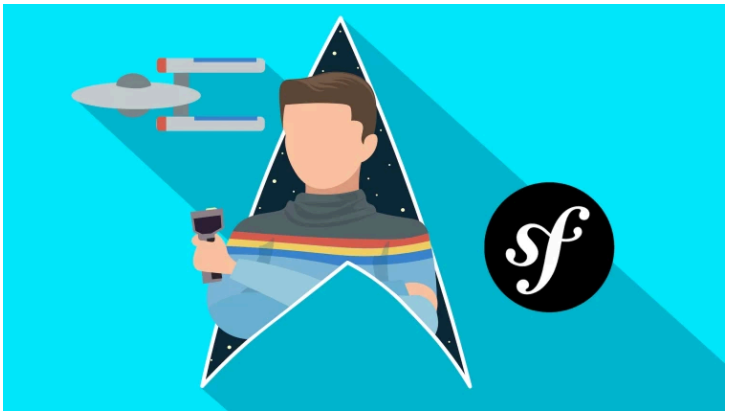
Codificación cósmica con Symfony 7



Chapter 1: Configurando nuestra App Symfony

¡Bienvenido al primer tutorial de Symfony 7! Me llamo Ryan - vivo aquí en el mundo de fantasía de Symfonycasts y... Estoy más que emocionado de ser tu guía a través de esta serie sobre Symfony, desarrollo web... chistes malos... animaciones espaciales, y lo más importante, construir cosas reales de las que podamos estar orgullosos. Para mí, es como si fuera la persona afortunada que consigue darte un tour personal por el Enterprise... o por cualquier cosa friki que te emocione más.

Y eso es porque me encantan estas cosas. Crear bases de datos, construir bonitas interfaces de usuario, escribir código de alta calidad... es lo que me levanta de la cama por las mañanas. Y Symfony es la mejor herramienta para hacer todo esto... y convertirme en un mejor desarrollador por el camino.

Y ese es realmente mi objetivo: quiero que disfrutes de todo esto tanto como yo... y que te sientas capacitado para construir todas las cosas increíbles que tienes flotando en tu mente.

Lo que hace especial a Symfony

Ahora, una de mis cosas favoritas sobre la enseñanza de Symfony es que nuestro proyecto va a empezar diminuto. Eso hace que sea fácil de aprender. Pero luego, escalará automáticamente a medida que necesitemos más herramientas mediante un sistema de recetas único. Symfony es en realidad una colección de más de 200 pequeñas librerías PHP. Así que son un montón de herramientas... pero podemos elegir lo que necesitamos.

Porque, puedes estar construyendo una API pura... o una aplicación web completa, que es en lo que nos centraremos en este tutorial. Aunque, si estás construyendo una API, sigue los primeros tutoriales de esta serie, y luego pasa a nuestros tutoriales sobre la API Platform. API Platform es un sistema alucinantemente divertido y potente para crear APIs, construido sobre Symfony.

Symfony también es rapidísimo, tiene versiones de soporte a largo plazo y se esfuerza mucho en crear una experiencia agradable para el desarrollador, al tiempo que mantiene las mejores

prácticas de programación. Esto significa que podemos escribir código de alta calidad y hacer nuestro trabajo rápidamente.

Vale, ya está bien de hablar maravillas de Symfony. ¿Listo para empezar a trabajar? Pues sube a bordo.

Instalar el binario de Symfony

Dirígete a https://symfony.com/download. Esta página tiene instrucciones sobre cómo descargar un binario independiente llamado `symfony`. Ahora bien, esto no es Symfony propiamente dicho... es sólo una pequeña herramienta que nos ayudará a hacer cosas, como iniciar nuevos proyectos Symfony, ejecutar un servidor web local o incluso desplegar nuestra aplicación en producción.

Una vez que lo hayas descargado e instalado, abre un terminal y entra en cualquier directorio. Comprueba que el binario `symfony` está listo para funcionar ejecutándolo:

```
symfony --help
```

Tiene un montón de comandos, pero sólo necesitaremos unos pocos. Antes de iniciar un proyecto, ejecuta también

```
symfony check:req
```

que significa comprobar requisitos. Esto asegura que tenemos todo lo necesario en nuestro sistema para ejecutar Symfony, como PHP en la versión correcta y algunas extensiones PHP.

Una vez que esto esté contento, ¡podemos empezar un nuevo proyecto! Hazlo con `symfony new` y luego un nombre de directorio. Yo llamaré al mío `starshop`. Más adelante hablaremos de ello.

```
symfony new starshop
```

Esto nos dará un proyecto pequeñito con sólo las cosas base instaladas. Luego, iremos añadiendo más cosas poco a poco por el camino. Pero más adelante, cuando te sientas cómodo con Symfony, si quieres empezar más rápidamente, puedes ejecutar el mismo comando, pero con `--webapp` para obtener un proyecto con muchas más cosas preinstaladas.

De todos modos, entra en el directorio - `cd starshop` - y luego escribiré `ls` para comprobar las cosas. ¡Genial! Conoceremos estos archivos en el próximo capítulo, pero este es nuestro proyecto... ¡y ya está funcionando!

Iniciando el Servidor Web symfony

Para verlo funcionando en un navegador, necesitamos iniciar un servidor web. Puedes utilizar el servidor web que quieras: Apache, Nginx, Caddy, lo que sea. Pero para el desarrollo local, recomiendo encarecidamente utilizar el binario `symfony` que acabamos de instalar. Ejecuta:

```
symfony serve
```

La primera vez que lo hagas, puede que te pida que ejecutes otro comando para configurar un certificado SSL, lo cual está bien porque así el servidor admite https.

Y... ¡bam! Tenemos un nuevo servidor web para nuestro proyecto ejecutándose en https://127.0.0.1:8000. Copia eso, gira a tu navegador más favorito, pega y... ¡bienvenido a Symfony 7! ¡Eso es lo que iba a decir!

A continuación, sentémonos, pidamos un té Earl Grey y hagámonos amigos de todos los archivos de nuestra nueva aplicación... que no son muchos.

Chapter 2: Conociendo nuestro pequeño proyecto

Vuelve a tu centro de comandos (también conocido como terminal). Esta primera pestaña está ejecutando el servidor web. Si necesitas detenerlo, pulsa Ctrl-C... y reinícialo con:

```
symfony serve
```

> **Tip**
>
> Puedes utilizar `symfony serve -d` para ejecutar el comando en "segundo plano" y poder seguir utilizando esta pestaña del terminal.

Lo dejaremos así y dejaremos que haga lo suyo.

Los 15 archivos de nuestro proyecto

Abre una segunda pestaña de terminal en el mismo directorio. Cuando ejecutamos el comando `symfony new`, descargó un pequeño proyecto e inicializó un repositorio Git con una confirmación inicial. ¡Eso estuvo muy bien! Para ver nuestros archivos, voy a abrir este directorio en mi editor favorito: PhpStorm. Más sobre este editor en unos minutos.

Ahora quiero que te des cuenta de lo pequeño que es nuestro proyecto Para ver la lista completa de archivos confirmados, vuelve a tu terminal y ejecuta:

```
git ls-files
```

Sí, eso es. Sólo hay unos 15 archivos confirmados en git

¿Dónde está Symfony?

Entonces... ¿dónde demonios está Symfony? Uno de nuestros 15 archivos es especialmente importante: `composer.json`.

composer.json

```
{
// ... lines 2 - 5
    "require": {
        "php": ">=8.2",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "symfony/console": "7.0.*",
        "symfony/dotenv": "7.0.*",
        "symfony/flex": "^2",
        "symfony/framework-bundle": "7.0.*",
        "symfony/runtime": "7.0.*",
        "symfony/yaml": "7.0.*"
    },
// ... lines 17 - 70
}
```

Composer es el gestor de paquetes de PHP. Su trabajo es sencillo: leer los nombres de los paquetes bajo esta clave `require` y descargarlos. Cuando ejecutamos el comando `symfony new`, descargó estos 15 archivos y también ejecutó `composer install`. Eso descargó todos estos paquetes en el directorio `vendor/`.

¿Dónde está Symfony? Está en `vendor/symfony/`... ¡y ya estamos utilizando unos 20 de sus paquetes!

Ejecuta Composer

El directorio `vendor/` no está registrado en git. Se ignora gracias a otro archivo con el que empezamos: `.gitignore`.

.gitignore

```
###> symfony/framework-bundle ###
/.env.local
/.env.local.php
/.env.*.local
/config/secrets/prod/prod.decrypt.private.php
/public/bundles/
/var/
/vendor/
###
```

Esto significa que si un compañero de equipo clona nuestro proyecto, no tendrá este directorio. ¡Y no pasa nada! Siempre podemos repoblarlo ejecutando `composer install`.

Observa: Haré clic con el botón derecho y borraré todo el directorio `vendor/`. Y ¡huy!

Si probamos ahora nuestra aplicación, se estropeará. ¡Mal rollo! Para arreglarlo y salvar el día, en tu terminal, ejecuta:

```
composer install
```

Y... ¡listo! El directorio vuelve a .... y por aquí, el sitio vuelve a funcionar.

Los 2 directorios que te importan

Si volvemos a mirar nuestros archivos, sólo hay dos directorios en los que tengamos que pensar. El primero es `config/`: contiene... ¡configuración! Aprenderemos lo que hacen estos archivos por el camino.

El segundo es `src/`. Aquí es donde vivirá todo tu código PHP.

¡Y eso es todo! el 99% del tiempo estás configurando algo o escribiendo código PHP. Eso ocurre en `config/` y `src/`.

¿Qué pasa con los otros 4 directorios? `bin/` contiene un único archivo ejecutable `console` que probaremos pronto. Pero nunca vamos a mirar o modificar ese archivo. El directorio

`public/` se conoce como la raíz de tu documento. Cualquier cosa que pongas aquí -como una imagen- será accesible públicamente. También contiene `index.php`.

public/index.php

```php
<?php

use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

Esto se conoce como tu "controlador frontal": es el archivo PHP principal que tu servidor web ejecuta al inicio de cada petición. Y aunque es superimportante... nunca editarás ni pensarás en este archivo.

El siguiente es `var/`. Esto también se ignora desde git: es donde Symfony almacena los archivos de registro y los archivos de caché que necesita internamente. Así que muy importante... pero no algo en lo que tengamos que pensar. Y ya hemos hablado de `vendor/`. ¡Eso es todo!

Preparando PhpStorm

Ahora, antes de ponernos a codificar, he mencionado que yo utilizo PhpStorm. Eres libre de utilizar el editor que quieras. Sin embargo, PhpStorm es increíble. Y una gran razón es el incomparable plugin Symfony. Si vas a PhpStorm -> Configuración y buscas "Symfony", aquí abajo bajo Plugins y luego Marketplace, podrás encontrarlo. Descarga e instala el plugin si aún no lo tienes. Después de la instalación, reinicia PhpStorm. Luego hay un paso más. Vuelve a la configuración y busca Symfony de nuevo. Esta vez tendrás una sección Symfony. Asegúrate de activar el plugin para cada proyecto Symfony en el que trabajes... de lo contrario no verás la misma magia que yo.

¡De acuerdo! Empecemos a codificar y construyamos nuestra primera página en Symfony a continuación.

Chapter 3: Rutas, controladores y respuestas

Bien, ésta es la primicia. Wesley Crusher -el alférez favorito de todos en Star Trek- se ha retirado de la Flota Estelar y colabora con nosotros para poner en marcha un nuevo negocio: La Tienda Estelar de Wesley. Alguien tiene que romper el monopolio ferengi en el negocio de reparación de naves estelares de la galaxia, y nos ha contratado para construir el sitio que lo haga. ¡Estamos a punto de darles a los ferengis una carrera por su latinio!

Creación del controlador

Y todo empieza con la creación de nuestra primera página. La idea detrás de cada página es siempre la misma. Primer paso: dale una URL chula. Eso se llama la ruta. Paso dos, escribir una función PHP que genere la página. Eso se conoce como el controlador. Y esa página puede ser HTML, JSON, arte ASCII, cualquier cosa.

En Symfony, el controlador es siempre un método dentro de una clase PHP. Así que, ¡necesitamos crear nuestro primer código PHP! ¿Dónde vive el código PHP en nuestra aplicación? Exacto, en el directorio `src/`.

Dentro de este directorio `src/Controller/`, crea un nuevo archivo. Normalmente seleccionaría nueva "clase PHP", pero para esta primera vez, crea un archivo vacío. Haremos cada parte a mano. Llámalo `MainController.php`, pero puedes ponerle el nombre que quieras.

Dentro, añade la etiqueta open PHP, y luego di `class MainController`. Encima de esto, añade un espacio de nombres de `App\Controller`.

src/Controller/MainController.php

```
// ... lines 1 - 2
namespace App\Controller;

class MainController
{
}
```

Espacios de nombres y directorios

Bien, algunas cosas sobre esto. Primero, el hecho de que ponga esta clase dentro de un directorio llamado `Controller` es opcional. Es sólo una convención. Podrías cambiarle el nombre por cualquiera que sea la palabra klingon para `Controller` y todo seguiría igual... ¡y probablemente sería más interesante!

Sin embargo, hay algunas reglas sobre las clases PHP en general. La primera es que cada clase debe tener un espacio de nombres y ese espacio de nombres tiene que coincidir con tu estructura de directorios. Siempre será `App\` y luego el directorio en el que estés. Sin entrar en demasiados detalles, es una regla que encontrarás en todos los proyectos PHP.

La segunda regla es que el nombre de tu clase debe coincidir con el nombre de tu archivo `.php`. Si te equivocas en cualquiera de estas dos cosas, recibirás un error de PHP diciendo que no puede encontrar tu clase. Los Ferengi nunca cometen este error.

Crear el método controlador y la ruta

De todos modos, nuestro objetivo es crear un controlador, que es un método en una clase que construye la página. Añade una nueva función pública y llámala `homepage`. Pero, de nuevo, el nombre no importa. Y... ¡sí! Aún no está hecho, ¡pero éste es nuestro controlador!

src/Controller/MainController.php

```
↕ // ... lines 1 - 4
5  class MainController
6  {
7      public function homepage()
8      {
9
10     }
11 }
```

Pero recuerda, una página es la combinación de un controlador y una ruta, que define la URL de la página. ¿Dónde ponemos la ruta? Justo encima del método controlador, utilizando una función de PHP llamada atributo. Escribe `#[]` y luego empieza a escribir `Route` con mayúscula `R`. ¡Fíjate en el autocompletado!

Cualquiera de las opciones funcionará, pero utiliza la de `Attribute` -que es más nueva- y luego pulsa tabulador. Cuando hice eso, ocurrió algo superimportante: mi editor añadió una

declaración `use` al principio de la clase. Siempre que utilices un atributo PHP, debes tener una declaración `use` correspondiente para él en el mismo archivo.

Estos atributos funcionan casi como las funciones PHP: puedes pasar un montón de argumentos. El primero es la ruta. Establécela en `/`.

src/Controller/MainController.php

```
// ... lines 1 - 4
use Symfony\Component\Routing\Attribute\Route;

class MainController
{
    #[Route('/')]
    public function homepage()
    {

    }
}
```

Gracias a esto, cuando alguien vaya a la página de inicio - `/` - ¡Symfony llamará a este método controlador para construir la página!

Controladores y respuestas

¿Qué... debería devolver nuestro método? Sólo el HTML que queremos, ¿verdad? ¿O el JSON si estamos construyendo una API?

Casi. La web funciona con un sistema bien conocido. Primero, un usuario solicita una página. Dice:

> *"Oye, quiero ver `/products`... o quiero ver `/users.json`."*

Lo que les devolvemos, sí, contiene HTML o JSON. Pero es más que eso. También les comunicamos un código de estado -que dice si la respuesta era correcta o tenía un error-, así como estas cosas llamadas cabeceras, que comunican un poco más de información, como el formato de lo que estamos devolviendo.

Todo este hermoso paquete se llama respuesta. Así que sí, la mayoría de las veces, sólo pensaremos en devolver HTML o JSON. Pero lo que realmente estamos enviando es esta cosa más grande y friki llamada respuesta.

Así que todo nuestro trabajo como desarrolladores web -independientemente del lenguaje en el que programemos- consiste en comprender la petición del usuario y, a continuación, crear y devolver la respuesta.

Y esto nos lleva de nuevo a algo que me encanta de Symfony. ¿Qué devuelve nuestro controlador? ¡Un nuevo objeto `Response` de Symfony! Y de nuevo, PhpStorm quiere autocompletar esto, sugiriendo unas cuantas clases diferentes de `Response`. Nosotros queremos la del componente `HttpFoundation` de Symfony. Esa es la librería de Symfony que contiene todo lo relacionado con peticiones y respuestas.

Pulsa tabulador. Una vez más, cuando hicimos eso, PhpStorm añadió una declaración `use` en la parte superior del archivo. Voy a utilizar este truco constantemente. Cada vez que hagas referencia a un nombre de clase, debes tener una declaración `use` correspondiente, de lo contrario PHP te dará un error diciendo que no puede encontrar la clase `Response`.

Dentro de esto, el primer argumento es el contenido que queremos devolver. Empieza con una cadena codificada.

src/Controller/MainController.php

```
// ... lines 1 - 4
use Symfony\Component\HttpFoundation\Response;
// ... lines 6 - 7
class MainController
{
    #[Route('/')]
    public function homepage()
    {
        return new Response('<strong>Starshop</strong>: your monopoly-busting option for Starship parts!');
    }
}
```

Ruta, ¡comprobado! Controlador que devuelve una Respuesta, ¡comprobado! Probemos esto. En el navegador, esta página era sólo una demo que muestra antes de que tengamos una página de inicio real. Ahora que la tenemos, al actualizar... ¡ahí está!

Sé que aún no es mucho, pero acabamos de aprender la primera parte fundamental de Symfony: que cada página es una ruta y un controlador... y que cada controlador devuelve una respuesta.

Ah, y es opcional, pero como nuestro controlador siempre devuelve un `Response`, podemos añadir un tipo de retorno `Response`. Eso no cambia el funcionamiento de nuestro código, pero lo hace más descriptivo de leer. Y si alguna vez hiciéramos una tontería y devolviéramos algo que no fuera una respuesta, PHP nos lo recordaría claramente.

src/Controller/MainController.php

```
↕ // ... lines 1 - 7
8  class MainController
9  {
10     #[Route('/')]
11     public function homepage(): Response
12     {
13         return new Response('<strong>Starshop</strong>: your monopoly-
   busting option for Starship parts!');
14     }
15 }
```

A continuación: para potenciar nuestro desarrollo, instalemos nuestro primer paquete de terceros y conozcamos el increíble sistema de recetas de Symfony.

Chapter 4: Recetas Flex Mágicas

Tengo un secreto. Cuando se creó nuestro proyecto, no eran 15 archivos. Era... un solo archivo. Si miraras dentro del código del comando `symfony new`, descubrirías que es un atajo para sólo dos cosas. Primero, clona un repositorio llamado `symfony/skeleton`... que es sólo un archivo si ignoras la licencia. Y en segundo lugar, ejecuta `composer install`.

Y ya está Pero espera, si ese es el caso, ¿de dónde han salido todos esos otros archivos? ¿Como las cosas de `bin/`, `config/` y `src/`? La respuesta empieza con un paquete especial dentro de nuestro archivo `composer.json` llamado `symfony/flex`. Flex es un complemento de Composer que añade dos superpoderes a Composer: alias y recetas.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 11
        "symfony/flex": "^2",
// ... lines 13 - 15
    },
// ... lines 17 - 70
}
```

Alias Flex

Los alias son sencillos. Para añadir un nuevo paquete a tu aplicación -lo que haremos en un minuto- ejecutas `composer require` y luego el nombre del paquete, como `symfony/http-client`. Flex da a los paquetes más importantes del ecosistema Symfony un nombre más corto, llamado alias. Por ejemplo, `symfony/http-client` tiene un alias llamado `http-client`. Sí, podríamos ejecutar `composer require http-client` y Flex lo traduciría al nombre final del paquete. Es sólo un atajo a la hora de añadir paquetes.

Si quieres ver todos los alias disponibles, ve a un repositorio llamado symfony/recetas... y luego haz clic en el enlace a `RECIPES.md`. A la derecha, ¡ahí están!

El sistema de recetas

El segundo superpoder que Symfony Flex añade a Composer son las recetas. Son fascinantes. Cuando añades un nuevo paquete, puede tener una receta, que es básicamente un conjunto de archivos que se añadirán a tu proyecto. Y resulta que todos los archivos con los que empezamos -en `bin/`, `config/`, `public/` - proceden de las recetas de los paquetes que se instalaron originalmente.

Por ejemplo, `symfony/framework-bundle` es el paquete "core" del Framework Symfony. Puedes comprobar su receta yendo al repositorio `symfony/recipes` y navegando a `symfony`, `framework-bundle`, y luego a la última versión. Echa un vistazo a `config/packages/`: ¡la mayoría de las cosas con las que empezamos proceden de esta receta!

Otra forma de ver las recetas es en tu línea de comandos. Ejecuta:

```
composer recipes
```

Aparentemente se instalaron las recetas de cuatro paquetes diferentes. Y podíamos obtener información sobre cualquiera de ellos añadiendo su nombre al final del comando.

De todos modos, las recetas son increíbles porque podemos instalar un paquete y obtener al instante cualquier archivo que necesitemos. En lugar de complicarnos con la configuración, nos ponemos manos a la obra.

Instalar PHP CS Fixer

Vamos a probar esto: añadamos un nuevo paquete llamado PHP-CS-Fixer que nos proporcionará un archivo ejecutable para arreglar el estilo de nuestro código. Por ejemplo, en `src/Controller/MainController.php`, si sigues las normas de codificación de PHP, la llave debe estar en la línea siguiente a una función. Si hiciéramos algo así, nuestro archivo violaría ahora esas normas. Eso no dañaría nada, pero ya sabes, queremos que nuestro código tenga un aspecto limpio. Y PHP-CS-Fixer puede ayudarnos a hacerlo.

Para instalarlo, ejecuta:

```
composer require cs-fixer-shim
```

Y sí, se trata de un alias. Encima, el paquete verdadero es `php-cs-fixer/shim`.

¿Este paquete venía con una receta? ¡Pues sí! El `Configuring php-cs-fixer/shim` nos lo indica. Pero, también podemos verlo ejecutando:

```
git status
```

El hecho de que `composer.json` y `composer.lock` estén modificados es un comportamiento 100% normal de Composer. Puedes ver que `composer.json` tiene la nueva biblioteca bajo la clave `require`.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 9
        "php-cs-fixer/shim": "^3.46",
// ... lines 11 - 16
    },
// ... lines 18 - 69
}
```

Pero todos los demás archivos modificados o nuevos lo son gracias a la receta del paquete.

Investigando la receta

¡Vamos a investigar esto! Abre `.gitignore`. ¡Genial! En la parte inferior, ha añadido dos nuevas entradas para dos archivos comunes que querrás ignorar cuando utilices PHP CS fixer.

.gitignore

```
// ... lines 1 - 11
###> php-cs-fixer/shim ###
/.php-cs-fixer.php
/.php-cs-fixer.cache
###
```

La receta también añadió un nuevo archivo `.php-cs-fixer.dist.php`. Este es el archivo de configuración de CS Fixer. ¡Y compruébalo!

.php-cs-fixer.dist.php

```
1
2
3  $finder = (new PhpCsFixer\Finder())
4      ->in(__DIR__)
5      ->exclude('var')
6  ;
7
8  return (new PhpCsFixer\Config())
9      ->setRules([
10         '@Symfony' => true,
11     ])
12     ->setFinder($finder)
13 ;
```

Está prediseñado para funcionar con nuestra aplicación Symfony. Le dice que arregle todos los archivos del directorio actual, pero que ignore el directorio `var/` porque es donde Symfony almacena sus archivos de caché. También le dice que utilice un conjunto de reglas llamado Symfony. Eso significa que queremos que el estilo de nuestro código coincida con el estilo de Symfony. La cuestión es en lugar de perder el tiempo buscando esta configuración por defecto... ¡simplemente la cogemos!

El último archivo modificado es `symfony.lock`. Esto mantiene un registro de qué recetas tenemos instaladas y en qué versión. Y sí, vamos a enviar todos estos archivos a nuestro repositorio.

Utilizar PHP-CS-Fixer

Ahora que hemos instalado el paquete, vamos a utilizarlo. Para ello, ejecuta:

```
./vendor/bin/php-cs-fixer
```

Eso mostrará todos los comandos disponibles. El que queremos se llama fix. Pruébalo:

```
./vendor/bin/php-cs-fixer fix
```

Y... ¡sí! ¡Ha encontrado la infracción en `MainController.php`! Cuando vamos a ese archivo... ¡sí! Movió mi llave rizada desde el final de la línea hasta la línea siguiente. Es fantástico.

A continuación, vamos a conocer e instalar una de mis bibliotecas favoritas de todo PHP: el motor de plantillas Twig.

Chapter 5: Twig y plantillas

Quiero devolver HTML para esta página. Podríamos poner ese HTML dentro del controlador... pero eso se va a poner feo rápidamente. Afortunadamente, hay una forma mejor: utilizar una biblioteca de plantillas llamada Twig.

Instalación de Twig

En tu terminal, asegúrate de haber confirmado tus cambios, porque quiero ver qué añade la receta de este nuevo paquete a nuestro proyecto. Ya lo he hecho. Instálalo con

```
composer require twig
```

Composer "Paquetes"

Probablemente reconozcas que `twig` es un alias... esta vez de un paquete llamado `symfony/twig-pack`. Y la palabra "paquete" es importante en Symfony. Un paquete es... una especie de paquete falso que ayuda a instalar varios paquetes a la vez.

Observa: abre `composer.json`. En lugar de un nuevo paquete aquí llamado `symfony/twig-pack`, tenemos tres nuevos paquetes... ¡y `twig-pack` ni siquiera es uno de ellos!

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 15
        "symfony/twig-bundle": "7.0.*",
// ... line 17
        "twig/extra-bundle": "^2.12|^3.0",
        "twig/twig": "^2.12|^3.0"
    },
// ... lines 21 - 72
}
```

Los tres paquetes nos dan todo lo que necesitamos para una configuración Twig completa y robusta. Así que cuando veas la palabra "paquete", no es gran cosa: sólo un atajo para instalar varios paquetes a la vez.

Paquetes Symfony

Vale, ¡vamos a ver qué ha hecho la receta! Ejecuta:

```
git status
```

Vemos los habituales `composer.json`, `composer.lock` y `symfony.lock`. Pero, por primera vez, también vemos una modificación de `config/bundles.php`. Un bundle es un paquete PHP que se integra con Symfony... es básicamente un plugin de Symfony. Siempre que instales un bundle, tienes que activarlo en este archivo `bundles.php`. Pero sinceramente, el sistema de recetas siempre lo hará por nosotros... así que es bueno darse cuenta, pero nunca editaremos este archivo a mano.

config/bundles.php

```
<?php

return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
    Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
];
```

La receta Twig

Lo segundo que hizo la receta fue crear un archivo `config/packages/twig.yaml`. El propósito de cada archivo en `config/packages/` es configurar un bundle.

config/packages/twig.yaml

```yaml
twig:
    default_path: '%kernel.project_dir%/templates'

when@test:
    twig:
        strict_variables: true
```

Por ejemplo, `twig.yaml` controla el comportamiento de TwigBundle. Esta línea de aquí le dice a Twig:

> *"¡Oye! Todos mis archivos de plantilla terminarán en `.twig`."*

Hay muchas más cosas que podríamos configurar, pero no hace falta. Y profundizaremos en estos archivos de configuración en el próximo tutorial.

Lo último que hizo la receta fue añadir un directorio `templates/`, que.... ¡lo has adivinado! Es donde vivirán nuestros archivos de plantilla Incluso nos inició con un archivo `base.html.twig` del que hablaremos en unos minutos.

Renderizar una plantilla

Así que ¡vamos a renderizar nuestra primera plantilla! Para ello, haz que tu controlador extienda una clase base llamada `AbstractController`. Asegúrate de pulsar el tabulador para que se añada la sentencia `use` en la parte superior. Extender esta clase base es opcional, pero nos proporciona un montón de métodos abreviados.

src/Controller/MainController.php

```php
// ... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
// ... lines 6 - 8
class MainController extends AbstractController
{
// ... lines 11 - 15
}
```

Por ejemplo, copia la cadena y luego, para renderizar una plantilla escribe `return $this->render()` y pasa un nombre de archivo a una plantilla. Utiliza: `main/homepage.html.twig`.

src/Controller/MainController.php

```
// ... lines 1 - 8
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        return $this->render('main/homepage.html.twig');
    }
}
```

El nombre de archivo de tu plantilla puede ser el que quieras, pero lo estándar es tener un directorio que coincida con el nombre de tu controlador y un nombre de archivo que coincida con el nombre de tu método.

¡Vamos a crearlo! En `templates/`, añade un nuevo directorio llamado `main`. Y dentro de él, un archivo llamado `homepage.html.twig`. Luego añade un `h1` y ponlo alrededor de todo.

templates/main/homepage.html.twig

```
<h1>
    Starshop: your monopoly-busting option for Starship parts!
</h1>
```

¡Hagamos esto! Actualiza. ¡Ya está!

Y por cierto, ¿qué devuelve nuestro controlador? Sigue siendo un objeto `Response`! Lo sé porque tenemos un tipo de retorno `Response`... y nuestro código no está explotando. `render()` es sólo un atajo para renderizar esta plantilla, coger esa cadena de HTML y ponerla en un objeto `Response`. Así que, aunque estemos renderizando una plantilla, seguimos volviendo a la idea de que un controlador devuelve una respuesta.

Pasar datos a una plantilla

¿Qué hay de pasar datos a la plantilla? Quizá consultemos la base de datos y le pasemos el número total de naves estelares. Aún no tenemos una base de datos en nuestra aplicación, así

que vamos a fingirlo diciendo que `$starshipCount` es igual a... No sé... 457. Parece un número falso creíble.

src/Controller/MainController.php

```
↕ // ... lines 1 - 8
9  class MainController extends AbstractController
10 {
11     #[Route('/')]
12     public function homepage(): Response
13     {
14         $starshipCount = 457;
↕ // ... lines 15 - 18
19     }
20 }
```

Para pasar variables a la plantilla, añade un segundo argumento a `render()`: una matriz. Pasa `numberOfStarships` ajustado a `$starshipCount`. La clave se convertirá en el nombre de la variable dentro de la plantilla Twig.

src/Controller/MainController.php

```
↕ // ... lines 1 - 8
9  class MainController extends AbstractController
10 {
11     #[Route('/')]
12     public function homepage(): Response
13     {
14         $starshipCount = 457;
15
16         return $this->render('main/homepage.html.twig', [
17             'numberOfStarships' => $starshipCount,
18         ]);
19     }
20 }
```

Renderizar variables

En la plantilla, añadiré un div y algo de texto. Para imprimir el número, escribe `{{`, el nombre de la variable, cierra `}}`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships }} starships!
</div>
```

¡Vale! Muévete y pruébalo. ¡Ya está! ¡Y acabamos de ver nuestro primer código Twig!

Twig es su propio lenguaje, pero es superamigable. Sólo tiene tres sintaxis diferentes. La primera es `{{` y yo la llamo la sintaxis "decir algo". Si estás imprimiendo algo, utilizarás `{{`. Dentro de los rizos, estamos escribiendo Twig, que es muy similar a JavaScript.

Etiquetas Twig y la sintaxis "hacer algo

Por ejemplo, podríamos imprimir la cadena `'numberOfStarships'`... o la variable `numberOfStarships`... o incluso `numberOfStarships` veces 10.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships * 10 }} starships!
</div>
```

La segunda sintaxis de las tres empieza por `{%`. Yo la llamo la sintaxis "hacer algo". Esto no imprime nada. En su lugar, se utiliza para construcciones del lenguaje como las sentencias `if`, los bucles for o establecer una variable.

Para hacer una sentencia if, di `if numberOfStarships > 400`, y ciérrala con `{% endif %}`. Dentro, añadiré un comentario.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships * 10 }} starships!

    {% if numberOfStarships > 400 %}
        <p>
            That's a shiploads of ships!
        </p>
    {% endif %}
</div>
```

¡Pruébalo! ¡Eso también funciona!

Twig es su propia biblioteca, pero está mantenida por Symfony... así que sus documentos están en https://twig.symfony.com. Haz clic en el enlace "Docs" y desplázate hacia abajo. ¿Ves las "etiquetas"? Resulta que hay un número finito de cosas que puedes utilizar con la sintaxis "hacer algo": son estas etiquetas. Por ejemplo, no puedes decir `{% applesauce` ... simplemente no funcionará. Sólo puedes usar `{%` y luego una de estas etiquetas. La lista es bastante corta... y probablemente sólo utilice 5 de ellas a diario.

La tercera y última sintaxis de Twig ni siquiera es una sintaxis: es para los comentarios. `{#` para escribir un comentario.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
<div>
    Browse through {{ numberOfStarships * 10 }} starships!

    {% if numberOfStarships > 400 %}
        <p>
            {# Do you think "shiploads" will pass the legal team? #}
            That's a shiploads of ships!
        </p>
    {% endif %}
</div>
```

Representación de una matriz asociativa

Así que estamos pasando un simple número a Twig e imprimiéndolo. Pero Twig puede manejar cualquier dato complejo que le pases. Por ejemplo, en el controlador, crea una nueva variable `$myShip`, configurada como una matriz asociativa. Luego pásala a la plantilla como una nueva variable: `myShip`.

src/Controller/MainController.php

```
// ... lines 1 - 8
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $starshipCount = 457;
        $myShip = [
            'name' => 'USS LeafyCruiser (NCC-0001)',
            'class' => 'Garden',
            'captain' => 'Jean-Luc Pickles',
            'status' => 'under construction',
        ];

        return $this->render('main/homepage.html.twig', [
            'numberOfStarships' => $starshipCount,
            'myShip' => $myShip,
        ]);
    }
}
```

En la plantilla, añade otro `div`... algo de texto y una tabla para imprimir los datos. En el `<td>`, no podemos simplemente imprimir `myShip`... porque imprimir una matriz asociativa no tiene sentido en PHP... y por tanto no tiene sentido en Twig. Obtendrás el famoso error sobre la conversión de array a cadena.

Lo que queremos es imprimir la clave `name` de esa matriz. La forma de hacerlo es exactamente igual que en JavaScript: `myShip.name`.

Ya está Y... funciona. Voy a pegar el resto de nuestra plantilla, que imprime las demás claves de la matriz. Tiene buena pinta.

templates/main/homepage.html.twig

```
// ... lines 1 - 15
<div>
    <h2>My Ship</h2>

    <table>
        <tr>
            <th>Name</th>
            <td>{{ myShip.name }}</td>
        </tr>
        <tr>
            <th>Class</th>
            <td>{{ myShip.class }}</td>
        </tr>
        <tr>
            <th>Captain</th>
            <td>{{ myShip.captain }}</td>
        </tr>
        <tr>
            <th>Status</th>
            <td>{{ myShip.status }}</td>
        </tr>
    </table>
</div>
```

Funciones y filtros Twig

Twig tiene algunos trucos más en la manga, pero nada complejo. Tiene funciones... que funcionan como las funciones de cualquier lenguaje. También tiene algo llamado pruebas, que son un poco exclusivas de Twig, pero bastante sencillas de entender. Mi concepto favorito son probablemente los filtros, que son básicamente funciones con una sintaxis más fresca y moderna.

Por ejemplo, hay un filtro llamado `upper` para enviar una cadena a mayúsculas. Para utilizar un filtro, busca la cadena que quieres convertir a mayúsculas y añade `|` y `upper`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 15
16 <div>
17     <h2>My Ship</h2>
18 
19     <table>
↕ // ... lines 20 - 27
28         <tr>
29             <th>Captain</th>
30             <td>{{ myShip.captain|upper }}</td>
31         </tr>
↕ // ... lines 32 - 35
36     </table>
37 </div>
```

El valor de la izquierda se pasa a través del filtro, muy parecido a utilizar una tubería en la línea de comandos. Funciona de maravilla.... y puedes volverte loco con los filtros: pasar a `upper`, luego a `lower` y después a `title` mayúsculas sólo para confundir a tus compañeros.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 15
16 <div>
17     <h2>My Ship</h2>
18 
19     <table>
↕ // ... lines 20 - 27
28         <tr>
29             <th>Captain</th>
30             <td>{{ myShip.captain|upper|lower|title }}</td>
31         </tr>
↕ // ... lines 32 - 35
36     </table>
37 </div>
```

Vale, acabamos de aprender prácticamente todo Twig en una sesión, excepto una cosa: la herencia de plantillas. Eso a continuación.

Chapter 6: Herencia de plantillas Twig

¿Qué tal si añadimos un diseño a nuestra página, como una cabecera y un pie de página? Echa un vistazo al HTML de la página: es sólo el HTML de la plantilla. No hay nada especial en Twig para que un diseño base con un encabezado y un pie de página se envuelva automáticamente alrededor de nuestro contenido. Lo que tengas en tu plantilla es lo que obtendrás en la página.

Sin embargo, la receta Twig añadió un archivo de diseño base llamado `base.html.twig`.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 128%22><text y=%221.2em%22 font-size=%2296%22>⚫</text></svg>">
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Ahora es muy sencillo, pero aquí es donde añadiremos nuestra navegación superior, el pie de página y cualquier otra cosa que deba aparecer en cada página. La pregunta es: ¿cómo podemos hacer que nuestra plantilla utilice esto?

Ampliando el diseño base

Con una función genial llamada herencia de plantillas. En `homepage.html.twig`, en la parte superior, escribe `{% extends` y luego el nombre de la plantilla base: `base.html.twig`. Y

fíjate: esta es la etiqueta hacer algo. No estamos imprimiendo esta plantilla, le estamos diciendo a Twig que queremos ampliarla.

templates/main/homepage.html.twig

```
{% extends 'base.html.twig' %}

<h1>
    Starshop: your monopoly-busting option for Starship parts!
</h1>
// ... lines 6 - 40
```

Si no hacemos nada más y actualizamos, obtendremos un error:

> *"una plantilla que extiende otra no puede incluir contenido fuera de los bloques Twig."*

Hmmm. Cuando extiendes una plantilla, le dices a Twig que quieres renderizar tu plantilla dentro de ese diseño base. Pero... Twig no tiene ni idea de dónde debe ir nuestro contenido. ¿Debería coger el HTML de nuestra página de inicio y ponerlo aquí abajo? ¿O aquí arriba? ¿O justo ahí? No lo sabe Así que lanza ese error.

La forma de decírselo es mediante estos bloques. Los bloques son huecos en los que una plantilla hija puede poner contenido. Y te habrás fijado en un bloque llamado `body`... que es exactamente donde queremos que vaya nuestro contenido. Para ponerlo ahí, rodea todo el contenido con un `{% block body %}`... y en la parte inferior, `{% endblock %}`.

templates/main/homepage.html.twig

```
{% extends 'base.html.twig' %}

{% block body %}
<h1>
    Starshop: your monopoly-busting option for Starship parts!
</h1>

<div>
// ... lines 9 - 16
</div>

<div>
// ... lines 20 - 39
</div>
{% endblock %}
```

Y ahora... ¡está vivo! No parece muy diferente, pero estamos dentro del diseño base.

Esto se llama herencia de plantillas porque funciona exactamente igual que la herencia de clases PHP. Imagina que tienes una clase `Homepage` que extiende a una clase `Base`. Esa clase `Base` tiene un método `body()`, y nosotros anulamos ese método `body()` en la clase `Homepage`. Es el mismo concepto en Twig.

Reemplazar el título de la página

Y estos nombres de bloques -como `javascripts`, `stylesheets` y `body` - no son nombres especiales... y no están registrados en ninguna parte. Siéntete libre de crear nuevos bloques como quieras y cuando quieras. Por ejemplo, supongamos que queremos cambiar el `title` de la página desde una plantilla hija. En este caso, la receta ya nos ha proporcionado un bloque llamado `title` para hacerlo. Y este bloque tiene contenido por defecto... por eso ya vemos `Welcome` en la pestaña del navegador. Anulemos esto en nuestra plantilla.

templates/base.html.twig

```
↕ // ... line 1
2 <html>
3     <head>
↕ // ... line 4
5         <title>{% block title %}Welcome!{% endblock %}</title>
↕ // ... lines 6 - 11
12     </head>
↕ // ... lines 13 - 15
16 </html>
```

En cualquier lugar fuera del bloque `body`, di `{% block title %}`, escribe algo, y luego `{% endblock %}`.

templates/main/homepage.html.twig

```
1 {% extends 'base.html.twig' %}
2
3 {% block title %}Starshop: Beam up some parts!{% endblock %}
4
5 {% block body %}
↕ // ... lines 6 - 42
43 {% endblock %}
```

Sustituir frente a añadir el bloque padre

Y ahora, ¡ya está! ¡Nuevo título! Y fíjate en que cuando anulamos un bloque, lo anulamos por completo. Ya no vemos la palabra `Welcome`. Ocasionalmente, puede que quieras añadir al bloque padre en lugar de sustituirlo. Puedes hacerlo diciendo `{{ parent() }}`.

¡Esto está muy bien! La función `parent()` coge el contenido del bloque `title` de la plantilla padre. Luego utilizamos `{{` para imprimirlo. Esta vez vemos la bienvenida y luego nuestro título.

Pero como en realidad no queremos eso, lo eliminaré.

Comprobación de estado: estamos devolviendo HTML y tenemos un diseño base. Sí, nuestro sitio sigue siendo horriblemente feo, pero lo arreglaremos dentro de un rato.

A continuación, vamos a ejecutar un comando y acceder al instante a algunas de las herramientas de depuración más potentes de la web.

Chapter 7: Depurando con el Asombroso Perfilador

Symfony presume de tener algunas de las herramientas de depuración más épicas de todo Internet. Pero como las aplicaciones Symfony empiezan tan pequeñas, aún no las tenemos instaladas. Es hora de arreglarlo. Dirígete a tu terminal y, como antes, confirma todos tus cambios para que podamos comprobar lo que harán las recetas. Ya lo he hecho.

Instalar las herramientas de depuración

Ejecuta:

```
composer require debug
```

¡Sí! Es otro alias de Flex. E... instala un paquete. Esto instala cuatro paquetes diferentes que añaden una variedad de bondades de depuración a nuestro proyecto. Gira y abre `composer.json`.

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 14
        "symfony/monolog-bundle": "^3.0",
// ... lines 16 - 20
    },
// ... lines 22 - 78
}
```

Vale, el paquete ha añadido una nueva línea bajo la clave `require` para `monolog-bundle`. Monolog es una biblioteca de registro.

Luego, al final, ha añadido tres paquetes a la sección `require-dev`.

composer.json

```
{
// ... lines 2 - 73
    "require-dev": {
        "symfony/debug-bundle": "7.0.*",
        "symfony/stopwatch": "7.0.*",
        "symfony/web-profiler-bundle": "7.0.*"
    }
}
```

Se conocen como dependencias de desarrollo... lo que significa que no se descargarán cuando los despliegues en producción. Por lo demás, funcionan igual que los paquetes de la clave `require`. Los tres ayudan a impulsar algo llamado perfilador, que veremos dentro de un minuto.

Antes de hacerlo, vuelve a tu terminal y ejecuta

```
git status
```

para que podamos ver lo que hicieron las recetas. Vale: actualizó los archivos normales, habilitó unos cuantos bundles nuevos y nos dio tres archivos de configuración nuevos para esos bundles.

¿Cuál es el resultado final de todo esto nuevo? Bueno, en primer lugar, ahora tenemos una biblioteca de registros. Así que, como por arte de magia, los registros empezarán a aparecer en un directorio `var/log/`.

Hola barra de herramientas de depuración web y perfilador

Pero el momento alucinante ocurre cuando actualizamos la página. ¡Woh! Una nueva y hermosa barra negra en la parte inferior llamada barra de herramientas de depuración web.

Está repleta de información. Aquí podemos ver la ruta y el controlador de esta página. Eso facilita ir a cualquier página de tu sitio -quizá una que ni siquiera hayas construido- y encontrar rápidamente el código que hay detrás. También podemos ver cuánto tardó en cargarse esta página, cuánta memoria utilizó, e incluso la plantilla Twig que se renderizó y cuánto tardó.

Pero la verdadera magia de la barra de herramientas de depuración web ocurre cuando haces clic en cualquiera de estos enlaces: saltas al perfilador. Éste tiene diez veces más información: detalles sobre la petición y la respuesta, registros que se produjeron mientras se cargaba la página, detalles de enrutamiento e incluso estadísticas sobre las plantillas Twig que se procesaron. Aparentemente, se estaban renderizando seis plantillas: la principal, el diseño base y algunas otras que alimentan la barra de herramientas de depuración web, que, por cierto, no se renderizarán ni se mostrarán cuando pasemos a producción. Pero de eso hablaremos en el próximo tutorial.

Luego está probablemente mi sección favorita: Rendimiento. Aquí se divide todo el tiempo de carga de nuestra página en diferentes partes. Esto me encanta. A medida que aprendas más sobre Symfony, te irás familiarizando con lo que son estas diferentes piezas. Esta sección es útil para saber qué parte de tu código puede estar ralentizando la página... pero también es una forma fantástica de profundizar en Symfony y entender todas sus piezas móviles.

Vamos a utilizar el perfilador a lo largo de esta serie, pero pasemos a otra herramienta de depuración: ¡una que ha estado instalada en nuestra aplicación todo este tiempo!

¡Hola bin/console!

Dirígete a la línea de comandos y ejecuta:

```
php bin/console
```

O, en la mayoría de las máquinas, puedes decir simplemente `./bin/console`. Esta es la consola de Symfony, y está repleta de comandos que pueden hacer todo tipo de cosas Aprenderemos sobre ellos a lo largo del camino. También puedes añadir tus propios comandos, cosa que haremos al final del tutorial.

Fíjate en que muchos de ellos empiezan por `debug`, como `debug:router`. Pruébalo:

```
php bin/console debug:router
```

¡Genial! Esto nos muestra todas las rutas de nuestra aplicación: la ruta de la página de inicio en la parte inferior y un montón de rutas añadidas por Symfony en el entorno `dev` que alimentan la barra de herramientas de depuración web y el perfilador.

Otro comando es `debug:twig`:

```
php bin/console debug:twig
```

Nos indica todas las funciones, filtros u otros elementos de Twig que existen en nuestra aplicación. Es como la documentación de Twig... salvo que también incluye funciones y filtros adicionales añadidos a Twig por los bundles que hemos instalado. Genial.

Estos comandos de `debug` son superútiles, y seguiremos probando más de ellos por el camino.

A continuación, vamos a crear nuestra primera ruta API y a conocer el potente componente serializador de Symfony.

Chapter 8: Creación de rutas API JSON

Si quieres crear una API, puedes hacerlo absolutamente con Symfony. De hecho, es una opción fantástica, en parte gracias a API Platform. Se trata de un marco para crear APIs construido sobre Symfony que agiliza la construcción de tu API y crea una API más robusta de lo que puedas imaginar.

Pero también es bastante sencillo devolver JSON desde un controlador. Veamos si podemos devolver algunos datos del barco como JSON.

Creación de la nueva Ruta y Controlador

Esta será nuestra segunda página. Bueno, en realidad es un "punto final", pero será nuestra segunda combinación de ruta y controlador. En `MainController`, podríamos añadir otro método aquí. Pero para organizarnos, vamos a crear una clase de controlador totalmente nueva. Iré a Nuevo -> Clase PHP y la llamaré `StarshipApiController`.

Como he ido a Nuevo -> Clase PHP, me ha creado la clase y el espacio de nombres ¡Súper bien! Además, en adelante, cada vez que cree un controlador, extenderé inmediatamente `AbstractController`... porque esos atajos son agradables y no hay inconveniente.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 2
3  namespace App\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
↕ // ... lines 6 - 8
9  class StarshipApiController extends AbstractController
10 {
↕ // ... lines 11 - 36
37 }
```

Añade un `public function getCollection()` porque esto devolverá información sobre una colección de naves estelares. Y, como siempre, puedes añadir el tipo de retorno `Response` u omitirlo. Encima de esto, añade la ruta con `#[Route()]`. Selecciona la de `Attribute` y pulsa tabulador.

Así que acabo de utilizar el autocompletado para añadir las declaraciones `use` para `AbstractController`,`Route`, y `Response`. Asegúrate de que las tienes todas. Para la URL, ¿qué tal`/api/starships`.

En su interior, pegaré una variable `$starships` que se establecerá en una matriz de tres matrices asociativas de datos de naves estelares.

Devolver JSON

Probablemente puedas imaginar qué aspecto tendrá esto como JSON. ¿Cómo lo convertimos en JSON? Bueno, puede ser así de sencillo: `return new Response` con`json_encode($starships)`.

¡Pero podemos hacerlo mejor! En lugar de eso, devuelve `$this->json($starships)`.

src/Controller/StarshipApiController.php

```php
// ... lines 1 - 2
namespace App\Controller;

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

class StarshipApiController extends AbstractController
{
    #[Route('/api/starships')]
    public function getCollection(): Response
    {
        $starships = [
            [
                'name' => 'USS LeafyCruiser (NCC-0001)',
                'class' => 'Garden',
                'captain' => 'Jean-Luc Pickles',
                'status' => 'taken over by Q',
            ],
            [
                'name' => 'USS Espresso (NCC-1234-C)',
                'class' => 'Latte',
                'captain' => 'James T. Quick!',
                'status' => 'repaired',
            ],
            [
                'name' => 'USS Wanderlust (NCC-2024-W)',
                'class' => 'Delta Tourist',
                'captain' => 'Kathryn Journeyway',
                'status' => 'under construction',
            ],
        ];

        return $this->json($starships);
    }
}
```

¡Vamos a probarlo! Busca tu navegador y dirígete a `/api/starships`. Vaya, ha sido fácil. Si te preguntas por qué el JSON está estilizado y tiene un aspecto chulo, no es cosa de Symfony. Tengo instalada una extensión de Chrome llamada JSONVue.

Añadir una clase modelo

Ahora, en el mundo real, cuando empecemos a consultar la base de datos, vamos a trabajar con objetos, no con matrices asociativas. No añadiremos una base de datos en este tutorial, pero podemos empezar a utilizar objetos para nuestros datos para hacer las cosas más realistas. En el directorio `src/`, crea un nuevo subdirectorio llamado `Model`.

Vale, algo importante: lo que vamos a hacer no tiene absolutamente nada que ver con Symfony. Simplemente estoy mirando este array y pensando:

> *"¿Sabes qué? En lugar de pasar por este array asociativo con `name`, `class`, `captain`, y `status` claves, prefiero tener una clase `Starship` y pasar objetos."*

Así que, por mi cuenta, independientemente de Symfony, he decidido crear un directorio `Model`-que podría llamarse cualquier cosa- y dentro una nueva clase llamada `Starship`. Y como esta clase es sólo para ayudarnos, podemos darle el aspecto que queramos, y no necesita extender ninguna clase base.

src/Model/Starship.php

```
↕ // ... lines 1 - 2
3 namespace App\Model;
4
5 class Starship
6 {
↕ // ... lines 7 - 39
40 }
```

Crea un `public function __construct()` con cinco propiedades: una `private int $id`, y luego cuatro propiedades más para cada una de las cuatro claves que tenemos en la matriz: `private string $name`, `private string $class`, `private string $captain` y `private string $status`.

src/Model/Starship.php

```
// ... lines 1 - 2
namespace App\Model;

class Starship
{
    public function __construct(
        private int $id,
        private string $name,
        private string $class,
        private string $captain,
        private string $status,
    ) {
    }
// ... lines 15 - 39
}
```

Ah, y mi editor está resaltando este archivo porque hemos instalado PHP-CS-Fixer y ha encontrado una violación del estilo del código. Puedo hacer clic aquí para corregirlo o ir aquí y pulsar Alt+Enter para corregirlo allí. ¡Súper bonito!

De todas formas, si no estás familiarizado con la sintaxis de este constructor, esto crea un constructor con cinco argumentos y, al mismo tiempo, crea cinco propiedades que se establecerán a lo que pasemos a estos argumentos.

Pero, como he decidido que estas propiedades sean privadas, si instanciáramos un nuevo objeto `Starship`... ¡no podríamos leer ninguno de los datos! Para permitirlo, podemos crear métodos getter. Pero, no voy a hacer esto a mano. En su lugar, ve a la opción de menú Código -> Generar -o Cmd + N en Mac-, selecciona getters y genera un getter para cada propiedad.

src/Model/Starship.php

```php
// ... lines 1 - 2
namespace App\Model;

class Starship
{
    public function __construct(
        private int $id,
        private string $name,
        private string $class,
        private string $captain,
        private string $status,
    ) {
    }

    public function getId(): int
    {
        return $this->id;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function getClass(): string
    {
        return $this->class;
    }

    public function getCaptain(): string
    {
        return $this->captain;
    }

    public function getStatus(): string
    {
        return $this->status;
    }
}
```

¡Qué bien! Cinco nuevos y brillantes métodos getter públicos.

Crear los objetos modelo

Vale, de vuelta en nuestro controlador, convirtamos estas matrices en objetos: `new Starship()` - pulsa tabulador, para que añada la declaración `use` - luego dale un id de, qué tal, 1... y transfiere los otros valores para `name`, `class`, `captain`, y finalmente `status`.

Y así de fácil, ¡ya tenemos nuestro primer objeto! Resaltaré las otras dos matrices y pegaré los dos objetos para ahorrar tiempo.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 4
5  use App\Model\Starship;
↕ // ... lines 6 - 9
10 class StarshipApiController extends AbstractController
11 {
↕ // ... line 12
13     public function getCollection(): Response
14     {
15         $starships = [
16             new Starship(
17                 1,
18                 'USS LeafyCruiser (NCC-0001)',
19                 'Garden',
20                 'Jean-Luc Pickles',
21                 'taken over by Q'
22             ),
23             new Starship(
24                 2,
25                 'USS Espresso (NCC-1234-C)',
26                 'Latte',
27                 'James T. Quick!',
28                 'repaired',
29             ),
30             new Starship(
31                 3,
32                 'USS Wanderlust (NCC-2024-W)',
33                 'Delta Tourist',
34                 'Kathryn Journeyway',
35                 'under construction',
36             ),
37         ];
↕ // ... lines 38 - 39
40     }
41 }
```

Ahora tenemos una matriz de 3 objetos `Starship`... que queda más bonita. Y los pasamos a `$this->json()`. ¿Seguirá funcionando? Por supuesto que no ¡Obtenemos una matriz de

tres objetos vacíos!

Eso es porque, internamente, `$this->json()` utiliza la función PHP `json_encode()`... y esa función no puede manejar propiedades privadas. Lo que necesitamos es algo más inteligente: algo que pueda reconocer que, aunque la propiedad `name` es privada, tenemos un método público `getName()` al que se puede llamar para leer el valor de esa propiedad.

Hola Serializador Symfony

¿Existe alguna herramienta que haga eso? Bueno, ¿recuerdas que Symfony es un enorme conjunto de componentes que resuelven problemas individuales? Un componente se llama serializador, y su trabajo consiste en tomar objetos y serializarlos a JSON... o tomar JSON y deserializarlo de nuevo a objetos. Y puede manejar totalmente situaciones en las que tienes propiedades privadas con métodos getter públicos.

Así que ¡a instalarlo!

```
composer require serializer
```

Y una vez más, amigos, sí, esto es un alias... y es un alias de un paquete. Este paquete instala el paquete `symfony/serializer`, así como algunos otros que lo hacen funcionar de forma realmente robusta.

Ahora, sin hacer nada más, vuelve atrás, actualiza, ¿y funciona? ¿Cómo?

Resulta que el método `$this->json()` es inteligente. Para verlo, mantén pulsado Comando en un Mac o Ctrl en otras máquinas y haz clic en el nombre del método para saltar al archivo principal de Symfony donde se encuentra.

¡Ah! El código aquí aún no tendrá todo el sentido, pero detecta si el sistema serializador está disponible.... y, si lo está, lo utiliza para transformar el objeto a JSON.

Pero, ¿qué quiero decir exactamente con "sistema serializador"? ¿Y cuál es la clave `serializer`... dentro de esta cosa del contenedor? O, ¿qué pasaría si necesitáramos transformar un objeto a JSON en otro lugar que no fuera nuestro controlador... donde no

tuviéramos acceso al acceso directo `->json()`? ¿Cómo podríamos acceder al sistema serializador desde allí?

Amigos, es hora de conocer el concepto más importante de Symfony: los servicios.

Chapter 9: Los Servicios: La columna vertebral de todo

Hablemos de los servicios. Son el concepto más importante en Symfony. Y una vez que los entiendas, sinceramente, serás capaz de hacer cualquier cosa.

¿Qué es un Servicio?

En primer lugar, un servicio es un objeto que hace un trabajo. Eso es todo. Por ejemplo, si instancias un objeto `Logger` que tiene un método `log()`, ¡eso es un servicio! Funciona: ¡registra cosas! O si creaste un objeto de conexión a la base de datos que hace consultas a la base de datos, entonces... ¡sí! Eso también es un servicio.

Entonces... si un servicio es sólo un objeto que funciona... ¿qué objetos perezosos no son servicios? Nuestra clase `Starship` es un ejemplo perfecto de no-servicio. Su función principal no es hacer trabajo: es guardar datos. Claro, tiene unos cuantos métodos públicos... e incluso podrías poner algo de lógica dentro de estos métodos para hacer algo. Pero en última instancia, no es un trabajador, es un poseedor de datos.

¿Y las clases controladoras? Sí, también son servicios. Su trabajo consiste en crear objetos de respuesta.

De todas formas, todo el trabajo que se hace en Symfony lo hace en realidad un servicio. ¿Escribir mensajes de registro en este archivo? Sí, hay un servicio para eso. ¿Descubrir qué ruta coincide con la URL actual? ¡Ese es el servicio `router`! ¿Y la representación de una plantilla Twig? Sí, resulta que el método `render()` es un atajo para encontrar el objeto de servicio correcto y llamar a un método en él.

El contenedor y debug:container

A veces también oirás que estos servicios están organizados en un gran objeto llamado "contenedor de servicios". Puedes pensar en el contenedor como en una gigantesca matriz

asociativa de objetos de servicio, cada uno con un identificador único. ¿Quieres ver una lista de todos los servicios de nuestra aplicación ahora mismo? Yo también

Busca tu terminal y ejecuta:

```
bin/console debug:container
```

¡Son muchos servicios! Déjame hacerlo más pequeño para que cada uno quepa en su propia línea... mejor.

A la izquierda, vemos el ID de cada servicio. Y a la derecha, la clase del objeto al que corresponde el ID. Genial, ¿verdad?

Vuelve a nuestro controlador y mantén pulsado control o comando para abrir de nuevo el método `json()`. ¡Ahora tiene más sentido! Está comprobando si el contenedor tiene un servicio cuyo ID es `serializer`. Si es así, coge ese servicio del contenedor y llama al método `serialize()` sobre él.

Cuando trabajemos con servicios, no será exactamente así. Pero lo superimportante es que ahora entendemos lo que está pasando.

Los bundles proporcionan servicios

Mi siguiente pregunta es: ¿de dónde vienen estos servicios? Por ejemplo, ¿quién dice que hay un servicio cuyo ID es `twig`... y que cuando se lo pedimos al contenedor, éste debe devolver un objeto Twig `Environment`? La respuesta es: totalmente de bundles. De hecho, ése es el objetivo principal de instalar un nuevo bundle. Los bundles nos proporcionan servicios.

¿Recuerdas cuando instalamos `twig`? Añadió un bundle a nuestra aplicación. ¿Y adivinas qué hizo ese bundle? Sí: nos proporcionó nuevos servicios, incluido el servicio `twig`. Los bundles nos dan servicios... y los servicios son herramientas.

Autocableado

Y aunque hay muchos servicios en esta lista, la gran mayoría son objetos de servicio de bajo nivel que nunca utilizaremos ni nos interesarán. Tampoco nos importará el ID de los servicios la mayoría de las veces.

En su lugar, ejecuta un comando relacionado llamado:

```
php bin/console debug:autowiring
```

Esto nos muestra todos los servicios que son autocableables, que es la técnica que utilizaremos para obtener servicios. Básicamente, es una lista curada de los servicios que es más probable que necesitemos.

Autoconexión del servicio Logger

Hagamos un reto: registremos algo desde nuestro controlador. He aquí un vistazo a cómo enfoco este problema en mi cerebro:

> *"Vale, ¡necesito registrar algo! Y... registrar es trabajo. Y... ¡los servicios funcionan! Por tanto, ¡tiene que haber un servicio de registro que pueda utilizar! ¡Quod erat demonstrandum!"*

Perdonadme, frikis del latín. La cuestión es: si queremos registrar algo, sólo tenemos que encontrar el servicio que hace ese trabajo. ¡De acuerdo! Vuelve a ejecutar el comando pero busca log:

```
php bin/console debug:autowiring log
```

¡Boom! Ha encontrado unos 10 servicios, todos ellos empiezan por `Psr\Log\LoggerInterface`. Hablaremos de cuáles son esos otros servicios en el próximo tutorial. Por ahora, céntrate en el principal. Esto me dice que hay un servicio en el contenedor para un registrador. Y para obtenerlo, podemos autoconectarlo utilizando esta interfaz.

¿Qué significa esto? En el método del controlador donde queremos el logger, añade un argumento de tipo `LoggerInterface` - pulsa tabulador - y luego di `$logger`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 5
6 use Psr\Log\LoggerInterface;
↕ // ... lines 7 - 10
11 class StarshipApiController extends AbstractController
12 {
↕ // ... line 13
14     public function getCollection(LoggerInterface $logger): Response
15     {
↕ // ... lines 16 - 41
42     }
43 }
```

En este caso, el nombre del argumento no es importante: podría ser cualquier cosa. Lo que importa es que el `LoggerInterface` -que corresponde a esta declaración `use` - coincida con el `Psr\Log\LoggerInterface` de `debug:autowiring`.

¡Así de sencillo! Symfony verá esta sugerencia de tipo y dirá:

> *"¡Oh! Como ese type-hint coincide con el tipo de autocableado de este servicio, deben querer que les pase ese objeto de servicio."*

No sé por qué Symfony suena como una rana en mi cabeza. En fin, veamos si esto funciona. Añade `dd($logger)`: `dd()` significa "volcar y morir" y viene de Symfony.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 5
6 use Psr\Log\LoggerInterface;
↕ // ... lines 7 - 10
11 class StarshipApiController extends AbstractController
12 {
↕ // ... line 13
14     public function getCollection(LoggerInterface $logger): Response
15     {
16         dd($logger);
↕ // ... lines 17 - 41
42     }
43 }
```

¡Actualiza! ¡Sí! Imprimió el objeto maravillosamente y luego detuvo la ejecución. ¡Funciona! Symfony nos pasa un objeto `Monolog\Logger`, que implementa ese `LoggerInterface`.

El truco que acabamos de hacer -llamado autocableado- funciona exactamente en dos lugares: los métodos de nuestro controlador y el método `__construct()` de cualquier servicio. Veremos esta segunda situación en el próximo capítulo.

Controlar el comportamiento de los servicios

Y si te estás preguntando de dónde salió este servicio `Logger` en primer lugar... ¡ya sabemos la respuesta! De un bundle. En este caso, `MonologBundle`. Y... ¿cómo podríamos configurar ese servicio... para que, no sé, se registre en un archivo diferente? La respuesta es: `config/packages/monolog.yaml`.

Esta configuración -incluida esta línea- configura `MonologBundle`... lo que en realidad significa que configura cómo funcionan los servicios que nos proporciona MonologBundle. Aprenderemos sobre esta sintaxis porcentual en el próximo tutorial, pero esto le dice al servicio `Logger` que registre en este archivo `dev.log`.

Utilizar el Logger

Bien, ahora que tenemos el servicio `Logger`, ¡vamos a utilizarlo! ¿Cómo? Bueno, por supuesto, puedes leer la documentación. Pero gracias a la sugerencia de tipo, ¡nuestro editor nos ayudará! `LoggerInterface` tiene un montón de métodos. Utilicemos `->info()` y digamos

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 5
6  use Psr\Log\LoggerInterface;
↕ // ... lines 7 - 10
11 class StarshipApiController extends AbstractController
12 {
↕ // ... line 13
14     public function getCollection(LoggerInterface $logger): Response
15     {
16         $logger->info('Starship collection retrieved');
↕ // ... lines 17 - 41
42     }
43 }
```

> *“Colección de naves recuperada.”*

Pruébalo: actualizar. La página funcionó... ¿pero registró algo? Podríamos comprobar el archivo `dev.log`. O podemos utilizar la sección Registro del perfilador para esta petición.

Ver el Perfilador de una petición API

Pero... ¡espera! Esto es una petición API... ¡así que no tenemos esa genial barra de herramientas de depuración web en la parte inferior! Es cierto... ¡pero Symfony sigue recopilando toda esa información! Para acceder al perfilador de esta petición, cambia la URL a `/_profiler`. Esto muestra las peticiones más recientes a nuestra aplicación, con la más reciente en la parte superior. ¿Ves ésta? ¡Es nuestra petición a la API de hace un minuto! Si haces clic en este token... ¡bum! Estamos viendo el perfilador de esa llamada a la API en todo su esplendor... incluyendo una sección de Registro... con nuestro mensaje.

Bien, ahora que hemos visto cómo utilizar un servicio, ¡vamos a crear nuestro propio servicio! Somos imparables!

Chapter 10: Crear tu propio Servicio

Sabemos que los servicios funcionan, y sabemos que Symfony está lleno de servicios que podemos utilizar. Si Ejecutas:

```
php bin/console debug:autowiring
```

Obtenemos el menú de servicios, en el que puedes pedir cualquiera de ellos añadiendo un argumento de tipo con la clase o interfaz correspondiente.

Por supuesto, también hacemos trabajo en nuestro código... con suerte. Ahora mismo, todo ese trabajo se realiza dentro de nuestro controlador, como la creación de los datos de la Nave Estelar. Claro, esto está codificado ahora mismo, pero imagina que fuera trabajo real: como una consulta compleja a una base de datos. Poner la lógica dentro de un controlador está "bien"... pero ¿y si quisiéramos reutilizar este código en otro sitio? ¿Y si, en nuestra página de inicio, quisiéramos obtener un recuento dinámico de las naves estelares tomando estos mismos datos?

Crear la clase de servicio

Para ello, tenemos que trasladar este "trabajo" a un servicio propio que puedan utilizar ambos controladores. En el directorio `src/`, crea un nuevo directorio `Repository` y una nueva clase PHP en su interior llamada `StarshipRepository`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 2
namespace App\Repository;

class StarshipRepository
{
}
```

Al igual que cuando creamos nuestra clase `Starship`, esta nueva clase no tiene absolutamente nada que ver con Symfony. Es sólo una clase que hemos decidido crear para

organizar nuestro trabajo. Por lo tanto, a Symfony no le importa cómo se llama, dónde vive o qué aspecto tiene. Yo la llamé `StarshipRepository` y la puse en un directorio `Repository` porque es un nombre de programación común para una clase cuyo "trabajo" es obtener un tipo de datos, como los datos de la nave estelar.

Autocableado del nuevo servicio

Vale, antes de hacer nada aquí, vamos a ver si podemos utilizar esto dentro de un controlador. Y, ¡buenas noticias! Sólo con crear esta clase, ya está disponible para autocableado. Añade un argumento `StarshipRepository $repository` y, para asegurarte de que funciona, `dd($repository)`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 5
6 use App\Repository\StarshipRepository;
↕ // ... lines 7 - 11
12 class StarshipApiController extends AbstractController
13 {
↕ // ... line 14
15     public function getCollection(LoggerInterface $logger,
   StarshipRepository $repository): Response
16     {
17         $logger->info('Starship collection retrieved');
18         dd($repository);
↕ // ... lines 19 - 43
44     }
45 }
```

Muy bien, gira, vuelve a hacer clic en nuestra ruta, y... ya está. Qué guay! Symfony ha visto la sugerencia de tipo `StarshipRepository`, ha instanciado ese objeto y nos lo ha pasado. Borra el `dd()`... y movamos los datos de la nave estelar dentro. Cópialo... y crea una nueva función pública llamada, qué tal, `findAll()`. Dentro, `return`, y pégala.

src/Repository/StarshipRepository.php

```php
// ... lines 1 - 4
use App\Model\Starship;

class StarshipRepository
{
    public function findAll(): array
    {
        return [
            new Starship(
                1,
                'USS LeafyCruiser (NCC-0001)',
                'Garden',
                'Jean-Luc Pickles',
                'taken over by Q'
            ),
            new Starship(
                2,
                'USS Espresso (NCC-1234-C)',
                'Latte',
                'James T. Quick!',
                'repaired',
            ),
            new Starship(
                3,
                'USS Wanderlust (NCC-2024-W)',
                'Delta Tourist',
                'Kathryn Journeyway',
                'under construction',
            ),
        ];
    }
}
```

De vuelta en `StarshipApiController`, borra eso... y queda maravillosamente sencillo: `$starships = $repository->findAll()`.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 10
class StarshipApiController extends AbstractController
{
    #[Route('/api/starships')]
    public function getCollection(LoggerInterface $logger,
StarshipRepository $repository): Response
    {
        $logger->info('Starship collection retrieved');
        $starships = $repository->findAll();
// ... lines 18 - 19
    }
}
```

¡Listo! Cuando lo probamos, sigue funcionando... y ahora el código para obtener naves estelares está bien organizado en su propia clase y es reutilizable en toda nuestra aplicación.

Autocableado del Constructor

Con esta victoria en nuestro haber, vamos a hacer algo más difícil. ¿Qué pasaría si, desde dentro de `StarshipRepository`, necesitáramos acceder a otro servicio que nos ayudara a hacer nuestro trabajo? ¡No hay problema! ¡Podemos utilizar el autocableado! Intentemos autocablear de nuevo el servicio logger.

La única diferencia esta vez es que no vamos a añadir el argumento a `findAll()`. Te explicaré por qué en un minuto. En lugar de eso, añade un nuevo `public function __construct()` y realiza el autocableado allí: `private LoggerInterface $logger`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;

class StarshipRepository
{
    public function __construct(private LoggerInterface $logger)
    {
    }
// ... lines 13 - 41
}
```

A continuación, para utilizarlo, copia el código de nuestro controlador, bórralo, pégalo aquí y actualízalo a `$this->logger`.

src/Repository/StarshipRepository.php

```
// ... lines 1 - 5
use Psr\Log\LoggerInterface;

class StarshipRepository
{
    public function __construct(private LoggerInterface $logger)
    {
    }

    public function findAll(): array
    {
        $this->logger->info('Starship collection retrieved');
// ... lines 17 - 40
    }
}
```

¡Genial! En el controlador, podemos eliminar ese argumento porque ya no lo vamos a utilizar.

¡Hora de probar! ¡Actualiza! No hay error: buena señal. Para ver si se ha registrado algo, ve a `/_profiler`, haz clic en la petición superior, Registros, y... ¡ahí está!

Te explicaré por qué hemos añadido el argumento de servicio al constructor. Si queremos obtener un servicio -como el registrador, una conexión a una base de datos, lo que sea-, ésta es la forma correcta de utilizar el autocableado: añadir un método `__construct` dentro de otro servicio. El truco que vimos antes -en el que añadimos el argumento a un método normal- sí, eso es especial y sólo funciona para los métodos del controlador. Es una comodidad adicional que se añadió al sistema. Es una gran característica, pero la forma del constructor... así es como funciona realmente el autocableado.

Y esta forma "normal", funciona incluso en un controlador. Podrías añadir un método `__construct()` con un argumento autocableable y funcionaría perfectamente.

La cuestión es: si estás en un método controlador, claro, añade el argumento al método, ¡está bien! Sólo recuerda que es algo especial que sólo funciona aquí. En cualquier otra parte, autowire a través del constructor.

Utilizar el Servicio en otra Página

Celebremos nuestro nuevo servicio utilizándolo en la página principal. Abre `MainController`. Este `$starshipCount` codificado es tan de hace 30 minutos. Autocablea `StarshipRepository $starshipRepository`, luego di `$ships = $starshipRepository->findAll()` y cuéntalos con `count()`.

src/Controller/MainController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(StarshipRepository $starshipRepository): Response
    {
        $ships = $starshipRepository->findAll();
        $starshipCount = count($ships);
// ... lines 17 - 22
    }
}
```

Ya que estamos aquí, en lugar de esta matriz `$myShip` codificada, vamos a coger un objeto `Starship` al azar. Podemos hacerlo diciendo `$myShip` igual a `$ships[array_rand($ships)]`

src/Controller/MainController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(StarshipRepository $starshipRepository): Response
    {
        $ships = $starshipRepository->findAll();
        $starshipCount = count($ships);
        $myShip = $ships[array_rand($ships)];
// ... lines 18 - 22
    }
}
```

¡Vamos a probarlo! Busca en tu navegador y dirígete a la página de inicio. ¡Ya está! Vemos el barco que cambia aleatoriamente aquí abajo, y el número de barco correcto aquí arriba...

porque lo estamos multiplicando por 10 en la plantilla.

Imprimiendo objetos en Twig

¡Y acaba de ocurrir algo alucinante! Hace un momento, `myShip` era una matriz asociativa. Pero lo hemos cambiado para que sea un objeto Starship. Y aún así, el código de nuestra página siguió funcionando. Acabamos de ver accidentalmente un superpoder de Twig. Ve a `templates/main/homepage.html.twig` y desplázate hasta el final. Cuando dices `myShip.name`, Twig es realmente inteligente. Si `myShip` es una matriz asociativa, cogerá la clave `name`. Si `myShip` es un objeto, como lo es ahora, cogerá la propiedad `name`. Pero aún más, si miras `Starship`, la propiedad `name` es privada, por lo que no podemos acceder a ella directamente. Twig se da cuenta de ello. Mira la propiedad `name`, ve que es privada, pero también ve que hay una `getName()` pública. Así que llama a esa.

Todo lo que tenemos que decir es `myShip.name`... y Twig se encarga de los detalles de cómo obtenerlo, lo cual me encanta.

Vale, un último pequeño ajuste. En lugar de pasar el `starshipCount` a nuestra plantilla, podemos hacer el recuento dentro de Twig. Elimina esta variable y, en su lugar, pasa una variable `ships`.

src/Controller/MainController.php

```
↕ // ... lines 1 - 9
10 class MainController extends AbstractController
11 {
↕ // ... line 12
13     public function homepage(StarshipRepository $starshipRepository): Response
14     {
15         $ships = $starshipRepository->findAll();
16         $myShip = $ships[array_rand($ships)];
↕ // ... line 17
18         return $this->render('main/homepage.html.twig', [
19             'myShip' => $myShip,
20             'ships' => $ships,
21         ]);
22     }
23 }
```

En la plantilla, ahí lo tenemos, para el recuento, podemos decir `ships`, que es una matriz, y luego utilizar un filtro Twig: `|length`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
↕ // ... lines 6 - 9
10 <div>
11     Browse through {{ ships|length * 10 }} starships!
12
13     {% if ships|length > 2 %}
↕ // ... lines 14 - 17
18     {% endif %}
19 </div>
↕ // ... lines 20 - 42
43 {% endblock %}
```

Así queda bien. Hagamos lo mismo aquí abajo... y cambiémoslo a mayor que 2. Pruébalo. ¡Nuestro sitio sigue funcionando!

Lo siguiente: creemos más páginas y aprendamos a hacer rutas aún más inteligentes.

Chapter 11: Rutas más sofisticadas: Requisitos, comodines y más

Con toda la nueva organización del código, celebrémoslo creando otra ruta API para obtener un único `starship`. Empieza como siempre: crea un `public function` llamado, qué tal, `get()`. Incluiré el tipo de retorno opcional `Response`. Encima de éste añade el `#[Route]` con una URL de `/api/starships/`... hmm. Esta vez, la última parte de la URL tiene que ser dinámica: debe coincidir con `/api/starships/5` o `/api/starships/25`. ¿Cómo podemos hacerlo? ¿Cómo podemos hacer que una ruta coincida con un comodín?

La respuesta es añadiendo `{`, un nombre, el `}`.

El nombre dentro de esto puede ser cualquier cosa. No importa lo que sea, ahora esta ruta coincidirá con `/api/starships/*`. Pero sea cual sea el nombre que le pongas, ahora puedes tener un argumento con un nombre que coincida: `$id`.

A continuación, vuelca esto para asegurarte de que funciona.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 9
10 class StarshipApiController extends AbstractController
11 {
↕ // ... lines 12 - 19
20     #[Route('/api/starships/{id}')]
21     public function get($id): Response
22     {
23         dd($id);
24     }
25 }
```

Restringir el comodín a un número

¡Vale! Acércate a `/api/starships/2` y... ¡funciona!

En nuestra app, el ID será un número entero. Si pruebo con algo que no sea un número entero -como `/wharf` - la ruta sigue coincidiendo y llama a nuestro controlador. Y eso casi siempre

está bien. En una aplicación real, si consultáramos la base de datos con `WHERE ID = 'wharf'`, no se produciría un error: ¡simplemente no encontraría un barco coincidente! Y entonces podríamos lanzar una página 404, que pronto te enseñaré cómo hacer.

Pero a veces podemos querer restringir estos valores. Puede que queramos decir

> *"Sólo coincide con esta ruta si el comodín es un número entero."*

Para ello, dentro de la llave, después del nombre, añade un `<`, `>` y dentro, una expresión regular `\d+`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 9
10 class StarshipApiController extends AbstractController
11 {
↕ // ... lines 12 - 19
20     #[Route('/api/starships/{id<\d+>}')]
21     public function get(int $id): Response
22     {
23         dd($id);
24     }
25 }
```

Esto significa: coincide con un dígito de cualquier longitud. Con esta configuración, si actualizamos la URL `wharf`, obtenemos un error 404. Sencillamente, nuestra ruta no coincidió -ninguna ruta coincidió-, por lo que nunca se llamó a nuestro controlador. Pero si volvemos a `/2`, sigue funcionando.

Y como ventaja añadida, ahora que esto sólo coincide con dígitos, podemos añadir un tipo `int` al argumento. Ahora, en lugar de la cadena `2`, obtenemos el `integer` 2. Estos detalles no son superimportantes, pero quiero que sepas qué opciones tienes.

Restringir el método HTTP de la ruta

Algo habitual en las API es hacer que las rutas sólo coincidan con un determinado método HTTP, como `GET` o `POST`. Por ejemplo, si quieres obtener todas las naves estelares, los usuarios deben hacer una petición a `GET`... lo mismo si quieres obtener una sola nave. Si siguiéramos construyendo nuestra API y creáramos una ruta que pudiera utilizarse para crear

un nuevo `Starship`, la forma estándar de hacerlo sería utilizar la misma URL: `/api/starships` pero con una petición a `POST`.

Ahora mismo, esto no funcionaría. Cada vez que el usuario solicitara `/api/starships` -no importa si utiliza una petición `GET` o `POST`, coincidiría con esta primera ruta.

Por eso, es habitual en una API añadir una opción `methods` establecida en una matriz, con `GET` o `POST`. Haré lo mismo aquí abajo: `methods: ['GET']`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 9
10 class StarshipApiController extends AbstractController
11 {
12     #[Route('/api/starships', methods: ['GET'])]
13     public function getCollection(StarshipRepository $repository):
   Response
↕ // ... lines 14 - 19
20     #[Route('/api/starships/{id<\d+>}', methods: ['GET'])]
21     public function get(int $id): Response
↕ // ... lines 22 - 24
25 }
```

No puedo probarlo fácilmente en un navegador, pero si hiciéramos una petición POST a`/api/starships/2`, no coincidiría con nuestra ruta.

Pero podemos ver el cambio en nuestro terminal. Ejecuta:

```
php bin/console debug:router
```

¡Perfecto! La mayoría de las rutas coinciden con cualquier método... pero nuestras dos rutas API sólo coinciden si se realiza una petición `GET` a esa URL.

Poner un prefijo a cada URL de ruta

Vale, tengo otro truco de enrutamiento que enseñarte... ¡y es divertido! Todas las rutas de este controlador empiezan con la misma URL: `/api/starships`. Tener la URL completa en cada ruta está bien. Pero si queremos, podemos prefijar automáticamente la URL de cada ruta. Encima de la clase, añade un atributo `#[Route]` con `/api/starships`.

A diferencia de cuando lo ponemos encima de un método, esto no crea una ruta. Sólo dice: cada ruta de esta clase debe ir prefijada con esta URL. Así que para la primera ruta, elimina la ruta por completo. Y para la segunda, sólo necesitamos la parte del comodín.

src/Controller/StarshipApiController.php

```
// ... lines 1 - 9
#[Route('/api/starships')]
class StarshipApiController extends AbstractController
{
    #[Route('', methods: ['GET'])]
    public function getCollection(StarshipRepository $repository): Response
// ... lines 15 - 20
    #[Route('/{id<\d+>}', methods: ['GET'])]
    public function get(int $id): Response
// ... lines 23 - 25
}
```

Prueba de nuevo con `debug:router`... y observa estas URL:

```
php bin/console debug:router
```

¡No cambian!

Finalizando la nueva ruta API

Muy bien. Vamos a terminar nuestra ruta. Tenemos que encontrar el barco que coincida con este ID. Normalmente consultaríamos la base de datos: `select * from ship where id =` este ID. Nuestras naves están codificadas ahora mismo, pero aún podemos hacer algo que se parecerá más o menos a lo que será, una vez que tengamos una base de datos.

Ya tenemos un servicio - `StarshipRepository` - cuyo trabajo consiste en obtener datos sobre naves estelares. Démosle un nuevo superpoder: la capacidad de obtener un único `Starship` para un id. Añade `public function find()` con un argumento `int $id` que devolverá un `Starship` anulable. Por tanto, un `Starship` si encontramos uno para este id, si no `null`.

Ahora mismo, la forma más fácil de escribir esta lógica es hacer un bucle sobre `$this->findAll()` como `$starship`... luego si `$starship->getId() === $id`, devolver `$starship`. Cambiaré mi `uf` por `if`. Mucho mejor.

Y si no encontramos nada, al final, `return null`.

src/Repository/StarshipRepository.php

```
↕ // ... lines 1 - 7
8  class StarshipRepository
9  {
↕ // ... lines 10 - 42
43     public function find(int $id): ?Starship
44     {
45         foreach ($this->findAll() as $starship) {
46             if ($starship->getId() === $id) {
47                 return $starship;
48             }
49         }
50
51         return null;
52     }
53 }
```

Gracias a esto, nuestro controlador es muy sencillo. Primero, autocablea el repositorio añadiendo un argumento: `StarshipRepository` y llámalo `$repository`. Por cierto, el orden de los argumentos en un controlador no importa.

Después `$starship = $repository->find($id)`. Termina al final con `return $this->json($starship)`.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 10
11 class StarshipApiController extends AbstractController
12 {
↕ // ... lines 13 - 21
22     public function get(int $id, StarshipRepository $repository): Response
23     {
24         $starship = $repository->find($id);
25
26         return $this->json($starship);
27     }
28 }
```

Creo que ya estamos listos Actualiza. ¡Perfecto!

Activar una página 404

Pero prueba con un id que no exista en nuestra base de datos falsa - como `/200`. La palabra `null` no es... lo que queremos. En esta situación, deberíamos devolver una respuesta con un código de estado 404.

Para ello, vamos a seguir un patrón común: consulta un objeto y comprueba si devuelve algo. Si no devuelve nada, lanza un 404. Hazlo con throw `$this->createNotFoundException()`. Le pasaré un mensaje.

src/Controller/StarshipApiController.php

```
↕ // ... lines 1 - 10
11 class StarshipApiController extends AbstractController
12 {
↕ // ... lines 13 - 21
22     public function get(int $id, StarshipRepository $repository): Response
23     {
24         $starship = $repository->find($id);
25 
26         if (!$starship) {
27             throw $this->createNotFoundException('Starship not found');
28         }
29 
30         return $this->json($starship);
31     }
32 }
```

Fíjate en la palabra clave `throw`: estamos lanzando una excepción especial que desencadena un 404. Eso está bien porque, en cuanto llegue a esta línea, no se ejecutará nada de lo que venga después.

¡Pruébalo! ¡Sí! ¡Una respuesta 404! El mensaje - "Nave no encontrada"- sólo se muestra a los desarrolladores en modo dev. En producción, se devolvería una página -o JSON- totalmente diferente. Puedes consultar la documentación para obtener más información sobre las páginas de error de producción.

A continuación: vamos a construir la versión HTML de esta página, una página que muestra detalles sobre una única nave estelar. Luego aprenderemos a enlazar entre páginas utilizando el nombre de la ruta.

Chapter 12: Generar URLs

Vamos a crear una "página de presentación" de barcos: una página que muestre los detalles de un solo barco. La página de inicio vive en `MainController`. Y así podríamos añadir otra ruta y método aquí. Pero a medida que mi sitio crezca, probablemente tendré varias páginas relacionadas con naves estelares: quizá para editarlas y eliminarlas. Así que, en lugar de eso, en el directorio `Controller/`, crea una nueva clase. Llámala `StarshipController`, y, como de costumbre, extiende `AbstractController`.

Crear la página Mostrar

Dentro, ¡manos a la obra! Añade un `public function` llamado `show()`, yo añadiré el tipo de retorno `Response`, luego la ruta, con `/starships/` y un comodín llamado `{id}`. Y de nuevo, es opcional, pero seré extravagante y añadiré el `\d+` para que el comodín sólo coincida con un número.

Ahora, como tenemos un comodín `{id}`, se nos permite tener un argumento `$id` aquí abajo. `dd($id)` para ver cómo vamos hasta ahora.

src/Controller/StarshipController.php

```
// ... lines 1 - 2
namespace App\Controller;

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

class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}')]
    public function show(int $id): Response
    {
        dd($id);
    }
}
```

Pruébalo. Dirígete a `/starships/2`. ¡Estupendo!

Ahora vamos a hacer algo familiar: tomar este `$id` y consultar nuestra base de datos imaginaria en busca del `Starship` coincidente. La clave para hacerlo es nuestro servicio `StarshipRepository` y su útil método `find()`.

En el controlador, añade un argumento `StarshipRepository $repository`... y luego di que `$ship` es igual a `$repository->find($id)`. Y si no es `$ship`, activa una página 404 con los lanzamientos `$this->createNotFoundException()` y `starship not found`.

¡Genial! En la parte inferior, en lugar de devolver JSON, renderiza una plantilla: devuelve `$this->render()` y sigue la convención de nomenclatura estándar para plantillas: `starship/show.html.twig`. Pasa esta variable: `$ship`.

src/Controller/StarshipController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}')]
    public function show(int $id, StarshipRepository $repository): Response
    {
        $ship = $repository->find($id);
        if (!$ship) {
            throw $this->createNotFoundException('Starship not found');
        }

        return $this->render('starship/show.html.twig', [
            'ship' => $ship,
        ]);
    }
}
```

Crear la plantilla

Controlador, ¡comprobado! A continuación, en el directorio `templates/`, podríamos crear un directorio `starship/` y `show.html.twig` dentro. Pero quiero mostrarte un atajo del plugin Symfony PhpStorm. Haz clic en el nombre de la plantilla, pulsa Alt+Enter y... ¡fíjate! En la parte superior pone "Twig: Crear plantilla". Confirma la ruta y ¡boom! ¡Ya tenemos nuestra nueva plantilla! Está... escondida por aquí. Ahí está: `starship/show.html.twig`.

Prácticamente todas las plantillas empiezan igual: `{% extend 'base.html.twig' %}`... ¡luego anula algunos bloques! Anula `title`... y esta vez, utiliza la variable `ship`: `ship.name`. Termina con `endblock`.

Y para el contenido principal, añade el bloque `body`... `endblock` y pon un `h1` dentro. Vuelve a imprimir `ship.name` y... Pegaré una tabla con algo de información.

src/Controller/StarshipController.php

```
// ... lines 1 - 4
use App\Repository\StarshipRepository;
// ... lines 6 - 9
class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}')]
    public function show(int $id, StarshipRepository $repository): Response
    {
        $ship = $repository->find($id);
        if (!$ship) {
            throw $this->createNotFoundException('Starship not found');
        }

        return $this->render('starship/show.html.twig', [
            'ship' => $ship,
        ]);
    }
}
```

Aquí no hay nada especial: sólo estamos imprimiendo datos básicos del barco.

Cuando probamos la página... ¡está viva!

Enlazar entre páginas

Siguiente pregunta: desde la página de inicio, ¿cómo podríamos añadir un enlace a la nueva página de presentación de barcos? La opción más obvia es codificar la URL, como `/starships/` y luego el id. Pero hay una forma mejor. En lugar de eso, vamos a decirle a Symfony:

> *"Oye, quiero generar una URL para esta ruta."*

La ventaja es que si más adelante decidimos cambiar la URL de esta ruta, todos los enlaces a ella se actualizarán automáticamente.

Déjame que te lo muestre. Busca tu terminal y ejecuta:

```
php bin/console debug:router
```

Aún no lo he mencionado, pero cada ruta tiene un nombre interno. Ahora mismo, están siendo autogeneradas por Symfony, lo cual está bien. Pero en cuanto quieras generar una URL a una ruta, debemos tomar el control de ese nombre para asegurarnos de que nunca cambie.

Busca la ruta show page y añade una clave `name`. Yo utilizaré `app_starship_show`.

src/Controller/StarshipController.php

```
// ... lines 1 - 9
class StarshipController extends AbstractController
{
    #[Route('/starships/{id<\d+>}', name: 'app_starship_show')]
    public function show(int $id, StarshipRepository $repository):
Response
// ... lines 14 - 23
}
```

El nombre podría ser cualquier cosa, pero ésta es la convención que yo sigo: `app` porque es una ruta que estoy creando en mi aplicación, y luego el nombre de la clase del controlador y el nombre del método.

Nombrar una ruta no cambia su funcionamiento. Pero sí nos permite generar una URL hacia ella. Abre `templates/main/homepage.html.twig`. Aquí abajo, convierte el nombre de la ruta en un enlace. Lo pondré en varias líneas y añadiré una etiqueta `a` con `href=""`. Para generar la URL, diré `{{ path() }}` y le pasaré el nombre de la ruta. Pondré la etiqueta de cierre en el otro lado.

Si nos detenemos ahora, esto no funcionará del todo. En la página de inicio:

> *"Faltan algunos parámetros obligatorios - `id` - para generar una URL para la ruta `app_starship_show`."*

¡Eso tiene sentido! Le estamos diciendo a Symfony:

> *“¡Hola! Quiero generar una URL para esta ruta.”*

Symfony entonces responde:

> *“Genial... excepto que esta ruta tiene un comodín. Así que... ¿qué quieres quieres que ponga en la URL para la parte `id`?”*

Cuando hay un comodín en la ruta, tenemos que añadir un segundo argumento a `path()` con `{}`. Esta es la sintaxis de matriz asociativa de Twig. Es exactamente igual que JavaScript: es una lista de pares clave-valor. Pasa `id` ajustado a `myShip.id`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
↕ // ... lines 6 - 20
21 <div>
↕ // ... lines 22 - 23
24     <table>
25         <tr>
26             <th>Name</th>
27             <td>
28                 <a href="{{ path('app_starship_show', {
29                     id: myShip.id
30                 }) }}">{{ myShip.name }}</a>
31             </td>
32         </tr>
↕ // ... lines 33 - 44
45     </table>
46 </div>
47 {% endblock %}
```

Y ahora... ¡ya está! Mira esa URL: `/starships/3`.

Muy bien, nuestro sitio sigue siendo feo. Es hora de empezar a arreglarlo incorporando Tailwind CSS y aprendiendo sobre el componente AssetMapper de Symfony.

Chapter 13: CSS y JavaScript con Asset Mapper

¿Qué pasa con las imágenes, CSS y JavaScript? ¿Cómo funciona eso en Symfony?

Las cosas públicas son... Público

En primer lugar, el directorio `public/` se conoce como la raíz de tu documento. Cualquier cosa dentro de `public/` es accesible para tu usuario final. Todo lo que no esté en `public/` no es accesible, ¡lo cual es genial! Ninguno de nuestros archivos fuente de alto secreto puede ser descargado por nuestros usuarios.

Así que si quieres crear un archivo CSS o un archivo de imagen o cualquier otra cosa, la vida puede ser tan simple como ponerlo en `public/`. Ahora puedo ir a `/foo.txt`... y vemos el archivo.

Hola Mapeador de Activos

Sin embargo, Symfony tiene un gran componente llamado Asset Mapper que nos permite hacer efectivamente lo mismo... pero con algunas características importantes y extra. Tenemos unos cuantos tutoriales que profundizan en este tema: uno sobre el Mapeador de Activos específicamente y otro sobre cómo construir cosas con el Mapeador de Activos llamado LAST Stack. Échales un vistazo para profundizar.

¡Pero vamos a sumergirnos en las amistosas aguas del Mapeador de Activos! Confirma todos tus cambios -yo ya lo he hecho- e instálalo con:

```
composer require symfony/asset-mapper
```

Esta receta hace varios cambios... y recorreremos cada uno poco a poco, ya que son importantes.

Pero ya, si nos desplazamos y actualizamos, ¡nuestro fondo es azul! Inspecciona Element en tu navegador y ve a la consola. ¡También tenemos un registro de consola!

> *"Este log viene de `assets/app.js`. Bienvenido al mapeador de activos."*

¡Muchas gracias!

Los 2 superpoderes de Asset Mapper

Asset Mapper tiene dos grandes superpoderes. El primero es que nos ayuda a cargar CSS y JavaScript. La receta nos ha proporcionado un nuevo directorio `assets/` con un archivo `app.js` y otro `styles/app.css`. Como hemos visto, el registro de la consola procede de `app.js`.

assets/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * This file will be included onto the page via the importmap() Twig function,
 * which should already be in your base.html.twig.
 */
import './styles/app.css';

console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉');
```

Así que este archivo se está cargando. Al parecer, también está incluyendo `app.css`, que es lo que nos da ese fondo azul.

assets/styles/app.css

```
body {
    background-color: skyblue;
}
```

Más adelante hablaremos más sobre cómo se cargan estos archivos y cómo funciona todo esto. Pero por ahora, basta con saber que `app.js` y `app.css` están incluidos en la página.

El segundo gran superpoder de Asset Mapper es un poco más sencillo. La receta ha creado un archivo `config/packages/asset_mapper.yaml`. No hay mucho aquí:

config/packages/asset_mapper.yaml

```yaml
framework:
    asset_mapper:
        # The paths to make available to the asset mapper.
        paths:
            - assets/
```

sólo `paths` apuntando a nuestro directorio `assets/`. Pero gracias a esta línea, cualquier archivo que pongamos en el directorio `assets/` estará disponible públicamente. Es como si el directorio `assets/` viviera físicamente dentro de `public/`. Esto es útil porque, sobre la marcha, Asset Mapper añade el versionado de activos: un importante concepto de frontend que veremos dentro de un minuto.

Listado de activos y ruta lógica

Pero antes, dirígete a tu terminal y ejecuta otro nuevo comando `debug`:

```
php bin/console debug:asset
```

Esto muestra todos los activos expuestos públicamente a través del Mapeador de Activos. Ahora mismo son sólo dos: `app.css` y `app.js`.

Si descargas el código del curso de esta página y lo descomprimes, encontrarás un directorio `tutorial/` con un subdirectorio `images/`. Cortaré esto... y luego lo pegaré en `assets/`.

Así que ahora tenemos un directorio `assets/images/` con 5 archivos dentro. Y, por cierto, puedes organizar el directorio `assets/` como quieras.

Pero ahora, vuelve atrás y ejecuta de nuevo `debug:asset`:

```
php bin/console debug:asset
```

¡Los nuevos archivos están ahí!

Representación de una imagen

A la izquierda, ¿ves esta "ruta lógica"? Es la ruta que utilizaremos para hacer referencia a ese archivo en Asset Mapper.

Te lo mostraré: vamos a renderizar una etiqueta `img` en el logotipo. Copia la ruta lógica `starshop-logo.png`. Luego dirígete a `templates/base.html.twig`. Justo encima del bloque del cuerpo -para que no quede anulado por el contenido de nuestra página- añade una etiqueta `<img>` con `src=""`. En lugar de intentar codificar una ruta, di `{{` y utiliza una nueva función Twig llamada `asset()`. Pásale la ruta lógica.

Ya está Vale, añadiré un atributo `alt`... para ser un buen ciudadano de la web.

templates/base.html.twig

```
// ... line 1
<html>
// ... lines 3 - 13
    <body>
        <img src="{{ asset('images/starshop-logo.png') }}" alt="Starshop Logo">
        {% block body %}{% endblock %}
    </body>
</html>
```

Probemos esto. Actualiza y... ¡estalla!

> *"¿Has olvidado ejecutar `composer require symfony/asset`. Función desconocida "activo"."*

Recuerda: nuestra aplicación empieza siendo pequeña. Y luego, a medida que necesitamos más funciones, instalamos más componentes Symfony. Y a menudo, si intentas utilizar una función de un componente que no está instalado, te lo dirá. La función Twig `asset()` proviene de otro componente diminuto llamado `symfony/asset`. Todo lo que tenemos que hacer es seguir el consejo. Copia el comando `composer require`, pasa a tu terminal y ejecútalo:

```
composer require symfony/asset
```

Cuando termine, muévete y actualiza. ¡Ahí está nuestro logotipo!

Versionado automático de activos

¿La parte más interesante? Ver el código fuente de la página y comprobar la URL: `/assets/images/starshop-logo-` y luego una larga cadena de letras y números, `.png`. Esta cadena se llama hash de la versión y se genera en función del contenido del archivo. Eso significa que si más adelante actualizamos nuestro logotipo, este hash cambiará automáticamente.

Esto es superimportante. A los navegadores les gusta almacenar en caché las imágenes, el JavaScript y los archivos CSS, lo que está muy bien: ayuda al rendimiento. Pero cuando cambiamos esos archivos, queremos que nuestros usuarios descarguen la nueva versión: no que sigan utilizando la versión obsoleta, almacenada en caché.

Pero como el nombre del archivo cambiará cuando actualicemos la imagen, ¡el navegador va a utilizar automáticamente el nuevo! Esto es así:

- El usuario va a nuestro sitio y descarga `logo-abc123.png`. Su navegador lo almacena en caché.
- En la siguiente visita, su navegador ve la etiqueta `img` para `logo-abc123.png`, encuentra el archivo en su caché y lo utiliza.
- Entonces llegamos nosotros, actualizamos ese archivo y lo desplegamos.
- La próxima vez que el usuario visite nuestro sitio, la etiqueta `img` apuntará a un nombre de archivo diferente: `logo-def456.png`. Y como el navegador no tiene ese archivo en su caché, lo descarga nuevo.

Se trata de un pequeño detalle, pero también es increíblemente importante para asegurarnos de que nuestros usuarios utilizan siempre los archivos más recientes. ¿Y lo mejor? Simplemente funciona. Ahora que te lo he explicado, no tendrás que volver a pensar en esto.

Ok equipo, vamos a instalar y empezar a usar Tailwind CSS a continuación.

Chapter 14: Tailwind CSS

¿Qué pasa con el CSS? Eres libre de añadir el CSS que quieras a `app/styles/app.css`. Ese archivo ya está cargado en la página.

¿Quieres utilizar CSS de Bootstrap? Consulta la documentación de Asset Mapper sobre cómo hacerlo. O, si quieres usar Sass, hay un symfonycasts/sass-bundle que te lo pone fácil. No obstante, te recomiendo que no te lances a usar Sass demasiado rápido, ya que muchas de las funciones por las que Sass es famoso pueden hacerse ahora en CSS nativo, como las variables CSS e incluso el anidamiento CSS.

Hola Tailwind

¿Cuál es mi elección personal para un framework CSS? Tailwind. Y parte de la razón es que Tailwind es increíblemente popular. Así que si buscas recursos o componentes preconstruidos, vas a tener mucha suerte si utilizas Tailwind.

Pero Tailwind es un poco extraño en un sentido: no es simplemente un gran archivo CSS que pones en tu página. En su lugar, tiene un proceso de construcción que escanea tu código en busca de todas las clases Tailwind que estés utilizando. Luego vuelca un archivo CSS final que sólo contiene el código que necesitas.

En el mundo Symfony, si quieres utilizar Tailwind, hay un bundle que lo hace realmente fácil. Gira tu terminal e instala un nuevo paquete: `composer require symfonycasts` - hey los conozco - `tailwind-bundle`:

```
composer require symfonycasts/tailwind-bundle
```

Para este paquete, la receta no hace nada más que activar el nuevo bundle. Para poner en marcha Tailwind, una vez en tu proyecto, ejecuta:

```
php bin/console tailwind:init
```

Esto hace tres cosas. En primer lugar, descarga un binario de Tailwind en segundo plano, algo en lo que nunca tendrás que pensar. En segundo lugar, crea un archivo `tailwind.config.js` en la raíz de nuestro proyecto. Esto indica a Tailwind dónde tiene que buscar en nuestro proyecto las clases CSS de Tailwind. Y tercero, actualiza nuestro `app.css` para añadir estas tres líneas. Éstas serán sustituidas por el código real de Tailwind en segundo plano por el binario.

Ejecutar Tailwind

Por último, hay que compilar Tailwind, así que tenemos que ejecutar un comando para hacerlo:

```
php bin/console tailwind:build -w
```

Esto escanea nuestras plantillas y genera el archivo CSS final en segundo plano. El `-w` lo pone en modo "vigilar": en lugar de construir una vez y salir, vigila nuestras plantillas en busca de cambios. Cuando detecte alguna actualización, reconstruirá automáticamente el archivo CSS. Lo veremos en un minuto.

Pero ya deberíamos ver una diferencia. Vamos a la página de inicio. ¿Lo has visto? El código base de Tailwind ha hecho un reinicio. Por ejemplo, ¡nuestro `h1` es ahora diminuto!

Ver Tailwind en acción

Probemos esto de verdad. Abre `templates/main/homepage.html.twig`. Encima de `h1`, hazlo más grande añadiendo una clase: `text-2xl`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
<h1 class="text-2xl">
    Starshop: your monopoly-busting option for Starship parts!
</h1>
// ... lines 9 - 46
{% endblock %}
```

En cuanto guardemos eso, podrás ver que tailwind se dio cuenta de nuestro cambio y reconstruyó el CSS. Y cuando actualizamos, ¡se hizo más grande!

Nuestro archivo fuente `app.css` sigue siendo super sencillo: sólo esas pocas líneas que vimos antes. Pero mira el código fuente de la página y abre el `app.css` que se está enviando a nuestros usuarios. ¡Es la versión construida de Tailwind! Entre bastidores, existe cierta magia que sustituye esas tres líneas de Tailwind por el código CSS real de Tailwind.

Ejecutar automáticamente Tailwind con el binario symfony

Y... ¡eso es todo! Simplemente funciona. Aunque hay una forma más fácil y automática de ejecutar Tailwind. Pulsa Ctrl+C en el comando Tailwind para detenerlo. A continuación, en la raíz de nuestro proyecto, crea un archivo llamado `.symfony.local.yaml`. Se trata de un archivo de configuración para el servidor web binario `symfony` que estamos utilizando. Dentro, añade `workers`, `tailwind`, y luego `cmd` configurados en una matriz con cada parte de un comando: `symfony`, `console`, `tailwind`, `build`, `--watch`, o podrías utilizar `-w`: es lo mismo.

Aún no he hablado de ello, pero en lugar de ejecutar `php bin/console`, también podemos ejecutar `symfony console` seguido de cualquier comando para obtener el mismo resultado. Hablaremos de por qué te conviene hacerlo en un futuro tutorial. Pero por ahora, considera que `bin/console` y `symfony console` son lo mismo.

Además, al añadir esta clave `workers`, significa que en lugar de que tengamos que ejecutar el comando manualmente, cuando iniciemos el servidor web `symfony`, éste lo ejecutará por nosotros en segundo plano.

Observa. En tu primera pestaña, pulsa Ctrl+C para detener el servidor web... luego vuelve a ejecutar

```
symfony serve
```

para que vea el nuevo archivo de configuración. Mira: ¡ahí está! ¡Está ejecutando el comando tailwind en segundo plano!

Podemos aprovecharnos de esto inmediatamente. En `homepage.html.twig`, cambia esto a `text-4xl`, gira y... ¡funciona! Ya ni siquiera tenemos que pensar en el comando `tailwind:build`.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
<h1 class="text-4xl">
    Starshop: your monopoly-busting option for Starship parts!
</h1>
// ... lines 9 - 46
{% endblock %}
```

Y como estilizaremos con Tailwind, elimina el fondo azul.

Copiar en plantillas estilizadas

Vale, este tutorial no trata sobre Tailwind ni sobre cómo diseñar un sitio web. Créeme, no quieres que Ryan dirija la carga del diseño web. Pero sí quiero tener un sitio bonito... y también es importante pasar por el proceso de trabajar con un diseñador.

Así que imaginemos que otra persona ha creado un diseño para nuestro sitio. E incluso nos han dado algo de HTML con clases de Tailwind para ese diseño. Si descargas el código del curso, en un directorio de `tutorial/templates/`, tenemos 3 plantillas. Uno a uno, voy a copiar cada archivo y pegarlo sobre el original. No te preocupes, veremos lo que ocurre en cada uno de estos archivos.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 128%22><text y=%221.2em%22 font-size=%2296%22>⚫</text></svg>">
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
{% block importmap %}{{ importmap('app') }}{% endblock %}
        {% endblock %}
    </head>
    <body class="text-white" style="background: radial-gradient(102.21% 102.21% at 50% 28.75%, #00121C 42.62%, #013954 100%);">
        <div class="flex flex-col justify-between min-h-screen relative">
            <div>
                <header class="h-[114px] shrink-0 flex flex-col sm:flex-row items-center sm:justify-between py-4 sm:py-0 px-6 border-b border-white/20 shadow-md">
                    <a href="#">
                        <img class="h-[42px]" src="{{ asset('images/starshop-logo.png') }}" alt="starshop logo">
                    </a>
                    <nav class="flex space-x-4 font-semibold">
                        <a class="hover:text-amber-400 pt-2" href="#">
                            Home
                        </a>
                        <a class="hover:text-amber-400  pt-2" href="#">
                            About
                        </a>
                        <a class="hover:text-amber-400 pt-2" href="#">
                            Contact
                        </a>
                        <a class="rounded-[60px] py-2 px-5 bg-white/10 hover:bg-white/20" href="#">
                            Get Started
                        </a>
                    </nav>
                </header>
                {% block body %}{% endblock %}
            </div>
            <div class="p-5 bg-white/5 mt-3 text-center">
                Made with ❤️ by <a class="text-[#0086C4]" href="https://symfonycasts.com">SymfonyCasts</a>
```

```
            </div>
        </div>
    </body>
</html>
```

Haz `homepage.html.twig`...

templates/main/homepage.html.twig

```
{% extends 'base.html.twig' %}

{% block title %}Starshop: Beam up some parts!{% endblock %}

{% block body %}
    <main class="flex flex-col lg:flex-row">
        <aside
            class="pb-8 lg:pb-0 lg:w-[411px] shrink-0 lg:block lg:min-h-screen text-white transition-all overflow-hidden px-8 border-b lg:border-b-0 lg:border-r border-white/20"
        >
            <div class="flex justify-between mt-11 mb-7">
                <h2 class="text-[32px] font-semibold">My Ship Status</h2>
                <button>
                    <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
                </button>
            </div>

            <div>
                <div class="flex flex-col space-y-1.5">
                    <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center" style="background: rgba(255, 184, 0, .1);">
                        <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                        <p class="uppercase text-xs">in progress</p>
                    </div>
                    <h3 class="tracking-tight text-[22px] font-semibold">
                        <a class="hover:underline" href="{{ path('app_starship_show', {
                            id: myShip.id
                        }) }}">{{ myShip.name }}</a>
                    </h3>
                </div>
                <div class="flex mt-4">
                    <div class="border-r border-white/20 pr-8">
                        <p class="text-slate-400 text-xs">Captain</p>
                        <p class="text-xl">{{ myShip.captain }}</p>
                    </div>

```

```
                    <div class="pl-8">
                        <p class="text-slate-400 text-xs">Class</p>
                        <p class="text-xl">{{ myShip.class }}</p>
                    </div>
                </div>
            </div>
        </aside>

        <div class="px-12 pt-10 w-full">
            <h1 class="text-4xl font-semibold mb-8">
                Ship Repair Queue
            </h1>

            <div class="space-y-5">
                <!-- start ship item -->
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11
flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-
[1174px]:justify-start">
                            <img class="h-[83px] w-[84px]"
src="/images/status-in-progress.png" alt="Status: in progress">
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex
justify-center w-32 items-center bg-amber-400/10">
                                    <div class="rounded-full h-2 w-2 bg-
amber-400 blur-[1px] mr-2"></div>
                                    <p class="uppercase text-xs text-
nowrap">in progress</p>
                                </div>
                                <h4 class="text-[22px] pt-1 font-
semibold">
                                    <a
                                        class="hover:text-slate-200"
                                        href="#"
                                    >USS LeafyCruiser</a>
                                </h4>
                            </div>
                        </div>
                        <div class="flex justify-center min-
[1174px]:justify-start mt-2 min-[1174px]:mt-0 shrink-0">
                            <div class="border-r border-white/20 pr-8">
                                <p class="text-slate-400 text-
xs">Captain</p>
                                <p class="text-xl">Jean-Luc Pickles</p>
                            </div>

                            <div class="pl-8 w-[100px]">
```

```
                                <p class="text-slate-400 text-xs">Class</p>
                                <p class="text-xl">Garden</p>
                            </div>
                        </div>
                    </div>
                    <!-- end ship item -->
                </div>

                <p class="text-lg mt-5 text-center md:text-left">
                    Looking for your next galactic ride?
                    <a href="#" class="underline font-semibold">Browse the {{ ships|length * 10 }} starships for sale!</a>
                </p>
            </div>
        </main>
{% endblock %}
```

y finalmente `show.html.twig`.

templates/starship/show.html.twig

```twig
{% extends 'base.html.twig' %}

{% block title %}{{ ship.name }}{% endblock %}

{% block body %}
<div class="my-4 px-8">
    <a class="bg-white hover:bg-gray-200 rounded-xl p-2 text-black" href="#">
        <svg class="inline text-black" xmlns="http://www.w3.org/2000/svg" height="16" width="14" viewBox="0 0 448 512"><!--!Font Awesome Free 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license/free Copyright 2024 Fonticons, Inc.--><path fill="#000" d="M9.4 233.4c-12.5 12.5-12.5 32.8 0 45.3l160 160c12.5 12.5 32.8 12.5 45.3 0s12.5-32.8 0-45.3L109.2 288 416 288c17.7 0 32-14.3 32-32s-14.3-32-32-32l-306.7 0L214.6 118.6c12.5-12.5 12.5-32.8 0-45.3s-32.8-12.5-45.3 0l-160 160z"/></svg>
        Back
    </a>
</div>
<div class="md:flex justify-center space-x-3 mt-5 px-4 lg:px-8">
    <div class="flex justify-center">
        <img class="max-h-[300px] md:max-h-[500px]" src="{{ asset('images/purple-rocket.png') }}" alt="purple ship launching">
    </div>
    <div class="space-y-5">
        <div class="mt-8 max-w-xl mx-auto">
            <div class="px-8 pt-8">
                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
                    <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                    <p class="uppercase text-xs">{{ ship.status }}</p>
                </div>

                <h1 class="text-[32px] font-semibold border-b border-white/10 pb-5 mb-5">
                    {{ ship.name }}
                </h1>
                <h4 class="text-xs text-slate-300 font-semibold mt-2 uppercase">Spaceship Captain</h4>
                <p class="text-[22px] font-semibold">{{ ship.captain }}</p>

                <h4 class="text-xs text-slate-300 font-semibold mt-2 uppercase">Class</h4>
                <p class="text-[22px] font-semibold">{{ ship.class }}</p>

```

```
33                 <h4 class="text-xs text-slate-300 font-semibold mt-2
   uppercase">Ship Status</h4>
34                 <p class="text-[22px] font-semibold">30,000 lys to next
   service</p>
35             </div>
36         </div>
37     </div>
38 </div>
39 {% endblock %}
```

Tip

Si copias los archivos (en lugar del contenido de los archivos), puede que el sistema de caché de Symfony no note el cambio y no veas el nuevo diseño. Si eso ocurre, borra la caché ejecutando `php bin/console cache:clear`.

Voy a borrar por completo el directorio `tutorial/` para no confundirme y editar las plantillas equivocadas.

Vale, ¡vamos a ver qué ha hecho esto! Actualizar. ¡Tiene un aspecto precioso! Me encanta trabajar dentro de un diseño bonito. Pero... algunas partes están rotas. En `homepage.html.twig`, ésta es nuestra cola de reparación de barcos... que queda muy bien... ¡pero no hay código Twig! El estado está codificado, el nombre está codificado y no hay bucle.

templates/main/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
    <main class="flex flex-col lg:flex-row">
// ... lines 7 - 42
        <div class="px-12 pt-10 w-full">
            <h1 class="text-4xl font-semibold mb-8">
                Ship Repair Queue
            </h1>

            <div class="space-y-5">
                <!-- start ship item -->
                    <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
                        <div class="flex justify-center min-[1174px]:justify-start">
                            <img class="h-[83px] w-[84px]" src="/images/status-in-progress.png" alt="Status: in progress">
                            <div class="ml-5">
                                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
                                    <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                                    <p class="uppercase text-xs text-nowrap">in progress</p>
                                </div>
                                <h4 class="text-[22px] pt-1 font-semibold">
                                    <a
                                        class="hover:text-slate-200"
                                        href="#"
                                    >USS LeafyCruiser</a>
                                </h4>
                            </div>
                        </div>
                        <div class="flex justify-center min-[1174px]:justify-start mt-2 min-[1174px]:mt-0 shrink-0">
                            <div class="border-r border-white/20 pr-8">
                                <p class="text-slate-400 text-xs">Captain</p>
                                <p class="text-xl">Jean-Luc Pickles</p>
                            </div>

                            <div class="pl-8 w-[100px]">
                                <p class="text-slate-400 text-xs">Class</p>
                                <p class="text-xl">Garden</p>
                            </div>
```

```
76                          </div>
77                      </div>
78                  <!-- end ship item -->
79              </div>
↕ // ... lines 80 - 84
85          </div>
86      </main>
87  {% endblock %}
```

A continuación: tomemos nuestro nuevo diseño y hagámoslo dinámico. También aprenderemos a organizar las cosas en parciales de plantilla e introduciremos un enum PHP, que son divertidos.

Chapter 15: Twig Parciales y para bucles

Acabamos de renovar el diseño de nuestro sitio... lo que significa que hemos actualizado nuestras plantillas para incluir elementos HTML con un montón de clases de Tailwind. ¿El resultado? Un sitio agradable a la vista.

En algunas partes de las plantillas, las cosas siguen siendo dinámicas: tenemos código Twig para imprimir el capitán y la clase. Pero en otras partes, todo está codificado. Y... esto es bastante típico: un desarrollador frontend puede codificar el sitio en HTML y Tailwind... pero dejarte a ti que lo hagas dinámico y le des vida.

Organizar en una Plantilla Parcial

En la parte superior de `homepage.html.twig`, este largo elemento `<aside>` es la barra lateral. Está bien que este código viva en `homepage.html.twig`... ¡pero ocupa mucho espacio! ¿Y si queremos reutilizar esta barra lateral en otra página?

Una gran característica de Twig es la posibilidad de tomar "trozos" de HTML y aislarlos en sus propias plantillas para que puedas reutilizarlos. Se llaman parciales de plantilla... ya que contienen el código de sólo una parte de la página.

Copia este código, y en el directorio `main/` -aunque esto puede ir en cualquier sitio- añade un nuevo archivo llamado `_shipStatusAside.html.twig`. Pega dentro.

templates/main/_shipStatusAside.html.twig

```
<aside
    class="pb-8 lg:pb-0 lg:w-[411px] shrink-0 lg:block lg:min-h-screen text-white transition-all overflow-hidden px-8 border-b lg:border-b-0 lg:border-r border-white/20"
>
    <div class="flex justify-between mt-11 mb-7">
        <h2 class="text-[32px] font-semibold">My Ship Status</h2>
        <button>
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
        </button>
    </div>

    <div>
        <div class="flex flex-col space-y-1.5">
            <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center" style="background: rgba(255, 184, 0, .1);">
                <div class="rounded-full h-2 w-2 bg-amber-400 blur-[1px] mr-2"></div>
                <p class="uppercase text-xs">in progress</p>
            </div>
            <h3 class="tracking-tight text-[22px] font-semibold">
                <a class="hover:underline" href="{{ path('app_starship_show', {
                    id: myShip.id
                }) }}">{{ myShip.name }}</a>
            </h3>
        </div>
        <div class="flex mt-4">
            <div class="border-r border-white/20 pr-8">
                <p class="text-slate-400 text-xs">Captain</p>
                <p class="text-xl">{{ myShip.captain }}</p>
            </div>

            <div class="pl-8">
                <p class="text-slate-400 text-xs">Class</p>
                <p class="text-xl">{{ myShip.class }}</p>
            </div>
        </div>
    </div>
```

```
35 </aside>
```

De vuelta en `homepage.html.twig`, borra eso, y luego inclúyelo con `{{` - para que diga algo de sintaxis - `include()` y el nombre de la plantilla:`main/_shipStatusAside.html.twig`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
7         {{ include('main/_shipStatusAside.html.twig') }}
↕ // ... lines 8 - 51
52     </main>
53 {% endblock %}
```

¡Pruébalo! Y... ¡no hay cambios! La declaración `include()` es sencilla:

> *"Renderiza esta plantilla y dale las mismas variables que yo tengo"*

Si te preguntas por qué he antepuesto un guión bajo a la plantilla... ¡no hay motivo! Es sólo una convención que me ayuda a saber que esta plantilla contiene sólo una parte de la página.

Haciendo un bucle sobre las naves en Twig

En la plantilla de la página de inicio, podemos centrarnos en la lista de naves de abajo, que es esta zona. Ahora mismo, sólo hay una nave... y está codificada. Nuestra intención es listar todas las naves que estamos reparando actualmente. Y ya tenemos una variable `ships` que estamos utilizando en la parte inferior: es una matriz de objetos `Starship`.

Así que, por primera vez en Twig, ¡tenemos que hacer un bucle sobre una matriz! Para ello, eliminaré este comentario, y diré `{%` -así que la etiqueta hacer algo- y luego`for ship in ships`. `ships` es la variable de matriz que ya tenemos y `ship` es el nuevo nombre de la variable en el bucle que representa un único objeto`Starship`. En la parte inferior, añade `{% endfor %}`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
↕ // ... lines 16 - 43
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Y ya... cuando lo probamos, ¡obtenemos tres naves codificadas! ¡Eso es una mejora!

A continuación: es hora de un giro argumental que nos llevará a crear un enum PHP.

Chapter 16: Enums PHP

Dentro del bucle, hacer que las cosas sean dinámicas no es nada nuevo... ¡lo cual es genial! Por ejemplo, `{{ ship.status }}`. Cuando actualizamos, ¡se imprime! Aunque, ¡ay! Los estados se están quedando sin espacio. ¡Nuestros datos no coinciden con el diseño!

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
↕ // ... lines 16 - 43
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

¡Giro argumental! Alguien cambió los requisitos del proyecto... ¡justo en medio! ¡Eso "nunca" ocurre! El nuevo plan es éste: cada nave debe tener un estado de `in progress`, `waiting`, o `completed`. En `src/Repository/StarshipRepository.php`, nuestras naves sí tienen un `status` -es este argumento-, pero es una cadena que puede establecerse con cualquier valor.

Crear un Enum

Así que tenemos que hacer algunas refactorizaciones para adaptarnos al nuevo plan. Pensemos: hay exactamente tres estados válidos. Este es un caso de uso perfecto para una enum PHP.

Si no estás familiarizado con los enums, son encantadores y una forma estupenda de organizar un conjunto de estados -como publicado, no publicado y borrador- o tamaños -pequeño,

mediano o grande- o cualquier cosa similar.

En el directorio `Model/` -aunque esto podría vivir en cualquier sitio... estamos creando el enum para nuestra propia organización- crea una nueva clase y llámala `StarshipStatusEnum`. En cuanto escribí la palabra enum, PhpStorm cambió la plantilla de `class` a una `enum`. Así que no estamos creando una clase, como puedes ver, creamos una `enum`

src/Model/StarshipStatusEnum.php

