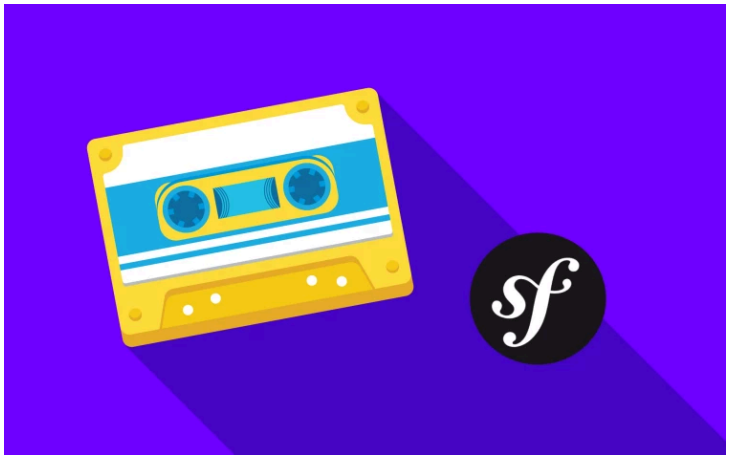
Desarrollo Armonioso con Symfony 6



Chapter 1: Hola Symfony

Bienvenido. Hola. Hola, mi nombre es Ryan y tengo el absoluto placer de presentarte el hermoso y fascinante y productivo mundo de Symfony 6. En serio, me siento como Willie Wonka invitándote a mi fábrica de chocolate, excepto que, con suerte, con menos lesiones relacionadas con el azúcar. De todos modos, si eres nuevo en Symfony, estoy... ¡sinceramente un poco celoso! Te va a encantar el viaje... y espero que te conviertas en un desarrollador aún mejor por el camino: definitivamente vas a construir cosas geniales.

La salsa secreta de Symfony es que empieza siendo diminuto, lo que hace que sea fácil de aprender. Pero luego, amplía sus características automáticamente a través de un sistema de recetas único. En Symfony 6, esas características incluyen nuevas herramientas de JavaScript y un nuevo sistema de seguridad... sólo por nombrar dos de las muchas cosas nuevas.

Symfony también es rápido como un rayo, con un gran enfoque en la creación de una experiencia alegre para el desarrollador, pero sin sacrificar las mejores prácticas de programación. Sí: consigues amar la codificación y amar tu código. Lo sé... ha sonado cursi, pero es cierto.

Así que ven conmigo y estarás en un mundo de pura elucidación.

Es la primera vez que canto en estos tutoriales... y quizá la última. Empecemos.

Instalar el binario "symfony

Dirígete a https://symfony.com/download. En esta página, encontrarás algunas instrucciones - que variarán en función de tu sistema operativo- sobre cómo descargar algo llamado el binario de Symfony.

Esto... no es realmente Symfony. Es sólo una herramienta de línea de comandos que nos ayudará a iniciar nuevos proyectos Symfony y nos dará algunas buenas herramientas de desarrollo local. Es opcional, pero lo recomiendo encarecidamente

Una vez que hayas instalado esto - yo ya lo he hecho - abre tu aplicación de terminal favorita. Yo estoy usando iTerm para mac, pero no importa. Si lo has configurado todo correctamente,

deberías poder ejecutarlo:

```
symfony
```

O incluso mejor

```
symfony list
```

para ver una lista de todas las "cosas" que puede hacer este binario de symfony. Hay muchas cosas aquí: cosas que ayudan al desarrollo "local"... y también algunos servicios opcionales para el despliegue. Vamos a repasar las cosas que necesitas saber a lo largo del camino.

¡Iniciemos una aplicación Symfony!

Bien, queremos iniciar una nueva y brillante aplicación Symfony. Para ello, ejecuta:

```
symfony new mixed_vinyl
```

Donde "mixed_vinyl" es el directorio en el que se descargará la nueva app. Se trata de nuestro proyecto secreto para combinar la mejor parte de los años 90 -no, no el Internet de acceso telefónico, hablo de las cintas de mezcla- con el deleite auditivo de los discos. Más adelante hablaremos de ello.

Entre bastidores, este comando utiliza Composer -el gestor de paquetes de PHP- para crear el nuevo proyecto. Más adelante hablaremos de ello.

El resultado final es que podemos pasar a nuestro nuevo directorio `mixed_vinyl`. Abre esta carpeta en tu editor favorito. Yo estoy usando PhpStorm y lo recomiendo encarecidamente.

Conociendo nuestro nuevo Proyecto

¿Qué ha hecho ese comando `symfony new`? Ha arrancado un nuevo proyecto Symfony! Ooh. Y ya tenemos un repositorio git. Ejecuta:

```
git status
```

Sí: en la rama principal, nada que confirmar. Prueba:

```
git log
```

Genial. Después de descargar el nuevo proyecto, el comando confirmó todos los archivos originales automáticamente... lo cual fue muy agradable. Aunque me gustaría que el primer mensaje de confirmación fuera un poco más rockero.

¡Lo que realmente quiero mostrarte es que nuestro nuevo proyecto es súper pequeño! Prueba este comando:

```
git show --name-only
```

¡Sí! Todo nuestro proyecto es... unos 17 archivos. Y aprenderemos sobre todos ellos a lo largo del camino. Pero quiero que te sientas cómodo: no hay mucho código aquí.

Vamos a añadir funciones poco a poco. Pero si quieres empezar con un proyecto más grande y con más funciones, puedes hacerlo ejecutando el comando `symfony new` con `--webapp`.

Tip

Si quieres una nueva aplicación Symfony con todas las funciones, echa un vistazo a https://github.com/dunglas/symfony-docker

Comprobación de los requisitos del sistema

Antes de saltar a la codificación, vamos a asegurarnos de que nuestro sistema está listo. Ejecuta otro comando del binario de symfony:

```
symfony check:req
```

¡Parece que está bien! Si a tu instalación de PHP le falta alguna extensión... o hay algún otro problema... como que tu ordenador es en realidad una tetera, esto te lo hará saber.

Iniciar el servidor web de desarrollo

Entonces: tenemos una nueva aplicación Symfony aquí... ¡y nuestro sistema está listo! Todo lo que necesitamos ahora es un subwoofer. Es decir, ¡un servidor web! Puedes configurar un servidor web real como Nginx o algo moderno como Caddy. Pero para el desarrollo local, el binario de Symfony puede ayudarnos. Corre:

```
symfony serve -d
```

Y... ¡tenemos un servidor web funcionando! ¡Vuelve!

La primera vez que ejecutes esto, es posible que te pida que ejecutes otro comando para configurar un certificado SSL, lo cual está bien porque entonces el servidor soporta https.

¡Momento de la verdad! Copia la URL, gira hacia tu navegador, aguanta la respiración y ¡woo! Hola página de bienvenida de Symfony 6... completa con extravagantes cambios de color cada vez que recargamos.

A continuación: conozcamos -y hagámonos amigos- del código dentro de nuestra aplicación, para poder desmitificar lo que hace cada parte. Luego codificaremos.

Chapter 2: Conoce nuestra Diminuta App

Vamos a conocer nuestro nuevo proyecto porque mi objetivo final es que entiendas realmente cómo funcionan las cosas. Como he mencionado, no hay mucho aquí todavía... unos 15 archivos. Y realmente sólo hay tres directorios en los que tengamos que pensar o preocuparnos.

El directorio public/

El primero es `public/`... y esto es sencillo: es la raíz del documento. En otras palabras, si necesitas que un archivo sea accesible públicamente -como un archivo de imagen o un archivo CSS- tiene que vivir dentro de `public/`.

Ahora mismo, esto contiene exactamente un archivo: `index.php`, que se llama "controlador frontal"

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']);
};
```

Ooo. Es una palabra elegante que significa que, independientemente de la URL a la que vaya el usuario, éste es el script que siempre se ejecuta primero. Su trabajo es arrancar Symfony y ejecutar nuestra aplicación. Y ahora que lo hemos visto, probablemente no tengamos que pensar ni abrirlo nunca más.

config/ & src/

Y, realmente, aparte de poner archivos CSS o de imagen en `public/`, este no es un directorio con el que vayas a tratar en el día a día. Lo que significa... Que en realidad sólo hay dos directorios en los que tenemos que pensar: `config/` y `src/`.

El directorio `config/` contiene... ¡gatos! Ya me gustaría. No, contiene archivos de configuración. Y `src/` contiene el 100% de tus clases PHP. Pasaremos el 95% de nuestro tiempo dentro del directorio `src/`.

composer.json & vendor/

Bien... ¿dónde está "Symfony"? Nuestro proyecto comenzó con un archivo `composer.json`. En él se enumeran todas las librerías de terceros que necesita nuestra aplicación. El comando "symfony new" que ejecutamos en secreto utilizó "composer" -es decir, el gestor de paquetes de PHP- para instalar estas librerías... que en realidad es sólo una forma de decir que Composer descargó estas librerías en el directorio `vendor/`.

El propio Symfony es en realidad una colección de un montón de pequeñas bibliotecas que resuelven cada una un problema específico. En el directorio `vendor/symfony/`, parece que ya tenemos unas 25 de ellas. Técnicamente, nuestra aplicación sólo requiere estos seis paquetes, pero algunos de ellos requieren otros paquetes... y Composer es lo suficientemente inteligente como para descargar todo lo que necesitamos.

De todos modos, "Symfony", o en realidad, un conjunto de bibliotecas de Symfony, vive en el directorio `vendor/` y nuestra nueva aplicación aprovecha ese código para hacer su trabajo. Más adelante hablaremos de Composer y de la instalación de paquetes de terceros. Pero en su mayor parte, `vendor/` es otro directorio del que... ¡no tenemos que preocuparnos!

bin/ y var/

Entonces, ¿qué queda? Bueno, `bin/` contiene exactamente un archivo... y siempre contendrá sólo este archivo. Hablaremos de lo que hace `bin/console` un poco más tarde. Y el directorio `var/` contiene archivos de caché y de registro. Esos archivos son importantes... pero nunca necesitaremos mirar o pensar en esas cosas.

Sí, vamos a vivir casi exclusivamente dentro de los directorios `config/` y `src/`.

Configuración de PhpStorm

Bien, una última tarea antes de empezar a codificar. Siéntete libre de utilizar el editor de código que quieras: PhpStorm, VS Code, code carrier pigeon, lo que sea. Pero recomiendo encarecidamente PhpStorm. Hace que desarrollar con Symfony sea un sueño... ¡y ni siquiera me pagan por decir eso! Aunque, si quieren empezar a pagarme, acepto el pago en stroopwafels.

Parte de lo que hace que PhpStorm sea tan bueno es un plugin diseñado específicamente para Symfony. Voy a mis preferencias de PhpStorm y, dentro, busco Plugins, Marketplace y luego busco Symfony. Aquí está. ¡Este plugin es increíble.... lo que puedes ver porque ha sido descargado 5,4 millones de veces! Añade toneladas de locas funciones de autocompletado que son específicas de Symfony.

Si aún no lo tienes, instálalo. Una vez instalado, vuelve a Configuración y busca aquí arriba "Symfony" para encontrar una nueva área de Symfony. El único truco de este plugin es que tienes que activarlo para cada proyecto. Así que haz clic en esa casilla. Además, no es demasiado importante, pero cambia el directorio web a `public/`.

Pulsa Ok y... ¡estamos listos! Vamos a dar vida a nuestra aplicación creando nuestra primera página a continuación.

Chapter 3: Rutas, controladores y respuestas

Tengo que decir que echo de menos los años 90. Bueno, no los beanie babies y... definitivamente no la forma de vestir de entonces, pero... las cintas de mezclas. Si no eras un niño en los 80 o los 90, quizá no sepas lo difícil que era compartir tus canciones favoritas con tus amigos. Oh sí, estoy hablando de un mashup de Michael Jackson, Phil Collins y Paula Abdul. La perfección.

Para aprovechar esa nostalgia, pero con un toque hipster, vamos a crear una nueva aplicación llamada Mixed Vinyl: una tienda en la que los usuarios pueden crear cintas de mezclas, con Boyz || Men, Mariah Carey y Smashing Pumpkins... sólo que prensadas en un disco de vinilo. Hmm, puede que tenga que poner un tocadiscos en mi coche.

La página que estamos viendo, que es súper bonita y cambia de color cuando refrescamos... no es una página real. Es sólo una forma de que Symfony nos diga "hola" y nos enlace a la documentación. Y por cierto, la documentación de Symfony es genial, así que no dudes en consultarla mientras aprendes.

Rutas y controladores

Vale: todo framework web en cualquier lenguaje tiene el mismo trabajo: ayudarnos a crear páginas, ya sean páginas HTML, respuestas JSON de la API o arte ASCII. Y casi todos los marcos lo hacen de la misma manera: mediante un sistema de rutas y controladores. La ruta define la URL de la página y apunta a un controlador. El controlador es una función PHP que construye esa página.

Así que ruta + controlador = página. Son matemáticas, gente.

Crear el controlador

Vamos a construir estas dos cosas... un poco al revés. Así que primero, vamos a crear la función del controlador. En Symfony, la función del controlador es siempre un método dentro de

una clase PHP. Te lo mostraré: en el directorio `src/Controller/`, crea una nueva clase PHP. Vamos a llamarla `VinylController`, pero el nombre puede ser cualquier cosa.

src/Controller/VinylController.php

```php
<?php

namespace App\Controller;

class VinylController
{
}
```

Y, ¡felicidades! ¡Es nuestra primera clase PHP! ¿Y adivina dónde vive? En el directorio `src/`, donde vivirán todas las clases PHP. Y en general, no importa cómo organices las cosas dentro de `src/`: normalmente puedes poner las cosas en el directorio que quieras y nombrar las clases como quieras. Así que da rienda suelta a tu creatividad.

Tip

En realidad, los controladores deben vivir en `src/Controller/`, a menos que cambies alguna configuración. La mayoría de las clases de PHP pueden vivir en cualquier lugar de `src/`.

Pero hay dos reglas importantes. En primer lugar, fíjate en el espacio de nombres que PhpStorm ha añadido sobre la clase: `App\Controller`. Independientemente de cómo decidas organizar tu directorio `src/`, el espacio de nombres de una clase debe coincidir con la estructura del directorio... empezando por `App`. Puedes imaginar que el espacio de nombres `App\` apunta al directorio `src/`. Entonces, si pones un archivo en un subdirectorio `Controller/`, necesita una parte `Controller` en su espacio de nombres.

Si alguna vez metes la pata, por ejemplo, si escribes algo mal o te olvidas de esto, lo vas a pasar mal. PHP no podrá encontrar la clase: obtendrás un error de "clase no encontrada". Ah, y la otra regla es que el nombre de un archivo debe coincidir con el nombre de la clase dentro de él, más `.php`. Por lo tanto, `VinylController.php`. Seguiremos esas dos reglas para todos los archivos que creemos en `src/`.

Crear el controlador

Volvemos a nuestra tarea de crear una función de controlador. Dentro, añade un nuevo método público llamado `homepage()`. Y no, el nombre de este método tampoco importa: prueba a ponerle el nombre de tu gato: ¡funcionará!

Por ahora, sólo voy a poner una declaración `die()` con un mensaje.

src/Controller/VinylController.php

```php
<?php

namespace App\Controller;

class VinylController
{
    public function homepage()
    {
        die('Vinyl: Definitely NOT a fancy-looking frisbee!');
    }
}
```

Crear la ruta

¡Buen comienzo! Ahora que tenemos una función de controlador, vamos a crear una ruta, que define la URL de nuestra nueva página y apunta a este controlador. Hay varias formas de crear rutas en Symfony, pero casi todo el mundo utiliza atributos.

Así es como funciona. Justo encima de este método, decimos `#[]`. Esta es la sintaxis de atributos de PHP 8, que es una forma de añadir configuración a tu código. Empieza a escribir `Route`. Pero antes de que termines, fíjate en que PhpStorm lo está autocompletando. Pulsa el tabulador para dejar que termine.

Eso, muy bien, completó la palabra `Route` para mí. Pero lo más importante es que ha añadido una declaración `use` en la parte superior. Siempre que utilices un atributo, debes tener una declaración `use` correspondiente en la parte superior del archivo.

Dentro de `Route`, pasa `/`, que será la URL de nuestra página.

src/Controller/VinylController.php

```php
<?php

namespace App\Controller;

use Symfony\Component\Routing\Annotation\Route;

class VinylController
{
    #[Route('/')]
    public function homepage()
    {
        die('Vinyl: Definitely NOT a fancy-looking frisbee!');
    }
}
```

Y... ¡listo! Esta ruta define la URL y apunta a este controlador... simplemente porque está justo encima de este controlador.

¡Vamos a probarlo! Refresca y... ¡felicidades! ¡Symfony miró la URL, vio que coincidía con la ruta - `/` o sin barra es lo mismo para la página de inicio - ejecutó nuestro controlador y golpeó la declaración `die`!

Ah, y por cierto, sigo diciendo función del controlador. Comúnmente se llama simplemente "controlador" o "acción"... sólo para confundir.

Devolver una respuesta

Bien, dentro del controlador -o acción- podemos escribir el código que queramos para construir la página, como hacer consultas a la base de datos, llamadas a la API, renderizar una plantilla, lo que sea. Al final vamos a hacer todo eso.

Lo único que le importa a Symfony es que tu controlador devuelva un objeto `Response`. Compruébalo: escribe `return` y luego empieza a escribir `Response`. Woh: hay bastantes clases `Response` ya en nuestro código... ¡y dos son de Symfony! Queremos la de HTTP foundation. HTTP foundation es una de esas librerías de Symfony... y nos da bonitas clases para cosas como la Petición, la Respuesta y la Sesión. Pulsa el tabulador para autocompletar y termina eso.

Oh, debería haber dicho devolver una nueva respuesta. Así está mejor. Ahora dale al tabulador. Cuando dejé que `Response` autocompletara la primera vez, muy importante, PhpStorm añadió

esta declaración de uso en la parte superior. Cada vez que hagamos referencia a una clase o interfaz, tendremos que añadir una sentencia `use` al principio del archivo en el que estemos trabajando.

Al dejar que PhpStorm autocompletara eso por mí, añadió la declaración `use` automáticamente. Lo haré cada vez que haga referencia a una clase. Ah, y si todavía eres un poco nuevo en lo que respecta a los espacios de nombres de PHP y las declaraciones `use`, echa un vistazo a nuestro breve y gratuito tutorial sobre espacios de nombres de PHP.

De todos modos, dentro de `Response`, podemos poner lo que queramos devolver al usuario: HTML, JSON o, por ahora, un simple mensaje, como el título del vinilo Mixto en el que estamos trabajando: PB y jams.

src/Controller/VinylController.php

