LARAVEL: INSTALACIÓN, BLADE Y VIEWS

- **Requisitos.** Php 8 con las siguientes extensiones habilitadas (Ctype, cURL, DOM, FileInfo, Filter, Hash, MbString, OpenSSL, PCRE, PDO, Session, Tokenizer, XML)
- **Entorno de desarrollo.** IDE (PHPStorm, Visual Studio Code), servidor web (Apache o Nginx), base de datos (MySQL), gestor de BBDD (PHPMyAdmin), Node.js, Composer, Artisan (para CLI)
    - Para trabajar con este framework:
    - Xampp (requiere instalación de componentes adicionales)
    - VirtualBox con ubuntu, Apache, MySQL y PHP (instalar todo manualmente)
    - Laravel Herd ← **RECOMENDADO - PARCIALMENTE GRATUITO**
    - Laragon ← **RECOMENDADO - GRATUITO**

- **Crear nuevo proyecto.**Via powershell o cmd: **laravel new [Nombre_web].** Nos preguntará si deseamos instalar un **paquete inicial:**
    - none (ningún paquete)
    - **breeze:** autenticación básica
    - **jetstream:** autenticación avanzada (comprobación de email, doble factor…)

  Si seleccionamos un paquete también habrá que indicar su stack. **Blade** es el motor de plantillas de Laravel.
  También se debe indicar si se desea soporte para modo oscuro.Después se debe elegir un **framework para tests:** PHPUnit o **Pest.**

  Por último, se debe elegir una **base de datos**:
    - MYSQL-> **RECOMENDADO**
    - PostgreSQL
    - SQLite ← por defecto
    - SQL Server

- **Cambiar la base de datos por defecto. La BBDD por defecto es SQLite** aunque indiquemos que el proyecto trabajará con otra (como MySQL). Para cambiar la BBDD se debe:
    1. Abrir el archivo **.env** que se ha generado para el proyecto.
    2. Buscar los parámetros de "DB", como "DB_CONNECTION" y cambiarlos según corresponda. Si trabajamos en local, la configuración para MySQL debería ser:

       DB_CONNECTION=mysql
       DB_HOST=127.0.0.1
       DB_POST=3306
       DB_DATABASE=[Nombre_Web]
       DB_USERNAME=root
       DB_PASSWORD=[Contraseña_root]

    3. En CMD, ejecutar: php artisan migrate. Crear la base de datos si no existe. ¡Ojo! Esto NO migra datos de usuario

- **Funcionamiento**

    1. Cliente realiza petición vía navegador.
    2. Laravel enruta la petición entrante según sus características.
    3. La petición llega a un controlador.
    4. El controlador carga, según corresponda, una vista para ser devuelta al cliente como respuesta.

- **Rutas en Laravel - Route.** Las rutas son lo primero que se carga al acceder a una aplicación web de Laravel. Estas rutas se definen en la carpeta routes, que contendrá los ficheros de php: auth, console, web. Nos centramos en web.php ya que define cómo se va a desplegar la web al acceder a la misma.
    - ➢ **web.php.** Mediante el fichero web.php se define la página web a cargar. Para ello lo primero que se realiza es una llamada a la **clase Route.** Esta clase de rutas, "Route" ofrece respuestas a peticiones de usuarios mediante métodos propios. Algunos de estos métodos son: get(), view(), post(), patch(), put(), delete(), options(), match() - para responder a varias peticiones simultáneamente, any() - para responder a todo tipo de peticiones.

- **Vistas en Laravel.** Las respuestas resultantes que se darán a los clientes tras procesar sus peticiones entrantes serán vistas (views): **páginas web construidas según la interacción realizada.** Para construir estas vistas se emplea el motor de plantillas blade (de aqui la extensión *.**blade.php**). Las vistas generadas por defecto están en la carpeta profile (en el ejemplo; welcome.blade.php). Esta carpeta a su vez está en resources→views. Todas estas vistas se pueden cargar mediante las funciones
  get(), view(), post()...

- **Rutas en Laravel - Route - web.php - get()**
  use App\Http\Controllers\ProfileController:
  use Illuminate\Support\Facades**Route**;

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

  El método **get()** tendrá dos parámetros de URL y una página web (view, código HTTP). Sirve para indicar qué vista (view) se debe mostrar según la ruta a la que accede el usuario.

- **Rutas en Laravel - Route - web.php - view().** Este método también responde a las peticiones de get y head. Esto significa que también se puede escribir: **Route::view**('/', 'welcome');

Útil para casos en los que no se requiere aplicar una lógica de negocio, solo mostrar una web.

- **Rutas en Laravel - Route - web.php - Nombrar rutas.** Es posible dar nombre a las rutas de web.php para trabajar solo con esta nomenclatura al crear enlaces (sin necesidad de indicar la URL específica a la web).

  Route::view('/', 'welcome')**->name('landing');**

  Para acceder a esta ruta se puede emplear la referencia: route('landing'). Ejemplo:
  <a href=" <?php echo **route('landing')** ?> ">Welcome

  Recomendable siempre dar nombre a las rutas. Si se intenta llamar a una ruta sin nombre definido, dará error.

- **Rutas en Laravel - Route - web.php - Orden (avanzado).** El orden con el que se definen las rutas dentro de web.php es **importante** (sobretodo cuando se trabaja con datos traídos de BD y/o variables).

  Ejemplo de rutas ordenadas (para /chat/):

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

  Route::get('**chat**', [MensajeController::class, 'index'])->name('chat.index');
  Route::get('**/chat/create**', [MensajeController::class, 'create'])->name('chat.create');
  Route::get('**/chat/{mensaje**}', [MensajeController::class, 'show'])->name('chat.show');

  {mensaje} contendrá Id's de mensajes. Por lo general, **poner al final las rutas que reciban parámetros variables.**

- **Rutas en Laravel - Route - ver rutas con artisan.** Es posible ver un listado de las rutas disponibles por comandos empleando artisan (ubicado en la carpeta de proyecto): **php artisan route:list**

- **Motor de plantillas de Laravel - Blade.** Blade Template Engine nos permite, entre otras funcionalidades, escribir código PHP dentro de HTML de forma más clara y descriptiva:

  Welcome

  {{ y }} en lugar de **<?php y ?>**.

  Blade traduce a PHP automáticamente (compilando las vistas). Se puede ver el resultado en la carpeta storage/framework/views. Laravel identifica archivos de blade cuando su extensión es: [NOMBRE_FICHERO].**blade**.php

- **Motor de plantillas de Laravel - Blade - {{ y }}**

  {{ y }} en lugar de **<?php y ?>**. ¡OJO! No son exactamente lo mismo.
  Blade convierte a:

  <a href=" <?php echo **e**(route('landing')) ?> ">Welcome

  La función **e()** propia de Laravel realiza escape de etiquetas HTML para evitar inyección de código (htmlespecialcharacters()).

  Es posible evitar este escape mediante **{!! y !!}**. La conversión con estos es literal:

  <a href=" <?php echo route('landing') ?> ">Welcome

- **Motor de plantillas de Laravel - Blade - comentarios**

  {{-- para iniciar un comentario y --}} para cerrar.
  Ejemplo: {{-- este texto esta comentado --}}

- **Rutas en Laravel - Route - Controladores**. Para pasar variables mediante Route se puede emplear el tercer parámetro opcional del método view (ejemplo partiendo de web.php):

  $mensajes = [['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje']];
  Route::view('chat', 'chat', **['mensajes' => $mensajes]**)->name('chat');

  Para pasar variables mediante Route se puede emplear el segundo parámetro del método get, que será un invocable o una función (ejemplo partiendo de web.php):

  ```
  Route::get('chat', function () {
          $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];
          return view('chat', ['mensajes' => $mensajes ]);
  } )->name('chat');
  ```

  Para inspeccionar estos datos en chat.blade.php: **@dump($mensajes)**

  Equivalente a var_dump() de PHP. Para iterar sobre los datos pasados a chat.blade.php mediante la directiva foreach de Laravel:

  ```
  @foreach( $mensajes as $thisMensaje )
          <h2> {{ $thisMensaje['titulo'] }} </h2>
  @endforeach
  ```

  También se puede emplear un foreach de PHP. Para mantener limpio el archivo de rutas web.php, es recomendable utilizar controladores.

