**Grupo de rutas**

Los grupos de rutas en Laravel son una herramienta poderosa para compartir atributos de ruta, como el middleware, entre un conjunto de rutas, lo que evita tener que definir estos atributos en cada ruta individual. Además, los grupos anidados pueden fusionar de forma inteligente los atributos con su grupo principal, incluyendo middleware y condiciones "where", mientras que agregan nombres y prefijos automáticamente.

Por ejemplo, para asignar middleware a todas las rutas dentro de un grupo, simplemente usa el método middleware antes de definir el grupo. El middleware se ejecuta en el orden en que aparecen en la matriz

Route::middleware(['first', 'second'])->group(function () {

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

// Usa los middlewares first y second...

});

Route::get('/user/profile', function () {

// Usa los middlewares first y second...

});

});

También puedes definir un controlador común para todas las rutas dentro de un grupo usando el método **controller**:

use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {

Route::get('/orders/{id}', 'show');

Route::post('/orders', 'store');

});

Además, los grupos de rutas también se pueden usar para manejar el enrutamiento de subdominios. A los subdominios se les pueden asignar parámetros de ruta, lo que le permite capturar una parte del subdominio para usar en su ruta o controlador. Para hacer esto, llama al método **domain** antes de definir el grupo:

Route::domain('{account}.example.com')->group(function () {

Route::get('user/{id}', function (string $account, string $id) {

// ...

});

});

El método **prefix** se puede usar para prefijar cada ruta en el grupo con un URI determinado, por ejemplo, puedes prefijar todos los URI de ruta dentro del grupo con "admin":

Route::prefix('admin')->group(function () {

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

// Coincide con la URL "/admin/users"

});

});

Finalmente, el método **name** se puede usar para prefijar cada nombre de ruta en el grupo con una cadena dada. Esto es útil para, por ejemplo, anteponer los nombres de todas las rutas del grupo con "admin":

Route::name('admin.')->group(function () {

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

// Asigna el nombre de ruta "admin.users"...

})->name('users');

});

**Como crear y registrar un provider**

**php artisan make:provider ViewServiceProvider**

Para escribir un proveedor de servicios en Laravel, debemos crear una nueva clase que implemente la Illuminate\Support\ServiceProvider interfaz. Esta interfaz define dos métodos que debemos implementar: register() y boot().

El método register() se usa para enlazar cosas en el contenedor de servicios de Laravel. Por ejemplo, podemos enlazar una instancia de una clase en el contenedor, lo que nos permitirá acceder a esa instancia en cualquier lugar de nuestra aplicación.

El método boot() se usa para realizar cualquier configuración que deba hacerse después de que se hayan registrado los enlaces del contenedor. Esto podría incluir la definición de rutas, la publicación de activos o la configuración de middleware.

Para registrar un proveedor de servicios con nuestra aplicación Laravel, debemos agregar la clase del proveedor a la providersmatriz en el config/app.phparchivo. Luego, cada vez que nuestra aplicación se inicie, se llamará automáticamente al register() y boot() métodos en nuestra clase de proveedor.

En resumen, los proveedores de servicios son una forma poderosa de extender y personalizar su aplicación Laravel. Al escribir sus propios proveedores de servicios, puede enlazar sus propias clases en el contenedor de servicios y realizar cualquier configuración necesaria para su aplicación.

A veces, es necesario compartir datos con todas las vistas generadas por su aplicación. Para hacer esto, puede utilizar el método **share** de la fachada **View**. Por lo general, debe realizar llamadas al método **share** dentro del método **boot** de un proveedor de servicios. Puede agregarlos a la clase **App\Providers\AppServiceProvider** o generar un proveedor de servicios separado para alojarlos.

namespace App\Providers;

use Illuminate\Support\Facades\View;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

/\*\*

\* Register any application services.

\*/

public function register()

{

// ...

}

/\*\*

\* Bootstrap any application services.

\*/

public function boot()

{

View::share('key', 'value');

}

}

**Vistas Blade**

**Variable $loop**

Al utilizar un bucle foreach en Laravel, se crea automáticamente una variable de bucle llamada $loop que proporciona información valiosa sobre la iteración actual. Con esta variable, se puede acceder al índice de la iteración actual, así como verificar si es la primera o la última iteración del ciclo con las propiedades $loop->first y $loop->last respectivamente.

En el siguiente ejemplo, se utiliza el bucle foreach para iterar sobre una matriz de usuarios y mostrar su ID en un párrafo HTML.

