.

6. Iniciar el Servidor

Con tu nuevo proyecto de aplicación creado, necesitarás iniciar un servidor. Para ello, escribe lo siguiente:

cd app-name

php artisan serve

7. Ejecuta el Proyecto en Tu Navegador

Una vez iniciado el servidor, deberías poder acceder a tu proyecto de aplicación a través de tu navegador web. Para ello, abre tu navegador y ve a la siguiente dirección: https://localhost:8000
Con esto, ya puedes empezar a desarrollar aplicaciones web utilizando Laravel en tu máquina Windows.

Estructura de directorios y flujo de solicitud

Estructura de directorios en Laravel

Laravel sigue una estructura de directorios bien definida para organizar y separar distintos aspectos de una aplicación web. Aquí se presenta una descripción general de algunos de los directorios clave en Laravel:

app: Contiene el código principal de la aplicación, incluyendo los controladores, modelos y otros elementos esenciales. es el directorio al que apunta el servidor cuando sirve el sitio web. Este contiene index.php, que es el controlador frontal que inicia el proceso de arranque
y enruta todas las solicitudes de manera adecuada. También es donde se almacenan los archivos públicos, como imágenes, hojas de estilo, scripts o descargas.

bootstrap: Incluye archivos necesarios para inicializar la aplicación y cargar el framework Laravel.

config: Contiene archivos de configuración para diversos componentes de la aplicación.

database: Aquí se encuentran las migraciones y las semillas de la base de datos.

public: Almacena los archivos accesibles públicamente, como imágenes, hojas de estilo y scripts.

resources: Contiene las vistas, así como los archivos de activos no compilados como LESS o SASS. es donde se encuentran los archivos que no son PHP y que son necesarios para otros scripts.

routes: Incluye archivos que definen las rutas de la aplicación.

Estos son solo algunos de los directorios clave; Laravel proporciona una estructura bien definida para facilitar el desarrollo ordenado de aplicaciones web.

El ciclo de vida de una solicitud en Laravel

El ciclo de vida de una solicitud en Laravel consta de varios pasos esenciales que demuestran cómo se gestiona una solicitud HTTP en una aplicación Laravel. Aquí está una explicación más detallada de cada paso:

- Inicio de la Solicitud: La solicitud HTTP llega al servidor y es gestionada por el servidor web.
- Kernel de Laravel: Laravel intercepta la solicitud a través del kernel, el cual sirve como el punto de entrada central.
- Middleware: Los middleware se ejecutan. Estos son filtros que permiten realizar acciones antes y después de que la solicitud llegue al controlador.
- Manejo de Rutas: Laravel determina la ruta correspondiente a la URL de la solicitud. Las rutas definen cómo se debe manejar la solicitud.
- Controlador: Si la ruta tiene un controlador asociado, se ejecuta. El controlador contiene la lógica para procesar la solicitud.
- Vistas y Respuestas: Se pueden cargar vistas y devolver respuestas, permitiendo la presentación de datos.
- Envío de Respuesta: La respuesta generada se envía de vuelta al cliente.

- Fin de la Solicitud: La solicitud completa su ciclo y el proceso finaliza.

Este ciclo proporciona una estructura organizada y modular para el manejo de solicitudes HTTP en Laravel, permitiendo un flujo controlado y extensible.

El patrón MVC en Laravel

El patrón de diseño Modelo-Vista-Controlador (MVC) en Laravel es esencial para la estructuración y organización eficiente de aplicaciones web. Aquí se presenta una breve descripción de cada componente en el contexto de Laravel:

- **Modelo (Model):**
  - Representa la capa de datos y la lógica de negocio.
  - En Laravel, los modelos son clases PHP que interactúan con la base de datos, facilitando operaciones como la lectura, escritura y actualización de datos.

- **Vista (View):**
  - Maneja la presentación y la interfaz de usuario.
  - En Laravel, las vistas son archivos que contienen código HTML y pueden integrarse con el motor de plantillas Blade para una gestión eficiente de la presentación.

- **Controlador (Controller):**
  - Actúa como intermediario entre el modelo y la vista.
  - En Laravel, los controladores son clases que gestionan la lógica de la aplicación, reciben las interacciones del usuario y actualizan el modelo o la vista según sea necesario.

Ventajas del MVC en Laravel:

- **Separación de Responsabilidades:** Cada componente tiene un propósito claro.
- **Facilita el Mantenimiento:** Cambios en un componente no afectan a los demás.
- **Reutilización de Código:** Los componentes son modulares y pueden reutilizarse en diferentes partes de la aplicación.

Routing en Laravel

Definición de rutas

En una aplicación Laravel, definirás tus rutas 'web' en routes/web.php y tus rutas 'API' en routes/api.php. Las rutas 'web' son aquellas que serán visitadas por tus usuarios finales; las rutas 'API' son para tu API, si tienes una. Por ahora, nos enfocaremos principalmente en las rutas en routes/web.php.

```
// routes/web.php
Route::get('/', function () {
return 'Hello, World!';
});
```

Tal y como hemos definido esta ruta, si visitas / (la raiz de tu dominio), el enrutador de Laravel debería ejecutar la clausura definida allí y devolver el resultado. Ten en cuenta que retornamos nuestro contenido y no lo emitimos ni imprimimos.

Una ruta en Laravel es un mecanismo que define cómo la aplicación web responde a las solicitudes entrantes. En el contexto de Laravel, una ruta es la URL que un usuario puede visitar en su navegador. Laravel utiliza un sistema de enrutamiento que permite asociar rutas específicas con funciones o controladores que manejan la lógica de la aplicación. Este enfoque sigue el

principio de "Convención sobre Configuración" (Convention over Configuration) para simplificar el desarrollo.

En el marco de Laravel, las rutas se definen en archivos específicos, como web.php o api.php, ubicados en el directorio routes. Estos archivos contienen expresiones que indican cómo manejar las solicitudes HTTP para rutas específicas. Pueden incluir parámetros dinámicos, restricciones y otros atributos que personalizan el comportamiento de la ruta.

Por ejemplo, una ruta puede definirse para manejar solicitudes HTTP GET en la URL "/usuarios" y asociarse con un controlador que recupera y muestra información sobre usuarios. Laravel también admite la agrupación de rutas para organizarlas y aplicar middleware, lo que agrega capas adicionales de funcionalidad, como la autenticación o la autorización.

Muchos sitios web simples podrían ser deifinidos completamente dentro del archivo web.php de la carpeta routes. Con unas simples rutas GET con algunas plantillas, puedes servir una web clásica de forma muy sencilla.

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

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

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

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

Verbos de ruta

Quizás hayas notado que hemos estado usando Route::get en nuestras definiciones de ruta. Este significa que le estamos diciendo a Laravel que solo coincida con estas rutas cuando la solicitud HTTP utiliza la acción GET. Pero, ¿qué pasa si es un formulario POST, o tal vez algún envío de JavaScript? ¿PONER o ELIMINAR solicitudes? Hay algunas otras opciones para métodos para llamar en una ruta. definición, como se ilustra en este ejemplo:

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

Route::post('/', function () {});

Route::put('/', function () {});

Route::delete('/', function () {});

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

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

Manejo de rutas

Como probablemente habrás adivinado, pasar un cierre a la definición de ruta no es el único manera de enseñarle cómo resolver una ruta. Los cierres son rápidos y sencillos, pero cuanto más grandes y más complicada sea tu aplicación, más complicado será poner toda su lógica de enrutamiento en un solo archivo. Además, las aplicaciones que utilizan cierres de rutas no pueden aprovechar las ventajas de Laravel.

La otra opción común es pasar un nombre de controlador y un método como una cadena en lugar del cierre, como en esrte ejemplo:

Route::get('/', 'WelcomeController@index');

Esto le dice a Laravel que pase solicitudes a esa ruta al método index() de la aplicación. \Http\Controllers\WelcomeController controlador. Este método será pasado el mismos parámetros y tratado de la misma manera que un cierre que, alternativamente, podría haber puesto en su lugar.

```
// Ejemplo de definición de ruta básica en web.php
Route::get('/inicio', 'HomeController@index');
```

En este ejemplo, la ruta "/inicio" se asocia al método index del controlador HomeController cuando se recibe una solicitud GET.

Parámetros de ruta

Si la ruta que estás definiendo tiene parámetros (segmentos de la estructura de la URL que son variables), es muy sencillo definirlos dentro de tu ruta y pasarlos al cierre.

```
Route::get('users/{id}/friends', function ($id) {
        //
});
```

En este caso, la ruta captura el valor del parámetro {id} y lo pasa al método show del controlador UserController:

```
// Ejemplo de ruta con parámetro en web.php
Route::get('/usuario/{id}', 'UserController@show');
```

Nombres de rutas

La forma más sencilla de refereisre a estas rutas en cualquier lugar de tu aplicación es usando su path. Pero hay un Helper url() que nos ayuda a simplificar el linkado en tus vistas, si lo necesita. El helper va a añadir el prefijo de tu rura con el dominio completo de tu web. Veamos este ejemplo:

```
<a href="<?php echo url('/'); ?>">

// outputs <a href="http://myapp.com/">
```

De cualquier manera, Laravel te permite llamar a cada ruta, lo que te permite referirte a ella sin referenciar explícirtamente la URL. Esto es de mucha ayuda porque significa que puedes dar nombres simples a rutas complejas, y también porque enlazar por el nombre implica no tener que reescribir los enlaces de tu frontend si los paths cambian.

```
// Definiendo la ruta con nombre en routes/web.php:
Route::get('members/{id}', 'MembersController@show')->name('members.show');
```

```
// Enlazando la ruta en una vista usando el helper route()
<a href="<?php echo route('members.show', ['id' => 14]); ?>”>
```

Este ejemplo ilustra un par de nuevos conceptos. Primero, que estamos usando la definición de la ruta para añadir el nombre, encadenando el método name después del método get. Esta forma de aplica el código nor permite nombraar la ruta, dando un alias corto que sea más fácil de referenciar el cualquier parte.

```
// Ejemplo de ruta con nombre en web.php
Route::get('/dashboard', 'DashboardController@index')->name('dashboard');
```

Ahora, se puede generar la URL utilizando el nombre de la ruta: route('dashboard').