  Siempre que necesitemos realizar alguna acción entre la petición y la respuesta, es preferible utilizar un controlador.

Un controlador es una clase de PHP que se encarga de recibir la petición y gestionar la respuesta. El método view() és un controlador.

Ver función view() en la definición de la clase ruta, en Route.php: delega sobre “ViewController” de Laravel (ViewController.php). Solo tiene el método invoke.

Los controladores se definen en **app/Http/Controllers**. Por convención la primera letra de cada palabra se escribe en mayúsculas, y siempre termina con
“Controller.php”.

Ejemplo de Controlador MensajeController.php:

```
namespace App\Http\Controllers;
class MensajeController{
}
```

En web.php:

```
use App\Http\Controllers\MensajeController;
Route::get('chat', MensajeController::class )->name('chat');
```

★ **Controlador __invoke ( )**

Ejemplo de Controlador MensajeController.php con método invoke para hacerlo “invocable”:

```
namespace App\Http\Controllers;
class MensajeController{

public function __invoke(){
        $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];
        return view('chat', ['mensajes' => $mensajes ]);
    }
}
```

★ **Controlador: Métodos propios**

Ejemplo de Controlador MensajeController.php con método propio:

```
namespace App\Http\Controllers;
class MensajeController{
    public function index(){
        $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];
        return view('chat', ['mensajes' => $mensajes ]);
    }
}
```

En web.php:

```
Route::get('chat', [ MensajeController::class, 'index' ] )->name('chat');
```

Los métodos necesarios se pasan con la clase como parámetros en array.

★ **Crear con Artisan**

Ejemplo de generación de controlador MensajeController.php por comandos empleando artisan (ubicados en la carpeta de proyecto):

php artisan make:controller MensajeController

- Añadir **-i** (invoke) para que sea invocable (añade método __invoke())
- Añadir **-r** (resource) para que incluya los métodos index(), create(), store(), show(), edit(), update(), destroy()
- Opción **--api** para que incluya los métodos index(), store(), show(), update(), destroy()

- **Rutas en Laravel - Controladores y formularios**

Continuando con el ejemplo MensajeController.php, definir método create():

```
namespace App\Http\Controllers;
class MensajeController{
    public function create(){
        return view('chat.create');
    }
}
```

Esto requiere una vista create.blade.php creada en una subcarpeta llamada “chat”.

El contenido de create.blade.php es:

```
<x-layout>
    <h1>Nuevo mensaje</h1>
    <form method="POST" action="{{ route('chat.store') }}">
        @csrf
        <label>Titulo: <input type="text" name="title" /></label>
        <label>Mensaje: <br><textarea name="message"></textarea></label><br>
```

```
                <button type="submit">Enviar</button>
        </form>
        <a href="{{ echo route('chat.index') }}">Atras</a>
</x-layout>
```

IMPORTANTE: se está empleando un componente creado anteriormente para reutilizar el código. Su uso no es obligatorio.

Definir las rutas para estas vistas en web.php es:

Para mostrar el formulario (con get):
Route::**get**('**/chat/create**', [MensajeController::class, 'create'])->name('chat.create');

Para recoger los datos del formulario (via POST):
Route::**post**('**/chat**', [MensajeController::class, 'store'])->name('chat.store');

Añadir el método store() a MensajeController.php:

```
namespace App\Http\Controllers;
class MensajeController{
        public function store(){
                return request();       // Convierte los datos entrantes automáticamente a json
        }
}
```

Nota: no requiere crear la vista store.blade.php.

Alternativa (buena práctica) para el método store() de MensajeController.php:

```
namespace App\Http\Controllers;
namespace Illuminate\Http\Request;
class MensajeController{
        public function store(Request $request){
                return $request;
        }
}
```

Para acceder a los valores del formulario individualmente, en concreto, al campo de título (title), dentro del método store() de MensajeController.php:

```
namespace App\Http\Controllers;
namespace Illuminate\Http\Request;
class MensajeController{
        public function store(Request $request){
                return $request->input('title');
        }
}
```

- ★ **CSRF.** La directiva **@csrf** es para prevenir de ataques CSRF. En Laravel, con formulario de método POST se debe añadir siempre. **Cross-Site Request Forgery (CSRF)** es un tipo de ataque que **engaña a un usuario para que realice acciones en una aplicación web sin su consentimiento** o conocimiento. Esto puede llevar a acciones no autorizadas como transferir fondos, cambiar contraseñas u otras operaciones no deseadas por el cliente/usuario.
  Laravel añade token oculto para verificar origen del POST. Este token es valido durante 2 horas (valor por defecto, configurable vía ".env").
  Es posible visualizarlo desde el navegador (inspeccionar).

CLASES PROPIAS DE LARAVEL MÁS RELEVANTES

1. **Illuminate\Http\Request**. Representa y gestiona las solicitudes HTTP entrantes, permitiendo acceder a datos como parámetros, cabeceras, cookies y archivos subidos.
2. **Illuminate\Http\Response**. Representa y gestiona las respuestas HTTP enviadas al cliente, permitiendo personalizar el contenido, cabeceras y códigos de estado.
   - En el middleware, se prefiere usar **Symfony\Component\HttpFoundation\Response** porque es más estándar y flexible para manejar la respuesta HTTP. Ambas clases sirven para manejar respuestas.
3. **Illuminate\Http\RedirectResponse**. Representa una respuesta HTTP de redirección, utilizada para redirigir al usuario a otra URL, ya sea interna o externa, y permite personalizar cabeceras, datos de sesión y códigos de estado HTTP.
4. **Illuminate\Support\Facades\Route**. Maneja la definición y gestión de rutas, proporcionando métodos para crear rutas y acceder a información relacionada.
5. **Illuminate\Support\Facades\Auth**. Gestiona la autenticación de usuarios, ofreciendo métodos para login, logout y verificación de usuarios autenticados.

6. **Illuminate\Support\Facades\DB**. Proporciona acceso a la base de datos, permitiendo ejecutar consultas SQL y usar el constructor de consultas.
7. **Illuminate\Support\Facades\Validator**. Maneja la validación de datos, ofreciendo reglas predefinidas y personalizables para verificar entradas de usuarios.
8. **Illuminate\Support\Facades\Session**. Permite gestionar datos de sesión de forma sencilla, como almacenar, recuperar o eliminar valores.
9. **Illuminate\Support\Facades\Storage**.Facilita el manejo de archivos en diferentes sistemas de almacenamiento, como locales, S3 o FTP.

Se puede hacer uso de estas clases llamandolas con **use.**

LARAVEL - VIEWS (GENERAR VISTAS EMPLEANDO BLADE)

- **Views.** Las vistas proporcionan una forma conveniente de **colocar el código HTML en archivos separados.** Las vistas **separan la lógica del controlador** o de la aplicación **de la lógica de presentación** y se almacenan en el directorio **resources/views.**
  Al usar Laravel, las plantillas de vista suelen estar escritas con el lenguaje de plantillas **Blade**.

- **Views - Ejemplo sencillo**

  <!-- View almacenada en resources/views/**greeting.blade.php** -->

  <html>

  <body>

  <h1>Hola, {{ $name }}</h1>

  </body>

  </html>

  Para cargar la vista anterior se puede configurar, en web.php, la ruta siguiente:

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

  return view('greeting', ['name' => 'Juan Luis']);

  });

- **Crear una vista.** Puedes crear una vista **colocando un archivo con la extensión .blade.php en el directorio resources/views** de tu aplicación **o** utilizando el **comando Artisan make:view: php artisan make:view greeting.** La extensión **.blade.php** indica al framework que el archivo **contiene una plantilla** Blade. Las plantillas Blade incluyen HTML y directivas de Blade, que permiten mostrar valores fácilmente, crear estructuras condicionales "if", iterar sobre datos y mucho más.

