JAVASCRIPT

Apuntes de estudio

fecha inicio de apuntes: 27 de noviembre de 2020

Primera Parte: lo básico

Técnicas de escritura de código

Para tener en cuenta

- **snake_case**: es usada para nombrar archivos.

```
mi_archivo_javascript.js
```

- **UPPER_CASE**: Se recomienda para definir Constantes.

```
const UNA_CONSTANTE = 'esto es una constante';
```

- **UpperCamelCase**: su uso es recomendado para nombrar clases - cada inicial va mayúscula.

```
class SerHumano = {
    ...
}
```

- **lowerCamelCase**: se usa para declarar objetos - primitivos - funciones - instancias; la primera letra en minúscula y luego cada inicial en mayúscula.

`objetos`

```
const unObjeto ={
    ...
}
```

`primitivos`

```
let unaCadena = "texto";
```

`funciones`

```
function unaPrueba (parametros){
    ...
}
```

ORDENAMIENTO DE CODIGO

1. Importación de Módulos.
2. Declaración de variables.
3. Declaración de funciones.
4. Ejecución de código.

TIPOS DE DATOS

1. `PRIMITIVOS` : Se accede directamente al valor
    - string
    - number
    - boolean
    - null
    - undefined
    - NaN
2. `COMPUESTOS` : se accede a la referencia del valor.
    - object= {}
    - array = []
    - function (){}
    - class {}
    - ...

sección : 3 - 4 || Script de base: basicos.js

Variables

las variables con scope funcional (ambito de bloque) se declaran con la palabra reservada `let`, y se declaran con `var` para que su scope sea global.

```
let saludo = 'hola'
```

las variables globales se definen en el objeto `window` de los navegadores o `global` en node.js, por lo tanto se considera mala practica.

Constantes

Se crean "variables" que no cambian en el transcurso de la aplicación, se declara con la palabra reservada `const`.

Tambien se puede usar para la definicion de datos compuestos (objetos y arreglos), debido a que se accede a la referencia del valor.

Ejemplo

`objeto`

```
const persona = {
    nombre : 'Andres',
    apellido : 'Gomez'
}
```

`array`

```
const colores = [
    'blanco',
    'negro',
    'azul'
]
```

sección : 5 || Script de base: basicos.js

Cadenas de texto (strings)

El objeto String se utiliza para representar y manipular una secuencia de caracteres dentro de una cadena.

Las cadenas son útiles para almacenar datos que se pueden representar en forma de texto.

Algunas de las operaciones más utilizadas en cadenas son verificar su "length", para construirlas y concatenarlas usando operadores de cadena + y +=, verificando la existencia o ubicación de subcadenas con `indexOf()` o extraer subcadenas con el método `substring()`.

Nota

los objetos tienen dos atributos importantes:

- las `propiedades` : atributos que dan información del objeto, caracteristicas que definen el objeto.
- Los `metodos` : Acciones que hace el objeto, todos lo metodos terminan con parentesis `()`

> en la seccion de funciones se detalla acerca de este tema

Crear cadenas

Las cadenas se pueden crear como primitivas, a partir de cadena literales o como objetos, usando el constructor `String()`

variaciones de uso

```
const string1 = "Una cadena primitiva",
const string2 = 'tambien un cadena prmitiva',
const string3 = `cadena primitiva, plantilla literal-interpolar expresiones`,
const string4 = new String("Un objeto string")
```

PROPIEDADES

La propiedad `.length` de un objeto `String` representa la longitud de una cadena

METODOS

Algunos de los métodos mas usados en Javascript son:

`toUpperCase()` Devuelve el valor de la cadena que lo llama convertido en mayúsculas.

`includes()` El metodo deternmina si una cadena de texto puede ser encontrada dentro de otra cadena de texto, devolviendo `true` o `false` según corresponda.

`trim()` Elimina los espacios en blanco en ambos extremos del string.

`split()` Divide un objeto de tipo `String` en un array(vector-arreglo) de cadenas mediante la separación de la cadena en subcadenas.

