Manual de Twig

Release 1.2.0

Traducido por Nacho Pacheco

November 05, 2011

Índice general

Introducción

Esta es la documentación de *Twig*, el flexible, rápido y seguro motor de plantillas para *PHP*.

Si has estado expuesto a otros lenguajes de plantilla basados en texto, tal como *Smarty, Django* o *Jinja*, debes sentirte como en casa con *Twig*. Es a la vez, un amigable ambiente para el diseñador y desarrollador apegado a los principios de *PHP*, añadiendo útil funcionalidad a los entornos de plantillas.

Las características clave son...

- *Rápido*: *Twig* compila las plantillas hasta código *PHP* regular optimizado. El costo general en comparación con código *PHP* regular se ha reducido al mínimo.
- *Seguro*: *Twig* tiene un modo de recinto de seguridad para evaluar el código de plantilla que no es confiable. Esto te permite utilizar *Twig* como un lenguaje de plantillas para aplicaciones donde los usuarios pueden modificar el diseño de la plantilla.
- *Flexible*: *Twig* es alimentado por flexibles analizadores léxico y sintáctico. Esto permite al desarrollador definir sus propias etiquetas y filtros personalizados, y crear su propio *DSL*.

1.1 Requisitos previos

Twig necesita por lo menos **PHP 5.2.4** para funcionar.

1.2 Instalando

Tienes varias formas de instalar *Twig*. Si no estás seguro qué hacer, descarga el archivo comprimido (`tarball`).

1.2.1 Desde la versión comprimida

1. Descarga el archivo comprimido más reciente desde la página de descarga
2. Descomprime el archivo
3. Mueve los archivos a algún lugar en tu proyecto

1.2.2 Instalando la versión de desarrollo

1. Instala desde *Subversión* o *Git*
2. Para *Subversión*: `svn co http://svn.twig-project.org/trunk/ twig`, para *Git*: `git clone git://github.com/fabpot/Twig.git`

1.2.3 Instalando el paquete *PEAR*

1. Instala *PEAR*
2. `pear channel-discover pear.twig-project.org`
3. `pear install twig/Twig` (o `pear install twig/Twig-beta`)

1.3 Uso básico de la *API*

Esta sección te ofrece una breve introducción a la *API PHP* de *Twig*.

El primer paso para utilizar *Twig* es registrar su cargador automático:

```
require_once '/ruta/a/lib/Twig/Autoloader.php';
Twig_Autoloader::register();
```

Sustituye `/ruta/a/lib/` con la ruta que utilizaste en la instalación de *Twig*.

Nota: *Twig* sigue la convención de nombres de *PEAR* para sus clases, lo cual significa que puedes integrar fácilmente las clases de *Twig* cargándolo en tu propio cargador automático.

```
$loader = new Twig_Loader_String();
$twig = new Twig_Environment($loader);

echo $twig->render('Hello {{ name }}!', array('name' => 'Fabien'));
```

Twig utiliza un cargador (`Twig_Loader_String`) para buscar las plantillas, y un entorno (`Twig_Environment`) para almacenar la configuración.

El método `render()` carga la plantilla pasada como primer argumento y la reproduce con las variables pasadas como segundo argumento.

Debido a que las plantillas generalmente se guardan en el sistema de archivos, *Twig* también viene con un cargador del sistema de archivos:

```
$loader = new Twig_Loader_Filesystem('/ruta/a/templates');
$twig = new Twig_Environment($loader, array(
  'cache' => '/ruta/a/compilation_cache',
));

echo $twig->render('index.html', array('name' => 'Fabien'));
```

Twig para diseñadores de plantillas

Este documento describe la sintaxis y semántica del motor de plantillas y será muy útil como referencia para quién esté creando plantillas *Twig*.

2.1 Sinopsis

Una plantilla simplemente es un archivo de texto. Esta puede generar cualquier formato basado en texto (*HTML*, *XML*, *CSV*, *LaTeX*, etc.) No tiene una extensión específica, `.html` o `.xml` están muy bien.

Una plantilla contiene **variables** o **expresiones**, las cuales se reemplazan por valores cuando se evalúa la plantilla, y las **etiquetas**, controlan la lógica de la plantilla.

A continuación mostramos una plantilla mínima que ilustra algunos conceptos básicos. Veremos los detalles más adelante en este documento:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <ul id="navigation">
        {% for item in navigation %}
            <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
        {% endfor %}
        </ul>

        <h1>My Webpage</h1>
        {{ a_variable }}
    </body>
</html>
```

Hay dos tipos de delimitadores: `{% ... %}` y `{{ ... }}`. El primero se utiliza para ejecutar declaraciones como bucles `for`, el último imprime en la plantilla el resultado de una expresión.

2.2 Integrando con *IDEs*

Los *IDEs* modernos son compatibles con el resaltado de sintaxis y autocompletado en una amplia gama de lenguajes.

- *Textmate* vía el paquete Twig
- *Vim* vía el complemento de sintaxis Jinja
- *Netbeans* vía el complemento de sintaxis Twig
- *PhpStorm* (nativo desde la versión 2.1)
- *Eclipse* vía el complemento Twig
- *Sublime Text* vía el paquete Twig
- *GtkSourceView* vía el Twig language definition (usado por *gedit* y otros proyectos)

2.3 Variables

La aplicación pasa variables a las plantillas para que puedas combinarlas en la plantilla. Las variables pueden tener atributos o elementos en ellas a los cuales puedes acceder también. Cómo se ve una variable, en gran medida, depende de la aplicación que la proporcione.

Puedes utilizar un punto (`.`) para acceder a los atributos de una variable (métodos o propiedades de un objeto *PHP*, o elementos de una matriz *PHP*), o la así llamada sintaxis de "subíndice" (`[]`).

```
{{ foo.bar }}
{{ foo['bar'] }}
```

Nota: Es importante saber que las llaves no son parte de la variable, sino de la declaración de impresión. Si accedes a variables dentro de las etiquetas no las envuelvas con llaves.

Si una variable o atributo no existe, recibirás un valor `nulo`.

Implementación

Por razones de conveniencia `foo.bar` hace lo siguiente en la capa *PHP*:

- Comprueba si `foo` es una matriz y `bar` un elemento válido;
- si no, y si `foo` es un objeto, comprueba que `bar` es una propiedad válida;
- si no, y si `foo` es un objeto, comprueba que `bar` es un método válido (incluso si `bar` es el constructor — usa `__construct()` en su lugar);
- si no, y si `foo` es un objeto, comprueba que `getBar` es un método válido;
- si no, y si `foo` es un objeto, comprueba que `isBar` es un método válido;
- si no, devuelve un valor `null`.

`foo['bar']` por el contrario sólo trabaja con matrices *PHP*:

- Comprueba si `foo` es una matriz y `bar` un elemento válido;
- si no, devuelve un valor `null`.

Nota: Si deseas obtener un atributo dinámico en una variable, utiliza la función *attribute* (Página 68) en su lugar.

2.3.1 Variables globales

Las siguientes variables siempre están disponibles en las plantillas:

- `_self`: hace referencia a la plantilla actual;
- `_context`: hace referencia al contexto actual;
- `_charset`: hace referencia al juego de caracteres actual.

2.3.2 Definiendo variables

Puedes asignar valores a las variables dentro de los bloques de código. Las asignaciones usan la etiqueta *set* (Página 51):

```
{% set foo = 'foo' %}
{% set foo = [1, 2] %}
{% set foo = {'foo': 'bar'} %}
```

2.4 Filtros

Los **filtros** pueden modificar variables. Los filtros están separados de la variable por un símbolo de tubo (`|`) y pueden tener argumentos opcionales entre paréntesis. Puedes encadenar múltiples filtros. La salida de un filtro se aplica al siguiente.

El siguiente ejemplo elimina todas las etiquetas *HTML* del `name` y lo formatea como nombre propio:

```
{{ name|striptags|title }}
```

Los filtros que aceptan argumentos llevan paréntesis en torno a los argumentos. Este ejemplo unirá una lista con comas:

```
{{ list|join(', ') }}
```

Para aplicar un filtro en una sección de código, envuélvelo con la etiqueta *filter* (Página 50):

```
{% filter upper %}
  Este texto cambia a mayúsculas
{% endfilter %}
```

Ve a la página de *filtros* (Página 61) para aprender más acerca de los filtros incorporados.

2.5 Funciones

Las funciones se pueden llamar para generar contenido. Las funciones son llamadas por su nombre seguido de paréntesis (`()`) y pueden tener argumentos.

Por ejemplo, la función `range` devuelve una lista que contiene una progresión aritmética de números enteros:

```
{% for i in range(0, 3) %}
    {{ i }},
{% endfor %}
```

Ve a la página *funciones* (Página 67) para aprender más acerca de las funciones incorporadas.

2.6 Estructuras de control

Una estructura de control se refiere a todas esas cosas que controlan el flujo de un programa — condicionales (es decir, `if`/`elseif`/`else`), bucles `for`, así como cosas tales como bloques. Las estructuras de control aparecen dentro de bloques `{% ... %}`.

Por ejemplo, para mostrar una lista de usuarios provista en una variable llamada `users`, usa la etiqueta *for* (Página 47):

```
<h1>Members</h1>
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

Puedes utilizar la etiqueta *if* (Página 49) para probar una expresión:

```
{% if users|length > 0 %}
    <ul>
        {% for user in users %}
            <li>{{ user.username|e }}</li>
        {% endfor %}
    </ul>
{% endif %}
```

Ve a la página *etiquetas* (Página 47) para aprender más acerca de las etiquetas incorporadas.

2.7 Comentarios

Para comentar parte de una línea en una plantilla, utiliza la sintaxis de comentario `{# ... #}`. Esta es útil para depurarción o para agregar información para los diseñadores de otra plantilla o para ti mismo:

```
{# nota: inhabilitado en la plantilla porque ya no utiliza
    {% for user in users %}
        ...
    {% endfor %}
#}
```

2.8 Incluyendo otras plantillas

La etiqueta *include* (Página 54) es útil para incluir una plantilla y devolver el contenido reproducido de esa plantilla a la actual:

```
{% include 'sidebar.html' %}
```

De manera predeterminada se pasa el contexto actual a las plantillas incluidas.

El contexto que se pasa a la plantilla incluida incorpora las variables definidas en la plantilla:

```
{% for box in boxes %}
    {% include "render_box.html" %}
{% endfor %}
```

La plantilla incluida `render_box.html` es capaz de acceder a `box`.

El nombre de archivo de la plantilla depende del gestor de plantillas. Por ejemplo, el `Twig_Loader_Filesystem` te permite acceder a otras plantillas, dando el nombre del archivo. Puedes acceder a plantillas en subdirectorios con una barra inclinada:

```
{% include "sections/articles/sidebar.html" %}
```

Este comportamiento depende de la aplicación en que integres *Twig*.

2.9 Herencia en plantillas

La parte más poderosa de *Twig* es la herencia entre plantillas. La herencia de plantillas te permite crear un "esqueleto" de plantilla base que contenga todos los elementos comunes de tu sitio y define los **bloques** que las plantillas descendientes pueden sustituir.

Suena complicado pero es muy básico. Es más fácil entenderlo si comenzamos con un ejemplo.

Vamos a definir una plantilla de base, `base.html`, la cual define el esqueleto de un documento *HTML* simple que puede usar para una sencilla página de dos columnas:

```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
            <title>{% block title %}{% endblock %} - My Webpage</title>
        {% endblock %}
    </head>
    <body>
        <div id="content">{% block content %}{% endblock %}</div>
        <div id="footer">
            {% block footer %}
                &copy; Copyright 2011 by <a href="http://domain.invalid/">you</a>.
            {% endblock %}
        </div>
    </body>
</html>
```

En este ejemplo, las etiquetas *{% block %}* (Página 54) definen cuatro bloques que las plantillas herederas pueden rellenar. Todas las etiquetas `bloque` le dicen al motor de plantillas que una plantilla heredera puede sustituir esas porciones de la plantilla.

Una plantilla hija podría tener este aspecto:

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Aquí, la clave es la etiqueta *{ % extends %}* (Página 51). Esta le dice al motor de plantillas que esta plantilla "extiende" otra plantilla. Cuando el sistema de plantillas evalúa esta plantilla, en primer lugar busca a la plantilla padre. La etiqueta `extends` debe ser la primera etiqueta de la plantilla.

Ten en cuenta que debido a que la plantilla heredera no define el bloque `footer`, en su lugar se utiliza el valor de la plantilla padre.

Es posible reproducir el contenido del bloque padre usando la función *parent* (Página 68). Esta devuelve el resultado del bloque padre:

```
{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

Truco: La página de documentación para la etiqueta *extends* (Página 51) describe características más avanzadas como el anidamiento de bloques, ámbito, herencia dinámica, y herencia condicional.

2.10 Escapando *HTML*

Cuando generas *HTML* desde plantillas, siempre existe el riesgo de que una variable incluya caracteres que afecten el *HTML* resultante. Hay dos enfoques: escapar cada variable manualmente o de manera predeterminada escapar todo automáticamente.

Twig apoya ambos, el escape automático está habilitado por omisión.

Nota: El escape automático sólo se admite si has habilitado la extensión *escaper* (el cual es el valor predeterminado).

2.10.1 Trabajando con el escape manual

Si está habilitado el escape manual es **tu** responsabilidad escapar las variables si es necesario. ¿Qué escapar? Si tienes una variable que *puede* incluir cualquiera de los siguientes caracteres (>, <, `&` o `"`) **tienes** que escaparla a menos que la variable contenga *HTML* bien formado y sea de confianza. El escape trabaja *entubando* la variable a través del filtro `|e`:

```
{{ user.username|e }}
{{ user.username|e('js') }}
```

2.10.2 Trabajando con escape automático

Ya sea que el escape automático esté habilitado o no, puedes marcar una sección de una plantilla para que sea escapada o no utilizando la etiqueta *autoescape* (Página 59):

```
{% autoescape true %}
    Todo en este bloque se va a escapar automáticamente
{% endautoescape %}
```

2.11 Escapando

A veces es deseable e incluso necesario contar con que *Twig* omita partes que de lo contrario manejaría como variables o bloques. Por ejemplo, si utilizas la sintaxis predeterminada y deseas utilizar `{{` como cadena sin procesar en la plantilla y no iniciar una variable, tienes que usar un truco.

La forma más sencilla es extraer la variable del delimitador (`{{`) usando una expresión variable:

```
{{ '{{' }}
```

Para secciones mayores tiene sentido marcar un bloque como *raw* (Página 60).

2.12 Macros

Las macros son comparables con funciones en lenguajes de programación regulares. Son útiles para poner modismos *HTML* utilizados frecuentemente en elementos reutilizables para no repetirlos.

Una macro se define a través de la etiqueta *macro* (Página 49). He aquí un pequeño ejemplo de una macro que reproduce un elemento de formulario:

```
{% macro input(name, value, type, size) %}
    <input type="{{ type|default('text') }}"
           name="{{ name }}"
           value="{{ value|e }}"
           size="{{ size|default(20) }}" />
{% endmacro %}
```

Las macros se pueden definir en cualquier plantilla, y es necesario "importarlas", antes de utilizarlas usando la etiqueta *import* (Página 55):

```
{% import "formularios.html" as forms %}

<p>{{ forms.input('username') }}</p>
```

Alternativamente, puedes importar nombres desde la plantilla al espacio de nombres actual vía la etiqueta *from* (Página 57):

```
{% from 'formularios.html' import input as campo_input, textarea %}

<dl>
    <dt>Username</dt>
    <dd>{{ input_field('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ input_field('password', type='password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

2.13 Expresiones

Twig acepta expresiones en cualquier parte. Estas funcionan de manera muy similar a *PHP* regular e incluso si no estás trabajando con *PHP* te debes sentir cómodo con estas.