@foreach ($users as $user)

@if ($loop->first)

This is the first iteration.

@endif

@if ($loop->last)

This is the last iteration.

@endif

<p>This is user {{ $user->id }}</p>

@endforeach

Si se está en un bucle anidado, se puede acceder a la variable $loop del bucle principal a través de la propiedad "parent". En el siguiente ejemplo, se utiliza un bucle anidado para iterar sobre los posts de cada usuario.

@foreach ($users as $user)

@foreach ($user->posts as $post)

@if ($loop->parent->first)

This is the first iteration of the parent loop.

@endif

@endforeach

@endforeach

Además, la variable $loop contiene otras propiedades útiles como $loop->index, que devuelve el índice de la iteración actual, $loop->iteration, que devuelve la iteración actual, $loop->remaining, que devuelve las iteraciones restantes en el ciclo, $loop->count, que devuelve el número total de elementos en la matriz que se está iterando, $loop->even, que devuelve true si se trata de una iteración par y $loop->odd, que devuelve true si es una iteración impar. Finalmente, $loop->depth devuelve el nivel de anidamiento del bucle actual.

**Componentes de clase**

Los componentes de clase en Laravel 10 son una herramienta útil para encapsular la funcionalidad de la vista en clases reutilizables y fáciles de mantener. En lugar de definir toda la lógica de presentación en una vista Blade, puedes crear un componente de clase que maneje la lógica y la reutilice en varias vistas.

Para crear un componente de clase en Laravel 10, debes crear una nueva clase que extienda la clase base de componentes de Laravel, que es **Illuminate\View\Component**. Esta clase debe definir un método **render** que devuelva la vista Blade que representa el componente.

Creando un componente **php artisan make:component Alert**

Por ejemplo, supongamos que quieres crear un componente de clase para mostrar una lista de tareas. Primero, puedes crear una nueva clase llamada **TaskList** que extienda la clase base de componentes de Laravel:

namespace App\View\Components;

use Illuminate\View\Component;

class TaskList extends Component

{

public function render()

{

return view('components.task-list');

}

}

Luego, puedes crear una vista Blade llamada 'task-list.blade.php

<ul>

@foreach ($tasks as $task)

<li>{{ $task->name }}</li>

@endforeach

</ul>

Finalmente, puedes incluir el componente de clase en una vista Blade existente llamando a la directiva 'x

<x-task-list :tasks="$tasks" />

**Componentes como plantilla**

En Laravel 10, los componentes como plantilla son una forma de definir plantillas de vista reutilizables que pueden ser utilizadas en toda tu aplicación. Puedes pensar en ellos como plantillas que contienen la estructura HTML y la lógica de presentación necesarias para representar un conjunto específico de datos.

php artisan make:component MiPlantilla

**Query Builder**

Consultar si ese conjunto de id se encuentran en la colección

return DB::table('users')

            ->whereIn('id', [1,2,5])

            ->get();

/traer registros por fecha

    return DB::table('users')

           ->whereDate('created\_at', '2023-10-09')

           ->get();

//traer registros por mes

    return DB::table('users')

           ->whereMonth('created\_at', '11')

           ->get();

//traer registros por día

    return DB::table('users')

           ->whereDay('created\_at', '8')

           ->get();

//traer registros por año

    return DB::table('users')

           ->whereYear('created\_at', '2022')

           ->get();

//traer registros por hora

    return DB::table('users')

           ->whereTime('created\_at', '16:24:31')

           ->get();

//comparar si dos fechas son iguales

    //trae los registros cuya fecha sea igual

    return DB::table('users')

           ->whereColumn('created\_at', 'updated\_at')

           ->get();

//agrupación lógica

    return DB::table('users')

        ->where('id', '>=', 10)

        ->where(function ($query) {

            $query->where('email', 'like', '%@example.org%')

                ->orwhere('email', 'like', '%@example.net%');

        })

        ->get();

//condicionales

    $prueba = true;

    return DB::table('users')

        ->when($prueba, function ($query) {

            $query->where('id', '>=', 10);

        })

        ->get();

//agregar usuarios

    DB::table('users')->insertOrIgnore(

        [

            'name' => 'Gerson Calvo',

            'email' => 'gedacaba@gmail.com',

            'password' => bcrypt('12345678')

        ],

        [

            'name' => 'Moster',

            'email' => 'jalvarez@gmail.com',

            'password' => bcrypt('12345678')

        ]

    );