CLASE 2 / 6

Indice

CONTROLADORES

En general la forma recomendable de trabajar será asociar las rutas a un método de un controlador. Esto permitirá separar mucho mejor el código y crear clases (controladores) que agrupen toda la funcionalidad de un determinado recurso. Por ejemplo, crear un controlador para gestionar toda la lógica asociada al control de usuarios o cualquier otro tipo de recurso.

Los controladores son el punto de entrada de las peticiones de los usuarios y son los que deben contener toda la lógica asociada al procesamiento de una petición, encargándose de realizar las consultas necesarias a la base de datos, de preparar los datos y de llamar a la vista correspondiente con dichos datos.

CONTROLADOR BÁSICO

Los controladores se almacenan en archivos PHP en la carpeta `app/Http/Controllers` y normalmente se les añade el sufijo `Controller`, por ejemplo `UserController.php` o `MoviesController.php`. A continuación se incluye un ejemplo básico de un controlador almacenado en el archivo `app/Http/Controllers/UserController.php`:

```
<?php
namespace App\Http\Controllers;

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

class UserController extends Controller
{
    /**
     * Mostrar información de un usuario.
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = User::findOrFail($id);
        return view('user.profile', ['user' => $user]);
    }
}
```

Todos los controladores tienen que extender de la clase base `Controller`. Esta clase viene ya creada por defecto con la instalación de Laravel, se encuentra en la carpeta `app/Http/Controllers`. Se utiliza para centralizar toda la lógica que se vaya a utilizar de forma compartida por los controladores de la aplicación. Por defecto solo carga código para validación y autorización, pero se pueden añadir en la misma todos los métodos que se necesiten.

En el código de ejemplo, el método `showProfile($id)` lo único que realiza es obtener los datos de un usuario, generar la vista `user.profile` a partir de los datos obtenidos y devolverla como valor de retorno para que se muestre por pantalla.

Una vez definido un controlador ya es posible asociarlo a una ruta. Para esto tenemos que modificar el archivo de rutas `routes/web.php` de la forma:

```
Route::get('user/{id}', 'UserController@showProfile');
```

En lugar de pasar una función como segundo parámetro, se debe escribir una cadena que contenga el nombre del controlador, seguido de una arroba `@` y del nombre del método que se quiere asociar. No es necesario añadir nada más, ni los parámetros que recibe el método en cuestión, todo esto se hace de forma automática.

Crear un nuevo controlador

Los controladores se almacenan dentro de la carpeta `app/Http/Controllers` como archivos PHP. Para crear uno nuevo bien se puede crear a mano y rellenar todo el código, o utilizar el siguiente comando de Artisan que adelantará todo el trabajo:

```
php artisan make:controller ProductosController
```

Este comando creará el controlador `ProductosController` dentro de la carpeta `app/Http/Controllers` y lo completará con el código básico visto antes. Al añadir la opción `--plain` no añadira ningún método al controlador, por lo que el cuerpo de la clase estará vacío.

Controladores y espacios de nombres

Se pueden crear subcarpetas dentro de la carpeta `Controllers` para organizar mejor. En este caso, la estructura de carpetas no tendrá nada que ver con la ruta asociada a la petición y, de hecho, a la hora de hacer referencia al controlador únicamente tendrá que hacerlo a través de su espacio de nombres.

Al referenciar el controlador en el archivo de rutas únicamente se indica su nombre y no toda la ruta ni el espacio de nombres `App\Http\Controllers`. Esto es porque el servicio encargado de cargar las rutas añade automáticamente el espacio de nombres raíz para los controladores. Si se crean subcarpetas y se organizan controladores en subespacios de nombres, entonces sí se tendrá que añadir esa parte.

Por ejemplo, si se crea un controlador en `App\Http\Controllers\Fotos\AdminController`, entonces para registrar una ruta hasta dicho controlador habrá que hacer:

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

Generar una URL a una acción

Para generar la URL que apunte a una acción de un controlador podemos usar el método `action` de la forma:

```
$url = action('FooController@method');
```

Por ejemplo, para crear en una plantilla con *Blade* un enlace que apunte a una acción haríamos:

```
<a href="{{ action('FooController@method') }}">¡Aprieta aquí!</a>
```

También se podría redirigir a un método de un controlador mediante el método `action` de la forma:

```
return redirect()->action('HomeController@index');
```

Para añadir parámetros para la llamada al método del controlador se deben añadir pasando un array como segundo parámetro:

```
return redirect()->action('UserController@profile', [1]);
```

Caché de rutas

Si definimos todas las rutas para que utilicen controladores podemos aprovechar la nueva funcionalidad para crear una caché de las rutas. Es importante que estén basadas en controladores porque si definimos respuestas directas desde el archivo de rutas (como en el capítulo anterior) la caché no funcionará.