Nota: La precedencia de los operadores es la siguiente, con los operadores de menor precedencia apareciendo en primer lugar: `or`, `and`, `==`, `!=`, `<`, `>`, `>=`, `<=`, `in`, `+`, `-`, `~`, `*`, `/`, `%`, `//`, `is`, `..`, y `**`.

2.13.1 Literales

La forma más simple de las expresiones son literales. Los literales son representaciones para tipos *PHP*, tal como cadenas, números y matrices. Existen los siguientes literales:

- `"Hello World"`: Todo lo que esté entre comillas simples o dobles es una cadena. Son útiles cuando necesitas una cadena en la plantilla (por ejemplo, como argumentos para llamadas a función, filtros o simplemente para extender o incluir una plantilla).
- `42` / `42.23`: Números enteros y números en coma flotante se crean tan sólo escribiendo el número. Si está presente un punto es un número en coma flotante, de lo contrario es un número entero.
- `["foo", "bar"]`: Las matrices se definen por medio de una secuencia de expresiones separadas por una coma (`,`) y envueltas entre paréntesis cuadrados (`[]`).
- `{"foo": "bar"}`: Los valores `hash` se definen con una lista de claves y valores separados por una coma (`,`) y envueltos entre llaves (`{}`). Un valor puede ser cualquier expresión válida.
- `true` / `false`: `true` representa el valor verdadero, `false` representa el valor falso.
- `null`: `null` no representa un valor específico. Este es el valor devuelto cuando una variable no existe. `none` es un alias para `null`.

Los arreglos y `hashes` se pueden anidar:

```
{% set foo = [1, {"foo": "bar"}] %}
```

2.13.2 Matemáticas

Twig te permite calcular valores. Esto no suele ser útil en las plantillas, pero existe por el bien de la integridad. Admite los siguientes operadores:

- `+`: Suma dos objetos (los operandos se convierten a números). `{{ 1 + 1 }}` es `2`.
- `-`: Sustrae el segundo número del primero. `{{ 3 - 2 }}` es `1`.
- `/`: Divide dos números. El valor devuelto será un número en coma flotante. `{{ 1 / 2 }}` es `{{ 0.5 }}`.
- `%`: Calcula el residuo de una división entera. `{{ 11 % 7 }}` es `4`.
- `//`: Divide dos números y devuelve el resultado entero truncado. `{{ 20 // 7 }}` es `2`.
- `*`: Multiplica el operando de la izquierda con el de la derecha. `{{ 2 * 2 }}` devolverá `4`.
- `**`: Eleva el operando izquierdo a la potencia del operando derecho. `{{ 2**3 }}` debe devolver `8`.

2.13.3 Lógica

Puedes combinar varias expresiones con los siguientes operadores:

- `and`: Devuelve `true` si ambos operandos izquierdo y derecho son `true`.
- `or`: Devuelve `true` si el operando izquierdo o derecho es `true`.
- `not`: Niega una declaración.
- `(expr)`: Agrupa una expresión.

2.13.4 Comparaciones

Los siguientes operadores de comparación son compatibles con cualquier expresión: ==, !=, <, >, >=, y <=.

2.13.5 Operador de contención

El operador `in` realiza la prueba de contención.

Esta devuelve `true` si el operando de la izquierda figura en el de la derecha:

```
{# devuelve true #}

{{ 1 in [1, 2, 3] }}

{{ 'cd' in 'abcde' }}
```

Truco: Puedes utilizar este filtro para realizar una prueba de contención en cadenas, arreglos u objetos que implementan la interfaz `Traversable`.

Para llevar a cabo una prueba negativa, utiliza el operador `not in`:

```
{% if 1 not in [1, 2, 3] %}

{# es equivalente a #}
{% if not (1 in [1, 2, 3]) %}
```

2.13.6 Operador de prueba

El operador `is` realiza pruebas. Puedes utilizar las pruebas para comprobar una variable con una expresión común. El operando de la derecha es el nombre de la prueba:

```
{# averigua si una variable es impar #}

{{ nombre is odd }}
```

Las pruebas también pueden aceptar argumentos:

```
{% if loop.index is divisibleby(3) %}
```

Puedes negar las pruebas usando el operador `not`:

```
{% if loop.index is not divisibleby(3) %}

{# es equivalente a #}
{% if not (loop.index is divisibleby(3)) %}
```

Ve a la página *Probando* (Página 71) para aprender más sobre las pruebas integradas.

2.13.7 Otros operadores

Los siguientes operadores son muy útiles pero no encajan en ninguna de las otras dos categorías:

- `..`: Crea una secuencia basada en el operando antes y después del operador (esta sólo es azúcar sintáctica para la función *range* (Página 67)).

- `|`: Aplica un filtro.
- `~`: Convierte todos los operandos en cadenas y los concatena. `{{ "Hello " ~ name ~ "!" }}` debería devolver (suponiendo que `name` es `'John'`) `Hello John!`.
- `.`, `[]`: Obtiene un atributo de un objeto.
- `?:`: El operador ternario de *PHP*: `{{ foo ? 'yes' : 'no' }}`

2.14 Controlando el espacio en blanco

Nuevo en la versión 1.1: La etiqueta para nivel controlar los espacios en blanco se añadió en la *Twig* 1.1. La primer nueva línea después de una etiqueta de plantilla se elimina automáticamente (como en *PHP*). El motor de plantillas no modifica el espacio en blanco, por lo tanto cada espacio en blanco (espacios, tabuladores, nuevas líneas, etc.) se devuelve sin cambios.

Utiliza la etiqueta `spaceless` para quitar los espacios en blanco entre las etiquetas *HTML*:

```
{% spaceless %}
    <div>
        <strong>foo</strong>
    </div>
{% endspaceless %}

{# Producirá <div><strong>foo</strong></div> #}
```

Además de la etiqueta `spaceless` también puedes controlar los espacios en blanco a nivel de etiquetas. Utilizando el modificador de control de los espacios en blanco en tus etiquetas, puedes recortar los espacios en blanco en ambos extremos:

```
{% set value = 'no spaces' %}
{#- No deja espacios en blanco en ambos extremos -#}
{%- if true -%}
    {{- value -}}
{%- endif -%}

{# produce 'sin espacios' #}
```

El ejemplo anterior muestra el modificador de control de espacios en blanco predeterminado, y cómo lo puedes utilizar para quitar los espacios en blanco alrededor de las etiquetas. Recortar el espacio debe consumir todos los espacios en blanco a ese lado de la etiqueta. Es posible utilizar el recorte de espacios en blanco en un lado de una etiqueta:

```
{% set value = 'no spaces' %}
<li>    {{- value }}    </li>

{# produce '<li>sin espacios    </li>' #}
```

2.15 Extendiendo

Puedes extender *Twig* fácilmente.

Si estás buscando nuevas etiquetas, filtros, o funciones, echa un vistazo al repositorio de extensiones oficial de Twig.

Si deseas crear una propia, lee *extensiones* (Página 31).

CAPÍTULO 3

Twig para desarrolladores

Este capítulo describe la *API* para *Twig* y no el lenguaje de plantillas. Será muy útil como referencia para aquellos que implementan la interfaz de plantillas para la aplicación y no para los que están creando plantillas *Twig*.

