# Inyección de dependencias en Laravel




El concepto de *inyección de dependencias* es muy habitual en el uso de frameworks. Consiste en un mecanismo que facilita recursos a los diferentes componentes de la aplicación, y es algo que ya hemos utilizado, sin saberlo, en los métodos que se han generado para los controladores.

## 1. Inyectando la petición del usuario

Cuando definimos un método en un controlador que necesita procesar una petición, se le pasa como parámetro un objeto de tipo `Request`. Automáticamente, Laravel procesa el tipo de dato y obtiene el objeto asociado (en este caso, la petición del cliente).

```
class LibroController extends Controller
{
    ...
    public function store(Request $request)
    {
        ...
    }
}
```

## 2. Inyectando la respuesta del servidor

Al igual que tenemos un objeto `Request` para obtener datos de la petición, también existe un `Response` para gestionar la respuesta. Laravel proporciona un método `response` al que le podemos pasar varios parámetros:

1. El contenido de la respuesta
2. El código de estado HTTP de respuesta (si no se especifica, por defecto es 200)
3. Un array con las cabeceras de respuesta (por defecto está vacío).

Así, si por ejemplo queremos emitir una respuesta determinada con su código de estado desde un controlador, podemos hacer esto (por ejemplo, para un código 201):

```
response("Mensaje de respuesta", 201);
```

Las cabeceras pueden especificarse como un array, o enlazando llamadas al método `header` (una para cada cabecera):

```
response("Mensaje de respuesta", 201)
    ->header('Cabecera1', 'Valor1')
    ->header('Cabecera2', 'Valor2');
```

En el caso de querer devolver un objeto como respuesta, podemos emplear el método `json` de la respuesta (más adelante veremos que todos los objetos emitidos directamente al cliente se envían en formato JSON), y así podremos adjuntar un código de estado diferente de 200:

```
return response()->json(['datos' => datos], 201)
    ->header('Cabecera1', 'Valor1')
    ...;
```

## 2.1. Usar la respuesta para hacer redirecciones

Existe también un método `redirect` que podemos emplear para redireccionar a una ruta desde otra, bien especificando la ruta como parámetro...

```
redirect('/');
```

... o bien indicando una ruta con nombre:

```
redirect()->route('inicio');
```

Podemos pasar valores a la siguiente redirección, almacenándolos en sesión con el método `with`, aunque estos valores se perderán en la siguiente petición (no se quedan almacenados en sesión):

```
redirect()->route('inicio')
    ->with('mensaje', 'Mensaje enviado correctamente');
```

Para acceder a este mensaje desde la vista afectada, debemos utilizar la función `session`:

```
@if(session()->has('mensaje'))
    {{ session('mensaje') }}
@endif
```

Por último, notar que si hacemos la redirección desde dentro de un método de un controlador (por ejemplo, para redigir a una ruta desde otra), deberemos *devolver* ( `return` ) el resultado de esa redirección para que surta efecto:

```
class LibroController extends Controller
{
    public function index()
    {
        ...
    }

    public function store(...)
    {
        ...
        return redirect()->route('libros.index');
    }
```

# 3. Los *helpers*

Para terminar esta introducción a lo que supone la inyección de dependencias en frameworks de desarrollo, vamos a hacer uso de una herramienta que nos puede ser útil en algunas situaciones: los *helpers*.

Un **helper** es básicamente una función de utilidad que podemos querer utilizar en diversos puntos de nuestra web, y que necesitamos tener localizada y compartida. Por ejemplo, imaginemos que queremos resaltar en nuestro menú de navegación la opción que tenemos actualmente visible.

Para ello, podemos definir una clase CSS con el estilo que queramos para resaltar (esto lo haremos aparte, en los archivos CSS del proyecto), y después utilizar esa clase CSS en una condición para cada menú de navegación.

Por ejemplo, supongamos que la clase CSS para identificar el menú activo se llama `activo` . En este caso, para un menú de varias opciones como éste, basta con utilizar el método `routeIs` de la petición ( `request` ) para comprobar si la ruta coincide con cada menú, y mostrarlo como activo o no, usando un operador ternario de comparación:

```
<nav>
    <ul>
        <li class="{{ request()->routeIs('inicio') ? 'activo' : '' }}">
            <a href="/">Inicio</a>
        </li>
        <li class="{{ request()->routeIs('contacto') ? 'activo' : '' }}">
            <a href="/contacto">Contacto</a>
        </li>
        ...
    </ul>
</nav>
```

Esta característica también funciona si las rutas tienen parámetros.

Podemos, en cambio, sacar fuera de la vista la lógica de establecer un campo como activo o no. Para ello, creamos un archivo de utilidad o *helper*. Lo podemos llamar `helpers.php`, y ubicarlo en la misma carpeta `app`. Dentro, definimos la función que nos va a devolver si una ruta está activa o no, a partir de su nombre:

```
<?php

function setActivo($nombreRuta)
{
    return request()->routeIs($nombreRuta) ? 'activo' : '';
}
```

Y de este modo, nuestra vista simplemente se dedica a llamar a esta función para cada elemento del menú:

```
<nav>
    <ul>
        <li class="{{ setActivo('inicio') }}">
            <a href="/">Inicio</a>
        </li>
        <li class="{{ setActivo('contacto') }}">
            <a href="/contacto">Contacto</a>
        </li>
        ...
    </ul>
</nav>
```

En el caso de querer mantener el enlace activo para cualquier subruta a partir de la original (por ejemplo, cuando estamos viendo la ficha de un registro a partir del listado general, podemos utilizar el *wildcard* de asterisco `*`):

```
<li class="{{ setActivo('peliculas.*') }}">
    <a href="{{ route('peliculas') }}">Peliculas</a>
</li>
```

Sin embargo, para que Laravel cargue el archivo `helpers.php` que acabamos de crear, como no es una clase, debemos indicarlo explícitamente (Laravel carga automáticamente todas las clases de la carpeta `app`, pero no archivos sueltos que no sean clases). Para ello, debemos ir al archivo `composer.json` de la raíz de nuestro proyecto, a la sección `autoload` y añadir una sección `files` con un array con los archivos que queramos que se carguen también:

```
"autoload": {
    "classmap": [ ... ],
    "psr-4": { ... },
    "files": ["app/helpers.php"]
},
```

Tras efectuar el cambio, debemos decirle a *composer* que vuelva a compilar el auto cargador. Desde la carpeta del proyecto, ejecutamos este comando:

```
composer dump-autoload
```