También presentamos el asistente route(). Al igual que url(), está pensado para ser utilizado en vistas para simplificar el enlace a una ruta con nombre. Si la ruta no tiene parámetros, puedes simplemente pase el nombre de la ruta: (route('members.index')) y reciba una cadena de ruta http://myapp.com/members/index).
.
En general, recomiendo usar nombres de rutas en lugar de rutas para hacer referencia a sus rutas. Y por lo tanto usar el asistente route() en lugar del asistente url(). A veces puede volverse un poco torpe, por ejemplo, si está trabajando con múltiples subdominios, pero proporciona un increíble nivel de flexibilidad para cambiar posteriormente el enrutamiento de la aplicación sin mayor penalización.

Cuando en tu ruta tienes parámetros (por ejemplo, users/{id}), necesitas definir esos parámetros cuando estés usando el helper route() para generar un link a la ruta.

Hay unas cuantas formas distintas de pasar esos parámetros. Imaginemos que tenemos una ruta definida como users/{userId}/comments/{commentId}. Si la id del usuario es 1 y la id del comentario es 2, tenemos distintas manertas de usarlos:

Opción 1:
```
route('users.comments.show', [1, 2])
// http://myapp.com/users/1/comments/2
```

Opción 2:
```
route('users.comments.show', ['userId' => 1, 'commentId' => 2])
// http://myapp.com/users/1/comments/2
```

Opción 3:
```
route('users.comments.show', ['commentId' => 2, 'userId' => 1])
// http://myapp.com/users/1/comments/2
```

Opción 4:
```
route('users.comments.show', ['userId' => 1, 'commentId' => 2, 'opt' => 'a'])
// http://myapp.com/users/1/comments/2?opt=a
```

Como puedes ver, los valores del array son asignados en orden. En el caso de usar un array asociativo, los valores deben casar con los parámetros emparejando sus claves, y cualquier cosa sobrante es añadido como un parámetro de una consulta.

Grupos de rutas

A menudo, un grupo de rutas comparten una característica particular (un cierto requerimiento de autenticación, un prefijo en el path, etc). Definiendo estas características compartidas una y otra vez en cada ruta no solo parece tedioso sino que también puede enfangar a las rutas y oscurecer la estructura de tu aplicación.

Los grupos de rutas permiten agrupar diferentes rutas juntas, y aplicar unas opciones de configuración compartidas para todo el grupo, reduciendo la duplicación. Adicionalmente, los grupos de rutas dan pistas visuales para futuros desarrolladores de que esas rutas están agrupadas juntas.

Para agrupar dos o mas rutas jungtas, rodeas las definiciones de las rutas con un grupo de ruta, como veremos en el siguiente ejemplo. En realidad, estas pasando un cierra a la definición de grupo, y defines las rutas agrupadas dentro de ese cierre.

```
Route::group([], function () {
        Route::get('hello', function () {
                return 'Hello';
        });
        Route::get('world', function () {
                return 'World';
        });
});
```

El array vacio que es el primer parámetro, permite pasar una variedad de opciones de configuración que se aplicará a todo el grupo.

```
// Ejemplo de agrupación de rutas en web.php
Route::middleware(['auth'])->group(function () {
        Route::get('/perfil', ‘ProfileController@show');
        Route::get('/ajustes', 'SettingsController@index');
});
```

En este ejemplo, las rutas dentro del grupo requieren autenticación, lo que significa que solo los usuarios autenticados pueden acceder a "/perfil" y “/ajustes".

Middleware

Middleware en Laravel es un componente esencial que actúa como un filtro intermedio entre las solicitudes HTTP entrantes y las aplicaciones. Funciona como una capa de procesamiento que puede realizar tareas específicas antes o después de que la solicitud alcance el controlador. Los middleware proporcionan un mecanismo flexible para manipular las solicitudes HTTP y permiten ejecutar código en diversas etapas del ciclo de vida de una solicitud.

Probablemente el uso más común para los grupos de rutas es para alplicar un middleware a un grupo de rutas. Aprenderemos más sobre el middleware más adelante, pero, entre otras cosas, es la manera que Laravel usa para la autenticación de usuarios y la restricción para usuarios invitados relativa al uso de partes de tu site.

En este ejemplo, vamos a crear un grupo de rutas alrededor de las vistas dashboard y account y aplicando el middleware auth a ambosa. En este ejemplo significa que los usuarios tienen que logarse en la aplicación para ver las páginas de dashboard o account

```
Route::group(['middleware' => 'auth'], function () {
        Route::get('dashboard', function () {
                return view('dashboard');
        });
        Route::get('account', function () {
                return view('account');
        });
});
```

Vistas

En algunos de los cierres de las rutas que hemos visto antes, hemos visto algo parecido a esto:

return view(‘account')

Si aún no estás familiarizado con el patrón MVC, las vistas son archivos que describen cómo debe aparecer una salida en particular. Puedes tener vistas en JSON, XML o emails, pero lo habitual es que esas vistas sean archivos HTML.

En Laravel, existen dos formatos de vistas que pueden usarse: PHP plano o plantillas Blade. La diferencia entre ambas es el nombre del archivo: about.php es renderizado usando el motor PHP, y about.blade.php será renderizado por el motor Blade.

Hay varias formas de cargar una vista desde una ruta:

La primera es esta:

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

Este códigomuestra como una vista en resources/views/home.blade.php o en resources/views/home.php, es cargada y se parsea el PHP hasta que se muestra la salida.

Pero qué ocurre si necesitamos pasar variables a la vista?

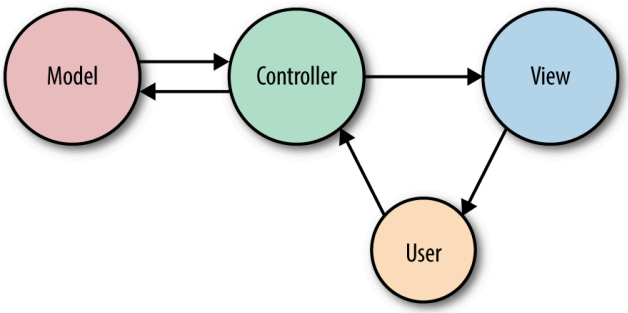
```
Route::get('tasks', function () {
        return view('tasks.index')
        ->with('tasks', Task::all());
});
```

Este cierre carga la vista resources/views/tasks/index.blade.php o resources/views/tasks/index.php y pasa una variable llamada tasks, que contiene el resultado del método Task::all(). Task::all() es una query a la base de datos en Eloquent que veremos más adelante.

Controladores y Vistas

Creación y uso de controladores

En el contexto de Laravel, los **controladores** son clases que gestionan la lógica de la aplicación y manejan las solicitudes HTTP entrantes. Su objetivo principal es separar la lógica de presentación de la lógica de negocio, siguiendo el principio de responsabilidad única.



Los controladores facilitan la estructuración y organización del código, mejorando la mantenibilidad y escalabilidad de las aplicaciones.

He mencionado los controladores varias veces, pero hasta ahora la mayoría de los ejemplos sólo muestran cierres de rutas. Si no está familiarizado con el patrón MVC, los controladores son esencialmente clases que organizan la lógica de una o más rutas juntas en un lugar. Los controladores tienden a agrupar rutas similares, especialmente si su aplicación está estructurada

según un formato tradicionalmente similar a CRUD; en este caso, un controlador podría manejar todas las acciones que se pueden realizar en un recurso en particular.

CRUD significa crear, leer, actualizar y eliminar, que son los cuatro principales operaciones que las aplicaciones web más comúnmente proporcionan en un recurso. Por ejemplo, puedes crear una nueva publicación de blog, puedes lee esa publicación, puedes actualizarla o puedes eliminarla.

Puede resultar tentador meter toda la lógica de la aplicación en los controladores, pero es mejor pensar en los controladores como policías de tráfico que enrutan las solicitudes HTTP de su aplicación.

Dado que hay otras formas en que las solicitudes pueden ingresar a su aplicación —trabajos cron, llamadas de línea de comandos de Artisan, trabajos en cola, etc.—es aconsejable no confiar en controladores para gran parte del comportamiento. Esto significa que el trabajo principal de un controlador es capturar la intención de una solicitud HTTP y pasarla al resto de la aplicación.

Entonces, creemos un controlador. Una manera fácil de hacer esto es con un comando Artisan, así que desde la línea de comando ejecute lo siguiente:

php artisan make:controller NombreControlador

Artisan se puede utilizar para ejecutar migraciones, crear usuarios y otros registros de la base de datos manualmente y realizar muchas otras tareas manualmente.

Bajo el espacio de nombres make, Artisan proporciona herramientas para generar archivos de esqueleto para una variedad de archivos del sistema. Eso es lo que nos permite ejecute php artisan make:controller.

php artisan make:controller TasksController

Esto creará un nuevo archivo llamado TaskController.php en app/Http/Controllers, con este contenido:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
class TasksController extends Controller
{
}
```

Modificamos el fichero anterior, creando un nuevo método público llamado home(). Sólo retornaremos un simple texto.

```
<?php
use App\Http\Controllers\Controller;
class TasksController extends Controller
{
	public function home()
	{
		return 'Hello, World!';
	}
}
```

Como aprendimos antes, ahora podemos linkarlo una ruta, de esta manera:

```
// routes/web.php
<?php
Route::get('/', ‘TasksController@home');
```

Si visitamos la ruta raiz / veremos las palabras 'Hello, World!'.

El uso más común de un método de controlador, entonces, será algo así:

```
// TasksController.php
...
public function index()
{
        return view('tasks.index')
        ->with('tasks', Task::all());
}
```

Este método de controlador carga resources/views/tasks/index.blade.php o resources/views/tasks/index.php y le pasa una única variable llamada tareas, que contiene el resultado del método de Eloquent Task::all().

Obtener información del usuario

La segunda acción más común a realizar en un método de controlador es tomar información del usuario y actuar en consecuencia. Esto introduce algunos conceptos nuevos, así que echemos un vistazo a un poco de código de muestra y recorra las nuevas piezas.

