Runroom
Digital crafts

Symfony & Drupal 8

Joel Rodríguez Alemán (@joelrguezaleman)
Marc Mauri Alloza (@opengeek)

Introducción y conceptos

Qué es Symfony?

Framework web

PHP

Opensource

Estructura

Buenas prácticas

Comunidad

Documentación

Entornos

Una aplicación en Symfony puede ejecutarse en diferentes entornos.

- dev -> http://localhost/app_dev.php/hello-world
- prod -> http://localhost/app.php/hello-world
- test -> Tests automatizados

Los entornos ejecutan el mismo código, pero tienen diferente configuración (logs, cache...).

Ficheros de configuración

- Genérico: config.yml
- dev: config_dev.yml
- prod: config_prod.yml

Posibilidad de crear nuevos entornos:

- http://symfony.com/doc/current/cookbook/configuration/environments.html

Bundles

Un Bundle es como un plugin de Symfony con la diferencia de que en Symfony todo, incluso el propio framework es un Bundle.

Un Bundle se crea situando en un directorio una clase que extienda de Bundle y cuyo nombre siga la convención.

En *app/AppKernel->registerBundles()* se instancian los Bundles en uso.

Cada Bundle sigue la siguiente convención de directorios:

- **Controller/:** Contiene los controladores
- **Resources/config/:** Contiene la configuración de rutas y servicios
- **Resources/views/:** Contiene las vistas
- **Resources/public/:** Contiene imágenes, estilos, ficheros javascript y otros assets
- **Tests/:** Contiene las pruebas del Bundle

http://symfony.com/doc/current/book/bundles.html

Comandos

Console component: https://github.com/symfony/console

Comandos

- Crear bundle: php app/console generate:bundle
- Borrar cache: php app/console cache:clear
- Comandos custom: http://symfony.com/doc/2.8/cookbook/console/console_command.html

Por defecto, los comandos se ejecutan en el entorno dev. Para ejecutar en prod:

- php app/console cache:clear --env=prod

MVC

Es un patrón de diseño que promueve la separación entre el código relacionado con los conceptos que maneja aplicación (modelo), su apariencia (vista) y el control de la aplicación (controlador).

Symfony ofrece abstracciones para los 3 elementos.

- El modelo está formado por clases PHP convencionales en la carpeta Model del Bundle. Los controladores manipulan el modelo a través de sus métodos y le pasan los elementos del modelo a las vistas
- Las vistas están formadas por plantillas Twig que residen en la carpeta Resources/view del Bundle. Pueden acceder a las propiedades de los modelos.
- Los controladores son llamados cuando se realiza un petición a una ruta concreta, su responsabilidad es devolver una respuesta a la petición que puede ser o una vista renderizada o simplemente datos en algún formato. Los controladores residen en la carpeta Controller del Bundle.

Componentes de Symfony en Drupal 8

YAML - Formato

YAML es un lenguaje usado para serializar datos que en Symfony se utiliza en ficheros de configuración y de contenido (por ejemplo traducciones).

El formato YAML tiene como objetivo ser capaz de serializar estructuras de datos arbitrarias y de mantenerse su fàcil lectura al usuario.

```
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
    - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
tax  : 251.42
```

YAML - Ficheros importantes

- app/config.yml
  - Es el fichero de configuración principal y importa generalmente al a los demás ficheros.
- parameters.yml
  - Es una buena práctica usarlo para almacenar variables que pueden cambiar.
- security.yml
  - Configuración de autenticación y autorización.
- routing.yml
  - En este fichero se describen las rutas de la aplicación. Puede haber uno en app y otro en cada bundle, tambien puede importar otros ficheros.
- services.yml
  - En este fichero se configura el contenedor de servicios: Puede haber uno en app y otro en cada Bundle.

Routing + Controllers

Anotaciones

```
class BlogController extends Controller
{
    /**
     * @Route("/blog/{slug}", name="blog_show")
     */
    public function showAction($slug)
    {
        // ...
    }
}
```

Routing + Controllers

Yaml

```yaml
# app/config/routing.yml
blog_show:
    path:      /blog/{slug}
    defaults: { _controller: AppBundle:Blog:show }
```

Routing + Controllers

Xml

```xml
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing
        http://symfony.com/schema/routing/routing-1.0.xsd">

    <route id="blog_show" path="/blog/{slug}">
        <default key="_controller">AppBundle:Blog:show</default>
    </route>
</routes>
```

Routing + Controllers

PHP

```php
$collection = new RouteCollection();
$collection->add('blog_show', new Route('/blog/{slug}', array(
    '_controller' => 'AppBundle:Blog:show',
)));

return $collection;
```

Routing + Controllers

Symfony carga todas las rutas desde el fichero **app/config/routing.yml**

Se pueden importar rutas de otros ficheros:

```yaml
app:
    resource: '@AppBundle/Controller/'
    type:     annotation

# app/config/routing.yml
app:
    resource: '@AcmeOtherBundle/Resources/config/routing.yml'
    prefix:   /site
```

Routing + Controllers

Opciones:

```
# app/config/routing.yml
blog:
    path:      /blog/{page}
    defaults: { _controller: AppBundle:Blog:index, page: 1 }
    methods: [GET, POST]
    requirements:
        page:  \d+
```

Routing + Controllers

Ejemplo de controlador:

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;


class HelloController // extends Controller
{
    public function helloAction($name, Request $request)
    {
        $request->query->get('page'); // $_GET parameter
        return new Response('Hello ' . $name);
    }
}
```

Twig

Motor de plantillas escrito en PHP

Características:

- Extensión de plantillas
- Variables
- Lógica
- Filtros y funciones
- Extensiones

Twig

Extensión de plantillas:

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


{% block body_content %}
    {{ parent() }}
     <!-- Index content -->
{% endblock %}
```