3.1 Fundamentos

Twig utiliza un objeto central llamado el **entorno** (de la clase `Twig_Environment`). Las instancias de esta clase se utilizan para almacenar la configuración y extensiones, y se utilizan para cargar plantillas del sistema de archivos o en otros lugares.

La mayoría de las aplicaciones debe crear un objeto `Twig_Environment` al iniciar la aplicación y usarlo para cargar plantillas. En algunos casos, sin embargo, es útil disponer de múltiples entornos lado a lado, si estás usando distintas configuraciones.

La forma más sencilla de configurar *Twig* para cargar plantillas para tu aplicación se ve más o menos así:

```
require_once '/ruta/a/lib/Twig/Autoloader.php';
Twig_Autoloader::register();

$loader = new Twig_Loader_Filesystem('/ruta/a/templates');
$twig = new Twig_Environment($loader, array(
    'cache' => '/ruta/a/compilation_cache',
));
```

Esto creará un entorno de plantillas con la configuración predeterminada y un cargador que busca las plantillas en el directorio `/ruta/a/templates/`. Hay diferentes cargadores disponibles y también puedes escribir el tuyo si deseas cargar plantillas de una base de datos u otros recursos.

Nota: Ten en cuenta que el segundo argumento del entorno es una matriz de opciones. La opción `cache` es un directorio de caché de compilación, donde *Twig* memoriza las plantillas compiladas para evitar la fase de análisis de las subsiguientes peticiones. Esta es muy diferente de la caché que posiblemente desees agregar para evaluar plantillas. Para tal necesidad, puedes utilizar cualquier biblioteca de caché *PHP* disponible.

Para cargar una plantilla desde este entorno sólo tienes que llamar al método `LoadTemplate()` el cual devuelve una instancia de `Twig_Template`:

```
$template = $twig->loadTemplate('index.html');
```

Para reproducir la plantilla con algunas variables, llama al método `render()`:

```
echo $template->render(array('the' => 'variables', 'go' => 'here'));
```

Nota: El método `display()` es un atajo para reproducir la plantilla directamente.

También puedes exponer los métodos de extensión como funciones en tus plantillas:

```
echo $twig->render('index.html', array('the' => 'variables', 'go' => 'here'));
```

3.2 Opciones del entorno

Al crear una nueva instancia de `Twig_Environment`, puedes pasar una matriz de opciones como segundo argumento del constructor:

```
$twig = new Twig_Environment($loader, array('debug' => true));
```

Las siguientes opciones están disponibles:

- `debug`: Cuando se establece en `true`, las plantillas generadas tienen un método `__toString()` que puedes utilizar para mostrar los nodos generados (el predeterminado es `false`).
- `charset`: El juego de caracteres usado por las plantillas (por omisión es `utf-8`).
- `base_template_class`: La clase de plantilla base utilizada para generar plantillas (por omisión `Twig_Template`).
- `cache`: Una ruta absoluta donde almacenar las plantillas compiladas, o `false` para desactivar el almacenamiento en caché (el cual es el valor predeterminado).
- `auto_reload`: Cuando desarrollas con *Twig*, es útil volver a compilar la plantilla cada vez que el código fuente cambia. Si no proporcionas un valor para la opción `auto_reload`, se determinará automáticamente en función del valor `debug`.
- `strict_variables`: Si se establece en `false`, *Twig* ignorará silenciosamente las variables no válidas (variables y/o atributos/métodos que no existen) y los reemplazará con un valor `null`. Cuando se establece en `true`, *Twig* produce una excepción en su lugar (el predeterminado es `false`).
- `autoescape`: Si se establece en `true`, el escape automático será habilitado de manera predeterminada para todas las plantillas (por omisión a `true`).
- `optimizations`: Una marca que indica cuales optimizaciones aplicar (por omisión a `-1` – todas las optimizaciones están habilitadas; para desactivarla ponla a `0`).

3.3 Cargadores

Los cargadores son responsables de cargar las plantillas desde un recurso como el sistema de archivos.

3.3.1 Caché de compilación

Todos los cargadores de plantillas en cache pueden compilar plantillas en el sistema de archivos para su futura reutilización. Esto acelera mucho cómo se compilan las plantillas *Twig* una sola vez; y el aumento del rendimiento es

aún mayor si utilizas un acelerador *PHP* como *APC*. Consulta las opciones anteriores `cache` y `auto_reload` de `Twig_Environment` para más información.

3.3.2 Cargadores integrados

Aquí está una lista de los cargadores incorporados de que dispone *Twig*:

- `Twig_Loader_Filesystem`: Carga las plantillas desde el sistema de archivos. Este cargador puede encontrar plantillas en los directorios del sistema de archivos y es la manera preferida de cargarlas:

  ```
  $loader = new Twig_Loader_Filesystem($templateDir);
  ```

  También puedes buscar plantillas en una matriz de directorios:

  ```
  $loader = new Twig_Loader_Filesystem(array($templateDir1, $templateDir2));
  ```

  Con esta configuración, *Twig* buscará primero las plantillas de `$templateDir1` y si no existen, regresará a buscar en `$templateDir2`.

- `Twig_Loader_String`: Carga plantillas desde una cadena. Es un cargador silencioso que va cargando el código fuente directamente a medida que se lo vas pasando:

  ```
  $loader = new Twig_Loader_String();
  ```

- `Twig_Loader_Array`: Carga una plantilla desde una matriz *PHP*. Se le pasa una matriz de cadenas vinculadas a los nombres de plantilla. Este cargador es útil para pruebas unitarias:

  ```
  $loader = new Twig_Loader_Array($templates);
  ```

Truco: Cuando utilices los cargadores de `matriz` o `cadena` con un mecanismo de caché, debes saber que se genera una nueva clave de caché cada vez que "cambia" el contenido de una plantilla (la clave de caché es el código fuente de la plantilla). Si no deseas ver que tu caché crezca fuera de control, es necesario tener cuidado de limpiar el archivo de caché antiguo en sí mismo.

3.3.3 Creando tu propio cargador

Todos los cargadores implementan la interfaz `Twig_LoaderInterface`:

```
interface Twig_LoaderInterface
{
    /**
     * Obtiene el código fuente de una plantilla, del nombre dado.
     *
     * @param  string $name cadena del nombre de la plantilla a cargar
     *
     * @return string The template source code
     */
    function getSource($name);

    /**
     * Obtiene la clave de la caché para usarla en un nombre de plantilla dado.
     *
     * @param  string $name cadena del nombre de la plantilla a cargar
     *
     * @return string La clave de caché
     */
```

```
    function getCacheKey($name);

    /**
     * Devuelve true si la plantilla aún está fresca.
     *
     * @param string    $name El nombre de la plantilla
     * @param timestamp $time Hora de la última modificación de la plantilla en caché
     */
    function isFresh($name, $time);
}
```

A modo de ejemplo, esto es lo que dice el `Twig_Loader_String` incorporado:

```
class Twig_Loader_String implements Twig_LoaderInterface
{
    public function getSource($name)
    {
      return $name;
    }

    public function getCacheKey($name)
    {
      return $name;
    }

    public function isFresh($name, $time)
    {
      return false;
    }
}
```

El método `isFresh()` debe devolver `true` si la plantilla actual en caché aún es fresca, dado el tiempo de la última modificación, o `false` de lo contrario.