```php
<?php

namespace App\Controller;

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

class VinylController
{
    #[Route('/')]
    public function homepage()
    {
        return new Response('Title: "PB and Jams"');
    }
}
```

Bien, equipo, ¡vamos a ver qué pasa! Actualiza y... ¡PB y mermeladas! Puede que no parezca gran cosa, ¡pero acabamos de construir nuestra primera página Symfony totalmente funcional! ¡Ruta + controlador = beneficio!

Y acabas de aprender la parte más fundamental de Symfony... y sólo estamos empezando. Ah, y como nuestros controladores siempre devuelven un objeto `Response`, es opcional, pero puedes añadir un tipo de retorno a esta función si lo deseas. Pero eso no cambia nada: sólo es una forma agradable de codificar.

src/Controller/VinylController.php

```
// ... lines 1 - 9
    #[Route('/')]
    public function homepage(): Response
// ... lines 12 - 16
```

A continuación me siento bastante seguro. Así que vamos a crear otra página, pero con una ruta mucho más elegante que coincide con un patrón comodín.

Chapter 4: Rutas comodín

La página de inicio será el lugar donde el usuario podrá diseñar y construir su próxima cinta de mezclas. Pero además de crear nuevas cintas, los usuarios también podrán explorar las creaciones de otras personas.

Crear una segunda página

Vamos a crear una segunda página para eso. ¿Cómo? Añadiendo un segundo controlador: función pública, qué tal `browse`: el nombre no importa realmente. Y para ser responsable, añadiré un tipo de retorno `Response`.

Por encima de esto, necesitamos nuestra ruta. Ésta será exactamente igual, salvo que pondremos la URL en `/browse`. Dentro del método, ¿qué es lo que siempre devolvemos de un controlador? Así es: ¡un objeto `Response`! Devuelve un nuevo `Response`... con un mensaje corto para empezar.

src/Controller/VinylController.php

```
↕ // ... lines 1 - 7
8  class VinylController
9  {
↕ // ... lines 10 - 15
16     #[Route('/browse')]
17     public function browse(): Response
18     {
19         return new Response('Breakup vinyl? Angsty 90s rock? Browse the
   collection!');
20     }
21 }
```

¡Vamos a probarlo! Si actualizamos la página de inicio, no cambia nada. Pero si vamos a `/browse`... ¡lo machacamos! ¡Una segunda página en menos de un minuto! ¡Caramba!

En esta página, acabaremos por listar las cintas de mezclas de otros usuarios. Para ayudar a encontrar algo que nos guste, quiero que los usuarios también puedan buscar por género. Por

ejemplo, si voy a `/browse/death-metal`, eso me mostraría todas las cintas de vinilo de death metal. Hardcore.

Por supuesto, si probamos esta URL ahora mismo... no funciona.

> *"No se ha encontrado la ruta"*

No se han encontrado rutas coincidentes para esta URL, por lo que nos muestra una página 404. Por cierto, lo que estás viendo es la elegante página de excepciones de Symfony, porque estamos desarrollando. Nos da muchos detalles cuando algo va mal. Cuando finalmente despliegues a producción, puedes diseñar una página de error diferente que verían tus usuarios.

{Cartel de la muerte} Rutas

De todos modos, la forma más sencilla de hacer que esta URL funcione es simplemente... cambiar la URL a `/browse/death-metal`

src/Controller/VinylController.php

```
↕ // ... lines 1 - 7
8  class VinylController
9  {
↕ // ... lines 10 - 15
16     #[Route('/browse/death-metal')]
17     public function browse(): Response
18     {
19         return new Response('Breakup vinyl? Angsty 90s rock? Browse the
   collection!');
20     }
21 }
```

Pero... no es súper flexible, ¿verdad? Necesitaríamos una ruta para cada género... ¡que podrían ser cientos! Y además, ¡acabamos de matar la URL `/browse`! Ahora es 404.

Lo que realmente queremos es una ruta que coincida con `/browse/<ANYTHING>`. Y podemos hacerlo con un comodín. Sustituye el código duro `death-metal` por `{}` y, dentro, `slug`. Slug es sólo una palabra técnica para designar un "nombre seguro para la URL". En realidad, podríamos haber puesto cualquier cosa dentro de las llaves, como `{genre}` o `{coolMusicCategory}`: no hay ninguna diferencia. Pero sea lo que sea que pongamos dentro de este comodín, se nos permite tener un argumento con ese mismo nombre: `$slug`.

src/Controller/VinylController.php

```
// ... lines 1 - 7
class VinylController
{
// ... lines 10 - 15
    #[Route('/browse/{slug}')]
    public function browse(): Response
    {
        return new Response('Breakup vinyl? Angsty 90s rock? Browse the collection!');
    }
}
```

Sí, si vamos a `/browse/death-metal`, coincidirá con esta ruta y pasará la cadena `death-metal` a ese argumento. La coincidencia se hace por nombre: `{slug}` conecta con `$slug`.

Para ver si funciona, devolvamos una respuesta diferente: `Genre` y luego la `$slug`.

src/Controller/VinylController.php

```
// ... lines 1 - 7
class VinylController
{
// ... lines 10 - 15
    #[Route('/browse/{slug}')]
    public function browse($slug): Response
    {
        return new Response('Genre: '.$slug);

        //return new Response('Breakup vinyl? Angsty 90s rock? Browse the collection!');
    }
// ... lines 23 - 24
```

¡Hora de probar! Vuelve a `/browse/death-metal` y... ¡sí! Prueba con `/browse/emo` y ¡sí! ¡Estoy mucho más cerca de mi cinta de mezcla de Dashboard Confessional!

Ah, y es opcional, pero puedes añadir un tipo `string` al argumento `$slug`. Eso no cambia nada... es sólo una bonita forma de programar: el `$slug` ya iba a ser siempre una cadena.

src/Controller/VinylController.php

```
↕ // ... lines 1 - 7
8  class VinylController
9  {
↕ // ... lines 10 - 15
16     #[Route('/browse/{slug}')]
17     public function browse(string $slug): Response
18     {
↕ // ... lines 19 - 21
22     }
↕ // ... lines 23 - 24
```

Un poco más adelante, aprenderemos cómo puedes convertir un comodín numérico -como el número 5- en un número entero si así lo deseas.

Usando el componente de cadena de Symfony

Hagamos esta página un poco más elegante. En lugar de imprimir el slug exactamente, vamos a convertirlo en un título. Digamos `$title = str_replace()` y sustituyamos los guiones por espacios. Luego, aquí abajo, utiliza el título en la respuesta. En un futuro tutorial, vamos a consultar la base de datos para estos géneros, pero, por ahora, al menos podemos hacer que tenga un aspecto más agradable.

src/Controller/VinylController.php

```
↕ // ... lines 1 - 7
8  class VinylController
9  {
↕ // ... lines 10 - 15
16     #[Route('/browse/{slug}')]
17     public function browse(string $slug): Response
18     {
19         $title = str_replace('-', ' ', $slug);
20
21         return new Response('Genre: '.$title);
22
↕ // ... line 23
24     }
↕ // ... lines 25 - 26
```

Si lo probamos, el Emo no se ve diferente... pero el death metal sí. ¡Pero quiero que sea más elegante! Añade otra línea con `$title =` y luego escribe `u` y autocompleta una función que se llama literalmente... `u`.

No utilizamos muchas funciones de Symfony, pero éste es un ejemplo raro. Proviene de una biblioteca de Symfony llamada `symfony/string`. Como he mencionado, Symfony tiene muchas bibliotecas diferentes -también llamadas componentes- y vamos a aprovechar esas bibliotecas todo el tiempo. Esta te ayuda a hacer transformaciones de cadenas... y resulta que ya está instalada.

Mueve el `str_replace()` al primer argumento de `u()`. Esta función devuelve un objeto sobre el que podemos hacer operaciones de cadena. Uno de los métodos se llama `title()`. Digamos `->title(true)` para convertir todas las palabras en mayúsculas y minúsculas.

src/Controller/VinylController.php

```
↕ // ... lines 1 - 8
9  class VinylController
10 {
↕ // ... lines 11 - 15
16
17     #[Route('/browse/{slug}')]
18     public function browse(string $slug): Response
19     {
20         $title = u(str_replace('-', ' ', $slug))->title(true);
21
22         return new Response('Genre: '.$title);
↕ // ... lines 23 - 24
25     }
↕ // ... lines 26 - 27
```

Ahora, cuando lo probamos... ¡qué bien! ¡Pone las letras en mayúsculas! El componente de la cadena no es especialmente importante, sólo quiero que veas cómo podemos aprovechar partes de Symfony para hacer nuestro trabajo.

Hacer que el comodín sea opcional

Bien: un último reto. Ir a `/browse/emo` o `/browse/death-metal` funciona. Pero ir a `/browse`... no funciona. ¡Está roto! Un comodín puede coincidir con cualquier cosa, pero, por defecto, se requiere un comodín. Tenemos que ir a `/browse/<something>`.

¿Podemos hacer que el comodín sea opcional? Por supuesto Y es deliciosamente sencillo: haz que el argumento correspondiente sea opcional.

src/Controller/VinylController.php

```
↕ // ... lines 1 - 8
9  class VinylController
10 {
↕ // ... lines 11 - 15
16
17     #[Route('/browse/{slug}')]
18     public function browse(string $slug = null): Response
19     {
↕ // ... lines 20 - 24
25     }
↕ // ... lines 26 - 27
```

En cuanto lo hagamos, le dirá a la capa de enrutamiento de Symfony que no es necesario que el `{slug}` esté en la URL. Así que ahora cuando refrescamos... funciona. Aunque no es un buen mensaje para la página.

Veamos. Si hay un slug, pon el título como estábamos. Si no, pon `$title` a "Todos los géneros". Ah, y mueve el "Género:" aquí arriba... para que abajo en el `Response` podamos pasar simplemente `$title`.

src/Controller/VinylController.php

```
↕ // ... lines 1 - 8
9  class VinylController
10 {
↕ // ... lines 11 - 15
16
17     #[Route('/browse/{slug}')]
18     public function browse(string $slug = null): Response
19     {
20         if ($slug) {
21             $title = 'Genre: '.u(str_replace('-', ' ', $slug))-
   >title(true);
22         } else {
23             $title = 'All Genres';
24         }
25
26         return new Response($title);
↕ // ... lines 27 - 28
29     }
↕ // ... lines 30 - 31
```

Inténtalo. En `/browse`... "Todos los géneros". En `/browse/emo`... "Género: Emo".

Siguiente: poner un texto como éste en un controlador.... no es muy limpio ni escalable, especialmente si empezamos a incluir HTML. No, tenemos que hacer una plantilla. Para ello, vamos a instalar nuestro primer paquete de terceros y seremos testigos del importantísimo sistema de recetas de Symfony en acción.

Chapter 5: Symfony Flex: Aliases, Paquetes y Recetas

Symfony es un conjunto de librerías que nos proporciona toneladas de herramientas: herramientas para registrar, hacer consultas a la base de datos, enviar correos electrónicos, renderizar plantillas y hacer llamadas a la API, por nombrar algunas. Si las cuentas, como hice yo, Symfony consta de unas 100 bibliotecas distintas. ¡Vaya!

Ahora quiero empezar a convertir nuestras páginas en verdaderas páginas HTML... en lugar de devolver sólo texto. Pero no vamos a meter un montón de HTML en nuestras clases de PHP, qué asco. En su lugar, vamos a renderizar una plantilla.

La filosofía de Symfony de empezar poco a poco e instalar funciones

Pero, ¿adivina qué? ¡No hay ninguna biblioteca de plantillas en nuestro proyecto! ¿Qué? Pero yo creía que acababas de decir que Symfony tiene una herramienta para renderizar plantillas!? ¡Mentira!

Bueno... Symfony sí tiene una herramienta para eso. Pero nuestra aplicación utiliza actualmente muy pocas de las bibliotecas de Symfony. Las herramientas que tenemos hasta ahora no suponen mucho más que un sistema de ruta-controlador-respuesta. Si necesitas renderizar una plantilla o hacer una consulta a la base de datos, no tenemos esas herramientas instaladas en nuestra app... todavía.

De hecho, me encanta esto de Symfony. En lugar de empezar con un proyecto gigantesco, con todo lo que necesitamos, más toneladas de cosas que no necesitamos, Symfony empieza de forma diminuta. Luego, si necesitas algo, lo instalas

Pero antes de instalar una biblioteca de plantillas, en tu terminal, ejecuta

```
git status
```

Vamos a confirmar todo:

```
git add .
```

Puedo ejecutar con seguridad `git add .` -que añade todo lo que hay en mi directorio a git- porque uno de los archivos con los que venía nuestro proyecto originalmente era un archivo `.gitignore`, que ya ignora cosas como el directorio `vendor/`, el directorio `var/` y varias otras rutas. Si te preguntas qué son estas cosas raras de los marcadores, está relacionado con el sistema de recetas, del que vamos a hablar.

En cualquier caso, ejecuta `git commit` y añade un mensaje:

```
git commit -m "route -> controller -> response -> profit"
```

¡Perfecto! Y ahora, estamos limpios.

Instalar una biblioteca de plantillas (Twig)

Bien, ¿cómo podemos instalar una biblioteca de plantillas? ¿Y qué bibliotecas de plantillas están disponibles para Symfony? ¿Y cuál es la recomendada? Bueno, por supuesto, una buena manera de responder a estas preguntas sería consultar la documentación de Symfony.

Pero también podemos simplemente... ¡adivinar! En cualquier proyecto PHP, puedes añadir nuevas bibliotecas de terceros a tu aplicación diciendo "composer require" y luego el nombre del paquete. Todavía no sabemos el nombre del paquete que necesitamos, así que simplemente lo adivinaremos:

```
composer require templates
```

Ahora bien, si has utilizado Composer antes, puede que ahora mismo estés gritando a tu pantalla ¿Por qué? Porque en Composer, los nombres de los paquetes son siempre

`something/something`. No es posible, literalmente, tener un paquete llamado simplemente `templates`.

Pero mira: cuando ejecutamos esto, ¡funciona! Y arriba dice que está usando la versión 1 para `symfony/twig-pack`. Twig es el nombre del motor de plantillas de Symfony.

Alias de Flex

Para entender esto, vamos a dar un pequeño paso atrás. Nuestro proyecto comenzó con un archivo `composer.json` que contiene varias bibliotecas de Symfony. Una de ellas se llama `symfony/flex`. Flex es un plugin de Composer. En realidad, añade tres superpoderes a Composer.

> **Tip**
>
> El servidor flex.symfony.com se cerró a favor de un nuevo sistema. ¡Pero aún puede ver una lista de todas las recetas disponibles en ¡ https://bit.ly/flex-recipes!

El primero, que acabamos de ver, se llama aliases de Flex. Dirígete a https://flex.symfony.com para ver una página gigante llena de paquetes. Busca "plantillas". Aquí está. En `symfony/twig-pack`, dice Aliases: template, templates, twig y twig-pack.

La idea que hay detrás de los alias de Flex es muy sencilla. Escribimos `composer require templates`. Y luego, internamente, Flex lo cambia por `symfony/twig-pack`. En última instancia, ése es el paquete que Composer instala.

Esto significa que, la mayoría de las veces, puedes simplemente "composer require" lo que quieras, como `composer require logger`, `composer require orm`, `composer require icecream`, lo que sea. Es sólo un sistema de acceso directo. Lo importante es que, lo que realmente se instaló fue `symfony/twig-pack`.

Paquetes Flex

Y eso significa que, en nuestro archivo `composer.json`, deberíamos ver ahora `symfony/twig-pack` bajo la clave `require`. Pero si te das la vuelta, ¡no está ahí!

¡Gracias! En su lugar, ha añadido `symfony/twig-bundle`, `twig/extra-bundle`, y `twig/twig`.

Estamos asistiendo al segundo superpoder de Symfony Flex: desempaquetar paquetes. Copiamos el nombre del paquete original y... podemos encontrar ese repositorio en GitHub entrando en https://github.com/symfony/twig-pack.

Y... sólo contiene un archivo: `composer.json`. Y esto requiere otros tres paquetes: los tres que acabamos de ver añadidos a nuestro proyecto.

Esto se llama paquete Symfony. Es... realmente un paquete falso que nos ayuda a instalar otros paquetes. Resulta que, si quieres añadir un motor de plantillas rico a tu aplicación, es recomendable instalar estos tres paquetes. Pero en lugar de hacer que los añadas manualmente, puedes hacer que Composer requiera `symfony/twig-pack` y los obtenga automáticamente. Cuando instalas un "paquete", como éste, Flex lo "desempaqueta" automáticamente: encuentra los tres paquetes de los que depende el paquete y los añade a tu archivo `composer.json`.

Así pues, los paquetes son un atajo para que puedas ejecutar un comando de `composer require` y conseguir que se añadan varias bibliotecas a tu proyecto.

Bien, ¿cuál es el tercer y último superpoder de Flex? Me alegro de que lo preguntes Para averiguarlo, en tu terminal, ejecuta