Sintaxis

```
cadena.split([separador][,limite])
```

separador: Especificael caracter a usar para la separación de la cadena.

limite: Opcional, Entero que especifica un limite sobre el número de divisiones a realizar

Ejemplo

uso de la propiedad `.length`

```
let nombre = 'Roberto';
Console.log(nombre.length);
```

```
| 7
```

Metodo `toUpperCase()`

```
let apellido = 'Rubiano';
Console.log(apellido.toUpperCase());
```

```
| RUBIANO
```

Metodo `includes()`

```
let oracion = "Lorem ipsum dolor sit amet consectetur adipisicing elit.";
console.log(oracion.includes("Lorem"));
```

```
true
```

Notación de scape

Los caracteres especiales se pueden codificar mediante notación de escape

Código	Descripción
'	Comilla Sencilla
"	Comilla Doble
\	Barra Inversa
\n	Nueva Linea
\v	Tabulación vertical
\t	Tabulación
\b	Retroceso
\f	Avance de página

Cuando las cadenas literales son muy largas se pueden dividir en varias lineas de codigo usando dos alternativas:

1. operador (+)

```
let cadenaLarga = "Esta es una cadena larga y necesita" +
                  "ser dividida en varias lineas para" +
                  "que sea mas legible"
```

2. barra invertida (\)

```
let cadenaLarga2 = "Esta es una cadena larga y necesita \
                  "ser dividida en varias lineas para \
                  "que sea mas legible"
```

caracteristicas de las cadenas de texto: concatenacion e interpolacion

Template strings

Las plantillas literales son cadenas literales que habilitan el uso de expresiones incrustadas. Con ellas es posible utilizar cadenas de caracteres de mas dse una linea y funcionalidades de interpolación.

se delimitan por tildes invertidas `` ` `` (acento grave).

Concatena las partes y forma una única cadena de caracteres.

En las plantillas literales se pueden insertar expresiones con `${}` e incluir caracteres de fin de linea literales

```
escritura del templates strings

`texto en una cadena de caracteres`;

`linea 1 de la cadena de caracteres
linea 2  de la cadena de caracteres`;

`texto de cadena ${expresion} texto adicional`;

etiqueta`texto de cadena ${expresion} texto adicional`;
```

uso de expresiones en el template

```
let a = 5;
let b = 10;
console.log(`Quince es ${a + b} y
no ${2 * a + b}.`);
```

```
| // Quince es 15 y
```

```
| // no 20
```

Tipos de Datos compuestos

Funciones

Una funcion es un fragmento de código que puede ser llamado por otro código o por si mismon, o por una variable que haga referencia a la función. Cuando se llama a una función, los parámetros se pasan a una funcion como entrada y la funcion puede devolver opcionalmente una salida. Las funciones se pueden ejecutar en cualquier momento

Una función en javaScript es también un objeto.

tipos de funcion

Funcion anónima

Son funciones que no tienen nombre

```
function(){
    cuerpo de la función
};
// se puede usar la notación con flecha o arrow function, ver mas adelante
```

==========

Inicio en consola

instalacion

```
composer global require laravel/installer
```

iniciar proyecto

```
laravel new nombreProyecto
```

Rutas

Las rutas más básicas de Laravel aceptan un URI y un Cierre `(closure)`, proporcionando un método muy simple y expresivo para definir rutas

las rutas son las url de entrada de la aplicacion, hay 2 tipos principales de rutas web y api.

Notas

la carpeta que contiene todas las rutas de la aplicación se llama `routes` está ubicada en el directorio raiz

para generar las rutas se hace a través de la llamada del `metodo`

```
Route::get()
```

las rutas corresponden a un verbo del `HTTP`, que es el nombre del método que se invoca sobre la clase `Route`.
El metodo recibe dos parámetros:

- *primer parámetro:* el patrón que debe cumplirse para que esa ruta se active `"/"` (el "home" de la aplicación), puede ser desde una simple cadena o más complejo generando partes de la ruta que sean parámetros variables.
- *segundo parámetro:* se indica una función con el código a ejecutar cuando laravel tenga que procesarla

Sintaxis de parametros del metodo

('**parametro_patrón**', función anónima o **closure**)

> lo retornado en el closure es lo que se presenta para el usuario

para definir que la ruta responda al ingreso de la raiz se hace:

```
Route::get('/', function)
    //ejemplo
    creaando.com => Route::get('/', function)
    creaando.com/contacto = Route::get ('contacto', function)
```

los tipo de peticiones son:

Route::get()
Route::post() = para el envío de formulario // form action=POST
Route::put()
Route::patch()
Route::delete()

verbos HTTP

Sirven para decir el tipo de acción que quieres realizar con un recurso (la URL), siendo posible especificar en el protocolo (la formalidad de la conexión por HTTP entre distintas máquinas) el verbo o acción que deseamos realizar.