Gracias a la caché Laravel indica que se puede acelerar el proceso de registro de rutas hasta 100 veces. Para generar la caché simplemente tenemos que ejecutar el comando de *Artisan*:

```
php artisan route:cache
```

Si se crean más rutas se debe volver a lanzar el mismo comando para agregarlas. Para borrar la caché de rutas y no generar una nueva caché se ejecuta:

```
php artisan route:clear
```

Controlador de Recurso (Resource Controller)

Con Laravel crear un controlador que tenga todos los métodos necesarios para un CRUD (Create, Read, Update and Delete) o que facilite la construcción de controladores tipo RESTful es muy simple, gracias a la consola interactiva se puede crear con este simple comando:

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

Además de crear un controlador para un CRUD, se puede asociar el controlador a un modelo añadiendo la opción model al comando anterior de esta forma:

```
php artisan make:controller UserController --resource --model=User
```

Para enlazar a estos controladores con todos sus métodos se puede usar el método `resource` en `routes/web.php`:

```
Route::resource('users', 'UserController');
```

Asociando los métodos de la siguiente forma:

- GET: index, create, show, edit.
- POST: store.
- PUT: update.
- DELETE: destroy.
- PATCH: update.

En el caso de no utilizar algún método en específico, se puede desactivar utilizando el método `except` y en el caso de que solo se quiera utilizar los que se especifiquen, se puede hacer con el método `only` como a continuación:

```
Route::resource('users', 'UserController')->only([ 'index', 'show',
'destroy']);

Route::resource('Users', 'UserController')->except(['create', 'store']);
```

Si se esta construyendo una API y se desea generar rápidamente un controlador de recursos que no incluya los métodos de create y edit, se utiliza la opción `--api` del comando `make:controller`:

```
php artisan make:controller Api/UserController --api
```

Para hacer referencia a este controlador desde las rutas se llama al método `apiResource`, este método cumple la misma función que el método `resource` pero excluye los métodos create y edit.

```
Route::apiResource('users', 'UserController');
```

Si existen otros controladores para recursos tipo API se puede utilizar el mismo método para añadir varios pasando un arreglo como argumento.

Para poder verificar que las rutas fueron creadas:

```
php artisan route:list
```

CONTROLADORES DE UN SOLO MÉTODO

Si un controlador cumple una función muy específica y única, tanto que solo dispone de un método, Laravel permite utilizar el método `__invoke` de esta forma:

```
<?php
namespace App\Http\Controllers;
use App\Ticket;
use App\Http\Controllers\Controller;

class ShowTicket extends Controller
{
    public function __invoke(Ticket $ticket)
    {
        return view('tickets.show', compact('ticket'));
    }
}
```

El método `__invoke` será lanzado al intentar llamar a la clase de un controlador sin especificar ningún método:

```
Route::get('ticket/{ticket}', 'ShowTicket');
```

Middleware o filtros

Los componentes llamados *Middleware* son un mecanismo proporcionado por Laravel para **filtrar las peticiones HTTP** que se realizan a una aplicación. Un filtro o *middleware* se define como una clase PHP almacenada en un archivo dentro de la carpeta `app/Http/Middleware`. Cada *middleware* se encargará de aplicar un tipo concreto de filtro y de decidir que realizar con la petición realizada: permitir su ejecución, dar un error o redireccionar a otra página en caso de no permitirla.

Laravel incluye varios filtros por defecto, uno de ellos es el encargado de realizar la autenticación de los usuarios. Este filtro se puede aplicar sobre una ruta, un conjunto de rutas o sobre un controlador en concreto. Este *middleware* se encargará de filtrar las peticiones a dichas rutas: en caso de estar logueado y tener permisos de acceso le permitirá continuar con la petición, y en caso de no estar autenticado lo redireccionará al formulario de login.

Laravel incluye *middleware* para gestionar la autenticación, el modo mantenimiento, verificación del token CSRF, encriptacion de cookies y algunos mas. Todos estos filtros se encuentran en la carpeta `app/Http/Middleware`, los cuales se pueden modificar o ampliar su funcionalidad. Pero además se pueden crear nuevos.

Definir un nuevo *Middleware*

Para crear un nuevo *Middleware* se utiliza el comando de Artisan:

```
php artisan make:middleware MiMiddleware
```

Este comando creará la clase `MiMiddleware` dentro de la carpeta `app/Http/Middleware` con el siguiente contenido por defecto:

```
<?php

namespace App\Http\Middleware;

use Closure;

class MiMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
```

```
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

El código generado por Artisan ya viene preparado para escribir directamente la implementación del filtro a realizar dentro de la función `handle`. Esta función solo incluye el valor de retorno con una llamada a `return $next($request);`, que lo que hace es continuar con la petición y ejecutar el método que tiene que procesarla. Como entrada la función `handle` recibe dos parámetros:

- `$request`: En la cual vienen todos los parámetros de entrada de la petición.
- `$next`: El método o función que tiene que procesar la petición.

Por ejemplo se podría crear un filtro que redirija al home si el usuario tiene menos de 18 años y en otro caso que le permita acceder a la ruta:

```
public function handle($request, Closure $next)
{
    if ($request->input('age') < 18) {
        return redirect('home');
    }

    return $next($request);
}
```

Se pueden hacer tres cosas con una petición:

- Si todo es correcto permitir que la petición continúe devolviendo: `return $next($request);`
- Realizar una redirección a otra ruta para no permitir el acceso con: `return redirect('home');`
- Lanzar una excepción o llamar al método `abort` para mostrar una página de error: `abort(403, 'Unauthorized action.');`

Middleware antes o después de la petición

Para hacer que el código de un *Middleware* se ejecute antes o después de la petición HTTP simplemente hay que poner el código antes o después de la llamada a `$next($request);`. Por ejemplo, el siguiente *Middleware* realizaría la acción **antes** de la petición:

```
public function handle($request, Closure $next)
{
    // Código a ejecutar antes de la petición

    return $next($request);
}
```

Mientras que el siguiente *Middleware* ejecutaría el código **después** de la petición:

```
public function handle($request, Closure $next)
{
    $response = $next($request);

    // Código a ejecutar después de la petición

    return $response;
}
```

USO DE *MIDDLEWARE*

Laravel permite la utilización de *Middleware* de tres formas distintas: global, asociado a rutas o grupos de rutas, o asociado a un controlador o a un método de un controlador. En los tres casos será necesario registrar primero el *Middleware* en la clase `app/Http/Kernel.php`.

MIDDLEWARE GLOBAL

Para hacer que un *Middleware* se ejecute con **todas** las peticiones HTTP realizadas a una aplicación simplemente hay que registrarlo en el array `$middleware` definido en la clase `app/Http/Kernel.php`. Por ejemplo:

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
    \App\Http\Middleware\MiMiddleware::class,
];
```

En este ejemplo se ha registrado la clase *MiMiddleware* al final del array. Si queremos que este *middleware* se ejecute antes que otro filtro simplemente habrá que colocarlo antes en la posición del array.

MIDDLEWARE ASOCIADO A RUTAS

En el caso de querer que el *middleware* se ejecute solo cuando se llame a una ruta o a un grupo de rutas también se tendrá que registrar en el archivo `app/Http/Kernel.php`, pero en el array `$routeMiddleware`. Al añadirlo a este array además habrá que asignarle un nombre o clave, que será el que después se utilizara para asociarlo con una ruta.

En primer lugar se añade el filtro al array y le asignamos el nombre "`es_mayor_de_edad`":

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\
AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'es_mayor_de_edad' => \App\Http\Middleware\MyMiddleware::class,
```

```
];
```

Una vez registrado el *middleware* ya lo se puede utilizar en el archivo de rutas `routes/web.php` mediante la clave o nombre asignado, por ejemplo:

```
Route::get('dashboard', ['middleware' => 'es_mayor_de_edad', function () {
    //...
}]);
```

En el ejemplo se ha asignado el *middleware* con clave `es_mayor_de_edad` a la ruta `dashboard`. Como se puede ver se utiliza un array como segundo parámetro, en el cual se indica el *middleware* y la acción. Si la petición supera el filtro entonces se ejecutara la función asociada.

Para asociar un filtro con una ruta que utiliza un método de un controlador se realizaría de la misma manera pero indicando la acción mediante la clave "`uses`":

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

Para asociar varios *middleware* con una ruta simplemente hay que añadir un array con las claves. Los filtros se ejecutarán en el orden indicado en dicho array:

```
Route::get('dashboard', ['middleware' => ['auth', 'es_mayor_de_edad'],
function () {
    //...
}]);
```

Laravel también permite asociar los filtros con las rutas usando el método `middleware()` sobre la definición de la ruta de la forma:

```
Route::get('/', function () {
    // ...
})->middleware(['first', 'second']);

// O sobre un controlador:
Route::get('profile', 'UserController@showProfile')->middleware('auth');
```

Para aplicar un filtro sobre todo un conjunto de rutas, que permite especificar el filtro una vez y además dividir las rutas en secciones (distinguiendo mejor a que secciones se les está aplicando un filtro):

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('/', function ()    {
        // Ruta filtrada por el middleware
    });

    Route::get('user/profile', function () {
        // Ruta filtrada por el middleware
    });
```

```
});
```

MIDDLEWARE DENTRO DE CONTROLADORES

También es posible indicar el *middleware* a utilizar desde dentro de un controlador. En este caso los filtros también tendrán que estar registrados en el array `$routeMiddleware` del archivo `app/Http/Kernel.php`. Para utilizarlos se recomienda realizar la asignación en el constructor del controlador y asignar los filtros usando su clave mediante el método `middleware`. Podremos indicar que se filtren todos los métodos, solo algunos, o todos excepto los indicados, por ejemplo:

```
class UserController extends Controller
{
    /**
     * Instantiate a new UserController instance.
     *
     * @return void
     */
    public function __construct()
    {
        // Filtrar todos los métodos
        $this->middleware('auth');

        // Filtrar solo estos métodos...
        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        // Filtrar todos los métodos excepto...
        $this->middleware('subscribed', ['except' => ['fooAction',
'barAction']]);
    }
}
```

Esto puede ser practico ya que las rutas no nos quedarían tan sobrecargadas, ya que el middleware queda dentro del controlador y ya no lo tendríamos que llamar dentro de la ruta.