```
git status
```

Recetas de Flex

Vaya. Normalmente, cuando ejecutas `composer require`, los únicos archivos que debería modificar -además de descargar paquetes en `vendor/` - son `composer.json` y`composer.lock`. El tercer superpoder de Flex es su sistema de recetas.

Siempre que instales un paquete, ese paquete puede tener una receta. Si la tiene, además de descargar el paquete en el directorio `vendor/`, Flex también ejecutará su receta. Las recetas pueden hacer cosas como añadir nuevos archivos o incluso modificar algunos archivos existentes.

Observa: si nos desplazamos un poco hacia arriba, ah sí: dice "configurando 2 recetas". Así que aparentemente había una receta para `symfony/twig-bundle` y también una receta para `twig/extra-bundle`. Y estas recetas aparentemente actualizaron el archivo `config/bundles.php` y añadieron un nuevo directorio y archivo.

El sistema de recetas es genial. Todo lo que tenemos que hacer es que Composer requiera una nueva biblioteca y su receta añadirá todos los archivos de configuración u otra configuración necesaria para que podamos empezar a usar esa biblioteca inmediatamente Se acabó el seguir 5 pasos de "instalación" manual en un README. Cuando añades una biblioteca, funciona de forma inmediata.

A continuación: Quiero profundizar un poco más en las recetas. Por ejemplo, ¿dónde viven? ¿Cuál es su color favorito? ¿Y qué ha añadido esta receta específicamente a nuestra aplicación y por qué? También voy a contarte un pequeño secreto: todos los archivos de nuestro proyecto -todos los archivos de `config/`, el directorio `public/`... todas estas cosas- se añadieron mediante una receta. Y lo demostraré.

Chapter 6: Recetas Flex

Acabamos de instalar un nuevo paquete ejecutando `composer require templates`. Normalmente, al hacerlo, Composer actualizará los archivos `composer.json` y `composer.lock`, pero nada más.

Pero cuando ejecutamos

```
git status
```

Hay otros cambios. Esto es gracias al sistema de recetas de Flex. Cada vez que instalamos un nuevo paquete, Flex comprueba en un repositorio central si ese paquete tiene una receta. Y si la tiene, la instala.

¿Dónde viven las recetas?

¿Dónde viven estas recetas? En la nube... o más concretamente en GitHub. Compruébalo. Ejecutar:

```
composer recipes
```

Este es un comando añadido a Composer por Flex. Enumera todas las recetas que se han instalado. Y si quieres más información sobre una, ejecútala:

```
composer recipes symfony/twig-bundle
```

Esta es una de las recetas que se acaba de ejecutar. Y... ¡guay! ¡Nos muestra un par de cosas bonitas! La primera es un árbol de los archivos que ha añadido a nuestro proyecto. La segunda es una URL de la receta que se instaló. Haré clic para abrirla.

¡Sí! Las recetas de Symfony viven en un repositorio especial llamado `symfony/recipes`. Se trata de un gran directorio organizado por nombre de paquete. Hay un directorio `symfony` que contiene las recetas de todos los paquetes que empiezan por `symfony/`. El que acabamos de ver... está aquí abajo: `twig-bundle`. Y luego hay diferentes versiones de la receta en función de tu versión del paquete. Nosotros estamos utilizando la última versión 5.4.

Cada receta tiene un archivo `manifest.json`, que controla lo que hace. El sistema de recetas sólo puede realizar un conjunto específico de operaciones, como añadir nuevos archivos a tu proyecto y modificar algunos archivos concretos. Por ejemplo, esta sección `bundles` le dice a flex que añada esta línea a nuestro archivo `config/bundles.php`.

Si volvemos a ejecutar `git status`... ¡sí! Ese archivo ha sido modificado. Si lo difundimos:

```
git diff config/bundles.php
```

Ha añadido dos líneas, probablemente una para cada una de las dos recetas.

¿Bolsos Symfony?

Por cierto, `config/bundles.php` no es un archivo en el que tengas que pensar mucho. Un bundle, en la tierra de Symfony, es básicamente un plugin. Así que si instalas un nuevo bundle en tu aplicación, eso te da nuevas características de Symfony. Para activar ese bundle, su nombre tiene que estar en este archivo.

Así que lo primero que hizo la receta para Twig-bundle, gracias a esta línea de aquí arriba, fue activarse dentro de `bundles.php`... para que no tuviéramos que hacerlo manualmente. Las recetas son como una instalación automática.

Archivos nuevos y copiados

La segunda sección del manifiesto se llama `copy-from-recipe`. Es sencillo: dice que hay que copiar los directorios `config/` y `templates/` de la receta en el proyecto. Si nos fijamos... la receta contiene un archivo `config/packages/twig.yaml`... y también un archivo `templates/base.html.twig`.

De vuelta al terminal, ejecuta de nuevo `git status`. Vemos estos dos archivos en la parte inferior: `config/packages/twig.yaml`... y dentro de `templates/`, `base.html.twig`.

Esto me encanta. Piénsalo: si instalas una herramienta de plantillas en tu aplicación, vamos a necesitar alguna configuración en algún lugar que le diga a esa herramienta de plantillas en qué directorio debe buscar nuestras plantillas. Ve a ver ese archivo `config/packages/twig.yaml`. Hablaremos más de estos archivos Yaml en el próximo tutorial. Pero a alto nivel, este archivo controla cómo se comporta Twig, el motor de plantillas de Symfony. Y fíjate en la clave `default_path` establecida en `%kernel.project_dir%/templates`. No te preocupes por esta sintaxis porcentual: es una forma elegante de referirse a la raíz de nuestro proyecto.

La cuestión es que esta configuración dice

> *"¡Hey Twig! Cuando busques plantillas, búscalas en el directorio `templates/`."*

Y la receta incluso ha creado ese directorio con un archivo de diseño dentro. Lo usaremos en unos minutos.

symfony.lock y el compromiso de los archivos

El último archivo no explicado que se ha modificado es `symfony.lock`. Esto no es importante: sólo mantiene un registro de las recetas que se han instalado... y deberías confirmarlo.

De hecho, deberíamos confirmar todo esto. La receta puede darnos archivos, pero luego son nuestros para modificarlos. Ejecuta:

```
git add .
```

Entonces:

```
git status
```

Genial. ¡Vamos a confirmarlo!

```
git commit -m "Adding Twig and its beautiful recipe"
```

Actualizar las recetas

¡Ya está! Por cierto, es posible que dentro de unos meses haya cambios en algunas de las recetas que has instalado. Y si los hay, cuando ejecutes

```
composer recipes
```

verás un pequeño "actualización disponible" junto a ellas. Ejecuta `composer recipes:update` para actualizar a la última versión.

Ah, y antes de que se me olvide, además de `symfony/recipes`, también hay un repositorio `symfony/recipes-contrib`. Así que las recetas pueden vivir en cualquiera de estos dos lugares. Las recetas de `symfony/recipes` están aprobadas por el equipo central de Symfony, por lo que su calidad está un poco más controlada. Aparte de eso, no hay ninguna diferencia.

Nuestro proyecto comenzó como un archivo

Ahora, el sistema de recetas es tan potente que cada uno de los archivos de nuestro proyecto se añadió mediante una receta Puedo demostrarlo. Ve a https://github.com/symfony/skeleton.

Cuando ejecutamos originalmente ese comando `symfony new` para iniciar nuestro proyecto, lo que realmente hizo fue clonar este repositorio... y luego ejecutó `composer install` dentro de él, que descarga todo en el directorio `vendor/`.

Sí Nuestro proyecto -el que vemos aquí- era originalmente un único archivo: `composer.json`. Pero luego, cuando se instalaron los paquetes, las recetas de esos paquetes añadieron todo lo que vemos. Ejecuta:

```
composer recipes
```

de nuevo. Una de las recetas es para algo llamado `symfony/console`. Comprueba sus detalles:

```
composer recipes symfony/console
```

Y... ¡sí! ¡La receta de `symfony/console` añadió el archivo `bin/console`! La receta de `symfony/framework-bundle` -uno de los otros paquetes que se instaló originalmente- añadió casi todo lo demás, incluido el archivo `public/index.php`. ¿No es genial?

Bien, a continuación: ¡hemos instalado Twig! ¡Así que volvamos al trabajo y utilicémoslo para renderizar algunas plantillas! Te va a encantar Twig.

Chapter 7: Twig ❤️

Las clases de controlador de Symfony no necesitan extender una clase base. Mientras tu función de controlador devuelva un objeto `Response`, a Symfony no le importa el aspecto de tu controlador. Pero normalmente, extenderás una clase llamada `AbstractController`.

¿Por qué? Porque nos da métodos de acceso directo.

Renderización de una plantilla

Y el primer atajo es `render()`: el método para renderizar una plantilla. Así que devuelve `$this->render()` y le pasa dos cosas. La primera es el nombre de la plantilla. ¿Qué tal `vinyl/homepage.html.twig`.

No es necesario, pero es habitual tener un directorio con el mismo nombre que la clase de tu controlador y un nombre de archivo que sea el mismo que el de tu método, pero puedes hacer lo que quieras. El segundo argumento es un array con las variables que quieras pasar a la plantilla. Vamos a pasar una variable llamada `title` y a ponerle el título de nuestra cinta de mezclas: "PB and Jams".

src/Controller/VinylController.php

```
↕ // ... lines 1 - 4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
↕ // ... lines 6 - 8
9
10 class VinylController extends AbstractController
11 {
12     #[Route('/')]
13     public function homepage(): Response
14     {
15         return $this->render('vinyl/homepage.html.twig', [
16             'title' => 'PB & Jams',
17         ]);
18     }
↕ // ... lines 19 - 34
```

Hecho aquí. Ah, pero, ¡examen sorpresa! ¿Qué crees que devuelve el método `render()`? Sí, es lo que siempre repito: un controlador siempre debe devolver un objeto `Response`. `render()` es sólo un atajo para renderizar una plantilla, obtener esa cadena y ponerla en un objeto `Response`. `render()` devuelve un objeto `Response`.

Crear la plantilla

Sabemos por lo que hemos dicho antes, que cuando renderizas una plantilla, Twig busca en el directorio `templates/`. Así que crea un nuevo subdirectorio `vinyl/`... y dentro de él, un archivo llamado `homepage.html.twig`. Para empezar, añade un `h1` y luego imprime la variable `title` con una sintaxis especial de Twig: `{{ title }}`. Y... Añadiré un texto TODO codificado.

templates/vinyl/homepage.html.twig

```
1  <h1>{{ title }}</h1>
2
3  {# TODO: add an image of the record #}
4
5  <div>
6      Our schweet track list: TODO
7  </div>
```

¡Vamos a ver si esto funciona! Estábamos trabajando en nuestra página web, así que ve allí y... ¡hola Twig!

Sintaxis de Twigs 3

Twig es una de las partes más bonitas de Symfony, y también una de las más fáciles. Vamos a repasar todo lo que necesitas saber... básicamente en los próximos diez minutos.

Twig tiene exactamente tres sintaxis diferentes. Si necesitas imprimir algo, utiliza `{{`. A esto lo llamo la sintaxis "decir algo". Si digo `{{ saySomething }}` se imprimiría una variable llamada `saySomething`. Una vez que estás dentro de Twig, se parece mucho a JavaScript. Por ejemplo, si lo encierro entre comillas, ahora estoy imprimiendo la cadena `saySomething`. Twig tiene funciones... por lo que llamaría a la función e imprimiría el resultado.

Así que la sintaxis nº 1 -la de "decir algo"- es `{{`

La segunda sintaxis... no cuenta realmente. Es `{#` para crear un comentario... y ya está.

templates/vinyl/homepage.html.twig

```
<h1>{{ title }}</h1>

{# TODO: add an image of the record #}

<div>
    Our schweet track list: TODO
</div>
```

La tercera y última sintaxis la llamo "hacer algo". Esto es cuando no estás imprimiendo, estás haciendo algo en el lenguaje. Ejemplos de "hacer algo" serían las sentencias if, los bucles for o la configuración de variables.

El bucle for

Vamos a probar un bucle `for`. Vuelve al controlador. Voy a pegar una lista de pistas... y luego pasaré una variable `tracks` a la plantilla ajustada a esa matriz.

src/Controller/VinylController.php

```php
<?php
// ... lines 2 - 8

class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $tracks = [
            'Gangsta\'s Paradise - Coolio',
            'Waterfalls - TLC',
            'Creep - Radiohead',
            'Kiss from a Rose - Seal',
            'On Bended Knee - Boyz II Men',
            'Fantasy - Mariah Carey',
        ];

        return $this->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
            'tracks' => $tracks,
        ]);
    }
// ... lines 29 - 42
}
```

Ahora, a diferencia de `title`, tracks es una matriz... así que no podemos imprimirla. Pero, ¡podemos intentarlo! ¡Ja! Eso nos da una conversión de matriz a cadena. No, tenemos que hacer un bucle sobre las pistas.

Añade una cabecera y un `ul`. Para hacer el bucle, usaremos la sintaxis "hacer algo", que es `{%` y luego la cosa que quieras hacer, como `for`, `if` o `set`. Te mostraré la lista completa de etiquetas de hacer algo en un minuto. Un bucle for tiene este aspecto: `for track in tracks`, donde pistas es la variable sobre la que hacemos el bucle y `track` será la variable dentro del bucle.

Después de esto, añade `{% endfor %}`: la mayoría de las etiquetas "hacer algo" tienen una etiqueta de fin. Dentro del bucle, añade un `li` y luego utiliza la sintaxis de decir algo para imprimir `track`.

templates/vinyl/homepage.html.twig

```
<h1>{{ title }}</h1>

{# TODO: add an image of the record #}

<div>
    Tracks:

    <ul>
        {% for track in tracks %}
            <li>
                {{ track }}
            </li>
        {% endfor %}
    </ul>
</div>
```

Uso de Sub.keys

Cuando lo probemos... ¡qué bien! Pero vamos a ponernos más complicados. De vuelta en el controlador, en lugar de utilizar un simple array, lo reestructuraré para que cada pista sea un array asociativo con las claves `song` y `artist`. Pondré ese mismo cambio para el resto.

src/Controller/VinylController.php

```
<?php
// ... lines 2 - 8

class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
        $tracks = [
            ['song' => 'Gangsta\'s Paradise', 'artist' => 'Coolio'],
            ['song' => 'Waterfalls', 'artist' => 'TLC'],
            ['song' => 'Creep', 'artist' => 'Radiohead'],
            ['song' => 'Kiss from a Rose', 'artist' => 'Seal'],
            ['song' => 'On Bended Knee', 'artist' => 'Boyz II Men'],
            ['song' => 'Fantasy', 'artist' => 'Mariah Carey'],
        ];
// ... lines 23 - 27
    }
// ... lines 29 - 42
}
```

¿Qué ocurre si lo probamos? Ah, volvemos a la conversión de "matriz a cadena". Cuando hacemos el bucle, cada pista es ahora una matriz. ¿Cómo podemos leer las claves `song` y `artist`?

¿Recuerdas cuando dije que Twig se parece mucho a JavaScript? Pues bien, no debería sorprender que la respuesta sea `track.song` y `track.artist`.

templates/vinyl/homepage.html.twig

```
// ... lines 1 - 7
    <ul>
        {% for track in tracks %}
            <li>
                {{ track.song }} - {{ track.artist }}
            </li>
        {% endfor %}
    </ul>
// ... lines 15 - 16
```

Y... eso hace que nuestra lista funcione.

Ahora que ya tenemos lo básico de Twig, vamos a ver la lista completa de etiquetas "hacer algo", a conocer los "filtros" de Twig y a abordar el importantísimo sistema de herencia de plantillas.

Chapter 8: Herencia Twig

Dirígete a https://twig.symfony.com... y haz clic para consultar su documentación. Hay mucho material bueno aquí. Pero lo que quiero que hagas es que te desplaces hasta la referencia a Twig. ¡Sí!

Etiquetas

Lo primero que debes mirar, a la izquierda, son estas cosas llamadas etiquetas. Esta lista representa todas las cosas posibles que puedes utilizar con la sintaxis de hacer algo. Sí, siempre será `{%` y luego una de estas cosas, como `for` o `if`. Y sinceramente, sólo vas a utilizar unas 5 de ellas en el día a día. Si quieres saber la sintaxis de uno de ellos, sólo tienes que hacer clic para ver su documentación.

Filtros

Además de las 20 etiquetas, Twig también tiene algo llamado filtros. Los filtros son básicamente funciones, pero con una sintaxis más moderna. Twig también tiene funciones, pero son menos: Twig prefiere los filtros: ¡son mucho más chulos!

Por ejemplo, hay un filtro llamado `upper`. Usar un filtro es como usar la tecla `|` en la línea de comandos. Tienes un valor y luego lo "canalizas" en el filtro que quieres, como `upper`.

¡Vamos a probar esto! Imprime `track.artist|upper`.

templates/vinyl/homepage.html.twig

```
↕ // ... lines 1 - 10
11                  {{ track.song }} - {{ track.artist|upper }}
↕ // ... lines 12 - 16
```

Y ahora... ¡está en mayúsculas! Si quieres confundir a tus compañeros de trabajo, puedes canalizarlo a `lower`... que devuelve las cosas a minúsculas. No hay ninguna razón real para hacer esto, pero los filtros pueden encadenarse así.

templates/vinyl/homepage.html.twig

```
// ... lines 1 - 10
                {{ track.song }} - {{ track.artist|upper|lower }}
// ... lines 12 - 16
```

De todos modos, echa un vistazo a la lista de filtros porque probablemente haya algo que te resulte útil.

Y... ¡eso es todo! Además de las funciones, también hay algo llamado "pruebas", que son útiles en las sentencias if: puedes decir cosas como "si el número es divisible por 5".

Herencia de Plantillas

Vale, sólo una cosa más que aprender sobre Twig, y es genial.

Mira el código fuente HTML de la página. Fíjate en que no hay estructura HTML: no hay etiquetas `html`, `head` o `body`. Literalmente el HTML que tenemos dentro de nuestra plantilla, es lo que obtenemos. Nada más.

Entonces, ¿hay... algún tipo de sistema de diseño en Twig en el que podamos añadir un diseño base a nuestro alrededor? Por supuesto. Y es increíble. Se llama herencia de plantillas. Si tienes una plantilla y quieres que utilice algún diseño base, en la parte superior del archivo, utiliza una etiqueta "hacer algo" llamada `extends`. Pásale el nombre del archivo de diseño: `base.html.twig`.

templates/vinyl/homepage.html.twig

```
{% extends 'base.html.twig' %}
// ... lines 2 - 18
```

Esto se refiere a esta plantilla de aquí. Antes de comprobarlo, si lo intentamos ahora, ¡vaya! Gran error:

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

Para saber qué significa esto, abre `base.html.twig`. Este es tu archivo de diseño base... y eres totalmente libre de personalizarlo como quieras. Ahora mismo... es en su mayor parte sólo etiquetas HTML aburridas... excepto por una serie de estos "bloques".

Los bloques son básicamente "agujeros" en los que una plantilla hija puede colocar contenido. Permíteme explicarlo de otra manera. Cuando decimos `extends 'base.html.twig'`, eso dice básicamente:

> *"¡Yo Twig! Cuando renderices esta plantilla, quiero que realmente renderices `base.html.twig`... y luego pongas mi contenido dentro de ella."*

Twig responde educadamente:

> *"Vale, genial... Puedo hacerlo. Pero, ¿en qué parte de `base.html.twig` quieres que ponga todo tu contenido? ¿Quieres que lo ponga al final de la página? ¿En la parte parte superior? ¿En algún lugar al azar en el medio?"*

La forma de decirle a Twig dónde poner nuestro contenido dentro de `base.html.twig` es anulando un bloque. Fíjate en que `base.html.twig` ya tiene un bloque llamado `body`... y ahí es justo donde queremos poner el HTML de nuestra plantilla.

Para ponerlo ahí, en nuestra plantilla, rodea todo el contenido con `{% block body %}`... y luego `{% endblock %}`.

templates/vinyl/homepage.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4  <h1>{{ title }}</h1>
5
6  {# TODO: add an image of the record #}
7
8  <div>
9      Tracks:
10
11     <ul>
12         {% for track in tracks %}
13             <li>
14                 {{ track.song }} - {{ track.artist }}
15             </li>
16         {% endfor %}
17     </ul>
18 </div>
19 {% endblock %}
```

A esto se le llama herencia de la plantilla porque estamos sobrescribiendo ese bloque `body` con este nuevo contenido. Así que ahora, cuando Twig renderice `base.html.twig`... y llegue a esta parte `block body`, va a imprimir el HTML `block body` de nuestra plantilla

Observa: actualiza y... el error ha desaparecido. Y si ves el código fuente de la página, ¡tenemos una página HTML completa!