Son 8 verbos en el protocolo: Head, Get, Post, Put, Delete, Trace, Options, Connect.

Get es la acción que se usa en el protocolo habitualmente, cuando se consulta un recurso. Sirve para recuperar información. Post por su parte es la acción que se realiza cuando se mandan datos, de un formulario generalmente. Los otros verbos no se usan de una manera muy habitual, pero sí se han dado utilidad en el desarrollo de lo que se conoce como API REST.

Verbos HTTP

significado de los verbos

- **Get**: recuperar información, podemos enviar datos para indicar qué se desea recuperar, pero mediante get en principio no se debería generar nada, ningún tipo de recurso en el servidor o aplicación, porque los datos se verán en la URL y puede ser inseguro.
- **Post**: enviar datos que se indicarán en la propia. Esos datos no se verán en la solicitud, puesto que viajan con la información del protocolo.

- **Put**: esto sirve para enviar un recurso, subir archivos al servidor, por ejemplo. No está activo en muchas configuraciones de servidores web. Con put se supone que los datos que estamos enviando son para que se cree algún tipo de recurso en el servidor.
- **Delete**: borrado de algo.
- ***Trace, patch, link, unlink, options y connect*** no están entre los verbos comunes de Laravel

Para tener en cuenta

observar las rutas posibles de la aplicación

el siguiente comando `artisan` sirve para mostrar todas las rutas disponibles en un proyecto o aplicacion con laravel

```
php artisan route:list
```

metodo corto

```
php artisan r:l
```

Rutas con parámetros

Las rutas en las aplicaciones web corresponden con patrones en los que en algunas ocasiones se encuentran textos fijos y en otras ocasiones textos que van a ser variables. en el presente se refiere a los textos variables con el nombre de parámetros y se pueden producir con una sintaxis de llaves.

para pasar paramentros en las urls, se usan las llaves `{}`

ejemplos de rutas con parámetros:

```
example.com/coloboradores/jose
example.com/coloboradores/luis
example.com/categoria/php
example.com/categoria/php/2
example.com/tienda/prodcutos/28
```

crear rutas con parámetros

en el siguiente ejemplo se crean unas de las rutas anteriores con el sistema de `routing` de laravel.

```
Route:get('colaboradores/{nombre}', function($nombre){
    return "mostrando el colaborador $nombre";
});

Route::get('tienda/productos/{id}', function($id_producto){
    return "Mostrando el producto $id_producto de la tienda";
});
```

Ejemplo Adicional

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

Parámetros Opcionales

al no pasar parametros arroja error 404, si no se quiere parametro obligatorio, se asigna un signo de interrogación `(?)` y se da un valor por defecto en los parametros de la funcion

> Los parámetros opcionales se indican con un símbolo de interrogación `?`

Tomando como base del ejemplo anterior

```
example.com/categoria/php
example.com/categoria/php/2
```

En este código en el patrón de la URI se observa que la página es opcional.

```
Route::get('categoria/{categoria}/{pag?}', function($categoria, $pag = 1){
    return "Viendo categoría $categoria y página $pag";
});
```

Ejemplo Adicional

```
Route::get('saludo/{nombre?}', function($nombre = "invitado"){
    return "saludo".$nombre;
});
```

Precedencia de las rutas

Con dos rutas registradas que tienen patrones distintos, si por un casual una URI puede encajar en el formato definido por ambos patrones, el que se ejecutará será el que primero se haya escrito en el archivo `routes.php.`

Ejemplo de Precedencia

```
Route::get('categoria/{categoria}', function($categoria){
return "Ruta 1- Viendo categoría $categoria y no recibo página";
});

Route::get('categoria/{categoria}/{pagina?}', function($categoria, $pagina=1){
    return "Ruta 2 - Viendo categoría $categoria y página $pagina";
});
```

En esas dos rutas tenemos patrones diferentes de URI, pero si alguien escribe:

```
example.com/categoria/laravel/
```

Esa URL podría casar con ambos patrones de URI. En este caso, el mensaje que obtendremos será:

```
Ruta 1 - Viendo categoría laravel y no recibo página
```

Aceptar determinados valores de parámetros

Hay una posibilidad muy útil con los parámetros de las rutas que consiste en definir expresiones regulares para especificar qué tipo de valores se acepta en los parámetros.