3.4 Usando extensiones

Las extensiones *Twig* son paquetes que añaden nuevas características a *Twig*. Usar una extensión es tan simple como usar el método `addExtension()`:

```
$twig->addExtension(new Twig_Extension_Sandbox());
```

Twig viene con las siguientes extensiones:

- *Twig_Extension_Core*: Define todas las características básicas de *Twig*.
- *Twig_Extension_Escaper*: Agrega escape automático y la posibilidad de escapar/no escapar bloques de código.
- *Twig_Extension_Sandbox*: Agrega un modo de recinto de seguridad para el entorno predeterminado de *Twig*, en el cual es seguro evaluar código que no es de confianza.
- *Twig_Extension_Optimizer*: Optimiza el nodo del árbol antes de la compilación.

El núcleo, las extensiones del mecanismo de escape y optimización no es necesario añadirlas al entorno *Twig*, debido a que se registran de forma predeterminada. Puedes desactivar una extensión registrada:

```
$twig->removeExtension('escaper');
```

3.5 Extensiones incorporadas

Esta sección describe las características agregadas por las extensiones incorporadas.

Truco: Lee el capítulo sobre la ampliación de *Twig* para que veas cómo crear tus propias extensiones.

3.5.1 Extensión `core`

La extensión `core` define todas las características principales de *Twig*:

- Etiquetas;
  - `for`
  - `if`
  - `extends`
  - `include`
  - `block`
  - `filter`
  - `macro`
  - `import`
  - `from`
  - `set`
  - `spaceless`
- Filtros:
  - `date`
  - `format`
  - `replace`
  - `url_encode`
  - `json_encode`
  - `title`
  - `capitalize`
  - `upper`
  - `lower`
  - `striptags`
  - `join`
  - `reverse`
  - `length`
  - `sort`
  - `merge`

        - `default`
        - `keys`
        - `escape`
        - `e`
    - Funciones:
        - `range`
        - `constant`
        - `cycle`
        - `parent`
        - `block`
    - Pruebas:
        - `even`
        - `odd`
        - `defined`
        - `sameas`
        - `null`
        - `divisibleby`
        - `constant`
        - `empty`

3.5.2 Extensión `escaper`

La extensión `escaper` añade a *Twig* el escape automático de la salida. Esta define una nueva etiqueta, `autoescape`, y un nuevo filtro, `raw`.

Al crear la extensión `escaper`, puedes activar o desactivar la estrategia de escape global de la salida:

```
$escaper = new Twig_Extension_Escaper(true);
$twig->addExtension($escaper);
```

Si se establece en `true`, se escapan todas las variables en las plantillas, excepto las que utilizan el filtro `raw`:

```
{{ article.to_html|raw }}
```

También puedes cambiar el modo de escape a nivel local usando la etiqueta `autoescape`:

```
{% autoescape true %}
  {% var %}
  {% var|raw %}      {# var no es escapada #}
  {% var|escape %}   {# var no se escapa doblemente #}
{% endautoescape %}
```

Advertencia: La etiqueta `autoescape` no tiene ningún efecto sobre los archivos incluidos.

Las reglas de escape se implementan de la siguiente manera:

- Literales (enteros, booleanos, matrices, ...) utilizados en la plantilla directamente como variables o argumentos de filtros no son escapados automáticamente:

```
{{ "Twig<br />" }} {# no es escapada #}

{% set text = "Twig<br />" %}
{{ text }} {# será escapado #}
```

- Expresiones cuyo resultado siempre es un literal o una variable marcada como segura nunca serán escapadas automáticamente:

```
{{ foo ? "Twig<br />" : "<br />Twig" }} {# no es escapada #}

{% set text = "Twig<br />" %}
{{ foo ? text : "<br />Twig" }} {# será escapado #}

{% set text = "Twig<br />" %}
{{ foo ? text|raw : "<br />Twig" }} {# no es escapada #}

{% set text = "Twig<br />" %}
{{ foo ? text|escape : "<br />Twig" }} {# el resultado de la expresión no será escapado #}
```

- El escape se aplica antes de la impresión, después de haber aplicado cualquier otro filtro:

```
{{ var|upper }} {# is equivalent to {{ var|upper|escape }} #}
```

- El filtro `raw` sólo se debe utilizar al final de la cadena de filtros:

```
{{ var|raw|upper }} {# será escapado #}

{{ var|upper|raw }} {# no es escapada #}
```

- No se aplica el escape automático si el último filtro de la cadena está marcado como seguro para el contexto actual (por ejemplo, `html` o `js`). `escaper` y `escaper('html')` están marcados como seguros para *html*, `escaper('js')` está marcado como seguro para *javascript*, `raw` está marcado como seguro para todo.

```
{% autoescape true js %}
{{ var|escape('html') }} {# será escapado para html y javascript #}
{{ var }} {# será escapado para javascript #}
{{ var|escape('js') }} {# won't be double-escaped #}
{% endautoescape %}
```

Nota: Ten en cuenta que el escape automático tiene algunas limitaciones puesto que el escapado se aplica en las expresiones después de su evaluación. Por ejemplo, cuando trabajas en concatenación, `{{foo|raw ~ bar }}` no dará el resultado esperado ya que el escape se aplica sobre el resultado de la concatenación y no en las variables individuales (por lo tanto aquí, el filtro `raw` no tendrá ningún efecto).

3.5.3 Extensión `sandbox`

La extensión `sandbox` se puede utilizar para evaluar código no confiable. El acceso a los atributos y los métodos inseguros está prohibido. El entorno recinto de seguridad es manejado por una política de la instancia. Por omisión, *Twig* viene con una política de clase: `Twig_Sandbox_SecurityPolicy`. Esta clase te permite agregar a la lista blanca algunas etiquetas, filtros, propiedades y métodos:

```
$tags = array('if');
$filters = array('upper');
```

```
$methods = array(
    'Article' => array('getTitle', 'getBody'),
);
$properties = array(
    'Article' => array('title', 'body'),
);
$functions = array('range');
$policy = new Twig_Sandbox_SecurityPolicy($tags, $filters, $methods, $properties, $functions);
```

Con la configuración anterior, la política de seguridad sólo te permitirá usar los filtros `if`, `tag` y `upper`. Por otra parte, las plantillas sólo podrán llamar a los métodos `getTitle()` y `getBody()` en objetos `Article`, y a las propiedades públicas `title` y `body`. Todo lo demás no está permitido y se generará una excepción `Twig_Sandbox_SecurityError`.

El objeto política es el primer argumento del constructor del recinto de seguridad:

```
$sandbox = new Twig_Extension_Sandbox($policy);
$twig->addExtension($sandbox);
```

De forma predeterminada, el modo de recinto de seguridad está desactivado y se activa cuando se incluye código de plantilla que no es de confianza usando la etiqueta `sandbox`:

```
{% sandbox %}
    {% include 'user.html' %}
{% endsandbox %}
```

Puedes poner todas las plantillas en el recinto de seguridad pasando `true` como segundo argumento al constructor de la extensión:

```
$sandbox = new Twig_Extension_Sandbox($policy, true);
```

3.5.4 Extensión `optimizer`

La extensión `optimizer` optimiza el nodo del árbol antes de compilarlo:

```
$twig->addExtension(new Twig_Extension_Optimizer());
```

Por omisión, todas las optimizaciones están activadas. Puedes seleccionar las que desees habilitar pasándolas al constructor:

```
$optimizer = new Twig_Extension_Optimizer(Twig_NodeVisitor_Optimizer::OPTIMIZE_FOR);

$twig->addExtension($optimizer);
```

3.6 Excepciones

Twig puede lanzar excepciones:

- `Twig_Error`: La excepción base para todos los errores.
- `Twig_Error_Syntax`: Lanzada para indicar al usuario que hay un problema con la sintaxis de la plantilla.
- `Twig_Error_Runtime`: Lanzada cuando se produce un error en tiempo de ejecución (cuando un filtro no existe, por ejemplo).
- `Twig_Error_Loader`: Se lanza al producirse un error durante la carga de la plantilla.

