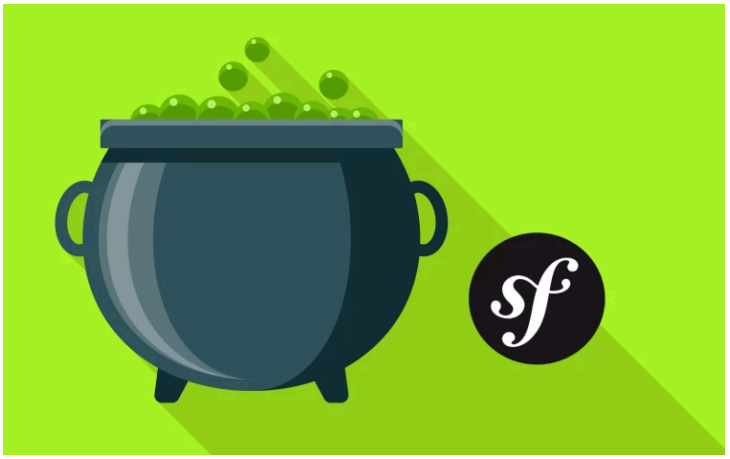
Desarrollo Encantador en Symfony 5



Chapter 1: Creando un Nuevo Proyecto de Symfony 5

Hola Amigos! y *bienvenidos* al mundo de Symfony 5... el cual *resulta* ser mi mundo favorito! Ok, quizás Disneylandia es mi mundo *favorito*... pero programar en Symfony 5 está en *segundo* lugar...

Symfony 5 es simple y eficiente: es muy veloz, empieza en pequeño, pero crece conforme a tu aplicación. Y esto *no* es solo jerga de Marketing! Tu aplicación de Symfony *literalmente* crecerá conforme necesites más funcionalidades. Ya hablaremos de eso más tarde.

Symfony 5 es *también* el producto de *años* de trabajo sobre experiencia de desarrollo. Básicamente, la gente detrás de Symfony quiere que *ames* utilizarlo sin sacrificar calidad. Asi es, escribes código del cual estás orgulloso, amas el proceso, *y* construyes cosas rápidamente.

Symfony es también el framework *más rápido* de PHP, lo cual no nos sorprende: - su creador *también* creó el sistema de análisis de rendimiento de PHP llamado Blackfire. Por lo que el rendimiento siempre está en la mira.

Go Deeper!

Mira nuestro Blackfire.io: Revealing Performance Secrets with Profiling curso sobre Blackfire.

Descargando el instalador de Symfony

Entonces... Manos a la obra! Empieza por abrir http://symfony.com y dar click en "Download". Lo que estamos *apunto* de descargar *no* es realmente Symfony. Es un ejecutable que va a hacer que tu experiencia de desarrollo con Symfony sea... Excelente.

Como estoy en una Mac, voy a copiar este comando. para luego abrir una terminal - yo ya tengo una abierta. No importa en *donde* lo ejecutes. Pégalo!

```
curl -sS https://get.symfony.com/cli/installer | bash
```

Esto *simplemente* descarga un archivo ejecutable y, para mi, lo guarda en mi carpeta home. Para poder hacerlo ejecutable en *cualquier* lugar en el sistema, Voy a seguir el consejo del comando y lo moveré a otro lugar:

```
mv /Users/weaverryan/.symfony/bin/symfony /usr/local/bin/symfony
```

Ok, inténtalo!

```
symfony --version
```

Symfony está vivo! Saluda al CLI de Symfony: una herramienta de linea de comandos que nos va a ayudar con varias cosas a lo largo de nuestro camino hacia la gloria de programación.

Empezando una nueva aplicación de Symfony

Su *primer* trabajo será ayudarnos en crear un nuevo projecto de Symfony 5. Ejecuta:

```
symfony new cauldron_overflow
```

Donde `cauldron_overflow` será el *directorio* donde la nueva aplicación vivirá. Este *también* resulta ser el nombre del sitio que vamos a construir... Pero ya hablaremos de eso más tarde.

Detrás de escenas, este comando no está haciendo nada especial: clona un repositorio de Git llamado `symfony/skeleton` y luego utiliza Composer para instalar las dependencias del proyecto. Hablaremos más sobre ese repositorio *y* de Composer un poco más adelante.

Cuando termine, muévete al nuevo directorio:

```
cd cauldron_overflow
```

Y luego *ábrelo* en tu editor favorito. Yo ya lo tengo abierto en *mi* editor favorito: PhpStorm, solo abre Archivo -> Abrir Directorio y selecciona la carpeta del nuevo proyecto. En fin, saluda a tu totalmente nuevo, brillante, prometedor proyecto de Symfony 5.

Nuestra aplicación es diminuta!

Antes de comenzar a mover aquí y allá, vamos a crear un nuevo repositorio de git y hacer un commit. Pero espera... Ejecuta:

```
git status
```

> *"En la rama master, nada por hacer commit."*

Sorpresa! El comando `new` de Symfony ya inicializó el repositorio de Git por nosotros e hizo el primer commit. Puedes verlo tras ejecutar:

```
git log
```

> *"Add initial set of files"*

Perfecto! Aunque, personalmente me hubiera gustado un mensaje ligeramente más épico... pero está bien.

Voy a oprimir "q" para salir.

Mencioné anteriormente que Symfony empieza en *pequeño*. Para probarlo, podemos ver una lista de *todos* los archivos agregados en el commit. Tras ejecutar:

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

Eso es! Nuestro proyecto, el cual está *completamente* listo para trabajar con Symfony tiene menos de 15 archivos... si no cuentas archivos como `.gitignore`. Simple y eficiente.

Revisando los Requerimientos

Conectemos un servidor web a nuestra aplicación y veámoslo en acción! Primero, asegurate que tu computadora tenga todo lo que necesita Symfony al ejecutar:

```
symfony check:req
```

Para revisar los requerimientos. Estamos bien - pero si tienes algún problema y necesitas ayuda, menciónalo en los comentarios.

Iniciando el Servidor Web de PHP

Para poner el proyecto en marcha, regresa a PhpStorm. Vamos a hablar más sobre cada directorio pronto. Pero la *primer* cosa que tienes que saber es que el directorio `public/` es el "documento raíz". Esto significa que necesitas apuntar tu servidor web - como Apache o Nginx - a este directorio. Symfony tiene documentación sobre como hacerlo.

Pero! para facilitarnos la vida, en vez de configurar un servidor web en *nuestra* maquina, podemos usar el servidor integrado de PHP. En la raíz de to proyecto, ejecuta:

```
php -S 127.0.0.1:8000 -t public/
```

Tan pronto hacemos eso Podemos regresar a nuestro navegador e ir a http://localhost:8000 para descubrir... Bienvenido a Symfony 5! Ooh, que elegancia!

Siguiente: tan *fácil* como fue ejecutar ese servidor web de PHP, Voy a mostrarte aun una *mejor* opción para el desarrollo local. Ahora vamos a conocer el *significado* de los directorios en

nuestra nueva aplicación *y* asegurarnos de que tenemos algunos plugins instalados en PhpStorm... los cuales hacen trabajar con Symfony todo un placer.

Chapter 2: Bienvenidos a nuestra pequeña Aplicación y setup de PhpStorm

Uno de mis objetivos principales en este tutorial será ayudarte a entender *realmente* cómo funciona Symfony, tu aplicación.

Para empezar, echemos un vistazo a la estructura de carpetas.

El Directorio public/

Hay solo 3 directorios que debes tener en cuenta. Primero, `public/` es el documento raíz: el cual contendrá todos los archivos que deben ser accesibles por un navegador. Y... por ahora hay uno solo: `index.php`:

public/index.php

```
// ... lines 1 - 2
use App\Kernel;
use Symfony\Component\ErrorHandler\Debug;
use Symfony\Component\HttpFoundation\Request;

require dirname(__DIR__).'/config/bootstrap.php';

if ($_SERVER['APP_DEBUG']) {
    umask(0000);

    Debug::enable();
}

if ($trustedProxies = $_SERVER['TRUSTED_PROXIES'] ?? $_ENV['TRUSTED_PROXIES'] ?? false) {
    Request::setTrustedProxies(explode(',', $trustedProxies), Request::HEADER_X_FORWARDED_ALL ^ Request::HEADER_X_FORWARDED_HOST);
}

if ($trustedHosts = $_SERVER['TRUSTED_HOSTS'] ?? $_ENV['TRUSTED_HOSTS'] ?? false) {
    Request::setTrustedHosts([$trustedHosts]);
}

$kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

Éste se llama el "front controller": un término complejo que los programadores inventaron para decir que este archivo es el que ejecuta tu servidor web.

Pero, en realidad, salvo poner archivos CSS o imágenes en `public/`, casi nunca tendrás que pensar en ello.

src/ y config/

Así que... En realidad te mentí. Hay en verdad *sólo dos* directorios que debes tener en cuenta: `config/` y `src/`. `config/` tiene... ehh... perritos? No, `config/` tiene archivos de configuración y `src/` es donde tu código PHP estará. Es así de simple.

Dónde está Symfony? Nuestro proyecto *comenzó* con un archivo `composer.json`:

composer.json

```
{
    "type": "project",
    "license": "proprietary",
    "require": {
        "php": "^7.2.5",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "symfony/console": "5.0.*",
        "symfony/dotenv": "5.0.*",
        "symfony/flex": "^1.3.1",
        "symfony/framework-bundle": "5.0.*",
        "symfony/yaml": "5.0.*"
    },
    "require-dev": {
    },
// ... lines 16 - 64
}
```

el cual contiene todas las librerías de terceros que nuestra aplicación *necesita*. Detrás de escenas, el comando `symfony new` utilizó a composer para instalarlas... Lo cual es una forma *sofisticada* de decir que Composer descargó un montón de librerías dentro del directorio `vendor/`... Incluyendo Symfony.

Más adelante hablaremos de los otros archivos y directorios, pero éstos todavía no nos importan.

Utilizando el Servidor Web Local de Symfony

Hace algunos minutos, usamos PHP para iniciar un servidor web local. Bien. Pero presiona Ctrl+C para salir del mismo. Por qué? Porque esa herramienta binaria symfony que instalamos viene con un servidor local mucho mas poderoso.

Ejecuta:

```
symfony serve
```

Eso es todo. La primera vez que lo corres, podría preguntarte sobre instalar un certificado. Esto es opcional. Si lo instalas - yo lo hice - iniciará el servidor web con https. Sip, tienes https local con cero esfuerzo. Una vez que corre, ve a tu navegador y refresca. Funciona! Y ese pequeño candado prueba que estamos usando https.

Para parar el servidor, solo presiona Control + C. Puedes ver todas estas opciones de comando al ejectuar:

```
symfony serve --help
```

Como por ejemplo, formas de controlar el número de puerto

Cuando uso este comando, usualmente corro:

```
symfony serve -d
```

-d significa correr como un daemon. Hace exactamente lo mismo excepto que *ahora* corre en segundo plano... Lo que significa que puedo seguir usando esta terminal. Si corro:

```
symfony server:status
```

Me muestra que el servidor está corriendo y

```
symfony server:stop
```

Lo apagará. Iniciemoslo de nuevo:

```
symfony serve -d
```

Instalando Plugins de PhpStorm

Ok: estamos a punto de comenzar a escribir *un montón* de código... ai que quiero asegurarme de que tu editor está listo para trabajar. Y, claro, puedes usar cualquier editor que tu quieras. Pero mi *mejor* recomendación es PhpStorm! En serio, hace que desarrollar en Symfony sea un *sueño*! Y no, las buenas personas de PhpStorm no me están pagando para decir esto... aunque... *sí* patrocinan a varios desarrolladores de código libre en PHP... lo que lo hace aún mejor.

Para tener un *fantástico* PhpStorm, tienes que hacer dos cosas. Primero, abre Preferencias, selecciona "Complementos" y click en "Marketplace". Haz una búsqueda por "Symfony". Este plugin es *increíble*... probado por casi 4 millones de descargas. Esto nos dará *toda clase* de auto-completes e inteligencia extra para Symfony.

Si no lo tienes aún, instálalo. Deberías también instalar los plugins "PHP Annotations" y "PHP toolbox". Si realizas una búsqueda por "php toolbox"... puedes ver los tres de ellos. Instálalos y luego reinicia PhpStorm.

Una vez reiniciado, vuelve a Preferencias y haz una búsqueda por Symfony. Además de instalar este plugin, tienes que habilitarlo en cada proyecto. Haz click en Habilitar y luego Aplicar. Dice que tienes que reiniciar PhpStorm... pero no creo que eso sea necesario.

La *segunda* cosa que tienes que hacer en PhpStorm es buscar Composer y encontrar la sección "Idiomas y Frameworks", "PHP", "Composer". Asegúrate de que la opción "Sincronizar ajustes IDE con composer.json" está activada... lo cual automáticamente configura algunas funciones útiles.

Haz click en "Ok" y... estamos listos! A continuación, vamos a crear nuestra primerísima página y veremos de qué se trata symfony.

Chapter 3: Rutas, Controladores & Respuestas!

La página que estamos viendo ahora... la cual es súper divertida... e incluso cambia de color... está aquí *solo* para decir "Hola!". Symfony muestra está página porque, en realidad, nuestra aplicación aun no tiene *ninguna* página. Cambiemos eso.

Ruta + Controlador = Página

Cada framework web... en *cualquier* lenguaje... tiene la misma labor principal: brindarte un sistema de ruteo -> controlador: un sistema de dos pasos para construir páginas. La ruta define la URL de la página y en él controlador es donde escribimos código PHP para *construir* esa página, como HTML ó JSON.

Abre `config/routes.yaml`:

config/routes.yaml

```
#index:
#    path: /
#    controller: App\Controller\DefaultController::index
```

Mira! ya tenemos un ejemplo! Descomentarízalo. Si no te es familiar el formato YAML, es súper amigable: es un formato de configuración tipo llave-valor que se separa mediante dos puntos. La identación también es importante.

Esto crea una simple ruta donde la URL es `/`. El controlador apunta a una *función* que va a *construir* esta página... en realidad, esto apunta a un método de una clase. En general, esta ruta dice:

> *"cuando el usuario vaya a la homepage, por favor ejecuta el método `index` de la clase DefaultController."*

Ah, y puedes ignorar esa llave `index` que está al principio del archivo: es solo el nombre interno de la ruta... y aun no es importante.

Nuestra Aplicación

El proyecto que estamos construyendo se llama "Cauldron Overflow". *Originalmente* queríamos crear un sitio donde los desarrolladores puedan hacer preguntas y otros desarrolladores pudieran responderlas pero... alguien ya nos ganó... hace como... unos 10 años. Así como cualquier otro impresionante startup, estamos pivoteando! Hemos notado que muchos magos accidentalmente se han hecho explotar... o invocan dragones que exhalan fuego cuando en realidad querían crear una pequeña fogata para azar malvaviscos. Así que... Cauldron Overflow está aquí para convertirse en *el* lugar donde magos y hechiceros pueden preguntar y responder sobre desventuras mágicas.

Creando un Controlador

En la homepage, eventualmente vamos a listar algunas de las preguntas más recientes. Así que vamos a cambiar la clase del controlador a `QuestionController` y el método a `homepage`.

config/routes.yaml

```
1 index:
2     path: /
3     controller: App\Controller\QuestionController::homepage
```

Ok, la ruta está lista: define la URL y apunta al controlador que va a construir la página. Ahora... necesitamos crear ese controlador! Dentro del directorio `src/` ya existe el directorio `Controller/` pero está vacío. Haré click derecho aquí y seleccionaré "Nueva clase PHP". Llamalo `QuestionController`.

Namespaces y el Directorio src/

Ooh, mira esto. El *namespace* ya está ahí! Sorprendente! Esto es gracias a la configuración de Composer en el PhpStorm que agregamos en el último capítulo.

Así está la cosa: cada clase que creamos dentro del directorio `src/` va a requerir un namespace. Y... por alguna razón que no es muy importante, el namespace debe iniciar con `App\` y continuar con el nombre del directorio donde vive el archivo. Como estamos creando este archivo dentro del directorio `Controller/`, su namespace debe ser `App\Controller.` PhpStorm va a autocompletar esto siempre.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
↕ // ... lines 4 - 6
7 class QuestionController
8 {
↕ // ... lines 9 - 12
13 }
```

Perfecto! Ahora, porque en `routes.yaml` decidimos nombrar al método `homepage`, crealo aquí: `public function homepage()`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 2
3 namespace App\Controller;
↕ // ... lines 4 - 6
7 class QuestionController
8 {
9     public function homepage()
10     {
↕ // ... line 11
12     }
13 }
```

Los Controladores Deben Regresar Una Respuesta

Y... felicitaciones! Estás dentro de una función del controlador, el cual algunas veces es llamado "acción"... solo para confundirnos. Nuestro trabajo aquí es simple: construir esa página. Podemos escribir `cualquier` código para hacerlo - como ejecutar queries en la base de datos, cachear cosas, realizar llamados a APIs, minar criptomonedas... lo que sea. La `única` regla es que la función del controlador `debe` regresar un objeto del tipo Symfony `Response`.

Escribe `return new Response()`. PhpStorm intenta autocompletar esto... pero existen multiples clases `Response` en nuestra app. La que queremos es la `Symfony\Component\HttpFoundation`. HttpFoundation es una de las partes - o "componentes" - más importantes en Symfony. Presiona tab para autocompletarlo.

Pero detente! Viste eso? Como dejamos que PhpStorm autocompletara esa clase por nosotros, escribió `Response`, pero *también* agregó la *declaración* de esa clase al principio del archivo! Esa es una de las *mejores* funciones de PhpStorm y lo utilizaré *bastante*. Me verás

constantemente escribir una clase y dejar que PhpStorm la autocomplete. Para que agregue la *declaración* en el archivo por mi.

Dentro de `new Response()`, agrega algo de texto:

> *"Pero qué controlador tan embrujado hemos conjurado!"*

src/Controller/QuestionController.php

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

use Symfony\Component\HttpFoundation\Response;

class QuestionController
{
    public function homepage()
    {
        return new Response('What a bewitching controller we have conjured!');
    }
}
```

Y... listo! Acabamos de crear nuestra primera página! Vamos a probarla! Cuando vamos a la homepage, debería ejecutar nuestra función del controlador... la cual regresa el mensaje.

