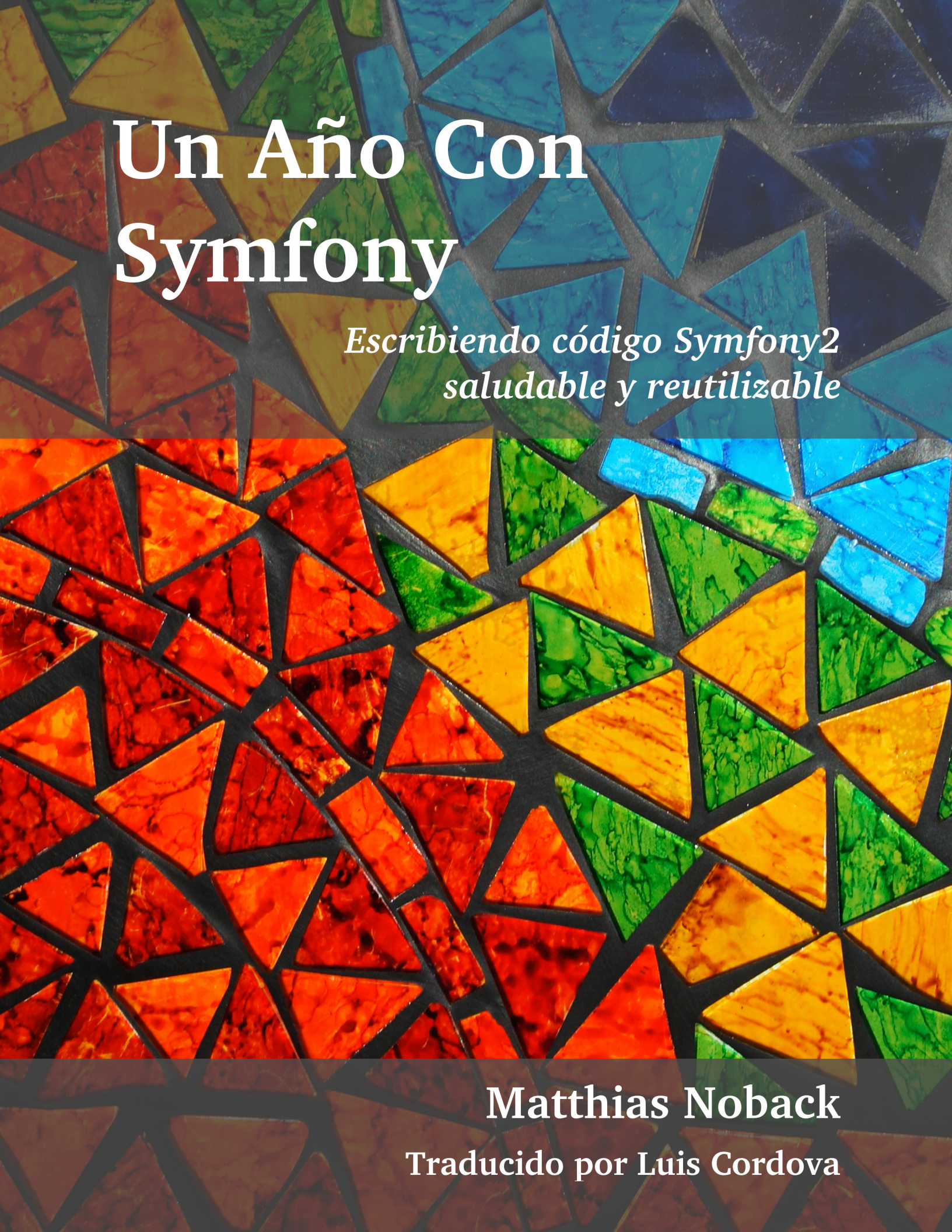

Un Año Con Symfony

Escribiendo código Symfony2 saludable y reutilizable

Matthias Noback

Traducido por Luis Cordova

Un Año Con Symfony

Escribiendo código Symfony2 saludable y reutilizable

Matthias Noback y Luis Cordova

Este libro está a la venta en http://leanpub.com/un-ano-con-symfony

Esta versión se publicó en 2014-05-05

To Liesbeth and Lucas, because this is... Life

Índice general

Sobre esta muestra

Este archivo contiene muestras representativas del libro mismo. Éstas son extraídas de las primeras dos partes y la última parte. ¡Espero que les gusten!

Panorama de los contenidos

La primera parte de este libro se llama El viaje desde la petición hasta la respuesta. Te llevará a lo largo desde el punto de entrada de una aplicación Symfony en el controlador frontal hasta el último respiro que da antes de enviar la respuesta de vuelta al cliente. A veces te mostraré cómo puedes colgarte en el proceso y modificación de flujo o simplemente cambiar los resultados de los pasos intermedios.

La siguiente parte se llama Patrones de inyección de dependencias. Ésta contiene una colección de patrones que son soluciones a problemas recurrentes cuando se quiere crear o modificar definiciones de servicio, basado en la configuración de un bundle. Te mostraré muchos ejemplos prácticos que puedes usar para modelar tu propia extensión del contenedor, clase de configuración y pasos de compilador desde tu bundle.

La tercera parte es sobre Estrucura de Proyecto. Sugiero varias maneras de obtener controladores más limpios, delegando acciones vía los form handlers, domain managers y mediante event listeners. Daremos también una mirada al estado y cómo evitarlo en la capa de servicios de tu aplicación.

Le sigue un rápido intermezzo acerca de las Convenciones de Configuration. Esta parte debiera de ayudarte estableciendo la configuración de tu aplicación. Anima a tí y a tu equipo a acordar sobre un tipo de convención sobre la configuración.

La quinta parte es muy importante para toda aplicación seria, con sesiones de usuario y datos sensibles. Trata sobre la Seguridad. Pareciera estar cubierta para todos los componentes Symfony (despues de todo el framework mismo ha sido auditado por problemas de seguridad), y Twig, pero desafortunadamente tal cosa no sería posible. Siempre tienes que pensar tú mismo en la seguridad. Esta parte del libro contiene varias sugerencias sobre cómo lidiar con la seguridad, dónde dirigir nuestra mirada, cuándo podemos confiar en el framework y cuándo necesitas tomar las cosas en tus manos con respecto a la seguridad.

La parte final curbre todas las maneras de [Ser un desarrollador Symfony] (#being-a-symfony-developer), sin embargo, en realidad esta parte es un gran ánimo a *no* ser un desarrollador Symfony sino más bien a escribir las cosas tan flojas y desacopladas al framework Symfony como sea posible. Esto significa separar el código en partes de código reutilizables y partes de código específicas del proyecto, luego separar el código reutilizable en librerías y en código de bundles. Discutiré otras diferentes ideas que harán tus bundles más pulcros, limpios y amigables para otros proyectos.

Disfrútalo!

I El viaje desde la petición a la respuesta

1 La interfaz `HttpKernelInterface`

Symfony es famoso por su `HttpKernelInterface`:

```
namespace Symfony\Component\HttpKernel;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

interface HttpKernelInterface
{
    const MASTER_REQUEST = 1;
    const SUB_REQUEST = 2;

    /**
      * @return Response
      */
    public function handle(
        Request $request,
        $type = self::MASTER_REQUEST,
        $catch = true
    );
}
```

Una implementación de esta interfaz sólo tendría que implementar un método y declarase capaz, por lo tanto, de convertir de alguna manera una petición `Request` en una respuesta `Response`. Cuando observas cualquiera de los controladores frontales en el directorio `/web` de un proyecto Symfony, verás que este método `handle()` juega un papel central en el procesamiento de peticiones web - como esperarías:

```
// en /web/app.php
$kernel = new AppKernel('prod', false);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
```

Primero, el `AppKernel` es instanciado. Esta clase es específica de tu proyecto, y la puedes encontrar en `/app/AppKernel.php`. Te permite registrar tus bundles, y cambiar algunas configuraciones globales,

como la ubicación del directorio de caché o el archivo de configuración que debe ser cargado. Los argumentos de su constructor son el nombre del entorno y si debe o no ejecutarse el kernel en modo debug.

Entorno

El entorno puede ser cualquier cadena de caracteres. Es principalmente una manera de determinar cuál archivo de configuracion será cargado (e.g. (e.g. `config_dev.yml` o `config_prod.yml`). Está explícito en `AppKernel`:

```
1  public function registerContainerConfiguration(LoaderInterface $loader)
2  {
3      $loader
4          ->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
5  }
```

Modo Depurado

En modo depurado tendremos:

- Una página de excepción bonita y mostrando detalles de errores, mostrando toda la información requerida para depurar los problemas.
- Mensajes de error en modo verbose en caso de que la página bonita no pueda mostrarse.
- Información elaborada sobre el tiempo requerido para ejecutar partes de la aplicación (inicialización, llamadas a base de datos, renderizado de las plantillas, etc.).
- Información extensiva sobre las peticiones (usando el web profiler y la barra de herramientas que le acompaña).
- Invalidación del caché automático: esto asegura que los cambios al `config.yml`, `routing.yml` y similares se tendrán en cuenta sin recompilar enteramente el contenedor de servicios o el matcher de las rutas por cada petición (lo cual podría tomar mucho tiempo).

Seguido de esto, el objeto de petición `Request` es creado basado en las variables superglobales de PHP (`$_GET`, `$_POST`, `$_COOKIE`, `$_FILES` and `$_SERVER`). La clase `Request` junto con otras clases del componente `HttpFoundation` proveen maneras orientada a objetos de envolver estas variables superglobales. Estas clases también cubren muchos casos excepcionales que puedas experimentar con diferentes versiones de PHP o en plataformas diferentes. Es sabio (en el contexto Symfony) siempre usar `Request` para obtener cualquier dato que podrías normalmente haber obtenido directamente de las variables superglobales.

Luego de esto se llama al método `handle()` de la instancia del `AppKernel`. Su único argumento es el objeto petición presente `Request`. Los argumentos por defecto para el tipo de petición (“master”) y

si se han de atrapar y manejar las excepciones (sí) se añadirán automáticamente.

El resultado del método handle() es garantizado ser una instancia de tipo Response (también parte del componente HttpFoundation). Finalmente la respuesta será enviada de vuelta al cliente que hizo la petición - por ejemplo un navegador.

1.1 Inicializando el kernel

De hecho, la magia sucede dentro del método handle() en el kernel. Encontrarás la implementación de este método en la clase Kernel, la cual es la clase padre de AppKernel:

```
1  // en Symfony\Component\HttpKernel\Kernel
2
3  public function handle(
4      Request $request,
5      $type = HttpKernelInterface::MASTER_REQUEST,
6      $catch = true
7  ) {
8      if (false === $this->booted) {
9          $this->boot();
10     }
11
12     return $this->getHttpKernel()->handle($request, $type, $catch);
13 }
```

Primero que todo, se asegura que el Kernel se inicializa, justo antes de que se le pida al HttpKernel que haga el resto. El proceso de inicialización incluye:

- inicialización de todos los bundles registrados
- inicialización del contenedor de servicios

Bundles como extensión del contenedor

Los Bundles son conocidos entre los desarrolladores Symfony como el lugar donde colocamos nuestro código a medida. Cada bundle debería tener un nombre que refleje qué tipo de cosas podrías hacer con el código que contiene. Por ejemplo podrías tener un BlogBundle, un CommunityBundle, un CommentBundle, etc. Registras tus bundles en AppKernel.php, añadiéndolos a la lista existente de bundles:

```php
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
            ...,
            new Matthias\BlogBundle()
        );

        return $bundles;
    }
}
```

Esto es definitivamente una buena idea - permite adherir funcionalidad o quitarla de tu proyecto con una simple linea de código. Sin embargo, cuando miramos al `Kernel` y como maneja todos los bundles, incluídos los tuyos, llega a ser evidente que los bundles son principalmente tratados como formas de extender el contenedor de servicios, no como bibliotecas de código. Este es el porqué encuentras un folder `DependencyInjection` dentro de muchos bundles, acompañados de una clase `{nameOfTheBundle}Extension`. Durante el proceso de inicialización del contenedor de servicios, a cada bundle se le permite registrar algunos servicios propios en el contenedor de servicios, quizá tambien algunos parámetros, y posiblemente modificar algunas definiciones de servicios antes de que el contenedor de servicios sea compilado y guardado en el directorio cache:

```php
namespace Matthias\BlogBundle\DependencyInjection;

use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;

class MatthiasBlogExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $loader = new XmlFileLoader($container,
            new FileLocator(__DIR__.'/../Resources/config'));

        // añade definiciones de servicios al contenedor
        $loader->load('services.xml');

        $processedConfig = $this->processConfiguration(
            new Configuration(),
```

```
            $configs
        );

        // establece un parámetro
        $container->setParameter(
            'matthias_blog.comments_enabled',
            $processedConfig['enable_comments']
        );
    }

    public function getAlias()
    {
        return 'matthias_blog';
    }
}
```

El nombre retornado por el método `getAlias()` de una extensión de contenedor es realmente clave y es bajo el cual tú puedes definir algunos valores de configuración `config.yml`:

```
matthias_blog:
    enable_comments: true
```

Leerás mas acerca de la configuración de bundles en Patrones de inyección de dependencias.

Cada clave de configuración corresponde a un bundle

En el ejemplo de arriba viste que `matthias_blog` es la clave de configuración para las definiciones relacionadas al bundle `MatthiasBlogBundle`. Ahora no sería de gran sorpresa que esto también sea válido para todas las claves que conoces de `config.yml` y similares: por ejemplo los valores bajo la clave `framework` están relacionados con el bundle `FrameworkBundle` y los valores bajo la clave `security` (a pesar de que éstos están definidos en un archivo separado llamado `security.yml`) están relacionados con el bundle `SecurityBundle`. ¡Tan simple como eso!

Creando el contenedor de servicios

Después de que todos los bundles hayan sido habilitados para añadir sus servicios y parámetros, el contenedor finaliza en un proceso que se llama "compilation". Durante este proceso es todavía posible hacer algunos ajustes de último minuto a las definiciones de servicios o sus parámetros. Es también el momento justo para validar y optimizar las definiciones de servicios. Después de esto, el contenedor se encuentra en su forma final, y se vuelca en dos diferentes formatos: en un archivo

XML con todas las definiciones resueltas y sus parámetros, y en un archivo PHP listo para ser usado como el único y verdadero contenedor de servicios en tu aplicación.

Ambos archivos pueden ser encontrados en el directorio cache correspondiente al entorno del kernel, por ejemplo `/app/cache/dev/appDevDebugProjectContainer.xml`. El archivo XML luce como cualquier archivo de definición de servicios en XML regular, sólo que mucho más grande:

```
<service id="event_dispatcher" class="...\ContainerAwareEventDispatcher">
  <argument type="service" id="service_container"/>
  <call method="addListenerService">
    <argument>kernel.controller</argument>
    ...
  </call>
  ...
</service>
```

El archivo PHP contiene un método para cada servicio que pueda ser requerido. Cualquier lógica de creación, como argumentos de controlador o llamadas a métodos después de la instanciación se pueden encontrar en este archivo, y es por lo tanto el lugar perfecto para depurar tus definiciones de servicios en caso de que algo parezca estar mal con ellos:

```
class appDevDebugProjectContainer extends Container
{
    ...

    protected function getEventDispatcherService()
    {
        $this->services['event_dispatcher'] =
            $instance = new ContainerAwareEventDispatcher($this);

        $instance->addListenerService('kernel.controller', ...);

        ...

        return $instance;
    }

    ...
}
```

1.2 Desde el `Kernel` al `HttpKernel`

Ahora que el kernel se ha inicializado (i.e. todos los bundles están inicializados, sus extensiones son registradas, y el contenedor de servicios ha sido finalizado), el verdadero manejo de la petición es

delegado a una instancia de HttpKernel:

```
// in Symfony\Component\HttpKernel\Kernel

public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    if (false === $this->booted) {
        $this->boot();
    }

    return $this->getHttpKernel()->handle($request, $type, $catch);
}
```

El HttpKernel implementa HttpKernelInterface y verdaderamente sabe como convertir una petición en una respuesta. El método handle() luce como sigue:

```
public function handle(
    Request $request,
    $type = HttpKernelInterface::MASTER_REQUEST,
    $catch = true
) {
    try {
        return $this->handleRaw($request, $type);
    } catch (\Exception $e) {
        if (false === $catch) {
            throw $e;
        }

        return $this->handleException($e, $request, $type);
    }
}
```

Como puedes ver, gran parte del trabajo se hará en el método privado handleRaw(), y el bloque try/catch está aqui para capturar cualquier excepción. Cuando el argumento inicial $catch es true (el cual es el valor por defecto para las peticiones de tipo “master”), toda excepción será tratada de buena manera. El HttpKernel intentará encontrar a alguien que pueda todavía crear un objeto Response decente de ahí en adelante (vease también Manejo de excepciones).

2 Eventos preparativos para la respuesta

El método `handleRaw()` del `HttpKernel` es una pieza hermosa de código, en el cual se muestra claramente que manejar una petición no es algo determinístico *per se*. Hay muchas varias maneras en las que puedes engancharte en el proceso y reemplazar completamente o sólamente modificar cualquier resultado intermedio.

2.1 Respuesta temprana

La primera oportunidad en la que puedes tomar el control de manejar la petición es justo al principio. Usualmente el `HttpKernel` intentará generar una respuesta ejecutando un controlador. Pero cualquier event listener que escuche el evento `KernelEvents::REQUEST` (`kernel.request`) tendrá permitido generar una respuesta completa totalmente a medida:

```
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

private function handleRaw(Request $request, $type = self::MASTER_REQUEST)
{
    $event = new GetResponseEvent($this, $request, $type);
    $this->dispatcher->dispatch(KernelEvents::REQUEST, $event);

    if ($event->hasResponse()) {
        return $this->filterResponse(
            $event->getResponse(),
            $request,
            $type
        );
    }

    ...
}
```

Como puedes ver, el objeto evento que se crea es una instancia de `GetResponseEvent` y le permite a los listeners asignar un objeto respuesta `Response` a medida usando su método `setResponse()`, por ejemplo:

```php
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class MaintenanceModeListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $response = new Response(
            'This site is temporarily unavailable',
            503
        );

        $event->setResponse($response);
    }
}
```

Registrando event listeners

El despachador de eventos usado por `HttpKernel` es el mismo que está también disponible como el servicio `event_dispatcher`. Cuando quieras automáticamente registrar alguna clase como event listener, puedes crear una definición de servicio para ésta y añadir el tag `kernel.event_listener` o `kernel.event_subscriber` (en el caso de que escojas implementar la interfaz `EventSubscriberInterface`).

```
<service id="..." class="...">
  <tag name="kernel.event_listener"
       event="kernel.request"
       method="onKernelRequest" />
</service>
```

Or:

```
<service id="..." class="...">
  <tag name="kernel.event_subscriber" />
</service>
```

Puedes opcionalmente dar a tu event listener una prioridad, para que tome precedencia sobre otros event listeners:

```
<service id="..." class="...">
  <tag name="kernel.event_listener"
       event="kernel.request"
       method="onKernelRequest"
       priority="100" />
</service>
```

Cuanto más alto el número, más temprano será notificado.

Algunos event listeners notables que escuchan el evento `kernel.request`

El framework mismo tiene muchos listeners para el evento `kernel.request`. Éstos son, la mayoría, listeners para poner a punto las cosas, antes de dejar que el kernel llame a cualquier controlador. Por ejemplo un listener se asegura de que la aplicación tenga un locale (ya sea el locale por defecto, o el `_locale` como parte de la URI), otro listener procesa las peticiones para fragmentos de páginas.

Hay, sin embargo, dos actores principales en la etapa temprana de una petición: El `RouterListener` y el `Firewall`. El `RouterListener` toma la información de ruta del objeto `Request` e intenta

relacionarla con alguna ruta conocida. Guarda el resultado del proceso de relación en el objeto `Request` como atributos, por ejemplo el nombre del controlador que corresponde a la ruta que fue encontrada:

```
namespace Symfony\Component\HttpKernel\EventListener;

class RouterListener implements EventSubscriberInterface
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $request = $event->getRequest();

        $parameters = $this->matcher->match($request->getPathInfo());

        ...

        $request->attributes->add($parameters);
    }
}
```

Cuando por ejemplo se le pregunta al matcher que relacione `/demo/hello/World`, y la configuración de ruta luce como sigue:

```
_demo_hello:
    path: /demo/hello/{name}
    defaults:
        _controller: AcmeDemoBundle:Demo:hello
```

Los parámetros retornados por la llamada a `match()` serán una combinación de valores definidos bajo la clave `defaults:` y los valores dinámicos encontrados por las etiquetas en la ruta (como `{name}`):

```
array(
  '_route' => '_demo_hello',
  '_controller' => 'AcmeDemoBundle:Demo:hello',
  'name' => 'World'
);
```

Éstas terminarán en la bolsa de parámetros de `Request` que se llama “attributes”. Como debes de suponer: `HttpKernel` examinará más adelante los atributos de la petición y ejecutará el controlador especificado.

Otro event listener importante es el `Firewall`. Como vimos antes, el `RouterListener` no provee al `HttpKernel` con un objeto `Response`, simplemente hace un poco de trabajo al principio de la petición. Al contrario, el `Firewall` a veces forzará un cierto objeto `Response`, por ejemplo cuando un usuario no está autenticado cuando debiera estarlo, dado que el usuario ha pedido una página protegida. El `Firewall` (a través de un complejo proceso) entonces fuerza un redireccionamiento a, por ejemplo, una página de login, o definirá algunas cabeceras, requiriendo así al usuario entrar sus credenciales usando autenticación HTTP.

II Patrones de Inyección de Dependencias

3 ¿Qué es un bundle?

Como vimos en el capítulo anterior: correr una aplicación Symfony significa inicializar el kernel y procesar una petición o ejecutar un comando, donde inicializar el kernel significa: cargar todos los bundles y registrar sus extensiones del contenedor de servicios (los cuales pueden ser encontrados en la carpeta `DependencyInjection` de un bundle). La extensión del contenedor usualmente carga un archivo `services.xml` (sin embargo este puede ser cualquier cosa) y la configuración del bundle, definida en una clase por separado, usualmente en el mismo namespace, llamado `Configuration`. Estas cosas juntas (*bundle*, *extensión del contenedor* y *configuración*) pueden ser utilizadas para implementar tu bundle: puedes definir parámetros y servicios de tal manera que la funcionalidad que provees dentro de tu bundle esté disponible a otras partes de la aplicación. Puedes también ir un paso más allá y registrar los llamados *compiler passes* para modificar adicionalmente el contenedor de servicios antes de que obtenga la forma final.

Después de crear muchos bundles, llegué a la conclusión de que mucho de mi trabajo como desarrollador, que es específicamente para aplicaciones Symfony, consiste en escribir código para exactamente estas partes: bundle, extensión y clases de configuración y compiler passes. Cuando sabes cómo escribir buen código, todavía necesitas aprender cómo crear buenos bundles, y esto básicamente significa que necesitas saber como crear buenas definiciones de servicios. Hay muchas maneras de hacer esto, y en este capítulo describiré la mayoría de ellas. Conocer tus opciones te permitirá tomar mejores decisiones cuando busques un patrón de inyección de dependencias.

No uses comandos generadores

Cuando comienzas a usar Symfony, te puedes sentir tentado a usar los comandos provistos por el `SensioGeneratorBundle` para generar bundles, controladores, entidades y tipos de formularios. No los recomiendo. Estas clases generadas pueden ser buenas como referencia para crear nuestras clases manualmente, pero en general contienen mucho código que no necesitaremos, o que no necesitamos para empezar. Puedes usar estos comandos generadores una vez, mirar cómo se deben hacer las cosas, y entonces aprender tú mismo a hacerlas igual en lugar de confiar en estos comandos. Esto te convertirá realmente en un rápido desarrollador, que entiende el framework bien.

4 Patrones de servicio

4.1 Patrón de estrategia para cargar servicios exclusivos

A menudo los bundles proveen multiples maneras de hacer una sola cosa. Por ejemplo, un bundle que proporciona cierto tipo de funcionalidad mailbox podría tener diferentes implementaciones de persistencia, tales como tener un storage manager para Doctrine ORM y uno para MongoDB. Para seleccionar un storage manager específico que sea configurable, deberías crear una clase `Configuration` como esta:

```
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('browser');

        $rootNode
            ->children()
                ->scalarNode('storage_manager')
                    ->validate()
                        ->ifNotInArray(array('doctrine_orm', 'mongo_db')
                        ->thenInvalid('Invalid storage manager')
                    ->end()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Entonces dado que hay dos archivos de definiciones de servicio uno por cada servicio storage manager, como por ejemplo `doctrine_orm.xml`:

```xml
<services>
  <service id="mailbox.doctrine_orm.storage_manager" class="...">
  </service>
</services>
```

Y `mongo_db.xml`:

```xml
<services>
  <service id="mailbox.mongo_db.storage_manager" class="...">
  </service>
</services>
```

Podrías entonces cargar cualquiera de estos archivos haciendo algo como esto en tu extensión de contenedor:

```php
class MailboxExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );

        // crea un XmlLoader
        $loader = ...;

        // carga solo los servicios para un storage manager dado
        $storageManager = $processedConfig['storage_manager'];
        $loader->load($storageManager.'.xml');

        // haciendo el storage manager específico disponible como el general
        $container->setAlias(
            'mailbox.storage_manager',
            'mailbox.'.$storageManager.'.storage_manager'
        );
    }
}
```

Una alias conveniente es creado al final para permitir que otras partes de la aplicación lo refieran simplemente como el servicio `mailbox.storage_manager`, en lugar de tratar de preocuparnos

por cuál servicio específico storage será usado. Sin embargo, la manera en que esta está hecha es un poco rígida: el id de cada servicio storage manager debería conformarse al patrón `mailbox.{storageManagerName}.storage_manager`. Será también mejor definir el alias dentro de los archivos de definición mismos:

```
<services>
  <service id="mailbox.doctrine_orm.storage_manager" class="...">
  </service>

  <service id="storage_manager"
    alias="mailbox.doctrine_orm.storage_manager">
  </service>
</services>
```

Usar el patrón de estrategia para cargar las definiciones de servicios tiene muchas ventajas:

- Sólamente se cargan los servicios que son útiles en la presente aplicación. Cuando no tengas el servidor MongoDB listo y corriendo no habrán servicios que accidentalmente lo refieran.
- El setup está abierto para ser extendido, ya que puedes añadir el nombre de otro storage manager a la lista de la clase `Configuration` y entonces añadir el archivo de configuración de servicio con los servicios y alias necesarios.

4.2 Cargando y configurando servicios adicionales

Digamos que tienes un bundle dedicado a filtrar las entradas. Probablemente estés ofreciendo diferentes servicios, como servicios para filtrar datos de formularios, y servicios para filtrar datos guardados usando Doctrine ORM. Debería entonces ser posible activar o desactivar cualquiera de estos servicios o colección de servicios en cualquier momento ya que no todos podrían ser aplicables a tu situación específica. Existe un atajo para definiciones de configuración que facilitan esto último:

```
class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('input_filter');

        $rootNode
            ->children()
                ->arrayNode('form_integration')
                    // será activado por defecto
```

```
                    ->canBeDisabled()
                ->end()
                ->arrayNode('doctrine_orm_integration')
                    // será desactivado por defecto
                    ->canBeEnabled()
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
}
```

Con un árbol de configuración como este, puedes activar o desactivar partes específicas del bundle en el archivo `config.yml`:

```
input_filter:
    form_integration:
        enabled: false
    doctrine_orm_integration:
        enabled: true
```

Dentro de tu extensión de contenedor puedes cargar los servicios apropiados:

```
class InputFilterExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $processedConfig = $this->processConfiguration(
            new Configuration(),
            $configs
        );

        if ($processedConfig['doctrine_orm_integration']['enabled']) {
            $this->loadDoctrineORMIntegration(
                $container,
                $processedConfig['doctrine_orm_integration']
            );
        }

        if ($processedConfig['form_integration']['enabled']) {
            $this->loadFormIntegration(
```

```
19                  $container,
20                  $processedConfig['form_integration']
21              );
22          }
23
24          ...
25      }
26
27      private function loadDoctrineORMIntegration(
28          ContainerBuilder $container,
29          array $configuration
30      ) {
31          // carga los servicios, etc.
32          ...
33      }
34
35      private function loadFormIntegration(
36          ContainerBuilder $container,
37          array $configuration
38      ) {
39          ...
40      }
41  }
```

Así cada una de las partes independientes del bundle pueden ser cargadas.

III Ser un programador Symfony

Muchos de los programadores que conozco se llaman “programadores PHP”. Algunos de ellos pueden incluso decir que son “programadores Symfony”. Otros dirán, “soy simplemente un programador”, sin revelar su lenguaje de programción favorito o el único lenguaje de programación que ellos conocen.

Cuando empecé a trabajar con Symfony, casi immediatamente me enamoré de este gran framework. Su primera versión era mucho mejor de lo que estaba acostumbrado. Pero la segunda versión, fue simplemente lo mejor para mi mente. Empecé a aprender mucho sobre el framework, profundizando en su código, escribiendo documentación para partes que estaban sin documentar, escribiendo artículos sobre cómo hacer ciertas cosas con Symfony, y a hablar en público sobre Symfony y sus librerías relacionadas de PHP.

Después de todo esto, me considero a mí mismo un programador Symfony. Pero la mejor parte de la historia es: que todo lo que he aprendido desde que trabajo con Symfony es generalmente aplicable al software escrito “para” cualquier otro framework en PHP. Incluso cuando trabajo con proyectos que no son Symfony, o una aplicación “antigua” en PHP (con código no reutilizable), aún así merece la pena pensar las formas en las que puedo usar código del “ecosistema de Symfony”, o de hecho, de cualquier librería que puedo agregar usando Composer, para hacer una mejor aplicación.

En esta parte demostraré que ser un buen programador Symfony es acerca de conocer bien el framework, pero también acerca de escribir código que sea beneficioso para cualquier proyecto en PHP, y terminando con una fina capa entre este código y el framework Symfony, para que la mayoría de tu código sea reutilizable incluso si es utilizado en un proyecto que no sea en Symfony.

5 El código reutilizable tiene bajo acoplamiento

5.1 Separa el código de tu empresa el código del producto

Tu situación más problable como programador Symfony es:

- Estás trabajando para una empresa.
- Tienes clientes (internos o externos) para quienes creas aplicaciones web.

Cuando empiezas a trabajar en una nueva aplicación, pondrás tu nuevo código en el directorio `/src`. Pero antes de que empieces, añade dos directorios dentro de este `/src`: `/src/NombreDeTuEmpresa` y `/src/NombreDelProducto`. De hecho, estos nombres de directorios deberán también reflejarse en los namespaces de las clases que crearás para tu proyecto.

Siempre que empieces a trabajar en una nueva funcionalidad para la aplicación - piensa en qué parte podrías reutilizar en *teoría*, y que parte es única de la aplicación en la que estás trabajando. Incluso si este código reutilizable *nunca* sea usado en la práctica, su calidad te beneficiará debido a esta forma de pensar. Empieza escribiendo clases en el namespace de la empresa. Sólamente cuando realmente sientas que necesitas usar algo específico al proyecto en curso, entonces debes moverlo al namespace de producto, o usar algún tipo de principio de extensión (una subclase, configuración, event listeners, etc.).

Escribir código reutilizable para tu empresa no significa que será open-source. Simplement significa que debes escribirlo *como si fuera* a ser open-source. No necesitas crear proyectos colaterales para todo el código de la empresa immediatamente. Puedes desarrollar este código dentro del proyecto en el que estás trabajando actualmente, y quizá más adelante prepararlo para reutilizarlo en otro proyecto. Lee más sobre el lado práctico de esto en Manejo de dependencias y control de versiones

Acoplamiento entre el código de la empresa y el código del producto

Cuando sigues una estricta separación entre código de empresa y producto sigue estas pautas que te ayudarán a hacer la separación de código más útil a tí mismo (cuando no sigues estas pautas, no tiene sentido mantener namespaces separados):

1. El *código de la empresa* puede saber sobre o depender de otro *código de la empresa.*
2. El *código de la empresa no* debe saber sobre o depender del *código específico del producto.*
3. El *código específico al producto* puede saber o depender del *código específico al producto*

Las primeras dos reglas deben ser tomadas en cuenta estrictamente, sólamente cuando haces esto el código será reutilizable o estará preparado para ser código abierto. La tercera regla al contrario *añade* mucha libertad, ya que por definición no será reutilizable ningún código específico al producto que esté acoplado a otro código específico al producto.

5.2 Separa el código de las librerías y el código de los bundles

Cuando notes que estas escribiendo clases e interfaces que no tienen relación con el framework entero de Symfony, o sólamente con algunas partes del framework, deberás separar tu código en código de "librería" y de "bundle". El código de librería es la parte de código que es más o menos autónoma (aunque puede tener algunas dependencias externas). El código de librería puede ser reutilizado por un programador que trabaje con el framework Zend, o con el micro-framework de Silex (para nombrar sólo algunos ejemplos). El código de un bundle reemplaza o extiende algunas clases del código de librería, añadiendo características adicionales y específicas a Symfony. Este define la configuración del bundle, y contiene definiciones de servicios, para crear instancias de las clases propias de la librería que estarán disponibles en la aplicación. Un bundle, de esta forma, pone a disposición el código de la librería, requiriendo un mínimo esfuerzo por parte del programador que quiere usarla dentro de su proyecto Symfony.

Estas son algunas cosas que están dentro de un bundle:

- Controladores
- Extensiones de contenedor
- Definiciones de servicios
- Compiler passes
- Event subscribers
- Clases container-aware que extiendan clases más genéricas

- Tipos de formularios
- Configuración de rutas
- Otra metainformación
- ...

La lista puede ser más larga. Pero también puede ser mucho más corta. Si lo piensas, sólamente el primer par de cosas de la lista son realmente específicos de los bundles (i.e. solamente usadas en el contexto de un proyecto Symfony estándar). El resto de cosas, pueden ser usadas en proyectos que solo hacen uso de componentes específicos de Symfony. Por ejemplo, los tipos de formularios, podrían ser usados en cualquier proyecto PHP con el Form Component instalado.

Ejemplos de códigos de librería y de bundle

Hay muchos ejemplos de esta separación de código de bundle y de librería. Cuando observas el código de Symfony, puedes ver claramente: que el directorio `Component` contiene todos los componentes Symfony (el código de "librería"), y el directorio `Bundle` contiene los bundles que integran todas las clases juntas, y proveen opciones de configuración que son pasadas como argumentos de constructor a todas estas clases. Podemos encontrar ejemplos excelentes de esta estratégia en el `FrameworkBundle` (el bundle "core" que: cuando está presente en un proyecto, llamamos propiamente al proyecto un "proyecto Symfony").

5.3 Reduciendo el acoplamiento al framework

En cuanto a desacoplar tu código de un componente Symfony2 como dependencia, o del framework Symfony2, puedes ir tan lejos como quieras, y de esta manera hacer que tu código sea más reutilizable.

Event listeners en lugar de event subscribers

Por ejemplo, deberías preferir event listeners en lugar de event subscribers. Los event subscribers son un tipo de event listeners pero especiales, que implementan la interfaz `EventSubscriberInterface`:

```php
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class SendConfirmationMailListener implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            AccountEvents::NEW_ACCOUNT_CREATED => 'onNewAccount'
        );
    }

    public function onNewAccount(AccountEvent $event)
    {
        ...
    }
}
```

Puedes registrar un event subscriber de forma limpia usando el tag de servicio `kernel.event_subscriber` así:

```xml
<service id="send_confirmation_mail_listener"
  class="SendConfirmationMailListener">
  <tag name="kernel.event_subscriber" />
</service>
```

Hay algunos problemas con este enfoque:

1. Un event subscriber es totalmente inútil cuando el componente EventDispatcher no está disponible, aunque no haya nada específico de Symfony en este event listener.
2. El método `onNewAccount()`, recibe un objeto `AccountEvent`, pero no se dice en ninguna parte que el objeto puede sólamente provenir de un evento con el nombre de `AccountEvents::NEW_ACCOUNT_CREATED`.

Por lo tanto, un event listener que es realmente un *event subscriber* no es tan reutilizable. Acopla nuestro código con el componente EventDispatcher. Es mejor eliminar la interfaz, eliminar el método requerido por la interfaz, y registrar los métodos del listener manualmente usando el tag de servicio `kernel.event_listener`:

```
<service id="send_confirmation_mail_listener"
  class="SendConfirmationMailListener">
  <tag name="kernel.event_listener"
    event="new_account_created" method="onNewAccount" />
</service>
```