Nombres de los bloques

Ah, y los nombres de estos bloques no son importantes. Si quieres cambiarles el nombre por el de tu personaje favorito de una sitcom de los 90, hazlo. Sólo recuerda actualizar también su nombre en cualquier plantilla hija.

También puedes añadir más bloques. Cada bloque que añadas es otro punto de anulación potencial.

Contenido del bloque por defecto

Ah, y habrás notado que los bloques pueden tener contenido por defecto. Mira la página ahora mismo: el título dice "Bienvenido". Eso es porque el bloque `title` tiene un contenido por defecto... y no lo vamos a anular. Vamos a cambiar el título por defecto a "Vinilo mixto".

templates/base.html.twig

```
↕ // ... lines 1 - 4
5         <title>{% block title %}Mixed Vinyl{% endblock %}</title>
↕ // ... lines 6 - 20
```

Así que ahora ese será el título de todas las páginas de nuestro sitio... a menos que lo anulemos. En nuestra plantilla, ya sea por encima del cuerpo del bloque o por debajo -el orden de los bloques no importa-, añade `{% block title %}`, `{% endblock %}` y, en medio, "Crear un nuevo disco".

templates/vinyl/homepage.html.twig

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

{% block title %}Create a new Record{% endblock %}

{% block body %}
// ... lines 6 - 20
{% endblock %}
```

Y ahora... ¡sí! Esta página tiene un título personalizado.

Añadir al bloque padre (en lugar de sustituirlo)

Ah, y puede que te preguntes

> *"¿Qué pasa si no quiero sustituir un bloque por completo.... sino que quiero añadir a un bloque?"*

Eso es totalmente posible. En `base.html.twig`, el bloque `title` está configurado como "Vinilo mixto". Si quisiéramos añadirle nuestro título personalizado, podríamos decir "Crear un nuevo disco" y luego utilizar la etiqueta "decir algo" para imprimir una función llamada `parent()`.

templates/vinyl/homepage.html.twig

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

{% block title %}Create a new Record | {{ parent() }}{% endblock %}

{% block body %}
// ... lines 6 - 20
{% endblock %}
```

Eso hace exactamente lo que esperarías: encuentra el contenido de la plantilla padre para este bloque... y lo imprime. Actualiza y... eso es muy bonito.

La herencia de plantillas es la herencia de clases

Si alguna vez estás confundido sobre cómo funciona la herencia de plantillas, es útil, al menos para mí, pensar en ella exactamente como en la herencia orientada a objetos. Cada plantilla es como una clase y cada bloque es como un método. Así, la "clase" de la página de inicio

extiende la "clase" de `base.html.twig`, pero anula dos de sus métodos. Si eso sólo te ha confundido, no te preocupes.

Así que... eso es todo para Twig. Básicamente eres un experto en Twig, lo que me han dicho que es un tema popular en las fiestas.

A continuación: una de las características más destacadas de Symfony son sus herramientas de depuración. Vamos a instalarlas y a comprobarlas.

Chapter 9: Perfilador: Tu mejor amigo para la depuración

Es hora de instalar nuestro segundo paquete. Y éste es divertido. Vamos a confirmar nuestros cambios primero: así será más fácil comprobar los cambios que hace la receta del nuevo paquete.

Añade todo:

```
git add .
```

Parece que está bien, así que... confirma:

```
git commit -m "Added some Tiwggy goodness"
```

Bonito.

El paquete de depuración

Ahora ejecuta:

```
composer require debug
```

Así que sí, este es otro alias de Flex... y aparentemente es un alias de `symfony/debug-pack`. Y sabemos que un paquete es una colección de paquetes. Así que, en lugar de añadir esta única línea a nuestro archivo `composer.json`, si lo comprobamos, parece que ha añadido un nuevo paquete en la sección `require` -se trata de

una biblioteca de registro- y... al final, ha añadido una nueva sección `require-dev` con otras tres bibliotecas.

La diferencia entre `require` y `require-dev` no es demasiado importante: todos estos paquetes se descargaron en nuestra aplicación, pero como mejor práctica, si instalas una biblioteca que sólo está pensada para el desarrollo local, deberías ponerla en `require-dev`. ¡El pack lo hizo por nosotros! ¡Gracias pack!

Cambios en la receta

De vuelta al terminal, ¡esto también instaló tres recetas! Ooh. Veamos qué han hecho. Limpio la pantalla y corro:

```
git status
```

Esto me resulta familiar: modificó `config/bundles.php` para activar tres nuevos bundles. De nuevo, los bundles son plugins de Symfony, que añaden más funciones a nuestra aplicación.

También añadió varios archivos de configuración al directorio `config/packages/`. Hablaremos más de estos archivos en el próximo tutorial, pero, a alto nivel, controlan el comportamiento de esos bundles.

La barra de herramientas de depuración web y el perfilador

¿Qué nos aportan estos nuevos paquetes? Para averiguarlo, dirígete a tu navegador y actualiza la página de inicio. ¡Santo cielo, Batman! Es la barra de herramientas de depuración web. Esto es una locura de depuración: una barra de herramientas llena de buena información. A la izquierda, puedes ver el controlador al que se ha llamado junto con el código de estado HTTP. También está la cantidad de tiempo que tardó la página, la memoria que utilizó y también cuántas plantillas se renderizaron a través de Twig: este es el bonito icono de Twig.

En el lado derecho, tenemos detalles sobre el servidor web local Symfony que se está ejecutando e información sobre PHP.

Pero aún no has visto la mejor parte: haz clic en cualquiera de estos iconos para saltar al perfilador. Esta es la barra de herramientas de depuración web... enloquecida. Está llena de datos sobre esa petición, como la petición y la respuesta, todos los mensajes de registro que se produjeron durante esa petición, información sobre las rutas y la ruta a la que se respondió, Twig te muestra qué plantillas se renderizaron y cuántas veces se renderizaron... y hay información de configuración aquí abajo. ¡Uf!

Pero mi sección favorita es la de Rendimiento. Muestra una línea de tiempo de todo lo que ha ocurrido durante la petición. Esto es genial por dos razones. La primera es bastante obvia: puedes usarla para encontrar qué partes de tu página son lentas. Así, por ejemplo, nuestro controlador tardó 20,4 milisegundos. Y dentro de la ejecución del controlador, la plantilla de la página de inicio se renderizó en 3,9 milisegundos y `base.html.twig` se renderizó en 2,8 milisegundos.

La segunda razón por la que esto es realmente genial es que descubre todas las capas ocultas de Symfony. Ajusta este umbral a cero. Antes, esto sólo mostraba las cosas que tardaban más de un milisegundo. Ahora lo muestra todo. No tienes que preocuparte por la gran mayoría de las cosas, pero es superguay ver las capas de Symfony: las cosas que ocurren antes y después de que se ejecute tu controlador. Tenemos un tutorial de inmersión profunda para Symfony si quieres aprender más sobre estas cosas.

La barra de herramientas de depuración web y el perfilador también crecerán con nuestra aplicación. En un futuro tutorial, cuando instalemos una librería para hablar con la base de datos, de repente tendremos una nueva sección que enumera todas las consultas a la base de datos que hizo una página y el tiempo que tardó cada una.

funciones dump() y dd()

Bien, el paquete de depuración instaló la barra de herramientas de depuración web. También ha instalado una biblioteca de registro que utilizaremos más adelante. Y ha instalado un paquete que nos proporciona dos fantásticas funciones de depuración.

Dirígete a `VinylController`. Imagina que estamos haciendo un desarrollo y necesitamos ver cómo es esta variable `$tracks`. En este caso es bastante obvio, pero a veces querrás ver lo que hay dentro de un objeto complejo.

Para ello, digamos `dd($tracks)`, donde "dd" significa "dump" y "die".

src/Controller/VinylController.php

```
// ... lines 1 - 9
class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
// ... lines 15 - 22
        dd($tracks);
// ... lines 24 - 28
    }
// ... lines 30 - 43
}
```

Así que si refrescamos... ¡sí! Eso vuelca la variable y mata la página. Y esto es mucho más potente -y más bonito- que usar `var_dump()`: podemos ampliar secciones y ver datos profundos con mucha facilidad.

En lugar de `dd()`, también puedes utilizar `dump()`.. para volcar y vivir. Pero esto podría no aparecer donde esperas. En lugar de imprimirse en el centro de la página, aparece abajo en la barra de herramientas de depuración de la web, bajo el icono del objetivo.

src/Controller/VinylController.php

```
// ... lines 1 - 9
class VinylController extends AbstractController
{
    #[Route('/')]
    public function homepage(): Response
    {
// ... lines 15 - 22
        dump($tracks);
// ... lines 24 - 28
    }
// ... lines 30 - 43
}
```

Si es demasiado pequeño, haz clic para ver una versión más grande en el perfilador.

Volcado en Twig

También puedes utilizar este `dump()` en Twig. Elimina el volcado del controlador... y luego en la plantilla, justo antes del `ul`, `dump(tracks)`.

templates/vinyl/homepage.html.twig

```
↕ // ... lines 1 - 9
10 <div>
11     Tracks:
12
13     {{ dump(tracks) }}
↕ // ... lines 14 - 20
21 </div>
↕ // ... lines 22 - 23
```

Y esto... se ve exactamente igual. Excepto que cuando haces el volcado en Twig, sí que se vuelca justo en el centro de la página

Y aún más útil, sólo en Twig, puedes utilizar `dump()` sin argumentos.

templates/vinyl/homepage.html.twig

```
↕ // ... lines 1 - 9
10 <div>
11     Tracks:
12
13     {{ dump() }}
↕ // ... lines 14 - 20
21 </div>
↕ // ... lines 22 - 23
```

Esto volcará todas las variables a las que tengamos acceso. Así que aquí está la variable `title`, `tracks` y, ¡sorpresa! Hay una tercera variable llamada `app`. Es una variable global que tenemos en todas las plantillas... y nos da acceso a cosas como la sesión y los datos del usuario. Y... ¡lo hemos descubierto por curiosidad!

Así que ahora que tenemos estas increíbles herramientas de depuración, pasemos a nuestro siguiente trabajo... que es hacer este sitio menos feo. ¡Es hora de añadir CSS y un diseño adecuado para dar vida a nuestro sitio!

Chapter 10: Activos, CSS, imágenes, etc

Si descargas el código del curso desde la página en la que estás viendo este vídeo, después de descomprimirlo, encontrarás un directorio `start/` que contiene la misma aplicación nueva de Symfony 6 que hemos creado antes. En realidad no necesitas ese código, pero contiene un directorio extra llamado `tutorial/`, como el que tengo aquí. Este contiene algunos archivos que vamos a utilizar.

Así que hablemos de nuestro siguiente objetivo: hacer que este sitio parezca un sitio real... en lugar de parecer algo que he diseñado yo mismo. Y eso significa que necesitamos un verdadero diseño HTML que incluya algo de CSS.

Añadir un diseño y archivos CSS

Sabemos que nuestro archivo de diseño es `base.html.twig`... y también hay un archivo `base.html.twig` en el nuevo directorio `tutorial/`. Copia eso... pégalo en las plantillas, y anula el original.

Antes de ver eso, copia también los tres archivos `.png` y ponlos en el directorio `public/`... para que nuestros usuarios puedan acceder a ellos.

Muy bien. Abre el nuevo archivo `base.html.twig`. Aquí no hay nada especial. Traemos algunos archivos CSS externos de algunos CDN para Bootstrap y FontAwesome. Al final de este tutorial, refactorizaremos esto para que sea una forma más elegante de manejar el CSS... pero por ahora, esto funcionará bien.

Por lo demás, todo sigue estando codificado. Tenemos una navegación codificada, el mismo bloque `body`... y un pie de página codificado. Vamos a ver cómo queda. ¡Refresca y woo! Bueno, no es perfecto, pero es mejor

Añadir un archivo CSS personalizado

El directorio `tutorial/` también contiene un archivo `app.css` con CSS personalizado. Para que esté disponible públicamente, de modo que el navegador de nuestro usuario pueda

descargarlo, tiene que estar en algún lugar del directorio `public/`. Pero no importa dónde o cómo organices las cosas dentro.

Creemos un directorio `styles/`... y luego copiamos `app.css`... y lo pegamos allí.

De vuelta en `base.html.twig`, dirígete a la parte superior. Después de todos los archivos CSS externos, vamos a añadir una etiqueta de enlace para nuestro `app.css`. Así que `<link rel="stylesheet"` y `href=""`. Como el directorio `public/` es la raíz de nuestro documento, para referirse a un archivo CSS o de imagen allí, la ruta debe ser con respecto a ese directorio. Así que esto será `/styles/app.css`.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 15
        <link rel="stylesheet" href="/styles/app.css">
// ... lines 17 - 25
    </head>
// ... lines 27 - 85
</html>
```

Vamos a comprobarlo. Actualiza ahora y... ¡aún mejor!

La función asset()

Quiero que te des cuenta de algo. Hasta ahora, Symfony no interviene para nada en cómo organizamos o utilizamos las imágenes o los archivos CSS. No. Nuestra configuración es muy sencilla: ponemos las cosas en el directorio `public/`... y luego nos referimos a ellas con sus rutas.

Pero, ¿tiene Symfony alguna función interesante para ayudar a trabajar con CSS y JavaScript? Por supuesto. Se llaman Webpack Encore y Stimulus. Y hablaremos de ambas hacia el final del tutorial.

Pero incluso en esta sencilla configuración -en la que sólo ponemos archivos en `public/` y apuntamos a ellos- Symfony tiene una característica menor: la función `asset()`.

Funciona así: en lugar de usar `/styles/app.css`, decimos `{{ asset() }}` y luego, entre comillas, movemos nuestra ruta allí... pero sin la apertura "/".

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
// ... lines 4 - 15
        <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
// ... lines 17 - 25
    </head>
// ... lines 27 - 85
</html>
```

Así, la ruta sigue siendo relativa al directorio `public/`... sólo que no necesitas incluir el primer "/".

Antes de hablar de lo que hace esto... vamos a ver si funciona. Actualiza y... ¡no lo hace! Error:

> *"Función desconocida: ¿te has olvidado de ejecutar `composer require symfony/asset`."*

Sigo diciendo que Symfony empieza con algo pequeño... y luego vas instalando cosas a medida que las necesitas. ¡Aparentemente, esta función `asset()` viene de una parte de Symfony que aún no tenemos! Pero conseguirla es fácil. Copia este comando composer require, pásalo a tu terminal y ejecútalo:

```
composer require symfony/asset
```

Se trata de una instalación bastante sencilla: sólo descarga este paquete... y no hay recetas.

Pero cuando probamos la página ahora... ¡funciona! Comprueba el código fuente HTML. Interesante: la etiqueta `link` `href` sigue siendo, literalmente, `/styles/app.css`. ¡Es exactamente lo que teníamos antes! Entonces, ¿qué diablos hace esta función `asset()`?

La respuesta es... no mucho. Pero sigue siendo una buena idea utilizarla. La función `asset()` te ofrece dos características. En primer lugar, imagina que te despliegas en un subdirectorio de un dominio. Por ejemplo, la página de inicio vive en https://example.com/mixed-vinyl.

Si ese fuera el caso, para que nuestro CSS funcione, el `href` tendría que ser `/mixed-vinyl/styles/app.css`. En esta situación, la función `asset()` detectaría el subdirectorio automáticamente y añadiría ese prefijo por ti.