Por ejemplo, un identificador de producto debe ser un valor numérico o un nombre de un colaborador debe aceptar solamente caracteres alfabéticos.

Si se ha entendido esta situación se observará que hasta el momento, tal como hemos registrado las rutas, se aceptarían todo tipo de valores en los parámetros, generando rutas que muchas veces no deberían devolver un valor de página encontrada. Por ejemplo:

Ejemplo de valores en parámetros

```
example.com/colaboradores/666
example.com/tienda/productos/kkk
```

colaboradores debería ser un valor de tipo alfabético y el identificador de producto solo puede ser numérico.

Así que se debe modificar las rutas para poder agregarle el código que nos permita no aceptar valores que no deseamos.

```
Route::get('colaboradores/{nombre}', function($nombre){
    return "Mostrando el colaborador $nombre";
})->where(array('nombre' => '[a-zA-Z]+'));
```

Como se puede ver, se le coloca en cadena, sobre la ruta generada, un método `where()` que permite especificar en un *array* todas las reglas que se deben aplicar a cada uno de los parámetros a restringir.

Rutas con Nombre // named Routes

para una localizacion mas rapida y posterior edicion se le agregan nombres a las rutas

Ejemplo de Nombre de ruta

```
Route::get('contactanos', function(){
    return "sección de contactos";
})->name('contacto); //nombre de ruta
```

y se llama desde la ruta

```
Route::get('/', function(){
    echo "<a href='".route('contacto')."'>Contactos</a>";
})
```

vistas

Las vistas son una de las capas que tiene el sistema `MVC`, que trata de la separación del código según sus responsabilidades. En este caso, las vistas mantienen el código de lo que sería la capa de presentación.

Como capa de presentación, las vistas se encargan de realizar la salida de la aplicación que generalmente en el caso de PHP será código HTML. Por tanto, una vista será un archivo PHP que contendrá mayoritariamente código HTML, que se enviará al navegador para que éste renderice la salida para el usuario.

Notas

las vistas se encuentran en: `"resources/views"`

se usa la extension `".blade.php"` para hacer referencia a las vistas

por lo tanto la forma correcta de mostrar el html al usuario es retornando las vistas

para esto se invoca a la funcion `view()` dentro del `closure` y se le indica el nombre de la vista a mostrar

Ejemplo

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

la funcion asume que las vistas se encuentran en la carpeta `"view"` y con extension `.blade.php`, por lo tanto no es necesario agregar la direccion completa `"resource/views/welcome.blade.php"`

`view()` es una funcion global, un helper global en laravel, que se encarga de cargar una vista y devolver una salida producida por ella

pasar datos a las vistas

Para pasar datos a las vistas, se hace desde el archivo de rutas `web.php` o su equivalente en el proyecto

Ejemplo de datos en las vistas

por medio del parámetro `with`

```
Route::get('/', function(){
    $nombre = "jose";
    return view('home')->with('nombre', $nombre);
})->name('home');
```

en forma de `array`

```
Route::get('/', function(){
    $nombre = "jose";
    return view('home')->with(['nombre' => $nombre]);
})->name('home');
```

array como segundo parámetro de la función `view()`

```
Route::get('/', function(){
    $nombre = "jose";
    return view('home', ['nombre' => $nombre]);
})->name('home');
```

con la funcion `compact`

```
Route::get('/', function(){
    $nombre = "jose";
    return view('home', compact('nombre')); //['nombre' => $nombre]
})->name('home');
```

devuelve el mismo array anterior siempre y cuando tenga el mismo nombre de la variable

otra forma: usando el metodo `view`

```
Route::view('/', 'home')->name('home');
```

tambien se le puede pasar la información por medio del array

```
Route::view('/', 'home', ['nombre' => $nombre]);
```

BLADE, motor de plantillas

Blade es un motor de plantillas simple y a la vez poderoso proporcionado por Laravel. A diferencia de otros motores de plantillas populares de PHP, Blade no impide utilizar código PHP plano en sus vistas.

Todas las vistas de Blade son compiladas en código PHP plano y almacenadas en caché hasta que sean modificadas, lo que significa que Blade no añade sobrecarga a la aplicación.

Nota

Para utilizar el sistema de plantillas de Laravel se debe renombrar las vistas para que tengan la extensión `.blade.php`

imprimir variables

Para imprimir una variable, se puede hacerlo utilizando la sintaxis de dobles llaves `{{ }}`

Ejemplo / comparación

Usando php

```
<?php echo $nombre ?? "invitado"?>
```

usando blade

```
{{ $nombre ?? "invitado" }}
```

Herencia de plantillas

definir un layout

Dos de los principales beneficios de usar Blade son la *herencia de plantillas* y *secciones*. Para empezar, veamos un ejemplo simple. Primero, vamos a examinar una página de layout "master". Ya que la mayoría de las aplicaciones web mantienen el mismo layout general a través de varias páginas, es conveniente definir este layout como una sola vista de Blade:

Ejemplo de layout

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @endsection

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

en el archivo aparte del marcado HTML, se usan la directivas `@section` y `@yield`

> `@section`: define una sección de contenido.
> `@yield`: muestra el contenido de una sección determinada

Extender un Layout

Al definir una vista hija, utiliza la directiva de Blade `@extends` para indicar el layout que deberá "heredarse" en la vista hija. Las vistas que extiendan un layout de Blade pueden inyectar contenido en la sección del layout usando la directiva ***@section***. Los contenidos de estas secciones se mostrarán en el layout usando ***@yield***:

Ejemplo de extends

```
@extends('layout')

@section('title', 'Contactenos')

@section('content')
    <h1>Contact</h1>
@endsection
```

No es necesario colocar la extensión del archivo. Tampoco es necesario colocar la ruta completa, ya que Laravel por defecto buscará el archivo dentro del directorio `"resources/views"`

Dado que el titulo es una sola línea, podemos pasar el contenido como el segundo argumento de `"@section"`

Estructuras de control

Blade proporciona accesos directos y convenientes para las estructuras de control comunes de PHP, tales como sentencias **condicionales** y **bucles**. Estos accesos directos proporcionan una manera muy limpia y concisa de trabajar con estructuras de control de PHP.

Para utilizar ciclos y estructuras condicionales, se puede utilizar directivas.
Las directivas de Blade van precedidas por un arroba `(@)` y luego el nombre de la directiva

Sentencias if

Se puede construir sentencias **if** usando las directivas `@if`, `@elseif`, `@else` y `@endif`. Estas directivas funcionan idénticamente a sus contrapartes PHP:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

> Blade también tiene la directiva `@unless`, que funciona como un condicional inverso

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

Directivas de autenticación

Las directivas `@auth` y `@guest` pueden ser utilizadas para determinar rápidamente si el usuario actual está autenticado o si es un invitado

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

Sentencias switch

Las sentencias switch pueden ser construidas usando las directivas `@switch`, `@case`, `@break`, `@default` y `@endswitch`

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Bucles // for each

```
@foreach ($users as $user)
    <li>{{ $user }}</li>
@endforeach
```

La directiva @if puede ser utilizada junto con un bloque else (utilizando @else):

```
@if ($portfolio)
    @foreach ($portfolio as $portfolioItem)
        <li> {{ $portfolioItem['title'] }} </li>
    @endforeach
@else
    <li> No hay datos para mostrar</li>
@endif
```

> en caso de no estar definida la variable, se verifica con la directiva `@isset`

```
@isset ($portfolio)
    @foreach ($portfolio as $portfolioItem)
        <li> {{ $portfolioItem['title'] }} </li>
    @endforeach
@else
    <li> No hay datos para mostrar</li>
@endisset
```

directiva especial en remplazo de `@foreach`

> Con la directiva `@forelse` podemos asignar una opción por defecto a un ciclo `@foreach` sin utilizar bloques anidados

```
@forelse ($portfolio as $portfolioItem)
    <li> {{ $portfolioItem['title'] }} </li>
@empty
    <li>no hay datos para mostrar</li>
@endforelse
```

En resumen

@forelse recorre el array para comprobar
@empty sección a mostrar cuando el array esta vacio

se puede utilizar la directiva `@empty` que es una forma más corta de escribir `@if (empty (...))`

La variable loop

Al realizar un ciclo, una variable `$loop` estará disponible dentro del ciclo. Esta variable proporciona acceso a un poco de información útil, como el índice del ciclo actual y si es la primera o la última iteración del ciclo

```
@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 padre a través de la propiedad `parent`

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

nota

Cuando se está dentro del bucle, se puede usar la variable `$loop` para obtener información valiosa acerca del bucle, como puede ser saber si estás en la primera o última iteración a través del bucle.

variable `$loop`

La variable $loop contiene una variedad de otras propiedades útiles

Propiedad	Descripción
$loop->index	El índice de la iteración del ciclo actual (comienza en 0).
$loop->iteration	Iteración del ciclo actual (comienza en 1).
$loop->remaining	Iteraciones restantes en el ciclo.
$loop->count	La cantidad total de elementos en el arreglo que se itera.
$loop->first	Si esta es la primera iteración a través del ciclo.
$loop->last	Si esta es la última iteración a través del ciclo.

Propiedad	Descripción
$loop->even	Si esta es una iteración par a través del ciclo.
$loop->odd	Si esta es una iteración impar a través del ciclo.
$loop->depth	El nivel de anidamiento del bucle actual.
$loop->parent	Cuando está en un bucle anidado, la variable de bucle del padre.

Ejemplo de uso

```
@forelse ($portfolio as $portfolioItem)
    <li> {{ $portfolioItem['title'] }}
        <small>{{ $loop->last ? 'Es el ultimo' : ''}}</small>
    </li>
@empty
    <li>no hay datos para mostrar</li>
@endforelse
```

Controladores

Su función es la de definir el código a ejecutar como comportamiento frente a una acción solicitada dentro de la aplicación.

En lugar de definir toda la lógica de manejo de solicitud como `Closure` en archivos de ruta, se puede desear organizar este comportamiento usando clases `Controller`. Los controladores pueden agrupar la lógica de manejo de solicitud relacionada dentro de una sola clase.

Notas

Los controladores están localizados en la carpeta `"app/Http/Controllers"` y podemos organizarlos en subcarpetas si lo deseamos. Como otras clases de Laravel están dentro del sistema de autocarga de clases, por lo que estarán disponibles siempre que los necesitemos en la aplicación.

Código de un controlador básico

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class ArticulosController extends Controller
{
    public function ver($id)
    {
        return view('articulos.ver', ['id' => $id]);
    }
}
```

- las dos primeras lineas pertenecen a los `namespaces` donde se está trabajando
- `class ArticuloController` clase que define el controlador

definir el controlador: por convención, se usa la primera letra en mayúscula y seguido se escribe el sufijo `"Controller"`.
`NombrecontroladorController`

Dentro de la clase se coloca las acciones que se desee. Cada acción se implementa a partir de un método, al que podemos invocar desde el sistema de `routing`.

Invocar un controlador desde el sistema de rutas

Es muy importante notar que no se necesita especificar el espacio de nombre completo del controlador al momento de definir la ruta del controlador. Debido a que el `RouteServiceProvider` carga sus archivos de ruta dentro de un grupo de ruta que contiene el espacio de nombre, solamente se necesita la porción del nombre de la clase que viene después de la porción `"App\Http\Controllers"` del espacio de nombre.

Los controladores se van a invocar normalmente desde el sistema de rutas, indicando el nombre del controlador y la acción (método) que debe ejecutarse para procesar una solicitud.

Ejemplo de registro de ruta de controlador

Sí la clase de controlador completa es `App\Http\Controllers\Photos\AdminController`, se debe registrar rutas al controlador de esta forma:

```
Route::get('foo', 'Photos\AdminController@method');
```

Generar los controladores automaticamente con artisan

Crear desde cero un controlador es una tarea repetitiva dentro de Laravel, por lo que existen atajos. `"artisan"` ofrece una utilidad para crear una nueva clase controlador de una manera automática. Para ejecutarlo se lanza en la consola.

Generar controlador

se invoca dentro de la carpeta del proyecto

```
php artisan make:controller NombrecontroladorController
```

se puede pasar al final la opcion `-h` para ver mas detalles sobre el comando.

con el metodo anaterior se genera un controllador con una estructura básica (vacio), con una Clase de php que se extiende al controlador base y asu vez a otro controllador