Twig

Variables, lógica, filtros y funciones:

```
{{ "link"|trans }}
{{ date() }}

{% if url is empty %}
    <a href="#" class="class-empty">Empty link</a>
{% else %}
    <a href="{{ url }}" class="class-not-empty">Non-empty link</a>
{% endif %}

<ul class="menu">
    {% for item in menu %}
        <li>{{ item.name }}</li>
    {% endfor %}
</ul>
```

Twig

Pasar variables a las plantillas:

```
class HelloController extends Controller
{
    public function indexAction()
    {
        return $this->render(
            'hello.html.twig',
            array('name' => 'Peter')
        );
    }
}
```

Event Dispatcher

Alternativa a los hooks

Implementa el Mediator Pattern

- El dispatcher recibe peticiones de objetos que quieren ser avisados cuando se produzca un determinado evento.
- Cuando un objeto quiere lanzar un evento, crea un objeto que contiene el evento y se lo pasa al dispatcher.
- El dispatcher lanza el evento y ejecuta el método de los listeners que están asociados al evento lanzado.

Event Dispatcher

Un objeto PHP le dice al Dispatcher (el objeto principal que actúa de mediador) que quiere estar atento al evento kernel.response:

```
use Symfony\Component\EventDispatcher\EventDispatcher;
$dispatcher = new EventDispatcher();

$listener = new AcmeListener();
$dispatcher->addListener(
    'kernel.response', array($listener, 'onFooAction')
);
```

Event Dispatcher

El kernel de Symfony le dice al Dispatcher que lance el evento kernel.response, y le pasa un objeto que contiene el evento y que tiene acceso al objeto que representa la response.

```
$dispatcher->dispatch('kernel.response', $event);
```

Event Dispatcher

El dispatcher lanza el evento y notifica a todos los listeners de ese objeto (invoca al método correspondiente de cada listener)

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


class AcmeListener
{
    public function onFooAction(Event $event)
    {
        // ... do something
    }
}
```

Event Dispatcher

Podemos crear nuestros propios eventos:

```php
use Symfony\Component\EventDispatcher\Event;
use Acme\StoreBundle\Order;

class FilterOrderEvent extends Event
{
    protected $order;

    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    public function getOrder()
    {
        return $this->order;
    }
}
```

Event Dispatcher

Usar EventSubscribers en vez de listeners (I)

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

class StoreSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array(
            'kernel.response' => array(
                array('onKernelResponsePre', 10),
                array('onKernelResponsePost', 0),
            ),
            'store.order'     => array('onStoreOrder', 0),
        );
    }
    //  Definir métodos onKernelResponsePre, onKernelResponsePost y onStoreOrder
}
```

Event Dispatcher

Usar EventSubscribers en vez de listeners (II)

```
use Acme\StoreBundle\Event\StoreSubscriber;


$subscriber = new StoreSubscriber();
$dispatcher->addSubscriber($subscriber);
```

PSR-4

Symfony carga las classes de forma automática si están en la carpeta src si se siguen las convenciones del PSR-4, se pueden usar directamente prefijando la clase con el namespace o bien usando el operador "use".

Para importar una clase:

```
use Symfony\Component\HttpFoundation\Request;
```

Para definir código en un namespace:

```
namespace Acme\BlogBundle\NombreNamespace;
```

No hace falta importar las clases del mismo namespace en el que estamos.

PSR-4

El responsable de la carga automática es el fichero autoload.php de composer.

Cuando usando Composer instalamos dependencias en la carpeta vendors registra las dependencias como namespaces con un prefijo y lo mapea a la carpeta donde estan las clases de la dependencia, por ejemplo.

```php
// vendor/composer/autoload_namespaces.php
return array(
    'Symfony\\' => array($vendorDir . '/symfony/symfony/src'),
    'Doctrine\\ORM' => $vendorDir . '/doctrine/orm/lib/',
    'Doctrine\\DBAL' => $vendorDir . '/doctrine/dbal/lib/',
    'Doctrine\\Common\\DataFixtures' => $vendorDir . '/doctrine/data-fixtures/lib/'
);
```

Inyección de dependencias

El uso de Symfony en Drupal 8 nos brinda la oportunidad de aplicar el patrón de diseño de la inyección de dependencias.

Este patrón se aplica para evitar que las clases contengan cantidades importantes de código dedicadas a conseguir instancias de las clases de las que depende.

Para conseguirlo delega a una clase externa (el Inyector) la responsabilidad de pasar a la clase sus dependencias ya instanciadas (Services).

Otra ventaja de este patrón es que podemos instanciar la misma clase con dependencias distintas cambiando la configuración o teniendo varias configuraciones mientras todas las dependencias cumplan la interfaz que la clase en la que son inyectadas espera.

Inyección de dependencias - Tipos

	Ventajas	Inconvenientes
Constructor	● Asegura que la dependencia está satisfecha al instanciar la clase. ● Asegura que nadie cambiara la dependencia en tiempo de ejecución. ● Se puede usar "type hinting" para comprobar que la dependencia pasada es del tipo correcto.	● No sirve para dependencias opcionales. ● No se puede extender de una clase que usa inyección por constructor, sobreescribir el constructor y cambiar la dependencia sin modificar su configuracion.
Setter	● Funciona bien para dependencias opcionales. ● Funciona bien para añadir varias dependencias en un array ya que se puede llamar varias veces. ● Se puede usar "type hinting" para comprobar que la dependencia pasada es del tipo correcto.	● La clase puede ser creada sin la dependencia. ● Se puede cambiar la dependencia en tiempo de ejecución si no se escribe un mecanismo de control en el setter.
Propiedad	● Útil para bibliotecas de terceros en que las dependencias están en atributos públicos.	● La des setter con el agravante de no poder escribir un mecanismo de control. ● No se puede usar "type hinting".

Inyección de dependencias - Servicio

Queremos ofrecer la clase mailer como servicio a otras clases.

```php
// vendor/composer/autoload_namespaces.php
class Mailer
{
    private $transport;

    public function __construct($transport)
    {
        $this->transport = $transport;
    }

    // ...
}
```

Inyección de dependencias - Registrar el Servicio

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->setParameter('mailer.transport', 'sendmail');
$container
    ->register('mailer', 'Mailer')
    ->addArgument('%mailer.transport%');
```

Para que esté disponible hay que registrar la clase en el contenedor. Además configuramos el tipo de transporte también aquí en lugar de meterlo directamente en la clase, de esta forma es más cambiable. Esto usará siempre la misma instancia de Mailer.

Inyección de dependencias - Consumir el Servicio

```
class NewsletterManager
{
    private $mailer;

    public function setMailer(\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Tenemos otra clase en nuestro sistema que depende de la clase mailer.

Inyección de dependencias - Consumir el Servicio

```
use Symfony\Component\DependencyInjection\ContainerBuilder;

$container = new ContainerBuilder();
$container->setParameter('mailer.transport', 'sendmail');
$container
    ->register('mailer', 'Mailer')
    ->addArgument('%mailer.transport%');

$container
    ->register('newsletter_manager', 'NewsletterManager')
    ->addMethodCall('setMailer" ,array(new Reference('mailer')));
```

Para que el framework pueda inyectar la dependencia hay que registrar la clase y definir la dependencia.

Inyección de dependencias - Configuración

Usar PHP para configurar el contenedor puede ser engorroso por eso Symfony ofrece la posibilidad de usar ficheros de configuración XML o YAML.

```yaml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }
    - { resource: services.yml }
    - { resource: "@NoSharedBundle/Resources/config/services.yml" }
```

Para Bundles propios de la aplicación y que no hay que reusar en otra aplicación se importa el fichero de configuración del Bundle des del fichero en app/config/config.yml

Inyección de dependencias - Configuración

```
class AcmeDemoExtension extends Extension {

    public function load(array $configs, ContainerBuilder $container) {
        $configuration = new Configuration();
        $config = $this->processConfiguration($configuration, $configs);
        $loader = new Loader\YamlFileLoader($container, new FileLocator
(__DIR__.'/../Resources/config'));
        $loader->load('services.yml');
    }

}
```

Para Bundles que deben ser reusables existe un mecanismo para que la configuración del Bundle se cargue dinámicamente sin necesidad de importarla explícitamente. El comando generate Bundle genera todo el código necesario si se le dice que el Bundle va a ser compartido.

Inyección de dependencias - YAML

```yaml
parameters:
    # ...
    mailer.transport: sendmail

services:
    mailer:
        class:     Mailer
        arguments: ['%mailer.transport%']
    newsletter_manager:
        class:     NewsletterManager
        calls:
- [setMailer, ["@mailer"]]
```

Aquí podemos observar como la misma configuración que hemos hecho en PHP también se puede escribir en YAML.

Process

Ejecuta comandos en subprocesos

Github: https://github.com/symfony/Process

Drupal hace uso de este componente para gestionar todas aquellas actividades que por naturaleza se realicen desde la consola de comandos

```
use Symfony\Component\Process\Process;


$process = new Process('ls -lsa');
$process->setTimeout(3600);
$process->mustRun();


print $process->getOutput();
```

Validación - Anotaciones

Symfony ofrece un servicio de validación que valida distintas restricciones que podemos definir en nuestras clases de forma que no tenemos que dedicar código a ello.

```php
// src/AppBundle/Entity/Author.php
// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Validación - Configuración

```yaml
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

Para que las anotaciones funcionen hay que activarlas explícitamente.

Validación - YAML

```yaml
# src/AppBundle/Resources/config/validation.yml
AppBundle\Entity\Author:
    properties:
        name:
            - NotBlank: ~
```

También se puede configurar en YAML.

Validación - Restricciones

Existen todo tipo de restricciones a nivel de propiedad, getter o clase:

- Básicas: *Blank, NotBlank, NotNull, IsNull, IsTrue, IsFalse, Type*
- Cadenas: *Email*, *Length*, *Url*, *Regex*, *Ip, Uuid*
- Números: *Range*
- Comparación: *EqualTo*, *NotEqualTo*, *IdenticalTo*, *NotIdenticalTo, LessThan, LessThanOrEqual, GreaterThan*, *GreaterThanOrEqual*
- Fecha: *Date, DateTime*, *Time*
- Colección: *Choice, Collection, Count, UniqueEntity, Language, Locale, Country*
- Archivo: *File*, *Image*
- Finanzas y Códigos de Producto: *Bic, CardScheme, Currency, Luhn, Iban, Isbn, Issn*
- Otras: *Callback, Expression, All, UserPassword, Valid*

También se pueden crear restricciones personalizadas.

Validación - Validador

El servicio de validación tiene un método *validate* que devuelve un array con los errores si los hay.

```
public function authorAction()
{
    $author = new Author();

    $errors = $this->validator->validate($author);

    if (count($errors) > 0) return new Response((string) $errors);

    return new Response('The author is valid! Yes!');
}
```

Validación - Formularios

Los formularios de Symfony tienen un método isValid() que valida que el POST del form cumple las restricciones del objeto que hay que poblar.

```
public function updateAction(Request $request)
{
        $author = new Author();
        $form = $this->createForm(AuthorType::class, $author);
    $form->handleRequest($request);

        if ($form->isValid()) {
            // the validation passed, do something with the $author object
        }
}
```

Se pueden definir grupos de restricciones a validar y el orden en que estos grupos se tienen que validar.

Translation

Componente de traducciones de Symfony.

Configuración:

```
# app/config/config.yml
framework:
    translator: { fallbacks: [en] }
```

Translation

Ficheros de traducciones:

- Nombre: messages.es.yml
- Lugares en orden de prioridad:
  - app/Resources/translations
  - app/Resources/<bundle name>/translations
  - Resources/translations/

Hay que borrar la caché después de añadir nuevas strings

- php app/console cache:clear

Translation

Leer traducciones:

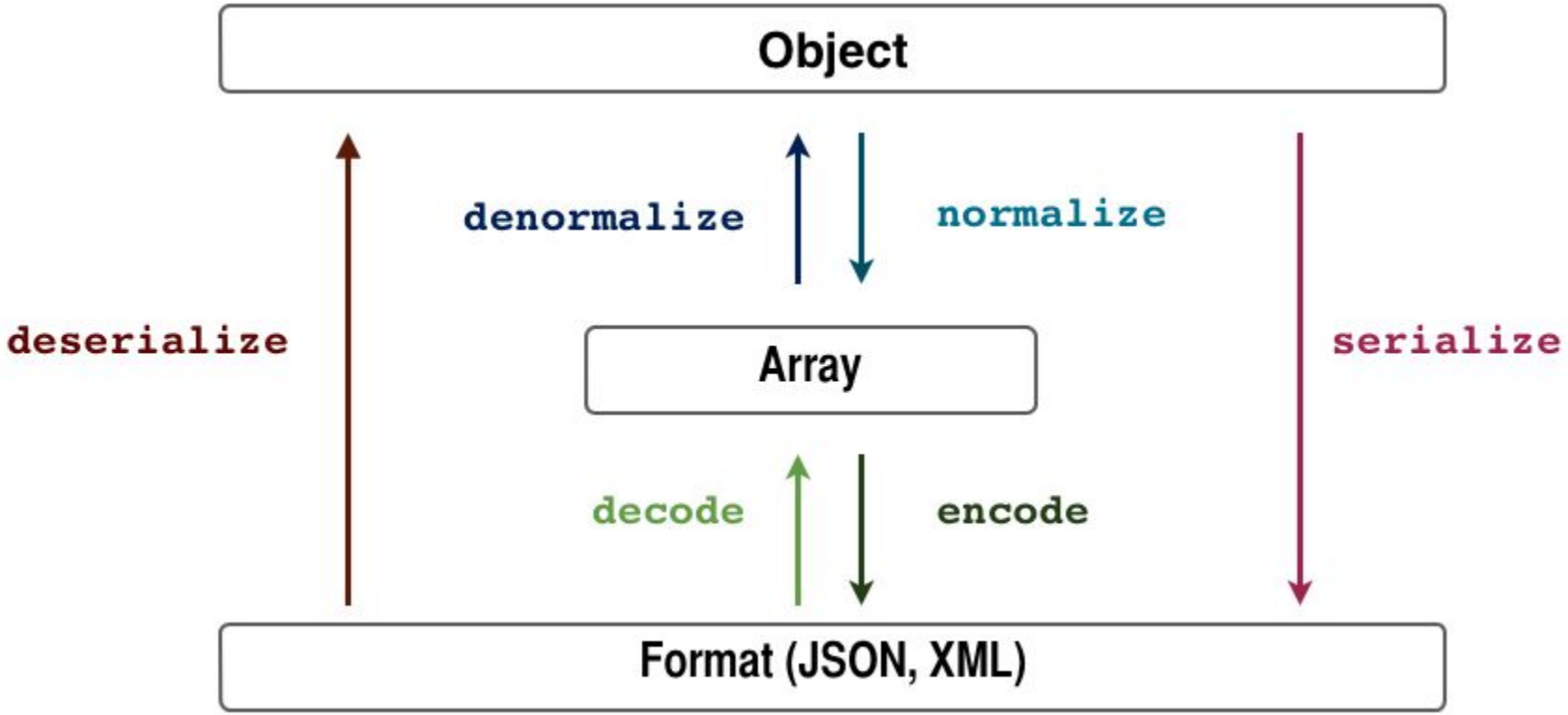
Desde Twig:

```
{{ 'string.key'|trans }}
```

Desde PHP:

```
$translated = $this->get('translator')->trans(
    'Hello ' . $name
);
```

Serializer - Introducción



Serializer - Configuración

Es responsabilidad del serializer transformar los objetos en formatos de representación de datos y a la inversa. Se realiza en dos pasos:

1. Normalización (Normalize): Transformar el objeto en un array.
2. Codificación (Encode): Transforma el array en el formato de representación deseado.

```
$encoders = array(new XmlEncoder(), new JsonEncoder());
$normalizers = array(new ObjectNormalizer());

$serializer = new Serializer($normalizers, $encoders);
```

En Drupal 8 ya disponéis de un Serializer configurado listo para ser inyectado.
En Symfony hay que crearlo e inyectarlo.

Serializer - Uso

Para serializar el primer parámetro es el objeto a serializar y el segundo el encoder que queremos usar.

```
$jsonContent = $serializer->serialize($person, 'json');
```

Para deserializar el primer parámetro es la cadena a deserializar, la segunda la clase con la que hay que instanciar el contenido y la tercera el formato de la cadena

```
$person = $serializer->deserialize($data, 'Acme\Person', 'xml');
```

Serializer - Avanzado

En algunos casos querremos serializar información distinta del mismo objeto, por ese motivo Symfony nos permite definir grupos de atributos que serializar mediante anotaciones.

```
/**
 * @Groups({"group1", "group2"})
 */
 public $foo;
```

En este caso hay que llamar primero al normalizer que extraerá un array con los datos de los grupos seleccionados y luego llamar al metodo encode.

```
$data = $serializer->normalize($obj, null, array('groups' => array('group1')));
```

Pausa

Componentes de terceros en Drupal 8

Guzzle - Cliente para ejecutar peticiones HTTP

```php
$client = new GuzzleHttp\Client();
$res = $client->request(
    'GET',
    'https://api.github.com/user',
    ['auth' => ['user', 'pass']]
);
echo $res->getStatusCode();
// "200"
echo $res->getHeader('content-type');
// 'application/json; charset=utf8'
echo $res->getBody();
// {"type":"User"...'
```

PSR3 Logging

Se adopta el estándar PSR-3 para tareas de logging.

- Objetivo: una forma universal y sencilla de loguear incidencias.

Watchdog queda obsoleto, ahora se usa:

```
\Drupal::logger($type)->log($severity, $message, $variables);
```

Se ha eliminado el hook_watchdog, ahora se ha de registrar un servicio:

```
services:
    logger.mylog:
        class: Drupal\mylog\Logger\MyLog
        tags:
            - { name: logger }
```

Composer

Herramienta para gestionar las dependencias de nuestro proyecto

Se distribuye en un binario: composer.phar

Instalación:

```
curl -s https://getcomposer.org/installer | php
```

Dos ficheros:

- composer.json: contiene las dependencias de nuestro proyecto
- composer.lock: establece las versiones que son usadas

Estos ficheros suelen incluirse en el repositorio del proyecto: en el momento de instalar las dependencias, se usan las que especifica el composer.lock.

Composer - Fichero composer.json

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

Normalmente, sólo necesitaremos modificar la key “require”

- Especificamos todas las dependencias
- Indicamos la versión máxima que estamos dispuestos a instalar de cada una

Composer - Comandos

```
php composer.phar init

php composer.phar install

php composer.phar update

php composer.phar require

php composer.phar self-update
```

Testing - PHPUnit

Es un framework para la realización de tests unitarios basado en SUnit, el framework de test unitario que Kent Beck creó para Smalltalk.

```php
//  Tests\BundleName should replicate the bundle structure
class MoneyTest extends PHPUnit_Framework_TestCase {
    public function setUp() { //  .. }
    public function testCanBeNegated() {
        $a = new Money(1);                          //  Arrange
        $b = $a->negate();                          //  Act
        $this->assertEquals(-1, $b->getAmount()); //  Assert
    }
    public function tearDown() { //  .. }
}
```

Testing - Runner

Existen distintas opciones en el comando para filtrar los test deseados.

```
# run all tests of the application
$ phpunit
# run all tests in the Util directory
$ phpunit tests/AppBundle/Util
# run tests for the Calculator class
$ phpunit tests/AppBundle/Util/CalculatorTest.php
# run all tests for the entire Bundle
$ phpunit tests/AppBundle/
```

Testing - Mocks con Prophecy

A veces nos interesa probar unitariamente clases que tienen dependencias. Para lograrlo creamos una instancia de esa clase y le inyectamos una falsa dependencia que hemos programado para que dé un resultado conocido. Esta falsa dependencia se llama mock. La biblioteca Prophecy ayuda a implementar mocks.

```php
public function testPasswordHashing() {
    $prophet = new \Prophecy\Prophet;
    $hasher = $prophet->prophesize('App\Security\Hasher');
    $user   = new App\Entity\User($hasher->reveal());
    $hasher->generate($user, 'qwerty')->willReturn('hashed_pass');
    $user->setPassword('qwerty');
    $this->assertEquals('hashed_pass', $user->getPassword());
}
```

Assetic

Asset management framework for PHP

- Assets: ficheros estáticos (imágenes, JS, CSS...)
- Filtros: operaciones que se pueden aplicar a esos ficheros
  - Compiladores (LESS, SASS, CoffeeScript...)
  - Minificadores
  - Procesamiento de imágenes

Assetic - Incluir ficheros

```
{% javascripts '@AppBundle/Resources/public/js/*' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}

{% stylesheets 'bundles/app/css/*' filter='cssrewrite' %}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}

{% image '@AppBundle/Resources/public/images/example.jpg' %}
    <img src="{{ asset_url }}" alt="Example" />
{% endimage %}
```

Assetic - Definir y usar filtros

```
# app/config/config.yml
assetic:
    filters:
        uglifyjs2:
            bin: /usr/local/bin/uglifyjs

<!-- Twig template -->
{% javascripts '@AppBundle/Resources/public/js/*'
filter='uglifyjs2' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Assetic

En el entorno de desarrollo, los ficheros compilados no existen físicamente: son renderizados y servidos directamente por Symfony

- Ventaja: los cambios se ven inmediatamente.
- Desventaja: lentitud.

Desactivar:

```
# app/config/config_dev.yml
assetic:
    use_controller: false


# Comandos
php app/console assetic:dump
php app/console assetic:watch
```

Assetic

En el entorno de producción, los ficheros JS y CSS se encuentran en un único fichero comprimido:

```
<script src="/js/abcd123.js"></script>
```

Este fichero ni existe ni es generado automáticamente. Debe ser generado con el siguiente comando:

```
php app/console assetic:dump --env=prod --no-debug
```

Hay que ejecutar este comando cada vez que se actualicen los assets.

Doctrine

ORM = Object Relational Mapping de tipo Data Mapper escrito en PHP

```
# app/config/config.yml
doctrine:
    dbal:
        driver:   '%database_driver%'
        host:     '%database_host%'
        dbname:   '%database_name%'
        user:     '%database_user%'
        password: '%database_password%'
```

Doctrine - Ejemplo de entidad (I)

```php
<?php

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="products")
 */
class Product
{
```

Doctrine - Ejemplo de entidad (II)

```
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(name="model", type="string", length=100)
     */
    protected $model;
}
```

Doctrine - Relación One to One unidireccional

```php
class Product
{
    /**
     * @OneToOne(targetEntity="Shipping")
     * @JoinColumn(name="shipping_id", referencedColumnName="id")
     */
    private $shipping;
}
```

Doctrine - Relación One to One bidireccional (I)

```php
class Customer
{
    /**
     * @OneToOne(targetEntity="Cart", mappedBy="customer")
     */
    private $cart;
}
```

Doctrine - Relación One to One bidireccional (II)

```php
class Cart
{
    /**
     * @OneToOne(targetEntity="Customer", inversedBy="cart")
     * @JoinColumn(name="customer_id", referencedColumnName="id")
     */
    private $customer;
}
```

Doctrine - Relación Many to One unidireccional

```php
class User
{
    /**
     * @ManyToOne(targetEntity="Address")
     * @JoinColumn(name="address_id", referencedColumnName="id")
     */
    private $address;
}
```

Doctrine - Relación One to Many, bidireccional (I)

```php
class Product
{
    /**
     * @OneToMany(targetEntity="Feature", mappedBy="product")
     */
    private $features;

    public function __construct() {
        $this->features = new ArrayCollection();
    }
}
```

Doctrine - Relación One to Many, bidireccional (II)

```php
class Feature
{
    /**
     * @ManyToOne(targetEntity="Product", inversedBy="features")
     * @JoinColumn(name="product_id", referencedColumnName="id")
     */
    private $product;
}
```

Doctrine - Relación Many to Many unidireccional

```
class User
{
    /**
     * @ManyToMany(targetEntity="Group")
     * @JoinTable(name="users_groups",
     *      joinColumns={@JoinColumn(name="user_id",
referencedColumnName="id")},
     *      inverseJoinColumns={@JoinColumn(name="group_id",
referencedColumnName="id")}
     *      )
     */
    private $groups;
```

Doctrine - Relación Many to Many bidireccional (I)

```php
class User
{
    /**
     * @ManyToMany(targetEntity="Group", inversedBy="users")
     * @JoinTable(name="users_groups")
     */
    private $groups;

    public function __construct() {
        $this->groups = new ArrayCollection();
    }
}
```

Doctrine - Relación Many to Many bidireccional (II)

```php
class Group
{
    /**
     * @ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

    public function __construct() {
        $this->users = new ArrayCollection();
    }
}
```

Doctrine - Insertar en la base de datos

```php
$product = new Product();
$product->setName('A Foo Bar');
$product->setPrice('19.99');
$product->setDescription('Lorem ipsum dolor');

$entity_manager->persist($product);
$entity_manager->flush();
```

Servicio de Symfony: doctrine.orm.entity_manager

Doctrine - Repositorios

Para poder leer registros de la base de datos, hay que crear un repositorio de una entidad concreta:

```
$repository = $entity_manager->getRepository('AppBundle:Product');
```

Este repositorio se puede crear directamente desde el services.yml:

```
acme.appbundle.product:
    class: Doctrine\ORM\EntityRepository
    factory_service: doctrine.orm.entity_manager
    factory_method: getRepository
    arguments:
        - Acme\AppBundle\Entity\Product
```

Doctrine - Consultas a la base de datos (I)

```php
//  Buscar por id
$product = $repository->find($id);

//  Buscar un producto por un campo específico
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');
$product = $repository->findOneBy(
    array('name' => 'foo', 'price' => 19.99)
);
```

Doctrine - Consultas a la base de datos (II)

```php
//  Devolver todos los productos
$products = $repository->findAll();

//  Buscar varios productos por un campo específico
$products = $repository->findByPrice(19.99);
$products = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

Doctrine - Actualizar y borrar registros

Actualizar un registro

```
$product = $em->getRepository('AppBundle:Product')->find($id);
$product->setName('New product name!');
$entity_manager->flush();
```

Borrar un registro

```
$product = $em->getRepository('AppBundle:Product')->find($id);
$entity_manager->remove($product);
$entity_manager->flush();
```

Doctrine - Queries complejas - DQL puro

```php
//  DQL = Doctrine Query Language
$query = $em->createQuery(
    'SELECT p
    FROM AppBundle:Product p
    WHERE p.price > :price
    ORDER BY p.price ASC'
)->setParameter('price', '19.99');
$products = $query->getResult();
```

Doctrine - Queries complejas - Query Builder

```php
//  Doctrine Query Builder
$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();
$products = $query->getResult();
```

Doctrine - Comandos

```
# Crea la base de datos con los datos del fichero parameters.yml
php app/console doctrine:database:create

# Generar getters y setters
php app/console doctrine:generate:entities
AppBundle/Entity/Product

# Actualizar la base de datos
php app/console doctrine:schema:update --force

# Devolver las queries SQL
php app/console doctrine:schema:update --dump-sql
```

Tutorial: Creación de un módulo en Drupal 8

Crear módulo

En Drupal 7 había que crear 2 ficheros (.info y .module).

En Drupal 8 basta con crear un .info.yml.

```
name: Drupal 8 Demo module
description: 'Demo module for Drupal 8'
type: module
core: 8.x
```

Los directorios custom/ y contrib/ ahora van en la raíz de la carpeta modules/

- Razón: el código del core se ha movido al directorio core/

Rutas + Controladores

hook_menu ya no es necesario. En su lugar, definimos las rutas en un yaml llamado nombre_del_modulo.routing.yml:

```
demo.demo:
    path: '/demo'
    defaults:
        _controller: '\Drupal\demo\Controller\DemoController::
demo'
        _title: 'Demo'
    requirements:
        _permission: 'access content'
```

Rutas + Controladores - DemoController

```php
namespace Drupal\demo\Controller;

class DemoController
{
    public function demo() {
        return array(
            '#markup' => t('Hello World!')
        );
    }
}
```

Bloques y formularios - Crear bloque (I)

```php
/**
 * @Block(
 *   id = "demo_block",
 *   admin_label = @Translation("Demo block"),
 * )
 */
namespace Drupal\demo\Plugin\Block;
use Drupal\Core\Block\BlockBase;
use Drupal\Core\Form\FormStateInterface;

class DemoBlock extends BlockBase {
```

Bloques y formularios - Crear bloque (II)

```
public function build() {
    $config = $this->getConfiguration();
    if (isset($config['demo_block_settings']) && !empty($config
['demo_block_settings'])) {
        $name = $config['demo_block_settings'];
    } else {
        $name = $this->t('to no one');
    }
    return array(
        '#markup' => $this->t('Hello @name!', array('@name' =>
$name)),
    );
}
```

Bloques y formularios - Configuración del bloque (I)

```
public function blockForm($form, FormStateInterface $form_state) {
    $form = parent::blockForm($form, $form_state);
    $config = $this->getConfiguration();
    $form['demo_block_settings'] = array(
        '#type' => 'textfield',
        '#title' => $this->t('Who'),
        '#description' => $this->t('Please write a name'),
        '#default_value' => isset($config['demo_block_settings'])
? $config['demo_block_settings'] : '',
    );
    return $form;
}
```

Bloques y formularios - Configuración del bloque (II)

```
   public function blockSubmit($form, FormStateInterface
$form_state) {
        $this->setConfigurationValue('demo_block_settings',
$form_state->getValue('demo_block_settings'));
    }
}
```

Bloques y formularios - Crear formulario (I)

```php
<?php

namespace Drupal\demo\Form;
use Drupal\Core\Form\FormBase;
use Drupal\Core\Form\FormStateInterface;

class DemoForm extends FormBase {

    public function getFormId() {
        return 'demo_form';
    }
```

Bloques y formularios - Crear formulario (II)

```
   public function buildForm(array $form, FormStateInterface
$form_state) {
        $form['email'] = array(
            '#type' => 'email',
            '#title' => $this->t('Your email address.')
        );
        $form['actions']['submit'] = array(
            '#type' => 'submit',
            '#value' => $this->t('Save configuration'),
            '#button_type' => 'primary',
        );
        return $form;
    }
```

Bloques y formularios - Crear formulario (III)

```
   public function validateForm(array &$form, FormStateInterface
$form_state) {
        if (filter_var($form_state->getValue('email'),
FILTER_VALIDATE_EMAIL)) {
            $form_state->setErrorByName('email', $this->t('This is
not a valid email.'));
        }
    }
```

Bloques y formularios - Crear formulario (IV)

```
    public function submitForm(array &$form, FormStateInterface
$form_state)
    {
        drupal_set_message($this->t('Your email address is
@email', array('@email' => $form_state->getValue('email'))));
    }
}
```

Bloques y formularios - Ruta del formulario

```
demo.form:
    path: '/demo/form'
    defaults:
        _form: '\Drupal\demo\Form\DemoForm'
        _title: 'Demo Form'
    requirements:
        _permission: 'access content'
```

Inyección de dependencias - Fichero yaml

Fichero nombre_de_modulo.services.yml en la raíz del módulo:

```
services:
    demo.demo_service:
        class: Drupal\demo\Service\DemoService
```

Clase DemoService:

```
namespace Drupal\demo\Service;
class DemoService {
    public function getValue() {
        return "Example service value";
    }
}
```

Inyección de dependencias - Inyectar servicio (I)

```php
namespace Drupal\demo\Controller;

use Drupal\Core\Controller\ControllerBase;
use Symfony\Component\DependencyInjection\ContainerInterface;
//  La clase debe implementar la interfaz
ContainerInjectionInterface
class DemoController extends ControllerBase
{
    protected $demo_service;
    public function __construct($demo_service) {
        $this->demo_service = $demo_service;
    }
```

Inyección de dependencias - Inyectar servicio (II)

```
    public static function create(ContainerInterface $container) {
        return new static($container->get('demo.demo_service'));
    }
    public function demo() {
        return array(
            '#markup' => t(
                'Hello @value!',
                array('@value' => $this->demo_service->getValue())
            )
        );
    }
}
```

Event Dispatcher - Inyectar en formulario

```php
// ...
use Symfony\Component\DependencyInjection\ContainerInterface;
// ...
protected $event_dispatcher;

public function __construct($event_dispatcher) {
    $this->event_dispatcher = $event_dispatcher;
}

public static function create(ContainerInterface $container) {
    return new static($container->get('event_dispatcher'));
}
```

Event Dispatcher - Lanzar evento

```php
public function submitForm(
    array &$form,
    FormStateInterface $form_state
) {
    $demo_event = new DemoEvent(
        $form_state->getValue('email')
    );
    $event = $this->event_dispatcher->dispatch(
        'demo_form.submit',
        $demo_event
    );
}
```

Event Dispatcher - Evento personalizado

```php
namespace Drupal\demo;
use Symfony\Component\EventDispatcher\Event;

class DemoEvent extends Event {
    protected $value;
    public function __construct($value) {
        $this->value = $value;
    }
    public function getValue() {
        return $this->value;
    }
}
```

Event Dispatcher - Subscriber

```php
namespace Drupal\demo\EventSubscriber;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class DemoSubscriber implements EventSubscriberInterface {
    static function getSubscribedEvents() {
        $events['demo_form.submit'][] = array('onSubmit', 0);
        return $events;
    }
    public function onSubmit($event) {
        error_log($event->getValue(), 0);
    }
}
```

Event Dispatcher - Registar un nuevo servicio

```
demo.demo_subscriber:
    class: Drupal\demo\EventSubscriber\DemoSubscriber
    tags:
      - { name: event_subscriber }
```

Es importante añadir la etiqueta event_subscriber

Menús de administración

Crear un fichero llamado nombre_del_modulo.links.menu.yml en la raíz del módulo

```
demo.demo:
    title: Demo Link
    description: 'This is a demo link'
    parent: system.admin_structure
    route_name: demo.demo
```

Configuración

La configuración se guarda en ficheros YAML.

Cuando se activa un módulo, esta configuración se importa a la base de datos.

Nuestros módulos pueden aportar su propia configuración:

- Carpeta: config/install/
- Archivo: nombre_del_modulo.settings.yml

Ejemplo:

```
demo:
    email_address: demo@demo.com
```

Tras añadir este fichero, hay que reinstalar el módulo.

Configuración - Formulario (I)

```php
namespace Drupal\demo\Form;

use Drupal\Core\Form\ConfigFormBase;
use Drupal\Core\Form\FormStateInterface;

class DemoConfigForm extends ConfigFormBase
{
    public function getFormId()
    {
        return 'demo_config_form';
    }
```

Configuración - Formulario (II)

```
   public function buildForm(array $form, FormStateInterface
$form_state) {
        $form = parent::buildForm($form, $form_state);
        $config = $this->config('demo.settings');
        $form['email'] = array(
            '#type' => 'email',
            '#title' => $this->t('Your email address.'),
            '#default_value' => $config->get('demo.email_address')
        );
        return $form;
    }
```

Configuración - Formulario (III)

```
    public function validateForm(array &$form, FormStateInterface
$form_state) {
        if (filter_var($form_state->getValue('email'),
FILTER_VALIDATE_EMAIL)) {
            $form_state->setErrorByName('email', $this->t('This is
not a valid email.'));
        }
    }

    protected function getEditableConfigNames() {
        return array('demo.settings');
    }
```

Configuración - Formulario (IV)

```
   public function submitForm(array &$form, FormStateInterface
$form_state) {
        $config = $this->config('demo.settings');
        $config->set('demo.email_address', $form_state->getValue
('email'));
        $config->save();

        drupal_set_message($this->t('Your email address is
@email', array('@email' => $form_state->getValue('email'))));
    }
}
```

Configuración - Ruta del formulario

```
demo.config_form:
    path: '/demo/config-form'
    defaults:
        _form: '\Drupal\demo\Form\DemoConfigForm'
        _title: 'Demo ConfigForm'
    requirements:
        _permission: 'access content'
```

EntityFieldQuery

Se ha reemplazado la clase EntityFieldQuery por el servicio entity.query (implementa la interfaz \Drupal\Core\Entity\Query\QueryInterface)

Podemos usar el servicio estáticamente...

```
$query = \Drupal::entityQuery('node');
```

…o bien, obtener una instancia del servicio:

```
$entity_query_service = $container->get('entity.query');
$query = $entity_query_service->get('node');
```

EntityFieldQuery - Queries

Obtener nodos publicados:

```
$query = \Drupal::entityQuery('node')
    ->condition('status', 1);
$nids = $query->execute(); //  Array of entity ids
```

Más condiciones:

```
$query = \Drupal::entityQuery('node')
    ->condition('status', 1)
    ->condition('title', 'cat', 'CONTAINS')
    ->condition('field_tags.entity.name', 'cats');
$nids = $query->execute();
```

EntityFieldQuery - Condiciones OR

```php
$query = \Drupal::entityQuery('node')
    ->condition('status', 1)
    ->condition('changed', REQUEST_TIME, '<');

$group = $query->orConditionGroup()
    ->condition('title', 'cat', 'CONTAINS')
    ->condition('field_tags.entity.name', 'cats');

$nids = $query->condition($group)->execute();
```

EntityFieldQuery - Cargar entidades

```php
//  Un objeto
$node = entity_load('node', $nids[1]);


//  Múltiples objetos
$nodes = entity_load_multiple('node', $nids);
```

Qué hacen ambas funciones:

- Obtienen el StorageManager de la entidad.
- Invocan al método load o loadMultiple.

EntityFieldQuery - Obtener StorageManager

Estáticamente:

```
$node_storage = \Drupal::entityManager()->getStorage('node');
$node_storage->loadMultiple($ids);
$node_storage->load($id);
```

Obtener el objeto:

```
$node_storage = $container->get('entity.manager')->getStorage
('node');
```

¡Gracias!