Lo segundo -y más importante- que hace la función `asset()` es permitirte cambiar fácilmente a una CDN más adelante. Como esta ruta pasa ahora por la función `asset()`, podríamos, a través de un archivo de configuración, decir:

> *"¡Hey Symfony! Cuando emitas esta ruta, por favor ponle el prefijo de la URL a mi CDN."*

Esto significa que, cuando carguemos la página, en lugar de `href="/styles/app.css`, sería algo como `https://mycdn.com/styles/app.css`.

Así que la función `asset()` puede que no haga nada que necesites hoy, pero siempre que hagas referencia a un archivo estático, ya sea un archivo CSS, un archivo JavaScript, una imagen, lo que sea, utiliza esta función.

De hecho, aquí arriba, estoy haciendo referencia a tres imágenes. Usemos `asset`: `{{ asset()` ... ¡y entonces se autocompleta la ruta! ¡Gracias plugin Symfony! Repite esto para la segunda imagen... y la tercera.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
↕ // ... lines 4 - 6
        <link rel="apple-touch-icon" sizes="180x180" href="{{ asset('apple-touch-icon.png') }}">
        <link rel="icon" type="image/png" sizes="32x32" href="{{ asset('favicon-32x32.png') }}">
        <link rel="icon" type="image/png" sizes="16x16" href="{{ asset('favicon-16x16.png') }}">
↕ // ... lines 10 - 15
        <link rel="stylesheet" href="{{ asset('styles/app.css') }}">
↕ // ... lines 17 - 25
    </head>
↕ // ... lines 27 - 85
</html>
```

Sabemos que esto no supondrá ninguna diferencia hoy... podemos refrescar el código fuente HTML para ver las mismas rutas... pero estamos preparados para una CDN en el futuro.

HTML de la página de inicio y de navegación

¡Así que el diseño ahora se ve muy bien! Pero el contenido de nuestra página de inicio está... como colgando... con un aspecto raro... como yo en la escuela secundaria. De vuelta al directorio `tutorial/`, copia la plantilla de la página de inicio... y sobrescribe nuestro archivo original.

Ábrelo. Esto sigue extendiendo `base.html.twig`... y sigue anulando el bloque `body`. Y además, tiene un montón de HTML completamente codificado. Vamos a ver qué aspecto tiene. Actualiza y... ¡se ve genial!

Excepto que... está 100% codificado. Vamos a arreglarlo. En la parte superior, aquí está el nombre de nuestro disco, imprime la variable `title`.

Y luego, abajo para las canciones... tenemos una larga lista de HTML codificado. Convirtamos esto en un bucle. Añade `{% for track in tracks %}` como teníamos antes. Y... al final, `endfor`.

Para los detalles de la canción, utiliza `track.song`... y `track.artist`. Y ahora podemos eliminar todas las canciones codificadas.

templates/vinyl/homepage.html.twig

```
{% extends 'base.html.twig' %}
// ... lines 2 - 4
{% block body %}
<div class="container">
    <h1 class="d-inline me-3">{{ title }}</h1> <i class="fas fa-edit"></i>
    <div class="row mt-5">
// ... lines 9 - 34
        <div class="col-12 col-md-8 ps-5">
            <h2 class="mb-4">10 songs (30 minutes of 60 still available)
</h2>
            {% for track in tracks %}
            <div class="song-list">
                <div class="d-flex mb-3">
                    <a href="#">
                        <i class="fas fa-play me-3"></i>
                    </a>
                    <span class="song-details">{{ track.song }} - {{
track.artist }}</span>
                    <a href="#">
                        <i class="fas fa-bars mx-3"></i>
                    </a>
                    <a href="#">
                        <i class="fas fa-times"></i>
                    </a>
                </div>
            </div>
            {% endfor %}
            <button type="button" class="btn btn-success"><i class="fas
fa-plus"></i> Add a song</button>
        </div>
    </div>
</div>
{% endblock %}
```

¡Genial! Vamos a probarlo. ¡Hey! ¡Está cobrando vida gente!

¡Falta una página más! La página `/browse`. Ya sabes lo que hay que hacer: copiar `browse.html.twig`, y pegar en nuestro directorio. Esto se parece mucho a la página de inicio: extiende `base.html.twig` y anula el bloque `body`.

En `VinylController`, no hemos renderizado antes una plantilla... así que hagámoslo ahora: `return $this->render('vinyl/browse.html.twig')` y pasemos el género. Añade una variable para ello: `$genre =` y si tenemos un slug... utiliza nuestro elegante código de

mayúsculas y minúsculas, si no, ponlo en null. Luego borra lo de `$title`... y pasa `genre` a Twig.

src/Controller/VinylController.php

```
<?php

// ... lines 3 - 9
class VinylController extends AbstractController
{
// ... lines 12 - 29
    #[Route('/browse/{slug}')]
    public function browse(string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) :
null;

        return $this->render('vinyl/browse.html.twig', [
            'genre' => $genre
        ]);
    }
}
```

De vuelta a la plantilla, utiliza esto en el `h1`. En Twig, también podemos utilizar una sintaxis de fantasía. Así que si tenemos un `genre`, imprime `genre`, si no imprime `All Genres`.

templates/vinyl/browse.html.twig

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

{% block body %}
<div class="container">
    <h1>Browse {{ genre ? genre : 'All Genres' }}</h1>
// ... lines 6 - 45
</div>
{% endblock %}
```

Es hora de probar. Dirígete a `/browse`: "Navega por todos los géneros" Y luego `/browse/death-metal`: Navega por el Death Metal. Amigos, ¡esto empieza a parecerse a un sitio real!

Excepto que estos enlaces en el navegador... ¡no van a ninguna parte! Vamos a arreglar eso aprendiendo a generar URLs. También vamos a conocer la mega-poderosa herramienta de línea de comandos `bin/console`.

Chapter 11: Generar Urls y bin/console

Hay dos formas diferentes de interactuar con nuestra aplicación. La primera es a través del servidor web... ¡y eso es lo que hemos hecho! Llegamos a una URL y... entre bastidores, se ejecuta `public/index.php`, que arranca Symfony, llama al enrutamiento y ejecuta nuestro controlador.

Hola bin/console

¿Cuál es la segunda forma de interactuar con nuestra aplicación? Todavía no la hemos visto: es a través de una herramienta de línea de comandos llamada `bin/console`. En tu terminal ejecuta:

```
php bin/console
```

... para ver un montón de comandos dentro de este script. Me encanta esta cosa. Está lleno de cosas que nos ayudan a depurar, con el tiempo tendrá comandos de generación de código, comandos para establecer secretos: todo tipo de cosas buenas que iremos descubriendo poco a poco.

Pero quiero señalar que... ¡no hay nada especial en este script de `bin/console`! Es sólo un archivo: hay literalmente un directorio `bin/` con un archivo `console` dentro. Probablemente nunca necesitarás abrir este archivo ni pensar en él, pero es útil. Ah, y en la mayoría de los sistemas, puedes simplemente ejecutar:

```
./bin/console
```

... que se ve mejor. O a veces puedes ver que ejecute:

```
symfony console
```

... que no es más que otra forma de ejecutar este archivo. Hablaremos más de esto en un futuro tutorial.

bin/consola debug:router

El primer comando que quiero comprobar dentro de `bin/console` es `debug:router`:

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

Esto es impresionante. Nos muestra todas las rutas de nuestra aplicación, como nuestras dos rutas para `/` y `/browse/{slug}`. ¿Qué son estas otras rutas? Vienen de la barra de herramientas de depuración web y del sistema de perfilado... y sólo están aquí mientras desarrollamos localmente.

Bien, de vuelta a nuestro sitio.... en la parte superior de la página, tenemos dos enlaces no funcionales a la página de inicio y a la página de navegación. Vamos a conectarlos. Abre `templates/ base.html.twig`... y busca las etiquetas `a`. Ya está.

Así que sería muy fácil hacer que esto funcionara con sólo `href="/"`. Pero en lugar de eso, cada vez que enlacemos una página en Symfony, vamos a pedir al sistema de enrutamiento que nos genere una URL. Diremos

> *"Por favor, genera la URL de la ruta de la página de inicio, o de la ruta de la página de navegación."*

Así, si alguna vez cambiamos la URL de una ruta, todos nuestros enlaces se actualizarán instantáneamente. Magia.

Cómo nombrar tu ruta

Empecemos por la página de inicio. ¿Cómo le pedimos a Symfony que genere una URL para esta ruta? Bueno, primero tenemos que dar un nombre a la ruta. ¡Sorpresa! Cada ruta tiene un nombre interno. Puedes verlo en `debug:router`. Nuestras rutas se llaman `app_vinyl_homepage` y `app_vinyl_browse`. Huh, esos son los nombres exactos de mis tortugas mascota cuando era niño.

¿De dónde vienen estos nombres? Por defecto, Symfony nos genera automáticamente un nombre, lo cual está bien. El nombre no se utiliza en absoluto hasta que generamos una URL a la misma. Y en cuanto necesitemos generar una URL a una ruta, recomiendo encarecidamente tomar el control de este nombre... sólo para asegurarnos de que nunca cambia accidentalmente.

Para ello, busca la ruta y añade un argumento: `name` ajustado a, qué tal, `app_homepage`. Me gusta utilizar el prefijo `app_`: facilita la búsqueda del nombre de la ruta más adelante.

src/Controller/VinylController.php

```
1  <?php
2
↕  // ... lines 3 - 9
10 class VinylController extends AbstractController
11 {
12     #[Route('/', name: 'app_homepage')]
13     public function homepage(): Response
14     {
↕  // ... lines 15 - 27
28     }
↕  // ... lines 29 - 38
39 }
```

Por cierto, los atributos de PHP 8 -como este atributo `Route` - están representados por clases PHP reales y físicas. Si mantienes pulsado command o ctrl, puedes abrirlo y mirar dentro. Esto es genial: el método `__construct()` muestra todas las diferentes opciones que puedes pasar al atributo.

Por ejemplo, hay un argumento `name`... y entonces estamos utilizando la sintaxis de argumentos con nombre de PHP para pasar esto al atributo. Abrir un atributo es una buena manera de conocer sus opciones.

Generar una URL desde Twig

De todos modos, ahora que le hemos dado un nombre, vuelve a nuestro terminal y ejecuta de nuevo `debug:router`:

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

Esta vez... ¡sí! ¡La ruta se llama `app_homepage`! Cópialo y vuelve a `base.html.twig`. Para generar una URL dentro de twig, di `{{` -porque vamos a imprimir algo- y luego utiliza una función Twig llamada `path()`. Pásale el nombre de la ruta.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
// ... lines 3 - 26
    <body>
// ... lines 28 - 31
                <a class="navbar-brand" href="{{ path('app_homepage') }}">
                    <i class="fas fa-record-vinyl"></i>
                    Mixed Vinyl
                </a>
// ... lines 36 - 84
    </body>
</html>
```

Ya está Actualiza... ¡y el enlace de aquí arriba funciona!

Falta un enlace más. Ya conocemos el primer paso: dar un nombre a la ruta. Así que `name:` y, qué tal, `app_browse`.

src/Controller/VinylController.php

```
<?php

// ... lines 3 - 9
class VinylController extends AbstractController
{
// ... lines 12 - 29
    #[Route('/browse/{slug}', name: 'app_browse')]
    public function browse(string $slug = null): Response
    {
// ... lines 33 - 37
    }
}
```

Copia eso, y... desplázate un poco hacia abajo. Aquí está: "Examinar mezclas". Cámbialo por `{{ path('app_browse') }}`.

templates/base.html.twig

```
1  <!DOCTYPE html>
2  <html>
↕  // ... lines 3 - 26
27     <body>
↕  // ... lines 28 - 40
41                         <li class="nav-item">
42                             <a class="nav-link" style="margin-top: 20px;" href="{{ path('app_browse') }}">Browse Mixes</a>
43                         </li>
↕  // ... lines 44 - 84
85     </body>
86 </html>
```

Y ahora... ¡ese enlace también funciona!

Generar URLs con comodines

Pero en esta página, tenemos algunos enlaces rápidos para ir a la página de exploración de un género específico. Y éstos aún no funcionan.

Esto es interesante. Queremos generar una URL como antes... pero esta vez necesitamos pasar algo al comodín `{slug}`. Abre `browse.html.twig`. Así es como lo hacemos. La primera parte es la misma: `{{ path() }}` y luego el nombre de la ruta: `app_browse`.

Si nos detuviéramos aquí, se generaría `/browse`. Para pasar valores a cualquier comodín de una ruta, `path()` tiene un segundo argumento: una matriz asociativa de esos valores. Y, de nuevo, al igual que en JavaScript, para crear una "matriz asociativa", utilizas `{` y `}`. Voy a pulsar intro para dividir esto en varias líneas... sólo para que sea legible. Dentro añade una clave `slug` a la matriz... y como este es el género "Pop", ponla en `pop`.

¡Genial! Repitamos esto dos veces más: `{{ path('app_browse') }}` pasar las llaves para un array asociativo, con `slug` fijado en `rock`. Y luego una vez más aquí abajo... que haré muy rápidamente.

templates/vinyl/browse.html.twig

```
↕ // ... lines 1 - 2
3 {% block body %}
↕ // ... lines 4 - 7
8     <ul class="genre-list ps-0 mt-2 mb-3">
9         <li class="d-inline">
10            <a class="btn btn-primary btn-sm" href="{{ path('app_browse',
      {
11                slug: 'pop'
12            }) }}">Pop</a>
13        </li>
14        <li class="d-inline">
15            <a class="btn btn-primary btn-sm" href="{{ path('app_browse',
      {
16                slug: 'rock'
17            }) }}">Rock</a>
18        </li>
19        <li class="d-inline">
20            <a class="btn btn-primary btn-sm" href="{{ path('app_browse',
      {
21                slug: 'heavy-metal'
22            }) }}">Heavy Metal</a>
23        </li>
24    </ul>
↕ // ... lines 25 - 52
53 {% endblock %}
```

¡Vamos a ver si funciona! Actualiza. ¡Ah! La variable `rock` no existe. Seguro que alguno de vosotros me ha visto hacer eso. Me olvidé de las comillas, así que esto parece una variable.

Inténtalo de nuevo. Ya está. Y prueba los enlaces... ¡sí! ¡Funcionan!

Siguiente: hemos creado dos páginas HTML. Ahora vamos a ver cómo queda la creación de una ruta de la API JSON.

Chapter 12: Ruta de la API JSON

En un futuro tutorial, vamos a crear una base de datos para gestionar las canciones, los géneros y los discos de vinilo mezclados que nuestros usuarios están creando. Ahora mismo, estamos trabajando completamente con datos codificados... pero nuestros controladores -y- especialmente las plantillas no serán muy diferentes una vez que hagamos todo esto dinámico.

Así que este es nuestro nuevo objetivo: quiero crear una ruta de la API que devuelva los datos de una sola canción como JSON. Vamos a usar esto en unos minutos para dar vida a este botón de reproducción. Por el momento, ninguno de estos botones hace nada, pero tienen un aspecto bonito.

Crear el controlador JSON

Los dos pasos para crear un punto final de la API son... exactamente los mismos que para crear una página HTML: necesitamos una ruta y un controlador. Como esta ruta de la API devolverá datos de canciones, en lugar de añadir otro método dentro de `VinylController`, vamos a crear una clase de controlador totalmente nueva. La forma en que organices este material depende enteramente de ti.

Crea una nueva clase PHP llamada `SongController`... o `SongApiController` también sería un buen nombre. En su interior, ésta comenzará como cualquier otro controlador, extendiendo`AbstractController`. Recuerda: esto es opcional... pero nos proporciona métodos de acceso directo sin inconvenientes.

A continuación, crea un `public function` llamado, qué tal, `getSong()`. Añade la ruta... y pulsa el tabulador para autocompletar esto de forma que PhpStorm añada la declaración de uso en la parte superior. Establece la URL como `/api/songs/{id}`, donde `id` será finalmente el id de la base de datos de la canción.

Y como tenemos un comodín en la ruta, se nos permite tener un argumento `$id`. Por último, aunque no necesitamos hacerlo, como sabemos que nuestro controlador devolverá un objeto `Response`, podemos establecerlo como tipo de retorno. Asegúrate de autocompletar el del componente `HttpFoundation` de Symfony.

Dentro del método, para empezar, `dd($id)`... sólo para ver si todo funciona.

src/Controller/SongController.php

```php
<?php

namespace App\Controller;

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

class SongController extends AbstractController
{
    #[Route('/api/songs/{id}')]
    public function getSong($id): Response
    {
        dd($id);
    }
}
```

¡Vamos a hacerlo! Dirígete a `/api/songs/5` y... ¡lo tienes! Otra página nueva.

De vuelta a ese controlador, voy a pegar algunos datos de la canción: finalmente, esto vendrá de la base de datos. Puedes copiarlo del bloque de código de esta página. Nuestro trabajo es devolverlo como JSON.

Entonces, ¿cómo devolvemos JSON en Symfony? Devolviendo un nuevo `JsonResponse` y pasándole los datos.

src/Controller/SongController.php

```php
<?php

// ... lines 3 - 5
use Symfony\Component\HttpFoundation\JsonResponse;
// ... lines 7 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}')]
    public function getSong($id): Response
    {
        // TODO query the database
        $song = [
            'id' => $id,
            'name' => 'Waterfalls',
            'url' => 'https://symfonycasts.s3.amazonaws.com/sample.mp3',
        ];

        return new JsonResponse($song);
    }
}
```

Lo sé... ¡demasiado fácil! Refresca y... ¡hola JSON! Ahora puedes estar pensando:

> *"¡Ryan! Nos has estado diciendo -repetidamente- que un controlador debe devolver siempre un objeto Symfony `Response`, que es lo que devuelve `render()`. ¿Ahora devuelve otro tipo de objeto `Response`?"*

Vale, es justo... pero esto funciona porque `JsonResponse` es una Respuesta. Me explico: a veces es útil saltar a las clases principales para ver cómo funcionan. Para ello, en PHPStorm - si estás en un Mac mantén pulsado comando, si no, mantén pulsado control- y luego haz clic en el nombre de la clase a la que quieras saltar. Y... ¡sorpresa! `JsonResponse` extiende `Response`. Sí, seguimos devolviendo un `Response`. Pero esta subclase está bien porque codifica automáticamente JSON nuestros datos y establece la cabecera `Content-Type` en `application/json`.

El método abreviado ->json()

Ah, y de vuelta a nuestro controlador, podemos ser aún más perezosos diciendo `return $this->json($song)`... donde `json()` es otro método abreviado que viene de `AbstractController`.

src/Controller/SongController.php

```php
<?php

// ... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}')]
    public function getSong($id): Response
    {
// ... lines 15 - 20

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

Hacer esto no supone ninguna diferencia, porque sólo es un atajo para devolver ... ¡un `JsonResponse`!

