Twig

Para seguir esta sección se deberían tener conocimientos de twig de:

- Variables
- Filtros
- Funciones
- Estructuras de control
- Comentarios
- Inclusión de plantillas
- Herencia de plantillas
- Operadores

Puedes encontrar documentación sobre los temas anteriores en:

http://symfony.com/doc/current/templating.html

Para activar el sistema de plantillas twig, si no lo tenemos instalado, hay que instalarlo.

> *composer require twig*

Twig

Output escaping

Para evitar ataques Cross Site Scripting (XSS) y/o para evitar roturas del código html, twig tiene activado por defecto el output escaping.

Veamos un ejemplo clásico:

Dado el siguiente twig

```
Hello {{ name }}
```

si el usuario introduce el siguiente texto para su nombre

```
<script>alert('hello!')</script>
```

Si no hubiera output escaping, el resultado renderizado sería el siguiente

```
Hello <script>alert('hello!')</script>
```

Es decir, el usuario habría conseguido ejecutar código javascript en nuestra aplicación.

Sin embargo, con output escaping, el renderizado sería el siguiente

```
Hello &lt;script&gt;alert(&#39;hello!&#39;)&lt;/script&gt;
```

NOTA: Si utilizas sistema de plantillas PHP entonces el output escaping NO es automático.

Deshabilitar el output escaping

Si quieres deshabilitar el output escaping en una variable concreta, basta con aplicarle el filtro *raw*.

```
Hello {{ name }}
```

Twig

Utilizar PHP como sistema de plantillas

Si queremos utilizar PHP como sistema de plantillas, lo primero que tenemos que hacer es instalar el componente de *templating*.

> *composer require templating*

Y habilitar el motor de php en la configuración

```
# config/packages/framework.yaml
framework:
    # ...
    templating:
        engines: ['twig', 'php']
```

Para renderizar una plantilla php, basta con renderizar un archivo con extensión .php en vez de .twig.

```
// src/Controller/HelloController.php

// ...
public function index($name)
{
```

```
    // template is stored in src/Resources/views/hello/ind
ex.html.php
    return $this->render('hello/index.html.php', array(
        'name' => $name
    ));
}
```

El motor de plantillas PHP también soporta herencia de plantillas, inclusión de plantillas, output escaping, etc.

http://symfony.com/doc/current/templating/PHP.html

Cómo depurar variables en twig

En el componente **VarDumper** existe una función llamada *dump()* muy útil para la depuración de variables tanto en twig como en los controladores.

Para utilizar esta función antes de nada tenemos que asegurarnos de tener el componente instalado:

> *composer require var-dumper*

Ahora ya podemos utilizarlo.

```
// src/Controller/ArticleController.php
namespace App\Controller;

// ...

class ArticleController extends Controller
{
public function recentList()
{
$articles = ...;
dump($articles);
```

```
    // ...
}
```

}

La función dump() renderiza el valor de la variable en la barra de depuración.

En twig tenemos dos formas de utilizar dump:

- {% dump foo.bar %}
- {{ dump(foo.bar) }}

La primera de ellas, renderiza el valor de la variable en la barra de depuración, pero la segunda, renderiza el valor de la variable en el propio html.

La función dump() por defecto solamente está disponible en los entornos dev y test. Utilizarla en el entorno de producción provocaría un error de PHP.

Twig

Cómo generar otros formatos de salida (css, javascript, xml...)

Realmente, el formato de un fichero Twig es independiente de su extensión, pero por convención, los ficheros twig llevan doble extensión, la primera extensión para informar sobre el formato y la segunda extensión, la propia de twig.

- article/show.xml.twig
- article/show.html.twig
- article/show.json.twig

Pero lo que realmente define el formato es el propio contenido del fichero.

Aquí tenemos un ejemplo de una acción que elige un twig u otro dependiendo del parámetro especial *_format*.

```
// ...
use Symfony\Component\Routing\Annotation\Route;

class ArticleController extends Controller
{
    /**
     * @Route("/{slug}.{_format}", defaults={"_format"="ht
```

```
ml"})
     */
    public function show(Request $request, $slug)
    {
        // retrieve the article based on $slug
        $article = ...;

        $format = $request->getRequestFormat();

        return $this->render('article/show.'.$format.'.twi
g', array(
            'article' => $article,
        ));
    }
}
```

Al crear enlaces a estar ruta, basta con indicar el formato.

```
<a href="{{ path('article_show', {'slug': 'about-us', '_fo
rmat': 'html'}) }}">
    Ver como HTML
</a>
<a href="{{ path('article_show', {'slug': 'about-us', '_fo
rmat': 'xml'}) }}">
    Ver como XML
</a>
<a href="{{ path('article_show', {'slug': 'about-us', '_fo
```

```
rmat': 'json'}) }}">
    Ver como JSON
</a>
```

http://symfony.com/doc/current/templating/formats.html

Twig

Cómo inyectar variables globales

En ocasiones queremos tener variables accesibles para todas las plantillas.

Esto se puede hacer en la sección *globals* de la configuración de twig.

Veamos un ejemplo:

```
# config/packages/twig.yaml
twig:
    # ...
    globals:
        ga_tracking: UA-xxxxx-x
```

Ahora cualquier plantilla puede renderizar la variable *ga_tracking*.

```
<p>The google tracking code is: {{ ga_tracking }}</p>
```

Por supuesto, podemos parametrizar la variable o que sea una variable de entorno.

```
# config/services.yaml
parameters:
```

```
    ga_tracking: UA-xxxxx-x
```

```
# config/packages/twig.yaml
twig:
    globals:
        ga_tracking: '%ga_tracking%'
```

O podemos también hacer que la variable global contenga un servicio

```
# config/packages/twig.yaml
twig:
    # ...
    globals:
        # the value is the service's id
        user_management: '@App\Service\UserManagement'
```

NOTA: El servicio no se carga de forma lazy. Dicho de otra forma, tan pronto como se carga Twig, el servicio es instanciado, aunque nunca se llegue a utilizar la variable global.

Twig

Sobrescribir plantillas de bundles de terceros

Si alguna vez necesitamos sobrescribir una plantilla Twig de un bundle de terceros, Symfony nos proporciona un método muy sencillo para hacerlo.

Imaginemos que queremos sobrescribir la plantilla **Resources/views/Blog/index.html.twig** del bundle **UnBundle**.

Para hacerlo, simplemente debemos copiar dicho archivo en **templates/bundles/UnBundle/Blog/index.html.twig** y realizar los cambios que deseemos en este archivo.

Después de esto, debemos limpiar la caché de symfony incluso si estamos trabajando en el entorno de desarrollo.

Twig

Crear plantillas sin controladores

Muchas veces necesitamos crear una página con contenido estático, sin programación.

En estos casos, podemos evitar crear un controlador para la plantilla utilizando un controlador específico de symfony.

```
# config/routes.yaml
textolegal_privacidad:
    path:         /privacidad
    controller:   Symfony\Bundle\FrameworkBundle\Controlle
r\TemplateController::templateAction
    defaults:
        template: textoslegales/privacy.html.twig
```

En estos casos de páginas estáticas es interesante además cachearlas

```
textolegal_privacidad:
    path:          /privacidad
    controller:    Symfony\Bundle\FrameworkBundle\Controll
er\TemplateController::templateAction
    defaults:
        template:  'textoslegales/privacy.html.twig'
```

```
        maxAge:    86400
        sharedAge: 86400
```

Veremos más asuntos acerca de la caché en el tema correspondiente.

Twig

Cómo crear una extensión de twig

Antes de crear una extensión de Twig, conviene comprobar si el filtro/función que necesitamos está ya implementado en las extensiones de Symfony o en las oficiales de Twig.

Las extensiones oficiales de twig se pueden encontrar en este enlace: https://github.com/twigphp/Twig-extensions

y se instalan con el siguiente comando de composer:

> *composer require twig/extensions*

1. Creación de la clase

Vamos a crear una extensión clásica que formatea un número en formato de moneda.

Se utilizaría de la siguiente forma, con parámetros opcionales:

```
{{ product.price|price }}

{{ product.price|price(2, ',', '.') }}
```

Habría que empezar creando una clase que extienda de la clase **AbstractExtension** definida por Twig y programar la lógica.

```
// src/Twig/AppExtension.php
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

class AppExtension extends AbstractExtension
{
    public function getFilters()
    {
        return array(
            new TwigFilter('price', array($this, 'priceFil
ter')),
        );
    }

    public function priceFilter($number, $decimals = 0, $d
ecPoint = '.', $thousandsSep = ',')
    {
        $price = number_format($number, $decimals, $decPoi
nt, $thousandsSep);
        $price = $price.'€';

        return $price;
```

```
    }
}
```

La función *getFilters()* devuelve un array con la lista de filtros personalizados.

```
    public function getFilters()
    {
        return array(
            new TwigFilter('filtro1', array($this, 'filtro
1Filter')),
            new TwigFilter('filtro2', array($this, 'filtro
2Filter')),
            new TwigFilter('filtro3', array($this, 'filtro
3Filter')),
            new TwigFilter('filtro4', array($this, 'filtro
4Filter')),
        );
    }
```

Dicho array es un array de objetos TwigFilter, en los que se indica el nombre del filtro y el método de esta clase que implementa la lógica del filtro.

Análogamente a getFilters(), se pueden implementar los métodos getFunctions(), getGlobals(), getTokenParsers(), getOperators() y getTests() para definir funciones, variables globales, tags, operadores y

funciones de tests respectivamente.

Más documentación:
https://twig.symfony.com/doc/2.x/advanced.html#id1

Una vez registrado el filtro y creado el método correspondiente, queda registrar la clase anterior como un servicio de Symfony.

2. Registrar la extensión como un servicio

Para que symfony reconozca la clase como una extensión de symfony, habría que registrarla en el fichero services.yaml con la etiqueta twig.extension.

Sin embargo, si utilizamos la configuración por defecto de Symfony, este paso no es necesario. Symfony reconocerá automáticamente la clase como extensión de Twig.

Entenderemos mejor esto cuando veamos el tema de Servicios.

Con estos dos pasos, ya podemos utilizar el filtro *price* en cualquier plantilla.

Crear extensiones Lazy-Loaded

Por defecto, Twig inicializa todas las extensiones antes de renderizar ninguna plantilla.

Si tenemos extensiones con dependencias complejas, podríamos estar perjudicando el rendimiento de la aplicación.

Desde Twig 1.26 es posible crear extensiones que se carguen bajo demanda.

Siguiendo el mismo ejemplo del filtro *price*, el primer cambio sería sacar el método priceFilter() de la extensión y llevarlo a otra clase.

```
// src/Twig/AppExtension.php
namespace App\Twig;

use App\Twig\AppRuntime;

class AppExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return array(
            // the logic of this filter is now implemented
 in a different class
            new \Twig_SimpleFilter('price', array(AppRunti
me::class, 'priceFilter')),
        );
    }
}
```

```php
// src/Twig/AppRuntime.php
namespace App\Twig;

class AppRuntime
{
    public function priceFilter($number, $decimals = 0, $decPoint = '.', $thousandsSep = ',')
    {
        $price = number_format($number, $decimals, $decPoint, $thousandsSep);
        $price = '$'.$price;

        return $price;
    }
}
```

El siguiente cambio, sería registrar la extensión en el fichero services.yaml con el tag *twig.runtime*.

```yaml
# app/config/services.yml
services:
    app.twig_runtime:
        class: App\Twig\AppRuntime
        public: false
        tags:
            - { name: twig.runtime }
```

De esta forma, esta extensión solamente se cargaría cuando fuera a ser utilizada.