Encuentra tu navegador. Ya estamos en la homepage... así que solo refresca. Saluda a nuestra *primerísima* página. Lo sé, no hay mucho que ver aun, pero acabamos de cubrir la parte más *fundamental* de Symfony: el sistema ruta-controlador.

A continuación, hagamos nuestra ruta más elegante al usar algo llamado anotaciones. También vamos a crear una segunda página con una ruta que utiliza *comodines*.

Chapter 4: Anotaciones y Rutas con Comodín

Es muy sencillo crear una ruta en YAML que apunte a una función del controlador. Pero hay una forma aun más *simple* de crear rutas... y me *encanta*. Se llama: anotaciones.

Primero, comenta la ruta en YAML. Básicamente, borrala. Para comprobar que no funciona, refresca la homepage. Asi es! Regresó a la página de bienvenida.

config/routes.yaml

```
#index:
#    path: /
#    controller: App\Controller\QuestionController::homepage
```

Instalación Soporte a Anotaciones

Las anotaciones son un formato especial de configuración y el soporte a anotaciones *no* es un standard en nuestra pequeña aplicación de Symfony. Y... eso está bien! De hecho, esa es *toda* la filosofía de Symfony: empieza pequeño y agrega funcionalidades cuando las necesites.

Para agregar soporte a anotaciones, vamos a utilizar Composer para requerir una nueva librería. Si aun no tienes Composer instalado, ve a https://getcomposer.org.

Una vez que lo *instales*, corre:

```
composer require annotations
```

Si estás familiarizado con Composer, el nombre de la librería se te ha de hacer extraño. Y en realidad, instaló una librería totalmente *diferente*: `sensio/framework-extra-bundle`. Casi al final del comando, menciona algo sobre dos recetas. Hablaremos sobre ello próximamente: es parte de lo que hace especial a Symfony.

Agregando Rutas con Anotaciones

En fin, ya que el soporte a anotaciones está instalado, podemos agregar de vuelta nuestra ruta usando anotaciones. Que significa eso? Arriba de la función del controlador, escribe `/**` y presiona enter para crear una sección PHPDoc Luego escribe `@Route` y autocompleta la del componente Routing. Tal como la otra vez, PhpStorm agregó automáticamente el `use` statement en la parte de arriba de la clase.

Dentro de los paréntesis, escribe `"/"`.

src/Controller/QuestionController.php

```
// ... lines 1 - 5
use Symfony\Component\Routing\Annotation\Route;

class QuestionController
{
    /**
     * @Route("/")
     */
    public function homepage()
    {
// ... line 15
    }
}
```

Eso es todo! Cuando el usuario vaya a la homepage, se va a ejecutar la función abajo de esto. Me *encantan* las anotaciones porque son simples de leer y mantienen la ruta y controlador uno junto del otro. Y si... las anotaciones son *literalmente* configuración dentro de comentarios de PHP. Si no te gustan, siempre puedes utilizar YAML o XML: Symfony es super flexible. Desde el punto de vista del rendimiento, todos los formatos son lo mismo.

Ahora cuando refrescamos la homepage... estamos de vuelta!

Una Segunda Ruta y Controlador

Esta página eventualmente va a listar algunas preguntas recientes. Cuando le das click a una pregunta en específico, necesitará su *propia* página. Vamos a crear una segunda ruta y controlador para ello. Como? creando un segundo metodo. Que tal: `public function show()`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 7
8  class QuestionController
9  {
↕ // ... lines 10 - 20
21     public function show()
22     {
↕ // ... line 23
24     }
25 }
```

Arriba de esto, agrega `@Route()` y asigna la URL a, que te parece, `/questions/how-to-tie-my-shoes-with-magic`. Eso seria grandioso!

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 7
8  class QuestionController
9  {
↕ // ... lines 10 - 17
18     /**
19      * @Route("/questions/how-to-tie-my-shoes-with-magic")
20      */
21     public function show()
22     {
↕ // ... line 23
24     }
25 }
```

Adentro, justo como la última vez, retorna una nueva *respuesta*: la de `HttpFoundation.`

> *"La página futura para mostrar preguntas"*

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 7
8  class QuestionController
9  {
↕ // ... lines 10 - 17
18     /**
19      * @Route("/questions/how-to-tie-my-shoes-with-magic")
20      */
21     public function show()
22     {
23         return new Response('Future page to show a question!');
24     }
25 }
```

Vamos a probarla! Copia la URL, ve a tu navegador, pega y... funciona! Acabamos de crear una *segunda* página... en menos de un minuto.

El Controlador Frontal: Trabajando Detrás De Cámaras

Por cierto, no importa a cual URL vayamos - como esta o la homepage - el archivo PHP que nuestro servidor web ejecuta es `index.php`. Es como *si* fuéramos a `/index.php/questions /how-to-tie-my-shoes-with-magic`. La única razón por la que no *necesitas* escribir `index.php` en la URL es porque nuestro servidor web local está configurado para ejecutar `index.php` automáticamente. En producción, tu configuración de Nginx o Apache debe de hacer lo mismo. Revisa la documentación de Symfony para aprender como hacerlo.

Rutas con Comodín

Eventualmente, vamos a tener una base de datos *llena* de preguntas. Y entonces, no, *no* vamos a crear manualmente una ruta por cada pregunta. En su lugar, podemos hacer más inteligente esta ruta. Reemplaza la parte `how-to-tie-my-shoes-with-magic` por `{slug}`.

Cuando pones algo entre llaves dentro de una ruta, se convierte en *comodín*. Esta ruta ahora aplica a `/questions/LO-QUE-SEA`. El nombre `{slug}` no es importante: pudimos haber puesto lo que sea... por ejemplo `{slugulusErectus}`! No hace ninguna diferencia.

Pero, *como sea* que llamemos a este comodín - ejemplo `{slug}` - ahora nos *permite* tener un argumento en nuestro controlador con el mismo *nombre*: `$slug`... el cual será asignado con el valor de esa parte de la URL.

src/Controller/QuestionController.php

```
// ... lines 1 - 7
class QuestionController
{
// ... lines 10 - 17
    /**
     * @Route("/questions/{slug}")
     */
    public function show($slug)
    {
// ... lines 23 - 26
    }
}
```

Utilicemoslo para hacer mas elegante a nuestra página! Usemos `sprintf()`, escribe " la pregunta" y agrega `%s` como comodín. Pasa `$slug` como comodín.

src/Controller/QuestionController.php

```
// ... lines 1 - 7
class QuestionController
{
// ... lines 10 - 17
    /**
     * @Route("/questions/{slug}")
     */
    public function show($slug)
    {
        return new Response(sprintf(
            'Future page to show the question "%s"!',
            $slug
        ));
    }
}
```

Bien! Cambia al navegador, refresca y... me encanta! Cambia la URL a `/questions /accidentally-turned-cat-into-furry-shoes` y... eso también funciona!

En el futuro, vamos a utilizar el `$slug` para extraer la pregunta de la base de datos. Pero como aun no llegamos ahí, usaré `str_replace()` ... y `ucwords()` *solo* para hacerlo un poco mas elegante. Aun es pronto, pero la página ya *comienza* a estar viva!

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 7
8  class QuestionController
9  {
↕ // ... lines 10 - 20
21     public function show($slug)
22     {
23         return new Response(sprintf(
↕ // ... line 24
25             ucwords(str_replace('-', ' ', $slug))
26         ));
27     }
28 }
```

A continuación, nuestra aplicación esconde un secreto! Una pequeña línea de comandos ejecutable que está *llena* de beneficios.

Chapter 5: La amada herramienta bin/console

Guardemos nuestro progreso hasta ahora. Voy a limpiar la pantalla y ejecutaré:

```
git status
```

Interesante: Hay algunos archivos *nuevos* aquí que yo no creé. No te preocupes: Vamos a hablar *precisamente* de eso en el siguiente capítulo. Agrega todo con:

```
git add .
```

Normalmente... Este comando puede ser peligroso - accidentalmente podríamos agregar algunos archivos que *no* queremos al commit! Afortunadamente, nuestro proyecto viene con un archivo `.gitignore` precargado que ignora cosas importantes como `vendor/` y otras rutas de las cuales hablaremos más tarde. Por ejemplo, `var/` contiene el caché y los archivos de logs. El punto es, que Symfony nos cuida la espalda.

Guarda los cambios con:

```
git commit -m "Lo estamos haciendo en grande con esto de Symfony"
```

Hola comando bin/console

Puedes interactuar de *dos* maneras diferentes con tu aplicación de Symfony. La primera es al cargar una página en tu navegador. La *segunda* es con un útil comando llamado `bin/console`. En tu terminal, ejecuta:

```
php bin/console
```

¡Orale! Este comando enlista un *montón* de cosas diversas que puedes hacer con eso, incluidas *múltiples* herramientas de depuración. Ahora, para desmitificar este asunto un poco, existe `literalmente` un directorio `bin/` en nuestra aplicación con un archivo llamado `console` adentro. Así que esta cosa `bin/console` no es un comando global que se ha instalado en nuestro sistema: estamos, literalmente ejecutando un archivo PHP.

El comando `bin/console` puede hacer *muchas* cosas - y descubriremos mis características favoritas a lo largo del camino. Por ejemplo, ¿Quieres ver un listado para *cada* ruta en tu aplicación? Ejecuta:

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

¡Sip! Ahí están nuestras *dos* rutas... además de otra que Symfony agrega automáticamente durante el desarrollo.

La herramienta `bin/console` contiene *muchos* comandos útiles como este. Pero la lista de comandos que soporta *no* es estática. Nuevos comandos pueden ser agregados por *nosotros*... *O* por nuevos paquetes que instalemos en nuestro proyecto. Este es mi "no tan sutil" presagio.

A continuación: Hablemos de Symfony Flex, alias con Composer y el sistema de recetas. Básicamente, las herramientas que hacen a Symfony verdaderamente único.

Chapter 6: Flex, Recetas & Aliases

Vamos a instalar un paquete *totalmente* nuevo dentro de nuestra aplicación llamado "security checker". El verificador de seguridad es una herramienta que revisa las dependencias de tu aplicación y te dice si alguna de estas tiene vulnerabilidades de seguridad conocidas. Pero, confidencialmente, tan *genial* como lo es..., la razón *real* por la que quiero instalar esta librería es porque es una *gran* manera de ver el importantísimo sistema de "recetas" de Symfony.

En tu terminal, ejecuta:

```
composer require sec-checker
```

En una aplicación real, probablemente *deberías* pasar `--dev` y agregar esto a tu dependencia *dev*... pero eso no nos preocupa a nosotros.

Flex Aliases

No obstante, *hay* algo extraño aquí. Específicamente... `sec-checker` *no* es un nombre de paquete válido! En el mundo de Composer, *cada* paquete *debe* ser `algo/algo-más`: no puede ser solamente `sec-checker`. Entonces que diantres está pasando?

De devuelta en PhpStorm, abre `composer.json`. Cuando iniciamos el proyecto, solamente teníamos unas *pocas* dependencias en este archivo. Una de ellas es `symfony/flex`.

composer.json

```
{
// ... lines 2 - 3
    "require": {
        "php": "^7.2.5",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "sensio/framework-extra-bundle": "^5.5",
        "sensiolabs/security-checker": "^6.0",
        "symfony/console": "5.0.*",
        "symfony/dotenv": "5.0.*",
        "symfony/flex": "^1.3.1",
        "symfony/framework-bundle": "5.0.*",
        "symfony/yaml": "5.0.*"
    },
// ... lines 16 - 67
}
```

Este es un *plugin* de composer que agrega *dos* características especiales al mismo Composer. El primero se llama "aliases".

En tu navegador, ve a http://flex.symfony.com para encontrar una larga página llena de paquetes. Busca por `security`. Mejor, busca por `sec-checker`. Bingo! La misma dice que hay un paquete llamado `sensiolabs/security-checker` y tiene los aliases de `sec-check`, `sec-checker`, `security-checker` y algunos más.

El sistema de alias es simple: pues Symfony Flex se encuentra en nuestra aplicación, podemos decir `composer require security-checker`, y *realmente* descargará `sensiolabs/security-checker`.

Puedes ver esto en nuestra termina: dijimos `sec-checker`, pero al final descargó `sensiolabs/security-checker`. Eso es algo que también Composer agregó a nuestro archivo `composer.json` Entonces... las aliases son una agradable característica de atajo... pero es realmente genial! Casi que puedes *adivinar* un alias cuando quieras instalar algo. Necesitas una bitácora? Ejecuta `composer require logger` para conseguir la bitácora recomendada. Necesitas enviar algo por correo electrónico? `composer require mailer` Necesitas comer un pastel? `composer require cake`!

Recetas de Flex

La *segunda* característica que Flex agrega a Composer es la más *importante*. Es el sistema de recetas

En la terminal, después de instalar el paquete, nos menciona:

> *"Symfony operations: 1 recipe configuring sensiolabs/security-checker."*

Interesante. Ejecuta:

```
git status
```

Wow! Esperábamos que `composer.json` y `composer.lock` fueran modificados... así es como Composer trabaja. Pero algo *también* modificó al archivo `symfony.lock`... y agregó un archivo totalmente nuevo `security_checker.yaml`!

Muy bien, primero `symfony.lock` es un archivo que es manejado por Flex. Tú no necesitas preocuparte por el, pero *deberías* asignarlo. Mantiene una gran lista de cuáles recetas se han instalado.

Entonces, ¿Quién creó el otro archivo? Ábrelo con `config/packages/security_checker.yaml`.

config/packages/security_checker.yaml

```yaml
services:
    _defaults:
        autowire: true
        autoconfigure: true

    SensioLabs\Security\SecurityChecker: null

    SensioLabs\Security\Command\SecurityCheckerCommand: null
```

Cada paquete que instales *puede* tener una receta de Flex. La idea es *maravillosamente* simple. En lugar de decirle a la gente que instale un paquete y *después* crear este archivo, y actualizar este otro para hacer que las cosas funcionen, Flex ejecuta una *receta* la cual... lo hace por ti! Este archivo ha sido agregado a la receta `sensiolabs/security-checker`!

No necesitas preocuparte por las especificaciones de que está *dentro* de este archivo por el momento. El punto es, *gracias* a este archivo, tenemos un nuevo comando `bin/console`.

Ejecuta:

```
php bin/console
```

Ves ese comando `security:check`? No estaba hace un segundo. Está ahí *ahora* gracias al nuevo archivo YAML. Intenta:

```
php bin/console security:check
```

Ningún paquete tiene vulnerabilidades conocidas! Genial!

Como funcionan las recetas

Aquí está el panorama en *general*: gracias al sistema de receta, siempre que instales un paquete, Flex realizará una comprobación si el paquete tiene una receta y, si lo tiene, lo instalará. Una receta puede hacer muchas cosas, como agregar archivos, crear directorios, o incluso *modificar* archivos nuevos, como agregar líneas a tu archivo `.gitignore`

El sistema de recetas *cambia las reglas del juego*. Me *encanta*, ya que cada vez que necesito una nueva librería, todo lo que tengo que hacer es instalarla. No necesito agregar archivos de configuración o modificar algo, pues la receta automatiza todo ese trabajo aburrido.

Las Recetas pueden Modificar Archivos

De hecho, esta receta hizo algo *más* que no nos dimos cuenta. En la terminal, ejecuta:

```
git diff composer.json
```

Esperábamos que Composer agregara esta nueva línea a la sección `require`. Pero *también* hay una nueva línea bajo la sección de `scripts`. Lo cual fue hecho por la receta.

composer.json

```
{
// ... lines 2 - 3
    "require": {
// ... lines 5 - 8
        "sensiolabs/security-checker": "^6.0",
// ... lines 10 - 14
    },
// ... lines 16 - 45
    "scripts": {
        "auto-scripts": {
// ... lines 48 - 49
            "security-checker security:check": "script"
        },
// ... lines 52 - 57
    },
// ... lines 59 - 67
}
```

Gracias a esto, cada vez ejecutes:

```
composer install
```

Después de terminar, automáticamente el comando security checker.

El punto es: para usar el comando security checker, lo *único* que teníamos que hacer era... instalarlo. Su receta se hizo cargo del resto de la configuración.

Ahora... si te estás preguntando:

> *"Oye! Dónde rayos vive esta receta? Puedo verla?"*

Esa es una *gran* pregunta! Vamos a averiguar donde viven las recetas y como se ven a continuación.

Chapter 7: Cómo Funcionan las Recetas

¿Dónde viven estas recetas Flex? Viven... en la *nube*. Específicamente, si miras en https://flex.symfony.com, puedes clickear para ver la receta de cualquier paquete. Esto va a... interesante: un repositorio GitHub llamado `symfony/recipes`.

Ve a la página principal de ese repositorio. Este es *el* repositorio central para recetas, organizado por el nombre de los paquetes... y luego cada paquete puede tener diferentes recetas para cada versión. Nuestra receta vive en `sensiolabs/security-checker/4.0`.

El Código de la Receta

Cada receta tiene *al menos* este archivo `manifest.json`, el cual describe todas las "cosas" que tiene que hacer. Este `copy-from-recipe` dice que el contenido del directorio `config/` en la receta debería ser copiado a nuestro proyecto. *Esta* es la razón por la cual un archivo `config/packages/security_checker.yaml` fue copiado a nuestra aplicación.

De vuelta en el manifesto, la sección `composer-scripts` le dice a Flex que agregue esta linea a nuestro archivo `composer.json`... y los aliases definen... bueno... los aliases que deberían *corresponderse* con este paquete.

Hay algunas cosas *más* que una receta puede hacer, pero esta es la idea básica.

Así que... *todas* las recetas de Symfony viven en *este* repositorio. Mmm, en realidad, esto no es así: Todas las recetas de symfony viven en este repositorio *o* en otro llamado `recipes-contrib`. No hay diferencia entre estos, excepto que el control de calidad es más alto para las recetas del repositorio *principal*.

Usando Composer Para Ver Recetas

Otra forma de ver los detalles de las recetas es a través del mismo Composer. Corre:

```
composer recipes
```

Estas son las 7 recetas que fueron instaladas en nuestra aplicación. Y si corremos:

```
composer recipes sensiolabs/security-checker
```

Podemos ver más detalles, como la URL de la receta y los archivos que copió a nuestra aplicación.

El sistema de recetas siempre será nuestro *mejor* amigo: permitiendo que nuestra app empiece pequeña, pero que crezca *automáticamente* cuando instalamos nuevos paquetes.

Removiendo un Paquete & Receta

Oh, y si decides que debes *remover* un paquete, su receta será *desinstalada*. Echa un vistazo:

```
composer remove sec-checker
```

Eso - claro está - removerá el paquete... pero *también* "desconfiguró" la receta. Cuando corremos:

```
git status
```

Está limpio! Revirtió el cambio en `composer.json` y removió el archivo de configuración.

composer.json

```
{
// ... lines 2 - 3
    "require": {
        "php": "^7.2.5",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "sensio/framework-extra-bundle": "^5.5",
        "symfony/console": "5.0.*",
        "symfony/dotenv": "5.0.*",
        "symfony/flex": "^1.3.1",
        "symfony/framework-bundle": "5.0.*",
        "symfony/yaml": "5.0.*"
    },
// ... lines 15 - 44
    "scripts": {
        "auto-scripts": {
            "cache:clear": "symfony-cmd",
            "assets:install %PUBLIC_DIR%": "symfony-cmd"
        },
// ... lines 50 - 55
    },
// ... lines 57 - 65
}
```

A continuación: Instalemos Twig - el sistema de templates de Symfony - para poder crear templates HTML. La receta de Twig va a hacer que esto sea *muy* fácil.

Chapter 8: La Receta de Twig

Salvo que estés creando una API pura - y hablaremos de retornar JSON más tarde en este tutorial - necesitarás escribir algo de HTML. Y... poner texto o HTML en un controlador así es... horrible.

No te preocupes! Symfony tiene una *excelente* integración con una *increíble* librería de templates llamada Twig. Hay solo un problema: nuestra app de Symfony es *tan* pequeña que Twig ni siquiera está instalado! Ah, pero eso no es *realmente* un problema... gracias al sistema de recetas.

Instalar Twig

Vuelve a https://flex.symfony.com y haz una búsqueda por "template". Ahí está! Aparentemente la librería de templates recomendada por Symfony es also llamado `twig-pack`. ¡Instalémosla!

```
composer require template
```

Esto instala algunos paquetes... Y sí! 2 recetas! Veamos lo que hicieron:

```
git status
```

Chequeando los Cambios de la Receta

Wow, impresionante. Muy bien: los cambios en `composer.json` `composer.lock` y `symfony.lock` eran de esperarse. *Todo lo demás* fue hecho por estas recetas.

¿Qué son los Bundles?

Veamos `bundles.php` primero:

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

Interesante: agregó dos lineas. Abre ese archivo: `config/bundles.php`.

config/bundles.php

```
// ... lines 1 - 2
return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
    Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
];
```

Un "bundle" es un *plugin* de Symfony. Comúnmente, cuando quieres agregar una funcionalidad a tu app, instalas un bundle. Y cuando instalas un bundle, necesitas *habilitarlo* en tu aplicación. Hace mucho tiempo atrás, lo hacíamos manualmente. Pero gracias a Symfony Flex, siempre que instalas un bundle de Symfony, automáticamente actualiza este archivo para habilitarla por tí. Así que... ahora que hemos hablado de este archivo, probablemente jamás necesitarás pensar en esto de nuevo.

Los directorios templates/ y config/

La receta también agregó un directorio `templates/`. Así que si te preguntabas donde se supone que viven tus templates... la receta contestó esa pregunta! También agregó un archivo de layout `base.html.twig` del cual hablaremos pronto.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

Entones... aparentemente nuestros templates se supone que viven en `templates/`. ¿Pero por qué? Quiero decir, esa ruta está fijada en algún archivo interno de la librería de Twig? No! Vive justo en nuestro código, gracias al archivo `twig.yaml` que fue creado por la receta. Revisémoslo: `config/packages/twig.yaml`.

config/packages/twig.yaml

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

Hablaremos más sobre estos archivos YAML en otro tutorial. Pero sin comprender demasiado sobre este archivo, él mismo... ya tiene sentido! Esta configuración `default_path` apunta al directorio `templates/`. ¿Quieres que tus templates vivan en algún otro lugar? Solo cambia esto y... Listo! Tú tienes el control.

Por cierto, no te preocupes por esta rara sintaxis `%kernel.project_dir%`. Aprenderemos sobre esto más adelante. Pero básicamente, es una forma sofisticada de apuntar al directorio raíz de nuestro proyecto.

La receta también creó otro archivo `twig.yaml` el cual es menos importante: `config/packages/test/twig.yaml`. El mismo hace un *pequeño* cambio a Twig para tus tests automatizados. Los detalles no importan realmente. El punto es: Hemos instalado Twig y su receta se encargó de todo lo demás. Estamos 100% listos para usarlo en nuestra app. ¡Hagamos esto a continuación!

Chapter 9: Twig ❤️

Hagamos que la acción `show()` del controlador renderé código HTML usando un template. Tan *pronto* como quieras representar un template, necesitarás que tu controlador herede del `AbstractController`. No olvides permitir que PhpStorm lo autocomplete para que pueda agregar el `import` necesario.

Ahora, obviamente, un controlador no *necesita* heredar esta clase base - A Symfony no le interesa eso. *Pero*, es *usual* heredar del `AbstractController` por una simple razón: nos brinda métodos útiles!

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
↕ // ... lines 6 - 8
9 class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 27
28 }
```

Rendereando un Template

El primer método útil es `render`. Podemos decir: `return this->render()` y pasar dos argumentos. El primero es el nombre del archivo del template: podemos poner lo que sea aquí, pero usualmente - porque valoramos nuestra cordura - lo nombramos igual que el controlador: `question/show.html.twig`.

El segundo argumento es un array de todas las variables que queremos pasar al template. Eventualmente, vamos a hacer una query específica a la base de datos y pasaremos los datos al template. Por el momento, hay que fingirlo. Voy a copiar la linea `ucwords()` y borrar el código viejo. Pasemos una variable al template llamada - que tal: `question` - asignada a este string.

src/Controller/QuestionController.php

```
// ... lines 1 - 8
class QuestionController extends AbstractController
{
// ... lines 11 - 21
    public function show($slug)
    {
        return $this->render('question/show.html.twig', [
            'question' => ucwords(str_replace('-', ' ', $slug))
        ]);
    }
}
```

Es hora de una pregunta! Qué valor crees que regresa el método `render()`? Un string? alguna otra cosa? La respuesta es: un objeto `Response`... conteniendo HTML. Porque recuerda: la *única* regla de un controlador es que *siempre* debe de regresar un objeto tipo `Response`.

Tip

Un controlador *puede* regresar algo *distinto* a un objeto Response, pero no te preocupes por eso ahorita... o tal vez nunca.

Creando el Template

Entonces, creemos ese template! Dentro de `templates/`, crea el subdirectorio `question`, luego un nuevo archivo llamado `show.html.twig`. Empecemos sencillo: un `<h1>` y luego `{{ question }}` para representar la *variable* question. Y... voy a poner un poco más de sintaxis.

templates/question/show.html.twig

```
<h1>{{ question }}</h1>

<div>
    Eventually, we'll print the full question here!
</div>

```

Las 3 Sintaxis de Twig!

Acabamos de escribir nuestro primer código de Twig! Twig es *muy* amigable: es un simple archivo HTML hasta que escribes una de sus *dos* sintaxis.

La primera es la sintaxis "imprime algo". `{{`, lo que quieres imprimir, luego `}}`. Dentro de las llaves, estás escribiendo código en Twig... el cual es muy similar a JavaScript. Esto imprime la variable `question`. Si pones comillas alrededor, imprimirá la *palabra* `question`. Y claro, puedes hacer cosas mas complejas - como el operador terneario. Es decir, es *muy* similar a JavaScrip.

La *segunda* sintaxis es la que yo llamo "haz algo". Va de esta forma `{%` seguido por lo que quieres hacer, por ejemplo un `if` o un `for`. Hablaremos más de esto en un momento.

Y... eso es todo! O estás imprimiendo algo con `{{` o *haciendo* algo, como un `if`, con `{%`.

Ok, una *pequeña* mentira, *existe* una tercera sintaxis... pero es solo para comentarios: `{#`, el comentario... luego `#}`.

templates/question/show.html.twig

```
<h1>{{ question }}</h1>

{# oh, I'm just a comment hiding here #}

<div>
    Eventually, we'll print the full question here!
</div>

```

Veamos si funciona! Abre la página, refresca y... Lo tenemos! Si miras el código fuente, puedes notar que *no* hay una estructura HTML aun. Es literalmente la estructura de nuestro template y nada mas. Le vamos a agregar una estructura base en algunos minutos.

Haciendo Bucles con el Tag {%

Ok: tenemos una pregunta falsa. Creo que se merece algunas respuestas falsas! De regreso al controlador, en la acción `show()`, voy a pegar 3 respuestas falsas.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
24         $answers = [
25             'Make sure your cat is sitting purrrfectly still ?',
26             'Honestly, I like furry shoes better than MY cat',
27             'Maybe... try saying the spell backwards?',
28         ];
↕ // ... lines 29 - 33
34     }
35 }
```

Como he dicho, una vez que hayamos hablado sobre base de datos, vamos a hacer un query en lugar de esto. Pero para comenzar, esto va a funcionar de maravilla. Pasalas al template como la *segunda* variable llamada `answers`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
24         $answers = [
25             'Make sure your cat is sitting purrrfectly still ?',
26             'Honestly, I like furry shoes better than MY cat',
27             'Maybe... try saying the spell backwards?',
28         ];
29
30         return $this->render('question/show.html.twig', [
31             'question' => ucwords(str_replace('-', ' ', $slug)),
32             'answers' => $answers,
33         ]);
34     }
35 }
```

De regreso al template. Como las podríamos imprimir? No podemos solo decir `{{ answers }}`... porque es un array. Lo que *realmente* queremos hacer es *recorrer* el array e imprimir cada respuesta individual. Para poder hacer esto, *tenemos* que hacer uso de nuestra primer función "haz algo"! Se vería algo así: `{% for answer in answers %}`. y la mayoría de las etiquetas "haz algo" también tienen una etiqueta de cierre: `{% endfor %}`.

Ponle una etiqueta `ul` alrededor y, dentro del ciclo, di `<li>` y `{{ answer }}`.

templates/question/show.html.twig

```
// ... lines 1 - 8
<h2>Answers</h2>

<ul>
    {% for answer in answers %}
        <li>{{ answer }}</li>
    {% endfor %}
</ul>
// ... lines 16 - 17
```

Me fascina! Ok navegador, refresca! Funciona! Digo, está muy, *muy* feo... pero lo vamos a arreglar pronto.

La Referencia de Twig: Tags, Filtros, Funciones

Dirígete a https://twig.symfony.com. Twig es su propia librería con su *propia* documentación. Aquí hay un montón de cosas útiles... Pero lo que *realmente* me gusta está aquí abajo: la Referencia de Twig.

Ves esas "Etiquetas" a la izquierda? Esas son *todas* las etiquetas "Haz algo" que existen. Asi es, *siempre* será `{%` y luego *una* de estas palabras - por ejemplo, `for`, `if` o `{% set`. Si intentas `{% pizza`, yo voy a pensar que es gracioso, pero Twig te va a gritar.

Twig también tiene funciones... como cualquier lenguaje... y una agradable funcionalidad llamada "tests", la cual es algo única. Esto te permite decir cosas como: `if foo is defined` o `if number is even`.

Pero la *mayor* y *mejor* sección es la de los "filtros". Los filtros son básicamente funciones... pero más hipster. Mira el filtro `length`. Los filtros funcionan como las "cadenas" en la linea de comandos: solo que aquí unimos la variable `users` en el filtro `length`, el cual solo los cuenta. El valor va de izquierda a derecha. Los filtros son en realidad funciones... con una sintaxis más amigable.

Usemos este filtro para imprimir el *número* de respuestas. Voy a poner algunos paréntesis, luego `{{ answer|length }}` Cuando lo probamos... de lujo!

templates/question/show.html.twig

```
↕ // ... lines 1 - 7
8
9  <h2>Answers {{ answers|length }}</h2>
10
↕ // ... lines 11 - 17
```

Herencia con Templates de Twig: extends

En este punto, ya eres *apto* para convertirte en un profesional de Twig. Solo hay *una* funcionalidad importante más de la cual hablar. y es una buena: herencia de templates.

La mayoría de nuestras páginas van a compartir una estructura HTML. Actualmente, no contamos con *ninguna* estructura HTML. Para hacer una, *arriba* del template, escribe `{% extends 'base.html.twig' %}`.

templates/question/show.html.twig

```
1  {% extends 'base.html.twig' %}
2
3  <h1>{{ question }}</h1>
↕ // ... lines 4 - 19
```

Esto le dice a Twig que queremos usar el template `base.html.twig` como la estructura base. En este momento, este archivo es *muy* básico, pero es *nuestro* para modificarlo - y lo haremos pronto.

Pero si refrescas la página... huye! Un gran error!

> *"Un template que hereda de otro no puede incluir contenido fuera de los bloques de Twig."*

Cuando *heredas* de un template, estás diciendo que quieres que el contenido de este template vaya *dentro* de `base.html.twig`. Pero... donde? debería Twig ponerlo arriba? Abajo? En algún lugar del medio? Twig no lo sabe!

Estoy seguro que ya habías notado estos bloques, como `stylesheets`, `title` y `body`. Los bloques son "hoyos" donde un template hijo puede *agregar* contenido. No podemos *simplemente* heredar de `base.html.twig`: necesitamos decirle en cuál *bloque* debe ir el contenido. El bloque `body` es el lugar perfecto.

Como hacemos esto? Arriba del contenido agrega `{% block body %}`, y después, `{% endblock %}`.

templates/question/show.html.twig

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

{% block body %}
// ... lines 4 - 18
{% endblock %}
// ... lines 20 - 21
```

Ahora intentalo. Funciona! No pareciera que es mucho... porque la estructura base es tan simple, pero si revisas el código fuente de la página, *tenemos* la estructura HTML básica.

Agregar, Remover, Cambiar Bloques?

Por cierto, estos bloques en `base.html.twig` no son especiales: los puedes renombrar, moverlos de lugar, agregar o remover. Entre más bloques agregues, más flexible es tu template "hijo" para agregar contenido en lugares diferentes.

La mayoría de los bloques existentes están vacíos... pero el bloque *puede* definir contenido por *defecto*. Como el bloque `title`. Ves ese `Welcome`? No es sorpresa, ese es el título actual de la página.

Como se encuentra dentro de un bloque, podemos *sobreescribirlo* en cualquier template. Mira esto: en donde sea dentro de `show.html.twig`, escribe `{% block title %}`, Question, imprime la pregunta, luego `{% endblock %}`.

templates/question/show.html.twig

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

{% block title %}Question: {{ question }}{% endblock %}

{% block body %}
// ... lines 6 - 20
{% endblock %}
// ... lines 22 - 23
```

Esta vez cuando recargamos... tenemos un *nuevo* título!

Ok, con Twig en nuestras espaldas, vamos a ver una de las funcionalidades más *útiles* de Symfony... y tu nuevo mejor amigo para depurar: Symfony profiler.

Chapter 10: Profiler: El mejor amigo de tu Debugger

Estamos teniendo un *muy* buen progreso - Deberías estar orgulloso! Veamos qué archivos hemos modificado:

```
git status
```

Agrega Todo:

```
git add .
```

Y haz commit:

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

Instalando el Profiler

Porque *ahora* quiero instalar una de mis herramientas de symfony favoritas. Corre:

```
composer require profiler --dev
```

Estoy usando `--dev` porque el profiler es una herramienta que sólo necesitaremos mientras estamos en *desarrollo*: No será usada en producción. Esto significa que Composer lo agrega a la sección `require-dev` de `composer.json`. No es tan importante, pero es la forma correcta de hacerlo.

Tip

En proyectos nuevos, en vez de `symfony/profiler-pack`, podrías ver 3 paquetes aquí, incluyendo `symfony/web-profiler-bundle`. ¡No hay problema! Explicaremos esto en algunos minutos.

composer.json

```
{
// ... lines 2 - 15
    "require-dev": {
        "symfony/profiler-pack": "^1.0"
    },
// ... lines 19 - 67
}
```

Y... en este punto, *no* debería sorprendernos que esto ha instalado una receta!

Corre:

```
git status
```

Saluda a la Barra de Herramientas Web Debug

¡Oh, wow! Agregó *tres* archivos de configuración. Gracias a éstos, el módulo funcionará *al instante*. Pruébalo: de vuelta a tu navegador, refresca la página. ¡Saluda a la barra de herramientas debug! La dichosa barrita en la parte inferior. Ahora esto aparecerá en *cada* página HTML mientras estamos desarrollando. Nos muestra el código de status, cuál controlador y ruta usamos, velocidad, memoria, llamadas Twig e incluso más íconos aparecerán a medida que empezamos a usar más partes de symfony.

Y Saluda al Profiler

La *mejor* parte es que puedes hacer click en cualquier de estos íconos para saltar... al *profiler*. Esta es básicamente la version expandida de la barra de herramientas y está llena de información sobre la carga de la página, incluyendo la información del request, response e incluso una maravillosa pestaña de performance. Esta *no solo* es una buena manera de hacer

un debug del performance de tu aplicación, *también* es una gran manera... de simplemente entender qué está sucediendo dentro de Symfony.

Hay otras secciones para Twig, configuración, cacheo y eventualmente habrá una pestaña para ver las queries a la base de datos. A propósito, esto no es solo para páginas HTML: *también* puedes acceder al profiler para las llamadas AJAX que haces a tu app. Te mostraré cómo más adelante.

Las Funciones dump() y dd()

Cuando instalamos el profiler, también obtuvimos otra herramienta útil llamada `dump()`. Haré click en atrás un par de veces para ir a la página. Abre el controlador: `src/Controller/QuestionController.php`.

Imagina que queremos ver una variable. Normalmente, usaría `var_dump()`. En vez de ello, usa `dump()` y vamos a imprimir el `$slug` y... qué tal el propio objeto `$this`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
↕ // ... lines 24 - 29
30         dump($slug, $this);
↕ // ... lines 31 - 35
36     }
37 }
```

Cuando refrescamos, órale! Funciona *exactamente* como `var_dump()` excepto... *muchísimo* más bello y útil. El controlador aparentemente tiene una propiedad llamada `container`... y podemos ir más y más profundo.

Y si eres *muy* haragán... Cómo la mayoría de nosotros lo es... también puedes usar `dd()` lo cual significa `dump()` y luego `die()`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
↕ // ... lines 24 - 29
30         dd($slug, $this);
↕ // ... lines 31 - 35
36     }
37 }
```

Ahora cuando refrescamos... hace el dump, pero *también* termina la ejecución en la página. Hemos perfeccionado el desarrollo basado en dump-and-die. ¿Creo que deberíamos estar orgullosos?

Instalando el Paquete de Debug

Cámbialo de vuelta a `dump()`... y *sólo* hagamos `dump($this)`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 21
22     public function show($slug)
23     {
↕ // ... lines 24 - 29
30         dump($this);
↕ // ... lines 31 - 35
36     }
37 }
```

Hay *otra* librería que podemos instalar para herramientas de debug. Esta es menos importante - pero de todas formas buena para tener en cuenta. En tu terminal, corre:

```
composer require debug
```

Esta vez *no* estoy usando `-- dev` porque esto instalará algo que sí quiero en producción. Esto instala el DebugBundle - eso no es algo que necesitemos en producción - pero *también* instala Monolog, que es una librería de logueo. Y probablemente sí querramos loguear cosas en producción.

Paquetes Composer?

Antes de hablar de lo que esto nos dió, hecha un vistazo al nombre del paquete que instaló: `debug-pack`. Esta no es la primera vez que hemos instalado una librería con "pack" en su nombre.

Un "pack" es un concepto especial en Symfony: Es como un tipo de paquete "falso" cuya única función es ayudar a instalar varios paquetes al mismo tiempo. Echale un vistazo: copia el nombre del paquete, busca tu navegador, y ve a https://github.com/symfony/debug-pack. Orale! No es nada más que un archivo `composer.json`! Esto nos da una manera fácil de instalar *solo* este paquete... pero en realidad obtener *todas* estas librerías.

> **Tip**
>
> En mi proyecto, instalar un "pack" solo agregaría *una* linea a `composer.json`: `symfony/debug-pack`. Pero a partir de `symfony/flex` 1.9, cuando instalas un pack, en vez de agregar `symfony/debug-pack` a `composer.json`, agregará 5 paquetes. Aún obtienes el mismo código, pero esto facilita el manejo de las versiones de los paquetes.

Así que gracias a esto, tenemos dos nuevas cosas en nuestra aplicación. La primera es un logguer! Si refrescamos la página... y hacemos click en el profiler, tenemos la sección "Logs" que nos muestra *todos* los logs para este request. Estos *también* son guardados en el archivo `var/log/dev.log`.

La segunda cosa nueva en nuestra aplicación es... bueno... si miraste atentamente, el `dump()` desapareció de la página! El DebugBundle integra la función `dump()` incluso *más* dentro de Symfony. Ahora si usamos *dump()*, en vez de imprimirlo en la mitad de la página, lo pone aquí abajo en la barra de herramientas debug. Puedes hacer click en ella para ver una versión más grande. No es tan importante... Es solo otro ejemplo de cómo Symfony se vuelve más listo a medida que instalas más cosas.

El Comando server:dump

Oh, ya que estamos hablando de ello, el DebugBundle nos dió un nuevo comando de la consola. En tu terminal, corre:

```
php bin/console server:dump
```

Esto inicia un pequeño servidor detrás de escena. *Ahora*, siempre que se ejecute `dump()` en nuestro código, aún se muestra en nuestra barra de herramientas... Pero *también* se imprime en la terminal! Esa es una excelente manera de ver información pedida en los requests de AJAX. Presionaré Control-C para detenerlo.

Expandiendo Packs

Oh, y hablando de estos "packs", si abres el archivo `composer.json`, el único problema con los packs es que aquí sólo vemos `debug-pack` version 1.0: no podemos controlar las versiones de los paquetes de dentro. Simplemente obtienes cualquiera que sea la versión que el pack solicita.

composer.json

```
{
// ... lines 2 - 3
    "require": {
// ... lines 5 - 9
        "symfony/debug-pack": "^1.0",
// ... lines 11 - 15
    },
// ... lines 17 - 68
}
```

Si necesitas mas control, no hay problema... Sólo extrae el pack:

```
composer unpack symfony/debug-pack
```

Eso hace exactamente lo que esperas: quita `debug-pack` de `composer.json` y *agrega* los paquetes subyacentes, como `debug-bundle` y `monolog`. Ah, y como el `profiler-pack` es

una dependencia del `debug-pack`, está en ambos lugares. Removeré el extra del `require`.

composer.json

```
{
// ... lines 2 - 3
    "require": {
        "php": "^7.2.5",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "easycorp/easy-log-handler": "^1.0.7",
        "sensio/framework-extra-bundle": "^5.5",
        "symfony/console": "5.0.*",
        "symfony/debug-bundle": "5.0.*",
        "symfony/dotenv": "5.0.*",
        "symfony/flex": "^1.3.1",
        "symfony/framework-bundle": "5.0.*",
        "symfony/monolog-bundle": "^3.0",
        "symfony/profiler-pack": "*",
        "symfony/twig-pack": "^1.0",
        "symfony/var-dumper": "5.0.*",
        "symfony/yaml": "5.0.*"
    },
// ... lines 21 - 72
}
```

A continuación, hagamos nuestro sitio más bello incluyendo CSS en nuestra aplicación.

Chapter 11: Assets: CSS, Imágenes, etc

Vamos muy bien pero, Cielos! Nuestro sitio está muy *feo*. Es hora de arreglarlo.

Si descargas el código del curso en esta página, después de que lo descomprimas, encontrarás el directorio `start/` con el directorio `tutorial/` ahí dentro: el mismo directorio `tutorial/` que ves aquí. Vamos a copiar algunos archivos de ahí en los próximos minutos.

Copiando el Layout Base y el Archivo CSS principal

El primero es `base.html.twig.` Lo voy a abrir, copiar el contenido, cerrarlo, y luego abriré nuestro `templates/base.html.twig.` Pega el nuevo contenido aquí.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}
            <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.c integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">
            <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Spartan&display=swap">
            <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.12.1/css/all.min.css" integrity="sha256-mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
            <link rel="stylesheet" href="/css/app.css">
        {% endblock %}
    </head>
    <body>
        <nav class="navbar navbar-light bg-light" style="height: 100px;">
          <a class="navbar-brand" href="#">
            <i style="color: #444; font-size: 2rem;" class="pb-1 fad fa-cauldron"></i>
            <p class="pl-2 d-inline font-weight-bold" style="color: #444;">Cauldron Overflow</p>
          </a>
          <button class="btn btn-dark">Sign up</button>
        </nav>

        {% block body %}{% endblock %}
        <footer class="mt-5 p-3 text-center">
            Made with <i style="color: red;" class="fa fa-heart"></i> by the guys and gals at <a style="color: #444; text-decoration: underline;" href="https://symfonycasts.com">SymfonyCasts</a>
        </footer>
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

Esto no fué un gran cambio: sólo agregó algunos archivos CSS - incluyendo Bootstrap - y un poco de HTML básico. Pero tenemos los mismos bloques que antes: `{% block body %}` en el medio, `{% block javascripts %}` , `{% block title %}`, etc.

Date cuenta que los link tags están *dentro* del bloque llamado `stylesheets`. Pero eso aun no es importante. Explicaré porque está hecho de esa forma dentro de poco.

templates/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
↕ // ... lines 4 - 5
        {% block stylesheets %}
            <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.c
integrity="sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
            <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Spartan&display=swap">
            <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.12.1/css/all.min.css" integrity="sha256-
mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
            <link rel="stylesheet" href="/css/app.css">
        {% endblock %}
    </head>
↕ // ... lines 13 - 27
</html>
```

Uno de los link tags está apuntando a `/css/app.css`. Ese es *otro* archivo que vive en el directorio `tutorial/`. De hecho, selecciona el directorio `images/` *y* `app.css` y cópialos. Ahora, selecciona el directorio `public/` y pégalos. Agrega otro directorio `css/` y mueve app.css adentro.

Recuerda: el directorio `public/` es nuestro documento raíz. Así que si necesitas que un archivo sea accesible por un usuario del navegador, entonces necesita vivir aquí. La ruta `/css/app.css` cargará el archivo `public/css/app.css`.

Vamos a ver como se ve! Muévete hacia tu navegador y refresca. *Mucho* mejor. El centro aun se ve terrible... pero eso es porque no hemos agregado ninguna etiqueta HTML al template para esta página.

A Symfony le Importan tus Assets

Así que hagamos una pregunta... y respondámosla: que funcionalidad nos ofrece Symfony cuando se trata de CSS y JavaScript? La respuesta es... ninguna... o muchas!

Symfony tiene dos niveles diferentes de integración con CSS y JavaScript. Por el momento estamos usando el nivel básico. De hecho, por ahora, Symfony no está haciendo *nada* por nosotros: creamos un archivo CSS, luego le agregamos un link tag muy tradicional con HTML. Symfony *no* está haciendo nada: todo depende de ti.

El *otro* tipo de integración de *mayor* nivel es utilizar algo llamado Webpack Encore: una *fantástica* librería que maneja minificación de archivos, soporte de Sass, soporte a React o VueJS y otras muchas cosas. Te voy a dar un curso rápido sobre Webpack Encore al final de este tutorial.

Pero por ahora, lo vamos a mantener simple: Crearás archivos CSS o de JavaScript, los pondrás dentro del directorio `public/`, y luego crearás un `link` o `script` tag que apunte a ellos.

La No Tán Importante Función asset()

Bueno, de hecho, incluso con esta integración "básica", hay *una* pequeña funcionalidad de Symfony que debes de utilizar.

Antes de mostrartelo, en PhpStorm abre preferencias... y busca de nuevo por " Symfony" para encontrar el plugin de Symfony. Ves esa option de directorio web? Cambiala a `public/` - solía ser llamada `web/` en versiones anteriores de Symfony. Esto nos ayudará a tener un mejor autocompletado próximamente. Presiona "Ok".

Así es como funciona: cada vez que hagas referencias a un archivo estático en tu sitio - como un archivo CSS, JavaScript o imagen, en vez de solo escribir `/css/app.css`, debes de usar la función de Twig llamada `asset()`. Entonces, `{{ asset() }}` y luego la misma ruta que antes, pero sin la `/` inicial: `css/app.css`.

templates/base.html.twig

```
// ... line 1
<html>
    <head>
// ... lines 4 - 5
        {% block stylesheets %}
// ... lines 7 - 9
            <link rel="stylesheet" href="{{ asset('css/app.css') }}">
        {% endblock %}
    </head>
// ... lines 13 - 27
</html>
```

Qué es lo que hace está increíble función `asset()`? Prácticamente.. nada. De hecho, esto va a retornar *exactamente* la misma ruta que antes: `/css/app.css`.

Entonces porque nos molestamos en utilizar una función que no hace nada? Bueno, en realidad *hace dos* cosas... las cuales pueden o no interesarte. Primero, si decides instalar tu aplicación en un *subdirectorio* de un dominio - como por ejemplo `ILikeMagic.com/cauldron_overflow`, la función `asset()` automáticamente agrega el prefijo `/cauldron_overflow` a todas las rutas. *Grandioso!* si es que te interesa.

La *segunda* cosa que hace es más útil: si decides instalar tus assets a un CDN, con agregar una linea a un archivo de configuración, repentinamente, Symfony agregará el prefijo en *todas* las rutas con la URL de tu CDN.

Asi que... en realidad no es *tán* importante, pero si utilizas `asset()` en todos lados, serás feliz en caso de que luego lo necesites.

Pero... si refrescamos... sorpresa! El sitio exploto!

> *"Acaso olvidaste correr `composer require symfony/asset`? La función `asset` no existe."*

Que tan genial es eso? Recuerda, Symfony empieza en pequeño: instalas las cosas sólo *cuando* las necesitas. En este caso, estamos tratando de utilizar una funcionalidad que no está instalada... por lo tanto Symfony nos da el comando *exacto* que tenemos que correr. Copialo, abre la terminal y ejecuta:

```
composer require symfony/asset
```

Cuando termine... regresa al navegador y... funciona. Si observas la fuente HTML y buscas `app.css`... Asi es! está imprimiendo la misma ruta que antes.

Mejorando la página "show"

Hagamos que el centro de nuestra página se vea un poco mejor. De vuelta en el directorio `tutorial/`, abre `show.html.twig`, copia el contenido, ciérralo, luego abre nuestra versión: `templates/question/show.html.twig`. Pega el nuevo código.

templates/question/show.html.twig

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

{% block title %}Question: {{ question }}{% endblock %}

{% block body %}
<div class="container">
    <div class="row">
        <div class="col-12">
            <h2 class="my-4">Question</h2>
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
                <div class="q-container-show p-4">
                    <div class="row">
                        <div class="col-2 text-center">
                            <img src="/images/tisha.png" width="100" height="100">
                        </div>
                        <div class="col">
                            <h1 class="q-title-show">{{ question }}</h1>
                            <div class="q-display p-3">
                                <i class="fa fa-quote-left mr-3"></i>
                                <p class="d-inline">I've been turned into a cat, any thoughts on how to turn back? While I'm adorable, I don't really care for cat food.</p>
                                <p class="pt-4"><strong>--Tisha</strong></p>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>

    <div class="d-flex justify-content-between my-4">
        <h2 class="">Answers <span style="font-size:1.2rem;">({{ answers|length }})</span></h2>
        <button class="btn btn-sm btn-secondary">Submit an Answer</button>
    </div>



    <ul class="list-unstyled">
        {% for answer in answers %}
            <li class="mb-4">
                <div class="d-flex justify-content-center">
                    <div class="mr-2 pt-2">
```

```
42                          <img src="/images/tisha.png" width="50"
height="50">
43                      </div>
44                      <div class="mr-3 pt-2">
45                          {{ answer }}
46                          <p>-- Mallory</p>
47                      </div>
48                      <div class="vote-arrows flex-fill pt-2" style="min-
width: 90px;">
49                          <a class="vote-up" href="#"><i class="far fa-
arrow-alt-circle-up"></i></a>
50                          <a class="vote-down" href="#"><i class="far fa-
arrow-alt-circle-down"></i></a>
51                          <span>+ 6</span>
52                      </div>
53                  </div>
54              </li>
55          {% endfor %}
56      </ul>
57  </div>
58  {% endblock %}
```

Recuerda, aquí no está pasando nada importante: seguimos sobreescribiendo el mismo bloque `title` y `body`. Estamos usando la misma variable `question` y seguimos haciendo el mismo ciclo sobre `answers` aquí abajo. Solo tenemos mucha más sintaxis HTML... lo cual... tu sabes... hace que luzca bien.