Si estás construyendo una API seria, Symfony tiene un componente `serializer` que es realmente bueno para convertir objetos en JSON... y luego JSON de nuevo en objetos. Hablamos mucho de él en nuestro tutorial de la Plataforma API, que es una potente biblioteca para crear APIs en Symfony.

A continuación, vamos a aprender cómo hacer que nuestras rutas sean más inteligentes, por ejemplo, haciendo que un comodín sólo coincida con un número, en lugar de coincidir con cualquier cosa.

Chapter 13: Rutas inteligentes: Sólo GET y Validar {Comodines}

Ahora que tenemos una nueva página, en tu terminal, ejecuta de nuevo `debug:router`.

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

Sí, ¡ahí está nuestra nueva ruta! Observa que la tabla tiene una columna llamada "Método" que dice "cualquiera". Esto significa que puedes hacer una petición a esta URL utilizando cualquier método HTTP -como GET o POST- y coincidirá con esa ruta.

Restringir las rutas sólo a GET o POST

Pero el objetivo de nuestra nueva ruta API es permitir a los usuarios hacer una petición GET para obtener datos de la canción. Técnicamente, ahora mismo, también podrías hacer una petición POST a esto... y funcionaría perfectamente. Puede que no nos importe, pero a menudo con las APIs, querrás restringir una ruta para que sólo funcione con un método específico como GET, POST o PUT. ¿Podemos hacer que esta ruta, de alguna manera, sólo funcione con peticiones GET?

Sí! Añadiendo otra opción a la `Route`. En este caso, se llama `methods`, ¡incluso se autocompleta! Establece esto como un array y, pon `GET`.

src/Controller/SongController.php

```php
<?php

// ... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}', methods: ['GET'])]
    public function getSong($id): Response
    {
// ... lines 15 - 22
    }
}
```

Voy a mantener pulsado Comando y a hacer clic en la clase `Route` de nuevo... para que podamos ver que... ¡sí! `methods` es uno de los argumentos.

Volvemos a `debug:router`:

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

Bien. La ruta ahora sólo coincidirá con las peticiones GET. Es... un poco difícil probar esto, ya que un navegador siempre hace peticiones GET si vas directamente a una URL... pero aquí es donde otro comando de `bin/console` resulta útil: `router:match`.

Si lo ejecutamos sin argumentos

```
php bin/console router:match
```

Nos da un error, ¡pero muestra cómo se utiliza! Inténtalo:

```
php bin/console router:match /api/songs/11
```

Y... ¡eso coincide con nuestra nueva ruta! Pero ahora pregúntate qué pasaría si hiciéramos una petición POST a esa URL con `--method=POST`:

```
php bin/console router:match /api/songs/11 --method=POST
```

¡Ninguna ruta coincide con esta ruta con ese método! Pero dice que casi coincide con nuestra ruta.

Restringir los comodines de ruta mediante Regex

Vamos a hacer una cosa más para restringir nuestra nueva ruta. Voy a añadir una pista de tipo `int` al argumento `$id`.

src/Controller/SongController.php

```
<?php

// ... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id}', methods: ['GET'])]
    public function getSong(int $id): Response
    {
// ... lines 15 - 22
    }
}
```

Eso... no cambia nada, excepto que ahora PHP tomará la cadena `id` de la URL que Symfony pasa a este método y la convertirá en un `int`, lo cual es... agradable porque entonces estamos tratando con un verdadero número entero en nuestro código.

Puedes ver la sutil diferencia en la respuesta. Ahora mismo, el campo `id` es una cadena. Cuando actualizamos, `id` es ahora un número verdadero en JSON.

Pero... si alguien se hiciera el remolón... y pasara a `/api/songs/apple`... ¡vaya! ¡Un error PHP, que, en producción, sería una página de error 500! Eso no me gusta.

Pero... ¿qué podemos hacer? El error se produce cuando Symfony intenta llamar a nuestro controlador y le pasa ese argumento. Así que no podemos poner código en el controlador para comprobar si `$id` es un número: ¡es demasiado tarde!

¿Y si, en cambio, pudiéramos decirle a Symfony que esta ruta sólo debe coincidir si el comodín `id` es un número? ¿Es posible? Totalmente

Por defecto, cuando tienes un comodín, coincide con cualquier cosa. Pero puedes cambiarlo para que coincida con una expresión regular personalizada. Dentro de las llaves, justo después del nombre, añade un `<`, luego `>` y, entre medias, `\d+`. Es una expresión regular que significa "un dígito de cualquier longitud".

src/Controller/SongController.php

```
<?php

// ... lines 3 - 9
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'])]
    public function getSong(int $id): Response
    {
// ... lines 15 - 22
    }
}
```

¡Pruébalo! Actualiza y... ¡sí! A 404. No se ha encontrado ninguna ruta: simplemente no ha coincidido con esta ruta. Un 404 está muy bien... pero un error 500... eso es algo que queremos evitar. Y si volvemos a `/api/songs/5`... eso sigue funcionando.

A continuación: si me preguntaras cuál es la parte más central e importante de Symfony, no lo dudaría: son los servicios. Descubramos qué es un servicio y cómo es la clave para liberar el potencial de Symfony.

Chapter 14: Objetos de servicio

Veo a Symfony como dos grandes partes. La primera parte es el sistema de ruta, controlador y respuesta. Es muy simple y bueno... ¡ya eres un experto en ello! La segunda mitad de Symfony se trata de los muchos objetos útiles que están flotando por ahí... ¡sólo esperando a que los usemos!

Objetos de servicio Hola

Por ejemplo, cuando renderizamos una plantilla, lo que estamos haciendo en realidad es aprovechar un objeto Twig y pedirle que renderice una plantilla. También hay un objeto registrador, un objeto caché, un objeto de conexión a la base de datos, un objeto que ayuda a hacer peticiones a la API, ¡y muchos, muchos más! Y cuando instalas un nuevo paquete, eso te da aún más objetos útiles.

La verdad es que todo lo que hace Symfony lo hace... uno de estos objetos útiles. Diablos, ¡hay incluso un objeto router que se encarga de encontrar la ruta adecuada para la página dada!

En el mundo de Symfony, y realmente en el mundo de la programación orientada a objetos en general, estos "objetos que hacen trabajo" tienen un nombre especial: servicios. Pero no dejes que esa palabra te confunda. Cuando oigas servicio, piensa: ¡es un objeto que hace trabajo! Como un objeto de plantilla que representa una plantilla o un objeto de conexión a la base de datos que realiza consultas.

Y como los objetos de servicio hacen trabajo, son básicamente... ¡herramientas que te ayudan a hacer tu trabajo! La segunda mitad de Symfony consiste en descubrir qué servicios están disponibles y cómo utilizarlos.

El comando debug:autowiring

Vamos a probar algo. En nuestro controlador, quiero registrar un mensaje... quizás algún mensaje de depuración. Como registrar un mensaje es un trabajo, lo hace un servicio. ¿Nuestra aplicación ya tiene un servicio de registro? Y si es así, ¿cómo lo conseguimos?

Para averiguarlo, ve a tu terminal y ejecuta otro comando `bin/console`:

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

Saluda a uno de los comandos más potentes de `bin/console`. Me encanta esta cosa! Esta lista todos los servicios que existen en nuestra aplicación. De acuerdo, en realidad no es la lista completa, pero esto muestra los servicios que probablemente necesites. Y aunque nuestra aplicación es pequeña, ¡hay muchas cosas aquí! Hay un servicio de sistema de archivos... y aquí abajo un servicio de caché. ¡Incluso hay un servicio Twig!

¿Hay un servicio de registro? Puedes mirar en esta lista... o puedes volver a ejecutar este comando y buscar la palabra log:

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

¡Excelente! Por ahora, ignora todo excepto la primera línea. Esta línea nos dice que hay un servicio de registro y que este objeto implementa una interfaz llamada `Psr\Log\LoggerInterface`.

Obtención de un servicio mediante autoconexión

Vale, ¿y por qué nos ayuda saber eso? Porque si quieres un servicio, lo pides utilizando la sugerencia de tipo que se muestra en este comando. Se llama autoconexión.

Vamos a probarlo. Dirígete a nuestro controlador y añade un segundo argumento. En realidad, el orden de los argumentos no importa. Lo que importa es que el nuevo argumento se indique con `LoggerInterface`. Pulsaré el tabulador para autocompletarlo... para que PhpStorm añada la declaración de uso en la parte superior.

En este caso, el argumento puede llamarse como sea, como `$logger`. Cuando Symfony ve esta sugerencia de tipo, busca dentro de la lista `debug:autowiring`... y como hay una coincidencia, nos pasará el servicio de registro.

Así que ahora conocemos dos tipos diferentes de argumentos que podemos tener en el controlador: puedes tener un argumento cuyo nombre coincida con un comodín de la ruta o un argumento cuyo tipo-hint coincida con uno de los servicios de nuestra app.

Utilizar el registrador

Bien, ahora que sabemos que Symfony nos pasará el objeto de servicio logger, ¡vamos a utilizarlo! No sé, todavía, qué métodos puedo llamar en él pero... si decimos `$logger->`... PhpStorm... ¡nos lo dice! ¡Ha sido fácil!

Voy a registrar algo en un nivel de prioridad `info()`. Digamos:

> *"Devolución de la respuesta de la API para la canción"*

Y luego el `$id`.

src/Controller/SongController.php

```
<?php

// ... lines 3 - 4
use Psr\Log\LoggerInterface;
// ... lines 6 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'])]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
// ... lines 16 - 22
        $logger->info('Returning API response for song '.$id);
// ... lines 24 - 25
    }
}
```

En realidad, podemos hacer algo aún más genial con este servicio de registro. Añade `{song}` al mensaje... y añade un segundo argumento, que es una matriz de información extra que quieres adjuntar al mensaje de registro. Pasa `song` ajustado a `$id`. En un minuto, verás que el registrador imprimirá el id real en lugar de `{song}`.

src/Controller/SongController.php

```
<?php

// ... lines 3 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'])]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
// ... lines 16 - 22
        $logger->info('Returning API response for song {song}', [
            'song' => $id,
        ]);
// ... lines 26 - 27
    }
}
```

En cualquier caso, este controlador es para nuestra ruta de la API. Así que vamos a refrescarlo. ¡Um... ok! Así que no hay error, ¡eso es bueno! ¿Pero ha funcionado? ¿Dónde se registra realmente el servicio de registro?

Averigüémoslo a continuación, aprendamos un truco para ver el perfilador incluso para las peticiones de la API y luego aprovechemos nuestro segundo servicio directamente.

Chapter 15: El servicio Twig y el perfilador de peticiones de la API

Como esta página acaba de cargarse sin ningún error, pensamos que acabamos de registrar con éxito un mensaje a través del servicio de registro. Pero... ¿dónde van los mensajes de registro? ¿Cómo podemos comprobarlo?

El servicio de registro lo proporciona una biblioteca que hemos instalado antes, llamada monolog, que forma parte del paquete de depuración. Y puedes controlar su configuración dentro del archivo `config/packages/monolog.yaml`, incluyendo dónde se registran los mensajes de registro, por ejemplo, en qué archivo. Nos centraremos más en la configuración en el siguiente tutorial.

El perfilador de peticiones de la API

Pero una forma de ver siempre los mensajes de registro de una petición es a través del perfilador Esto es muy útil. Ve a la página de inicio, haz clic en cualquier enlace de la barra de herramientas de depuración web... y luego ve a la sección Registros. Ahora veremos todos los mensajes de registro que se hicieron sólo durante esa última petición a la página de inicio.

¡Genial! Excepto que... nuestro mensaje de registro se hace en una ruta de la API... ¡y las rutas de la API no tienen una barra de herramientas de depuración web en la que podamos hacer clic! ¿Estamos atascados? No! Actualiza esta página una vez más... y luego ve manualmente a `/_profiler`. Esta es... una especie de puerta secreta al sistema de perfiles... y esta página muestra las últimas diez peticiones realizadas en nuestro sistema. La segunda en la parte superior es la petición de la API que acabamos de hacer. Haz clic en el pequeño enlace del token para ver... ¡sí! ¡Estamos viendo el perfil de esa petición de la API! En la sección de Registros... ¡ahí está!

> *“Respuesta de la API para la canción 5”*

... e incluso puedes ver la información extra que hemos pasado.

Renderizar una plantilla Twig manualmente

Vale, los servicios son tan importantes que... Quiero hacer un ejemplo rápido más. Vuelve a `VinylController`. El método `render()` es realmente un atajo para obtener el servicio "Twig", llamar a algún método de ese objeto para renderizar la plantilla... y luego poner la cadena HTML final en un objeto `Response`. Es un gran atajo y deberías utilizarlo.

Pero! Como reto, ¿podríamos renderizar una plantilla sin usar ese método? ¡Por supuesto! Hagámoslo.

Primer paso: encontrar el servicio que hace el trabajo que necesitas hacer. Así que tenemos que encontrar el servicio Twig. Volvamos a hacer nuestro truco:

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

Y... ¡sí! Al parecer, el tipo de pista que tenemos que utilizar es `Twig\Environment`.

¡De acuerdo! Vuelve a nuestro método, añade un argumento, escribe `Environment`, y pulsa el tabulador para autocompletarlo y que PhpStorm añada la sentencia `use`. Vamos a llamarlo `$twig`.

A continuación, en lugar de usar `render`, digamos `$html =` y luego `$twig->`. Al igual que con el registrador, no necesitamos saber qué métodos tiene esta clase, porque, gracias a la sugerencia de tipo, PhpStorm puede decirnos todos los métodos. El método `render()` parece que es probablemente lo que queremos. El primer argumento es el nombre de la cadena de la plantilla a renderizar y el argumento `$context` contiene las variables. Así que... tiene los mismos argumentos que ya estábamos pasando.

Para ver si funciona, `dd($html)`.

src/Controller/VinylController.php

```php
<?php

// ... lines 3 - 10
class VinylController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(Environment $twig): Response
    {
// ... lines 16 - 24
        $html = $twig->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
            'tracks' => $tracks,
        ]);
        dd($html);
    }
// ... lines 31 - 40
}
```

¡Hora de probar! Dirígete a la página de inicio... ¡y sí! ¡Acabamos de renderizar una plantilla manualmente! ¡Increíble! Y podemos terminar esta página envolviendo eso en una respuesta: `return new Response($html)`.

src/Controller/VinylController.php

```php
<?php

// ... lines 3 - 10
class VinylController extends AbstractController
{
    #[Route('/', name: 'app_homepage')]
    public function homepage(Environment $twig): Response
    {
// ... lines 16 - 24
        $html = $twig->render('vinyl/homepage.html.twig', [
            'title' => 'PB & Jams',
            'tracks' => $tracks,
        ]);

        return new Response($html);
    }
// ... lines 32 - 41
}
```

Y ahora... ¡la página funciona! Y entendemos que la verdadera forma de renderizar una plantilla es a través del servicio Twig. Algún día te encontrarás en una situación en la que necesites

renderizar una plantilla pero no estés en un controlador... y por tanto no tengas el método abreviado `$this->render()`. Saber que hay un servicio Twig que puedes recuperar será la clave para resolver ese problema. Más sobre esto en el próximo tutorial.

Pero en una aplicación real, en un controlador, no hay razón para hacer todo este trabajo extra. Así que voy a revertir esto... y volver a usar `render()`. Y... entonces ya no necesitamos autocablear ese argumento... e incluso podemos limpiar la declaración `use`.

Aquí están los tres grandes, gigantescos e importantes puntos de partida. En primer lugar, Symfony está repleto de objetos que hacen su trabajo... a los que llamamos servicios. Los servicios son herramientas. Segundo, todo el trabajo en Symfony lo hace un servicio... incluso cosas como el enrutamiento. Y en tercer lugar, podemos utilizar los servicios para ayudarnos a realizar nuestro trabajo mediante la autoconexión de los mismos.

En el próximo tutorial de esta serie, profundizaremos en este concepto tan importante.

Pero antes de que terminemos este tutorial, quiero hablar de otra cosa increíble y asombrosa: Webpack Encore, la clave para escribir CSS y JavaScript de forma profesional. A lo largo de estos últimos capítulos, vamos a dar vida a nuestro sitio e incluso a hacerlo tan responsivo como una aplicación de una sola página.

Chapter 16: Configuración de Webpack Encore

Nuestra configuración de CSS está bien. Ponemos los archivos en el directorio `public/` y luego... apuntamos a ellos desde dentro de nuestras plantillas. Podríamos añadir archivos de JavaScript de la misma manera.

Pero si queremos tomarnos realmente en serio la escritura de CSS y JavaScript, tenemos que llevar esto al siguiente nivel. E incluso si te consideras un desarrollador principalmente de backend, las herramientas de las que vamos a hablar te permitirán escribir CSS y JavaScript de forma más fácil y menos propensa a errores que a lo que probablemente estés acostumbrado.

La clave para llevar nuestra configuración al siguiente nivel es aprovechar una biblioteca de nodos llamada Webpack. Webpack es la herramienta estándar de la industria para empaquetar, minificar y analizar tu CSS, JavaScript y otros archivos del frontend. Pero no te preocupes: Node es sólo JavaScript. Y su papel en nuestra aplicación será bastante limitado.