```
// routes/web.php
Route::get('tasks/create', 'TasksController@create');
Route::post('tasks', ‘TasksController@store');
```

Observa que estamos vinculando la acción GET de tasks/create (que muestra el formulario) y la acción POST de tasks/ (que es donde PUBLICAMOS cuando creamos una nueva tarea). Podemos asumir que el método create() en nuestro controlador solo muestra un formulario, por lo que veamos el método store():

```
// TasksController.php
...
public function store()
{
        $task = new Task;
        $task->title = Input::get('title');
        $task->description = Input::get('description');
        $task->save();
        return redirect('tasks');
}
```

Este ejemplo hace uso de modelos Eloquent y la funcionalidad de redirect(), de los que hablaremos más de ellos más adelante, pero puedes ver lo que estamos haciendo aquí: creamos un nueva tarea, extraiga datos de la entrada del usuario y configúrelo en la tarea, guárdelo y luego redirija la aplicación para volver a la página que muestra todas las tareas.

Hay dos formas principales de obtener la entrada del usuario desde un POST: el input facade, que hemos utilizado aquí, y el objeto Request, del que hablaremos a continuación.

Importación de facades

Si sigue alguno de estos ejemplos, ya sea en controladores o cualquier otra clase PHP que tenga un espacio de nombres, es posible que encuentre errores mostrando que no se puede encontrar el facade. Esto se debe a que no están presentes en todos los espacios de nombres, sino que están disponibles en el espacio de nombres raíz.

Entonces, en este ejemplo, necesitaríamos importar el facade de entrada en el parte superior del archivo. Hay dos formas de hacerlo: o podemos importar \Input, o podemos importar Illuminate\Support\Facades\Input.

Por ejemplo:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Support\facades\Input;
class TasksController
{
        public function store()
        {
                $task = new Task;
                $task->title = Input::get('title');
                $task->description = Input::get('description');
                $task->save();
        return redirect('tasks');
        }
}
```

Como puedes ver, podemos obtener el valor de cualquier información proporcionada por el usuario, ya sea de un parámetro de consulta o un valor POST, usando Input::get('fieldName'). Entonces nuestro usuario llenó dos campos en la página "add task": "title" y "description". recuperamos ambos usando la facade de Input, guárdelos en la base de datos y luego regrese.

Ejemplo

Vamos a crear un controlador llamado EjemploController que retornará una vista Laravel Blade para ello hacemos lo siguiente:

1º Creamos el controlador EjemploController mediante Laravel Artisan:

php artisan make:controller EjemploController

Como vimos anteriormente este comando creará un archivo en el directorio ‘app/Http/Controllers’ que se llamará ‘EjemploController’. Accediendo al controlador veremos que es una clase con el mismo nombre.

2º Preparar controlador EjemploController:

Una vez hemos creado el controlador el siguiente paso es modificarlo para que nos devuelva la vista que vamos a mostrar al usuario, para ello abrimos nuestro controlador y modificamos el archivo:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class EjemploController extends Controller
{
    public function index() {

        return view(‘ejemplo’);

    }
}
```

3º Preparar la vista que vamos a mostrar al usuario

Ya tenemos el controlador preparado para mostrar la vista Laravel Blade, con lo que lo siguiente que deberemos hacer será crearla y prepararla con la información que queramos mostrar. Vamos al directorio 'resources/views' y creamos un archivo que se llame 'ejemplo.blade.php' y lo editamos con la siguiente información:

```
<!DOCTYPE html>
<html>
<head>
   <title>Ejemplo</title>
   <style>
      h1 {
            color: green;
      }
    </style>
</head>
<body>
   <h1>Ejemplo de prueba de controladores</h1>

   <h3>¡Esta funcionando!</h3>
</body>
</html>
```

4º Asignar Ruta al controlador

Para poder llamar a nuestro controlador que va a cargar la vista deberemos crear una ruta GET. Vamos al directorio 'routes' y abrimos el archivo 'web.php'. Una vez tenemos el archivo abierto creamos la ruta get:

Laravel 9 y posterior:
Route::get('ejemplo', [EjemploController::class, 'index']);

Versiones inferiores a Laravel 9
Route::get('ejemplo', 'EjemploController@index');

5º Ejecutar la aplicación y probar ruta
Una vez hemos hechos los pasos anteriores solo nos queda probar, para ello ejecutamos nuestra aplicación y vamos a la ruta creada.

php artisan serve

Abrimos nuestro navegador favorito y vamos a la url (En mi caso localhost:8000/ejemplo). Como podemos ver si hemos realizado correctamente los pasos anteriores veremos la vista creada

Ejercicio

Crea una pequeña aplicación en Laravel que gestione información de libros. Implementa las siguientes características:

- Ruta de Listado de Libros:
  - Crea una ruta que muestre un listado de libros disponibles.
  - Utiliza un controlador para manejar la lógica de esta ruta.
  - En el controlador, define un array de libros ficticios con campos como título, autor y año de publicación.
  - En la vista asociada, muestra la información de cada libro en una lista.

- Ruta de Detalle de Libro:
  - Crea una ruta que permita ver los detalles de un libro específico.
    - Utiliza un parámetro en la URL para identificar el libro (por ejemplo, /libro/{id}).
    - En el controlador, busca el libro correspondiente según el parámetro y pasa la información a la vista.
    - La vista debería mostrar todos los detalles del libro, como título, autor y año de publicación.
  - Ruta de Creación de Libro:
    - Crea una ruta que permita agregar nuevos libros al sistema.
    - Utiliza un formulario para recopilar la información del nuevo libro (título, autor, año de publicación).
    - En el controlador, agrega la lógica para procesar el formulario y almacenar el nuevo libro en el array.

Solución

1. Ruta de Listado de Libros:

```
// routes/web.php

use App\Http\Controllers\LibroController;

Route::get('/libros', ‘LibroController@listado’);


// app/Http/Controllers/LibroController.php

namespace App\Http\Controllers;

class LibroController extends Controller
{
    public function listado()
    {
        $libros = [
            ['titulo' => 'Libro 1', 'autor' => 'Autor 1', 'anio' => 2022],
            ['titulo' => 'Libro 2', 'autor' => 'Autor 2', 'anio' => 2020],
            // Agrega más libros según sea necesario
        ];

        return view('libros.listado', compact('libros'));
    }
}
```

2. Ruta de Detalle de Libro:

```
// routes/web.php

use App\Http\Controllers\LibroController;

Route::get('/libro/{id}', ‘LibroController@detalle’);

// app/Http/Controllers/LibroController.php

namespace App\Http\Controllers;

class LibroController extends Controller
{
    public function detalle($id)
    {
```

```
        $libros = [
            ['titulo' => 'Libro 1', 'autor' => 'Autor 1', 'anio' => 2022],
            ['titulo' => 'Libro 2', 'autor' => 'Autor 2', 'anio' => 2020],
            // Agrega más libros según sea necesario
        ];

        $libro = $libros[$id - 1]; // Suponiendo que los IDs de los libros son secuenciales

        return view('libros.detalle', compact('libro'));
    }
}
```

3. Ruta de Creación de Libro:

```
// routes/web.php

use App\Http\Controllers\LibroController;

Route::get('/libros/crear', LibroController@formularioCreacion’);
Route::post('/libros/crear', ‘LibroController@crear’];

// app/Http/Controllers/LibroController.php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class LibroController extends Controller
{
    public function formularioCreacion()
    {
        return view('libros.formulario_creacion');
    }

    public function crear(Request $request)
    {
        // Agrega la lógica para validar y almacenar el nuevo libro en el array
        // Después de almacenar, redirige a la ruta de listado de libros

        return redirect('/libros');
    }
}
```

Estos son ejemplos básicos y puedes personalizarlos según tus necesidades. Asegúrate de crear las vistas correspondientes en el directorio resources/views/libros para que las rutas funcionen correctamente.

Controladores Resource en Laravel

Habitualmente cuando creamos una aplicación en Laravel, necesitamos realizar Operaciones CRUD (Create, Read, Update, Delete). Con Laravel tenemos la facilidad de que con un **simple comando** con Laravel Artisan **podemos crear un controlador con las funciones preparadas para el CRUD** y **además con una única línea de código podemos crear una ruta que pueda registrar todos los métodos de este CRUD**.

Tomemos un ejemplo, queremos una aplicación con un CRUD de gestor de imágenes mediante Laravel:

1º Creamos un controlador ImageController con CRUD incluido
Creamos un controlador en Laravel mediante el comando que hemos usado anteriormente, pero si **añadiendo --resource**, este añadido **lo que hace es preparar todos los métodos para hacer el CRUD** a nuestro gusto.

php artisan make:controller ImageController —resource

2º Creamos la ruta en web.php para CRUD
En lugar de tener que crear una ruta para cada método podemos crear una ruta que se asigne automáticamente a cada uno de los métodos del controlador creado anteriormente para ello abrimos web.php y creamos la siguiente ruta:

Laravel 9 y posterior:
Route::resource(images, [ImageController::class]);

Versiones inferiores a Laravel 9
Route::resource('images', 'ImageController');

Como podemos ver a través del siguiente comando tenemos todas las rutas sin tener que ir añadiéndolas una a una

php artisan route:list

Cuando creamos un controlador Resource, automáticamente se generan una serie de métodos en el propio controlador:

```
//Muestra una lista de elementos. Es de tipo GET

  public function index()
  {
      //
  }


  // Munestra el formulario para crear un elemento. Es de tipo GET
   public function create()
   {
      //
   }

  // Guarda un nuevo elemento creado. Es de tipo POST
   public function store(Request $request)
   {
      //
   }

   // Muestra el elemento indicado por su id. Es de tipo GET
   public function show(string $id)
   {
      //
   }

   // Muestra el formulario para editar el elemento especificado. Es de tipo GET
    */
   public function edit(string $id)
   {
      //
   }
```

```
    // Actualiza el elemento especificado. Es de tipo PUT
    public function update(Request $request, string $id)
    {
        //
    }

    // Borra el elemento especificado. Es de tipo DELETE
    public function destroy(string $id)
    {
        //
    }
```

Vincular un controlador de recursos

Entonces, hemos visto que estos son los nombres de rutas convencionales a usar en Laravel, y también que es fácil generar un controlador de recursos con métodos para cada uno de estos valores predeterminados rutas. Afortunadamente, no es necesario generar rutas para cada uno de estos métodos a mano, si no lo desea. En cambio, hay un truco para eso y se llama "vinculación del controlador de recursos". Eche un vistazo al ejemplo:

```
// routes/web.php
Route::resource('tasks', ‘TasksController');
```

Esto vinculará automáticamente todas las rutas de este recurso al nombre del método apropiado dentro del controlador. También nombrará apropiadamente estas rutas; por ejemplo, el método index() en el controlador de recursos de tareas se denominará tareas.index.

Inyección de dependencias y controladores
La inyección de dependencias hace que nuestras aplicaciones web sean más fáciles de probar y mantener.

Constructor Injection
Laravel gestiona las dependencias de clase y resuelve todos los controladores. Tu controlador puede necesitar dependencias en el constructor, con Laravel, puedes escribir la mayoría de estas dependencias. El controlador de servicios de Laravel va a resolver todas las dependencias automáticamente y las inyectará posteriormente en la instancia del controlador.

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class MyController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;
    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

Method Injection
Aparte de inyectar dependencias en el constructor, también puedes inyectarlas en métodos de tu controlador. Por ejemplo, la instancia Illuminate\Http\Request se puede inyectar en el método del controlador de la siguiente manera:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MyController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->name;
        //
    }
}
```

Características de los Controladores de Laravel
A continuación vas a ver algunas de las características de los controladores de Laravel:

- **Soporte MVC:** Como dijimos anteriormente Laravel está basado en la arquitectura MVC. El uso de esta arquitectura hace que el desarrollo de nuestras aplicaciones web sea más rápido ya que por ejemplo un programador pueda centrarse en la lógica y otro en la vista, por ejemplo. Además, Laravel soporte múltiples vistas para un modelo sin duplicación ya que la lógica de negocio está separada de la lógica de presentación.

- **Authenticacion:** Laravel incluye un sistema para la authenticacion con lo que solo tienes que ocuparte de otros aspectos como son la configuración de modelos, vistas y controladores.

- **Seguridad:** Como todos sabemos, la seguridad es el factor más importante a tener en cuenta durante el desarrollo de una aplicación. Laravel proporciona una gran seguridad gracias a sus sistema de seguridad incorado.

- **Artisan:** Como te habrás dado cuenta los comandos mediante Laravel Artisan nos permiten ahorrarnos mucho tiempo ya que nos ayuda a realizar tareas repetitivas sin que tengan que ser realizadas manualmente por nosotros.

- **Plantillas:** Gracias al innovador y potente motor de plantillas de Laravel (las vistas Blade) podemos crear aplicaciones web Dinámicas fácilmente.

Trabajo con vistas y plantillas Blade

En comparación con la mayoría de los otros lenguajes backend, PHP en realidad funciona relativamente bien con un lenguaje de plantillas. Pero tiene sus defectos y además es feo usar

php en línea en todas partes, por lo que puede esperar que la mayoría de los marcos modernos ofrezcan un lenguaje de plantillas.

Laravel ofrece un motor de plantillas personalizado llamado Blade, que está inspirado en .NET. Cuenta con una sintaxis concisa, una curva de aprendizaje poco profunda, un potente y modelo de herencia intuitivo y fácil extensibilidad.

Para ver rápidamente cómo se ve escribir Blade, veamos un ejemplo:

```
<h1>{{ $group->title }}</h1>

{!! $group->heroImageHtml() !!}

@forelse ($users as $user)
        {{ $user->first_name }} {{ $user->last_name }}<br>
@empty
No users in this group.
@endforelse
```

Como puedes ver, Blade introduce una convención en la que sus etiquetas personalizadas, llamadas “directivas", tienen el prefijo @. Utilizarás directivas para todas tus estructuras de control, y también para herencia y cualquier funcionalidad personalizada que desee agregar.

La sintaxis de Blade es limpia y concisa, por lo que en esencia es más agradable y ordenado trabajar que las alternativas. Pero en el momento en que necesitas algo de cualquier complejidad en sus plantillas (herencia anidada, condicionales complejos o recursividad) Blade empieza a brillar de verdad.

Además, dado que toda la sintaxis de Blade se compila en código PHP normal y luego almacenado en caché, es rápido y le permite usar PHP nativo en sus archivos Blade si lo deseas. Sin embargo, recomendaría evitar el uso de PHP si es posible, generalmente si necesitas hacer algo que no puedas hacer con Blade o una directiva Blade personalizada.

Enseñando datos

Como se puede ver en el ejemplo anterior, hemos usado secciones entre {{ }} que funcionan como un echo en PHP. Es decir, {{ $variable }} es lo mismo que <?php echo $variable;?> en PHP plano. Aunque es diferente en cierta manera, porque Blade hace echo de las variables usando htmlentities, para proteger a los usuarios de la inyección de código malicioso. Esto significa que {{ $variable }} es lo mismo que <?php echo htmlentities($variable);?>

Estructuras de control

La mayoría de las estructuras de control en Blade te resultarán muy familiares. Muchas de ellas tienen el mismo nombre y estructura que en PHP.

Condicionales

@if ($condicion) se compila como <? If (condicion) ?>. Veamos un ejemplo:

```
@if (count($talks) === 1)
        There is one talk at this time period.
@elseif (count($talks) === 0)
        There are no talks at this time period.
@else
        There are {{ count($talks) }} talks at this time period.
@endif
```

Al igual que con los condicionales nativos de PHP, puedes mezclarlos y combinarlos como quieras. No tienen ninguna lógica especial; hay literalmente un analizador buscando algo

con la forma de @if ($condición) y reemplazándolo con el código PHP apropiado.

@unless and @endunless

@unless, por otro lado, es una nueva sintaxis que no tiene un equivalente directo en PHP. Es el inverso directo de @if. @unless ($condición) es lo mismo que <?php if (! $condición). Véalo en uso en el siguiente ejemplo:

```
@unless ($user->hasPaid())
You can complete your payment by switching to the payment tab.
@endunless
```

Bucles

Veamos ahora como Blade trata a los bucles.

@for, @foreach, and @while

@for, @foreach, y @while trabajan de la misma manera que lo hacen en PHP:

```
@for ($i = 0; $i < $talk->slotsCount(); $i++)
The number is {{ $i }}<br>
@endfor

@foreach ($talks as $talk)
• {{ $talk->title }} ({{ $talk->length }} minutes)<br>
@endforeach

@while ($item = array_pop($items))
{{ $item->orSomething() }}<br>
@endwhile
```

@forelse

@forelse is como un@foreach que permite programar un fallback en el caso en que el objeto que estás iteando esté vacío.

```
@forelse ($talks as $talk)
• {{ $talk->title }} ({{ $talk->length }} minutes)<br>
@empty
No talks this day.
@endforelse
```

Las directivas @foreach y @forelse añaden una característica que no está disponible en PHPpara cada bucle: el la variable $loop. Usada dentro de un bucle @foreach o @forelse, esta variable devolverá un objeto stdClass con las siguientes propiedades:

index: el índice del término en curso, sabiendo que el primer item tiene índice 0.

Iieration: el índice de base 1 del item en curso en el bucle.

remaining: cuántos items quedan en el bucle. Si el item en curso es el primero de tres, este valor valdrá 2.

count: el número de items en el bucle.

first: un booleano que indica que éste es el primer item del bucle.

last: un booleano que indica que éste es el último item del bucle.

depth: cuántos niveles de profundidad tiene el bucle: 1 para un bucle, 2 para un bucle dentro de un bucle…

parent: Una referencia a la variable $loop para el elemento del bucle principal; si este bucle está dentro otro bucle @foreach; de lo contrario, nulo.

```
<ul>
        @foreach ($pages as $page)
                <li>{{ $loop->iteration }}: {{ $page->title }}
                        @if ($page->hasChildren())
                                <ul>
                                @foreach ($page->children() as $child)
                                        <li>{{ $loop->parent->iteration }}.
                                                {{ $loop->iteration }}:
                                                {{ $child->title }}
                                        </li>
                                @endforeach
                                </ul>
                        @endif
                </li>
        @endforeach
</ul>
```

or

Si alguna vez no está seguro de si una variable está configurada, probablemente esté acostumbrado a verifica isset() en él antes de repetirlo, y repetir algo más si no está configurado. La hoja tiene un asistente de conveniencia, o que hace esto por usted y le permite establecer un respaldo predeterminado:

{{ $title or "Default" }} repetirá el valor de $title si está configurado, o “Default" si no.

Herencia de plantillas

Blade proporciona una estructura de herencia de plantillas que permite extender las vistas, modificarlas e incluir a otras.

Un motor de plantillas facilita la escritura del código frontend y ayuda a reutilizar el código. Todos los archivos blade tienen una extensión de **.blade.php* . En Laravel, la mayoría de las veces los archivos de interfaz se almacenan en el directorio de *recursos/vistas* . Los archivos Blade son compatibles con PHP y se compilan en PHP simple y se almacenan en caché en el servidor para que no tengamos que hacer el trabajo adicional de compilar las plantillas nuevamente cuando un usuario accede a una página nuevamente, por lo que usar Blade es tan eficiente como usar archivos PHP. en sí mismo en la interfaz.

Herencia de plantilla: en la mayoría de las páginas web modernas, se sigue un tema fijo en todas las páginas web. Por lo tanto, es muy efectivo poder reutilizar su código para que no tenga que volver a escribir las partes repetitivas en su código y Blade lo ayuda enormemente a lograrlo.