Al refrescar... mira! Hermoso! De vuelta en el template, nota que esta página tiene algunas tags `img`... pero *no* están usando la función `asset()`. Hay que arreglarlo. Utilizaré un atajo! Simplemente escribo "tisha", oprimo tab y... boom! el resto se agrega solo! Buscar por `img`... y reemplaza también esta con "tisha". Te preguntas quien es tisha? Oh, es solo una de los multiples gatos que tenemos aquí en el staff de SymfonyCasts. Esta controla a Vladimir.

templates/question/show.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-12">
// ... line 9
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
                <div class="q-container-show p-4">
                    <div class="row">
                        <div class="col-2 text-center">
                            <img src="{{ asset('images/tisha.png') }}" width="100" height="100">
                        </div>
// ... lines 16 - 23
                    </div>
                </div>
            </div>
        </div>
    </div>
// ... lines 29 - 36
    <ul class="list-unstyled">
        {% for answer in answers %}
            <li class="mb-4">
                <div class="d-flex justify-content-center">
                    <div class="mr-2 pt-2">
                        <img src="{{ asset('images/tisha.png') }}" width="50" height="50">
                    </div>
// ... lines 44 - 52
                </div>
            </li>
        {% endfor %}
    </ul>
</div>
{% endblock %}
```

Por cierto, en una aplicación real, en vez de que estas imágenes sean archivos estáticos en el proyecto, podrían ser archivos que los usuarios *suben*. No te preocupes: tenemos todo un tutorial sobre como manejar la subida de archivos.

Asegurate de que esto funciona y... si funciona.

Puliendo la Homepage

La *última* página que no hemos estilizado es el homepage... la cual en este momento... imprime un texto. Abre el controlador: `src/Controller/QuestionController.php`. Asi es! Solamente retorna un nuevo objeto `Response()` y texto. Podemos hacerlo mejor. Cambialo por `return $this->render()`. Llamemos al template `question/homepage.html.twig.` y... por ahora... No creo que necesitemos pasar alguna variable al template... Asi que dejaré vacío el segundo argumento.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function homepage()
15     {
16         return $this->render('question/homepage.html.twig');
17     }
↕ // ... lines 18 - 34
35 }
```

Dentro de `templates/question/`, crea un nuevo archivo: `homepage.html.twig`.

La mayoría de los templates empiezan de la *misma* forma. Genial consistencia! En la parte de arriba, `{% extends 'base.html.twig' %}`, `{% block body %}` y `{% endblock %}`. En medio, agrega más HTML para ver si esto funciona.

templates/question/homepage.html.twig

```
1 {% extends 'base.html.twig' %}
2
3 {% block body %}
4     <h1>Voilà</h1>
5 {% endblock %}
```

Muy bien... refresca la página y... excelente! Excepto por la parte de que se ve horrible.

Robemos algo de código del directorio `tutorial/` *una* última vez. Abre `homepage.html.twig`. Esto es *solo* un montón de HTML estático para hacer que se vea mejor. Copialo, cierra ese archivo... y luego pegalo en nuestro código `homepage.html.twig`

templates/question/homepage.html.twig

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

{% block body %}
<div class="jumbotron-img jumbotron jumbotron-fluid">
    <div class="container">
        <h1 class="display-4">Your Questions Answered</h1>
        <p class="lead">And even answers for those questions you didn't think to ask!</p>
    </div>
</div>
<div class="container">
    <div class="row mb-3">
        <div class="col">
            <button class="btn btn-question">Ask a Question</button>
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
                <div class="q-container p-4">
                    <div class="row">
                        <div class="col-2 text-center">
                            <img src="{{ asset('images/tisha.png') }}" width="100" height="100">
                            <div class="d-block mt-3 vote-arrows">
                                <a class="vote-up" href="#"><i class="far fa-arrow-alt-circle-up"></i></a>
                                <a class="vote-down" href="#"><i class="far fa-arrow-alt-circle-down"></i></a>
                            </div>
                        </div>
                        <div class="col">
                            <a class="q-title" href="#"><h2>Reversing a Spell</h2></a>
                            <div class="q-display p-3">
                                <i class="fa fa-quote-left mr-3"></i>
                                <p class="d-inline">I've been turned into a cat, any thoughts on how to turn back? While I'm adorable, I don't really care for cat food.</p>
                                <p class="pt-4"><strong>--Tisha</strong></p>
                            </div>
                        </div>
                    </div>
                </div>
                <a class="answer-link" href="#" style="color: #fff;">
                    <p class="q-display-response text-center p-3">
```

```
                        <i class="fa fa-magic magic-wand"></i> 6 answers
                    </p>
                </a>
            </div>
        </div>

        <div class="col-12 mt-3">
            <div class="q-container p-4">
                <div class="row">
                    <div class="col-2 text-center">
                        <img src="{{ asset('images/magic-photo.png') }}" width="100" height="100">
                        <div class="d-block mt-3 vote-arrows">
                            <a class="vote-up" href="#"><i class="far fa-arrow-alt-circle-up"></i></a>
                            <a class="vote-down" href="#"><i class="far fa-arrow-alt-circle-down"></i></a>
                        </div>
                    </div>
                    <div class="col">
                        <a class="q-title" href="#"><h2>Pausing a Spell</h2></a>
                        <div class="q-display p-3">
                            <i class="fa fa-quote-left mr-3"></i>
                            <p class="d-inline">I mastered the floating card, but now how do I get it back to the ground?</p>
                            <p class="pt-4"><strong>--Jerry</strong></p>
                        </div>
                    </div>
                </div>
            </div>
            <a class="answer-link" href="#" style="color: #fff;">
                <p class="q-display-response text-center p-3">
                    <i class="fa fa-magic magic-wand"></i> 15 answers
                </p>
            </a>
        </div>
    </div>
</div>
{% endblock %}

```

Y ahora... se ve *mucho* mejor.

Así que esta es la integración *básica* de CSS y Javascript dentro de Symfony: tu te encargas de manejarlo. Claro, *debes* de utilizar la función `asset()`, pero no hace nada muy impresionante.

Si quieres más, estás de suerte! En el *último* capítulo, llevaremos nuestros assets al siguiente nivel. Te va a fascinar.

A continuación: nuestro sitio tiene algunos links! Y todos te llevan a ninguna parte! Aprendamos como generar URLs con rutas.

Chapter 12: Generando URLs

Vuelve a la página "show" para una cuestión. El logo de arriba es un link... que no va a ninguna parte aún. Este *debería* llevarnos a la página de inicio.

Como forma parte del layout, el link vive en `base.html.twig`. Aquí está: `navbar-brand` con `href="#"`.

templates/base.html.twig

```
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13     <body>
14         <nav class="navbar navbar-light bg-light" style="height: 100px;">
15           <a class="navbar-brand" href="#">
↕ // ... lines 16 - 17
18           </a>
↕ // ... line 19
20         </nav>
↕ // ... lines 21 - 26
27     </body>
28 </html>
```

Para hacer que esto nos lleve a la página de inicio, podemos simplemente cambiarlo a `/`, ¿Cierto? *Podrías* hacerlo, pero en Symfony, una mejor forma es pedirle a Symfony que *genere* una URL hacia esta ruta. De esta forma, si decidimos cambiar esta URL en el futuro, todos nuestros links se actualizarán automáticamente.

¡Cada Ruta Tiene un Nombre!

Para ver cómo hacer esto, ve a tu terminal y corre:

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

Esto muestra un listado de *cada* ruta del sistema... ¡Y, hey! Desde la última vez que lo corrimos, hay un *montón* de rutas nuevas. Estas alimentan a la barra de herramientas debug y el profiler y son agregadas automáticamente por el WebProfilerBundle cuando estamos en modo `dev`.

De todas formas, lo que *realmente* quiero ver es la columna "Name". *Toda* ruta tiene un nombre interno, incluyendo las dos rutas que hicimos. Aparentemente sus nombres son `app_question_homepage` y `app_question_show`. Pero... eh... ¿De dónde vinieron? ¡No recuerdo haber escrito ninguno de éstos!

Entonces... A cada ruta *debe* serle dada un nombre interno. Pero cuando usas rutas en anotación... te deja hacer trampa: elige un nombre *por ti* basado en la clase y método del controlador... ¡Lo cual es asombroso!

Pero... tan pronto como necesitas generar la URL de una ruta, yo recomiendo darle un nombre *explícito*, en lugar de depender de este nombre autogenerado, el cual podría cambiar de repente si le cambias el nombre al método. Para darle un nombre a una ruta, agrega `name=""` y... Que tal: `app_homepage`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 8
9  class QuestionController extends AbstractController
10 {
11     /**
12      * @Route("/", name="app_homepage")
13      */
14     public function homepage()
15     {
↕ // ... line 16
17     }
↕ // ... lines 18 - 34
35 }
```

Me gusta mantener los nombres de mis rutas cortos, pero `app_` lo hace lo suficientemente largo como para poder realizar una búsqueda a partir de esta cadena si alguna vez lo necesito.

Ahora, si corremos `debug:router` nuevamente:

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

¡Bien! Tomamos el control del nombre de nuestra ruta. Copia el nombre `app_homepage` y luego vuelve a `base.html.twig`. El objetivo es simple, queremos decir:

> *"¡Hey symfony! ¿Puedes por favor decirme la URL para la ruta `app_homepage`?"*

Para hacer esto en Twig, usa `{{ path() }}` y pásale el nombre de la ruta.

templates/base.html.twig

```
↕ // ... line 1
2 <html>
↕ // ... lines 3 - 12
13     <body>
14         <nav class="navbar navbar-light bg-light" style="height: 100px;">
15           <a class="navbar-brand" href="{{ path('app_homepage') }}">
↕ // ... lines 16 - 17
18           </a>
↕ // ... line 19
20         </nav>
↕ // ... lines 21 - 26
27     </body>
28 </html>
```

¡Eso es todo! Cuando volvemos y refrescamos... *Ahora* esto va hacia la página principal.

Apuntando a una Ruta con {Comodines}

En la página principal, tenemos dos preguntas escritas a mano... y cada una tiene dos links que actualmente no van a ninguna parte. ¡Arreglémoslos!

Paso uno: ahora que queremos generar una URL de esta ruta, encuentra la ruta y agrega `name="app_question_show"`.

src/Controller/QuestionController.php

```
// ... lines 1 - 8
class QuestionController extends AbstractController
{
// ... lines 11 - 18
    /**
     * @Route("/questions/{slug}", name="app_question_show")
     */
    public function show($slug)
    {
// ... lines 24 - 33
    }
}
```

Copia esto y abre el template: `templates/question/homepage.html.twig`. Veamos... Justo debajo de la parte de votar, aquí está el primer link a una pregunta que dice "Reversing a spell". Quita el signo numeral, agrega `{{ path() }}` y pega `app_question_show`.

Pero... no podemos detenernos aquí. ¡Si probamos la página ahora, un error glorioso!

> *"Algunos parámetros obligatorios están faltando - "slug""*

¡Eso tiene sentido! ¡No podemos simplemente decir "genera la URL hacia `app_question_show`" porque esa ruta tiene un comodín! Symfony necesita saber qué valor debería usar para `{slug}`. ¿Cómo le decimos? Agrega un *segundo* parámetro a `path()` con `{}`. El `{}` es un array asociativo de Twig... nuevamente, tal como en JavaScript. Pásale `slug` igual a... Veamos... Esta es una pregunta escrita a mano por el momento, así que escribe `reversing-a-spell`.

templates/question/homepage.html.twig

```
↕ // ... lines 1 - 2
3 {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
17         <div class="col-12">
18             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
19                 <div class="q-container p-4">
20                     <div class="row">
↕ // ... lines 21 - 27
28                         <div class="col">
29                             <a class="q-title" href="{{ path('app_question_show', { slug: 'reversing-a-spell' }) }}"><h2>Reversing a Spell</h2></a>
↕ // ... lines 30 - 34
35                         </div>
36                     </div>
37                 </div>
38                 <a class="answer-link" href="{{ path('app_question_show', { slug: 'reversing-a-spell' }) }}" style="color: #fff;">
↕ // ... lines 39 - 41
42                 </a>
43             </div>
44         </div>
↕ // ... lines 45 - 71
72     </div>
73 </div>
74 {% endblock %}
↕ // ... lines 75 - 76
```

Cópialo *todo*, porque hay un link más aquí abajo para la misma pregunta. Para la segunda pregunta... Pégalo nuevamente, pero cámbialo a `pausing-a-spell` para igualar el nombre. Copiaré eso... Encuentra la última ocurrencia... Y pégalo.

templates/question/homepage.html.twig

```
↕ // ... lines 1 - 2
3 {% block body %}
↕ // ... lines 4 - 9
10 <div class="container">
↕ // ... lines 11 - 15
16     <div class="row">
↕ // ... lines 17 - 45
46         <div class="col-12 mt-3">
47             <div class="q-container p-4">
48                 <div class="row">
↕ // ... lines 49 - 55
56                     <div class="col">
57                         <a class="q-title" href="{{ path('app_question_show', { slug: 'pausing-a-spell' }) }}"><h2>Pausing a Spell</h2></a>
↕ // ... lines 58 - 62
63                     </div>
64                 </div>
65             </div>
66             <a class="answer-link" href="{{ path('app_question_show', { slug: 'pausing-a-spell' }) }}" style="color: #fff;">
↕ // ... lines 67 - 69
70             </a>
71         </div>
72     </div>
73 </div>
74 {% endblock %}
↕ // ... lines 75 - 76
```

Más adelante, cuando implementemos una base de datos, vamos a mejorar esto y evitaremos repetirnos tantas veces. ¡Pero! Si volvemos, refrescamos... ¡Y hacemos click en el link, funciona! Ambas páginas van hacia la misma ruta, pero con un valor diferente para el slug.

A continuación, llevemos nuestro sitio al siguiente nivel, al crear una interface API JSON que consumiremos con JavaScript.

Chapter 13: Rutas JSON en la API

Una de las funcionalidades en nuestro sitio... la cual aun no funciona... es la de votar a favor o en contra en las respuestas de las preguntas. Eventualmente, cuando hagas click arriba o abajo, esto hará un request tipo AJAX a una ruta de una API que vamos a crear. Esa ruta va a guardar el voto en la base de datos y va a *responder* con un JSON que contendrá el *nuevo* total de votos y así nuestro JavaScript podrá actualizar el contador.