REVISAR LOS FILTROS ASIGNADOS

Al crear una aplicación Web es importante asegurarse de que todas las rutas definidas son correctas y que las partes privadas realmente están protegidas. Para esto Laravel incluye el siguiente método de Artisan:

```
php artisan route:list
```

Este método muestra una tabla con todas las rutas, métodos y acciones. Ademas para cada ruta indica los filtros asociados, tanto si están definidos desde el archivo de rutas como **desde dentro de un controlador**. Por lo tanto es muy útil para comprobar que todas las rutas y filtros que se han definido se hayan creado correctamente.

PASO DE PARÁMETROS

Un *Middleware* también puede recibir parámetros. Por ejemplo, se puede crear un filtro para comprobar si el usuario logueado tiene un determinado rol indicado por parámetro. Para esto lo primero es añadir un tercer parámetro a la función `handle` del *Middleware*:

```
<?php

namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string  $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // No tiene el rol esperado!
        }

        return $next($request);
    }

}
```

En el código de ejemplo se ha añadido el tercer parámetro `$role` a la función. Si el filtro necesita recibir más parámetros simplemente hay que añadirlos de la misma forma a esta función.

Para pasar un parámetro a un *middleware* en la definición de una ruta se debe añadir a continuación del nombre del filtro separado por dos puntos, por ejemplo:

```
Route::put('post/{id}', ['middleware' => 'role:editor', function ($id) {
    //
}]);
```

Si hay que pasar más de un parámetro al filtro, estos se separan por comas, por ejemplo: `role:editor,admin`.

FORMULARIOS

CREAR FORMULARIOS

Para abrir y cerrar un formulario que apunte a la URL actual y utilice el método POST hay que usar las siguientes etiquetas HTML:

```
<form method="POST">
    ...
</form>
```

Si se desea cambiar la URL de envío de datos, utilizar el atributo `action` de la forma:

```
<form action="{{ url('foo/bar') }}" method="POST">
    ...
</form>
```

La función `url` generará la dirección a la ruta indicada. Ademas también se puede usar la función `action` para indicar directamente el método de un controlador a utilizar, por ejemplo: `action('HomeController@getIndex')`

En Laravel es posible definir distintas acciones para procesar peticiones realizadas a una misma ruta pero usando un método distinto (GET, POST, PUT, DELETE). Por ejemplo, definir la ruta "`user`" de tipo `GET` para que devuelva la página con el formulario para crear un usuario, y por otro lado definir la ruta "`user`" de tipo `POST` para procesar el envío del formulario. De esta forma cada ruta apuntará a un método distinto de un controlador y facilitará la separación del código.

HTML solo permite el uso de formularios de tipo GET o POST. Si se desea enviar un formulario usando otros de los métodos (o verbos) definidos en el protocolo REST, como son PUT, PATCH o DELETE, se tendrá que añadir un campo oculto para indicarlo. Laravel establece el uso del nombre "`_method`" para indicar el método a usar, por ejemplo:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    ...
</form>
```

Laravel se encargará de recoger el valor de dicho campo y de procesarlo como una petición tipo PUT (o la que se indique). Además, para facilitar más la definición de este tipo de formularios ha añadido la función `method_field` que directamente creará este campo oculto:

```
<form action="/foo/bar" method="POST">
    {{ method_field('PUT') }}
    ...
</form>
```

Protección contra CSRF

El CSRF (del inglés *Cross-site request forgery* o falsificación de petición en sitios cruzados) es un tipo de exploit malicioso de un sitio web en el que comandos no autorizados son transmitidos por un usuario en el cual el sitio web confía.

Laravel proporciona una forma fácil de protegernos de este tipo de ataques. Simplemente se agrega la directiva de Blade `@csrf` después de abrir el formulario, este añadirá un campo oculto (token del usuario) ya configurado con los valores necesarios.

```
<form action="/foo/bar" method="POST">
    @csrf
    ...
</form>
```

Elementos de un formulario

En todos los tipos de campos en los que se requiere recoger datos es importante añadir sus atributos `name` e `id`, ya que servirán después para recoger los valores rellenados por el usuario.

Campos de texto

Para crear un campo de texto se usa la etiqueta de HTML `input`, para la cual hay que indicar el tipo `text` y su nombre e identificador de la forma:

```
<input type="text" name="nombre" id="nombre">
```

El atributo `name` indica el nombre de variable donde se guardará el texto introducido por el usuario y que después se utilizara desde el controlador para acceder al valor.

Se puede especificar una ayuda para el usuario con el atributo `placeholder`:

```
<input type="text" name="email" placeholder="Email...">
```

Se puede especificar un valor por defecto usando el atributo `value`:

```
<input type="text" name="asunto" value="Asunto: ">
```

Desde una vista con *Blade* podemos asignar el contenido de una variable (en el ejemplo `$nombre`) para que aparezca el campo de texto con dicho valor. Esta opción es muy útil para crear formularios en los que tenemos que editar un contenido ya existente, como por ejemplo editar los datos de usuario.