- **Vistas en subcarpetas.** Las vistas también pueden estar anidadas dentro de subdirectorios, dentro del directorio resources/views. Se puede utilizar la **notación de "punto" (.) para hacer referencia** a la ruta de las vistas. Por ejemplo, si tu vista se encuentra en resources/views/admin/profile.blade.php, puedes traerla desde una de las rutas (en web.php) o controladores de tu aplicación mediante:
  return view('**admin**.profile', $data); Nota: estos directorios **NO deben contener** . en su nombre.

  Pasar datos a las vistas. Como se ha visto en el ejemplo previo, se puede pasar datos mediante un array. return view('greetings', **['name' => 'Juan Luis', 'occupation' => 'informático']**);

  Al pasar información de esta manera, los datos deben ser un **array con pares clave => valor.**

  Después de proporcionar datos a una vista, se puede **acceder a cada valor dentro de la vista usando las claves del array,** por ejemplo:

  <?php echo **$name**; ?>

  - ★ **Método WITH.** Como alternativa a pasar un array completo de datos a la función view, se puede utilizar el **método 'with' para agregar datos individuales a la vista.**
    with() devuelve una instancia del objeto de vista, lo que **permite encadenar** métodos antes de retornarla:

    return view('greeting')
    ->with('name', Juan Luis)
    ->with('occupation', 'informático');

- **Trabajar con datos pasados**

  <!-- View almacenada en resources/views/greeting.blade.php →

  <!-- Laravel reemplazará {{ $name }} y {{ $occupation }}
  con los valores proporcionados en el array. -->

  <html>

  <body>

<h1>Hola, {{ $name }}</h1>

  Tu ocupación es: {{ $occupation }}

</body>

</html>

- **Optimizar las vistas.** Con Blade, **las vistas se compilan bajo demanda.**
  Cuando se ejecuta una solicitud que renderiza (carga) una vista, **Laravel verifica si existe una versión compilada** de la vista. Si el archivo existe, Laravel comprobará si la vista ha sido modificada recientemente, en comparación con la versión compilada. **Si la vista compilada no existe o la versión sin compilar ha cambiado, Laravel recompilará la vista.** La compilación de vistas durante la solicitud puede tener un pequeño impacto en el rendimiento. Por ello, Laravel proporciona el **comando Artisan view:cache** para **pre-compilar todas las vistas** utilizadas por la aplicación. Para mejorar el rendimiento, se puede ejecutar este comando como parte del proceso de despliegue: **php artisan view:cache.** Por el contrario, para limpiar la cache de vistas, se puede usar el siguiente comando: **php artisan view:clear**

LARAVEL - MIDDLEWARE

- **¿Qué es?.** Se trata de un **mecanismo para inspeccionar y filtrar peticiones HTTP**. Intercepta peticiones entrantes y "pre-procesa". Ejemplo; Laravel incluye un "middleware" para verificar que el usuario de la aplicación se ha autenticado. Al llegar una petición, si el usuario se ha identificado, el proceso continúa con normalidad (la petición se redirige donde deba). En caso contrario, middleman redirige a la página de login. Otro ejemplo de middleware incluido con Laravel es la directiva @CSRF, para la protección de datos provenientes de formularios.

- **Crear un Middleware.** Todos los middleware creados (no los incluidos por defecto) se deben guardar en: **app/Http/Middleware**. Para crear un Middleware se puede emplear la siguiente instrucción de artisan: ***php artisan make:middleware*** *[NombreMiddleWare]*
  Ejemplo:

  php artisan make:middleware EnsureTokenIsValid

  Este comando crea una clase "EnsureTockenIsValid" en
  app/Http/Middleware.

- **Ejemplo Middleware - EnsureTokenIsValid**

```
<?php
namespace App\Http\Middleware;
use Closure; // clase anónima que permite definir funciones o callbacks sin nombre
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid {
   // Controlar peticion entrante. Requiere parámetros de clase Request y clase Closure (respuesta)
   // @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)  $next

 public function handle(Request $request, Closure $next): Response {
      // código del middleware - realiza acción
      if ($request->input('token') !== 'mi-token-secreto') {
         return redirect('/home');                    // middleware DETIENE EL FLUJO
      }
      // Middleware cede el paso
      return $next($request);                                  // middleware permite que la solicitud continue
   }
}
```

  El método se llama **handle()** porque se encarga de "**manejar**" o procesar la solicitud entrante en el middleware. Este método permite a los middleware realizar su tarea (como la validación del token en este caso) y luego decidir si la solicitud debe continuar o si debe

redirigirse o rechazarse.

Continuando con el código anterior, sobre los parámetros de handle():

- **Request $request** es una instancia de la clase *Illuminate\Http\Request*, que contiene todos los datos de la solicitud HTTP que está siendo procesada
- La función/**método $next** representa el siguiente paso o acción a realizar en la cadena de middleware después de que el middleware actual haya ejecutado su lógica.
- **Closure** se utiliza como tipo de parámetro para la función $next. Es una clase anónima que permite definir funciones o callbacks sin nombre.

Así pues, **$next($request)** pasa la solicitud al siguiente paso (middleware o controlador).

Si el token proporcionado no coincide con "mi-token-secreto", el middleware devolverá una redirección HTTP al cliente; de lo contrario, la solicitud se enviará más adelante en la aplicación.

Para pasar la solicitud a la aplicación (permitiendo que el middleware la "pase"), debe llamar al *callback* $next (tipo closure) con el $request.

Es mejor imaginar el **middleware como una serie de "capas" o barreras que las solicitudes HTTP deben atravesar** antes de llegar a la aplicación. **Cada capa puede examinar la solicitud e incluso rechazarla por completo.**

- **Middleware con acción post-procesado.** Se puede configurar que la acción del middleware se realice después del paso de la petición, enviando la petición antes y devolviendo la respuesta:

```
class AfterMiddleware {

public function handle(Request $request, Closure $next): Response {
    // Middleware cede el paso, pero no termina
    $response = $next($request);
    // código del middleware - realiza acción
    return $response;
  }

}
```

- **Uso de Middleware - registrar Middleware**. Hay dos formas de registrar un Middleware:

1. **Registrar globalmente**: siempre pasará automáticamente por toda petición entrante.
2. **Asignar a rutas** (Route): para revisar únicamente peticiones específicas.

- **Uso de Middleware - registrar Middleware globalmente**. Laravel posee un stack (listado) de middlewares global; es posible añadir middlewares generados a este listado, añadiendo sus clases en **bootstrap/app.php**. Por ejemplo, para añadir EnsureTokenIsValid:

```
use App\Http\Middleware\EnsureTokenIsValid;

->withMiddleware(function (Middleware $middleware) {
    $middleware->append(EnsureTokenIsValid::class);
})
```

El objeto $middleware es una instancia de Illuminate\Foundation\Configuration\Middleware.
Es el responsable de gestionar los middlewares asignados a todas las rutas de la aplicación. El método append() añade el middleware indicado al listado global de middlewares.