Aún no tenemos una base de datos en nuestra aplicación, pero estamos listos para construir *todas* las otras partes de esta funcionalidad.

Creando una Ruta JSON

Comencemos por crear una ruta tipo JSON para la API a la cual accederemos con AJAX cuando el usuario vote en una respuesta arriba o abajo.

Podríamos crear esto en `QuestionController` como un nuevo método. Pero como esta ruta en *realidad* trabaja con un "comment", vamos a crear un *nuevo* controlador. Llámalo `CommentController`.

Como la vez pasada, vamos a escribir `extends AbstractController` y presionar tab para que PhpStorm autocomplete esto y agregue el `import` en la parte de arriba. Al extender de esta clase nos brinda métodos de atajo... y me encantan los atajos!

src/Controller/CommentController.php

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

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class CommentController extends AbstractController
{
}
```

Dentro, crea una *función pública*. Puede tener cualquier nombre... que tal `commentVote()`. Agrega la ruta arriba: `/**`, luego `@Route`. Autocompleta la del componente Routing para que

asi PhpStorm agregue el `import`.

Para la URL, que tal `/comments/{id}` - esto eventualmente será el id del comentario específico en la base de datos - `/vote/{direction}`, donde `{direction}` será cualquiera de las palabras `arriba` o `abajo`.

y como tenemos estos dos comodines, podemos agregar dos argumentos: `$id` y `$direction`. Empezaré con un comentario: el `$id` será *súper* importante después cuando tengamos una base de datos... pero no lo vamos a usar por ahora.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 6
7  use Symfony\Component\Routing\Annotation\Route;
↕ // ... line 8
9  class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction}")
13      */
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 25
26     }
27 }
```

Sin una base de datos, vamos a simular la lógica. Si `$direction == 'up'`, entonces normalmente guardaríamos el voto a favor en la base de datos y consultaríamos el nuevo total de votos. En vez de eso, escribe `$currentVoteCount = rand(7, 100)`.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 8
9  class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
16         // todo - use id to query the database
17
18         // use real logic here to save this to the database
19         if ($direction === 'up') {
20             $currentVoteCount = rand(7, 100);
21         } else {
↕ // ... line 22
23         }
↕ // ... lines 24 - 25
26     }
27 }
```

El conteo de votos está escrito directamente en el template con un total de 6. Así que esto hará que el nuevo conteo de votos parezca ser un *nuevo* número mayor que este. En el else, haz lo opuesto: un número aleatorio entre 0 y 5.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 8
9  class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
16         // todo - use id to query the database
17
18         // use real logic here to save this to the database
19         if ($direction === 'up') {
20             $currentVoteCount = rand(7, 100);
21         } else {
22             $currentVoteCount = rand(0, 5);
23         }
↕ // ... lines 24 - 25
26     }
27 }
```

Si, esto será *mucho* más interesante cuando tengamos una base de datos, pero va a funcionar muy bien para nuestro propósito.

Regresando un JSON?

La pregunta ahora es: después de "Guardar" el voto en la base de datos, que debería de retornar el controlador? Bueno, probablemente debería retornar un JSON... y *sé* que quiero incluir el nuevo conteo en la respuesta para que nuestro JavaScript pueda utilizarlo y actualizar el número de votos.

Así que... como regresamos un JSON? Recuerda: nuestro *único* trabajo en un controlador es retornar un objeto de tipo Symfony `Response`. JSON no es nada más que una respuesta cuyo contenido es una cadena JSON en vez de HTML. Así que *podemos* poner: `return new Response()` con `json_encode()` con algún dato.

Pero! en vez de eso, `return new JsonResponse()` - autocompleta esto para que PhPStorm agregue el `import`. Pasa un array con los datos que queremos. Que tal pasar la llave `votes` con `$currentVoteCount`.

src/Controller/CommentController.php

```
// ... lines 1 - 5
use Symfony\Component\HttpFoundation\JsonResponse;
// ... lines 7 - 8
class CommentController extends AbstractController
{
// ... lines 11 - 13
    public function commentVote($id, $direction)
    {
        // todo - use id to query the database

        // use real logic here to save this to the database
        if ($direction === 'up') {
            $currentVoteCount = rand(7, 100);
        } else {
            $currentVoteCount = rand(0, 5);
        }

        return new JsonResponse(['votes' => $currentVoteCount]);
    }
}
```

Ahora... *tal vez* estés pensando:

> *"Ryan! Te la pasas diciendo que debemos retornar un objeto Response... y acabas de retornar algo diferente. Esto es una locura!"*

Es un punto *válido*. Pero! si presionas Command or Ctrl y das click en la clase `JsonResponse`, vas a aprender que `JsonResponse hereda de Response`. Esta clase no es nada más que un atajo para crear respuestas tipo JSON: esto hace el JSON encode a los datos que le pasamos *y* se asegura que la cabecera `Content-Type` sea asignada a `application/json`, la cual ayuda a las librerías AJAX a entender que estamos regresando un JSON.

Así que... ah! Probemos nuestra nueva y brillante ruta de la API! Copia la URL, abre un nueva nueva pestaña en el navegador, pega y llena los comodines: que tal 10 para el `{id}` y para el voto "up". Presiona enter. Hola ruta JSON!

El punto clave más importante aquí es: las respuestas JSON no son nada especiales.

El Método atajo json()

La clase `JsonResponse` nos hace la vida más sencilla.. pero podemos ser aún más *flojos*! En vez de `new JsonResponse` simplemente escribe return `$this->json()`.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 8
9  class CommentController extends AbstractController
10 {
↕ // ... lines 11 - 13
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 24
25         return $this->json(['votes' => $currentVoteCount]);
26     }
27 }
```

Esto no cambia nada: es solo un atajo para crear el *mismo* objeto `JsonResponse`. Pan comido.

El Serializador de Symfony

Por cierto, uno de los "componentes" de Symfony se llama "Serializer", y es *muy* bueno convirtiendo *objetos* a JSON o XML. Aún no lo hemos instalado, pero *si* lo hiciéramos, el `$this->json()` empezaría a utilizarlo para serializar cualquier cosa que le pasemos. No haría ninguna diferencia en nuestro caso donde pasamos un array, pero significa que podrías

empezar a pasar *objetos* a `$this->json()`. Si quieres saber más - o quieres construir una muy sofisticada API - ve nuestro tutorial sobre API Platform: un increíble bundle de Symfony para construir APIs.

A continuación, escribamos algo de JavaScript que hará un llamado AJAX a nuestra nueva ruta. También vamos a aprender como agregar JavaScript global *y* JavaScript específico a una página

Chapter 14: JavaScript, AJAX y el Profiler

Este es nuestro próximo objetivo: escribir algo de JavaScript para que cuando hagamos click en los íconos de arriba o abajo, se realice un request AJAX a nuestra ruta JSON. Este "simula" guardar el voto en la base de datos y retorna el nuevo recuento de votos, el cual usaremos para actualizar el número de votos en la página.

Agregando Clases js- al Template

El template de esta página es: `templates/question/show.html.twig`. Para cada respuesta, tenemos estos links de `votar-arriba` y `votar-abajo`. Voy a agregar algunas clases a esta sección para ayudar a nuestro JavaScript. En el elemento `vote-arrows`, agrega una clase `js-vote-arrows`: lo usaremos en el JavaScript para encontrar el elemento. Luego, en el link de `vote-up`, agrega un atributo data llamado `data-direction="up"`. Haz lo mismo para el link de abajo: `data-direction="down"`. Esto nos ayudará a saber en cuál link se hizo click. Finalmente, rodea el numero de votos - el 6 - con un span que contenga otra clase: `js-vote-total`. Usaremos esto para encontrar el elemento para poder actualizar ese número.

templates/question/show.html.twig

```
// ... lines 1 - 4
{% block body %}
<div class="container">
// ... lines 7 - 36
    <ul class="list-unstyled">
        {% for answer in answers %}
            <li class="mb-4">
                <div class="d-flex justify-content-center">
// ... lines 41 - 47
                    <div class="vote-arrows flex-fill pt-2 js-vote-arrows" style="min-width: 90px;">
                        <a class="vote-up" href="#" data-direction="up"><i class="far fa-arrow-alt-circle-up"></i></a>
                        <a class="vote-down" href="#" data-direction="down"><i class="far fa-arrow-alt-circle-down"></i></a>
                        <span>+ <span class="js-vote-total">6</span></span>
                    </div>
                </div>
            </li>
        {% endfor %}
    </ul>
</div>
{% endblock %}
```

Agregando JavaScript Dentro del Bloque javascripts.

Para simplificar las cosas, el código JavaScript que escribiremos usará jQuery. De hecho, si tu sitio usa jQuery, *probablemente* querrás incluir jQuery en *cada* página... Lo cual significa que queremos agregar una etiqueta `script` a `base.html.twig`. En la parte de abajo, fíjate que tenemos un bloque llamado `javascripts`. Dentro de este bloque, voy a pegar una etiqueta `<script>` para descargar jQuery desde un CDN. Puedes copiar esto desde el bloque de código en esta página, o ir a jQuery para obtenerlo.

Tip

En los nuevos proyectos de Symfony, el bloque `javascripts` se encuentra en la parte superior de este archivo - dentro de la etiqueta `<head>`. Puedes dejar el bloque `javascripts` en `<head>` o moverlo aquí abajo. Si lo dejas dentro de `head`, asegúgate de agregar un atributo `defer` a cada etiqueta `script`: Esto hará que tu JavaScript sea ejecutado *luego* de que la página termine de cargar.

templates/base.html.twig

```
// ... line 1
<html>
// ... lines 3 - 12
    <body>
// ... lines 14 - 25
        {% block javascripts %}
            <script
              src="https://code.jquery.com/jquery-3.4.1.min.js"
              integrity="sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
              crossorigin="anonymous"></script>
        {% endblock %}
    </body>
</html>
```

Si te preguntas *por qué* pusimos esto dentro del bloque `javascripts`... Más allá de que "parece" un lugar lógico, te mostraré por qué en un minuto. Ya que, *técnicamente*, si pusiéramos esto *luego* del bloque `javascripts` o antes, no habría ninguna diferencia por el momento. Pero ponerlos dentro va a ser útil pronto.

Para nuestro propio JavaScript, dentro del directorio `public/`, crea un nuevo directorio llamado `js/`. Y luego, un archivo: `question_show.js`.

Esta es la idea: usualmente tendrás algún código JavaScript que querrás incluir en cada página. No tenemos ninguno por el momento, pero si lo *tuviéramos*, yo crearía un archivo `app.js` y agregaría una etiqueta `script` para ello en `base.html.twig`. Luego, en ciertas páginas, podrías necesitar incluir algún JavaScript específico para la página, como por ejemplo, para hacer funcionar el voto de comentarios que solo vive en una página.

Esto es lo que estoy haciendo y esta es la razón por la que creé un archivo llamado `question_show.js`: Es JavaScript específico para esa página.

Dentro de `question_show.js`, voy a pegar al rededor de 15 líneas de código.

public/js/question_show.js

```
/**
 * Simple (ugly) code to handle the comment vote up/down
 */
var $container = $('.js-vote-arrows');
$container.find('a').on('click', function(e) {
    e.preventDefault();
    var $link = $(e.currentTarget);

    $.ajax({
        url: '/comments/10/vote/'+$link.data('direction'),
        method: 'POST'
    }).then(function(response) {
        $container.find('.js-vote-total').text(response.votes);
    });
});
```

Esto encuentra el elemento `.js-vote-arrows` - el cual agregamos aquí - encuentra cualquier etiqueta dentro del mismo, y registra una función para el evento `click` allí. Al hacer click, hacemos una llamada AJAX a `/comments/10` - el 10 es escrito a mano por ahora - `/vote/` y luego leemos el atributo `data-direction` del elemento `<a>` para saber si este es un voto `arriba` o `abajo`. Al finalizar exitosamente, jQuery nos pasa los datos JSON de nuestra llamada. Renombremos esa variable a `data` para ser más exactos.

public/js/question_show.js

```
// ... lines 1 - 4
$container.find('a').on('click', function(e) {
// ... lines 6 - 8
    $.ajax({
// ... lines 10 - 11
    }).then(function(data) {
        $container.find('.js-vote-total').text(data.votes);
    });
});
```

Luego usamos el campo `votes` de los datos - porque en nuestro controlador, estamos retornando una variable `votes` - para actualizar el total de votos.

Sobreescribiendo el Bloque javascripts.

Entonces... ¿Cómo incluimos este archivo? Si quisiéramos incluir esto en *cada* página, sería bastante fácil: agrega otra etiqueta script abajo de jQuery en `base.html.twig`. Pero

queremos incluir esto *solo* en la página show. Aquí es donde tener el script de jQuery dentro del bloque `javascripts` es útil. Porque, en un template "hijo", podemos *sobreescribir* ese bloque.

Echemos un vistazo: en `show.html.twig`, no importa donde - pero vayamos al final, di `{% block javascripts %} {% endblock %}`. Dentro, agrega una etiqueta `<script>` con `src=""`. Ah, tenemos que recordar usar la función `asset()`. Pero... PhpStorm nos sugiere `js/question_show.js`. Selecciona ese. ¡Muy bien! Agregó la función `asset()` por nosotros.

templates/question/show.html.twig

```
↕ // ... lines 1 - 59
60 {% block javascripts %}
↕ // ... lines 61 - 62
63     <script src="{{ asset('js/question_show.js') }}"></script>
64 {% endblock %}
```

Si paráramos ahora, esto literalmente *sobreescribiría* el bloque `javascripts` de `base.html.twig`. Por lo que jQuery no sería incluido en la página. ¡En vez de *sobreescribir* el bloque, lo que *realmente* queremos es *agregar* algo a él! En el HTML final, queremos que nuestra nueva etiqueta `script` vaya justo *debajo* de jQuery.

¿Cómo podemos hacer esto? Sobre nuestra etiqueta script, di `{{ parent() }}`.

templates/question/show.html.twig

```
↕ // ... lines 1 - 59
60 {% block javascripts %}
61     {{ parent() }}
62
63     <script src="{{ asset('js/question_show.js') }}"></script>
64 {% endblock %}
```

¡Me encanta! La función `parent()` toma el contenido del bloque *padre*, y lo imprime.

¡Probémoslo! Refresca y... Haz click en up. ¡Se actualiza! Y si hacemos click en down, vemos un número muy bajo.

Requests AJAX en el Profiler

Ah, y ¿Ves este número "6" aquí abajo en la barra de herramientas debug? Esto es genial. Refresca la página. Fíjate que el ícono *no* está aquí abajo. ¡Pero, tan pronto como nuestra página hace una llamada AJAX, aparece! Sip, la barra de herramientas debug *detecta* llamadas AJAX y las enlista aquí. ¡La mejor parte es que puedes usar esto para saltar al *profiler* para cualquiera de estos requests! Voy a hacer click con el botón derecho y abriré este link de voto "abajo" en una nueva pestaña.

Este es el profiler completo para la llamada en *todo* su esplendor. Si usas `dump()` en alguna parte de tu código, la variable volcada para esa llamada AJAX estará aquí. Y luego, tendremos una sección de base de datos aquí. Esta es una funcionalidad *maravillosa*.

A continuación, ajustemos nuestra ruta de la API: No deberíamos poder hacer un request GET al mismo - como si lo abriéramos en nuestro navegador. Y... ¿Tenemos algo que valide que el comodín `{direction}`... de la URL sea `up` o `down` pero nada más? Todavía no.

Chapter 15: Rutas Inteligentes: Solo POST y Validación de {Comodín}

Dentro de nuestro JavaScript, estamos haciendo una petición POST a la API. Y tiene sentido. El tema de "cual método HTTP" - como GET, POST, PUT, etc - se *supone* debes usar para un llamado a la API... puede ser complicado. Pero como nuestra ruta eventualmente va a *cambiar* algo en la base de datos, como práctica recomendable, no queremos permitir a la gente que hagan llamados tipo GET a nuestra ruta. Por ahora, podemos hacer un llamado `GET` con tan solo poner la URL en nuestro navegador. Hey! Acabo de votar!

Para mejorar esto, en el `CommentController`, podemos hacer más inteligente a nuestra ruta, podemos hacer que *solo* funcione cuando el método sea POST. Para lograrlo agrega `methods="POST"`.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 8
9  class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction}", methods="POST")
13      */
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 25
26     }
27 }
```

Tan *pronto* lo hagamos, al refrescar... error 404! La ruta ya no se encuentra

Tip

De hecho, es un error 405! Método HTTP no permitido.

El Comando router:match

Otra buena forma de ver esto es en tu terminal. Corre: `php bin/console router:match`. Luego copia la URL... y pegala.

```
php bin/console router:match /comments/10/vote/up
```

Este divertido comando nos dice cuál *ruta* le pertenece a una URL. En este caso, *ninguna* ruta fue encontrada pero esto nos dice que *casi* encuentra la ruta `app_comment_commentvote`.