```
<input type="text" name="nombre" id="nombre" value="{{ $nombre }}">
```

Para mostrar los valores introducidos en una petición anterior se usara el método `old`, el cual recuperará las variables almacenadas en la petición anterior. Por ejemplo, se desea crear un formulario para el registro de usuarios y al enviar el formulario se comprueba que el usuario introducido está repetido. En ese caso se tendría que volver a mostrar el formulario con los datos

introducidos y marcar dicho campo como erróneo. Para esto, después de comprobar que hay un error en el controlador, habría que realizar una redirección a la página anterior añadiendo la entrada con `withInput()`, por ejemplo: `return back()->withInput();`. El método `withInput()` añade todas las variables de entrada a la sesión, y esto permite recuperarlas después de la forma:

```
<input type="text" name="nombre" id="nombre" value="{{ old('nombre') }}">
```

Otros campos tipo *input*

Utilizando la etiqueta `input` se pueden crear más tipos de campos como contraseñas o campos ocultos:

```
<input type="password" name="password" id="password">

<input type="hidden" name="oculto" value="valor">
```

Los campos para contraseñas lo único que hacen es ocultar las letras escritas. Los campos ocultos se suelen utilizar para almacenar opciones o valores que se desean enviar junto con los datos del formulario pero que no se tienen que mostrar al usuario.

También pueden crearse otros tipos de *inputs* como *email*, *number*, *tel*, etc. (tipos permitidos aquí: http://www.w3schools.com/html/html_form_input_types.asp). Para definir estos campos se hace exactamente igual que para un campo de texto pero cambiando el tipo por el deseado, por ejemplo:

```
<input type="email" name="correo" id="correo">

<input type="number" name="numero" id="numero">

<input type="tel" name="telefono" id="telefono">
```

Textarea

Para crear un área de texto se usara la etiqueta HTML `textarea` de la forma:

```
<textarea name="texto" id="texto"></textarea>
```

Esta etiqueta además permite indicar el número de filas (`rows`) y columnas (`cols`) del área de texto. Para insertar un texto o valor inicial se pone entre la etiqueta de apertura y la de cierre.

```
<textarea name="texto" id="texto" rows="4" cols="50">Texto por
defecto</textarea>
```

Etiquetas

Las etiquetas permiten poner un texto asociado a un campo de un formulario para indicar el tipo de contenido que se espera en dicho campo. Por ejemplo añadir el texto "Nombre" antes de un `input` tipo texto donde el usuario tendrá que escribir su nombre.

Para crear una etiqueta hay que usar el *tag* "`label`" de HTML:

```
<label for="nombre">Nombre</label>
```

Donde el atributo `for` se utiliza para especificar el identificador del campo relacionado con la etiqueta. De esta forma, al pulsar sobre la etiqueta se marcará automáticamente el campo relacionado.

```
<label for="correo">Correo electrónico:</label>
<input type="email" name="correo" id="correo">
```

Checkbox y Radio buttons

Para crear campos tipo *checkbox* o tipo *radio button* se debe utilizar también la etiqueta `input`, pero indicando el tipo `chekbox` o `radio` respectivamente.

```
<label for="terms">Aceptar términos</label>
<input type="checkbox" name="terms" id="terms" value="1">
```

En este caso, al enviar el formulario, si el usuario marca la casilla nos llegaría la variable con nombre `terms` con valor `1`. En caso de que no marque la casilla no llegaría nada, ni siquiera la variable vacía.

Para crear una lista de *checkbox* o de *radio button* es importante que todos tengan el **mismo nombre**(para la propiedad `name`). De esta forma los valores devueltos estarán agrupados en esa variable, y además, el *radio button* funcionará correctamente: al apretar sobre una opción se desmarcará la que este seleccionada en dicho grupo (entre todos los que tengan el mismo nombre). Por ejemplo:

```
<label for="color">Elige tu color favorito:</label>
<br>
<input type="radio" name="color" id="color" value="rojo">Rojo<br>
<input type="radio" name="color" id="color" value="azul">Azul<br>
<input type="radio" name="color" id="color" value="amarillo">Amarillo<br>
<input type="radio" name="color" id="color" value="verde">Verde<br>
```

Además se puede añadir el atributo `checked` para marcar una opción por defecto:

```
<label for="clase">Clase:</label>
<input type="radio" name="clase" id="clase" value="turista" checked>
Turista <br>
<input type="radio" name="clase" id="clase" value="preferente"> Preferente
<br>
```

Archivos

Para generar un campo para subir archivos se utiliza también la etiqueta *input* indicando en su tipo el valor `file`, por ejemplo:

```
<label for="imagen">Sube la imagen:</label>
<input type="file" name="imagen" id="imagen">
```

Para enviar archivos la etiqueta de apertura del formulario tiene que cumplir dos requisitos importantes:

- El método de envío tiene que ser POST o PUT.
- Se debe añadir el atributo *enctype="multipart/form-data"* para indicar la codificación.