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class CategoriaController extends Controller
{
    //
}
```

Controladores de acción única

agregando al final `-i (--invokable)` genera el metodo `__invoke()`; se usa cuando solo se quiere tener un metodo en el controlador.

Controladores de recursos

El enrutamiento de recurso de Laravel asigna las rutas típicas "CRUD" a un controlador con una sola línea de código.

se agrega al final `-r (--resource)` genera los siete métodos rest

controlador de recursos

Acciones manejadas por el controlador de recursos

TIPO	ACCIÓN	METODO	PROPOSITO
GET	*índice*	Index	Listar recursos
GET	*crear*	create	Mostrar formulario para crear nuevo recurso

TIPO	ACCIÓN	METODO	PROPOSITO
POST	*guardar*	store	Guardar el recurso en la Base de Datos
GET	*mostrar*	show	Mostrar un recurso específico encontrado por el identificador `($id)`
GET	*editar*	edit	Mostrar el formulario para editar un recurso que ya existe
PUT/PATCH	*actualizar*	update	Guardar los cambios realizados en el metodo "edit"
DELETE	*eliminar*	destroy	Eliminar el recurso por su identificador `($id)`

Ejemplo controlador de recursos

```
php artisan make:controller PortfolioController -r
```

Genera el siguiente controlador

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PortfolioController extends Controller
{
    public function index()
    {
        //
    }
    public function create()
    {
        //
    }
    public function store(Request $request)
    {
        //
    }
    public function show($id)
    {
        //
    }
    public function edit($id)
    {
        //
    }
    public function update(Request $request, $id)
    {
        //
    }
    public function destroy($id)
    {
        //
    }
}
```

invocar al controlador desde el sistema de rutas `(routing)`

```
Route::view('/portfolio', 'PortfolioController@index')->name('portfolio');
```

`"PortfolioController"` : controlador a invocar.
`"@index"` : metodo a ejecutar.

Para generar de manera rapida las siete rutas para los metodos definidos en el controlador de recursos, se utilza el metodo `::resource`

```
Route::resource()
```

```
Route::resource('nombreDelRecurso', 'nombreDelControlador');
```

```
Route::resource('projects', 'PortfolioController');
```

Rutas de recursos parciales

Al momento de declarar una ruta de recurso, se puede especificar un subconjunto de acciones que el controlador debería manejar en lugar de conjunto completo de acciones por defecto.

para esto se tiene otros metodos para complementar el trabajo del metodo `::resource`

- only
- except

`only`: Se elige por medio de un array cual de los siete metodos Rest se registran

Ejemplo método `only`

```
Route::resource('projects', 'PortfolioController')->only(['index', 'show']);
```

al ejecutar `"php artisan r:l"` se puede ver solo las dos rutas (index y show)

`except`: Se elige por medio de un array cual de los siete metodos Rest no se registrarán

Ejemplo método `except`

```
Route::resource('projects', 'PortfolioController')->except(['index', 'show']);
```

al ejecutar `"php artisan r:l"` se puede ver todas las rutas excepto las dos (index y show)

Controlador de recursos API

Al momento de declarar rutas de recursos que serán consumidas por APIs, normalmente es conveniente excluir rutas que presentan plantillas HTML tales como `create` y `edit`. Se puede usar el método `"apiResource()"` para excluir automáticamente éstas dos rutas.

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

La diferencia con el controlador `--resource` es que este excluye los metodos edit y create

Para generar de manera rapida las rutas para los metodos definidos en el controlador de recursos API, se usa el metodo `::apiResource`

```
Route::apiResource('projects', 'PortfolioController');
```

> al ejecutar `"php artisan r:l"` se puede ver que se registran solamente los cinco metodos

HTTP Request

Toda aplicación web recibe solicitudes para completar todo tipo de acciones. Cada solicitud que recibe el servidor viene acompañada de una serie de datos, que se envían en el protocolo HTTP. Entre la información que recibe PHP podemos encontrar desde el user-agent del visitante o su IP, hasta datos que viajan en las cabeceras ante una operación post.

Esos datos en Laravel se acceden a través del objeto `Request`, que podemos recibir en el controlador mediante la inyección de dependencias, o mediante la correspondiente facade.

> **Para tener en cuenta**
>
> La clase Request está en el namespace `"Illuminate\Http\Request"`

```
use Illuminate\Http\Request;
```

Inyección de dependencias Y parametros de rutas

Si tu método de controlador también está esperando la entrada de un parámetro de ruta deberías listar tus parámetros de ruta después de tus otras dependencias.

> **Ejemplo de `Route`**
>
> Por ejemplo, si tu ruta es definida como sigue:

```
Route::put('user/{id}', 'UserController@update');
```