Definición de un diseño: Hagámoslo con un ejemplo y creemos un archivo llamado **layout.blade.php** en el directorio de **recursos/vistas** como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>@yield('title')</title>
</head>
<body>
   <div>
      @yield('content')
   </div>
</body>
</html>
```

Ahora, en el código anterior, usamos la directiva **@yield** para decirle al Blade que vamos a ampliar aún más esta parte en las páginas secundarias del Blade. Además, observe que cada una de las directivas de rendimiento tiene un nombre como **título** para la primera y **contenido** para la segunda. Estos nombres se usarán más adelante en la página secundaria para indicar que esta sección se extiende aquí.

Extender un diseño: Hagámoslo también ahora y creemos una página en el directorio de **recursos/vistas** llamada *mypage.blade.php* como se indica a continuación:

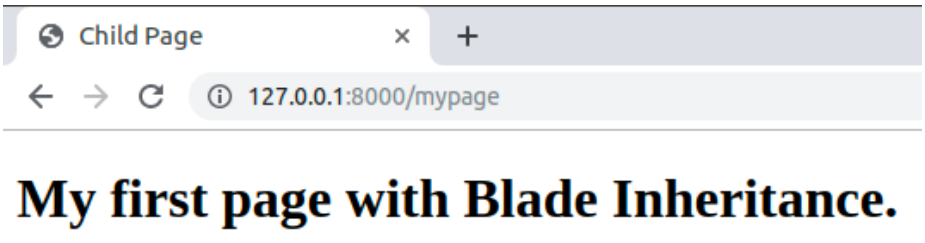
```
@extends('layout')


@section('title')
   Child Page
@endsection


@section('content')
   <h1>My first page with Blade Inheritance.</h1>
@endsection
```

En este código, primero usamos la directiva **@extends** que indica de qué página blade heredamos esta página. En nuestro caso, será el diseño, ya que heredaremos esta página de **layout.blade.php** , que creamos anteriormente.

Además, usamos la directiva **@section** para extender cada una de las directivas **@yield** del archivo blade principal. Tenemos que decir el nombre de cada directiva **@yield** que estamos extendiendo aquí en la directiva **@section** como lo hemos hecho en el código anterior.

Después de escribir el código, asegúrese de finalizar la directiva con @*endsection* . Las secciones se reemplazarán con el código respectivo en las páginas de hoja secundaria. Una última cosa que queda para que esto funcione es agregar una ruta como se indica a continuación en tus **rutas/ web.php** .

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



En el resultado, puede ver cómo @yield(‘title’) se reemplaza con Child Page y @yield(‘content’) se reemplaza con My first page with Blade Inheritance.

Inclusión de subvistas

Aunque es libre de utilizar la directiva @include, los componentes Blade proporcionan una funcionalidad similar y ofrecen varias ventajas sobre la directiva @include, como la vinculación de datos y atributos.

La directiva @include de Blade permite incluir una vista Blade dentro de otra vista. Todas las variables disponibles para la vista padre estarán disponibles para la vista incluida:

```
<div>
   @include('shared.errors')

   <form>
      <!-- Form Contents -->
   </form>
</div>
```

Aunque la vista incluida heredará todos los datos disponibles en la vista padre, también puede pasar una array de datos adicionales que se pondrán a disposición de la vista incluida:

```
@include('view.name', ['status' => ‘complete'])
```

Si intentas incluir una vista que no existe, Laravel arrojará un error. Si quieres incluir una vista que puede o no estar presente, debes usar la directiva @includeIf:

```
@includeIf('view.name', ['status' => ‘complete'])
```

Si quieres incluir una vista si una expresión booleana dada es true o false, puedes usar las directivas @includeWhen y @includeUnless:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])

@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

Para incluir la primera vista que exista de un array de vistas dado, puede utilizar la directiva includeFirst:

```
@includeFirst(['custom.admin', 'admin'], ['status' => ‘complete'])
```

Advertencia

Evita utilizar las constantes __DIR__ y __FILE__ en tus vistas Blade, ya que harán referencia a la ubicación de la vista compilada en caché.

Renderizado de vistas para colecciones

Puede combinar bucles e includes en una sola línea con la directiva @each de Blade:

```
@each('view.name', $jobs, ‘job')
```

El primer argumento de la directiva @each es la vista a mostrar para cada elemento del array o colección. El segundo argumento es la array o colección sobre la que desea iterar, mientras que el tercer argumento es el nombre de la variable que se asignará a la iteración actual dentro de la vista. Así, por ejemplo, si estás iterando sobre un array de jobs, normalmente querrás acceder a

cada trabajo como una variable de job dentro de la vista. La clave de la array para la iteración actual estará disponible como variable key dentro de la vista.

También puede pasar un cuarto argumento a la directiva @each. Este argumento determina la vista que se mostrará si la array dada está vacía.

@each('view.name', $jobs, 'job', ‘view.empty')

La Directiva @once

La directiva @once permite definir una parte de la plantilla que sólo se evaluará una vez por ciclo de renderizado. Esto puede ser útil para empujar una determinada pieza de JavaScript en la cabecera de la página utilizando pilas. Por ejemplo, si está renderizando un componente determinado dentro de un bucle, puede que sólo desee insertar el JavaScript en la cabecera la primera vez que se renderice el componente:

```
@once
   @push('scripts')
      <script>
         // Your custom JavaScript...
      </script>
   @endpush
@endonce
```

Dado que la directiva @once se utiliza a menudo junto con las directivas @push o @prepend, las directivas @pushOnce y @prependOnce están disponibles para su conveniencia:

```
@pushOnce('scripts')
   <script>
      // Your custom JavaScript...
   </script>
@endPushOnce
```

PHP en crudo

En algunas situaciones, es útil incrustar código PHP en las vistas. Puede utilizar la directiva @php de Blade para ejecutar un bloque de PHP plano dentro de su plantilla:

```
@php
   $counter = 1;
@endphp
```

Si sólo necesita escribir una única sentencia PHP, puede incluir la sentencia dentro de la directiva @php:

```
@php($counter = 1)
```

Blade también le permite definir comentarios en sus vistas. Sin embargo, a diferencia de los comentarios HTML, los comentarios de Blade no se incluyen en el HTML devuelto por su aplicación:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Recogiendo y manejando datos del usuario

Los sitios web que se benefician de un marco como Laravel a menudo no solo ofrecen contenido estático. Muchos tratan con fuentes de datos complejas y mixtas, y uno de los más comunes (y la más compleja) de estas fuentes es la entrada del usuario en sus innumerables formas: rutas URL, parámetros de consulta, datos POST y cargas de archivos.

Laravel proporciona una colección de herramientas para recopilar, validar, normalizar y filtrar. datos proporcionados por el usuario.

Inyectar un objeto Request

La herramienta más común para acceder a los datos del usuario en Laravel es inyectar una instancia de el objeto Illuminate\Http\Request. Proporciona fácil acceso a todas las formas en que los usuarios puede proporcionar información a su sitio: POST, JSON, GET (parámetros de consulta) y segmentos URL.

Dado que estamos planeando inyectar un objeto Request , echemos un vistazo rápido a cómo obtener el objeto $request al que llamaremos todos estos métodos:

```
Route::post('form', function (Illuminate\Http\Request $request) {
// $request->etc()
});
```

$request->all()

Tal como sugiere el nombre, $request->all() le proporciona una matriz que contiene todos los información que el usuario ha proporcionado, de todas las fuentes. Digamos que, por alguna razón, decidiste tener un formulario POST a una URL con un parámetro de consulta, por ejemplo, enviar un POST a http://myapp.com/post?utm=12345. Eche un vistazo al ejemplo para ver lo que obtendría desde $request->all(). (Tenga en cuenta que $request->all() también contiene información sobre cualquier archivo que se haya subido, pero lo cubriremos más adelante en el capítulo).

```
<!-- GET route form view at /get-route -->
<form method="post" action="/post-route?utm=12345">
{{ csrf_field() }}
<input type="text" name="firstName">
<input type="submit">
</form>

Route::post('/post-route', function (Request $request) {
var_dump($request->all());
});
// Outputs:
/**
* [
* '_token' => 'CSRF token here',
* 'firstName' => 'value',
* 'utm' => 12345
* ]
*/
```

$request->except() y $request->only()

$request->except() proporciona el mismo resultado que $request->all, pero puedes elija uno o más campos para excluir, por ejemplo, _token. Puedes pasarlo ya sea un cadena o una serie de cadenas.

```
Route::post('/post-route', function (Request $request) {
var_dump($request->except('_token'));
});
// Outputs:
/**
* [
* 'firstName' => 'value',
* 'utm' => 12345
* ]
*/
```

$request-only() es la función inversa de $request->except():

```
Route::post('/post-route', function (Request $request) {
var_dump($request->only(['firstName', 'utm']));
});
// Outputs:
/**
* [
* 'firstName' => 'value',
* 'utm' => 12345
* ]
*/
```

$request->has() y $request->exists()

Con $request->has() puedes detectar si una parte particular de la entrada del usuario es disponible para ti. Consulte el ejemplo para ver un ejemplo de análisis con nuestra consulta utm. parámetro de cadena de los ejemplos anteriores.

```
// POST route at /post-route
if ($request->has('utm')) {
// Do some analytics work
}
```

$request->exists() y $request->has() se diferencian en que manejan valores vacíos de manera diferente: has() devuelve FALSO si la clave existe y está vacía; existe() devuelve VERDADERO si la clave existe, incluso si está vacía.

$request->input()

Mientras que $request->all(), $request->except() y $request->only() operan en la gama completa de entradas proporcionadas por el usuario, $request->input() le permite obtener el valor de un solo campo. El ejemplo proporciona un ejemplo. Tenga en cuenta que el segundo parámetro es el valor predeterminado, por lo que si el usuario no ha pasado un valor, puede tener un un respaldo sensato (e irrompible).

```
Route::post('/post-route', function (Request $request) {
$userName = $request->input('name', '(anonymous)');
});
```

Array Input

Laravel también proporciona ayudas convenientes para acceder a los datos desde la entrada de la matriz. Justo use la notación de "punto" para indicar los pasos para profundizar en la estructura de la matriz, como en el ejemplo.

```
<!-- GET route form view at /get-route -->
<form method="post" action="/post-route">
{{ csrf_field() }}
<input type="text" name="employees[0][firstName]">
<input type="text" name="employees[0][lastName]">
<input type="text" name="employees[1][firstName]">
<input type="text" name="employees[1][lastName]">
<input type="submit">
</form>

// POST route at /post-route
Route::post('/post-route', function (Request $request) {
$employeeZeroFirstName = $request->input('employees.0.firstName');
$allLastNames = $request->input('employees.*.lastName');
$employeeOne = $request->input('employees.1');
});
// If forms filled out as "Jim" "Smith" "Bob" "Jones":
// $employeeZeroFirstName = 'Jim';
// $allLastNames = ['Smith', 'Jones'];
// $employeeOne = ['firstName' => 'Bob', 'lastName' => ‘Jones']
```

JSON Input (y $request->json())

Hasta ahora hemos cubierto la entrada de cadenas de consulta (GET) y envíos de formularios (POST). Pero hay otra forma de entrada del usuario que se está volviendo más común con la llegada de Aplicaciones JavaScript de una sola página (SPA): la solicitud JSON. Es esencialmente solo una solicitud POST con el cuerpo configurado en JSON en lugar de un formulario POST tradicional.

Echemos un vistazo a cómo se vería enviar algún JSON a una ruta Laravel. y cómo usar $request->input() para extraer esos datos.

```
POST /post-route HTTP/1.1
Content-Type: application/json
{
"firstName": "Joe",
"lastName": "Schmoe",
"spouse": {
"firstName": "Jill",
"lastName":"Schmoe"
}
}
// post-route
Route::post('post-route', function (Request $request) {
$firstName = $request->input('firstName');
$spouseFirstname = $request->input('spouse.firstName');
});
```

Dado que $request->input() es lo suficientemente inteligente como para extraer datos del usuario de GET, POST o JSON, quizás te preguntes por qué Laravel incluso ofrece $request->json(). Hay dos razones por la que es posible que prefieras $solicitud->json(). En primer lugar, es posible que desee ser más explícito al hacer entender a otros programadores en su proyecto sobre dónde espera que lleguen los datos. Y segundo, si el POST no tiene los encabezados application/json correctos, $request->input() no lo recogerá como JSON, pero $request->json() sí.

Upload de archivos

Hemos hablado de diferentes formas de interactuar con la entrada de texto de los usuarios, pero también existe la Cuestión de carga de archivos a considerar. La facade Request brinda acceso a cualquier archivo cargado. usando el método Request::file(), que toma el nombre de entrada del archivo como un parámetro y devuelve una instancia de Component\HttpFoundation\File \UploadedFile.

```
<form method="post" enctype="multipart/form-data">
{{ csrf_field() }}
<input type="text" name="name">
<input type="file" name="profile_picture">
<input type="submit">
</form>

Route::post('form', function (Request $request) {
var_dump($request->all());
});
// Output:
// [
// "_token" => "token here"
// "name" => "asdf"
// "profile_picture" => UploadedFile {}
// ]
Route::post('form', function (Request $request) {
if ($request->hasFile('profile_picture')) {
var_dump($request->file('profile_picture'));
}
});
// Output:
// UploadedFile (details)
```

Bases de datos y Eloquent

Laravel proporciona un conjunto de herramientas para interactuar con las bases de datos de su aplicación, pero el más notable es Eloquent, ActiveRecord ORM de Laravel (objeto-relacional). mapeador).

Eloquent es una de las funciones más populares e influyentes de Laravel. Es un gran ejemplo de en qué se diferencia Laravel de la mayoría de los frameworks PHP; en un mundo de datos Mapper ORMs potentes pero complejos, Eloquent destaca por su sencillez.

Hay una clase por tabla, que es responsable de recuperar, representar y persistir. datos en esa tabla.

Sin embargo, ya sea que elijas usar Eloquent o no, seguirás obteniendo muchos beneficios. de las otras herramientas de base de datos que proporciona Laravel. Entonces, antes de profundizar en Eloquent, comenzaremos cubriendo los conceptos básicos de la funcionalidad de la base de datos de Laravel: migraciones, seeders, y el generador de consultas.

Luego cubriremos Eloquent: definir sus modelos; insertar, actualizar y eliminar; personalizar sus respuestas con accesores, mutadores y conversión de atributos; y finalmente relaciones. Están sucediendo muchas cosas aquí y es fácil sentirse abrumado, pero simplemente da un paso a la vez y lo lograremos.

Configuración

Antes de ver cómo usar las herramientas de base de datos de Laravel, hagamos una pausa por un segundo y vayamos sobre cómo configurar las credenciales y conexiones de su base de datos. La configuración para el acceso a la base de datos se encuentra en config/database.php. Como muchos otros áreas de configuración en Laravel, puede definir múltiples "conexiones" y luego decidir cual es la que el código utilizará de forma predeterminada.

Conexiones a la base de datos

Por defecto, hay una conexión por cada uno de los tipos de conexión, como puedes ver en el ejemplo:

```
'connections' => [
'sqlite' => [
'driver' => 'sqlite',
'database' => database_path('database.sqlite'),
'prefix' => '',
],
'mysql' => [
'driver' => 'mysql',
'host' => env('DB_HOST', 'localhost'),
'database' => env('DB_DATABASE', 'forge'),
'username' => env('DB_USERNAME', 'forge'),
'password' => env('DB_PASSWORD', ''),
'charset' => 'utf8',
'collation' => 'utf8_unicode_ci',
'prefix' => '',
'strict' => false,
'engine' => null,
],
'pgsql' => [
'driver' => 'pgsql',
'host' => env('DB_HOST', 'localhost'),
'database' => env('DB_DATABASE', 'forge'),
'username' => env('DB_USERNAME', 'forge'),
'password' => env('DB_PASSWORD', ''),
'charset' => 'utf8',
'prefix' => '',
'schema' => 'public',
],
'sqlsrv' => [
'driver' => 'sqlsrv',
'host' => env('DB_HOST', 'localhost'),
'database' => env('DB_DATABASE', 'forge'),
'username' => env('DB_USERNAME', 'forge'),
'password' => env('DB_PASSWORD', ''),
'charset' => 'utf8',
'prefix' => '',
],
]
```

Sin embargo, podrías crear nuevas conexiones con nombre y aún así poder configurar los controladores (MySQL, Postgres, etc.) en esas nuevas conexiones con nombre. Entonces, si bien hay una conexión por controlador de forma predeterminada, eso no es una restricción.

Cada conexión le permite definir las propiedades necesarias para conectarse y personalizar cada tipo de conexión. Hay algunas razones para la idea de múltiples controladores. Para empezar, las “conexiones”, tal como sale de la caja es una plantilla simple que facilita el inicio.

Las aplicaciones utilizan cualquiera de los tipos de conexión de base de datos admitidos. En muchas aplicaciones, puedes elegir la conexión de base de datos que utilizará, complete su información e incluso elimine los demás si quieres. Generalmente los guardo todos allí, en caso de que eventualmente pueda usarlos.

Pero también hay algunos casos en los que es posible que necesites varias conexiones dentro del misma aplicación. Por ejemplo, podría utilizar diferentes conexiones de bases de datos para dos diferentes tipos de datos, o puede leer de uno y escribir en otro. El apoyo para múltiples conexiones hacen esto posible.

Migraciones

Los frameworks modernos como Laravel facilitan la definición de la estructura de su base de datos con migraciones basadas en código. Cada nueva tabla, columna, índice y clave se puede definir en código, y cualquier entorno nuevo se puede trasladar desde la base de datos básica al entorno perfecto de su aplicación en segundos.

Una migración es un único archivo que define dos cosas: las modificaciones deseadas al ejecutar esta migración y las modificaciones deseadas al ejecutar esta migración abajo.

Las migraciones siempre se ejecutan en orden por fecha. Cada archivo de migración tiene un nombre así: 2014_10_12_000000_create_users_table.php. Cuando se migra un nuevo sistema, el sistema toma cada migración, comenzando en la fecha más temprana, y ejecuta el método up(): lo estás migrando “hacia arriba” en este punto.

Pero el sistema migratorio también permite "revertir" su conjunto de migraciones más reciente. Agarrará a cada uno de ellos y correrá. su método down(), que debería deshacer cualquier cambio realizado en la migración hacia arriba. Entonces, el método up() de una migración debería “realizar” su migración, y el método down() debería “deshacerlo”.

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateUsersTable extends Migration
{
/* Run the migrations.
@return void
*/
public function up()
{
Schema::create('users', function (Blueprint $table) {
$table->increments('id');
$table->string('name');
$table->string('email')->unique();
$table->string('password', 60);
$table->rememberToken();
$table->timestamps();
});
}
/**
* Reverse the migrations.
*
* @return void
*/
public function down()
{
Schema::drop('users');
}
}
```

Como puede ver, tenemos un método up() y un método down(). up() le dice a la migración para crear una nueva tabla llamada usuarios con algunos campos, y down() le dice que se elimine la tabla de usuarios.

Creando una migración

Podemos crear migraciones usando comandos de Artisan para ello. Por ejemplo, para crear la migración que acabamos de ver, usamos

php artisan make:migration create_users_table

Hay otras dos opciones que le podemos añadir al este comando de creación de migraciones:

—create=table_name , que crea una tabla
—table=_table_name_ , que crea la migración para una tabla ya existente

Por ejemplo:

php artisan make:migration create_users_table
php artisan make:migration add_votes_to_users_table --table=users
php artisan make:migration create_users_table —create=users

Creando tablas

Para crear una nueva tabla en una migración, use el método create(); el primer parámetro es el nombre de la tabla, y el segundo es un cierre que define sus columnas:

```
Schema::create('tablename', function (Blueprint $table) {
// Create columns here
});
```

Creando columnas

Para crear nuevas columnas en una tabla, o modificar la tabla, usaremos la instancia de Blueprint:

```
Schema::create('users', function (Blueprint $table) {
$table->string('name');
});
```

Veamos los métodos que disponemos en Blueprint para crear columnas:

integer(colName), tinyInteger(colName), smallInteger(colName), mediumInte
ger(colName), bigInteger(colName)
Añañde una columna de tipo entero

string(colName, OPTIONAL length)
Añade una columna de tipo VARCHAR

boolean(colName)
Añade una columna de tipo booleano

datetime(colName)
Añade una columna DATETIME

decimal(colName, precision, scale)
Añade una columna de tipo DECIMAL column, con una precisión de “precision” y una escala de “scale”

enum(colName, [choiceOne, choiceTwo])
Añade una columna ENUM con los valores proporcionados

text(colName), mediumText(colName), longText(colName)
Añade una columna TEXT

time(colName)
Añade una columna TIME

timestamp(colName)
Añade una columna TIMESTAMP

Además tenemos algunas funciones o métodos especiales:

increments(colName) and bigIncrements(colName)
Añade una clave primaria o autoincremental ID

rememberToken()
Añade iun `remember_token` column (`VARCHAR(100)`) para usar como un token “remember me”.

