Actualizar a Symfony 8

Chapter 1: PHP 8.4 y actualizaciones de recetas

¡Hola amigos! Bienvenidos a un nuevo curso sobre cómo actualizar una aplicación Symfony de Symfony 7 a Symfony 8. Por suerte, las nuevas versiones de Symfony hacen que este proceso de actualización sea pan comido. Pero todavía hay algunas cosas que hacer para que el proceso de actualización sea más suave.

Para seguir adelante, descarga el código del curso desde la parte superior de esta página, abre el directorio `start` en tu IDE, y verás lo que tengo aquí. Sigue el README para configurarlo todo, cuando estés listo, en tu terminal, ejecuta:

```
symfony server:start -d
```

(la bandera `-d` ejecuta el servidor en segundo plano). Haz clic en el enlace de la salida para abrir la aplicación en tu navegador.

¡Bienvenido a la Starshop! Si has seguido nuestros cursos anteriores sobre Symfony 7, esto debería resultarte familiar.

Abajo, en la barra de herramientas de depuración web, puedes ver que estamos en Symfony 7.3.5 y PHP 8.3.30.

¿Nuestro objetivo? ¡Actualizar este chico malo a Symfony 8!

Dado que Symfony 8 requiere PHP 8.4, un buen primer paso es actualizar esta aplicación para que requiera PHP 8.4. Al mismo tiempo, también actualizaremos a la última versión de Symfony 7.3.

Si ya has hecho estas actualizaciones importantes antes, puede que estés pensando que deberíamos actualizar a 7.4, la última versión de Symfony 7. Lo haremos, pero me gusta dar pequeños pasos para evitar agobiarme con demasiados cambios a la vez.

En tu IDE, abre `composer.json`. En la sección `require`, cambia `>=8.3` por `>=8.4`:

composer.json

```
{
// ... lines 2 - 5
    "require": {
        "php": ">=8.4",
// ... lines 8 - 40
    },
// ... lines 42 - 103
}
```

Esto indica a Composer que se necesita al menos PHP 8.4 para este proyecto.

Desplázate hacia abajo y encuentra la sección `config.platform.php`. Esto es lo que utiliza la CLI de Symfony para determinar qué versión local de PHP utilizar cuando ejecutas `symfony console` o `symfony php`. Cámbialo por `8.4`:

composer.json

```
{
// ... lines 2 - 41
    "config": {
        "platform": {
            "php": "8.4"
        },
// ... lines 46 - 51
    },
// ... lines 53 - 103
}
```

Este paso no es necesario, pero desplázate hacia abajo hasta la sección `replace`. Esta es una lista de paquetes que Composer ignorará al instalar dependencias. Puedes ver que son todos paquetes polyfill. Se trata de paquetes que proporcionan características de versiones más recientes de PHP a versiones más antiguas. Como estamos en PHP 8.4, podemos añadir los paquetes polyfill para 8.3 y 8.4 a esta lista:

composer.json

```
{
// ... lines 2 - 62
    "replace": {
// ... lines 64 - 71
        "symfony/polyfill-php83": "*",
        "symfony/polyfill-php84": "*"
    },
// ... lines 75 - 103
}
```

Ahorra unos bytes de espacio en disco y unos milisegundos de tiempo de instalación...

En nuestro terminal, confirmemos que ahora estamos utilizando PHP 8.4. Ejecuta:

```
symfony php --version
```

Bien, ¡8.4.20!

Ahora, hagamos una actualización de Composer para capturar todos los cambios que hemos hecho en nuestro archivo `composer.json` y actualicemos a las últimas dependencias permitidas. Ejecuta:

```
symfony composer update
```

Sin errores, ¡genial! Salta al navegador y actualiza la página...

Qué asco, esto es un error desagradable. No se parece a los errores estándar de Symfony porque ni siquiera ha podido cargar Symfony. Se trata de un error a nivel del autoloader de Composer.

Dice que estamos ejecutando PHP 8.3 pero nuestras dependencias requieren PHP 8.4. ¿Recuerdas que antes ejecutamos el servidor Symfony en segundo plano? Lo ejecutamos con PHP 8.3. Solución sencilla, reinicia el servidor con:

```
symfony server:stop symfony server:start -d
```

Ahora actualiza la página... ¡Genial! Volvemos a estar operativos. Observa en la barra de herramientas de depuración web que ahora estamos en Symfony 7.3.11 y PHP 8.4.20.