```
<form enctype="multipart/form-data" method="post">
    <label for="imagen">Sube la imagen:</label>
    <input type="file" name="imagen" id="imagen">
</form>
```

Listas desplegables

Para crear una lista desplegable se utiliza la etiqueta HTML `select`. Las opciones se indicaran entre la etiqueta de apertura y cierre usando elementos `option`, de la forma:

```
<select name="marca">
  <option value="vw">VW</option>
  <option value="fiat">Fiat</option>
  <option value="renault">Renault</option>
  <option value="peugeot">Peugeot</option>
</select>
```

En el ejemplo anterior se creará una lista desplegable con cuatro opciones. Al enviar el formulario el valor seleccionado nos llegará en la variable `marca`. Además, para elegir una opción por defecto se puede utilizar el atributo `selected`.

```
<label for="talla">Elige la talla:</label>
<select name="talla" id="talla">
  <option value="XS">XS</option>
  <option value="S">S</option>
  <option value="M" selected>M</option>
  <option value="L">L</option>
  <option value="XL">XL</option>
</select>
```

Botones

En un formulario pueden añadirse tres tipos distintos de botones:

- `submit` para enviar el formulario,
- `reset` para restablecer o borrar los valores introducidos y
- `button` para crear botones normales para realizar otro tipo de acciones (como volver a la página anterior).

```
<button type="submit">Enviar</button>
<button type="reset">Borrar</button>
<button type="button">Volver</button>
```

Validación restringida

Los siguientes elementos de sintaxis de HTML5 pueden ser usados para restringir datos en el formulario.

- El atributo `required` en los elementos `<input>`, `<select>` y `<textarea>` indica que se debe ingresar algún dato. (En el elemento `<input>`, `required` sólo se aplica con ciertos valores del atributo `type`.)
- El atributo `pattern` en el elemento `<input>` restringe el valor para que concuerde con una expresión regular específica.
- Los atributos `min` y `max` del elemento `<input>` restringen los valores máximos y mínimos que pueden ser ingresados.
- El atributo `step` del elemento `<input>` (cuando se usa en combinación con los atributos `min` y `max`) restringe la granularidad de los valores ingresados. Un valor que no se corresponda con un valor permitido no será validado.
- El atributo `maxlength` de los elementos `<input>` y `<textarea>` restringe el máximo número de caracteres (en puntos de código unicode) que el usuario puede ingresar.
- Los valores `url` y `email` para `type` restringen el valor para una URL o dirección de correo válida respectivamente.
- Con `autocomplete = "off"` en un elemento `<input>` se obtiene que la casilla de ingreso no muestre datos guardados por el navegador.
- Se puede prevenir la validación restringida especificando el atributo `novalidate` en el elemento `<form>`, o el atributo `formnovalidate` en el elemento `<button>` y en el elemento `<input>` (cuando `type` es `submit` o `image`). Estos atributos indican que el formulario no será validado cuando se envíe.

DATOS DE ENTRADA

Laravel facilita el acceso a los datos de entrada del usuario a través de solo unos pocos métodos. No importa el tipo de petición que se haya realizado (POST, GET, PUT, DELETE), si los datos son de un formulario o si se han añadido a la *query string*, en todos los casos se obtendrán de la misma forma.

Para conseguir acceso a estos métodos Laravel utiliza inyección de dependencias. Esto es simplemente añadir la clase `Request` al constructor o método del controlador en el que se necesite. Laravel se encargará de inyectar dicha dependencia ya inicializada y directamente se podrá usar este parámetro para obtener los datos de entrada.

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller
{
    public function store(Request $request)
    {
        $name = $request->input('nombre');

        //...
    }
}
```

En este ejemplo como se puede ver se ha añadido la clase `Request` como parámetro al método `store`. Laravel automáticamente se encarga de inyectar estas dependencias por lo que directamente se puede usar la variable `$request` para obtener los datos de entrada.

Si el método del controlador tuviera más parámetros simplemente se añaden a continuación de las dependencias, por ejemplo:

```
public function edit(Request $request, $id)
{
    //...
}
```

Obtener los valores de entrada

Para obtener el valor de una variable de entrada se usa el método `get` indicando el nombre de la variable:

```
$name = $request->get('nombre');

// O simplemente....
$name = $request->nombre;
```

También se puede especificar un valor por defecto como segundo parámetro:

```
$name = $request->get('nombre', 'Pedro');
```

Comprobar si una variable existe

Si es necesario, es posible comprobar si un determinado valor existe en los datos de entrada:

```
if ($request->has('nombre'))
{
    //...
}
```

Obtener datos agrupados

También es posible obtener todos los datos de entrada a la vez (en un array) o solo algunos de ellos:

```
// Obtener todos:
$input = $request->all();
```

```
// Obtener solo los campos indicados:
$input = $request->only('username', 'password');

// Obtener todos excepto los indicados:
$input = $request->except('credit_card');
```

Obtener datos de un array

Si la entrada proviene de un *input* tipo array de un formulario (por ejemplo una lista de *checkbox*), se utiliza la notación con puntos para acceder a los elementos del array de entrada:

```
$input = $request->input('products.0.name');
```

JSON

Si la entrada está codificada en formato JSON (por ejemplo cuando la aplicación se comunica a través de una API es bastante común) también se accede a los diferentes campos de los datos de entrada de forma normal con los métodos vistos, por ejemplo:

```
$nombre = $request->get('nombre');
```

Archivos de entrada

Laravel facilita una serie de clases para trabajar con los archivos de entrada. Por ejemplo para obtener un archivo que se ha enviado en el campo con nombre `photo` y guardarlo en una variable, tenemos que hacer:

```
$file = $request->file('photo');

// O simplemente...
$file = $request->photo;
```

Se puede comprobar si un determinado campo tiene un archivo asignado:

```
if ($request->hasFile('photo')) {
    //...
}
```

El objeto recuperado con `$request->file()` es una instancia de la clase `Symfony\Component\HttpFoundation\File\UploadedFile`, la cual extiende la clase de PHP `SplFileInfo` (http://php.net/manual/es/class.splfileinfo.php), por lo tanto, hay muchos métodos que podemos utilizar para obtener datos del archivo o para gestionarlo.