Configurar Webpack puede ser complicado. Por eso, en el mundo Symfony, utilizamos una herramienta ligera llamada Webpack Encore. Sigue siendo Webpack... ¡sólo lo hace más fácil! Y tenemos un tutorial gratuito sobre ello si quieres profundizar.

Instalar Encore

Pero vamos a hacer un curso intensivo ahora mismo. Primero, en tu línea de comandos, asegúrate de que tienes instalado Node:

```
node -v
```

También necesitarás `npm` -que viene con Node automáticamente- o `yarn`:

```
yarn --version
```

Npm y yarn son gestores de paquetes de Node: son el Compositor para el mundo de Node... y puedes usar cualquiera de los dos. Si decides usar yarn - que es lo que yo usaré - asegúrate de instalar la versión 1.

Estamos a punto de instalar un nuevo paquete... así que vamos a confirmar todo:

```
git add .
```

Y... se ve bien:

```
git status
```

Así que confirma todo:

```
git commit -m "Look mom! A real app"
```

Para instalar Encore, ejecuta:

```
composer require encore
```

Esto instala WebpackEncoreBundle. Recuerda que un bundle es un plugin de Symfony. Y este paquete tiene una receta: una receta muy importante. Ejecuta:

```
git status
```

La receta de Encore

Por primera vez, la receta ha modificado el archivo `.gitignore`. Vamos a comprobarlo. Abre `.gitignore`. Lo de arriba es lo que teníamos originalmente... y lo de abajo es lo nuevo que

ha añadido WebpackEncoreBundle. Está ignorando el directorio `node_modules/`, que es básicamente el directorio `vendor/` para Node. No necesitamos confirmarlo porque esas bibliotecas de proveedores se describen en otro archivo nuevo de la receta: `package.json`. Este es el archivo `composer.json` de Node: describe los paquetes de Node que necesita nuestra aplicación. El más importante es el propio Webpack Encore, que es una biblioteca de Node. También tiene algunos otros paquetes que nos ayudarán a realizar nuestro trabajo.

La receta también ha añadido un directorio `assets/`... y un archivo de configuración para controlar Webpack: `webpack.config.js`. El directorio `assets/` ya contiene un pequeño conjunto de archivos para que podamos empezar.

Instalar las dependencias de Node

Bien, con Composer, si no tuviéramos este directorio `vendor/`, podríamos ejecutar `composer install` que le diría que leyera el archivo `composer.json` y volviera a descargar todos los paquetes en `vendor/`. Lo mismo ocurre con Node: tenemos un archivo `package.json`. Para descargarlo, ejecuta

```
yarn install
```

O:

```
npm install
```

¡Go node go! Esto tardará unos instantes mientras se descarga todo. Probablemente recibirás algunas advertencias como ésta, que puedes ignorar.

¡Genial! Esto hizo dos cosas. En primer lugar, descargó un montón de archivos en el directorio `node_modules/`: el directorio de "proveedores" de Node. También creó un archivo `yarn.lock`... o `package-lock.json` si estás usando npm. Esto sirve para el mismo propósito de `composer.lock`: almacena las versiones exactas de todos los paquetes para que obtengas las mismas versiones la próxima vez que instales tus dependencias.

En su mayor parte, no necesitas preocuparte por estos archivos de bloqueo... excepto que debes confirmarlos. Hagámoslo. Ejecuta:

```
git status
```

Entonces:

```
git add .
```

Hermoso:

```
git status
```

Y confirma:

```
git commit -m "Adding Webpack Encore"
```

¡Hey! ¡Ya está instalado Webpack Encore! Pero... ¡todavía no hace nada! Aprovechado. A continuación, vamos a utilizarlo para llevar nuestro JavaScript al siguiente nivel.

Chapter 17: Empaquetar JS y CSS con Encore

Cuando instalamos Webpack Encore, su receta nos dio este nuevo directorio `assets/`. Mira el archivo `app.js`. Es interesante. Observa cómo importa este archivo `bootstrap`. En realidad es `bootstrap.js`: este archivo de aquí. La extensión `.js` es opcional.

Importaciones de JavaScript

Esta es una de las cosas más importantes que nos da Webpack: la capacidad de importar un archivo JavaScript de otro. Podemos importar funciones, objetos... realmente cualquier cosa desde otro archivo. Vamos a hablar más sobre este archivo `bootstrap.js` dentro de un rato.

Esto también importa un archivo CSS? Si no has visto esto antes, puede parecer... raro: ¿JavaScript importando CSS?

Para ver cómo funciona todo esto, en `app.js`, añade un `console.log()`.

assets/app.js

```
// ... lines 1 - 12

console.log('Hi! My name is app.js!');
```

Y `app.css` ya tiene un fondo de cuerpo... pero añade un `!important` para que podamos ver definitivamente si se está cargando.

assets/styles/app.css

```
body {
    background-color: lightgray !important;
}
```

Vale... ¿entonces quién lee estos archivos? Porque... no viven en el directorio `public/`... así que no podemos crear etiquetas `script` o `link` que apunten directamente a ellos.

webpack.config.js

Para responder a esto, abre `webpack.config.js`. Webpack Encore es un binario ejecutable: vamos a ejecutarlo en un minuto. Cuando lo hagamos, cargará este archivo para obtener su configuración.

Y aunque hay un montón de funciones dentro de Webpack, lo único en lo que tenemos que centrarnos ahora es en esta: `addEntry()`. Este `app` puede ser cualquier cosa... como `dinosaur`, no importa. Te mostraré cómo se utiliza en un minuto. Lo importante es que apunta al archivo `assets/app.js`. Por ello, `app.js` será el primer y único archivo que Webpack analizará.

Esto es bastante bueno: Webpack leerá el archivo `app.js` y luego seguirá todas las declaraciones de `import` recursivamente hasta que finalmente tenga una colección gigante de todo el JavaScript y el CSS que nuestra aplicación necesita. Entonces, lo escribirá en el directorio `public/`.

Ejecutando Webpack Encore

Vamos a verlo en acción. Busca tu terminal y ejecuta:

```
yarn watch
```

Esto es, como dice, un atajo para ejecutar `encore dev --watch`. Si miras tu archivo `package.json`, viene con una sección `script` con algunos atajos.

En cualquier caso, `yarn watch` hace dos cosas. En primer lugar, crea un nuevo directorio `public/build/` y, dentro, los archivos `app.css` y `app.js` Pero no dejes que los nombres te engañen: `app.js` contiene mucho más que lo que hay dentro de `assets/app.js`: contiene todo el JavaScript de todas las importaciones que encuentra. `app.css` contiene todo el CSS de todas las importaciones.

La razón por la que estos archivos se llaman `app.css` y `app.js` es por el nombre de la entrada.

Así que la conclusión es que, gracias a Encore, de repente tenemos nuevos archivos en el directorio `public/build/` que contienen todo el JavaScript y el CSS que necesita nuestra aplicación

Las funciones Twig de Encore

Y si te diriges a tu página de inicio y la actualizas... ¡woh! Ha funcionado al instante!? El fondo ha cambiado... y en mi inspector... ¡está el registro de la consola! ¿Cómo diablos ha ocurrido eso?

Abre tu diseño base: `templates/base.html.twig`. El secreto está en las funciones `encore_entry_link_tags()` y `encore_entry_script_tags()`. Apuesto a que puedes adivinar lo que hacen: añadir la etiqueta `link` a `build/app.css` y la etiqueta `script` a `build/app.js`.

Puedes ver esto en tu navegador. Mira la fuente de la página y... ¡sí! La etiqueta link para `/build/app.css`... y la etiqueta `script` para `/build/app.js`. Ah, pero también ha renderizado otras dos etiquetas `script`. Eso es porque Webpack es muy inteligente. Por motivos de rendimiento, en lugar de volcar un gigantesco archivo `app.js`, a veces Webpack lo divide en varios archivos más pequeños. Afortunadamente, estas funciones Twig de Encore son lo suficientemente inteligentes como para manejar eso: incluirá todas las etiquetas de enlace o de script necesarias.

Lo más importante es que el código que tenemos en nuestro archivo `assets/app.js`-incluyendo todo lo que importa- ¡ahora funciona y aparece en nuestra página!

Vigilancia de los cambios

Ah, y como hemos ejecutado `yarn watch`, Encore sigue funcionando en segundo plano en busca de cambios. Compruébalo: entra en `app.css`... y cambia el color de fondo. Guarda, pasa y actualiza

assets/styles/app.css

```
1 body {
2     background-color: maroon !important;
3 }
```

¡Se actualiza instantáneamente! Eso es porque Encore se ha dado cuenta del cambio y ha recompilado el archivo construido muy rápidamente.

A continuación: vamos a trasladar nuestro CSS existente al nuevo sistema y a aprender cómo podemos instalar e importar bibliotecas de terceros -mira Bootstrap o FontAwesome-

directamente en nuestra configuración de Encore.

Chapter 18: Instalación de código de terceros en nuestro JS/CSS

Ahora tenemos un nuevo y bonito sistema de JavaScript y CSS que vive completamente dentro del directorio `assets/`. Vamos a trasladar nuestros estilos públicos a éste. Abre `public/styles/app.css`, copia todo esto, borra todo el directorio... y pégalo en el nuevo `app.css`. Gracias a `encore_entry_link_tags()` en `base.html.twig`, el nuevo CSS se está incluyendo... y ya no necesitamos la antigua etiqueta `link`.

Ve a comprobarlo. Refresca y... ¡todavía se ve muy bien!

Instalación de bibliotecas JavaScript/CSS de terceros

Vuelve a `base.html.twig`. ¿Qué pasa con estas etiquetas de enlace externo para bootstrap y FontAwesome? Bueno, puedes mantener totalmente estos enlaces CDN. Pero también podemos procesar estas cosas a través de Encore. ¿Cómo? Instalando Bootstrap y FontAwesome como bibliotecas de proveedor e importándolas.

Elimina todas estas etiquetas de enlace... y luego actualiza. ¡Vaya! Vuelve a parecer que he diseñado este sitio. Vamos... primero a volver a añadir bootstrap. Busca tu terminal. Ya que el comando watch se está ejecutando, abre una nueva pestaña de terminal y ejecútalo:

```
yarn add bootstrap --dev
```

Esto hace tres cosas. Primero, añade `bootstrap` a nuestro archivo `package.json`. Segundo, descarga bootstrap en nuestro directorio `node_modules/`... lo encontrarías aquí abajo. Y tercero, actualiza el archivo `yarn.lock` con la versión exacta de bootstrap que acaba de descargar.

Si nos detuviéramos ahora... ¡esto no supondría ninguna diferencia! Hemos descargado bootstrap -yay- pero no lo estamos utilizando.

Para usarlo, tenemos que importarlo. Entra en `app.css`. Al igual que en los archivos JavaScript, podemos importar desde dentro de los archivos CSS diciendo `@import` y luego el archivo. Podemos hacer referencia a un archivo en el mismo directorio con `./other-file.css`. O, si quieres importar algo del directorio `node_modules/` en CSS, hay un truco: un `~` y luego el nombre del paquete: `bootstrap`.

assets/styles/app.css

```
1  @import '~bootstrap';
↕  // ... lines 2 - 34
```

Eso es todo En cuanto hicimos eso, la función de vigilancia de Encore reconstruyó nuestro archivo `app.css`... ¡que ahora incluye Bootstrap! Observa: actualiza la página y... ¡volvemos a estar de vuelta! ¡Qué bien!

Las otras dos cosas que nos faltan son `FontAwesome` y una fuente específica. Para añadirlas, vuelve al terminal y ejecútalas:

```
yarn add @fontsource/roboto-condensed --dev
```

Revelación completa: hice algunas búsquedas antes de grabar para saber los nombres de todos los paquetes que necesitamos. Puedes buscar los paquetes en https://npmjs.com.

Añadamos también el último que necesitamos:

```
yarn add @fortawesome/fontawesome-free --dev
```

De nuevo, esto descargó las dos bibliotecas en nuestro proyecto... pero no las utiliza automáticamente todavía. Como esas bibliotecas contienen archivos CSS, vuelve a nuestro archivo `app.css` e impórtalos: `@import '~'` y luego `@fortawesome/fontawesome-free`. Y `@import '~@fontsource/roboto-condensed'`.

assets/styles/app.css

```
@import '~bootstrap';
@import '~@fortawesome/fontawesome-free';
@import '~@fontsource/roboto-condensed';

// ... lines 5 - 34
```

El primer paquete debería arreglar este icono... y el segundo debería hacer que la fuente cambie en toda la página. Observa el tipo de letra cuando refrescamos... ¡ha cambiado! Pero... los iconos siguen estando algo rotos.

Importar archivos específicos de node_modules/

Para ser totalmente honesto, no estoy seguro de por qué esto no funciona fuera de la caja. Pero la solución es bastante interesante. Mantén pulsado command en un Mac -o ctrl en caso contrario- y haz clic en esta cadena `fontawesome-free`.

Cuando usas esta sintaxis, va a tu directorio `node_modules/`, a`@fortawesome/fontawesome-free`... y entonces, si no pones ningún nombre de archivo después de esto, hay un mecanismo en el que esta biblioteca le dice a Webpack qué archivo CSS debe importar. Por defecto, importa este archivo `fontawesome.css`. Por alguna razón... eso no funciona. Lo que queremos es este `all.css`.

Y podemos importarlo añadiendo la ruta: `/css/all.css`. No necesitamos el archivo minificado porque Encore se encarga de minificar por nosotros.

assets/styles/app.css

```
@import '~bootstrap';
@import '~@fortawesome/fontawesome-free/css/all.css';
@import '~@fontsource/roboto-condensed';

// ... lines 5 - 34
```

Y ahora... ¡estamos de vuelta!

La principal razón por la que me encanta Webpack Encore y este sistema es que nos permite utilizar importaciones adecuadas. Incluso podemos organizar nuestro JavaScript en pequeños archivos -poniendo clases o funciones en cada uno- y luego importarlos cuando los necesitemos. Ya no son necesarias las variables globales.

Webpack también nos permite utilizar cosas más serias como React o Vue: incluso puedes ver, en `webpack.config.js`, los métodos para activarlos.

Pero, por lo general, me gusta utilizar una encantadora biblioteca de JavaScript llamada Stimulus. Y quiero hablarte de ella a continuación.

Chapter 19: Stimulus: Un JavaScript Sensato y Bonito

Quiero hablar de Stimulus. Stimulus es una pequeña pero encantadora biblioteca de JavaScript que me encanta. Y Symfony tiene un soporte de primera clase para ella. También es muy utilizada por la comunidad de Ruby on Rails.

SPA vs. Aplicaciones "tradicionales"

Hay dos filosofías en el desarrollo web. La primera es que devuelves el HTML de tu sitio, como hemos hecho en nuestra página de inicio y de navegación, y luego añades el comportamiento de JavaScript a ese HTML. La segunda filosofía es utilizar un marco de trabajo de JavaScript para construir todo tu HTML y JavaScript, lo que supone una aplicación de una sola página.

La solución correcta depende de tu aplicación, pero a mí me gusta mucho el primer enfoque. Y utilizando Stimulus -así como otra herramienta de la que hablaremos en unos minutos llamada Turbo- podemos crear aplicaciones altamente interactivas que se ven y se sienten tan responsivas como una aplicación de una sola página.

Tenemos un tutorial completo sobre Stimulus, pero vamos a probarlo. Ya puedes ver cómo funciona en el ejemplo de su documentación. Creas una pequeña clase JavaScript llamada controlador... y luego adjuntas ese controlador a uno o más elementos de la página. Y ya está Stimulus te permite adjuntar escuchas de eventos -como eventos de clic- y tiene otras cosas buenas.

Controladores Stimulus en nuestra aplicación

Tip

En versiones recientes de Symfony (y, especificamente, WebpackEncoreBundle v2), Stimulus ya no viene instalado con `symfony/webpack-encore-bundle`. Para instalarlo, ejecuta:
composer require symfony/stimulus-bundle

En nuestra aplicación, cuando instalamos Encore, nos dio un directorio `controllers/`. Aquí es donde vivirán nuestros controladores Stimulus. Y en `app.js`, importamos `bootstrap.js`. No es un archivo que tengas que mirar mucho, pero es súper útil. Esto pone en marcha Stimulus -sí, ya está instalado- y registra todo lo que hay en el directorio `controllers/` como un controlador Stimulus. Esto significa que si quieres crear un nuevo controlador Stimulus, ¡sólo tienes que añadir un archivo a este directorio `controllers/`!

Y obtenemos un controlador de Estímulos fuera de la caja llamado `hello_controller.js`. Todos los controladores de Estímulos siguen la práctica de nombrar algo con "guión bajo"`controller.js` o algo con guión `controller.js`. La parte que precede a `_controller` -por tanto, `hello` - se convierte en el nombre del controlador.

Adjuntar un controlador a un elemento

Adjuntemos esto a un elemento. Abre `templates/vinyl/homepage.html.twig`. Veamos... en la parte principal de la página, voy a añadir un div... y luego para adjuntar el controlador a este elemento, añade `data-controller="hello"`.

templates/vinyl/homepage.html.twig

```
↕ // ... lines 1 - 35
36             <div data-controller="hello"></div>
↕ // ... lines 37 - 59
```

¡Vamos a probarlo! Actualiza y... ¡sí! ¡Ha funcionado! El estímulo ha visto este elemento, ha instanciado el controlador... y luego nuestro código ha cambiado el contenido del elemento. El elemento al que está unido este controlador está disponible como `this.element`.

¡El estímulo ve dinámicamente nuevos elementos!

Así que... esto ya es muy bonito... porque conseguimos trabajar dentro de un objeto JavaScript ordenado... que está ligado a un elemento específico.

Pero déjame mostrarte la parte más genial de Stimulus: lo que hace que cambie el juego. Inspecciona el elemento en las herramientas de tu navegador cerca del elemento. Voy a modificar el HTML del elemento padre. Justo encima de éste -aunque no importa dónde- añade otro elemento con `data-controller="hello"`.