```
↕ // ... lines 1 - 2
3 namespace App\Model;
4
5 enum StarshipStatusEnum: string
6 {
↕ // ... lines 7 - 9
10 }
```

Añade un `: string` al enum para hacer lo que se llama un "enum respaldado por cadena". No profundizaremos demasiado, pero esto nos permite definir cada estado -como `WAITING` y asignarlo a una cadena, lo que será útil en un minuto. Añade un estado para `IN_PROGRESS` y finalmente uno para `COMPLETED`.

src/Model/StarshipStatusEnum.php

```
↕ // ... lines 1 - 2
3 namespace App\Model;
4
5 enum StarshipStatusEnum: string
6 {
7     case WAITING = 'waiting';
8     case IN_PROGRESS = 'in progress';
9     case COMPLETED = 'completed';
10 }
```

Y ya está Eso es todo lo que es un enum: un conjunto de "estados" que se centralizan en un solo lugar.

A continuación: abre la clase `Starship`. El último argumento es actualmente un estado `string`. Cámbialo para que sea un `StarshipStatusEnum`. Y en la parte inferior, el método `getStatus` devolverá ahora un `StarshipStatusEnum`.

src/Model/StarshipStatusEnum.php

```php
// ... lines 1 - 2
namespace App\Model;

enum StarshipStatusEnum: string
{
    case WAITING = 'waiting';
    case IN_PROGRESS = 'in progress';
    case COMPLETED = 'completed';
}
```

Por último, en `StarshipRepository` donde creamos cada `Starship`, mi editor está enfadado. Dice:

> *"¡Eh! ¡Este argumento acepta un `StarshipStatusEnum`, pero estás pasando una cadena!"*

Vamos a calmarlo. Cambia esto a `StarshipStatusEnum::`... ¡y autocompleta las opciones! Hagamos que la primera sea `IN_PROGRESS`. Y eso añadió la declaración `use` para el `enum` al principio de la clase. Para la siguiente, que sea `COMPLETED`... y para la última, `WAITING`.

src/Repository/StarshipRepository.php

```
↕ // ... lines 1 - 5
6  use App\Model\StarshipStatusEnum;
↕ // ... lines 7 - 8
9  class StarshipRepository
10 {
↕ // ... lines 11 - 14
15     public function findAll(): array
16     {
↕ // ... lines 17 - 18
19         return [
20             new Starship(
↕ // ... lines 21 - 24
25                 StarshipStatusEnum::IN_PROGRESS
26             ),
27             new Starship(
↕ // ... lines 28 - 31
32                 StarshipStatusEnum::COMPLETED
33             ),
34             new Starship(
↕ // ... lines 35 - 38
39                 StarshipStatusEnum::WAITING
40             ),
41         ];
42     }
↕ // ... lines 43 - 53
54 }
```

¡Refactorización realizada! Bueno... tal vez. Cuando actualizamos, ¡arruinado! Dice

> *"el objeto de clase `StarshipStatusEnum` no se ha podido convertir a cadena"*

Y viene de la llamada a Twig de `ship.status`.

Tiene sentido: `ship.status` es ahora un enum... que no puede imprimirse directamente como cadena. La solución más fácil, en `homepage.html.twig`, es añadir `.value`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
16                     <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
17                         <div class="flex justify-center min-[1174px]:justify-start">
↕ // ... line 18
19                             <div class="ml-5">
20                                 <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
↕ // ... line 21
22                                     <p class="uppercase text-xs text-nowrap">{{ ship.status.value }}</p>
23                                 </div>
↕ // ... lines 24 - 29
30                             </div>
31                         </div>
↕ // ... lines 32 - 42
43                     </div>
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Como hemos hecho que nuestro enum esté respaldado por una cadena, tiene una propiedad `value`, que será la cadena que asignamos al estado actual. Pruébalo ahora. ¡Tiene una pinta estupenda! En curso, completado, esperando.

A continuación: vamos a aprender cómo podemos hacer este último cambio un poco más elegante creando métodos más inteligentes en nuestra clase `Starship`. Luego daremos los toques finales a nuestro diseño.

Chapter 17: Métodos del modelo inteligente y dinamización del diseño

Añadir el `.value` al final del enum para imprimirlo funciona a las mil maravillas. Pero quiero mostrar otra solución más elegante.

Añadir métodos de modelo inteligentes

En `Starship`, probablemente será habitual que queramos obtener el estado de la cadena de un `Starship`. Para facilitarlo, ¿por qué no añadir aquí un método abreviado llamado`getStatusString()`? Éste devolverá un `string`, y dentro, devolverá `$this->status->value`.

src/Model/Starship.php

```
↕ // ... lines 1 - 4
5  class Starship
6  {
↕ // ... lines 7 - 40
41     public function getStatusString(): string
42     {
43         return $this->status->value;
44     }
45 }
```

Gracias a esto, en la plantilla, podemos acortar a `ship.statusString`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
16                     <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
17                         <div class="flex justify-center min-[1174px]:justify-start">
↕ // ... line 18
19                             <div class="ml-5">
20                                 <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
↕ // ... line 21
22                                     <p class="uppercase text-xs text-nowrap">{{ ship.statusString }}</p>
23                                 </div>
↕ // ... lines 24 - 29
30                             </div>
31                         </div>
↕ // ... lines 32 - 42
43                     </div>
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Ah, ¡y esto es más inteligencia Twig! ¡No hay ninguna propiedad llamada `statusString` en `Starship`! Pero a Twig no le importa. Ve que hay un método `getStatusString()` y lo llama.

Observa: cuando actualizamos, la página sigue funcionando. Me gusta mucho esta solución, así que la copiaré... y la repetiré aquí arriba para el atributo `alt`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
16                     <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
17                         <div class="flex justify-center min-[1174px]:justify-start">
18                             <img class="h-[83px] w-[84px]" src="/images/status-in-progress.png" alt="Status: {{ ship.statusString }}">
19                             <div class="ml-5">
20                                 <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
↕ // ... line 21
22                                     <p class="uppercase text-xs text-nowrap">{{ ship.statusString }}</p>
23                                 </div>
↕ // ... lines 24 - 29
30                             </div>
31                         </div>
↕ // ... lines 32 - 42
43                     </div>
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Y mientras arreglamos esto, en `show.html.twig`, imprimiremos el estado allí también. Así que haré ese mismo cambio... y luego cerraré esto.

templates/starship/show.html.twig

```
// ... lines 1 - 4
{% block body %}
// ... lines 6 - 11
<div class="md:flex justify-center space-x-3 mt-5 px-4 lg:px-8">
// ... lines 13 - 15
    <div class="space-y-5">
        <div class="mt-8 max-w-xl mx-auto">
            <div class="px-8 pt-8">
                <div class="rounded-2xl py-1 px-3 flex justify-center w-32 items-center bg-amber-400/10">
// ... line 20
                    <p class="uppercase text-xs">{{ ship.statusString }}</p>
                </div>
// ... lines 23 - 34
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Terminando nuestra Plantilla Dinámica

Bien: vamos a terminar de hacer dinámica nuestra plantilla de página de inicio: a partir de aquí todo es coser y cantar. Para el nombre del barco, `{{ ship.name }}`, para el capitán, `{{ ship.captain }}`. Y aquí abajo para la clase, `{{ ship.class }}`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
16                     <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11 flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
17                         <div class="flex justify-center min-[1174px]:justify-start">
↕ // ... line 18
19                             <div class="ml-5">
↕ // ... lines 20 - 23
24                                 <h4 class="text-[22px] pt-1 font-semibold">
25                                     <a
↕ // ... lines 26 - 27
28                                     >{{ ship.name }}</a>
29                                 </h4>
30                             </div>
31                         </div>
32                         <div class="flex justify-center min-[1174px]:justify-start mt-2 min-[1174px]:mt-0 shrink-0">
33                             <div class="border-r border-white/20 pr-8">
34                                 <p class="text-slate-400 text-xs">Captain</p>
35                                 <p class="text-xl">{{ ship.captain }}</p>
36                             </div>
37
38                             <div class="pl-8 w-[100px]">
39                                 <p class="text-slate-400 text-xs">Class</p>
40                                 <p class="text-xl">{{ ship.class }}</p>
41                             </div>
42                         </div>
43                     </div>
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Ah, y rellenemos el enlace: `{{ path() }}` y luego el nombre de la ruta. Estamos enlazando con la página del espectáculo del barco, así que la ruta es `app_starship_show`. Y como esto tiene un comodín `id`, pasa `id` a `ship.id`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5  {% block body %}
6      <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9          <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
16                     <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11
   flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
17                         <div class="flex justify-center min-
   [1174px]:justify-start">
↕ // ... line 18
19                             <div class="ml-5">
↕ // ... lines 20 - 23
24                                 <h4 class="text-[22px] pt-1 font-
   semibold">
25                                     <a
26                                         class="hover:text-slate-200"
27                                         href="{{ path('app_starship_show',
   { id: ship.id }) }}"
28                                     >{{ ship.name }}</a>
29                                 </h4>
30                             </div>
31                         </div>
↕ // ... lines 32 - 42
43                     </div>
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Y ahora, ¡mucho mejor! Se ve bien y podemos hacer clic en estos enlaces.

Rutas de imagen dinámicas

Pero... la imagen sigue rota. Antes, cuando copiamos las imágenes en nuestro directorio `assets/`, incluí tres archivos para los tres estados. Aquí arriba, estamos apuntando "más o menos" al estado en curso... pero ésta no es la forma correcta de referenciar imágenes en el directorio `assets/`. En su lugar, di `{{ asset() }}` y pasa la ruta relativa al directorio `assets/`, llamada ruta "lógica".

Si lo intentamos ahora... estamos más cerca. Pero la parte "en curso" tiene que ser dinámica en función del estado. Algo que podríamos intentar es la concatenación Twig: añadir `ship.status` a la cadena. Eso es posible, aunque es un poco feo.

En lugar de eso, volvamos a la solución que utilizamos hace un momento: hacer que todos los datos sobre nuestro `Starship` sean fácilmente accesibles... desde la clase `Starship`.

Esto es lo que quiero decir: añade un `public function getStatusImageFilename()` que devuelva una cadena.

src/Model/Starship.php

```
↕ // ... lines 1 - 4
5  class Starship
6  {
↕ // ... lines 7 - 45
46     public function getStatusImageFilename(): string
47     {
↕ // ... lines 48 - 52
53     }
54 }
```

Vamos a hacer toda la lógica para crear el nombre de archivo aquí mismo. Pondré una función `match`.

Esto dice: comprueba `$this->status` y si es igual a `WAITING`, devuelve esta cadena. Si es igual a `IN_PROGRESS` devuelve esta cadena y así sucesivamente.

src/Model/Starship.php

```
// ... lines 1 - 4
class Starship
{
// ... lines 7 - 45
    public function getStatusImageFilename(): string
    {
        return match ($this->status) {
            StarshipStatusEnum::WAITING => 'images/status-waiting.png',
            StarshipStatusEnum::IN_PROGRESS => 'images/status-in-progress.png',
            StarshipStatusEnum::COMPLETED => 'images/status-complete.png',
        };
    }
}
```

Y exactamente igual que antes, como tenemos un método `getStatusImageFilename()`, podemos, en Twig, hacer como si tuviéramos una propiedad `statusImageFilename`.

templates/main/homepage.html.twig

```
↕ // ... lines 1 - 4
5 {% block body %}
6     <main class="flex flex-col lg:flex-row">
↕ // ... lines 7 - 8
9         <div class="px-12 pt-10 w-full">
↕ // ... lines 10 - 13
14             <div class="space-y-5">
15                 {% for ship in ships %}
16                     <div class="bg-[#16202A] rounded-2xl pl-5 py-5 pr-11
   flex flex-col min-[1174px]:flex-row min-[1174px]:justify-between">
17                         <div class="flex justify-center min-
   [1174px]:justify-start">
↕ // ... line 18
19                             <div class="ml-5">
↕ // ... lines 20 - 23
24                                 <h4 class="text-[22px] pt-1 font-
   semibold">
25                                     <a
26                                         class="hover:text-slate-200"
27                                         href="{{ path('app_starship_show',
   { id: ship.id }) }}"
28                                     >{{ ship.name }}</a>
29                                 </h4>
30                             </div>
31                         </div>
↕ // ... lines 32 - 42
43                     </div>
44                 {% endfor %}
45             </div>
↕ // ... lines 46 - 50
51         </div>
52     </main>
53 {% endblock %}
```

Y ahora, ¡ya lo tenemos!

Últimos detalles para dinamizar el diseño

¡Últimos detalles! Rellenemos algunos enlaces que faltan, como este logotipo que debería ir a la página de inicio. Ahora mismo... no va a ninguna parte.

Recuerda que cuando queremos enlazar a una página, tenemos que asegurarnos de que esa ruta tiene un nombre. En `src/Controller/MainController.php`... nuestra página de

inicio no tiene nombre. Vale, tiene un nombre autogenerado, pero no queremos confiar en eso.

Añade `name:` ajustado a `app_homepage`. O puedes utilizar `app_main_homepage`.

src/Controller/MainController.php

```
// ... lines 1 - 9
class MainController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(StarshipRepository $starshipRepository): Response
// ... lines 14 - 22
}
```

Para enlazar el logo, en `base.html.twig`... aquí está... Utiliza `{{ path('app_homepage') }}`.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 13
    <body class="text-white" style="background: radial-gradient(102.21% 102.21% at 50% 28.75%, #00121C 42.62%, #013954 100%);">
        <div class="flex flex-col justify-between min-h-screen relative">
            <div>
                <header class="h-[114px] shrink-0 flex flex-col sm:flex-row items-center sm:justify-between py-4 sm:py-0 px-6 border-b border-white/20 shadow-md">
                    <a href="{{ path('app_homepage') }}">
// ... line 19
                    </a>
// ... lines 21 - 34
                </header>
// ... line 36
            </div>
// ... lines 38 - 40
        </div>
    </body>
</html>
```

Cópialo y repítelo a continuación para otro enlace de inicio.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 13
    <body class="text-white" style="background: radial-gradient(102.21% 102.21% at 50% 28.75%, #00121C 42.62%, #013954 100%);">
        <div class="flex flex-col justify-between min-h-screen relative">
            <div>
                <header class="h-[114px] shrink-0 flex flex-col sm:flex-row items-center sm:justify-between py-4 sm:py-0 px-6 border-b border-white/20 shadow-md">
                    <a href="{{ path('app_homepage') }}">
// ... line 19
                    </a>
                    <nav class="flex space-x-4 font-semibold">
                        <a class="hover:text-amber-400 pt-2" href="{{ path('app_homepage') }}">
                            Home
                        </a>
// ... lines 25 - 33
                    </nav>
                </header>
// ... line 36
            </div>
// ... lines 38 - 40
        </div>
    </body>
</html>
```

Dejaremos estos otros enlaces para un futuro tutorial.

De vuelta al navegador, ¡haz clic en ese logotipo! Ya está. El último enlace que falta está en la página del programa. Este enlace "atrás" también debería ir a la página de inicio.
Abre `show.html.twig`. Y arriba -ahí está- pegaré ese mismo enlace.

templates/starship/show.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="my-4 px-8">
    <a class="bg-white hover:bg-gray-200 rounded-xl p-2 text-black" href="{{ path('app_homepage') }}">
        <svg class="inline text-black" xmlns="http://www.w3.org/2000/svg" height="16" width="14" viewBox="0 0 448 512"><!--!Font Awesome Free 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license/free Copyright 2024 Fonticons, Inc.--><path fill="#000" d="M9.4 233.4c-12.5 12.5-12.5 32.8 0 45.3l160 160c12.5 12.5 32.8 12.5 45.3 0s12.5-32.8 0-45.3L109.2 288 416 288c17.7 0 32-14.3 32-32s-14.3-32-32-32l-306.7 0L214.6 118.6c12.5-12.5 12.5-32.8 0-45.3s-32.8-12.5-45.3 0l-160 160z"/></svg>
        Back
    </a>
</div>
// ... lines 12 - 38
{% endblock %}
```

Ok equipo, ¡el diseño está hecho! ¡Enhorabuena! Regálate un té... o un café con leche... o un donut o un paseo por la naturaleza para celebrarlo. ¡Porque esto es enorme! Nuestro sitio parece y se siente real. Estoy encantada.

Ahora podemos centrarnos en los detalles más sutiles. Por ejemplo, cuando hacemos clic en este enlace, se supone que la barra lateral se colapsa. Para ello, quiero presentarte mi herramienta favorita para escribir JavaScript: Stimulus.

Chapter 18: Stimulus: Escribir JavaScript profesional

Sabemos cómo escribir HTML en nuestras plantillas. Y manejamos CSS con Tailwind. ¿Qué pasa con JavaScript? Bueno, como con CSS, hay un archivo `app.js`, y ya está incluido en la página. Así que puedes poner aquí el JavaScript que quieras.

Pero te recomiendo encarecidamente que utilices una pequeña, pero malvada, biblioteca JavaScript llamada Stimulus. Es una de mis cosas favoritas de Internet. Tomas una parte de tu HTML existente y lo conectas a un pequeño archivo JavaScript, llamado controlador. Esto te permite añadir un comportamiento: por ejemplo, cuando pulses este botón, se llamará al método `greet` del controlador.

¡Y eso es todo! Seguro que Stimulus tiene más funciones, pero ya entiendes el núcleo de su funcionamiento. A pesar de su simplicidad, nos permitirá construir cualquier funcionalidad JavaScript y de interfaz de usuario que necesitemos, de forma fiable y predecible. Así que vamos a instalarlo.

Instalar Stimulus

Stimulus es una librería JavaScript, pero Symfony tiene un bundle que ayuda a integrarla. En tu terminal, si quieres ver lo que hace la receta, confirma tus cambios. Yo ya lo he hecho. Luego ejecuta:

```
composer require symfony/stimulus-bundle
```

Cuando esto termine... la receta ha hecho algunos cambios. Veamos los más importantes. El primero está en `app.js`: nuestro archivo JavaScript principal. Ábrelo y ya está.

assets/app.js

```
import './bootstrap.js';
/*
 * Welcome to your app's main JavaScript file!
 *
 * This file will be included onto the page via the importmap() Twig function,
 * which should already be in your base.html.twig.
 */
import './styles/app.css';

console.log('This log comes from assets/app.js - welcome to AssetMapper! 🎉');
```

Añadió un `import` en la parte superior - `./bootstrap.js` - a un nuevo archivo que vive justo al lado de éste.

assets/bootstrap.js

```
import { startStimulusApp } from '@symfony/stimulus-bundle';

const app = startStimulusApp();
// register any custom, 3rd party controllers here
// app.register('some_controller_name', SomeImportedController);
```

El propósito de este archivo es iniciar el motor Stimulus. Además, en `importmap.php`, la receta añadió el paquete JavaScript `@hotwired/stimulus` junto con otro archivo que ayuda a arrancar Stimulus dentro de Symfony.

importmap.php

```
// ... lines 1 - 15
return [
// ... lines 17 - 20
    '@hotwired/stimulus' => [
        'version' => '3.2.2',
    ],
    '@symfony/stimulus-bundle' => [
        'path' => './vendor/symfony/stimulus-bundle/assets/dist/loader.js',
    ],
];
```

Por último, la receta creó un directorio `assets/controllers/`. Aquí es donde vivirán nuestros controladores personalizados. ¡E incluía un controlador de demostración para que pudiéramos empezar! ¡Gracias!

assets/controllers/hello_controller.js

```
import { Controller } from '@hotwired/stimulus';

/*
 * This is an example Stimulus controller!
 *
 * Any element with a data-controller="hello" attribute will cause
 * this controller to be executed. The name "hello" comes from the
filename:
 * hello_controller.js -> "hello"
 *
 * Delete this file or adapt it for your use!
 */
export default class extends Controller {
    connect() {
        this.element.textContent = 'Hello Stimulus! Edit me in
assets/controllers/hello_controller.js';
    }
}
```

Estos archivos de controlador tienen una importante convención de nombres. Como se llama `hello_controller.js`, para conectarlo con un elemento de la página, utilizaremos `data-controller="hello"`.

Cómo funciona Stimulus

Así es como funciona. En cuanto Stimulus vea un elemento en la página con `data-controller="hello"`, instanciará una nueva instancia de este controlador y llamará al método `connect()`. Así, este controlador `hello` cambiará automática e instantáneamente el contenido del elemento al que está unido.

Y ya podemos verlo. Actualiza la página. Stimulus está ahora activo en nuestro sitio. Esto significa que está buscando elementos con `data-controller`. Hagamos algo salvaje: inspecciona los elementos de la página, busca cualquier elemento -como esta etiqueta de anclaje- y añade `data-controller="hello"`. Observa lo que ocurre cuando hago clic en desactivar para activar este cambio. ¡Pum! Stimulus ha visto ese elemento, ha instanciado nuestro controlador y ha llamado al método `connect()`. Y puedes hacer esto tantas veces como quieras en la página.

La cuestión es: no importa cómo llegue un elemento `data-controller` a tu página, Stimulus lo ve. Así que si hacemos una llamada Ajax que devuelva HTML y ponemos eso en la página...

sí, Stimulus va a verlo y nuestro JavaScript va a funcionar. Ésa es la clave: cuando escribes JavaScript con Stimulus, tu JavaScript siempre funcionará, independientemente de cómo y cuándo se añada ese HTML a la página.

Crear un controlador Stimulus que se pueda cerrar

Utilicemos Stimulus para activar nuestro botón de cierre. En el directorio `assets/controller/`, duplica `hello_controller.js` y crea uno nuevo llamado `closeable_controller.js`.

Borraré casi todo y me limitaré a lo más básico: importa `Controller` de Stimulus... y luego crea una clase que lo extienda.

assets/controllers/closeable_controller.js

```
1  import { Controller } from '@hotwired/stimulus';
2
3  export default class extends Controller {
↕  // ... lines 4 - 6
7  }
```

Esto no hace nada, pero ya podemos adjuntarlo a un elemento de la página. Éste es el plan: vamos a adjuntar el controlador a todo el elemento `aside`. Luego, cuando pulsemos este botón, eliminaremos el elemento `aside`.

Ese elemento vive en `templates/main/_shipStatusAside.html.twig`. Para adjuntar el controlador, añade `data-controller="closeable"`. ¿Ves ese autocompletado? Proviene de un plugin de Stimulus para PhpStorm.

templates/main/_shipStatusAside.html.twig

```
1  <aside
↕  // ... line 2
3      data-controller="closeable"
4  >
↕  // ... lines 5 - 35
36 </aside>
```

Si nos desplazamos y actualizamos, aún no ocurrirá nada: el botón de cerrar no funciona. Pero abre la consola de tu navegador. ¡Qué bien! Stimulus añade útiles mensajes de depuración: que se está iniciando y luego - lo que es importante - `closeable initialize`, `closeable connect`.

Esto significa que sí vio el elemento `data-controller` e inicializó ese controlador.

Así que volvamos a nuestro objetivo: cuando pulsemos este botón, queremos llamar a código dentro del controlador cerrable que elimine el `aside`. En `closeable_controller.js`, añade un nuevo método llamado, qué tal, `close()`. Dentro, digamos `this.element.remove()`.

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    close() {
        this.element.remove();
    }
}
```

En Stimulus, `this.element` será siempre el elemento al que esté unido tu controlador. Por tanto, este elemento `aside`. Pero por lo demás, este código es JavaScript estándar: cada Elemento tiene un método `remove()`.

Para llamar al método `close()`, en el botón, añade `data-action=""` luego el nombre de nuestro controlador - `closeable` - un signo `#`, y el nombre del método: `close`.

templates/main/_shipStatusAside.html.twig

```
<aside
// ... line 2
    data-controller="closeable"
>
    <div class="flex justify-between mt-11 mb-7">
// ... line 6
        <button data-action="closeable#close">
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
        </button>
    </div>
// ... lines 11 - 35
</aside>
```

Animar el cierre

Ya está Hora de probar. ¡Clic! ¡Ya está! ¡Pero quiero que sea más elegante! Quiero que se anime al cerrarse en lugar de ser instantáneo. ¿Podemos hacerlo? ¡Claro que sí! Y no necesitamos mucho JavaScript... porque el CSS moderno es increíble.

Sobre el elemento `aside`, añade una nueva clase CSS -puede ir en cualquier sitio- llamada `transition-all`.

Es una clase Tailwind que activa las transiciones CSS. Esto significa que si cambian ciertas propiedades de estilo -como que la anchura se ponga de repente a 0- hará una transición de ese cambio, en lugar de cambiarlo instantáneamente.

También añade `overflow-hidden` para que, al reducirse la anchura, no cree una extraña barra de desplazamiento.

Si probamos esto ahora, se sigue cerrando instantáneamente. Eso es porque no hay nada que transicionar: no estamos cambiando la anchura... sólo eliminando el elemento.

Pero fíjate en esto. Inspecciona el elemento y busca el `aside`: aquí está. Cambia manualmente la anchura a 0. ¡Genial! ¡Vas pequeñito, grande, pequeñito, grande, pequeñito! El lado CSS de las cosas está funcionando.

De vuelta en nuestro controlador, en lugar de eliminar el elemento, tenemos que cambiar la anchura a cero, esperar a que termine la transición CSS y luego eliminar el elemento. Podemos hacer lo primero con `this.element.style.width = 0`.

templates/main/_shipStatusAside.html.twig

```
<aside
// ... line 2
    data-controller="closeable"
>
    <div class="flex justify-between mt-11 mb-7">
// ... line 6
        <button data-action="closeable#close">
            <svg xmlns="http://www.w3.org/2000/svg" width="20" height="20" viewBox="0 0 448 512"><!--!Font Awesome Pro 6.5.1 by @fontawesome - https://fontawesome.com License - https://fontawesome.com/license (Commercial License) Copyright 2024 Fonticons, Inc.--><path fill="#fff" d="M384 96c0-17.7 14.3-32 32-32s32 14.3 32 32V416c0 17.7-14.3 32-32 32s-32-14.3-32-32V96zM9.4 278.6c-12.5-12.5-12.5-32.8 0-45.3l128-128c12.5-12.5 32.8-12.5 45.3 0s12.5 32.8 0 45.3L109.3 224 288 224c17.7 0 32 14.3 32 32s-14.3 32-32 32l-178.7 0 73.4 73.4c12.5 12.5 12.5 32.8 0 45.3s-32.8 12.5-45.3 0l-128-128z"/></svg>
        </button>
    </div>
// ... lines 11 - 35
</aside>
```

La parte complicada es esperar a que termine la transición CSS antes de eliminar el elemento. Para ayudarte con eso, voy a pegar un método en la parte inferior de nuestro controlador.

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    async close() {
        this.element.style.width = '0';
// ... lines 6 - 8
    }

    #waitForAnimation() {
        return Promise.all(
            this.element.getAnimations().map((animation) => animation.finished),
        );
    }
}
```

Si no estás familiarizado, el signo `#` hace que éste sea un método privado en JavaScript: un pequeño detalle. Este código parece lujoso, pero tiene una función sencilla: pedir al elemento que nos diga cuándo han terminado todas sus animaciones CSS.

Gracias a eso, aquí arriba, podemos decir `await this.#waitForAnimation()`. Y siempre que utilices `await`, tienes que poner `async` en la función alrededor de esto. No entraré en detalles sobre `async`, pero eso no cambiará el funcionamiento de nuestro código.

assets/controllers/closeable_controller.js

```
// ... lines 1 - 2
export default class extends Controller {
    async close() {
        this.element.style.width = '0';

        await this.#waitForAnimation();
        this.element.remove();
    }

    #waitForAnimation() {
        return Promise.all(
            this.element.getAnimations().map((animation) => animation.finished),
        );
    }
}
```

¡Comprobemos el resultado! Actualiza. Y... Me encanta.

A continuación, todo el mundo quiere una aplicación de página única, ¿verdad? Un sitio en el que no haya refrescos de página completa. Pero para construir una, ¿no necesitamos utilizar un framework JavaScript como React? ¡No! Vamos a transformar nuestra aplicación en una aplicación de una sola página en... unos 3 minutos con Turbo.

Chapter 19: Turbo: Tu aplicación de una sola página

Cuando construyo una interfaz de usuario, quiero que sea bonita, interactiva y fluida. Personalmente, elijo no utilizar frameworks frontales como React o Vue o Next. Pero tú puedes... y no tienen nada de malo: son herramientas estupendas. Además, ¡construir una API en Symfony es genial!

Pero si quieres construir tu HTML en Twig -como a mí me encanta hacer-, ¡podemos tener una interfaz de usuario interactiva, receptiva y súper rica!

Una gran pieza de una interfaz elegante es eliminar las recargas de página completa. Ahora mismo, cuando hago clic, mira: es rápido, pero son recargas de página completa. Eso no ocurre si utilizas algo como React o Vue.

Para eliminarlas, vamos a utilizar otra biblioteca de la misma gente que hizo Stimulus, llamada Turbo. Turbo puede hacer muchas cosas, pero su función principal es eliminar los refrescos de página completa. Al igual que Stimulus, es una biblioteca de JavaScript. Y también como Stimulus, Symfony tiene un bundle que ayuda a integrarla.

Instalación de Turbo

Busca tu terminal y ejecuta:

```
composer require symfony/ux-turbo
```

Esta vez, la receta ha hecho dos cambios interesantes. Te los mostraré. El primero está en `importmap.php`: añadió el paquete JavaScript `@hotwired/turbo`.

importmap.php

```
// ... lines 1 - 15
return [
// ... lines 17 - 26
    '@hotwired/turbo' => [
        'version' => '7.3.0',
    ],
];
```

El segundo cambio está en `assets/controllers.json`. Antes no hablamos de este archivo, pero lo añadió la receta StimulusBundle: es una forma de activar los controladores Stimulus que viven dentro de paquetes de terceros.

assets/controllers.json

```
{
    "controllers": {
        "@symfony/ux-turbo": {
            "turbo-core": {
                "enabled": true,
                "fetch": "eager"
            },
            "mercure-turbo-stream": {
                "enabled": false,
                "fetch": "eager"
            }
        }
    },
    "entrypoints": []
}
```

Así que el paquete PHP `symfony/ux-turbo` que acabamos de instalar tiene dentro un controlador JavaScript llamado `turbo-core`. Y como tenemos `enabled: true` aquí, significa que ese controlador está ahora registrado y disponible: es como si viviera en nuestro directorio `assets/controllers/`.

Ahora no vamos a utilizar este controlador directamente: no vamos a adjuntarlo a un elemento. Pero el hecho de que esté cargado y registrado en Stimulus es suficiente para activar Turbo en nuestro sitio.

Se acabaron los refrescos de página completa

¿Qué diablos significa esto? Es como magia: refresca la página y ¡bam! ¡Las recargas de página completa desaparecen! Fíjate bien: cuando vuelva a hacer clic, no verás que se recarga ¡Boom! Es superrápido y todo ocurre a través de Ajax.

Así es como funciona. Cuando hacemos clic en este enlace, Turbo intercepta el clic y, en lugar de recargar toda la página, hace una llamada Ajax a esa página. Esa llamada Ajax devuelve el HTML completo de esa página y luego Turbo lo pone en esta página.

Esa pequeña cosa transforma nuestro proyecto en una aplicación de una sola página y marca una gran diferencia en la rapidez de nuestro sitio.

Llamadas AJAX y la barra de herramientas de depuración web

Pero hay una cosa más. Actualizaré para que podamos verlo. Cada vez que haces una llamada Ajax en una aplicación Symfony - ya sea a través de Turbo o de cualquier otra forma - la Barra de Herramientas de Depuración Web lo nota. Míralo por aquí cuando haga clic. Ejecuta una lista de todas las llamadas Ajax realizadas en esta página. Y si queremos ver el perfil de cualquiera de esas peticiones Ajax, podemos hacer clic en el enlace.

Y sí... ahí lo tenemos. Aquí está la petición Ajax que se hizo para la página de inicio. Aunque con Turbo, ni siquiera necesitas recurrir a este truco porque, a medida que hacemos clic, toda esta barra es sustituida por la nueva Barra de Herramientas de Depuración Web para la página.

Ah, y escucha esto: en Turbo 8, que ya está a la venta, tu sitio parecerá aún más rápido, gracias a una nueva función llamada Clic Instantáneo. Con ella, cuando pasas el ratón por encima de un enlace, Turbo hace una llamada Ajax a esa página antes de que hagas clic. Entonces, cuando hagas clic, se cargará instantáneamente... o al menos tendrá una ventaja.

Turbo tiene muchas otras funciones, y utilizamos un montón de ellas en nuestro Tutorial LAST Stack, donde construimos un frontend con popovers, modales, notificaciones tostadas y mucho más.

Turbo requiere un buen JavaScript

Pero una nota sobre Turbo. Dado que las recargas de página completa ya no existen, tu JavaScript debe estar diseñado para gestionarlas. Mucho JavaScript espera recargas de

página completas... y si de repente se añade HTML a la página sin una recarga, se rompe. La buena noticia es que si escribes tu JavaScript en Stimulus, todo irá bien.

Observa. No importa cómo lleguemos a la página de inicio, nuestro JavaScript para cerrar la barra lateral sigue funcionando.

Muy bien equipo, ¡estamos en la recta final! Antes de terminar, quiero hacer un último capítulo extra en el que jugaremos con la impresionante herramienta de generación de Symfony: MakerBundle.

Chapter 20: Maker Bundle: ¡Generemos algo de código!

Me quito el sombrero por haber superado casi por completo el primer tutorial de Symfony. Has dado un gran paso hacia la construcción de lo que quieras en la web. Para celebrarlo, quiero jugar con MakerBundle: La impresionante herramienta de Symfony para la generación de código.

Composer require vs require-dev

Vamos a instalarlo:

```
composer require symfony/maker-bundle --dev
```

Aún no hemos visto la bandera `--dev`, pero no es tan importante. Muévete y abre `composer.json`. Gracias a la bandera, en lugar de que `symfony/maker-bundle` vaya bajo la clave `require`, se ha añadido aquí abajo, bajo `require-dev`.

composer.json

```
{
// ... lines 2 - 84
    "require-dev": {
// ... line 86
        "symfony/maker-bundle": "^1.52",
// ... lines 88 - 89
    }
}
```

Por defecto, cuando ejecutes `composer install`, descargará todo lo que esté bajo `require` y `require-dev`. Pero `require-dev` está pensado para paquetes que no necesitan estar disponibles en producción: paquetes que sólo necesitas cuando desarrollas localmente. Esto se debe a que, cuando despliegues, si quieres, puedes decirle a Composer:

> *“¡Eh! Instala sólo los paquetes de mi clave `require`: no instales las cosas de `require-dev`.”*

Eso puede darte un pequeño aumento de rendimiento en producción. Pero, en general, no es gran cosa.

Los comandos Maker

Acabamos de instalar un bundle. ¿Recuerdas lo principal que nos proporcionan los bundles? Exacto: servicios. Esta vez, los servicios que nos ha proporcionado MakerBundle son servicios que proporcionan nuevos comandos de consola. Redoble de tambores, por favor. Ejecuta:

```
php bin/console
```

O, en realidad, empezaré a ejecutar `symfony console`, que es lo mismo. ¡Gracias al nuevo bundle, tenemos un montón de comandos que empiezan por `make`! Comandos para generar un sistema de seguridad, hacer un controlador, generar entidades de doctrina para hablar con la base de datos, formularios, oyentes, un formulario de registro.... ¡muchas, muchas cosas!

Generar un comando de consola

Utilicemos uno de éstos para crear nuestro propio comando de consola personalizado. Ejecuta:

```
symfony console make:command
```

Esto nos preguntará interactivamente por nuestro comando. Llamémoslo: `app:ship-report`. ¡Listo!

Esto ha creado exactamente un archivo: `src/Command/ShipReportCommand.php`. ¡Vamos a comprobarlo!

src/Command/ShipReportCommand.php

```php
// ... lines 1 - 2
namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;

#[AsCommand(
    name: 'app:ship-report',
    description: 'Add a short description for your command',
)]
class ShipReportCommand extends Command
{
    public function __construct()
    {
        parent::__construct();
    }

    protected function configure(): void
    {
        $this
            ->addArgument('arg1', InputArgument::OPTIONAL, 'Argument description')
            ->addOption('option1', null, InputOption::VALUE_NONE, 'Option description')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $arg1 = $input->getArgument('arg1');

        if ($arg1) {
            $io->note(sprintf('You passed an argument: %s', $arg1));
        }

        if ($input->getOption('option1')) {
            // ...
        }

```

```
45          $io->success('You have a new command! Now make it your own! Pass --help to see your options.');
46
47          return Command::SUCCESS;
48      }
49  }
```

¡Genial! Esta es una clase normal - es un servicio, por cierto - pero con un atributo encima: `#[AsCommand]`. Esto le dice a Symfony:

> *"¡Eh! ¿Ves este servicio? No es sólo un servicio: Me gustaría que lo incluyeras en la lista de comandos de la consola."*

El atributo incluye el nombre del comando y una descripción. Además, la propia clase tiene un método `configure()` en el que podemos añadir argumentos y opciones. Pero la parte principal es que, cuando alguien llame a este comando, Symfony llamará a `execute()`.

Esta variable `$io` es genial. Nos permite mostrar cosas -como `$this->note()`o `$this->success()` - con diferentes estilos. Y aunque no lo veamos aquí, también podemos hacer preguntas al usuario de forma interactiva.

¿Y lo mejor? Con sólo crear esta clase, ¡ya está lista para usar! Pruébala:

```
symfony console app:ship-report
```

¡Qué guay! El mensaje de aquí abajo procede del mensaje de éxito de la parte inferior del comando. Y gracias a `configure()`, tenemos un argumento llamado`arg1`. Los argumentos son cadenas que pasamos después del comando, como:

```
symfony console app:ship-report ryan
```

Dice

> *"Has pasado un argumento: ryan"*

... que viene de este lugar del comando.

Construir una barra de progreso

Hay muchas cosas divertidas que puedes hacer con los comandos... y quiero jugar con una de ellas. Uno de los superpoderes del objeto `$io` es crear barras de progreso animadas.

Imagina que estamos construyendo un informe sobre un barco... y requiere algunas consultas pesadas. Así que queremos mostrar una barra de progreso en la pantalla. Para ello, decimos `$io->progressStart()` y le pasamos el número de filas de datos que estemos recorriendo y manejando. Imaginemos que estamos haciendo un bucle sobre 100 filas de datos para este informe.

En lugar de hacer un bucle sobre datos reales, crea un bucle falso con `for`. ¡Incluso voy a incluir la variable `$i` en el medio! Dentro, para hacer avanzar la barra de progreso, di `$io->advance()`. Entonces, aquí es donde haríamos nuestra consulta pesada o trabajo pesado. Finge eso con un `usleep(10000)` para crear una breve pausa.

Después del bucle, termina con `$io->progressFinish()`.

src/Command/ShipReportCommand.php

```
↕ // ... lines 1 - 16
17 class ShipReportCommand extends Command
18 {
↕ // ... lines 19 - 31
32     protected function execute(InputInterface $input, OutputInterface
$output): int
33     {
↕ // ... lines 34 - 44
45         $io->progressStart(100);
46         for ($i = 0; $i < 100; ++$i) {
47             $io->progressAdvance();
48             usleep(10000);
49         }
50         $io->progressFinish();
↕ // ... lines 51 - 54
55     }
56 }
```

Ya está Gira y pruébalo:

```
symfony console app:ship-report ryan
```

Qué guay.

Y... ¡eso es todo, gente! Choca esos cinco contigo mismo... o, mejor, ¡sorprende a un compañero de trabajo con un choca esos cinco saltarín! Después, celébralo con una merecida cerveza, un té, un paseo por la manzana o un partido de frisbee con tu perro. Porque... ¡lo has conseguido! Has dado el primer gran paso para ser peligroso con Symfony. Entonces, vuelve y prueba estas cosas: juega con ellas, construye un blog, crea unas cuantas páginas estáticas, lo que sea. Eso marcará una gran diferencia.

Y si alguna vez tienes alguna pregunta, miramos atentamente la sección de comentarios debajo de cada vídeo y respondemos a todo. Además, ¡sigue adelante! En el próximo tutorial, vamos a ponernos aún más peligrosos profundizando en la configuración y los servicios de Symfony: los sistemas que dirigen todo lo que harás en Symfony.

Muy bien, amigos, ¡hasta la próxima!

With <3 from SymfonyCasts