Añadiendo otras propiedades a los campos

El resto de propiedades de un campo se añaden como un segundo parámetro en la creación del campo. Por ejemplo, veamos como se crea un campo email no nulo después de last_name:

```
Schema::table('users', function (Blueprint $table) {
$table->string('email')->nullable()->after('last_name');
});
```

Veamos los siguientes métodos para añadir propiedades a los campos:

nullable()
Permite el valor NULL.

default('default content')
Especifica contenido por defecto para un campo.

after(colName) (MySQL only)
Coloca el campo despues de colName

unique()
Añade un campo único

primary()
Añade una clave primaria

index()
Añade un campo index.

Dropar una tabla

Si queremos deshacernos de una tabla:

```
Schema::drop(‘contacts');
```

Modificar columnas

Para modificar una columna, sólo necesitas escribir el código que escribirías si fueras a crear esa columna con los nuevos parámetros. Por ejemplo, si tenemos un campo de caracteres de 255 caracteres de longitud y lo queremos cambiar a 100:

```
Schema::table('users', function ($table) {
$table->string('name', 100)->change();
});
```

Para renombrar una columna:

```
Schema::table('contacts', function ($table)
{
$table->renameColumn('promoted', 'is_promoted');
});
```

Podemos usar una simple llamada para cambiar más de un campo, en lugar de hacerlo de uno en uno:

```
public function up()
{
Schema::table('contacts', function (Blueprint $table)
{
$table->dropColumn('is_promoted');
});
Schema::table('contacts', function (Blueprint $table)
{
$table->dropColumn('alternate_email');
});
}
```

Modificar claves primarias e índices.

Sabemos que, una vez creadas las columnas, añadimos las claves primarias y demás.

```
// after columns are created...
$table->primary('primary_id'); // Primary key; innecesario si se usa increments()
$table->primary(['first_name', 'last_name']); // Clave primaria compuesta
$table->unique('email'); // Índice único
$table->unique('email', 'optional_custom_index_name'); // Índice único
$table->index('amount'); // Índice
$table->index('amount', 'optional_custom_index_name'); // Indice
```

Para borrar esos índices:

```
$table->dropPrimary('contacts_id_primary');
$table->dropUnique('contacts_email_unique');
$table->dropIndex(‘optional_custom_index_name’);
```

Claves foráneas

Para añadir una clave foránea:

```
$table->foreign(‘user_id')->references('id')->on('users');
```

La clave foránea es user_id, que hace referencia a id de la tabla users.

Para borrarla:

$table->dropForeign(‘contacts_user_id_foreign');

Ejecutando las migraciones

Una vez que están definidas las migraciones, debemos ejecutarlas:

php artisan migrate

Hay unas cuantas opciones que podemos añadir a la ejecución de la migración:

php artisan migrate —seed // Esto añade el sembrado de datos de prueba para ejecutar nuestra aplicación

migrate:install crea las tablas de labase de datos

migrate:reset
deshace todas las migraciones de bases de datos que haya ejecutado en esta instalación.

migrate:refresh
revierte cada migración de base de datos que haya ejecutado en esta instalación,
y luego ejecuta todas las migraciones disponibles. Es lo mismo que ejecutar migrar: restablecer
y luego migrar, uno tras otro.

migrate:rollback
revierte solo las migraciones que se ejecutaron la última vez que ejecutó
migrar o, con la opción agregada --step=1, revertir el número de migraciones que tu hayas específicado.

migrate:status
muestra una tabla que enumera cada migración, con una Y o una N al lado de cada una
mostrando si ya se ha ejecutado o no en este entorno.

Sembrado de datos

Sembrar con Laravel es tan simple que ha obtenido una adopción generalizada como parte de la rutina normal de los flujos de trabajo de desarrollo de una manera que no lo había hecho en marcos PHP anteriores.

Hay una carpeta de databases/seed que viene con una clase DatabaseSeeder, que tiene una función run(), método que se llama cuando llamas a la sembradora.

Hay dos formas principales de ejecutar los sembradores: junto con una migración o por separado.
Para ejecutar una sembradora junto con una migración, simplemente agregue --seed a cualquier llamada de migración:

php artisan migrate --seed
php artisan migrate:refresh —seed

Y la otra es ejecutarlas independientemente:

php artisan db:seed
php artisan db:seed —class=VotesTableSeeder

Esto ejecutará todo lo que haya definido en los métodos run() de cada clase de sembradora.
(o simplemente la clase a la que pasaste —class).

Creando un sembrador

Para crear un sembrador, sólo necesita el comando de Artisan:

php artisan make:seeder ContactsTableSeeder

Ahora verá una nueva clase ContactsTableSeeder en la carpeta dabatabes/seeds. Antes de editarla, añadiremos la clase DatabaseSeeder que se ejecutará cuando ejecutemos nuestros sembradores:

```
// database/seeds/DatabaseSeeder.php
...
public function run()
{
$this->call(ContactsTableSeeder::class);
}
```

Ahora editaremos el sembrador. La cosa más simple que podemos hacer es es insertar manualmente registros:

```
<?php
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;
class ContactsTableSeeder extends Seeder
{
public function run()
{
DB::table('contacts')->insert([
'name' => 'Lupita Smith'
'email' => 'lupita@gmail.com',
]);
}
}
```

Esto insertará un solo registro, lo que es un buen comienzo. Pero para semillas verdaderamente funcionales, probablemente querrás recorrer algún tipo de generador aleatorio y ejecutar este insert() muchas veces ¿no?

Las fábricas de modelos definen uno (o más) patrones para crear entradas falsas para su base de datos. De forma predeterminada, llevan el nombre de una clase Eloquent, pero también puedes simplemente asígnales el nombre después del nombre de la tabla si no vas a trabajar con Eloquent. Aquí esta la misma tabla configurada en ambos sentidos:

```
$factory->define(User::class, function (Faker\Generator $faker) {
return [
'name' => $faker->name,
];
});
$factory->define('users', function (Faker\Generator $faker) {
return [
'name' => $faker->name,
];
});
```

Las fábricas de modelos se definen en la base de datos/factories/ModelFactory.php. Cada fábrica tiene un nombre y una definición de cómo crear una nueva instancia de la clase definida. El método $factory->define() toma el nombre de la fábrica como primer parámetro y un cierre
que se ejecuta para cada generación como segundo parámetro. La fábrica más simple que podríamos definir podría verse así:

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
return [
'name' => 'Lupita Smith',
'email' => 'lupita@gmail.com',
];
});
```

Ahora podemos usar el asistente global factory() para crear una instancia de Contact en nuestro siembra y prueba:

```
// Create one
$contact = factory(Contact::class)->create();
// Create many
factory(Contact::class, 20)->create();
```

Sin embargo, si usáramos esa fábrica para crear 20 contactos, los 20 tendrían el mismo información. Eso es menos útil. Obtendremos aún más beneficios de las fábricas cuando aprovechemos la instancia de Faker que pasó al cierre; Faker facilita la aleatorización del creación de datos falsos estructurados. El ejemplo anterior ahora se convierte en este:

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
return [
'name' => $faker->name,
'email' => $faker->email,
];
});
```

Ahora, cada vez que creamos un contacto falso usando esta fábrica de modelos, todas sus propiedades serán únicas.

Construyendo consultas

Ahora que está conectado y ha migrado y sembrado sus tablas, comencemos con cómo utilizar las herramientas de base de datos. En el centro de cada pieza de Laravel tenemos la funcionalidad de la base de datos es el generador de consultas, una interfaz fluida para interactuar con su base de datos. La arquitectura de la base de datos de Laravel puede conectarse a MySQL, Postgres, SQLite y SQL Server a través de una única interfaz, con sólo cambiar algunos ajustes de configuración.

Antes de comenzar a crear consultas complejas con encadenamiento del método Fluent, tomemos algunos comandos de base de datos de muestra. La clase de la base de datos se utiliza tanto para consultas y encadenamiento de constructores y para consultas sin procesar más simples, como se ilustra en el ejemplo:

```
// basic statement
DB::statement('drop table users’)

// raw select, and parameter binding
DB::select('select * from contacts where validated = ?', [true]);

// select using the fluent builder
$users = DB::table(‘users')->get();

// joins and other complex calls
DB::table('users')
->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')
        ->where('contacts.type', 'donor');
}) ->get();
```

Como vimos en el ejemplo, es posible realizar cualquier llamada sin formato a la base de datos utilizando la clase de base de datos y el método state(): DB::statement('declaración SQL aquí'). Pero también existen métodos específicos para varias acciones comunes: select(), insert(), update() y delete(). Estas todavía son decisiones sin elaborar, pero hay diferencias. Primero, usar update() y delete() devolverá el número de filas afectadas, mientras que statement() no lo hará; En segundo lugar, con estos métodos queda más claro para los futuros desarrolladores exactamente qué tipo de declaración estás haciendo.

La forma más sencilla del método DB es select(). Puede ejecutarse sin ningún parámetro adicional:

```
$users = DB::select('select * from users');
```

La arquitectura de la base de datos de Laravel permite el uso del enlace de parámetros PDO, que protege sus consultas de posibles ataques SQL. Pasar un parámetro a una declaración es tan simple como reemplazar el valor en su declaración con ?, luego agregar el valor a el segundo parámetro de su llamada:

```
$usersOfType = DB::select(
'select * from users where type = ?',
[$type]
);
```

A partir de aquí, todos los comandos sin formato se ven más o menos iguales. Las inserciones aparecen así:

```
DB::insert(
'insert into contacts (name, email) values (?, ?)',
['sally', 'sally@me.com']
);
```

Updates son así:

```
$countUpdated = DB::update(
'update contacts set status = ? where id = ?',
['donor', $id]
);
```

Y el borrado:

```
$countDeleted = DB::delete(
'delete from contacts where archived = ?',
[true]
);
```

Hasta ahora, no hemos utilizado el generador de consultas per se. Acabamos de usar simples llamadas a métodos en la clase de la base de datos. De hecho, creamos algunas consultas.
El generador de consultas permite encadenar métodos para, como habrás adivinado, construir una consulta. Al final de tu cadena usarás algún método (probablemente get()) para desencadenar la ejecución real de la consulta que acaba de crear.