Por ejemplo, para comprobar si el archivo que se ha subido es válido:

```
if ($request->file('photo')->isValid()) {
    //...
}
```

O para mover el archivo de entrada a una ruta determinada:

```
// Mover el archivo a la ruta conservando el nombre original:
$request->file('photo')->move($destinationPath);

// Mover el archivo a la ruta con un nuevo nombre:
```

```
$request->file('photo')->move($destinationPath, $fileName);
```

En la última versión de Laravel se ha incorporado una nueva librería que permite gestionar el acceso y escritura de archivos en un almacenamiento. Lo interesante de esto es que permite manejar de la misma forma el almacenamiento en local, en Amazon S3 y en Rackspace Cloud Storage, simplemente hay que configurar en `config/filesystems.php` y posteriormente se usara de la misma forma. Por ejemplo, para almacenar un archivo subido mediante un formulario se usa el método `store` indicando como parámetro la ruta donde almacenar el archivo (sin el nombre del archivo):

```
$path = $request->photo->store('images');
$path = $request->photo->store('images', 's3');  // Especificar un
almacenamiento
```

Estos métodos devolverán el path hasta el archivo almacenado de forma relativa a la raíz de disco configurada. Para el nombre del archivo se generará automáticamente un UUID (identificador único universal). Si se quiere especificar el nombre tendríamos que usar el método `storeAs`:

```
$path = $request->photo->storeAs('images', 'filename.jpg');
$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

Otros métodos para recuperar información del archivo son:

```
// Obtener la ruta:
$path = $request->file('photo')->getRealPath();

// Obtener el nombre original:
$name = $request->file('photo')->getClientOriginalName();

// Obtener la extensión:
$extension = $request->file('photo')->getClientOriginalExtension();

// Obtener el tamaño:
$size = $request->file('photo')->getSize();

// Obtener el MIME Type:
$mime = $request->file('photo')->getMimeType();
```

VALIDACIÓN DE FORMULARIOS

Laravel proporciona varios enfoques diferentes para validar los datos entrantes de una aplicación. De forma predeterminada, la clase base del controlador de Laravel usa una característica `ValidatesRequests` la cual proporciona un método conveniente para validar la solicitud HTTP entrante con una variedad de poderosas reglas de validación, si se cumplen las condiciones impuestas en esta clase el formulario se envía pero encaso contrario el usuario será redireccionado.

Se va a suponer que tendremos un sistema de controlador para el manejo de funciones que procesan los datos y rutas+vistas que nos lancen los formularios de recogida de datos.

```
// rutas para mostrar y procesar el formulario en routes/web.php
Route::get('alumno/create', 'alumnoController@create');
Route::post('alumno/store', 'alumnoController@store');


// clase con métodos para el manejo del modelo Alumno en
app/Http/Controllers/alumnoController.php
class alumnoController extends Controller
{
    public function create()
    {
        return view('alumno.create');
    }


    public function store(Request $request)
    {
        /* recogida - validación - guardado */
    }
}
```

MÉTODO 1.- VALIDATOR::MAKE

Este método se puede dividir en 3 pasos: recogida, reglas y validación.

Lo primero de todo es la recogida de los request input del formulario de la vista. Seguidamente se generan una serie de reglas que entenderá `ValidatesResquests` para restringirle una norma y para finalizar, pasar por validación los datos y las reglas a la clase `Validator` mediante el método `make()`:

```
// @vars Request $request
public function store(Request $request)
{
  $data = Request::all();

  $rules = [
    'nombre' => 'required|max:191',
    'materia' => 'required|max:255',
    'nota' => 'required|integer|min:0|max:10
  ];

  $validation = Validator::make($data, $rules);

  if( $validation->fails() )
  {
    return redirect()->back()
      ->withErrors( $validator->errors() )
      ->withInput();
  }