Y ahora, ¡las depreciaciones! Cuando Symfony, o el propio PHP quieren eliminar o cambiar una característica, no hacen el cambio sin más. En lugar de eso, marcan la función como obsoleta durante un tiempo, lo que significa que sigue funcionando como se espera, pero activa una advertencia. Normalmente, la advertencia explica cómo solucionar el problema y actualizar a la nueva forma de hacer las cosas.

Symfony tiene una política de desaprobación realmente buena. Garantizan que nada se romperá al actualizar a una nueva versión menor, como de la 7.3 a la 7.4. Los cambios de ruptura se producen en la versión mayor, como pasar de la 7 a la 8. En la 7.4, todos los cambios de ruptura que se producirán en la 8 se marcan como obsoletos. Esto te da tiempo y orientación para arreglarlos. Sólo pasarás a la 8 cuando ya no queden depreciaciones.

Entonces, ¡cómo encontramos las obsoletas! Este panel de registro de depreciaciones va a ser tu mejor amigo cuando actualices Symfony.

Parece que tenemos 3. La primera es una deprecación de PHP. El método `addDroid()` de nuestra entidad `Starship` necesita que un parámetro se marque explícitamente como anulable. Arreglemos eso primero.

Abre `src/Entity/Starship.php` y desplázate hasta el método `addDroid()`. Ahh, incluso PhpStorm me avisa de esto. La solución es sencilla, añade un `?` antes de `DateTimeImmutable`:

src/Entity/Starship.php

```
↕ // ... lines 1 - 15
16 class Starship
17 {
↕ // ... lines 18 - 207
208     public function addDroid(Droid $droid, ?\DateTimeImmutable $assignedAt =
    null): static
↕ // ... lines 209 - 263
264 }
```

Vuelve al navegador, actualiza la página de inicio, y... todavía 3 deprecations... No se ha borrado. A veces, cuando se corrigen las depreciaciones, es necesario borrar manualmente la caché de Symfony para que las recoja. Esto se debe a que algunas deprecaciones se detectan en tiempo de compilación cuando se está construyendo el contenedor. Así que en tu terminal, ejecuta:

```
symfony console cache:clear
```

Actualiza la página de nuevo... Vaya, ahora tenemos 4 desaprensiones. Abre el panel. Nuestra deprecación `addDroid()` ha desaparecido, así que la hemos arreglado. A veces, cuando limpias la caché manualmente, se activan otras imprecisiones. Vale, 3 de ellas son de Doctrine.

Tendremos un capítulo entero sobre la actualización de Doctrine un poco más adelante, así que las ignoraremos por ahora.

La cuarta es una opción de configuración obsoleta de zenstruck/foundry.

Antes de arreglarla manualmente, actualicemos primero nuestras recetas Symfony Flex. A veces, simplemente actualizándolas se arreglan las obsoletas por ti.

Para actualizar las recetas, en tu terminal, ejecuta:

```
symfony composer recipe:update
```

No ha funcionado. Dice que tenemos cambios no comprometidos. El comando de actualización de recetas debe ejecutarse en una pizarra limpia, sin cambios pendientes. Ejecuta:

```
git status
```

Para ver nuestros cambios. `composer.json` `composer.lock` , y nuestro `Starship.php`. Tiene sentido, vamos a confirmarlos:

```
git commit -a -m "composer update"
```

`-a` añade todos los archivos modificados a la lista de cambios antes de confirmar y `-m` nos permite establecer un mensaje de confirmación.

Borrón y cuenta nueva, así que vuelve a ejecutar el comando de actualización de la receta:

```
symfony composer recipe:update
```

Ooo, tenemos un montón que actualizar. Los revisaremos uno a uno. Al pulsar enter utilizaremos el primero de la lista, `doctrine/deprecations`.

"No se ha modificado ningún archivo...". Ejecuta `git status` para ver qué pasa. Sólo se ha modificado el archivo `symfony.lock`. Éste es el archivo de configuración interna que Symfony Flex utiliza para llevar un registro de tus recetas. Podemos confirmar esto y seguir adelante:

```
git commit -a -m "update recipes"
```

¡Adelante! Ejecuta de nuevo el comando de actualización de recetas:

```
symfony composer recipe:update
```

`asset-mapper` es lo siguiente... Otra vez el archivo `symfony.lock`.

Cuando actualizo recetas, me gusta mantener todas las actualizaciones en un único commit. Así que modificaremos los nuevos cambios con el commit anterior con:

```
git commit -a --amend
```

Esto abre un editor de texto para actualizar opcionalmente el mensaje de confirmación. Saldré para mantenerlo igual.

Ahora actualiza la receta de `framework-bundle`... Ooo, este tiene algunos cambios reales. Lo bueno de este comando es que muestra el CHANGELOG del repositorio symfony/recipe, así que puedes indagar fácilmente para entender por qué se ha hecho un cambio.

Ejecuta `git status` para ver los cambios. Aparte del archivo `symfony.lock`, se ha modificado `public/index.php`. ¡Vamos a comprobarlo! Abre `public/index.php` y observa el cambio... Parece que se ha añadido la palabra clave `static`. Creo que es una microoptimización útil, ¡la mantendremos!

Este es un buen momento para mencionar que nunca debes actualizar recetas a ciegas. Comprueba siempre los cambios y, si no estás seguro de un cambio, sigue el registro de cambios hasta el repositorio symfony/recipe para encontrar el razonamiento.

Modifica el commit y pasa a la siguiente actualización de recetas. El `monolog-bundle` también tiene algunos cambios. Parecen de configuración del paquete, así que abre `config/packages/monolog.yaml` para verlos. Sólo se han eliminado algunos comentarios. No pasa nada... modifica el commit.

A continuación, actualiza la receta `stimulus-bundle` y observa los cambios. Abre `assets/stimulus_bootstrap.js`. Sólo se han añadido algunas cosas. Modifica el commit y pasa a actualizar la receta `twig-bundle`. Nuestra plantilla base fue modificada, así que abre `templates/base.html.twig`. Se han añadido algunas cosas de FrankenPHP. De momento no utilizamos FrankenPHP, pero no está de más dejarlo por si lo hacemos en el futuro

Modifica el commit y actualiza la última receta: `zenstruck/foundry`. Se modificó el archivo de configuración, así que abre `config/packages/zenstruck_foundry.yaml`. Sólo se ha actualizado un comentario, así que no pasa nada.

Modifica el commit y ¡habremos terminado! Ejecuta de nuevo el comando de actualización de la receta para confirmar. Bien! "Todos los paquetes parecen estar actualizados"

Antes de comprobar la aplicación, limpia de nuevo la caché...

Ahora actualiza la página de inicio. 2 desapariciones. La del cargador automático de Doctrine sigue aquí, pero la ignoraremos hasta que actualicemos Doctrine. La de Foundry también sigue aquí, así que la actualización de la receta no la ha arreglado. Vamos a arreglarlo nosotros. Dice que esta configuración de `enable_auto_refresh_with_lazy_objects` se forzará a `true` en la 3.0, así que pongámosla ahora en `true`. Copia la opción y abre`config/packages/zenstruck_foundry.yaml`. En `zenstruck_foundry`, pégala, y ajústala a `true`:

config/packages/zenstruck_foundry.yaml

```
when@dev: &dev
// ... line 2
    zenstruck_foundry:
        enable_auto_refresh_with_lazy_objects: true
// ... lines 5 - 18
```

Totalmente al margen, pero si tienes curiosidad por saber qué es esta extraña sintaxis de `&dev` y `*dev`, se trata de un ancla y un alias de YAML. El `&dev` marca esta configuración como un ancla llamada `dev`, y luego el `*dev` es un alias que hace referencia a esa ancla. Así que esto está diciendo: "para el entorno `test`, utiliza la misma configuración que `dev`". Es una forma genial de evitar la duplicación en tus archivos YAML.

Vale, vuelve al navegador y actualiza la página de inicio. ¡Ya no hay depreciaciones!

Pero... borremos la caché... y actualicemos de nuevo... ahora hay 3, pero son las de Doctrine que arreglaremos más tarde.

Vale, hemos actualizado con éxito nuestra aplicación a PHP 8.4. A continuación, actualizaremos a Symfony 7.4, la última versión de Symfony 7.

Chapter 2: Actualización a Symfony 7.4

Muy bien, vamos a meternos de lleno en la actualización a Symfony 7.4. Abre tu archivo `composer.json`. Observa que los paquetes Symfony utilizan el formato `7.3.*`. Esto es ligeramente diferente del resto de nuestros paquetes, que suelen utilizar el formato de prefijo `^`. Esto hace que sea muy fácil encontrar y actualizar los paquetes `symfony`.

Si utilizas PhpStorm, ve a editar... buscar... reemplazar. Busca `7.3.*` y sustitúyelo por `7.4.*`. Podemos ver que tenemos 19 ocurrencias que reemplazar. Pulsa reemplazar todo... y... ¡boom!

composer.json

```
{
// ... lines 2 - 5
    "require": {
// ... lines 7 - 18
        "symfony/asset": "7.4.*",
        "symfony/asset-mapper": "7.4.*",
        "symfony/console": "7.4.*",
        "symfony/dotenv": "7.4.*",
// ... line 23
        "symfony/form": "7.4.*",
        "symfony/framework-bundle": "7.4.*",
        "symfony/http-client": "7.4.*",
// ... line 27
        "symfony/property-access": "7.4.*",
        "symfony/property-info": "7.4.*",
        "symfony/runtime": "7.4.*",
        "symfony/security-csrf": "7.4.*",
        "symfony/serializer": "7.4.*",
// ... line 33
        "symfony/twig-bundle": "7.4.*",
// ... line 35
        "symfony/validator": "7.4.*",
        "symfony/yaml": "7.4.*",
// ... lines 38 - 40
    },
// ... lines 42 - 103
}
```

Vamos a hacer una doble comprobación para asegurarnos de que no se nos ha pasado ninguna. En `require`... sí, parece que está bien. Ahora en `require-dev`...

composer.json

```
{
// ... lines 2 - 96
    "require-dev": {
        "symfony/debug-bundle": "7.4.*",
// ... line 99
        "symfony/stopwatch": "7.4.*",
        "symfony/web-profiler-bundle": "7.4.*",
// ... line 102
    }
}
```

Sí, ahí también se ve bien. `maker-bundle` tiene una estrategia de versionado diferente a la de los componentes principales y bundles de Symfony. Por eso tiene un aspecto diferente.

Observa que bajo la sección `extra`, tenemos esta configuración `symfony` `require` .

composer.json

```
{
// ... lines 2 - 90
    "extra": {
        "symfony": {
// ... line 93
            "require": "7.4.*"
        }
    },
// ... lines 97 - 103
}
```

Esto le dice a Symfony Flex qué versión de Symfony debe utilizar al instalar los componentes Symfony. Algunos de nuestros componentes Symfony necesarios requieren otros componentes Symfony como dependencias. Estas se llaman dependencias transitivas (¡palabra elegante!), y pueden permitir una amplia gama de versiones de Symfony. Como Symfony 6, 7 u 8. Esta configuración garantiza que sólo se instalen las versiones `7.4`. Así que cuando actualices Symfony, es importante que también actualices esta configuración.

¡Genial! Ahora que hemos actualizado nuestro `composer.json`, vamos a ejecutar nuestra actualización de Composer:

```
symfony composer update
```

Verificando la actualización

Perfecto, ¡parece que ha funcionado! Vayamos a nuestra aplicación y actualicémosla. Sí, ahora estamos en 7.4.8, la última versión 7.4.

De vuelta a nuestro terminal, ejecuta:

```
git status
```

Nuestros archivos `composer.json` y `composer.lock` están modificados, es de esperar. Pero también tenemos este nuevo `config/reference.php`.

Ábrelo en tu editor. Se trata de un archivo autogenerado que Symfony crea al construir el contenedor. Symfony tiene ahora un formato de configuración basado en arrays PHP, una alternativa a YAML. Este archivo se genera para proporcionar un mejor autocompletado cuando se utiliza ese formato. YAML sigue siendo el formato recomendado, y lo que estamos utilizando en esta aplicación, por lo que este archivo no es importante para nosotros en este momento. Si Symfony cambia su recomendación en el futuro, ¡estaremos preparados! Echa un vistazo a esta entrada del blog para obtener más información al respecto.

Puedes añadir este archivo a tu `.gitignore` o confirmarlo. La mejor práctica ahora mismo es confirmarlo, así que vamos a hacerlo. Ejecuta en tu terminal:

```
git add config/reference.php
```

Después de ejecutar `git status` para confirmar que estamos bien, podemos confirmar con:

```
git commit -a -m "update composer"
```

(Sí, realmente deberíamos usar un mensaje de confirmación mejor aquí)

Actualizar recetas

¡Veamos si hay alguna actualización de recetas para la 7.4! Ejecuta:

```
symfony composer recipe:update
```

Sólo dos, empieza por `framework-bundle`. Ejecuta `git status` para ver los cambios. `.env` y `config/services.yaml` han sido modificadas.

Abre primero `.env`. Se ha añadido una nueva variable de entorno: `APP_SHARE_DIR`. Cuando se ejecuta Symfony en una arquitectura multiservidor, éste es un directorio que debe compartirse entre los servidores. Antes, tenías que compartir todo el directorio de caché, lo que no es idea. Esta nueva configuración te permite tener un control más preciso sobre lo que se comparte entre servidores. Si te interesa saber más sobre esto, consulta esta entrada del blog.

Abre nuestro segundo archivo modificado, `config/services.yaml`. Sólo se han modificado los comentarios de la parte superior. ¡Pero proporciona una nueva función genial! ¿Ves esto de `yaml-language-server $schema`? Esto configura un esquema JSON para este archivo. Espera, ¿JSON? Esto es YAML. Como YAML es compatible con JSON, podemos utilizar esquemas JSON para validar nuestros archivos YAML. Esto es genial, pero lo mejor es que nos proporciona autocompletado en nuestros IDEs si lo soportan. ¡Y PhpStorm lo soporta! ¡Aquí tienes una entrada de blog si quieres saber más sobre ello!

Bien, vamos a confirmar estos cambios en nuestro terminal con:

```
git commit -a -m "update recipes"
```

Ejecuta de nuevo el comando recipe update y actualiza el paquete `routing`. Ejecuta `git status` para ver los cambios. `config/routes.yaml`: ábrelo. En la parte superior, se ha añadido una configuración de esquema YAML JSON.

Abajo, mira la nueva configuración simplificada. Con la configuración anterior, sólo se cargaban los controladores en `src/Controller`. Con esta nueva configuración, cualquier clase que contenga atributos `#[Route]` se cargará como controlador. Sigue siendo la mejor práctica poner todos tus controladores en `src/Controller`. Pero si tienes una aplicación más compleja con varios dominios y sus propios controladores, éstos se cargarán independientemente de dónde se encuentren.

Vamos a confirmar estos cambios con:

```
git commit -a --amend
```

Probando la actualización

¡Perfecto! Entra en nuestra aplicación y actualiza la página de inicio para asegurarte de que todo sigue funcionando correctamente. Bien, ¡la actualización a la 7.4 ha sido un éxito!

A continuación, vamos a actualizar Doctrine y explorar una interesante mejora de la última versión del ORM. ¡Permanece atento!

Chapter 3: Actualización de Doctrine y objetos perezosos nativos

Tenemos que actualizar el bundle de Doctrine... pero antes de hacerlo, quiero refrescarte la memoria sobre una función muy interesante de Doctrine: los objetos perezosos.

Abre `src/Entity/StarshipPart.php`. Esta entidad tiene una propiedad `Starship` con una relación de muchos a uno con nuestra entidad `Starship`. Cada `StarshipPart` tiene un `Starship`, y cada `Starship` puede tener muchos `StarshipParts`. Cuando se obtienen entidades con relaciones, Doctrine utiliza cierta magia para evitar consultas innecesarias a la base de datos. Veamos cómo funciona en acción.

Configurar un controlador temporal

En tu terminal, crea un nuevo controlador:

```
symfony console make:controller
```

Nómbralo `LazyController`, y no hace falta que hagas pruebas.

Abre el nuevo controlador en `src/Controller/LazyController.php`. Vale, tenemos este método `index()` cuya ruta está establecida en `/lazy`.
Inyecta `StarshipPartRepository $repository` en él. Luego, coge la primera parte del repositorio con `$part = $repository->find(1)`. Volcarlo con `dump($part)`:

src/Controller/LazyController.php

```
// ... lines 1 - 9
final class LazyController extends AbstractController
{
    #[Route('/lazy', name: 'app_lazy')]
    public function index(StarshipPartRepository $repository): Response
    {
        $part = $repository->find(1);

        dump($part);
// ... lines 18 - 21
    }
}
```

Ahora, regresa a nuestra aplicación y navega manualmente a la url `/lazy`.

Explorando los objetos perezosos de Doctrine

Si observamos la barra de herramientas de depuración web, veremos una única consulta. Es la que obtiene el objeto `StarshipPart`. Si abrimos el panel del perfilador de volcado, veremos el objeto obtenido `StarshipPart`. Fíjate en la propiedad `starship`, su tipo es este extraño Proxy CG. Se trata de un objeto Doctrine lazy. Mira en su interior. Todas las propiedades están desactivadas excepto el ID. El ID es lo único que Doctrine conoce de `Starship` hasta que consulta a la base de datos el resto de los datos. Sólo cuando accede a una propiedad de `Starship`, Doctrine lanza una segunda consulta para obtener el resto.

¡Vamos a lanzar esta segunda consulta! En `LazyController::index()`, añade `$part->getStarship()->getName()` como primer argumento a `dump()`:

src/Controller/LazyController.php

```
// ... lines 1 - 9
final class LazyController extends AbstractController
{
// ... line 12
    public function index(StarshipPartRepository $repository): Response
    {
// ... lines 15 - 16
        dump($part->getStarship()->getName(), $part);
// ... lines 18 - 21
    }
}
```

Actualiza la página `/lazy`... Ahora hay dos consultas. La primera busca el `StarshipPart`, y la segunda busca el `Starship` porque hemos accedido a su nombre.

En el panel de volcado, podemos ver que el `Starship` está ahora completamente cargado con todas sus propiedades.

Así que este Proxy CG es una clase real que Doctrine genera sobre la marcha. Contiene toda la lógica para obtener los datos cuando sea necesario. La clave está en que extiende la entidad real `Starship`. Así es como puede utilizarse como sustituta de `Starship` hasta que se necesiten los datos reales. También por eso las entidades no pueden ser finales, necesitan ser extensibles por estas clases proxy.

Como puedes imaginar, la lógica para hacer todo esto es bastante compleja y difícil de mantener. Pero... todo eso cambió en PHP 8.4, que introdujo objetos lazy nativos en el propio PHP.

Y Doctrine Bundle 3, ¡permite que nuestras aplicaciones Symfony se aprovechen de ello! Así que ¡a actualizar!

Actualización del bundle Doctrine

En primer lugar, abre nuestro `composer.json`. Busca donde requerimos el `doctrine-bundle`. Cambia su versión a `^3.0`.

Ahora, en el terminal, ejecuta:

```
symfony composer update
```

Oooo, un error de Composer. No se han podido resolver nuestras dependencias... `composer.json` requiere `doctrine-bundle` 3. Sí... `doctrine-bundle` 3 requiere `doctrine/dbal` 4 pero esto entra en conflicto con nuestro requisito de `doctrine/dbal` 3.

Como referencia, `doctrine/dbal` es la capa de abstracción de bases de datos que Doctrine utiliza para comunicarse con diferentes bases de datos.

Comprobemos nuestro `composer.json`. Sólo requerimos `dbal` versión 3, pero `doctrine-bundle` necesita la versión 4 según ese error.

Vale, esto es un problema heredado. Las versiones anteriores de `doctrine-bundle` no soportaban `dbal` 4, así que teníamos que asegurarnos de que se utilizaba la versión 3. Esto ya no es necesario, así que podemos eliminarlo por completo, y dejar que `doctrine-bundle` decida qué versión utilizar.

Vuelve a intentar la actualización...

Otro error, pero diferente. Desplázate un poco hacia arriba... podemos ver que `doctrine-bundle` 3 y `dbal` 4 se instalaron correctamente.

El error se produjo al intentar borrar la caché. Parece que nuestra configuración de `doctrine` está utilizando algunas opciones que ya no son compatibles.

Podríamos arreglarlas manualmente, pero estoy bastante seguro de que actualizar la receta Flex lo resolverá.

Necesitamos un estado git limpio antes de actualizar la receta, así que vamos a comprobar nuestro `git status`. Tenemos algunos cambios modificados y algunos archivos sin seguimiento. Ejecuta:

```
git add .
```

Ejecuta: Para rastrearlo todo y vuelve a ejecutar `git status`. Ahora que todo está rastreado, podemos confirmarlo con:

```
git commit -a -m "upgrade doctrine-bundle"
```

`git status` de nuevo para confirmar que estamos limpios. Por último, actualiza la receta con:

```
symfony composer recipe:update
```

Efectivamente, ha encontrado una actualización para `doctrine-bundle`. ¡Aplícala!

Ejecuta `git status` para ver los cambios. Perfecto, ha actualizado el archivo `doctrine.yaml`.

Comprobación de los cambios

De vuelta a nuestro código, abre `config/packages/doctrine.yaml`. En primer lugar, ha eliminado 3 opciones de configuración. Éstas eran las que provocaban el error al borrar la caché.

A continuación, parece que ha cambiado la estrategia de nomenclatura por defecto... y ha eliminado algunas opciones de configuración del controlador.

Debajo de la configuración de producción, eliminó las opciones `auto_generate_proxy_classes` y `proxy_dir`. Con el sistema de objetos perezosos original, en producción, Doctrine generaba archivos para las clases proxy con el fin de mejorar el rendimiento.

¡Nada de eso es ya necesario con los objetos perezosos nativos!

Bien, veamos qué aspecto tienen ahora nuestros objetos perezosos. En primer lugar, vuelve a `LazyController:index()` y elimina el primer argumento de `dump()`:

src/Controller/LazyController.php

```
↕ // ... lines 1 - 9
10 final class LazyController extends AbstractController
11 {
↕ // ... line 12
13     public function index(StarshipPartRepository $repository): Response
14     {
↕ // ... lines 15 - 16
17         dump($part);
↕ // ... lines 18 - 21
22     }
23 }
```

Esto debería volcar la parte con una instancia de nave estelar que no está completamente cargada.

Actualiza la página `/lazy` en tu navegador. Vale, sólo una consulta, que es lo esperado. Abre el panel de depuración y comprueba la propiedad `starship`. Ahora es nuestra entidad Starship normal, no esa clase proxy generada.

Pero ¡mira dentro! Todas las propiedades están sin establecer excepto el ID. Esto se parece a lo que vimos en la antigua clase proxy. ¡Esto son objetos perezosos nativos en acción!

De vuelta en `LazyController::index()`, vuelve a añadir el `$part->getStarship()->getName()` al `dump()`...

src/Controller/LazyController.php

```
↕ // ... lines 1 - 9
10 final class LazyController extends AbstractController
11 {
↕ // ... line 12
13     public function index(StarshipPartRepository $repository): Response
14     {
↕ // ... lines 15 - 16
17         dump($part->getStarship()->getName(), $part);
↕ // ... lines 18 - 21
22     }
23 }
```

y vuelve a actualizar la página `/lazy`. Dos consultas, y si miramos la propiedad `starship` en el panel de volcado. Sigue siendo sólo nuestra entidad `Starship` normal... y si la expandimos... ¡se cargan todas sus propiedades!

Finalizando Entidades

Vale... esto mola, pero ¿qué significa realmente para mí como desarrollador? Bueno, probablemente haya alguna mejora de rendimiento con los objetos lazy nativos.

Pero la principal conclusión es que ahora podemos, por fin, marcar nuestras entidades como `final`. Así que, sí, no es que nos cambie la vida, pero está bien que ya no necesitemos una solución complicada.

Así que... ¡marquemos nuestras entidades como finales!

`src/Entity/Droid.php` final:

src/Entity/Droid.php

```
// ... lines 1 - 10
final class Droid
// ... lines 12 - 121
```

`Starship.php`: final:

src/Entity/StarshipPart.php

```
// ... lines 1 - 11
final class StarshipPart
// ... lines 13 - 90
```

`StarshipDroid.php`: final:

src/Entity/StarshipDroid.php

```
// ... lines 1 - 8
final class StarshipDroid
// ... lines 10 - 73
```

y finalmente `StarshipPart.php`: final:

src/Entity/StarshipPart.php

```
// ... lines 1 - 11
final class StarshipPart
// ... lines 13 - 90
```

Actualiza nuestra página perezosa... y... ¡todo sigue funcionando!

Ya casi estamos listos para dar el salto a Symfony 8, pero antes de hacerlo, volvamos a las desaprobaciones.

Chapter 4: Tracking & Fixing Deprecations

Coming soon...

Chapter 5: Upgrading to Symfony 8.0!

Coming soon...

With <3 from SymfonyCasts