Para ver si un llamado `POST` sería encontrado, pasa `--method=POST`:

```
php bin/console router:match /comments/10/vote/up --method=POST
```

Y... Bum! Nos muestra la ruta que pudo encontrar y todos los detalles, incluyendo el controlador.

Restringiendo un {Comodín}

Pero hay algo más que no está del todo bien con nuestra ruta. La ruta *espera* que la parte `{direction}` sea `arriba` o `abajo`. Pero... técnicamente, alguien podría poner `plátano` en la URL. De hecho, probémoslo: Cambia la dirección por `plátano`:

```
php bin/console router:match /comments/10/vote/banana --method=POST
```

Si! Votamos "plátano" para este comentario! No es el fin del mundo... si un usuario intenta hackear nuestro sistema y hace esto, solo significaría un voto negativo. Pero podemos hacerlo mejor.

Como has de saber, *normalmente* un comodín se empareja con *cualquier* cosa. Sin embargo, si quisieras, puedes controlarlo con una expresión regular. Dentro de `{}`, pero después del nombre, agrega `<>`. Dentro, escribe `up|down`.

src/Controller/CommentController.php

```
// ... lines 1 - 8
class CommentController extends AbstractController
{
    /**
     * @Route("/comments/{id}/vote/{direction<up|down>}", methods="POST")
     */
    public function commentVote($id, $direction)
    {
// ... lines 16 - 25
    }
}
```

Ahora prueba el comando router:match

```
php bin/console router:match /comments/10/vote/banana --method=POST
```

Si! *No* encuentra la ruta porque `plátano` no es arriba o abajo. Si cambiamos esto por `arriba`, funciona:

```
php bin/console router:match /comments/10/vote/up --method=POST
```

Como Hacer que el id Solo Funcione con Enteros?

Por cierto, podrías ser tentado a hacer más inteligente el comodín `{id}`. Asumiendo que usamos ids con auto incremento en la base de datos, sabemos que el `id` debe de ser un entero. Para hacer que esta ruta solo funcione si la parte del `id` es un número, puedes agregar `<\d+>`, lo que significa: encuentra un "dígito" con cualquier tamaño.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 8
9  class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id<\d+>}/vote/{direction<up|down>}",
   methods="POST")
13      */
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 25
26     }
27 }
```

Pero... En realidad *no* voy a poner esto aquí. Por qué? Eventualmente vamos a usar `$id` para llamar a la base de datos. Si alguien escribe `plátano` aquí, a quien le importa? El query no va a encontrar ningún comentario con `plátano` como id y vamos a agregar algo de código para retornar una página 404. Incluso si alguien intenta hacer un ataque de inyección de SQL, como aprenderás más tarde en nuestro tutorial de base de datos, no *habría* problema, porque la capa de la base de datos nos protege de ello.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 8
9  class CommentController extends AbstractController
10 {
11     /**
12      * @Route("/comments/{id}/vote/{direction<up|down>}", methods="POST")
13      */
14     public function commentVote($id, $direction)
15     {
↕ // ... lines 16 - 25
26     }
27 }
```

Hay que asegurarnos que todo aún funciona. Voy a cerrar una pestaña del navegador y refrescar la página. Eso! los votos aún se ven bien.

A continuación, demos un vistazo a la parte más *fundamental* de Symfony: Los servicios.

Chapter 16: Objetos Servicio

En realidad, Symfony tiene dos partes... y acabamos de aprender *una* de ellas.

La primera parte es el sistema ruta-controlador. Y espero que te sientas muy cómodo: crea una ruta, la ruta ejecuta una función del controlador, regresamos una respuesta.

La *segunda* mitad de Symfony es todo sobre los múltiples "objetos útiles" que flotan alrededor de Symfony. Por ejemplo, cuando hacemos un render de un template, lo que en *realidad* hacemos es aprovecharnos del objeto twig y decirle que haga el render. El método `render()` es solo un atajo para utilizar ese objeto. También existe un objeto logger, el objeto del caché y muchos otros, como el objeto de la conexión con la base de datos y un objeto que ayuda a hacer llamados HTTP a otras APIs.

Básicamente... *cada cosa* que Symfony realiza - o *nosotros* - *realmente* es hecha por uno de estos objetos útiles. Demonios, incluso el *rúter* es un objeto que busca cuál ruta se empareja con el request actual.

En el mundo de Symfony - bueno, en realidad, en el mundo de programación orientada a objetos - estos "objetos que hacen algún trabajo" se les otorga un nombre especial: servicios. Pero no permitas que te confunda: cuando escuches "servicio", solo piensa:

> *“¡Hey! Es un objeto que hace algún trabajo - como el objeto logger o el objeto que hace consultas a la base de datos.”*

Listando Todos los Servicios

Dentro del `CommentController`, vamos a registrar un log. Para hacerlo, necesitamos el servicio "logger". ¿Cómo lo podemos obtener?

Encuentra tu terminal y corre:

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

Saluda a uno de los comandos *más* importantes de `bin/console`. Esto nos muestra una lista de *todos* los objetos servicio en nuestra app. Bueno, está bien, estos no son *todos*: pero *es* una lista que contiene todos los servicios que *probablemente* necesites.

Incluso en nuestra pequeña app, hay muchas cosas aquí: hay algo llamado `Psr\Log\LoggerInterface`, hay cosas para el caché y mucho más. Conforme instalamos más bundles, esta lista va a crecer. Más servicios significa más herramientas.

Para encontrar qué servicio nos permite crear "logs", corre:

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

Esto retorna un *montón* de cosas... pero ignora todos los de aquí abajo por ahora y enfocate en la línea de arriba. Esto nos dice que hay un objeto servicio logger y su clase implementa una `Psr\Log\LoggerInterface`. ¿Por qué es esto importante? Porque para *pedir* el servicio logger, lo haces utilizando este type-hint. Se le llama "autowiring".

Usando Autowiring para el Servicio del Logger

Así es como obtienes un servicio desde un controlador. Agrega un tercer argumento a tu método - aunque el orden de los argumentos no importa. Escribe `LoggerInterface` - autocompleta el del `Psr\Log\LoggerInterface` - y `$logger`.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 4
5  use Psr\Log\LoggerInterface;
↕ // ... lines 6 - 9
10 class CommentController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function commentVote($id, $direction, LoggerInterface $logger)
16     {
↕ // ... lines 17 - 28
29     }
30 }
```

Esto agregó el import arriba de la clase para `Psr\Log\LoggerInterface`, el cual es él *mismo* type-hint que el `debug:autowiring` nos dijo que usáramos. Gracias a este type-hint,

cuando Symfony hace un render de nuestro controlador, sabrá que queremos que nos pase el servicio del logger a este argumento.

Entonces... si: ahora existen *dos* tipos de argumentos que puedes agregar a tus métodos del controlador. Primero, puedes tener un *argumento* que se empareja con un comodín de tu ruta. Y segundo, puedes tener un argumento cuyo *type-hint* sea el mismo a una de las clases o interfaces listadas en `debug:autowiring`. `CacheInterface` es otro type-hint que podemos usar para tener el servicio de *caché*.

Utilizando el Servicio del Logger

Así que... ¡Vamos a usar este objeto! ¿Qué métodos nos permite llamar? ¡No tengo idea! Pero como escribimos el type-hint apropiado, podemos decir `$logger->` y PhpStorm nos dice *exactamente* cuales métodos tiene. Utilicemos `$logger->info()` para decir "Voting up!". Cópialo y di "Voting down!" en el else.

src/Controller/CommentController.php

```
↕ // ... lines 1 - 9
10 class CommentController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function commentVote($id, $direction, LoggerInterface $logger)
16     {
↕ // ... lines 17 - 19
20         if ($direction === 'up') {
21             $logger->info('Voting up!');
↕ // ... line 22
23         } else {
24             $logger->info('Voting down!');
↕ // ... line 25
26         }
↕ // ... lines 27 - 28
29     }
30 }
```

¡Es hora de probarlo! Refresca la página y... Hagamos click en arriba, abajo, arriba. Esto... por lo menos no parece que esté *roto*.

Mueve el mouse sobre la parte del AJAX de la herramienta web debug y abre el profiler para uno de estos llamados. El profiler tiene una sección de "Logs", la cual ofrece una forma *fácil* de

ver los logs para un solo Request. ¡Ahí está! "Voting up!". También puedes encontrar esto en el archivo `var/log/dev.log`.

El punto es: Symfony tiene *muchos*, muchos objetos útiles, digo "servicios". Y poco a poco, vamos a empezar a utilizar más de ellos... Cada vez agregando un *type-hint* para decirle a Symfony cual servicio queremos.

Autowiring & Utilizando el Servicio de Twig

Veamos otro ejemplo. El *primer* servicio que usamos en nuestro código es el servicio de *Twig*. Lo usamos... de forma "indirecta" al llamar `$this->render()`. En realidad, ese método es un atajo para utilizar el *servicio* Twig detrás de escenas. Y eso *no* debería de sorprenderte. Como dije antes, *todo* lo que se realiza en Symfony es hecho en *realidad* por un servicio.

Como reto, vamos a suponer que la función `render()` no existe. Gasp! En el controlador del `homepage()` comentariza la línea `render()`.

Entonces... ¿Cómo podemos utilizar el servicio de Twig directamente para hacer un render de un template? ¡No lo sé! *Definitivamente* podemos encontrar algo de documentación al respecto... pero vamos a ver si podemos descubrirlo por nosotros mismos con la ayuda del comando `debug:autowiring`

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

Y, ¡Voilà! Aparentemente existe una clase llamada `Twig\Environment` que podemos usar como "type-hint" para obtener el servicio de Twig. En nuestro controlador, escribe `Environment` y presiona tab para agregar el `import` arriba. Voy a nombrar al argumento `$twigEnvironment`.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 7
8  use Twig\Environment;
↕ // ... line 9
10 class QuestionController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(Environment $twigEnvironment)
16     {
↕ // ... lines 17 - 21
22         //return $this->render('question/homepage.html.twig');
23     }
↕ // ... lines 24 - 40
41 }
```

Dentro, escribe `$html = $twigEnvironment->`. De nuevo, sin leer *nada* de documentación, gracias al hecho de que estamos escribiendo código responsablemente y usamos type-hints, PhpStorm nos muestra *todos* los métodos de esta clase. ¡Mira! ¡Este método `render()` parece que es el que necesitamos! Pasa el mismo nombre del template de antes.

src/Controller/QuestionController.php

```
↕ // ... lines 1 - 9
10 class QuestionController extends AbstractController
11 {
↕ // ... lines 12 - 14
15     public function homepage(Environment $twigEnvironment)
16     {
17         // fun example of using the Twig service directly!
18         $html = $twigEnvironment->render('question/homepage.html.twig');
↕ // ... lines 19 - 21
22         //return $this->render('question/homepage.html.twig');
23     }
↕ // ... lines 24 - 40
41 }
```

Cuando usas twig directamente, en vez de retornar un objeto tipo Response, retorna un string con el HTML. No hay problema: termina con `return new Response()` - la de `HttpFoundation` - y pasa `$html`.

src/Controller/QuestionController.php

```
// ... lines 1 - 5
use Symfony\Component\HttpFoundation\Response;
// ... lines 7 - 9
class QuestionController extends AbstractController
{
// ... lines 12 - 14
    public function homepage(Environment $twigEnvironment)
    {
        // fun example of using the Twig service directly!
        $html = $twigEnvironment->render('question/homepage.html.twig');

        return new Response($html);

        //return $this->render('question/homepage.html.twig');
    }
// ... lines 24 - 40
}
```

Esto ahora está haciendo *exactamente* lo mismo que `$this->render()`. Para probarlo, haz click en la página de inicio. Todavía funciona.

Ahora en realidad, más allá de ser un "gran ejercicio" para entender los servicios, no hay razón para tomar el camino más *largo*. solo quiero que entiendas que los servicios *realmente* son las "cosas" que hacen el trabajo detrás de escenas. Y si quisieras hacer algo - como un log o un render de un template - lo que *realmente* necesitas es encontrar que *servicios* hacen ese trabajo. Confía en mi, *esta* es la clave para liberar todo tu potencial de Symfony.

Pongamos de vuelta el código anterior más corto, y comentariza el otro ejemplo.

src/Controller/QuestionController.php

```
// ... lines 1 - 9
class QuestionController extends AbstractController
{
// ... lines 12 - 14
    public function homepage(Environment $twigEnvironment)
    {
        /*
        // fun example of using the Twig service directly!
        $html = $twigEnvironment->render('question/homepage.html.twig');

        return new Response($html);
        */

        return $this->render('question/homepage.html.twig');
    }
// ... lines 26 - 42
}
```

Muy bien, ya *casi* has terminado el primer tutorial de symfony. ¡Eres el mejor! Como premio, vamos a terminar con algo divertido: Una introducción al sistema llamado Webpack Encore que te va a permitir hacer cosas *alocadas* con tu CSS y JavaScript.

Chapter 17: Hola Webpack Encore

Nuestra configuración de CSS y JavaScript está correcta: tenemos el directorio `public/` con los archivos `app.css` y `question_show.js`. Dentro de nuestros templates - por ejemplo `base.html.twig` - incluimos los archivos con la etiqueta tradicional link o script. Claro, utilizamos la función `{{ asset() }}`, pero esta no hace nada importante. Symfony para nada está tocando nuestros archivos del frontend.

Eso está bien. Pero si quieres ponerte serio con el desarrollo de frontend - como utilizar un framework como React o Vue - necesitas llevarlo al siguiente nivel.

Para hacerlo, vamos a utilizar una librería de Node llamada Webpack: la cual es una herramienta estándar en la industria para el manejo de los archivos del frontend. Combina y unifica tus archivos CSS y JavaScript... aunque eso es solo la punta del iceberg de lo que puede hacer.

Pero... para hacer que Webpack funcione en *realidad* bien necesitas de mucha configuración complicada. Asi que, en el mundo de Symfony, utilizamos una *grandiosa* librería llamada Webpack Encore. Es una capa ligera por *encima* de Webpack que... ¡Lo hace más fácil! Y tenemos todo un [tutorial gratuito] (https://symfonycasts.com/screencast/webpack-encore) aquí en SymfonyCasts.

Pero tengamos un curso rápido ahora mismo.

Instalando Webpack Encore

Primero, asegúrate que tienes node instalado:

```
node -v
```

Y también yarn:

```
yarn -v
```

> **Tip**
>
> Si no tienes Node o Yarn instalado - ve manuales oficiales sobre como instalarlos para *tu* SO. Para Node, ve https://nodejs.org/en/download/ y para Yarn: https://classic.yarnpkg.com/en/docs/install . Recomendamos utilizar Yarn en la versión 1.x para seguir este tutorial.

Yarn es un gestor de paquetes para Node... básicamente es un Composer para Node.

Antes de que instalemos Encore, asegúrate de guardar todos tus cambios - Yo ya lo hice. Luego corre:

```
composer require "encore:^1.8"
```

Espera... hace un minuto dije que Encore es una librería de *Node*. Entonces, por qué lo estamos instalando con Composer? Excelente pregunta! Este comando en realidad *no* instala Encore. Nop, instala un diminuto bundle llamado `webpack-encore-bundle`, el cual ayuda a *integrar* nuestra app de Symfony con Webpack Encore. Lo *mejor* de esto es que el bundle contiene una receta *muy* útil. Mira esto, corre:

```
git status
```

Wow! La receta hizo *bastante* por nosotros! Algo interesante es que modificó nuestro archivo `.gitignore`. Ábrelo en tu editor.

.gitignore

```
// ... lines 1 - 11
###> symfony/webpack-encore-bundle ###
/node_modules/
/public/build/
npm-debug.log
yarn-error.log
###
```

Bien! Ahora ignoramos `node_modules/` - el cual es la version de Node del directory `vendor/` - y algunas otras rutas.

La receta también agregó algunos archivos YAML, los cuales ayudan a configurar algunas cosas - pero en realidad no necesitas verlos.

Lo *más* importante que hizo la receta fue darnos estos 2 archivos: `package.json` - el cual es el `composer.json` de Node - y `webpack.config.js`, el cual es el archivo de configuración para Webpack Encore.

Revisa el archivo `package.json`. Esto le dice a Node qué librerías debería descargar y ya tiene las cosas básicas que necesitamos. Aún más importante: `@symfony/webpack-encore`.

package.json

```
{
    "devDependencies": {
        "@symfony/webpack-encore": "^0.28.2",
        "core-js": "^3.0.0",
        "regenerator-runtime": "^0.13.2",
        "webpack-notifier": "^1.6.0"
    },
    "license": "UNLICENSED",
    "private": true,
    "scripts": {
        "dev-server": "encore dev-server",
        "dev": "encore dev",
        "watch": "encore dev --watch",
        "build": "encore production --progress"
    }
}
```

Instalando Dependencias de Node con Yarn

Para decirle a Node que instale esas dependencias, corre:

```
yarn install
```

Este comando lee `package.json` y descarga un *montón* de archivos y directorios dentro de la nueva carpeta `node_modules/`. Puede tomar algunos minutos en descargar todo y construir un par de paquetes.