- **Uso de Middleware - registrar Middleware a rutas**. Si se desea a**signar middleware a rutas específicas**, se puede invocar el **método middleware()** al definir la ruta. Usando el ejemplo de EnsureTokenIsValid:

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {
    // código de la ruta…
})->middleware(EnsureTokenIsValid::class);
```

- **Middleware con Alias**. Es posible **registrar globalmente un Middleware y asignarle un Alias para poder incluirlo solo cuando sea necesario**. Para ello, al registrar globalmente, en lugar de incluirlo con append(), indicaremos un nombre a la clase del Middleware, dentro de un array, con **alias()**. Por ejemplo, para EnsureTokenIsValid, nuevamente en **bootstrap/app.php**:

use App\Http\Middleware\EnsureTokenIsValid;

->withMiddleware(function (Middleware $middleware) {
*$middleware->**alias(['ensureToken' => EnsureTokenIsValid::class]);***
})

Ahora, al definir una ruta con Middleware, es posible usar solo el alias:

Route::get('/profile', function () {
// código de la ruta…
*})->middleware('**ensureToken**');*

- **Alias por defecto**

Alias	Middleware
auth	Illuminate\Auth\Middleware\Authenticate
auth.basic	Illuminate\Auth\Middleware\AuthenticateWithBasicAuth
auth.session	Illuminate\Session\Middleware\AuthenticateSession
cache.headers	Illuminate\Http\Middleware\SetCacheHeaders
can	Illuminate\Auth\Middleware\Authorize
guest	Illuminate\Auth\Middleware\RedirectIfAuthenticated
password.confirm	Illuminate\Auth\Middleware\RequirePassword
precognitive	Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests
signed	Illuminate\Routing\Middleware\ValidateSignature
subscribed	\Spark\Http\Middleware\VerifyBillableIsSubscribed
throttle	Illuminate\Routing\Middleware\ThrottleRequests or Illuminate\Routing\Middleware\ThrottleRequestsWithRedis
verified	Illuminate\Auth\Middleware\EnsureEmailIsVerified

- **"Terminable" Middleware.** Consisten en **Middleware que ejecutan código una vez que ya se ha enviado una respuesta al navegador** del cliente. Se definen con el **método "terminate"** en la clase del middleware. Requieren que el servidor web utilice FastCGI. FastCGI es un protocolo para mejorar el rendimiento de aplicaciones web al permitir una comunicación más rápida y eficiente entre el servidor web y las aplicaciones.

LARAVEL CONTROLLERS (LÓGICA DE GESTIÓN DE SOLICITUDES)

- **Controladores.** Organiza la **lógica de gestión** (o control) **de solicitudes** utilizando **clases "controller"**.
Por ejemplo, una clase 'UserController' podría gestionar todas las solicitudes entrantes relacionadas con los usuarios, incluyendo mostrar, crear, actualizar y eliminar usuarios (CRUD).
Por defecto, en Laravel los controladores se almacenan en el directorio '**app/Http/Controllers**'.

- **Crear un controlador**. Para generar rápidamente un nuevo controlador, puedes ejecutar el comando Artisan **make:controller**. Ejemplo para crear un controlador para gestionar solicitudes sobre "usuarios": ***php artisan make:controller** UserController.* También puedes crear un controlador manualmente en la carpeta app/Http/Controllers, con formato "[nombre]**Controller.php**". **Un controlador puede tener cualquier número de métodos públicos para responder a solicitudes HTTP entrantes.**

- **Ejemplo de controlador - UserController.php**

```
<?php
Namespace App\Http\Controllers;
use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller{        // Recomendable extender Controllers, no obligatorio

   /**  Mostrar perfil de un user.    */
   public function show(string $id): View  {
      return view('user.profile', [
         'user' => User::findOrFail($id)
      ]);
   }
}
```

- **Invocar UserController.php.** Una vez **escrito una clase de controlador** y un método, se puede **definir una ruta** (en web.php) hacia dicho método de la siguiente manera:
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);

Cuando una solicitud entrante coincida con la URI de la ruta especificada, se invocará el método 'show' de la clase

'App\Http\Controllers\UserController' y **los parámetros de la ruta se pasarán** al método.

- **Controladores de una sola acción - __invoke()**. Si solo se requiere controlar una única acción puede ser conveniente declarar una clase de controlador con un único método dedicado a dicha acción. Para lograr esto, se puede definir un único método **__invoke()** dentro del controlador:

  ```
  public function __invoke() {
      // lógica …
    }
  ```

  Cuando se declara un controlador de una única acción, no es necesario indicar el método en la ruta de llamada. Si UserController solo tuviera __invoke():

  Route::get('/user/{id}', [UserController::class]);

- **Controladores - métodos propios**

  Ejemplo de Controlador *UserController.php* con método propio:

  ```
  namespace App\Http\Controllers;
  class userController{
          public function index(){
                  $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];
                  return view('chat', ['mensajes' => $mensajes ]);
          }
      }
  ```

  En *web.php*:

  Route::get('chat', [UserController::class, 'index'])->name('chat');

  Los métodos necesarios se pasan con la clase como parámetros en array.

- **Controladores - crear con Artisan**. Ejemplo de generación de controlador *MensajeController.php* por comandos empleando artisan (ubicados en la carpeta de proyecto):

  ***php artisan make:controller** UserController*

  - Añadir ***-i*** o --invokable (invoke) para que sea invocable (añade método __invoke())
  - Añadir ***-r*** o --resource (resource) para que incluya los métodos index(), create(), store(), show(), edit(), update(), destroy()
  - Opción ***--api*** para que incluya los métodos index(), store(), show(), update(), destroy()

- **Controladores con Middleware.** Al invocar controladores definiendo la ruta es posible añadir Middleware:

  *Route::get('/profile', [UserController::class, 'show'])**->middleware('auth');***

  También se puede **registrar un Middleware desde la clase del controlador**, **con** un **método Middleware estático**:

  ```
  use Illuminate\Routing\Controllers\HasMiddleware;
  use Illuminate\Routing\Controllers\Middleware;
  class UserController extends Controller implements HasMiddleware {

    // Registrar Middleware que se debería ejecutar cuando lleguen peticiones al Controlador
    public static function middleware(): array    {
      return [
        'auth',
        new Middleware('log', only: ['index']),
        new Middleware('subscribed', except: ['store']),
      ];
    }
  }
  ```

- **Controladores de recursos - Resource**. Si se considera cada modelo Eloquent (**tablas** identificadas por el ORM) **de una aplicación como un "recurso"**, **es habitual realizar las mismas acciones CRUD** sobre cada recurso. Debido a este caso de uso común, Laravel **asigna las rutas típicas de crear, leer, actualizar y eliminar ("CRUD") a un controlador con una sola línea de código**. Como hemos visto, estos controladores se pueden crear empleando el tag de "-r" o --resources con el comando de artisan create:controller: *php artisan make:controller PhotoController **--resource***

  Este comando generará un controlador en **app/Http/Controllers** llamado *PhotoController.php*, que contendrá un método para cada operación del recurso:

  - index(): para mostrar listado de registros
  - show(): para mostrar un registro (requiere ID)
  - create(): para mostrar un formulario de creación de registro
  - store(): para guardar registro nuevo en el modelo
  - edit(): para mostrar formulario de edición de un registro existente (requiere ID)
  - update(): para actualizar un registro existente en el modelo (requiere ID)
  - destroy(): para eliminar un registro existente (requiere ID)

- **Controladores de recursos - Resource - Ruta**. Para definir rutas a los métodos de los controladores de registro, **basta con registrar una única ruta "Route::resource"** que apunte al controlador:

  use App\Http\Controllers\PhotoController;

  *Route::**resource**('photos', PhotoController::class);*

  Esta **única declaración de ruta crea múltiples rutas** para manejar las diversas acciones a realizar sobre el recurso. Además, siempre se puede obtener una visión general rápida de las rutas de la aplicación ejecutando el comando ***php artisan route:list***

Verb	URI	Action	Route Name
GET	/photos	**index**	photos.index
GET	/photos/create	**create**	photos.create
POST	/photos	**store**	photos.store
GET	/photos/{photo}	**show**	photos.show
GET	/photos/{photo}/edit	**edit**	photos.edit
PUT/PATCH	/photos/{photo}	**update**	photos.update
DELETE	/photos/{photo}	**destroy**	photos.destroy

- **Controladores de recursos - Recurso no localizado**. Por lo general, **se genera una respuesta HTTP 404 si un recurso** del modelo vinculado **no se encuentra**. Sin embargo, **se puede personalizar este comportamiento llamando al método missing() al definir la ruta al controlador** de recurso. Este método acepta una closure que se ejecutará si no se puede encontrar recurso para alguna de las rutas del recurso (es decir, ID de registro no localizado).

  Ejemplo de control personalizado para un recurso no localizado:

```
use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
    ->missing(function (Request $request) {
        // Redirige al método index() si no se encuentra el registro/recurso
        return Redirect::route('photos.index');
    });
```

- **Controladores de recursos - dar nombre.** Por defecto, **todas las acciones de los controladores de tipo resource tienen un nombre de ruta**; no obstante, s**e pueden sobrescribir estos nombres** pasando un array al **método names()** con los nombres de ruta deseados.

  Por ejemplo:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->names([
    'create' => 'photos.build'
]);
```

- **Controladores de recursos y el orden de las rutas**. Recordatorio: **El orden con el que se declaran las rutas importa**. Las rutas de controladores de recursos no son excepción; **si tenemos métodos propios en dichos controladores, conviene definirlos antes que** la definición de ruta de l**os métodos de recursos** "Route::resource()".