select(): permite elegir las columnas que vas a seleccionar

```
$emails = DB::table('contacts')
->select('email', 'email2 as second_email')
->get();
```

where(): te permite limitar el alcance de lo que se devuelve usando WHERE. Por defecto, la firma del método Where() es que toma tres parámetros: la columna,el operador de comparación y el valor:

```
$newContacts = DB::table('contact')
->where('created_at', '>', Carbon::now()->subDay())
->get();
```

En el caso de que el comparador sea un igual, podemos hacerlo así:

```
$vipContacts = DB::table(‘contacts')->where('vip',true)->get();.
```

orWhere(): permite establecer comparaciones OR

```
$priorityContacts = DB::table('contacts')
->where('vip', true)
->orWhere('created_at', '>', Carbon::now()->subDay())
->get();
```

whereBetween(colName, [low, high]): permite hacer una consulta donde una columna debe estar entre low y high

```
$mediumDrinks = DB::table('drinks')
->whereBetween('size', [6, 12])
->get();
```

whereIn(colName, [1, 2, 3]): permite hacer una comparación donde un campo debe tener alguno de los valores de una lista

```
$closeBy = DB::table('contacts')
->whereIn('state', ['FL', 'GA', 'AL'])
->get();
```

whereNull(colName) and whereNotNull(colName): permite hacer una comparación del valor de una columna con NULL

whereRaw(): permite pasar la condición WHERE en crudo, sin usar métodos

```
$goofs = DB::table('contacts')->whereRaw('id = 12345’)->get().
```

orderBy(colName, direction): ordena el resultado. El segundo parámetro puede ser ASC o DESC

```
$contacts = DB::table('contacts')
->orderBy('last_name', 'asc')
->get();
```

groupBy() y having() o havingRaw(): agrupa los resultados por una columna. Having permite filtrar por el criterio de ordenación.

```
$populousCities = DB::table('contacts')
->groupBy('city')
->havingRaw('count(contact_id) > 30')
->get();
```

skip() and take(): muy usados en paginadores, permiten definir cuántos filas deben devolverse y cuántas se van a saltar, como un número de página y un número de resultados por página

```
$page4 = DB::table(‘contacts')->skip(30)->take(10)->get();
```

get(): toma los resultados de la consulta

```
$contacts = DB::table(‘contacts')->get();
```

```
$vipContacts = DB::table('contacts')->where('vip', true)->get();
```

first() y firstOrFail(): toma solo un resultado, como el get() pero con LIMIT 1 añadido

```
$newestContact = DB::table('contacts')
->orderBy('created_at', 'desc')
->first();
```

find(id) y findOrFail(id): como first(), pero pasando la primary key del elemento a buscar.

```
$contactFive = DB::table(‘contacts')->find(5);
```

count(): retorna el total de los elementos que cumplen la condición

```
$countVips = DB::table('contacts')
->where('vip', true)
->count();
```

min() y max(): devuelve el valor máximo y mínimo de una columna particular

```
$highestCost = DB::table('orders')->max('amount');
```

sum() and avg(): devuelve la suma o la media de la columna seleccionada.

```
$averageCost = DB::table('orders')
->where('status', 'completed')
->avg(‘amount');
```

Para hacer una consulta con joins, left joins o right joins, lo usamos como en el siguiente ejemplo:

```
$users = DB::table('users')
->join('contacts', 'users.id', '=', 'contacts.user_id')
->select('users.*', 'contacts.name', 'contacts.status')
->get();
```

Inserciones

El método insert() es muy sencillo. Le pasamos un array para insertar una sola fila o un arrays de arrays para insertar múltiples filas, y usamos insertGetID() en lugar de insert() para tomar el valor de la clave primaria autoincremental como valor retornado.

```
$id = DB::table('contacts')->insertGetId([
'name' => 'Abe Thomas',
'email' => 'athomas1987@gmail.com',
]);

DB::table('contacts')->insert([
['name' => 'Tamika Johnson', 'email' => ‘tamikaj@gmail.com'],
['name' => 'Jim Patterson', 'email' => ‘james.patterson@hotmail.com'],
]);
```

Actualizaciones

Las actualizaciones son también muy simples:

```
DB::table('contacts')
->where('points', '>', 100)
->update(['status' => 'vip']);
```

Borrados

Son también muy simples:

```
DB::table('users')
->where('last_login', '<', Carbon::now()->subYear())
->delete();
```

Introducción a Eloquent

Eloquent es un ORM ActiveRecord, lo que significa que es una capa de abstracción de base de datos que proporciona una única interfaz para interactuar con múltiples tipos de bases de datos. “Active Record” significa que una única clase Eloquent es responsable no sólo de proporcionar la capacidad de interactuar con la tabla como un todo (por ejemplo, User::all() obtiene todos los usuarios), sino que también representa una fila de tabla individual (por ejemplo, $sharon = new User). Además, cada instancia es capaz de gestionar su propia persistencia; puedes llamar a $sharon->save() o $sharon->delete().

Eloquent se centra principalmente en la simplicidad y, como el resto del marco, se basa en la “convención sobre la configuración” para permitirle construir modelos potentes con código mínimo.

```
<?php
use Illuminate\Database\Eloquent\Model;
class Contact extends Model {}

public function save(Request $request)
{
// Create and save a new contact from user input
$contact = new Contact();
$contact->first_name = $request->input('first_name');
$contact->last_name = $request->input('last_name');
$contact->email = $request->input('email');
$contact->save();
return redirect('contacts');
}
public function show($contactId)
{
// Return a JSON representation of a Contact based on a URL segment;
// if the contact doesn't exist, throw an exception
return Contact::findOrFail($contactId);
}
public function vips()
{
// Unnecessarily complex example, but still possible with basic Eloquent
// class; adds a "formalName" property to every VIP entry
return Contact::where('vip', true)->get()->map(function ($contact) {
$contact->formalName = "The exalted {$contact->first_name} of the
{$contact->last_name}s";
return $contact;
});
}
```

¿Cómo? Convención. Eloquent asume el nombre de la tabla (el contacto se convierte en contactos), y con eso tienes un modelo Eloquent completamente funcional.

Creando y definiendo modelos Eloquent

Primero, creamos un modelo usando Artisan:

php artisan make:model Contact

Esto es lo que aparecerá en app/Contact.php

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Contact extends Model
{
//
}
```

El comportamiento predeterminado para los nombres de tablas es que Laravel hace “csnakew case" y pluraliza su nombre de clase, por lo que SecondaryContact accedería a una tabla llamada second_contacts. Si desea personalizar el nombre, establezca la propiedad $table explícitamente en el modelo:

```
protected $table = ‘contacts_secondary’;
```

Laravel asume que por defecto cada tabla tiene una clave primaria que es entera y autoincremental, a la qiue llamará id. Si quiere cambiar el nombre de la clave primaria, puede cambiarla:

```
protected $primaryKey = ‘contact_id'
```

Obteniendo datos con Eloquent

Empecemos obteniéndolo todo:

```
$allContacts = Contact::all();
```

Es fácil. Podemos filtrarlo un poco:

```
$vipContacts = Contact::where('vip', true)->get();
```

Podemos encadenar las restricciones:

```
$newestContacts = Contact::orderBy('created_at', 'desc')
->take(10)
->get();
```

Las agregaciones están también permitidas en Eloquent:

```
$countVips = Contact::where('vip', true)->count();
$sumVotes = Contact::sum('votes');
$averageSkill = User::avg(‘skill_level');
```

Para insertar un dato en Eloquent, tenemos dos maneras de hacerlo: creando una nueva instancia de la clase Eloquent, establecer las propiedades y grabar

```
$contact = new Contact;
$contact->name = 'Ken Hirata';
$contact->email = 'ken@hirata.com';
$contact->save();
```

O, de la misma forma:

```
$contact = new Contact([
'name' => 'Ken Hirata',
'email' => ‘ken@hirata.com'
]);
$contact->save();
```

La otra manera de insertarlo es pasando los parámetros a create():

```
$contact = Contact::create([
'name' => 'Keahi Hale’,
]);
```

Actualizar registros es muy parecido a crearlos:

```
$contact = Contact::find(1);
$contact->email = 'natalie@parkfamily.com';
$contact->save();
```

La manera más sencilla de borrar un registro es:

```
$contact = Contact::find(5);
$contact->delete();
```

También puede borrar así:

```
Contact::destroy(1);
// or
Contact::destroy([1, 5, 7]);
```

Scopes

Normalmente, vamos a escribir los filtrados en nuestras queries de manera manual y normal, pero si pensamos que tenemos un filtrado que repetimos una y otra vez, debería ser posible escribirlo una sola vez y usarlo muchas veces. Eso es un scope. Por ejemplo:

```
$activeVips = Contact::where('vip', true)->where('trial', false)->get();
```

No etaría mal poder escribir:

```
$activeVips = Contact::activeVips()->get();
```

Podemos definir un scope local de una manera fácil:

```
class Contact
{
public function scopeActiveVips($query)
{
return $query->where('vip', true)->where('trial', false);
}
```

Para definir un scope local, agregamos un método a la clase Eloquent que comienza con “scope” y luego contiene la versión en mayúsculas y minúsculas del nombre del alcance. Este método pasa un generador de consultas y necesita devolver un generador de consultas, pero, por supuesto, puede modificar la consulta antes de regresar; ese es el punto.

Podemos definir scopes locales que admitan parámetros:

```
class Contact
{
public function scopeStatus($query, $status)
{
return $query->where('status', $status);
}
```

Y podemos usarlo del mismo modo, pasando el parámetro al scope:

```
$friends = Contact::status('friend')->get();
```

Eloquent ORM
- Introducción a Eloquent ORM
- Definición de modelos
- Consultas y relaciones en Eloquent

Formularios y Validación
- Creación y validación de formularios
- Uso de form requests
- Mensajes de error personalizados

Autenticación y autorización
- Configuración de autenticación
- Protección de rutas y recursos
- Creación de roles y permisos

APIs RESTful
- Creación de APIs RESTful en Laravel
- Autenticación de API
- Documentación de API con herramientas como Swagger

Tareas en segundo plano y programación de tareas
- Colas y trabajos en segundo plano
- Programación de tareas con Laravel Scheduler
- Uso de supervisor para administrar colas

Pruebas en Laravel
- Tipos de pruebas en Laravel
- Creación y ejecución de pruebas unitarias y de integración
- Uso de PHPUnit y herramientas de pruebas en Laravel