- `Twig_Sandbox_SecurityError`: Lanzada cuando aparece una etiqueta, filtro, o se llama a un método no permitido en una plantilla de un recinto de seguridad.

CAPÍTULO 4

Extendiendo *Twig*

Twig se puede extender en muchos aspectos; puedes añadir etiquetas adicionales, filtros, pruebas, operadores, variables globales y funciones. Incluso puedes extender el propio analizador con visitantes de nodo.

Nota: Este capítulo describe cómo extender *Twig* fácilmente. Si deseas reutilizar tus cambios en diferentes proyectos o si quieres compartirlos con los demás, entonces, debes crear una extensión tal como se describe en el siguiente capítulo.

Antes de extender *Twig*, debes entender las diferencias entre todos los diferentes puntos de extensión posibles y cuándo utilizarlos.

En primer lugar, recuerda que el lenguaje de *Twig* tiene dos construcciones principales:

- `{{ }}`: Utilizada para imprimir el resultado de la evaluación de la expresión;
- `{% %}`: Utilizada para ejecutar declaraciones.

Para entender por qué *Twig* expone tantos puntos de extensión, vamos a ver cómo implementar un generador *Lorem ipsum* (este necesita saber el número de palabras a generar).

Puedes utilizar una *etiqueta* `Lipsum`:

```
{% lipsum 40 %}
```

Eso funciona, pero usar una etiqueta para `lipsum` no es una buena idea por al menos tres razones principales:

- `lipsum` no es una construcción del lenguaje;
- La etiqueta produce algo;
- La etiqueta no es flexible ya que no la puedes utilizar en una expresión:

  ```
  {{ 'some text' ~ {% lipsum 40 %} ~ 'some more text' }}
  ```

De hecho, rara vez es necesario crear etiquetas; y es una muy buena noticia porque las etiquetas son el punto de extensión más complejo de *Twig*.

Ahora, vamos a utilizar un *filtro* `lipsum`:

```
{{ 40|lipsum }}
```

Una vez más, funciona, pero se ve raro. Un filtro transforma el valor que se le pasa a alguna otra cosa, pero aquí utilizamos el valor para indicar el número de palabras a generar.

En seguida, vamos a utilizar una *función* `lipsum`:

```
{{ lipsum(40) }}
```

Aquí vamos. Para este ejemplo concreto, la creación de una función es el punto de extensión a usar. Y la puedes usar en cualquier lugar en que se acepte una expresión:

```
{{ 'algún texto' ~ lipsum(40) ~ 'algo más de texto' }}

{% set lipsum = lipsum(40) %}
```

Por último pero no menos importante, también puedes utilizar un objeto *global* con un método capaz de generar texto *Lorem Ipsum*:

```
{{ text.lipsum(40) }}
```

Como regla general, utiliza funciones para las características más utilizadas y objetos globales para todo lo demás.

Ten en cuenta lo siguiente cuando desees extender *Twig*:

¿Qué?	¿dificultad para implementación?	¿Con qué frecuencia?	¿Cuándo?
macro	trivial	frecuente	Generación de contenido
global	trivial	frecuente	Objeto ayudante
function	trivial	frecuente	Generación de contenido
filter	trivial	frecuente	Transformación de valor
tag	complejo	raro	Constructor del lenguaje *DSL*
test	trivial	raro	Decisión booleana
operator	trivial	raro	Transformación de valores

4.1 Globales

Una variable global es como cualquier otra variable de plantilla, excepto que está disponible en todas las plantillas y macros:

```
$twig = new Twig_Environment($loader);
$twig->addGlobal('text', new Text());
```

Entonces puedes utilizar la variable `text` en cualquier parte de una plantilla:

```
{{ text.lipsum(40) }}
```

4.2 Filtros

Un filtro es una función *PHP* regular o un método de objeto que toma el lado izquierdo del filtro (antes del tubo `|`) como primer argumento y los argumentos adicionales pasados al filtro (entre paréntesis `()`) como argumentos adicionales.

La definición de un filtro es tan fácil como asociar el nombre del filtro con un ejecutable de *PHP*. Por ejemplo, digamos que tienes el siguiente código en una plantilla:

```
{{ 'TWIG'|lower }}
```

Al compilar esta plantilla para *PHP*, *Twig* busca el ejecutable *PHP* asociado con el filtro `lower`. El filtro `lower` es un filtro integrado en *Twig*, y simplemente se asigna a la función *PHP* `strtolower()`. Después de la compilación, el código generado por *PHP* es más o menos equivalente a:

```
<?php echo strtolower('TWIG') ?>
```

Como puedes ver, la cadena `'TWIG'` se pasa como primer argumento a la función de *PHP*.

Un filtro también puede tomar argumentos adicionales como en el siguiente ejemplo:

```
{{ now|date('d/m/Y') }}
```

En este caso, los argumentos adicionales son pasados a la función después del argumento principal, y el código compilado es equivalente a:

```
<?php echo twig_date_format_filter($now, 'd/m/Y') ?>
```

Vamos a ver cómo crear un nuevo filtro.

En esta sección, vamos a crear un filtro `rot13`, el cual debe devolver la transformación rot13 de una cadena. Aquí está un ejemplo de su uso y los resultados esperados:

```
{{ "Twig"|rot13 }}

{# debería mostrar Gjvt #}
```

Agregar un filtro es tan sencillo como llamar al método `addFilter()` en la instancia de `Twig_Environment`:

```
$twig = new Twig_Environment($loader);
$twig->addFilter('rot13', new Twig_Filter_Function('str_rot13'));
```

El segundo argumento de `addFilter()` es una instancia de `Twig_Filter`. Aquí, utilizamos `Twig_Filter_Function` puesto que el filtro es una función *PHP*. El primer argumento pasado al constructor `Twig_Filter_Function` es el nombre de la función *PHP* a llamar, aquí `str_rot13`, una función nativa de *PHP*.

Digamos que ahora deseas poder añadir un prefijo antes de la cadena convertida:

```
{{ "Twig"|rot13('prefijo_') }}

{# debe mostrar prefijo_Gjvt #}
```

Como la función `str_rot13()` de *PHP* no es compatible con este requisito, vamos a crear una nueva función *PHP*:

```
function project_compute_rot13($string, $prefix = '')
{
    return $prefix.str_rot13($string);
}
```

Como puedes ver, el argumento `prefix` del filtro se pasa como un argumento adicional a la función `project_compute_rot13()`.

La adición de este filtro es tan fácil como antes:

```
$twig->addFilter('rot13', new Twig_Filter_Function('project_compute_rot13'));
```

Para una mejor encapsulación, también puedes definir un filtro como un método estático de una clase. También puedes utilizar la clase `Twig_Filter_Function` para registrar métodos estáticos, tal como filtros:

```
$twig->addFilter('rot13', new Twig_Filter_Function('SomeClass::rot13Filter'));
```

Truco: En una extensión, también puedes definir un filtro como un método estático de la clase extendida.

4.2.1 Entorno consciente de filtros

La clase `Twig_Filter` toma opciones como su último argumento. Por ejemplo, si deseas acceder a la instancia del entorno actual en tu filtro, establece la opción `needs_environment` a `true`:

```
$filter = new Twig_Filter_Function('str_rot13',
                                   array(
                                         'needs_environment' => true
                                        ));
```

Twig entonces pasará el entorno actual como primer argumento al invocar el filtro:

```
function twig_compute_rot13(Twig_Environment $env, $string)
{
    // obtiene el juego de caracteres actual, por ejemplo
    $charset = $env->getCharset();

    return str_rot13($string);
}
```

4.2.2 Escapando automáticamente

Si está habilitado el escape automático, puedes escapar la salida del filtro antes de imprimir. Si tu filtro actúa como un escapista (o explícitamente produce código *html* o *javascript*), desearás que se imprima la salida cruda. En tal caso, establece la opción `is_safe`:

```
$filter = new Twig_Filter_Function('nl2br',
                                   array('is_safe' => array('html')
                                  ));
```

Algunos filtros posiblemente tengan que trabajar en valores ya escapados o seguros. En tal caso, establece la opción `pre_escape`:

```
$filter = new Twig_Filter_Function('somefilter',
                                   array('pre_escape' => 'html',
                                         'is_safe' => array('html')
                                        ));
```

4.3 Funciones

Una función es una función *PHP* regular o un método de objeto que puedes llamar desde las plantillas.

```
{{ constant("DATE_W3C") }}
```

Al compilar esta plantilla para *PHP*, *Twig* busca el *PHP* ejecutable asociado con la función `constant`. La función `constant` está integrada en la función *Twig*, y simplemente asignada a la función `constant()` de *PHP*. Después de la compilación, el código generado por *PHP* es más o menos equivalente a:

```
<?php echo constant('DATE_W3C') ?>
```

Agregar una función es similar a agregar un filtro. Esto se puede hacer llamando al método `addFunction()` en la instancia de `Twig_Environment`:

```
$twig = new Twig_Environment($loader);
$twig->addFunction('functionName', new Twig_Function_Function('someFunction'));
```

También puedes exponer los métodos de extensión como funciones en tus plantillas:

```
// $this es un objeto que implementa a Twig_ExtensionInterface.
$twig = new Twig_Environment($loader);
$twig->addFunction('otherFunction', new Twig_Function_Method($this, 'someMethod'));
```

Las funciones también son compatibles con los parámetros `needs_environment` e `is_safe`.

4.4 Etiquetas

Una de las características más interesantes de un motor de plantillas como *Twig* es la posibilidad de definir nuevas construcciones del lenguaje. Esta también es la característica más compleja que necesitas comprender de cómo trabaja *Twig* internamente.

Vamos a crear una simple etiqueta `set` que te permita definir variables simples dentro de una plantilla. Puedes utilizar la etiqueta de la siguiente manera:

```
{% set name = "value" %}

{{ name }}

{# debe producir value #}
```

Nota: La etiqueta `set` es parte de la extensión `core` y como tal siempre está disponible. La versión integrada es un poco más potente y de manera predeterminada es compatible con múltiples asignaciones (consulta el capítulo *Twig para diseñadores de plantillas* (Página 3) para más información).

---

para definir una nueva etiqueta son necesarios tres pasos:

- Definir una clase para analizar segmentos (responsable de analizar el código de la plantilla);
- Definir una clase Nodo (responsable de convertir el código analizado a *PHP*);
- Registrar la etiqueta.

4.4.1 Registrando una nueva etiqueta

Agregar una etiqueta es tan simple como una llamada al método `addTokenParser` en la instancia de `Twig_Environment`:

```
$twig = new Twig_Environment($loader);
$twig->addTokenParser(new Project_Set_TokenParser());
```

4.4.2 Definiendo un analizador de fragmentos

Ahora, vamos a ver el código real de esta clase:

```
class Project_Set_TokenParser extends Twig_TokenParser
{
    public function parse(Twig_Token $token)
    {
        $lineno = $token->getLine();
        $name = $this->parser->getStream()->expect(Twig_Token::NAME_TYPE)->getValue();
        $this->parser->getStream()->expect(Twig_Token::OPERATOR_TYPE, '=');
```

```
        $value = $this->parser->getExpressionParser()->parseExpression();

        $this->parser->getStream()->expect(Twig_Token::BLOCK_END_TYPE);

        return new Project_Set_Node($name, $value, $lineno, $this->getTag());
        }

    public function getTag()
    {
        return 'set';
    }
}
```

El método `getTag()` debe devolver la etiqueta que queremos analizar, aquí `set`.

El método `parse()` se invoca cada vez que el analizador encuentra una etiqueta `set`. Este debe devolver una instancia de `Twig_Node` que representa el nodo (la llamada para la creación del `Project_Set_Node` se explica en la siguiente sección).

El proceso de análisis se simplifica gracias a un montón de métodos que se pueden llamar desde el flujo del segmento (`$this->parser->getStream()`):

- `getCurrent()`: Obtiene el segmento actual del flujo.
- `next()`: Mueve al siguiente segmento en la secuencia, *pero devuelve el antiguo*.
- `test($type)`, `test($value)` o `test($type, $value)`: Determina si el segmento actual es de un tipo o valor particular (o ambos). El valor puede ser una matriz de varios posibles valores.
- `expect($type[, $value[, $message]])`: Si el segmento actual no es del tipo/valor dado lanza un error de sintaxis. De lo contrario, si el tipo y valor son correctos, devuelve el segmento y mueve el flujo al siguiente segmento.
- `look()`: Busca el siguiente segmento sin consumirlo.

Las expresiones de análisis se llevan a cabo llamando a `parseExpression()` como lo hicimos para la etiqueta `set`.

Truco: Leer las clases `TokenParser` existentes es la mejor manera de aprender todos los detalles esenciales del proceso de análisis.

4.4.3 Definiendo un nodo

La clase `Project_Set_Node` en sí misma es bastante simple:

```
class Project_Set_Node extends Twig_Node
{
    public function __construct($name, Twig_Node_Expression $value, $lineno)
    {
        parent::__construct(array('value' => $value), array('name' => $name), $lineno);
        }

    public function compile(Twig_Compiler $compiler)
    {
        $compiler
            ->addDebugInfo($this)
            ->write('$context[\''.$this->getAttribute('name').'\'] = ')
            ->subcompile($this->getNode('value'))
```

```
            ->raw(";\n")
        ;
    }
}
```

El compilador implementa una interfaz fluida y proporciona métodos que ayudan a los desarrolladores a generar código *PHP* hermoso y fácil de leer:

- `subcompile()`: Compila un nodo.
- `raw()`: Escribe la cadena dada tal cual.
- `write()`: Escribe la cadena dada añadiendo sangría al principio de cada línea.
- `string()`: Escribe una cadena entre comillas.
- `repr()`: Escribe una representación *PHP* de un valor dado (consulta `Twig_Node_For` para un ejemplo real).
- `addDebugInfo()`: Agrega como comentario la línea del archivo de plantilla original relacionado con el nodo actual.
- `indent()`: Aplica sangrías el código generado (consulta `Twig_Node_Block` para un ejemplo real).
- `outdent()`: Quita la sangría el código generado (consulta `Twig_Node_Block` para un ejemplo real).

CAPÍTULO 5

Creando una extensión *Twig*

La principal motivación para escribir una extensión es mover el código usado frecuentemente a una clase reutilizable como agregar apoyo para la internacionalización. Una extensión puede definir etiquetas, filtros, pruebas, operadores, variables globales, funciones y visitantes de nodo.

La creación de una extensión también hace una mejor separación del código que se ejecuta en tiempo de compilación y el código necesario en tiempo de ejecución. Como tal, esto hace tu código más rápido.

La mayoría de las veces, es útil crear una extensión para tu proyecto, para acoger todas las etiquetas y filtros específicos que deseas agregar a *Twig*.

Nota: Antes de escribir tus propias extensiones, echa un vistazo al repositorio de extensiones oficial de *Twig*: http://github.com/fabpot/Twig-extensions.