  Por ejemplo:

```
use App\Http\Controller\PhotoController;

Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

  Con esta declaración no existe riesgo de que la ruta photos/popular se confunda con un photos/{photos}.

- **Controladores - Inyeccion de métodos externos**. Los controladores permite incluir llamadas a métodos de otros controladores o clases. Esto puede servir, por ejemplo, para procesar datos y posteriormente redirigir a otra ruta.

  Si tenemos la siguiente Ruta para el controlador UserController:

  use App\Http\Controllers\UserController;

  Route::put('/user/{id}', [UserController::class, 'update']);

  Interesa que, tras realizar el update, se redirija al usuario devuelta a la primera página "/users".

```
namespace App\Http\Controllers;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller {
```

```
    // Método Update the un user dado.
    public function update(Request $request, string $id): RedirectResponse {          // Clase Request ANTES de los parámetros pasados
      // Buscar y actualizar al usuario
      $user = User::findOrFail($id); // Lanza un error si no encuentra el usuario
      $user->name =  $request->name;
      // repetir para el resto de datos de User
      $user->save(); // Guarda los cambios en la base de datos
      // Tras finalizar la actualización, redirige al listado de usuarios - indicando estado y mensaje
      return redirect('/users')->with('success', 'Usuario actualizado correctamente.');
    }
}
```

LARAVEL: REQUESTS Y RESPONSES (PETICIONES Y RESPUESTAS + COOKIES)

- **Request - Solicitudes.** La clase ***Illuminate\Http\Request*** de Laravel proporciona una forma orientada a objetos de **interactuar con la solicitud HTTP** actual que está siendo procesada por la aplicación, así como poder recuperar los **datos introducidos** por el usuario, **cookies y archivos** enviados con la solicitud. Para poder acceder a la instancia de la petición actual, se debe hacer uso de esta clase Request al procesar rutas o llamar a métodos de de controladores: ***use*** *Illuminate\Http\Request;*
- **Request - Ejemplo acceso vía ruta**

  El siguiente ejemplo define una ruta en *web.php*:

```
<?php
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {

    // procesado de Request según se requiera

return view('welcome'); // carga de la vista que se requiera
});
```

- **Request - Ejemplo acceso vía controlador**

  El siguiente ejemplo define un método *store()* en un controlador *UserController.php*:

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

class UserController extends Controller {

    public function store(Request $request): RedirectResponse {

        $name = $request->input('name');          // Extracción de datos procedentes de la petición del usuario

        // Código para guardar el usuario
       return redirect('/users');          // redirección tras finalizar el guardado
    }
}
```

- **Request - Parámetros de ruta**. Si el método de un controlador también espera recibir un parámetro de una ruta (ej. ID de un registro), se deben listar los parámetros de ruta después de las demás dependencias.

  Por ejemplo, si la ruta está definida de la siguiente manera:

```
use App\Http\Controllers\UserController;

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

  Desde el controlador, se puede usar la clase Request y acceder al valor del parámetro {id} definido en la ruta entrante

  El siguiente ejemplo define un método *update()* en un controlador *UserController.php*:

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

class UserController extends Controller {
    public function store(Request $request, string $id): RedirectResponse {
      $name = $request->input('name');          // Extracción de datos procedentes de la petición del usuario
```

```
    $thisUser = User::findOrFail($id);// Recuperación de instancia representada por id facilitado en /user/{id}

 // Código para comparar valores y actualizar el usuario
     return redirect('/users');            // redirección tras finalizar la actualización de datos
   }
}
```

- **Request - métodos propios - Path.** El método ***path()*** devuelve la **información de la ruta** de la solicitud/petición. Por lo tanto, si la solicitud entrante está dirigida a **http://example.com/foo/bar**, el método path() devolverá **foo/bar**. Ejemplo de uso: *$uri = $request->path();*

- **Request - métodos propios - is y routeIs.** El método ***is()*** permite **verificar si la ruta de la solicitud entrante coincide con un patrón dado**. Se puede usar el carácter * como un comodín al utilizar este método. El siguiente ejemplo será *true* y accede al *if* si la petición entrante viene de una ruta /admin/[lo_que_sea]:

  if ($request->is('admin/')) { // … }*

  El método ***routeIs()*** permite **verificar si la ruta de la solicitud entrante coincide con un nombre de ruta dado**. Se puede usar el carácter * como un comodín al utilizar este método. El siguiente ejemplo será *true* y accede al *if* si la petición entrante viene de una ruta con *->name('admin')*: *if ($request->routeIs('admin.*')) { // … }*

- **Request - métodos propios - URL**. Para obtener la **URL completa de la solicitud entrante**, se pueden usar los métodos ***url()*** o ***fullUrl()***. El método *url()* devolverá la **URL sin la cadena de consulta**, mientras que el método *fullUrl()* **incluirá la cadena de consulta**.

  Ejemplos:
  $url = $request->url();
  $urlWithQueryString = $request->fullUrl();

  - ➔ **Añadir datos a la cadena de consulta.** Con el método ***fullUrlWithQuery()*** es posible obtener una URl que combine la cadena de consulta de la petición con un array de datos. Ejemplo: *$request->fullUrlWithQuery(['type' => 'phone']);*

- **Request - métodos propios - method e isMethod**. El método ***method()*** devolverá el tipo de la solicitud HTTP realizada (ej. Get, Post, Put…). Por otro lado, se puede usar el método ***isMethod()*** para verificar que el tipo de solicitud HTTP coincide con una cadena dada.

  Ejemplos:

  $method = $request->method();   // devuelve el tipo de la solicitud

  if ($request->isMethod('post')) { // Accede al if si la petición es tipo POST
  }

- **Request - Headers.** Se puede **obtener un encabezado de la solicitud** desde la instancia de *Illuminate\Http\Request* utilizando el método ***header()***. Si el encabezado no está presente en la solicitud, se devolverá *null*. Sin embargo, el método header() acepta un segundo argumento opcional que será devuelto en caso de que el encabezado no esté presente en la solicitud. También se puede verificar si la solicitud contiene un determinado encabezado, con el método ***hasHeader()*** e indicando la cabecera a buscar. Ejemplo: *$value = $request->header('X-Header-Name');*

- **Encabezados HTTP [RECORDATORIO].** Los encabezados ayudan a personalizar, autenticar y estructurar la **comunicación entre cliente y servidor**. Transmiten detalles importantes como el tipo de contenido enviado, el formato esperado de la respuesta, credenciales de autenticación, información sobre la conexión, entre otros. Por ejemplo:
  - **Content-Type**: Indica el tipo de contenido (por ejemplo; JSON, HTML).
  - **Authorization**: Incluye credenciales para autenticar al cliente.
  - **User-Agent**: Identifica el software que realiza la solicitud (navegador).
  - **Accept**: Define los formatos que el cliente puede procesar.

- **Request - Dirección IP**. El método ***ip()*** puede ser utilizado para **obtener la dirección IP del cliente** que realizó la solicitud a la aplicación. También se puede usar *ips()* para obtener un array de IPs que incluye todas las direcciones IP del cliente que fueron reenviadas por proxies. En general, las direcciones IP **deben considerarse como datos no confiables, controlados por el usuario**, y solo utilizarse con fines informativos. Ejemplo: *$ipAddresses = $request->ips();*

- **Request - Input - Recuperar datos del cliente/usuario**. Se pueden **recuperar todos los datos** pasados con la solicitud entrante como un array utilizando el método ***all()***.