Cuando termine, vas a ver dos cosas nuevas. Primero, tienes un nuevo y flamante directorio `node_modules/` con *demasiadas* cosas en él. Y esto ya está siendo ignorado por git. Segundo, creó un archivo `yarn.lock`, el cual tiene la misma función que `composer.lock`. Asi que... debes hacer commit del archivo `yarn.lock`, pero no te preocupes por él.

Ok, Encore está instalado! A continuación, vamos a refactorizar nuestra configuración del fronted para utilizarlo.

Chapter 18: Webpack Encore: La Grandeza de Javascript

Tip

Ahora la receta agrega estos dos archivos en un lugar ligeramente diferente:

- `assets/app.js`
- `assets/styles/app.css`

Pero el propósito de cada uno es exactamente el mismo.

Muy bien: Así es como funciona todo esto. La receta agregó una nuevo directorio `assets/` con un par de archivos CSS y JS como ejemplo. El archivo `app.js` básicamente hace un `console.log()` de algo:

assets/js/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * We recommend including the built version of this JavaScript file
 * (and its CSS file) in your base layout (base.html.twig).
 */

// any CSS you import will output into a single css file (app.css in this case)
import '../css/app.css';

// Need jQuery? Install it with "yarn add jquery", then uncomment to import it.
// import $ from 'jquery';

console.log('Hello Webpack Encore! Edit me in assets/js/app.js');
```

El `app.css` cambia el color del fondo a gris ligero:

assets/css/app.css

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

Webpack Encore está completamente configurado por un solo archivo: `webpack.config.js`.

webpack.config.js

```
var Encore = require('@symfony/webpack-encore');

// Manually configure the runtime environment if not already configured yet by the "encore" command.
// It's useful when you use tools that rely on webpack.config.js file.
if (!Encore.isRuntimeEnvironmentConfigured()) {
    Encore.configureRuntimeEnvironment(process.env.NODE_ENV || 'dev');
}

Encore
    // directory where compiled assets will be stored
    .setOutputPath('public/build/')
    // public path used by the web server to access the output path
    .setPublicPath('/build')
    // only needed for CDN's or sub-directory deploy
    //.setManifestKeyPrefix('build/')

    /*
     * ENTRY CONFIG
     *
     * Add 1 entry for each "page" of your app
     * (including one that's included on every page - e.g. "app")
     *
     * Each entry will result in one JavaScript file (e.g. app.js)
     * and one CSS file (e.g. app.css) if your JavaScript imports CSS.
     */
    .addEntry('app', './assets/js/app.js')
    //.addEntry('page1', './assets/js/page1.js')
    //.addEntry('page2', './assets/js/page2.js')

    // When enabled, Webpack "splits" your files into smaller pieces for greater optimization.
    .splitEntryChunks()

    // will require an extra script tag for runtime.js
    // but, you probably want this, unless you're building a single-page app
    .enableSingleRuntimeChunk()

    /*
     * FEATURE CONFIG
     *
     * Enable & configure other features below. For a full
     * list of features, see:
     * https://symfony.com/doc/current/frontend.html#adding-more-features
     */
```

```
    .cleanupOutputBeforeBuild()
    .enableBuildNotifications()
    .enableSourceMaps(!Encore.isProduction())
    // enables hashed filenames (e.g. app.abc123.css)
    .enableVersioning(Encore.isProduction())

    // enables @babel/preset-env polyfills
    .configureBabelPresetEnv((config) => {
        config.useBuiltIns = 'usage';
        config.corejs = 3;
    })

    // enables Sass/SCSS support
    //.enableSassLoader()

    // uncomment if you use TypeScript
    //.enableTypeScriptLoader()

    // uncomment to get integrity="..." attributes on your script & link tags
    // requires WebpackEncoreBundle 1.4 or higher
    //.enableIntegrityHashes(Encore.isProduction())

    // uncomment if you're having problems with a jQuery plugin
    //.autoProvidejQuery()

    // uncomment if you use API Platform Admin (composer req api-admin)
    //.enableReactPreset()
    //.addEntry('admin', './assets/js/admin.js')
;

module.exports = Encore.getWebpackConfig();
```

No hablaremos mucho sobre este archivo - lo vamos a guardar para el tutorial sobre Encore - pero ya está configurado para *apuntar* a los archivos `app.js` y `app.css`: Encore sabe que necesita procesarlos.

Corriendo Encore

Para ejecutar Encore, ve a tu terminal y corre:

```
yarn watch
```

Este es un atajo para correr `yarn run encore dev --watch`. ¿Qué hace esto? Lee esos dos archivos en `assets/`, hace algo de procesamiento, y emite una versión *construida* de cada uno dentro del nuevo directorio `public/build/`. Aquí está el archivo `app.css` ya construido... y el archivo `app.js`. Si corriéramos Encore en modo de producción - el cual es solamente otro comando - *minificaría* el contenido de cada archivo.

Incluyendo los Archivos CSS y JS Construidos

Ocurren muchas otras cosas interesantes, pero esta es la idea básica: ponemos el código en el directorio `assets/`, pero apuntamos a los archivos *construidos* en nuestros templates.

Por ejemplo, en `base.html.twig`, en vez de apuntar al viejo archivo `app.css`, queremos apuntar al que está en el directorio `build/`. Eso es muy simple, pero WebpackEncoreBundle tiene un atajo para hacerlo incluso más fácil: `{{ encore_entry_link_tags() }}` y pasa este `app`, porque ese es el nombre del archivo fuente - se le llama "entry" en el mundo de Webpack.

templates/base.html.twig

```
↕ // ... line 1
2 <html>
3     <head>
↕ // ... lines 4 - 5
6         {% block stylesheets %}
7             <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.c
integrity="sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
8             <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Spartan&display=swap">
9             <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/5.12.1/css/all.min.css" integrity="sha256-
mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
10            {{ encore_entry_link_tags('app') }}
11        {% endblock %}
12    </head>
↕ // ... lines 13 - 33
34 </html>
```

Abajo, agrega la etiqueta script con `{{ encore_entry_script_tags('app') }}`.

templates/base.html.twig

```
↕ // ... line 1
2  <html>
↕ // ... lines 3 - 12
13     <body>
↕ // ... lines 14 - 25
26         {% block javascripts %}
27             <script
28               src="https://code.jquery.com/jquery-3.4.1.min.js"
29               integrity="sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="
30               crossorigin="anonymous"></script>
31             {{ encore_entry_script_tags('app') }}
32         {% endblock %}
33     </body>
34 </html>
```

¡Vamos a probarlo! Ve al navegador y refresca. ¿Funcionó? ¡Lo hizo! El color de fondo es gris... y si abro la consola, ahí está el log:

> *"Hello Webpack Encore!"*

Si miras la fuente HTML, ahí no está ocurriendo nada especial: tenemos una simple etiqueta link apuntando a `/build/app.css`.

Moviendo nuestro Código a Encore

Ahora que esto está funcionando, vamos a mover *nuestro* CSS hacia el nuevo sistema. Abre `public/css/app.css`, copia todo esto, luego haz click derecho y borrar el archivo. Ahora abre el *nuevo* `app.css` dentro de `assets/` y pega.

assets/css/app.css

```css
body {
    font-family: spartan;
    color: #444;
}

.jumbotron-img {
    background: rgb(237,116,88);
    background: linear-gradient(302deg, rgba(237,116,88,1) 16%, rgba(51,61,81,1) 97%);
    color: #fff;
}

.q-container {
    border-top-right-radius: .25rem;
    border-top-left-radius: .25rem;
    background-color: #efefee;
}

.q-container-show {
    border-top-right-radius: .25rem;
    border-top-left-radius: .25rem;
    background-color: #ED7458 ;
}

.q-container img, .q-container-show img {
    border: 2px solid #fff;
    border-radius: 50%;
}

.q-display {
    background: #fff;
    border-radius: .25rem;
}
.q-title-show {
    text-transform: uppercase;
    font-size: 1.3rem;
    color: #fff;
}
.q-title {
    text-transform: uppercase;
    color: #444;
}

.q-title:hover {
    color: #2B2B2B;
}
```

```
46
47  .q-title h2 {
48      font-size: 1.3rem;
49  }
50
51  .q-display-response {
52      background: #333D51;
53      color: #fff;
54  }
55
56  .answer-link:hover .magic-wand {
57    transform: rotate(20deg);
58  }
59
60  .vote-arrows {
61      font-size: 1.5rem;
62  }
63
64  .vote-arrows span {
65      font-size: 1rem;
66  }
67
68  .vote-arrows a {
69      color: #444;
70  }
71
72  .vote-up:hover {
73      color: #3D9970;
74  }
75  .vote-down:hover {
76      color: #FF4136;
77  }
78
79  .btn-question {
80    color: #FFFFFF;
81    background-color: #ED7458;
82    border-color: #D45B3F;
83  }
84
85  .btn-question:hover,
86  .btn-question:focus,
87  .btn-question:active,
88  .btn-question.active,
89  .open .dropdown-toggle.btn-question {
90    color: #FFFFFF;
91    background-color: #D45B3F;
92    border-color: #D45B3F;
```

```
}

.btn-question:active,
.btn-question.active,
.open .dropdown-toggle.btn-question {
  background-image: none;
}

.btn-question.disabled,
.btn-question[disabled],
fieldset[disabled] .btn-question,
.btn-question.disabled:hover,
.btn-question[disabled]:hover,
fieldset[disabled] .btn-question:hover,
.btn-question.disabled:focus,
.btn-question[disabled]:focus,
fieldset[disabled] .btn-question:focus,
.btn-question.disabled:active,
.btn-question[disabled]:active,
fieldset[disabled] .btn-question:active,
.btn-question.disabled.active,
.btn-question[disabled].active,
fieldset[disabled] .btn-question.active {
  background-color: #ED7458;
  border-color: #D45B3F;
}

.btn-question .badge {
  color: #ED7458;
  background-color: #FFFFFF;
}

footer {
    background-color: #efefee;
}
```

Tan pronto como hago eso, cuando refresco... ¡Funciona! ¡Nuestro CSS está de vuelta! La razón es que - si revisas tu terminal - `yarn watch` está *observando* a nuestros archivos por cambios. Tan pronto modificamos el archivo `app.css`, esto vuelve a leer el archivo y arroja una nueva versión dentro del directorio `public/build`. Esa es la razón por la cual corremos esto en segundo plano.

Hagamos lo mismo para nuestro JavaScript particular. Abre `question_show.js` y, en vez de tener un archivo JavaScript específico por página - donde solo incluimos esto en nuestra

página "show" - para mantener las cosas simples, voy a poner esto dentro del nuevo `app.js`, el cual es cargado en *cada* página.

assets/js/app.js

```
// ... lines 1 - 13
/**
 * Simple (ugly) code to handle the comment vote up/down
 */
var $container = $('.js-vote-arrows');
$container.find('a').on('click', function(e) {
    e.preventDefault();
    var $link = $(e.currentTarget);

    $.ajax({
        url: '/comments/10/vote/'+$link.data('direction'),
        method: 'POST'
    }).then(function(data) {
        $container.find('.js-vote-total').text(data.votes);
    });
});
```

Luego ve a borrar el directorio `public/js/` completamente... y `public/css/`. También abre `templates/question/show.html.twig` y, al final, remueve la vieja etiqueta script.

templates/question/show.html.twig

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

{% block title %}Question: {{ question }}{% endblock %}

{% block body %}
// ... lines 6 - 57
{% endblock %}
```

Con algo de suerte, Encore ya *reconstruyó* mi `app.js`. Asi que si damos click para ver una pregunta - Voy a refrescar solo para estar seguros - y... da click en los íconos para votar. ¡Si! Todavía funciona.

Instalando e Importando Librerías Externas (jQuery)

Ya que estamos usando Encore, existen algunas cosas *muy* interesantes que podemos hacer. Esta es una: en vez de enlazar a una CDN o descargar jQuery directamente en nuestro proyecto y agregarlo al commit, podemos *importar* jQuery e instalarlo en nuestro directorio

`node_modules/`... lo cual es *exactamente* como lo haríamos en PHP: Instalamos una librería pública dentro de `vendor/` en vez de descargarla manualmente.

Para hacer eso, abre una nueva terminal y corre:

```
yarn add jquery --dev
```

Esto es lo equivalente a correr el comando `composer require`: Agrega jquery al archivo `package.json` y lo descarga dentro de `node_modules/`. La parte `--dev` no es importante.

Después, dentro de `base.html.twig`, remueve por completo jQuery del layout.

templates/base.html.twig

```
// ... line 1
<html>
// ... lines 3 - 12
    <body>
// ... lines 14 - 25
        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}
    </body>
</html>
```

Si regresas a tu navegador y refrescas la página ahora... Está completamente roto:

> *"$ is not defined"*

...viniendo de `app.js`. Eso tiene sentido: *Solamente* descargamos jQuery en nuestro directorio `node_modules/` - aquí puedes encontrar un directorio llamado `jquery` - pero aún no lo estamos *usando*.

¿Cómo lo utilizamos? Dentro de `app.js`, descomentariza la línea del `import: import $ from 'jquery'`.

assets/js/app.js

```
// ... lines 1 - 9

// Need jQuery? Install it with "yarn add jquery", then uncomment to
import it.
import $ from 'jquery';

// ... lines 14 - 29
```

Esto "carga" el paquete `jquery` que instalamos y lo *asigna* a la variable `$`. Todas esas variables $ de más abajo están haciendo referencia al valor que importamos.

Esta es la parte *realmente* interesante: sin hacer *ningún* otro cambio, cuando refrescamos, ¡Funciona! Webpack se *dio* cuenta que estamos importando `jquery` y automáticamente lo empaquetó *dentro* del archivo `app.js` final. Importamos las cosas que necesitamos, y Webpack se encarga de... empaquetar todo.

Tip

De hecho, Webpack los separa en multiples archivos por cuestión de eficiencia. En realidad, jQuery vive dentro de un archivo diferente en public/build/ ¡Pero eso no es importante!

Importando el CSS de Bootstrap

Podemos hacer lo mismo para el CSS de Boostrap. En `base.html.twig`, arriba, elimina la etiqueta que *enlaza* a Bootstrap.

templates/base.html.twig

```
↕ // ... line 1
2 <html>
3     <head>
↕ // ... lines 4 - 5
6         {% block stylesheets %}
7             <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Spartan&display=swap">
8             <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.12.1/css/all.min.css" integrity="sha256-mmgLkCYLUQbXn0B1SRqzHar6dCnv9oZFPEC1g1cwlkk=" crossorigin="anonymous" />
9             {{ encore_entry_link_tags('app') }}
10        {% endblock %}
11    </head>
↕ // ... lines 12 - 28
29 </html>
```

Nada nuevo, cuando refrescamos, nuestro sitio se ve terrible.

Para arreglarlo, encuentra tu terminal y corre:

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

Esto descarga el paquete de `bootstrap` dentro de `node_modules/`. Este paquete contiene ambos JavaScript y CSS. Queremos activar el CSS.

Para hacerlo, abre `app.css` y, en la parte de arriba, utiliza la vieja y confiable sintaxis `@import`. Dentro de las comillas, escribe `~bootstrap`:

assets/css/app.css

```
1 @import "~bootstrap";
2
↕ // ... lines 3 - 129
```

En CSS, la `~` es una forma especial de decir que quieres cargar el CSS del paquete de `bootstrap` dentro de `node_modules/`.

Ve al navegador, refresca y... estamos de vuelta! Webpack vio el import, tomó el CSS del paquete de bootstrap, y lo incluyó en el archivo `app.css` final. ¿Qué tan bueno es eso?

¿Qué Otras Cosas Puede Hacer Encore?

Esto es solo el comienzo de lo que Webpack Encore puede hacer. También puede minificar tus archivos para producción, puede compilar código Sass o LESS, viene con soporte para React y Vue.js, maneja versiones para los archivos y más. Para aprender más, mira nuestro tutorial gratuito sobre Webpack Encore.

Y... ¡Eso es todo para este tutorial! ¡Felicitaciones por llegar al final junto conmigo! Ahora ya entiendes las partes más importantes de Symfony. En el siguiente tutorial, vamos a hacer crecer incluso aún *más* tu potencial de Symfony al revelar el secreto de los servicios. Serás imparable.

Como siempre, si tienes preguntas, problemas o tienes una historia divertida - especialmente si involucra a tu gato - nos *encantaría* escuchar sobre ti en los comentarios.

Muy bien amigos - ¡Nos vemos la próxima vez!

With <3 from SymfonyCasts