Y... ¡boom! ¡Vemos el mensaje! Esta es la característica estrella de Stimulus: puedes añadir estos elementos `data-controller` a la página cuando quieras. Por ejemplo, si haces una llamada Ajax... que añade HTML fresco a tu página, Stimulus se dará cuenta de ello y ejecutará los controladores a los que el nuevo HTML deba estar unido. Si alguna vez has tenido problemas en los que has añadido HTML a tu página mediante Ajax... pero el JavaScript de ese nuevo HTML está roto porque le faltan algunos escuchadores de eventos, pues Stimulus acaba de resolverlo.

La función stimulus_controller ()

Cuando usas Stimulus dentro de Symfony, obtenemos unas cuantas funciones de ayuda para hacernos la vida más fácil. Así, en lugar de escribir `data-controller="hello"` a mano, podemos decir `{{ stimulus_controller('hello') }}`.

templates/vinyl/homepage.html.twig

```
↕ // ... lines 1 - 35
36             <div {{ stimulus_controller('hello') }}></div>
↕ // ... lines 37 - 59
```

Pero eso es sólo un atajo para renderizar ese atributo exactamente igual que antes.

Bien, ahora que tenemos lo básico de Stimulus, vamos a utilizarlo para hacer algo real, como hacer una petición Ajax cuando hagamos clic en este icono de reproducción. Eso es lo siguiente.

Chapter 20: Ejemplo de Stimulus en el mundo real

Pongamos a prueba a Stimulus. Éste es nuestro objetivo: cuando hagamos clic en el icono de reproducción, haremos una petición Ajax a nuestra ruta de la API... la que está en `SongController`. Esto devuelve la URL donde se puede reproducir esta canción. Entonces usaremos eso en JavaScript para... ¡reproducir la canción!

Toma `hello_controller.js` y cámbiale el nombre a, qué tal `song-controls_controller.js`. Dentro, sólo para ver si esto funciona, en `connect()`, registra un mensaje. El método `connect()` se llama cada vez que Stimulus ve un nuevo elemento coincidente en la página.

assets/controllers/song-controls_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() {
        console.log('I just appeared into existence!');
    }
}
```

Ahora, en la plantilla, hola ya no va a funcionar, así que quita eso. Lo que quiero hacer es rodear cada fila de canciones con este controlador.... así que es este elemento `song-list`. Después de la clase, añade `{{ stimulus_controller('song-controls') }}`.

templates/vinyl/homepage.html.twig

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

{% block title %}Create a new Record | {{ parent() }}{% endblock %}

{% block body %}
<div class="container">
// ... lines 7 - 36
            {% for track in tracks %}
            <div class="song-list" {{ stimulus_controller('song-controls') }}>
// ... lines 39 - 50
            </div>
            {% endfor %}
// ... lines 53 - 55
</div>
{% endblock %}
```

Vamos a probarlo Actualiza, comprueba la consola y... ¡sí! Golpeó nuestro código seis veces! Una vez por cada uno de estos elementos. Y cada elemento recibe su propia instancia de controlador, por separado.

Añadir acciones de Stimulus

Bien, a continuación, cuando hagamos clic en reproducir, queremos ejecutar algún código. Para ello, podemos añadir una acción. Tiene este aspecto: en la etiqueta `a`, añade `{{ stimulus_action() }}` -otra función de acceso directo- y pásale el nombre del controlador al que estás adjuntando la acción - `song-controls` - y luego un método dentro de ese controlador que debe ser llamado cuando alguien haga clic en este elemento. ¿Qué te parece `play`.

templates/vinyl/homepage.html.twig

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

{% block title %}Create a new Record | {{ parent() }}{% endblock %}

{% block body %}
<div class="container">
// ... lines 7 - 36
            {% for track in tracks %}
            <div class="song-list" {{ stimulus_controller('song-controls') }}>
                <div class="d-flex mb-3">
                    <a href="#" {{ stimulus_action('song-controls', 'play') }}>
                        <i class="fas fa-play me-3"></i>
                    </a>
// ... lines 43 - 49
                </div>
            </div>
            {% endfor %}
// ... lines 53 - 55
</div>
{% endblock %}
```

Genial, ¿no? De vuelta en el controlador de la canción, ya no necesitamos el método `connect()`: no tenemos que hacer nada cada vez que veamos otra fila `song-list`. Pero sí necesitamos un método `play()`.

Y al igual que con los escuchadores de eventos normales, éste recibirá un objeto `event`... y entonces podremos decir `event.preventDefault()` para que nuestro navegador no intente seguir el clic del enlace. Para probar, `console.log('Playing!')`.

assets/controllers/song-controls_controller.js

```
import { Controller } from '@hotwired/stimulus';
// ... lines 2 - 11
export default class extends Controller {
    play(event) {
        event.preventDefault();

        console.log('Playing!');
    }
}
```

¡Vamos a ver qué pasa! Actualiza y... haz clic. Ya funciona. Así de fácil es enganchar un oyente de eventos en Stimulus. Ah, y si inspeccionas este elemento... esa

función `stimulus_action()` es sólo un atajo para añadir un atributo especial `data-action` que Stimulus entiende.

Instalar e importar Axios

Bien, ¿cómo podemos hacer una llamada Ajax desde dentro del método `play()`? Bueno, podríamos utilizar la función integrada `fetch()` de JavaScript. Pero en su lugar, voy a instalar una biblioteca de terceros llamada Axios. En tu terminal, instálala diciendo:

```
yarn add axios --dev
```

Ahora sabemos lo que hace: descarga este paquete en nuestro directorio `node_modules`, y añade esta línea a nuestro archivo `package.json`.

Ah, y nota al margen: puedes utilizar absolutamente jQuery dentro de Stimulus. No lo haré, pero funciona muy bien - y puedes instalar - e importar - jQuery como cualquier otro paquete. Hablamos de ello en nuestro tutorial de Stimulus.

Bien, ¿cómo utilizamos la biblioteca `axios`? Importándola

Al principio de este archivo, ya hemos importado la clase base `Controller` de `stimulus`. Ahora `import axios from 'axios'`. En cuanto lo hagamos, Webpack Encore cogerá el código fuente de `axios` y lo incluirá en nuestros archivos JavaScript construidos.

assets/controllers/song-controls_controller.js

```
// ... lines 1 - 11
import axios from 'axios';
// ... lines 13 - 21
```

Ahora, aquí abajo, podemos decir `axios.get()` para hacer una petición GET. Pero... ¿qué debemos pasar para la URL? Tiene que ser algo como `/api/songs/5`... pero ¿cómo sabemos cuál es el "id" de esta fila?

Valores de Stimulus

Una de las cosas más interesantes de Stimulus es que te permite pasar valores de Twig a tu controlador Stimulus. Para ello, declara qué valores quieres permitir que se pasen a través de una propiedad estática especial: `static values = {}`. Dentro, vamos a permitir que se pase un valor de `infoUrl`. Me acabo de inventar ese nombre: creo que pasaremos la URL completa a la ruta de la API. Establece esto como el tipo que será. Es decir, un `String`.

Aprenderemos cómo pasamos este valor desde Twig a nuestro controlador en un minuto. Pero como tenemos esto, abajo, podemos referenciar el valor diciendo `this.infoUrlValue`.

assets/controllers/song-controls_controller.js

```
↕ // ... lines 1 - 11
12 import axios from 'axios';
↕ // ... line 13
14 export default class extends Controller {
15     static values = {
16         infoUrl: String
17     }
↕ // ... line 18
19     play(event) {
↕ // ... lines 20 - 21
22         console.log(this.infoUrlValue);
23         //axios.get()
24     }
25 }
```

Entonces, ¿cómo lo pasamos? De vuelta en `homepage.html.twig`, añade un segundo argumento a `stimulus_controller()`. Este es un array de los valores que quieres pasar al controlador. Pasa a `infoUrl` el conjunto de la URL.

Hmm, pero tenemos que generar esa URL. ¿Esa ruta tiene ya un nombre? No, añade `name: 'api_songs_get_one'`.

src/Controller/SongController.php

```
<?php

// ... lines 3 - 10
class SongController extends AbstractController
{
    #[Route('/api/songs/{id<\d+>}', methods: ['GET'], name:
'api_songs_get_one')]
    public function getSong(int $id, LoggerInterface $logger): Response
    {
// ... lines 16 - 27
    }
}
```

Perfecto. Copia eso... y de nuevo en la plantilla, establece `infoURl` a `path()`, el nombre de la ruta... y luego una matriz con cualquier comodín. Nuestra ruta tiene un comodín `id`.

En una aplicación real, estas rutas probablemente tendrían cada una un id de base de datos que podríamos pasar. Todavía no lo tenemos... así que para, en cierto modo, falsear esto, voy a utilizar `loop.index`. Esta es una variable mágica de Twig: si estás dentro de un bucle de Twig `for`, puedes acceder al índice -como 1, 2, 3, 4- utilizando `loop.index`. Así que vamos a usar esto como una identificación falsa. Ah, y no olvides decir `id:` y luego `loop.index`.

templates/vinyl/homepage.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="container">
// ... lines 7 - 36
            {% for track in tracks %}
            <div class="song-list" {{ stimulus_controller('song-controls',
{
                infoUrl: path('api_songs_get_one', { id: loop.index })
            }) }}>
// ... lines 41 - 52
            </div>
            {% endfor %}
// ... lines 55 - 57
</div>
{% endblock %}
```

¡Hora de probar! Refresca. Lo primero que quiero que veas es que, cuando pasamos `infoUrl` como segundo argumento a `stimulus_controller`, lo único que hace es dar salida a un atributo muy especial `data` que Stimulus sabe leer. Así es como se pasa un valor a un controlador.

Haz clic en uno de los enlaces de reproducción y... lo tienes. ¡A cada objeto controlador se le pasa su URL correcta!

Hacer la llamada Ajax

¡Vamos a celebrarlo haciendo la llamada Ajax! Hazlo con `axios.get(this.infoUrlValue)` -sí, acabo de escribirlo-, `.then()` y una devolución de llamada utilizando una función de flecha que recibirá un argumento `response`. Esto se llamará cuando termine la llamada Ajax. Registra la respuesta para empezar. Ah, y corrige para usar `this.infoUrlValue`.

assets/controllers/song-controls_controller.js

```
1  import { Controller } from '@hotwired/stimulus';
2
↕  // ... lines 3 - 11
12 import axios from 'axios';
↕  // ... line 13
14 export default class extends Controller {
↕  // ... lines 15 - 18
19     play(event) {
20         event.preventDefault();
21
22         axios.get(this.infoUrlValue)
23             .then((response) => {
24                 console.log(response);
25             });
26     }
27 }
```

Muy bien, actualiza... ¡y haz clic en el enlace de reproducción! ¡Sí! Ha volcado la respuesta... y una de sus claves es `data`... ¡que contiene el `url`!

¡Es hora de dar la vuelta de la victoria! De vuelta a la función, podemos reproducir ese audio creando un nuevo objeto `Audio` -es un objeto JavaScript normal-, pasándole `response.data.url`... y llamando a continuación a `play()`.

assets/controllers/song-controls_controller.js

```
import { Controller } from '@hotwired/stimulus';
// ... lines 2 - 11
import axios from 'axios';

export default class extends Controller {
// ... lines 15 - 18
    play(event) {
        event.preventDefault();

        axios.get(this.infoUrlValue)
            .then((response) => {
                const audio = new Audio(response.data.url);
                audio.play();
            });
    }
}
```

Y ahora... cuando le demos al play... ¡por fin! Música para mis oídos.

Si quieres aprender más sobre Stimulus - esto ha sido un poco rápido - tenemos un tutorial entero sobre ello... y es genial.

Para terminar este tutorial, vamos a instalar otra biblioteca de JavaScript. Ésta hará que nuestra aplicación se sienta instantáneamente como una aplicación de una sola página. Eso a continuación.

Chapter 21: Turbo: Supercarga tu aplicación

Bienvenido al último capítulo de nuestro tutorial de introducción a Symfony 6. Si estás viendo esto, ¡lo estás petando! Y es hora de celebrarlo instalando un paquete más de Symfony. Pero antes de hacerlo, como sabes, me gusta confirmar todo primero... por si el nuevo paquete instala una receta interesante:

```
git add .
git commit -m "Never gonna let you go..."
```

Instalando symfony/ux-turbo

Bien, vamos a instalar el nuevo paquete:

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

¿Ves ese "ux" en el nombre del paquete? Symfony UX es un conjunto de bibliotecas que añaden funcionalidad JavaScript a tu aplicación... a menudo con algo de código PHP para ayudar. Por ejemplo, hay una biblioteca para renderizar gráficos... y otra para usar un Cropper de imágenes con el sistema de formularios.

Recetas UX de Symfony

Así que, como puedes ver, esto instaló una receta. OoOOo. Ejecuta

```
git status
```

para que podamos ver lo que ha hecho. La mayor parte es normal, como `config/bundles.php` que significa que habilitó el nuevo bundle. Los dos cambios interesantes son `assets/controllers.json` y `package.json`. Comprobemos primero `package.json`.

Cuando instalas un paquete UX, lo que suele significar es que te estás integrando con una biblioteca JavaScript de terceros. Y así, la receta de ese paquete añade esa biblioteca a tu código. En este caso, la biblioteca JavaScript con la que nos estamos integrando se llama `@hotwired/turbo`. Además, el propio paquete PHP `symfony/ux-turbo` viene con algo de JavaScript adicional. Esta línea especial dice

> *"¡Hey Node! Quiero incluir un paquete llamado `@symfony/ux-turbo`... pero en lugar de de descargarlo, puedes encontrar su código en el directorio directorio `vendor/symfony/ux-turbo/Resources/assets`."*

Puedes buscar literalmente en esa ruta `vendor/symfony/ux-turbo/Resources/assets` para encontrar un mini paquete JavaScript. Ahora, debido a que esto actualizó nuestro archivo `package.json`, tenemos que volver a instalar nuestras dependencias para descargarlo y tenerlo todo listo.

De hecho, busca tu terminal que está ejecutando `yarn watch`. Tenemos un error! Dice que no se puede encontrar el archivo `@symfony/ux-turbo/package.json`, intenta ejecutar `yarn install --force`.

¡Vamos a hacerlo! Pulsa control+C para detener esto... y luego ejecuta

```
yarn install --force
```

o `npm install --force`. Luego, reinicia Encore con:

```
yarn watch
```

El otro archivo que la receta modificó fue `assets/controllers.json`. Vamos a echarle un vistazo: `assets/controllers.json`. Esta es otra cosa que es exclusiva de Symfony UX. Normalmente, si queremos añadir un controlador Stimulus, lo ponemos en el

directorio `controllers/`. Pero a veces, puede que instalemos un paquete PHP y que queramos añadir su propio controlador Stimulus en nuestra aplicación. Esta sintaxis dice básicamente

> *"¡Hey Stimulus! Ve a cargar este controlador Stimulus desde ese nuevo `@symfony/ux-turbo` paquete."*

Ahora bien, este controlador Stimulus en particular es un poco raro. No es uno que vayamos a utilizar directamente dentro de la función `stimulus_controller()` Twig. Es una especie de controlador falso. ¿Qué hace? Sólo con que se cargue, va a activar la biblioteca Turbo.

¡Hola Turbo! Por la actualización de la página completa

Sigo hablando de Turbo. ¿Qué es Turbo? Bueno, al ejecutar ese comando composer require... y luego reinstalar yarn, el JavaScript de Turbo está ahora activo y funcionando en nuestro sitio. ¿Qué hace? Es sencillo: convierte cada clic en un enlace y cada envío de un formulario de nuestro sitio en una llamada Ajax. Y eso hace que nuestro sitio sea rápido como un rayo.

Compruébalo. Haz una última actualización completa. Y luego observa... si hago clic en Examinar, ¡no hay actualización completa de la página! Si hago clic en estos iconos, ¡no hay actualización! Turbo intercepta esos clics, hace una llamada Ajax a la URL, y luego pone ese HTML en nuestro sitio. Esto es enorme porque, de repente, nuestra aplicación se ve y se siente como una aplicación de una sola página... ¡sin que nosotros hagamos nada!

La barra de herramientas de depuración web y el perfilador de peticiones Ajax

Ahora, otra cosa interesante que notarás es que, aunque las recargas de páginas completas han desaparecido, estas llamadas Ajax aparecen en la barra de herramientas de depuración web. Y puedes hacer clic para ir a ver el perfil de esa llamada Ajax muy fácilmente. Esta parte de la barra de herramientas de depuración web es aún más útil con las llamadas Ajax para una ruta de la API. Si pulsamos el icono de reproducción... ese 7 acaba de subir a 8... ¡y aquí está el perfilador de esa petición de la API! Abriré ese enlace en una nueva ventana. Esa es una forma súper fácil de llegar al perfilador de cualquier petición Ajax.

Así que Turbo es increíble... y puede hacer más. Hay algunas cosas que debes saber sobre él antes de enviarlo a producción, y si te interesa, ¡sí! tenemos un tutorial completo sobre Turbo. Quería mencionarlo en este tutorial porque Turbo es más fácil si lo añades a tu aplicación desde el principio.

Muy bien, ¡felicidades! ¡El primer tutorial de Symfony 6 está en los libros! Date una palmadita en la espalda... o mejor, busca a un amigo y choca los cinco.

¡Y sigue adelante! Acompáñanos en el siguiente tutorial de esta serie, que te hará pasar de ser un desarrollador de Symfony en ciernes a alguien que realmente entiende lo que está pasando. Cómo funcionan los servicios, el sentido de todos estos archivos de configuración, los entornos Symfony, las variables de entorno y mucho más. Básicamente todo lo que necesitarás para hacer lo que quieras con Symfony.

Y si tienes alguna pregunta o idea, estamos aquí para ti en la sección de comentarios debajo del vídeo.

Muy bien amigos, ¡hasta la próxima!

With <3 from SymfonyCasts