  Ejemplo: *$input = $request->all();*

  Alternativamente, también se puede llamar al método ***input()*** **sin argumentos** para **obtener todos los valores de entrada como un array asociativo**.

  Ejemplo: *$input = $request->input();*

  Para recuperar UN valor de entrada específico, se puede acceder a los datos introducidos por el usuario desde la instancia de Illuminate\Http\Request sin preocuparse por qué tipo de solicitud HTTP se utilizó. Esto se realiza mediante el método ***input() y el nombre del campo buscado***. También sirve para JSON.

  Ejemplo: *$name = $request->input('name');*

  Se puede pasar un valor predeterminado como segundo argumento: dicho valor será devuelto si el valor buscado sobre la solicitud no está presente o es null.

  Ejemplo: *$name = $request->input('name', 'Nombre_por_defecto');*

- **Request - Collection - Recuperar datos del cliente/usuario.** Usando el método ***collect()***, también se puede obtener todos los datos de

entrada de la solicitud entrante como una colección de datos de Laravel (clase propia *Collection*).

- **Request - Query - Recuperar datos del cliente/usuario.** Es posible recuperar datos de la cadena de la consulta mediante el método ***query()***.

  Ejemplo: *$name = $request**->query('name');*** También se puede llamar al método *query()* **sin argumentos** para **obtener todos los valores de la cadena de consulta como un array asociativo**.

- **Request - Recuperar datos del cliente/usuario por tipo**. Si los valores entrantes son de un tipo específico, se pueden recoger como tal:
  - **string** *$name = $request**->string**('name')*
  - **integer** *$numPage = $request**->integer**('num_page');*
  - **boolean** *$valid = $request**->boolean**(valid);*
  - **date** (segundo parametro define formato, tercero la zona horaria) *$elapsed = $request**->date**('elapsed', '!H:i', 'Europe/Madrid');*
- **Request - Condicionales según los datos de la solicitud**

  ★ Método ***has()***: **Determina si un valor está presente** en la solicitud. Ejemplo:
  *if ($request**->has**('name')) { // … }*
  Con un array, determina si todos los valores especificados están presentes:
  *if ($request->has**(['name', 'email'])**) { // … }*

  ★ Método ***hasAny()***: Devuelve **true si alguno de los valores especificados está presente** en la solicitud.
  *if ($request**->hasAny**(['name', 'email'])) { // … }*

  ★ Método ***whenHas()***: Ejecuta un cierre (closure) si un valor está presente. Puede verse como un has() en un *if*:
  *$request**->whenHas**('name', function (string $input) { // … }*

  ★ Método ***filled()***: Determina si un **valor está presente y no es una cadena vacía**. Equivalente a **!**is_empty() de PHP:
  *if ($request**->filled**('name')) { // … }*

  ★ Método ***isNotFilled()***: Determina si un **valor falta o es una cadena vacía**. Equivalente a is_empty() de PHP:
  *if ($request->**isNotFilled**('name')) { // … }*

  ★ Método ***anyFilled()***: Devuelve **true si alguno de los valores especificados no es una cadena vacía**.
  *if ($request**->anyFilled**(['name', 'email'])) { // … }*

  ★ Método ***whenFilled()***: Ejecuta un cierre (closure, entra en la función declarada) **si un valor está presente y no es una cadena vacía**:
  *$request**->whenFilled**('name', function (string $input) { // … });*

  ★ Métodos ***missing()*** y ***whenMissing()***: Para determinar si una clave dada esta ausente dentro del Requesta.
  *if ($request**->missing**('name')) { // … }*

- **Request - Cookies**. Todas las **cookies creadas por el framework Laravel** están **cifradas y firmadas con un código de autenticación**, lo que significa que serán consideradas inválidas si han sido modificadas por el cliente. Para obtener el valor de una cookie de la solicitud, se utiliza el método cookie en una instancia de *Illuminate\Http\Request*: $value = $request**->cookie**('name');

  NOTA: En Laravel las cookies se crean utilizando el método *cookie()* de la clase ***Illuminate\Http\Response***.

- **Request - Files - Ficheros**. Se pueden recuperar archivos subidos desde una instancia de Illuminate\Http\Request utilizando el método ***file()***. El método *file()* **devuelve una instancia de la clase Illuminate\Http\UploadedFile**, la cual extiende la clase PHP *SplFileInfo* y proporciona una variedad de métodos para interactuar con el archivo.

  Ejemplo de recuperación de un fichero: *$file = $request**->file**('photo');* La clase UploadedFile incluye métodos para guardar el fichero; *store()* y *storeAs()*.

- **Response: Respuesta básica a Solicitud.** Todas las **rutas y controladores deben devolver una respuesta** que será enviada **al navegador** del cliente. Laravel ofrece varias formas diferentes de devolver respuestas. La respuesta más básica consiste en devolver una cadena de caracteres (string) desde una ruta o un controlador. El framework convertirá automáticamente este string en una respuesta HTTP completa. Por ejemplo, desde web.php:

```
Route::get('/', function () {
        return 'Hola Mundo';
});
```

- **Response - Instancia de Respuesta a Solicitud.** Sin embargo, tras procesar una solicitud, una aplicación no devolverá solo un string o array; devolverá vistas (*views*) o instancias de ***Illuminate\Http\Response***. Devolver una instancia completa de ***Response*** permite personalizar el código de estado HTTP y los encabezados de la respuesta.

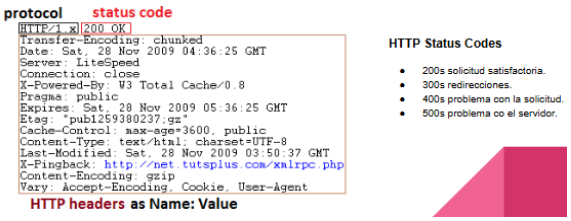
  Por ejemplo, en *web.php* se puede definir una ruta con la siguiente respuesta:

```
Route::get('/home', function () {
   return response('Hello World', 200)
          ->header('Content-Type', 'text/plain');
});
```

  ➔ ***Header.*** En el ejemplo anterior, mediante el método ***header()*** se puede modificar los datos de la cabecera de la respuesta. Este método **se puede encadenar** para aplicar múltiples configuraciones sobre el encabezado:
  return response($content)

*->**header**('Content-Type', $type)*
*->**header**('X-Header-One', 'Header Value')*
*->**header**('X-Header-Two', 'Header Value');*

➢ ***Encabezados HTTP [RECORDATORIO 2].*** En una **respuesta**, los encabezados pueden incluir el **tipo de contenido, el código de estado HTTP, o la fecha de expiración de los datos**.



- **Response - Ficheros (visualización o descarga)**. El método ***file()*** permite **mostrar un fichero directamente en navegador, sin realizar la descarga**. Esto es útil para ficheros como imágenes o PDFs. Se debe pasar la **ruta al fichero a descargar**. Sigue el formato: *return response()**->file($pathToFile)**;* Por otro lado, el método ***download()*** genera una respuesta que **fuerza al navegador del cliente la descarga de un fichero**. Se debe pasar la **ruta al fichero a descargar**. Este método acepta un nombre de fichero como segundo parámetro, para renombrar el fichero a descargar. Sigue el formato: *return response()**->download($pathToFile)**;*

- **Response - Cookies**. Se pueden pasar **cookies** de Laravel con las instancias de tipo *Response*. Se debe indicar su **nombre**, un **valor** y los **minutos** de validez. Estas cookies están encriptadas; no pueden ser modificadas por el cliente.

  *return response('Hello World')**->cookie('name', 'value', $minutes)**;* También se pueden añadir argumentos propios de la función de PHP *setcookie()*.