  $alumno = Alumno::create($request->all());
  return redirect('alumno/create')->with('status', 'Alumno creado OK');
}
```

La variable `$data` almacena toda la información recogida en los inputs del formulario.

El array `$rules` contiene un conjunto campo/reglas asociativo donde se detalla la regla que debe seguir cada campo del formulario. En este caso, todos los campos son requeridos, 'nota' tiene que ser un número entero entre 0 y 10, límites de caracteres para los campos nombre y materia… Para un mayor detalle de la clase de reglas que se pueden pasar al validador, aquí está en enlace https://laravel.com/docs/5.8/validation#available-validation-rules

Accepted	Distinct	Not Regex
Active URL	E-Mail	Nullable
After (Date)	Ends With	Numeric
After Or Equal (Date)	Exists (Database)	Present
Alpha	File	Regular Expression
Alpha Dash	Filled	Required
Alpha Numeric	Greater Than	Required If
Array	Greater Than Or Equal	Required Unless
Bail	Image (File)	Required With
Before (Date)	In	Required With All
Before Or Equal (Date)	In Array	Required Without
Between	Integer	Required Without All
Boolean	IP Address	Same
Confirmed	JSON	Size
Date	Less Than	Sometimes
Date Equals	Less Than Or Equal	Starts With
Date Format	Max	String
Different	MIME Types	Timezone
Digits	MIME Type By File Extension	Unique (Database)
Digits Between	Min	URL
Dimensions (Image Files)	Not In	UUID

El objeto `$validation` contiene la validación al que se le pasa la información y las reglas.

Si su método `fails()` es afirmativo, se ordenará que vuelva al formulario con los errores del validador y el contenido que tenían los campos. Si el validador no falla, se grabará el nuevo registro y retornará con un mensaje afirmativo.

MÉTODO 2.- VALIDATE()

Con las mismas reglas que se tenían configuradas anteriormente ahora se hace la llamada al método `validate()` que recibe un array con las reglas a procesar aplicándolo a la función `request()`.

Este método interrumpirá inmediatamente el proceso de guardado si encuentra algún fallo, devolviéndolos a la vista:

```
public function store()
{
```

```
  request()->validate([
    'nombre' => 'required|max:191',
    'materia' => 'required|max:255',
    'nota' => 'required|integer|min:0|max:10'
  ]);

  $alumno = Alumno::create($request->all());
  return redirect('alumno/create')->with('status', 'Alumno creado OK');
}
```

MÉTODO 3.- FORM REQUEST

Para la creación de un customizable Form Request, se utilizará el comando del cliente de artisan `make:request`. Al ejecutar este comando, se creará (de no existir) un directorio en `app/Http/Requests/[nombredelaclase.php]`

```
php artisan make:request CreateAlumnoRequest
```

La clase resultante está compuesta básicamente por dos métodos: `authorize` y `rules`.

Para que no haya problema de credenciales y que todo usuario pueda hacer uso del formulario se habilita a verdadero el método `authorize`.

```
// Determina si el usuario está autorizado para enviar el request
public function authorize()
{
  return true;
}
```

En `rules` se escribirán las reglas que deberá cumplir cada variable pasada desde el formulario.

```
// Se aplican las reglas de validación al request
public function rules()
{
  return [
    'nombre' => 'required|max:191',
    'materia' => 'required|max:255',
    'nota' => 'required|integer|min:0|max:10'
  ];
}
```

Y ahora en el controlador se enviará un objeto de esta clase que se acaba de crear

```
// @vars $request
public function store(CreateAlumnoRequest $request)
{
  $alumno = Alumno::create($request->all());
  return redirect('alumno/create')->with('status', 'Alumno creado OK');
}
```

Como detalle, en el controlador se deberá agregar la ruta de la clase en la cabecera:

```
use App\Http\Requests\CreateAlumnoRequest;
```

DEVOLVER ERRORES PERSONALIZADOS

Con Form Request se puede personalizar los mensajes de error de `Validate`. Ya que por defecto, Laravel utiliza un conjunto de cadenas a las que tendría que traducirse en la carpeta `lang`…

Simplemente, en la clase `CreateAlumnoRequest` creada anteriormente se sobrescriben los mensajes con la función `messages()`. Esta función devolverá un array de pares de cadenas asociativas para cada uno de los '`campo.regla`':

```
// Devuelve un mensaje por cada atributo erróneo
public function messages()
{
  return [
    'nombre.required' => 'Olvidaste el nombre del alumno',
    'materia.required' => 'Indica que materia registrar',
    'materia.max' => 'Te has pasado de caracteres, solo 255',
    'nota.min' => 'La nota como mínimo debe tener 0',
    'nota.max' => 'La nota como máximo debe tener 10',
    'nota.integer' => 'La nota debe ser una cifra entre 0 y 10'
  ];
}
```

`Validate()` envía un conjunto de errores (de tenerlos) a la vista, los cuales se pueden recorrer y generar un listado de errores con `@foreach` o segregarlos por cada uno de los campos que le corresponda.

Por ejemplo, en el campo del nombre, habrá que imprimir el error que corresponda obviamente a 'nombre', accediendo a su primer error encontrado:

```
<input type="text" name="nombre" value="{{ old('nombre') }}" required>

@if ($errors->has('nombre'))
  {{ $errors->first('nombre') }}
@endif
O bien, si queremos embellecerlo con BootStrap:

<input type="text" class="form-control{{ $errors->has('nombre') ? ' is-
invalid' : '' }}" name="nombre" value="{{ old('nombre') }}" required>

@if ($errors->has('nombre'))
  <span class="invalid-feedback">
    <strong>{{ $errors->first('nombre') }}</strong>
  </span>
@endif
```

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar