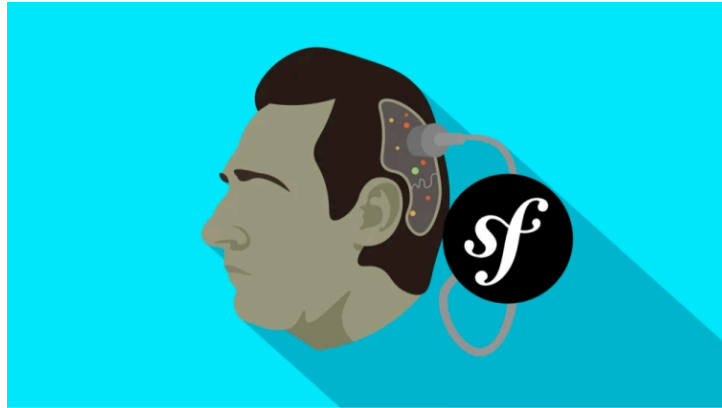


Actualizar a Symfony 8



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
```

```
1 {  
  // ... Lines 2 - 5  
6   "require": {  
7     "php": ">=8.4",  
  // ... Lines 8 - 40  
41  },  
  // ... Lines 42 - 103  
104 }
```

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
```

```
1 {  
  // ... Lines 2 - 41  
42  "config": {  
43    "platform": {  
44      "php": "8.4"  
45    },  
  // ... Lines 46 - 51  
52  },  
  // ... Lines 53 - 103  
104 }
```

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
```

```
1 {  
  // ... Lines 2 - 62  
63  "replace": {  
  // ... Lines 64 - 71  
72    "symfony/polyfill-php83": "*",  
73    "symfony/polyfill-php84": "*",  
74  },  
  // ... Lines 75 - 103  
104 }
```

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 depreciación 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 depreciaciones 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 depreciación `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
```

```
1 when@dev: &dev
```

```
↕ // ... line 2
```

```
3     zenstruck_foundry:
```

```
4         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
```

```
1 {
2 // ... Lines 2 - 5
3
4     "require": {
5 // ... Lines 7 - 18
6         "symfony/asset": "7.4.*",
7         "symfony/asset-mapper": "7.4.*",
8         "symfony/console": "7.4.*",
9         "symfony/dotenv": "7.4.*",
10 // ... Line 23
11         "symfony/form": "7.4.*",
12         "symfony/framework-bundle": "7.4.*",
13         "symfony/http-client": "7.4.*",
14 // ... Line 27
15         "symfony/property-access": "7.4.*",
16         "symfony/property-info": "7.4.*",
17         "symfony/runtime": "7.4.*",
18         "symfony/security-csrf": "7.4.*",
19         "symfony/serializer": "7.4.*",
20 // ... Line 33
21         "symfony/twig-bundle": "7.4.*",
22 // ... Line 35
23         "symfony/validator": "7.4.*",
24         "symfony/yaml": "7.4.*",
25 // ... Lines 38 - 40
26     },
27 // ... Lines 42 - 103
28 }
29 }
```

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
```

```
1 {
  // ... Lines 2 - 96
97   "require-dev": {
98     "symfony/debug-bundle": "7.4.*",
  // ... Line 99
100    "symfony/stopwatch": "7.4.*",
101    "symfony/web-profiler-bundle": "7.4.*",
  // ... Line 102
103  }
104 }
```

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
```

```
1 {
  // ... Lines 2 - 90
91   "extra": {
92     "symfony": {
  // ... Line 93
94       "require": "7.4.*"
95     }
96   },
  // ... Lines 97 - 103
104 }
```

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
```

Nómbalo `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
10 final class LazyController extends AbstractController
11 {
12     #[Route('/lazy', name: 'app_lazy')]
13     public function index(StarshipPartRepository $repository): Response
14     {
15         $part = $repository->find(1);
16
17         dump($part);
↕ // ... Lines 18 - 21
22     }
23 }
```

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
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 }
```

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:

A screenshot of a terminal window with a dark blue header and a light gray body. The terminal shows the command `symfony composer update` being entered. There are three white circles in the top left corner of the terminal header, representing window control buttons.

```
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 "update 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`. ¡Aplicala!

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.phpfinal:
```

```
src/Entity/Droid.php
```

```
↕ // ... Lines 1 - 10
```

```
11 final class Droid
```

```
↕ // ... Lines 12 - 121
```

Starship.php: final:

```
src/Entity/StarshipPart.php
```

```
↕ // ... Lines 1 - 11
```

```
12 final class StarshipPart
```

```
↕ // ... Lines 13 - 90
```

StarshipDroid.php: final:

```
src/Entity/StarshipDroid.php
```

```
↕ // ... Lines 1 - 8
```

```
9 final class StarshipDroid
```

```
↕ // ... Lines 10 - 73
```

y finalmente StarshipPart.php: final:

```
src/Entity/StarshipPart.php
```

```
↕ // ... Lines 1 - 11
```

```
12 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: Rastrear y corregir las desapropiaciones

Volvamos a las deprecaciones, ya que es importante corregirlas. Como se mencionó anteriormente, antes de que puedas saltar con seguridad a Symfony 8.0, debes asegurarte de que estás ejecutando Symfony 7.4 libre de depreciaciones. Esto significa que no debería haber ninguna deprecación en toda tu aplicación.

Sabemos que podemos encontrar las deprecaciones en la barra de herramientas de depuración web, pero... esto sólo muestra las deprecaciones de la página actual... Aquí no hay ninguna... pero haz clic en esta nave estelar. Esta página sí tiene una desaprobación.

Profundizando en el panel de obsoletos

Exploremos el panel de obsoletos con más detalle. Haz clic en el icono de la barra de herramientas para abrirlo. Esta desaprobación parece provenir de Doctrine, algo sobre el acceso a valores de campo sin procesar... Pero ¡mira esto! ¡Un enlace a un pull request justo en el mensaje! Muy útil, ¡me encanta! Copia el enlace y pégalo en tu navegador...

Este es el PR de Doctrine que modificó algunos comportamientos y provocó esta desaprobación. Parece que soluciona algunos problemas de memoria. Hay una gran discusión para ayudarnos a tener más contexto si queremos. Pero, ¿cómo lo arreglamos? En la descripción del PR, hay una sección de ruta de migración. El nuevo comportamiento es opcional, y puede activarse pasando `accessRawFieldValues: true` al constructor `Criteria`.

Vale... ¿cómo encontramos dónde necesita nuestra aplicación esta corrección? ¡El panel de desaprobación al rescate!

Haz clic en "mostrar seguimiento" debajo del mensaje de desaprobación. Este es el seguimiento de cómo tu aplicación ha llegado a esta desaprobación. Puedes leerlo de arriba abajo e ignorar las líneas que no están en tu directorio `src`. La primera línea de nuestro directorio `src` es probablemente la que está provocando la eliminación. En este caso, es la línea 23 de `StarshipPartRepository`. Incluso podemos ampliarla para ver el código real que está provocando la eliminación. ¡Genial!

También en este panel está el enlace "mostrar contexto". Muestra la excepción real lanzada, pero en su mayor parte muestra la misma información. El mensaje... y el seguimiento de la pila. Normalmente, sólo utilizo el enlace de seguimiento de la pila.

Así que... para encontrar todas las imprecisiones con la barra de herramientas de depuración web, tendríamos que ir localmente a cada página, a cada envío de formulario, a cada ruta de la API, etcétera. Eso no es muy eficiente. Entonces, ¿cuál es la mejor forma de encontrar las desaprobaciones?

Hay dos formas alternativas principales. La primera es tu conjunto de pruebas. PHPUnit puede configurarse para mostrar las depreciaciones, pero... a menos que tengas una cobertura de pruebas del 100%, esto no lo detectará todo. Vaya, ni siquiera tengo pruebas en esta aplicación...

Registrar las imprecisiones

La última forma es un método infalible. Despliegas tu aplicación Symfony 7.4 en producción durante un tiempo, unas semanas o un mes. Durante ese tiempo, registra todas las depreciaciones que se produzcan en producción. Recuerda que las depreciaciones no son errores, por lo que no romperán tu aplicación. Cuando se produzcan, o al final del periodo de tiempo, revisa esos registros y corrige las depreciaciones. Haz este ciclo unas cuantas veces hasta que no tengas más depreciaciones en producción. Ahora estás listo para actualizar a Symfony 8.0 de forma segura.

Veamos cómo funciona el registro y cómo se puede mejorar. Abre `config/packages/monolog.yaml`. Desplázate hasta el gestor `deprecation` en la configuración de producción. Este gestor sólo escribe las depreciaciones en el flujo de error estándar. Por supuesto, puedes personalizarlo a tu gusto. Mi favorito es llenar de ellas el canal de Slack de mi equipo

Vamos a añadir este controlador a nuestro entorno de desarrollo para verlo en acción. Copia el manejador `deprecation` y pégalo en la sección `when@dev`... No quiero enviar esto a error estándar aquí, así que copia la ruta del manejador anterior y pégala aquí. Sufija el archivo con `deprecations.json`:

```
config/packages/monolog.yaml
```

```
↕ // ... Lines 1 - 4
```

```
5 when@dev:  
6     monolog:  
7         handlers:
```

```
↕ // ... Lines 8 - 16
```

```
17             deprecation:  
18                 type: stream  
19                 channels: [ deprecation ]  
20                 path: "%kernel.logs_dir%/%kernel.environment%.deprecations.json"  
21                 formatter: monolog.formatter.json
```

```
↕ // ... Lines 22 - 61
```

Esto ya está usando el formateador JSON y usar esa extensión me ayudará a hacerlo bonito en PhpStorm.

De vuelta en nuestra aplicación, actualiza la página que provoca la depreciación. Ahora, abre `var/log`... Hmm, no veo el archivo. Quizás PhpStorm aún no lo ha cogido. Lo recargaré desde el disco... ¡y ahí está!

`dev.deprecations.json`. Ábrelo. Daré formato al código para que sea más fácil de leer.

Incluir trazas de pila en los registros

Vemos el mismo mensaje de desaprobación que vimos en el panel del perfilador... y un montón de cosas más... pero... falta la traza de pila. Por defecto, es difícil saber dónde se dispara la depreciación.

Por suerte, ¡podemos incluir el seguimiento de pila! Borraré este archivo de registro para que podamos empezar de cero.

De vuelta en `monolog.yaml`, en la configuración de nuestro gestor de desaprobación dev, añade `include_stacktraces: true`:

```
config/packages/monolog.yaml
```

```
↕ // ... Lines 1 - 4
```

```
5 when@dev:  
6     monolog:  
7         handlers:
```

```
↕ // ... Lines 8 - 16
```

```
17             deprecation:
```

```
↕ // ... Lines 18 - 21
```

```
22                 include_stacktraces: true
```

```
↕ // ... Lines 23 - 62
```

Actualiza la página... comprueba el archivo de registro... formatealo... ¡y ya está! El seguimiento de pila tiene el mismo aspecto que en el panel del perfilador. Y efectivamente, podemos ver que la línea 23 de `StarshipPartRepository` está activando la depreciación.

Procesadores de registro

Hay una cosa que creo que todavía falta... Estaría bien saber qué URL lo desencadenó. Esto ayudaría a duplicar el problema localmente y confirmar la solución. Monolog tiene algo llamado procesadores que pueden añadir datos adicionales a las entradas del registro. Hay un montón de procesadores incorporados, uno para añadir detalles de autenticación, detalles de comandos de consola e información de depuración adicional. El que quiero activar es el `WebProcessor`. Éste añade los detalles de la petición, como la URL, el método HTTP y la IP.

Los procesadores incorporados no se registran como servicios por defecto, pero es superfácil hacerlo nosotros mismos.

Volveré a borrar este archivo de registro.

Abre `config/services.yaml` y en la parte inferior, debajo de `services`, añade `monolog.processor.web` con la clase `WebProcessor`. Elige la del Puente Monolog, ya que está preparada para funcionar con Symfony:

```
config/services.yaml
```

```
↕ // ... Lines 1 - 10
```

```
11 services:
```

```
↕ // ... Lines 12 - 24
```

```
25     monolog.processor.web:
```

```
26         class: Symfony\Bridge\Monolog\Processor\WebProcessor
```

¡Ya está! Está autocableado y autoconfigurado, así que no necesitamos hacer nada más.

Vuelve a actualizar la página... comprueba el archivo de registro... formatéalo... ¡y ya está!
Tenemos el mensaje, la traza de la pila, y abajo del todo, en `extra`, tenemos la URL, la IP y el método HTTP.

Sólo tienes que modificar nuestra configuración para producción en tus aplicaciones y desplegar. ¡Encontrar, duplicar y corregir depreciaciones será pan comido!

Arreglar la depreciación

Bien, ¡ahora vamos a arreglar esta depreciación! Se activó en `StarshipPartRepository` en la línea 23, ¿verdad? Abre eso... y busca la línea 23... Recuerda que tenemos que pasar `true` al constructor `Criteria`. Este `Criteria::create()` no es el constructor real, sino que es un constructor con nombre. Cuando saltamos a ese método... efectivamente, tiene el parámetro `accessRawFieldValues`... pero está comentado. ¿Qué pasa con eso? Forma parte del proceso de desaprobación. No pueden añadir el nuevo parámetro sin más. Esta clase y este método no son definitivos, por lo que, para mantener la compatibilidad con versiones anteriores, tienen que conservar la firma actual, por lo que comentan el nuevo parámetro y activan la depreciación si no se pasa. Esto de `func_num_args()` es una forma de comprobar si se ha pasado el nuevo parámetro sin añadirlo realmente a la firma del método.

La próxima versión principal de este paquete tendrá el parámetro real en la firma del método, y se eliminará la forma antigua. Este es el último paso del ciclo de desaprobación.

Basta de charla, ¡arreglémoslo! De vuelta en `StarshipPartRepository`, pasa `true` a `Criteria::create()`:

```
src/Repository/StarshipPartRepository.php
```

```
↕ // ... lines 1 - 13
```

```
14 class StarshipPartRepository extends ServiceEntityRepository
15 {
```

```
↕ // ... lines 16 - 20
```

```
21     public static function createExpensiveCriteria(): Criteria
22     {
23         return Criteria::create(true)->andWhere(Criteria::expr()->gt('price',
           50000));
24     }
```

```
↕ // ... lines 25 - 56
```

```
57 }
```

De vuelta al navegador. Cuando actualicemos, esta desaprobación debería haber desaparecido... ¡Y así es!

¡Por fin estamos listos para pasar a Symfony 8! ¡Eso a continuación!

Chapter 5: ¡Actualizando a Symfony 8.0!

¿Symfony 7.4? Comprobado. ¿Sin obsoletos? Comprobado.

Bien, ¡por fin ha llegado el momento de actualizar a Symfony 8! Por suerte, ya hemos hecho la mayor parte del trabajo duro, así que el proceso de actualización debería ser sencillo.

Empecemos abriendo nuestro archivo `composer.json`.

Haremos lo mismo que hicimos al actualizar a 7.4. Editar... Busca... Reemplaza... `7.4.*` y sustitúyelo por `8.0.*`. Reemplaza todo... Bien.

Voy a comprobarlo un momento. La sección `require` tiene buen aspecto... Ah, genial, nuestra opción `extra symfony require` se ha actualizado. ¡Y `require-dev` también se ve bien!

composer.json

```
1 {
2 // ... Lines 2 - 5
3
4     "require": {
5 // ... Lines 7 - 17
6         "symfony/asset": "8.0.*",
7         "symfony/asset-mapper": "8.0.*",
8         "symfony/console": "8.0.*",
9         "symfony/dotenv": "8.0.*",
10 // ... Line 22
11         "symfony/form": "8.0.*",
12         "symfony/framework-bundle": "8.0.*",
13         "symfony/http-client": "8.0.*",
14 // ... Line 26
15         "symfony/property-access": "8.0.*",
16         "symfony/property-info": "8.0.*",
17         "symfony/runtime": "8.0.*",
18         "symfony/security-csrf": "8.0.*",
19         "symfony/serializer": "8.0.*",
20 // ... Line 32
21         "symfony/twig-bundle": "8.0.*",
22 // ... Line 34
23         "symfony/validator": "8.0.*",
24         "symfony/yaml": "8.0.*",
25 // ... Lines 37 - 39
26     },
27 // ... Lines 41 - 89
28     "extra": {
29         "symfony": {
30 // ... Line 92
31             "require": "8.0.*"
32         }
33     },
34     "require-dev": {
35 // ... Line 98
36         "symfony/stopwatch": "8.0.*",
37         "symfony/web-profiler-bundle": "8.0.*",
38 // ... Line 101
39     }
40 }
41 }
```

Dirígete al terminal y ejecuta:

```
symfony composer update
```

Resolver errores

Ooo, un error, desplacémonos un poco hacia arriba. Vale, nuestro composer.json raíz requiere `symfonycasts/tailwind-bundle ^0.9.0` . Este bundle requiere algunos paquetes de Symfony, pero por desgracia, las versiones de Symfony 8 no son compatibles. Veamos si hay una versión actualizada.

En el navegador, ve a packagist.org. Busca `tailwind-bundle`... Aquí lo tienes. Aquí a la derecha, están todas las versiones, selecciona `v0.9.0`, la versión en la que estamos. Efectivamente, los paquetes Symfony que requiere sólo soportan hasta Symfony 7. Aquí está la versión más reciente: `v0.12.0`, y ¡qué bien, ésta sí es compatible con Symfony 8!

De vuelta a nuestro `composer.json`... busca el `tailwind-bundle`... y cambia su versión a `^0.12.0`:

```
composer.json
1 {
2 // ... Lines 2 - 5
6     "require": {
7 // ... Lines 7 - 36
37         "symfonycasts/tailwind-bundle": "^0.12.0",
38 // ... Lines 38 - 39
40     },
41 // ... Lines 41 - 102
103 }
```

Probemos de nuevo la actualización.

```
symfony composer update
```

Resolviendo más errores

Huh, otro error. Veamos qué está pasando. Nuestro `composer.json` requiere `monolog-bundle ^3.0`, pero parece que `monolog-bundle 3` no es compatible con Symfony 8. Volvamos rápidamente a Packagist y busquemos `monolog-bundle`.

Sí, la versión 3 no soporta Symfony 8... ¡pero la versión 4 sí!

Volvamos a `composer.json`, busquemos `monolog-bundle`... y cambiemos su versión a `^4.0`:

```
composer.json
1 {
  ↕ // ... Lines 2 - 5
6     "require": {
  ↕ // ... Lines 7 - 25
26        "symfony/monolog-bundle": "^4.0",
  ↕ // ... Lines 27 - 39
40    },
  ↕ // ... Lines 41 - 102
103 }
```

Note

Si te preguntas por qué `monolog-bundle` utiliza una estrategia de versionado diferente a la del resto de Symfony, es porque tiene su propio repositorio y ciclo de publicación. No forma parte del monorepo del núcleo de Symfony.

¿Será la tercera actualización la vencida?

```
symfony composer update
```

¡Perfecto! ¡Esta vez ha funcionado! Confirmémoslo ejecutando:

```
symfony console --version
```

¡Excelente! ¡Symfony 8.0.10!

Comprobación de dependencias obsoletas

Ahora déjame mostrarte un práctico comando de Composer que comprueba si tienes dependencias desactualizadas. Es decir, dependencias que tienen versiones más recientes disponibles.

```
symfony composer outdated -D
```

La opción `-D` significa dependencias directas, es decir, sólo los paquetes que están en el archivo `composer.json` de nuestra aplicación. No las dependencias transitivas, que son las dependencias de nuestras dependencias. En realidad, sólo me interesan las directas.

Parece que tenemos 3 paquetes que tienen versiones más recientes disponibles.

`phpdocumentor/reflection-docblock`, `symfony/stimulus-bundle` y `symfony/ux-turbo` pueden actualizarse a la siguiente versión mayor. Vamos a hacerlo en nuestro archivo `composer.json`.

Busca `phpdocumentor/reflection-docblock` y cámbialo por `^6.0`. A continuación, `stimulus-bundle`, cámbialo por `^3.0`. Por último, `ux-turbo`, cámbialo por `^3.0`:

```
composer.json
```

```
1 {
  // ... Lines 2 - 5
6   "require": {
  // ... Lines 7 - 14
15    "phpdocumentor/reflection-docblock": "^6.0",
  // ... Lines 16 - 31
32    "symfony/stimulus-bundle": "^3.0",
  // ... Line 33
34    "symfony/ux-turbo": "^3.0",
  // ... Lines 35 - 39
40   },
  // ... Lines 41 - 102
103 }
```

¡Actualicemos!

```
symfony composer update
```

¡Excelente! Que no haya errores significa que no hay conflictos con los paquetes actualizados, así que estamos listos. Ejecutar `git status` confirma que sólo han cambiado nuestros archivos `composer.lock`, `composer.json` y `reference.php`. Recuerda que el archivo `reference.php` ayuda a tu IDE con la configuración de Symfony basada en matrices y se genera automáticamente. Podemos ignorarlo ya que utilizamos una configuración basada en YAML.

Confirma nuestros cambios con:

```
git commit -a -m "upgrade to Symfony 8!"
```

Hmm, al terminal no le ha gustado mi entusiasmo con el signo de exclamación... Cancelaré este comando y volveré a intentarlo sin él:

```
git commit -a -m "upgrade to Symfony 8"
```

¡Ya está!

Comprobación de actualizaciones de recetas

Ahora que hemos hecho borrón y cuenta nueva, comprueba si hay actualizaciones de recetas con:

```
symfony composer recipe:update
```

No, ¡no hay recetas nuevas!

Verificar la actualización

Vuelve al navegador y actualiza la página de inicio. ¡Genial! ¡Estamos en Symfony 8 sin errores!

Ya que hemos actualizado `ux-turbo`, voy a confirmar que funciona haciendo clic en un enlace... Las peticiones turbo se realizan mediante AJAX y, efectivamente, nuestra barra de herramientas de depuración web muestra que se ha realizado una petición AJAX.

También hemos actualizado `stimulus-bundle`, así que vamos a comprobar que funciona correctamente. Abre las herramientas de desarrollador y comprueba la pestaña de la consola... Estos registros muestran que Stimulus y nuestros controladores Stimulus se están inicializando correctamente

¡Eso es todo, amigos! Espero que hayas aprendido algo valioso de este tutorial y consigas actualizar tus aplicaciones a Symfony 8 sin problemas.

Hasta la próxima, ¡feliz programación!

Chapter 6: Auditoría de Composer y actualizaciones de seguridad

¡Una última cosa! Un tema extra: las actualizaciones de seguridad. Esto no tiene por qué estar relacionado con la actualización a Symfony 8, pero es un tema importante, ahora más que nunca.

Hace poco, el ecosistema de Symfony se sometió a una gran revisión de seguridad que sacó a la luz muchas vulnerabilidades en varios paquetes. ¿El motivo? La IA ha cambiado las reglas del juego.

Las modernas herramientas de auditoría asistidas por IA pueden ayudar a los investigadores de seguridad a analizar grandes bases de código e identificar posibles vulnerabilidades mucho más rápido que antes. Esto está dando lugar a más descubrimientos, más revelaciones y, en última instancia, más correcciones.

Y eso es algo bueno.

Encontrar estas vulnerabilidades no significa que Symfony sea cada vez menos seguro. Más bien al contrario. Unas herramientas mejores permiten que los problemas se identifiquen y se solucionen más rápido.

Como desarrolladores, eso significa que es más importante que nunca estar al día de las actualizaciones de las dependencias y prestar atención cuando se publican avisos de seguridad.

Por suerte, Composer tiene una herramienta integrada que lo hace súper fácil.

Composer Audit

Para verlo en acción, en tu terminal, ejecuta:



```
symfony composer audit
```

Esto comprueba los paquetes que tienes instalados actualmente en tu proyecto con respecto a los avisos de seguridad conocidos.

Y... ¡parece que tengo unas cuantas! Este resultado se ve bastante mal en esta pantalla tan pequeña, así que volveré a ejecutar el comando con `--format=plain`.

Para cada vulnerabilidad, Composer muestra el paquete afectado, un nivel de gravedad y enlaces donde podemos obtener más información sobre el problema.

La mayoría de las veces, la solución es sencilla: actualiza tus dependencias.

A menudo, Composer actualizará el paquete vulnerable a una versión parcheada y el problema desaparece.

Pero no siempre.

A veces, la corrección solo está disponible en una versión principal más reciente a la que aún no estás preparado para actualizar. Otras veces, la actualización introduce cambios que provocan fallos en tu aplicación.

Si te encuentras en cualquiera de estos casos, es importante que entiendas la vulnerabilidad y el riesgo que supone para tu aplicación.

Sigue el aviso

Empecemos por la última vulnerabilidad de mi lista.

Cada aviso incluye una URL con más información. Si abrimos esta, nos redirige a una entrada de blog en la web de Symfony con todos los detalles.

Normalmente, este es el mejor punto de partida para nuestra investigación.

La entrada explica:

- en qué consiste la vulnerabilidad;
- qué versiones se ven afectadas;
- en qué circunstancias se puede aprovechar;
- y qué versiones incluyen la corrección.

No todos los paquetes utilizan una entrada de blog para compartir los detalles de la vulnerabilidad. Volviendo a nuestra terminal, podemos seguir el enlace «Advisory ID», que nos lleva al aviso en packagist.org. Aquí se muestran los detalles relacionados con el paquete y, si se creó a través de GitHub, el enlace al aviso de GitHub.

Es este enlace de GHSA de aquí. Si lo sigues, verás una página estandarizada sobre la vulnerabilidad. Aquí tienes el nombre del paquete, las versiones afectadas, las versiones parcheadas, el nivel de gravedad y una descripción del problema y cómo lo resuelve la corrección.

Para ver todos los avisos de seguridad publicados sobre un paquete de GitHub, desde la página de su repositorio, haz clic en «Seguridad y calidad». Aquí está la lista completa. Al hacer clic en uno de ellos, se abre su página de aviso.

¿Qué es un CVE?

Seguramente te habrás dado cuenta de que cada vulnerabilidad tiene un identificador CVE.

¿Qué es eso?

CVE son las siglas de «Common Vulnerabilities and Exposures» (Vulnerabilidades y exposiciones comunes).

Es una base de datos pública de vulnerabilidades de seguridad en la que a cada problema se le asigna un identificador único. El objetivo es sencillo: ofrecer a todo el mundo una forma común de referirse a la misma vulnerabilidad.

Y esto no es específico de PHP, Composer ni siquiera del software de código abierto. Los CVE se utilizan en todo el sector del software. Los sistemas operativos, los navegadores web, las bases de datos, las plataformas en la nube, los dispositivos de hardware y las bibliotecas de código abierto pueden tener asignados CVE.

Por ejemplo, si alguien menciona `CVE-2026-46634`, los desarrolladores, los investigadores de seguridad, los responsables de paquetes y las herramientas de seguridad saben que están hablando exactamente del mismo problema.

Los CVE los asignan unas organizaciones llamadas «Autoridades de Numeración de CVE» o CNA.

GitHub es una CNA, lo cual es una gran noticia para los mantenedores de código abierto, ya que pueden crear avisos de seguridad y realizar peticiones para obtener identificadores CVE directamente a través de las herramientas de GitHub.

Puedes consultar los registros oficiales de CVE en cve.org.

En cuanto a las vulnerabilidades creadas a través de GitHub, me parece que su interfaz es un poco más fácil de usar que la de cve.org, pero es bueno conocer ambos sitios.

Ignorar una vulnerabilidad

Si no puedes actualizar a una versión parcheada de inmediato, y entiendes la vulnerabilidad y el riesgo que supone, tienes la opción de ignorarla con Composer.

Copia el ID del aviso (el identificador CVE también sirve) y abre tu archivo `composer.json`.

Bajo la clave « `config` », añade una sección « `audit` » con una clave « `ignore` ». Usa el ID que has copiado como clave... y, como valor, indica el motivo por el que la ignoras. Esto es importante para que tú mismo en el futuro y tus compañeros de equipo entendáis por qué aceptasteis el riesgo. A mí también me gusta incluir cuándo podremos eliminarla.

```
composer.json
```

```
1 {
  ↕ // ... Lines 2 - 40
41     "config": {
  ↕ // ... Lines 42 - 50
51         "audit": {
52             "ignore": {
53                 "PKSA-21g2-dzjv-sky5": "Risk acceptable, remove when upgrading to
    Twig 4"
54             }
55         }
56     },
  ↕ // ... Lines 57 - 107
108 }
```

Ahora, si ejecuto:

```
symfony composer audit
```

La vulnerabilidad no ha desaparecido. En cambio, se ha trasladado a una sección de «Avisos ignorados», justo al principio de la lista. Aquí también se muestra el motivo.

Es importante que no la ocultemos por completo. Simplemente debe quedar constancia de que la hemos revisado, entendemos el riesgo y, de momento, hemos decidido no abordarla.

Por supuesto, ignorar una vulnerabilidad debería ser la excepción, no la regla. Siempre que sea posible, actualizar a una versión parcheada es la opción más segura.

Vamos a eliminar esta configuración de «ignorar» y a solucionar el problema como es debido.

Aplicar las correcciones

Actualiza nuestras dependencias con:



```
symfony composer update
```

“No se han encontrado avisos de vulnerabilidades de seguridad.”

Genial, parece que ha funcionado. Comprobémoslo ejecutando de nuevo el comando de auditoría.



```
symfony composer audit
```

¡Genial! Ya no hay vulnerabilidades.

Automatizar las auditorías de seguridad

Por último, no te fíes de tu memoria para ejecutar las auditorías manualmente. Automatízalas.

Si usas GitHub para alojar tu código, una solución estupenda es una acción programada de GitHub que ejecute `composer audit` de forma periódica. Al ejecutar ese comando, la acción fallará si se encuentran vulnerabilidades, lo que os alertará a ti y a tu equipo del problema.

¡Te voy a enseñar la que yo uso! Crea un nuevo archivo:

`.github/workflows/composer-audit.yaml`. Voy a pegar la definición, pero también la puedes encontrar en el script de abajo:

```
.github/workflows/composer-audit.yaml
```

```
1 name: Composer Security Audit
2
3 on:
4   schedule:
5     - cron: '0 12 * * 1' # Every Monday at 12:00 PM (Noon) UTC
6
7 permissions:
8   contents: read
9
10 jobs:
11   composer-audit:
12     runs-on: ubuntu-latest
13
14     steps:
15       - name: Checkout code
16         uses: actions/checkout@v6
17
18       - name: Setup Composer
19         uses: shivammathur/setup-php@v2
20         with:
21           tools: composer
22
23       - name: Run Composer audit
24         run: composer audit --locked
25
26 #     - name: Notify Slack on failure
27 #       if: failure()
28 #       uses: slackapi/slack-github-action@v3.0.3
29 #       with:
30 #         method: chat.postMessage
31 #         token: ${ secrets.SLACK_TOKEN }
32 #         payload: |
33 #           channel: <CHANNEL_ID>
34 #           username: "Composer Audit Failed"
35 #           icon_emoji: ":rotating_light:"
36 #           text: |
37 #             Repo: ${ github.repository }
38 #             Branch: ${ github.ref_name }
39 #
40 #             <${ github.server_url }/${ github.repository
41 #             }/actions/runs/${ github.run_id }|View details>
```

Repasemos este flujo de trabajo. En primer lugar, se activa según una programación: todos los lunes a mediodía (UTC).

Note

Ten en cuenta que las tareas programadas se ejecutan en tu rama predeterminada.

Vale, pasemos a la tarea en sí. Primero, hacemos un checkout del código. Luego configuramos PHP y Composer usando una acción muy habitual. Por último, ejecutamos `composer audit`. Fíjate en este indicador « `--locked` ». Esto le indica a Composer que compruebe las versiones instaladas en `composer.lock` con respecto a los avisos de seguridad. Así nos ahorramos tener que ejecutar `composer install`, lo que nos ahorra unos minutos de acción. Además, significa que no tenemos que especificar la versión de PHP en el paso anterior, lo que simplifica el proceso.

Esta tarea fallará si se encuentra alguna vulnerabilidad que no esté ignorada.

Si tú y tu equipo usáis Slack, esta sección comentada es un ejemplo de cómo puedes publicar una alerta allí en `failure()`.

Tip

Un último consejo: cuando sea posible, crea pull requests específicas para las actualizaciones de seguridad. Mantenerlas separadas de el resto del trabajo facilita las revisiones y ayuda a garantizar que las correcciones de seguridad se fusionen rápidamente.

Y ya está.

Actualizar tu aplicación es solo una parte del proceso. El mantenimiento continuo implica estar atento a tus dependencias, entender los avisos de seguridad y aplicar las actualizaciones cuando sea necesario.

Por suerte, Composer nos ofrece herramientas excelentes para que ese proceso sea mucho más sencillo.

Go Deeper!

Para más información, echa un vistazo a nuestra entrada del blog sobre [avisos de seguridad de Composer](#).

Hasta la próxima, ¡Que disfrutes programando sin vulnerabilidades!

With <3 from SymphonyCasts