- **Cookies de Laravel**. Para crear una cookie de Laravel, se puede usar la función de ayuda "helper" ***cookie()***. Se deben indicar los argumentos de nombre, valor y minutos de validez. Esta cookie **NO se envía de vuelta al cliente** automáticamente; se debe pasar vía instancia de Response con el método ***->cookie()*** visto previamente.

  *// **Crear cookie***
  *$cookie = **cookie('name', 'value', $minutes)**;*

  *// **Enviar cookie***
  *return **response**('Hello World')**->cookie($cookie)**;*

  Para **eliminar** una cookie de Laravel, se debe hacer lo mismo que se hace con cookies de PHP (Creadas con *setcookie()*): se debe **forzar su expiración** o validez. Con Laravel, este paso se puede realizar con la instancia de Respuesta del usuario, mediante el método ***withoutCookie()*** e indicando el nombre: *return response('Hello World')**->withoutCookie('name')**;* Alternativamente, se puede forzar la expiración directamente, mediante el método ***expire()*** de la clase propia Cookie, e indicando el nombre: ***Cookie::expire('name');***

- **RedirectResponse.** Las **respuestas de redirección** son instancias de la clase ***Illuminate\Http\RedirectResponse*** y contienen los **encabezados necesarios para redirigir al cliente a otra URL**. Existen varias formas de generar una instancia de *RedirectResponse*. El método más simple es utilizar el asistente global (helper) ***redirect()***.

  Un ejemplo de ruta redireccionada que podría declararse en web.php:

  Route::get('/dashboard', function () {
  *return **redirect('/home/dashboard')**;*
  });

  ➔ **Redirigir a rutas con nombre**. Cuando se llama al helper ***redirect()*** **sin parámetros**, se devuelve una instancia de *Illuminate\Routing\Redirector* (redirector de rutas), lo que permite invocar cualquier método en dicha instancia. Esto significa que, para generar una redirección, ***RedirectResponse,*** **a una ruta con nombre**, se puede utilizar el método ***route()***.

  Por ejemplo:

  *return redirect()**->route('login')**;*

  Si la ruta contiene parámetros, estos se pueden pasar como segundo argumento del método:

  *// Para una ruta con la siguiente URI: **/profile/{id} donde id = 1***

  *return redirect()->**route**('**profile**', **['id' => 1]**); // ruta: profile/1*

  ➔ **Redirigir a método de controlador**. También es posible redirigir a una acción o método de un controlador, para ello, se emplea ***redirect()*** **sin argumentos** y el método ***action()*** con llamada a la clase del controlador y su método.

  Ejemplo usando el controlador *UserController.php* en una redirección (en *web.php*, en un controlador, …):

  use App\Http\Controllers\UserController;

  *return redirect()**->action([UserController::class, 'index']);***

  Si el método requiere parámetros, se deben pasar como segundo parámetro de action():

  *return redirect()**->action**(*
  *[UserController::class, 'profile'], **['id' => 1]***

);

➔ **Redirigir con información adicional.** Redirigir a una nueva ruta y agregar datos para la nueva vista a cargar suele hacerse al mismo tiempo. Esto es común después de realizar una acción con éxito, como por ejemplo tras un store o update (guardado en BD) finalizado correctamente. Para realizar este paso de datos, es posible crear una instancia de *RedirectResponse* y agregar dicha información a la sesión de Laravel mediante el método encadenado ***with()***. Por ejemplo:

```
Route::post('/user/profile', function () {
    //  Acción a realizar...
    return redirect('/dashboard')->with('status', '¡Perfil actualizado!');
});
```

Una vez **pasados datos adicionales vía sesión dentro de una redirección**, es posible capturarlos haciendo referencia al **nombre del datos con session() utilizando sintaxis de Blade**. Por ejemplo, continuando con la vista a cargar tras guardar datos en BD se puede recoger el estado 'status' pasado mediante *with()* con el siguiente bucle:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}            {{-- Esto imprime "¡Perfil actualizado! --}}
    </div>
@endif
```

➔ **Redirigir fuera de la app web.** A veces es necesario redirigir a un dominio fuera de la aplicación. Esto se puede lograr utilizando el método ***away()***, que crea una *RedirectResponse* sin aplicar codificación de URL adicional, validación o verificación.

El siguiente ejemplo manda al famoso buscador google: ***return redirect()->away('https://www.google.com');***

Laravel: Modelo-Vista-Controlador (Crear aplicación básica MVC con datos persistentes en Laravel)

1. Crear proyecto de Laravel
2. Configurar la conexión con la base de datos (editar .env)
3. Crear migration con estructura de tablas (esquema) necesaria y generar tablas en base de datos.
4. Crear Modelo para cada tabla.
5. Crear Controlador de recursos para cada tabla.
6. Crear las rutas necesarias para poder acceder al servicio (al controlador).
7. Crear vistas para mostrar datos de las tablas y con formularios para trabajar con registros. Se requieren, mínimo, vistas para index, show y create/edit (formulario).

Con Artisan es posible crear un migration, modelo y controlador de recursos simultáneamente (pasos 3, 4 y 5). Por ejemplo, al crear el modelo se puede indicar que añada controlador y migration: **php artisan make:model Product --migration --controller --resource**

El anterior comando generará los siguientes ficheros:
- migration: database/migrations/xxxx_xx_xx_create_products_table.php (x es la fecha de creación)
- Modelo: app/Models/Product.php
- Controlador: app/Http/Controllers/ProductController.php

Los ficheros de migration, modelo y controlador de recursos creados con artisan NO están completos. Falta:

➢ **migration** - database/migrations/xxxx_xx_xx_create_products_table.php:
1. Completar las acciones del metodo up() introduciendo el esquema de la tabla a crear. Si Laravel ha interpretado correctamente que se trata de una nueva tabla, bastará con indicar las columnas de la tabla
2. Completar las acciones del metodo down(): debe ser opuesto a lo que realiza up(). En este caso, drop de la nueva tabla (posiblemente Laravel ya lo ha rellenado).
3. Crear la tabla en base de datos ejecutando el comando: php artisan migrate

➢ **Modelo**: app/Models/Product.php:
- Si tenemos un nombre de tabla que no sigue el convenio; añadir a la clase Modelo creada el atributo protected $table = 'Nombre_tabla'
- Si tenemos una primary key con nombre de columna diferente de 'id' (no sigue el convenio), añadir a la clase Modelo creada el atributo protected $primaryKey = 'Nuestro_id'
- Si se deben permitir crear o editar registros de forma masiva; (pe. pasando un array o emplear el método Product::create()), definir el atributo protected $fillable = ['columna1', columna2];

➢ **Controlador**: app/Http/Controllers/ProductController.php:
1. Asegurar que el controlador hace uso del Modelo. Laravel incluye este use si se crea en conjunto.
2. Completar los 7 métodos RESTful para conseguir CRUD con nuestra aplicación web.
3. Incluir el controlador en routes/web.php y crear las rutas (paso 6). Estas rutas se pueden definir automáticamente indicando Route::resource(). Por ejemplo:
   use App\Http\Controllers\ProductController;
   Route::resource('products', ProductController::class);

Los controladores de recursos están orientados a ofrecer servicios CRUD sobre datos persistentes. Por convenio, contiene 7 métodos (llamados RESTful) que deben realizar tareas específicas sobre registros:

- index() - Mostrar listado de registros
- show() - Mostrar detalles de un registro

- create() - Carga formulario para crear un registro
- store() - Recibe POST de formulario con datos de un nuevo registro y guarda en base de datos
- edit() - Carga formulario para modificar un registro
- update() - Recibe PUT de formulario con datos de un registro existente y actualiza en base de datos
- destroy() - Recibe DELETE y elimina un registro de base de datos

Nota: create() / store(), edit() / update() trabajan en conjunto.

★ **Ejemplo MVC - Gestión de productos (products)**

➔ **Pasos para generar MVC - migration**

❖ Vía comando, ejecutar: php artisan make: model Product --migration --controller --resource

Dentro del migration creado (X_create_products_table), completar el método up():

