Unidad 13 - Controladores

Introducción

Un controlador es una función PHP creada por nosotros, que toma información desde la petición HTTP (Request), construye una respuesta HTTP (Response) y la devuelve. La respuesta podría ser una página HTML, un documento XML, un arreglo JSON serializado, una imagen, una redirección, un error 404 o cualquier otra cosa que se te ocurra. El controlador contiene toda la lógica arbitraria que tu aplicación necesita para reproducir el contenido de la página.

El objetivo de un controlador siempre es el mismo: crear y devolver un objeto Response. Al mismo tiempo, este puede leer la información de la petición, cargar un recurso de base de datos, enviar un correo electrónico, o fijar información en la sesión del usuario. Pero en todos los casos, el controlador eventualmente devuelve el objeto Response que será entregado al cliente.

```
// src/Controller/LuckyController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class LuckyController
{
	#[Route('/lucky/number/{max}', name: 'app_lucky_number')]
	public function number(int $max): Response
	{
		$number = random_int(0, $max);

		return new Response(
			'<html><body>Lucky number: '.$number.'</body></html>'
		);
	}
}
```

Ruteo

Tener URLs legibles y amigables son imprescindibles para cualquier aplicación web seria. Esto significa dejar atrás URLs feas como index.php?article_id=57 a favor de algo como /read/curso-symfony.

Tener flexibilidad es aún más importante. ¿Qué pasa si necesitas cambiar la URL de una página? ¿Cuántos enlaces tendría que buscar y actualizar para hacer el cambio? El sistema de ruteo de Symfony permite que este cambio sea simple.

El sistema de enrutamiento de Symfony transforma URL en controladores. Más en concreto, determina cuál es el controlador que se debe ejecutar para cada URL solicitada por los usuarios.

Por ejemplo:

```php
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
    /**
     * Matches /blog exactly
     */

    #[Route('/blog', name: 'blog_list')]
    public function list()
    {
        // ...
    }

    /**
     * Matches /blog/*
     */

    #[Route('/blog/{slug}', name: 'blog_show')]
    public function show($slug)
    {
        // ...
    }
}
```

Si el usuario va a /blog, la primera ruta coincide y se ejecuta list();

Si el usuario se dirige a /blog/*, la segunda ruta coincide y se ejecuta show(). Como la ruta es /blog/{slug}, se pasa una variable $slug al método show() del controlador que coincide con el valor indicado en la ruta. Por ejemplo, si el usuario va a /blog/yay-routing, entonces $slug tendrá el valor yay-routing.

Cada ruta también tiene un nombre interno: blog_list y blog_show. Estos pueden ser cualquiera (siempre y cuando cada uno sea único en todo el sistema) y se utilizan para poder nombrar a la ruta de una forma más fácil.

Coincidencia de métodos HTTP

De forma predeterminada, las rutas coinciden con cualquier verbo HTTP (GET, POST, PUT, etc.). Se puede utilizar la opción de métodos para restringir los verbos a los que debe responder cada ruta:

```
// src/Controller/BlogApiController.php
namespace App\Controller;

// ...

class BlogApiController extends AbstractController
{
      #[Route('/api/posts/{id}', methods: ['GET', 'HEAD'])]
      public function show(int $id): Response
      {
            // ... return a JSON response with the post
      }

      #[Route('/api/posts/{id}', methods: ['PUT'])]
      public function edit(int $id): Response
      {
            // ... edit a post
      }
}
```

Validación de parámetros

Imagina que la ruta blog_list contendrá una lista paginada de entradas de blog, con URLs como /blog/2 y /blog/3 para las páginas 2 y 3. Si cambia la ruta de la ruta a /blog/{page}, tendrá un problema:

blog_list: /blog/{page} coincidirá con /blog/*;
blog_show: /blog/{slug} también coincidirá con /blog/*.

Cuando dos rutas coinciden con la misma URL, la primera ruta que se carga gana.

Desafortunadamente, eso significa que /blog/yay-routing coincidirá con el blog_list. No es bueno!

Para solucionar este problema, añada el requisito de que el comodín {página} sólo pueda coincidir con números (dígitos):

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
      #[Route('/blog/{page}', name: 'blog_list', requirements: ['page' => '\d+'])]
      public function list(int $page): Response
      {
            // ...
      }

      #[Route('/blog/{slug}', name: 'blog_show')]
      public function show($slug): Response
      {
            // ...
      }
}
```

El \d+ es una expresión regular que coincide con un dígito de cualquier longitud.

Parámetros opcionales

En el ejemplo anterior, blog_list tiene una ruta de /blog/{page}. Si el usuario visita /blog/1, coincidirá. Pero si visitan /blog, no coincidirá. Entonces, ¿cómo puedes hacer que blog_list vuelva a coincidir cuando el usuario visita /blog? Añadiendo un valor por defecto en el parámetro del método del controlador:

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
      #[Route('/blog/{page}', name: 'blog_list', requirements: ['page' => '\d+'])]
      public function list(int $page = 1): Response
      {
            // ...
      }
}
```

Ahora, cuando el usuario visita /blog, la ruta blog_list coincidirá y la página $ tendrá un valor por defecto de 1.

Parámetro de prioridad

Symfony evalúa las rutas en el orden en que son definidas. Si el path de la ruta coincide con diferentes patrones, se ejecuta la primera ruta que se encuentra y las otras no se ejecutan.

Si necesitamos cambiar el orden de una ruta, debemos mover todo el código del método, y eso no es muy cómodo, además que puede producir errores.

Para solucionar este inconveniente se puede usar el parámetro opcional “priority” para definir las rutas que deben evaluarse antes que las otras.

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController
{
	/**
	* This route has a greedy pattern and is defined first.
	*/
	#[Route('/blog/{slug}', name: 'blog_show')]
	public function show(string $slug): Response
	{
		// ...
	}

	/**
	* This route could not be matched without defining a higher priority than 0.
	*/
	#[Route('/blog/list', name: 'blog_list', priority: 2)]
	public function list(): Response
	{
		// ...
	}
}
```

El parámetro priority espera un valor entero. Las rutas con la prioridad más alta, se evalúan primero. El valor por defecto de la prioridad es 0.

Generación de URLs

El sistema de enrutamiento también puede generar URLs. En realidad, el enrutamiento es un sistema bidireccional: asigna la URL a un controlador y también una ruta de regreso a una URL.

Para generar una URL, debe especificar el nombre de la ruta (por ejemplo, sign_up). Con esta información, cualquier URL puede ser generada fácilmente:

```
// src/Controller/BlogController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

class BlogController extends AbstractController
{
      #[Route('/blog', name: 'blog_list')]
      public function list(): Response
      {
            // generate a URL with no route arguments
            $signUpPage = $this->generateUrl('sign_up');

            // generate a URL with route arguments
            $userProfilePage = $this->generateUrl('user_profile', [
            'username' => $user->getUserIdentifier(),
            ]);

            // ...
      }
}
```

Generar URL en una plantilla

Para generar un link en una página HTML se puede utilizar la función path() de Twig. A la función path se le pasa como primer parámetro el nombre de la ruta y como segundo parámetro, un objeto que representa los parámetros de la ruta.

```
<a href="{{ path('blog_index') }}">Homepage</a>

{# ... #}

{% for post in blog_posts %}
      <h1>
      <a href="{{ path('blog_post', {slug: post.slug}) }}">{{ post.title }}</a>
      </h1>

      <p>{{ post.excerpt }}</p>
{% endfor %}
```

Controlador base

Para ayudarnos en el desarrollo Symfony proporciona una clase Controladora base que nos provee de una serie de helpers (ayudantes), para hacer más fácil la programación. Esta clase base se llama AbstractController y si hacemos que nuestro controlador derive de esta clase, vamos a tener a

nuestra disposición una serie de métodos que nos va a simplificar el trabajo. Por ejemplo, los métodos render y generateUrl vistos anteriormente.

```
// src/Controller/LuckyController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class LuckyController extends AbstractController
{
      // ...
}
```

Otras funciones muy útiles que nos provee esta clase son las funciones de redireccionamiento. Con estas funciones podemos hacer que la aplicación salte hacia otra página.