```
Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('nombre');
        $table->decimal('precio', 8, 2);
        $table->timestamps();
});
```

Nota: método down() ya está creado con drop de la tabla products.

❖ Vía comando, ejecutar: php artisan migrate

➔ **Pasos para generar MVC - modelo**. Para poder trabajar con inserciones o actualizaciones masivas en bases de datos (pasar inserts y updates de múltiples registros a Laravel vía arrays y métodos ::create() y ::update(), conviene añadir el atributo protected $fillable al modelo Producto.php:

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Product extends Model {
        protected $fillable = ['nombre','precio'];
}
```

➔ **Pasos para generar MVC - controlador**

```
<?php

namespace App\Http\Controllers;

use App\Models\Product;
use Illuminate\Http\Request;

class ProductController extends Controller {
  public function index() {
    $productos = Product::all();  // equivale a: select * from products
    return view('products.index', ['productos' => $productos]); // ALTERNATIVA: compact('productos')
  }

  public function show($id) {
    $producto = Product::findOrFail($id);  // Localiza en base de datos un registro por su id
    //dd($producto);   // para debuggear - se puede eliminar (recordatorio)
    return view('products.show', ['producto' => $producto]);
  }

  public function create() {
    return view('products.create');  // devuelve vista create.blade.php
  }

  public function store(Request $request) {
    $request->validate([  // validación del formulario en el servidor (revisa solicitud entrante)
      'nombre' => 'required',  // debe haber dato con clave nombre
      'precio' => 'required|numeric'  // debe haber dato con clave precio Y debe ser numérico
    ]);  // Localiza en base de datos un registro por su id
    Product::create($request->all());  // Update en base de datos, “all()” pasa los inputs en array asociativo
    // Redirección a products/index.blade.php con dato en session, clave “success”
    return redirect()->route('products.index')->with('success', '¡Producto creado!');
  }

  public function edit($id) {
    $producto = Product::findOrFail($id);
    return view('products.edit', ['producto' => $producto]);
  }

  public function update(Request $request, $id) {
```

```
        $request->validate([  // validación del formulario en el servidor (revisa solicitud entrante)
            'nombre' => 'required',  // debe haber dato con clave nombre
            'precio' => 'required|numeric'  // debe haber dato con clave precio Y debe ser numérico
        ]);  // Si validate falla, devuelve error (y recarga última vista)
        $producto = Product::findOrFail($id);  // Localiza en base de datos un registro por su id
        $producto->update($request->all());  // Update en base de datos, "all()" pasa los inputs en array asociativo
        return redirect()->route('products.index')->with('success', '¡Producto actualizado!');  // Redirección
    }

    public function destroy($id) {
        Product::findOrFail($id)->delete();  // Delete en base de datos del registro con id dada.
        return redirect()->route('products.index')->with('success', '¡Producto eliminado!');
    }
}
```

➔ **Pasos para generar MVC - Crear las Vistas (Blade Templates)**

- ➢ Crear la **carpeta en resources/views dónde guardar las vistas**. En este caso: resources/views/products/
- ➢ Dentro de la nueva carpeta, crear las **vistas**:
  - ☐ **index.blade.php** - Para mostrar la lista de productos
  - ☐ **show.blade.php** - Para ver detalles de un producto existente
  - ☐ **create.blade.php** - Para mostrar formulario de creación de producto
  - ☐ **edit.blade.php** - Para mostrar formulario de modificación de un producto existente

Nota: La distribución de los formularios de create.blade.php y edit.blade.php debería ser la misma.

➔ **Pasos para generar MVC - Crear las Vistas - index**

```
<h1>Lista de Productos</h1>
@if(session('success'))
        <p style="color: green;">{{ session('success') }}</p>
@endif

<a href="{{ route('products.create') }}">Agregar Producto</a>
@foreach($productos as $producto)
<p>
  {{ $producto->nombre }} - ${{ $producto->precio }}
        <a href="{{ route('products.show', $producto->id) }}">Ver</a>
        <a href="{{ route('products.edit', $producto->id) }}">Editar</a>
        <form action="{{ route('products.destroy', $producto->id) }}" method="POST" style="display:inline;">
        @csrf
        @method('DELETE')
        <button type="submit">Eliminar</button>
        </form>
</p>
@endforeach
```

➔ **Pasos para generar MVC - Crear las Vistas - show**

```
<h1>{{ $producto->nombre }}</h1>
<p>Precio: ${{ $producto->precio }}</p>
<a href="{{ route('products.edit', $producto->id) }}">Editar</a>
<form action="{{ route('products.destroy', $producto->id) }}" method="POST">
        @csrf
        @method('DELETE')
        <button type="submit">Eliminar</button>
</form>
<a href="{{ route('products.index') }}">Volver</a>
```

➔ **Pasos para generar MVC - Crear las Vistas - create**

```
<h1>Crear Producto</h1>
@if($errors->any())
        <ul>
        @foreach($errors->all() as $error)
        <li style="color: red;">{{ $error }}</li>
        @endforeach
        </ul>
@endif
<form action="{{ route('products.store') }}" method="POST">
        @csrf
        <label>Nombre:</label>
        <input type="text" name="nombre">
        <label>Precio:</label>
        <input type="text" name="precio">
        <button type="submit">Guardar</button>
</form>
<a href="{{ route('products.index') }}">Volver</a>
```

➔ **Pasos para generar MVC - Crear las Vistas - edit**

```
<h1>Editar Producto</h1>
@if($errors->any())
        <ul>
        @foreach($errors->all() as $error)
        <li style="color: red;">{{ $error }}</li>
        @endforeach
        </ul>
@endif
<form action="{{ route('products.update', $producto->id) }}" method="POST">
        @csrf
        @method('PUT')
        <label>Nombre:</label>
        <input type="text" name="nombre" value="{{ $producto->nombre }}">
        <label>Precio:</label>
        <input type="text" name="precio" value="{{ $producto->precio }}">
        <button type="submit">Actualizar</button>
</form>
<a href="{{ route('products.index') }}">Volver</a>
```

➔ **Pasos para generar MVC - Definir las rutas**

- En routes/web.php definir la siguiente ruta:
  use App\Http\Controllers\ProductController;
  Route::resource('products', ProductController::class);

Una vez completados todos los puntos anteriores, **ya está** creado un servicio CRUD que sigue el patrón de diseño Modelo-Vista-Controlador en Laravel.

➔ **Añadir Login al MVC - Breeze.** Para asegurar el **acceso al servicio creado mediante login** establecido por el paquete **Breeze**, basta con modificar la ruta al controlador de recursos **añadiendo un middleware de "auth"** sobre la agrupación de route::resource():

```
// Las 7 rutas del controlador de recursos están protegidas con middleware 'auth'
Route::middleware(['auth'])->group(function () {
        Route::resource('products', ProductController::class);
});
// Requiere rutas de autenticación Breeze, ya incluido en web.php
require __DIR__.'/auth.php';
```

➔ **Añadir Login al MVC - Breeze (alternativa).** Otra forma para **asegurar que los métodos CRUD quedan protegidos detrás del login de Breeze** es definir el middleware en el constructor del controlador de recursos (productoController.php):

```
public function __construct() {
   $this->middleware('auth');
}
```

➔ **Añadir Login al MVC - Breeze - Añadir enlaces.** Dado que ahora se requiere autenticación, conviene **añadir los enlaces a iniciar sesión** y registrarse al principio de la vista index.blade.php para que los clientes puedan realizar login:

```
@if(Auth::check())
        <p>Bienvenido, {{ Auth::user()->name }} | <a href="{{ route('logout') }}"
        onclick="event.preventDefault(); document.getElementById('logout-form').submit();">
        Cerrar sesión
        </a></p>
        <form id="logout-form" action="{{ route('logout') }}" method="POST" style="display: none;">
        @csrf
        </form>
@else
        <a href="{{ route('login') }}">Iniciar sesión</a>
        <a href="{{ route('register') }}">Registrarse</a>
@endif
```

También es buena idea añadir este código al resto de vistas.

Resultado: aplicación web MVC en Laravel