```
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Response;

// ...
public function index(): RedirectResponse
{
    // redirects to the "homepage" route
    return $this->redirectToRoute('homepage');

    // redirect to a route with parameters
    return $this->redirectToRoute('app_lucky_number', ['max' => 10]);

    // redirects externally
    return $this->redirect('http://symfony.com/doc');
}
```

Accediendo a servicios

Symfony viene repleto de muchas clases y funcionalidades útiles, llamadas servicios. Estos se utilizan para enviar correos electrónicos, consultar la base de datos y cualquier otro "trabajo" que necesitemos hacer.

Si se necesita un servicio en un controlador, se puede agregar un argumento con su nombre de clase (o interfaz). Symfony te pasará automáticamente el servicio que se necesita:

```
use Psr\Log\LoggerInterface;
use Symfony\Component\HttpFoundation\Response;
// ...

#[Route('/lucky/number/{max}')]
public function number(int $max, LoggerInterface $logger): Response
{
      $logger->info('We are logging!');
      // ...
}
```

Objeto Request

Symfony encapsula toda la información de la petición del usuario en un objeto de tipo Request. Así se facilita y centraliza el acceso a toda la información de las variables globales de PHP como $_GET, $_REQUEST, $_SERVER, $_FILES y $_COOKIE. Dentro de un controlador, podemos obtener el objeto de la petición como argumento del controlador:

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

public function index(Request $request): Response
{
    $request->isXmlHttpRequest(); // is it an Ajax request?

    $request->getPreferredLanguage(['en', 'fr']);

    // retrieves GET variables
    $request->query->get('page');

    // retrieves POST variables
    $request->request->get('page');

    // retrieves SERVER variables
    $request->server->get('HTTP_HOST');

    // retrieves an instance of UploadedFile identified by foo
    $request->files->get('foo');

    // retrieves a COOKIE value
    $request->cookies->get('PHPSESSID');

    // retrieves an HTTP request header, with normalized, lowercase keys
    $request->headers->get('host');
    $request->headers->get('content-type');
}
```

Para más información consultar:
https://symfony.com/doc/5.4/components/http_foundation.html#request

Objeto Response

El único requisito para un controlador es que devuelva un objeto Response. Un objeto Response contiene toda la información que debe enviarse al cliente a partir de una solicitud determinada. El constructor toma hasta tres argumentos: el contenido de la respuesta, el código de estado y una matriz de encabezados HTTP.

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response(
    'Hello World',
    Response::HTTP_OK,
    ['content-type' => 'text/html']
);
```

Esta información puede ser manipulada luego de crearse el objeto Response:

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response();

$response->setContent('Hello World');

$response->setStatusCode(Response::HTTP_NOT_FOUND);

// the headers public attribute is a ResponseHeaderBag
$response->headers->set('Content-Type', 'text/plain');
```

Podemos usar el método file() del controlador para devolver un archivo, en vez de devolver HTML.

```
use Symfony\Component\HttpFoundation\Response;
// ...

public function download(): Response
{
    // send the file contents and force the browser to download it
    return $this->file('/path/to/some_file.pdf');
}
```

Para más información consultar:
https://symfony.com/doc/5.4/components/http_foundation.html#response

Mensajes Flash

Se pueden almacenar mensajes especiales, llamados mensajes “flash”. Por diseño, los mensajes flash están destinados a usarse exactamente una vez: desaparecen de la sesión automáticamente tan pronto como los recupera. Esta característica hace que los mensajes "flash" sean particularmente buenos para almacenar notificaciones de usuarios.

Por ejemplo, imagine que está procesando un formulario enviado:

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
// ...

public function update(Request $request): Response
{
    // ...

    if ($this->guardarForm()) {
        $this->addFlash(
            'notice',
            'Los cambios se guardaron correctamente'
        );
        $this->addFlash(
            'notice',
            'Por favor vuelva a ingresar al sistema'
        );
        return $this->redirectToRoute('home');
    }

    return $this->render(...);
}
```

Después de procesar la solicitud, el controlador establece un mensaje flash en la sesión y luego redirige. La clave del mensaje (notice en este ejemplo) puede ser cualquier cosa. Se usará esta clave para recuperar el mensaje.

En la plantilla de la página siguiente, se leen los mensajes flash usando el método flashes() provisto por Twig:

```
{% for message in app.flashes('notice') %}
    <div class="flash-notice">
        {{ message }}
    </div>
{% endfor %}
```