Una extensión es una clase que implementa la siguiente interfaz:

```
interface Twig_ExtensionInterface
{
    /**
     * Inicia el entorno en tiempo de ejecución.
     *
     * Aquí es donde puedes cargar algún archivo que contenga funciones de filtro, por ejemplo.
     *
     * @param Twig_Environment $environment La instancia actual de Twig_Environment
     */
    function initRuntime(Twig_Environment $environment);

    /**
     * Devuelve instancias del analizador de segmentos para añadir a la lista existente.
     *
     * @return array Un arreglo de instancias Twig_TokenParserInterface o Twig_TokenParserBrokerInter
     */
    function getTokenParsers();

    /**
     * Devuelve instancias del visitante de nodos para añadirlas a la lista existente.
     *
     * @return array Un arreglo de instancias de Twig_NodeVisitorInterface
     */
    function getNodeVisitors();
```

```
    /**
     * Devuelve una lista de filtros para añadirla a la lista existente.
     *
     * @return array Un arreglo de filtros
     */
    function getFilters();

    /**
     * Devuelve una lista de pruebas para añadirla a la lista existente.
     *
     * @return array Un arreglo de pruebas
     */
    function getTests();

    /**
     * Devuelve una lista de funciones para añadirla a la lista existente.
     *
     * @return array Un arreglo de funciones
     */
    function getFunctions();

    /**
     * Devuelve una lista de operadores para añadirla a la lista existente.
     *
     * @return array Un arreglo de operadores
     */
    function getOperators();

    /**
     * Devuelve una lista de variables globales para añadirla a la lista existente.
     *
     * @return array Un arreglo de variables globales
     */
    function getGlobals();

    /**
     * Devuelve el nombre de la extensión.
     *
     * @return string El nombre de la extensión
     */
    function getName();
}
```

Para mantener tu clase de extensión limpia y ordenada, puedes heredar de la clase `Twig_Extension` incorporada en lugar de implementar toda la interfaz. De esta forma, sólo tienes que implementar el método `getName()` como el que proporcionan las implementaciones vacías de `Twig_Extension` para todos los otros métodos.

El método `getName()` debe devolver un identificador único para tu extensión.

Ahora, con esta información en mente, vamos a crear la extensión más básica posible:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getName()
    {
        return 'project';
    }
}
```

Nota: Por supuesto, esta extensión no hace nada por ahora. Vamos a personalizarla en las siguientes secciones.

A *Twig* no le importa dónde guardas tu extensión en el sistema de archivos, puesto que todas las extensiones se deben registrar explícitamente para estar disponibles en tus plantillas.

Puedes registrar una extensión con el método `addExtension()` en tu objeto `Environment` principal:

```
$twig = new Twig_Environment($loader);
$twig->addExtension(new Project_Twig_Extension());
```

Por supuesto, tienes que cargar primero el archivo de la extensión, ya sea utilizando `require_once()` o con un cargador automático (consulta la sección spl_autoload_register()).

Truco: Las extensiones integradas son grandes ejemplos de cómo trabajan las extensiones.

5.1 Globales

Puedes registrar las variables globales en una extensión vía el método `getGlobals()`:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getGlobals()
    {
        return array(
            'text' => new Text(),
        );
    }

    // ...
}
```

5.2 Funciones

Puedes registrar funciones en una extensión vía el método `getFunctions()`:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getFunctions()
    {
        return array(
            'lipsum' => new Twig_Function_Function('generate_lipsum'),
        );
    }

    // ...
}
```

5.3 Filtros

Para agregar un filtro a una extensión, es necesario sustituir el método `getFilters()`. Este método debe devolver una matriz de filtros para añadir al entorno *Twig*:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getFilters()
    {
        return array(
            'rot13' => new Twig_Filter_Function('str_rot13'),
        );
    }

    // ...
}
```

Como puedes ver en el código anterior, el método `getFilters()` devuelve una matriz donde las claves son el nombre de los filtros (`rot13`) y los valores de la definición del filtro (`new Twig_Filter_Function('str_rot13')`).

Como vimos en el capítulo anterior, también puedes definir filtros como métodos estáticos en la clase de la extensión:

```
$twig->addFilter('rot13', new Twig_Filter_Function('Project_Twig_Extension::rot13Filter'));
```

También puedes utilizar `Twig_Filter_Method` en lugar de `Twig_Filter_Function` cuando definas un filtro que usa un método:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getFilters()
    {
        return array(
            'rot13' => new Twig_Filter_Method($this, 'rot13Filter'),
        );
    }

    public function rot13Filter($string)
    {
        return str_rot13($string);
    }

    // ...
}
```

El primer argumento del constructor de `Twig_Filter_Method` siempre es `$this`, el objeto extensión actual. El segundo es el nombre del método a llamar.

Usar métodos de filtro es una gran manera de empaquetar el filtro sin contaminar el espacio de nombres global. Esto también le da más flexibilidad al desarrollador a costa de una pequeña sobrecarga.

5.3.1 Sustituyendo los filtros predeterminados

Si algunos filtros predeterminados del núcleo no se ajustan a tus necesidades, fácilmente puedes sustituirlos creando tu propia extensión del núcleo. Por supuesto, no es necesario copiar y pegar el código del núcleo en toda tu extensión de *Twig*. En lugar de eso la puedes extender y sustituir los filtros que deseas reemplazando el método `getFilters()`:

```
class MyCoreExtension extends Twig_Extension_Core
{
    public function getFilters()
    {
        return array_merge(parent::getFilters(), array(
            'date' => new Twig_Filter_Method($this, 'dateFilter'),
            // ...
        ));
    }

    public function dateFilter($timestamp, $format = 'F j, Y H:i')
    {
        return '...'.twig_date_format_filter($timestamp, $format);
    }

    // ...
}
```

Aquí, reemplazamos el filtro `date` con uno personalizado. Usar esta nueva extensión del núcleo es tan simple como registrar la extensión `MyCoreExtension` llamando al método `addExtension()` en la instancia del entorno:

```
$twig = new Twig_Environment($loader);
$twig->addExtension(new MyCoreExtension());
```

Pero ya puedo escuchar a algunas personas preguntando cómo pueden hacer que la extensión del núcleo se cargue por omisión. Eso es cierto, pero el truco es que ambas extensiones comparten el mismo identificador único (`core` - definido en el método `getName()`). Al registrar una extensión con el mismo nombre que una ya existente, realmente sustituyes la predeterminada, incluso si ya está registrada:

```
$twig->addExtension(new Twig_Extension_Core());
$twig->addExtension(new MyCoreExtension());
```

5.4 Etiquetas

Puedes agregar una etiqueta en una extensión reemplazando el método `getTokenParsers()`. Este método debe devolver una matriz de etiquetas para añadir al entorno *Twig*:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getTokenParsers()
    {
        return array(new Project_Set_TokenParser());
    }

    // ...
}
```

En el código anterior, hemos añadido una sola etiqueta nueva, definida por la clase `Project_Set_TokenParser`. La clase `Project_Set_TokenParser` es responsable de analizar la etiqueta y compilarla a *PHP*.

5.5 Operadores

El método `getOperators()` te permite añadir nuevos operadores. Aquí tienes cómo añadir los operadores `!`, `||` y `&&`:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getOperators()
    {
        return array(array(
                            '!' => array('precedence'     => 50,
                                         'class'          => 'Twig_Node_Expression_Unary_Not'),
                        ),
                    array(
                            '||' => array('precedence'    => 10,
                                          'class'         => 'Twig_Node_Expression_Binary_Or',
                                          'associativity' => Twig_ExpressionParser::OPERATOR_LEFT),
                            '&&' => array('precedence'    => 15,
                                          'class'         => 'Twig_Node_Expression_Binary_And',
                                          'associativity' => Twig_ExpressionParser::OPERATOR_LEFT),
                        ),
                    );
    }

    // ...
}
```

5.6 Pruebas

El método `getTests()` te permite añadir funciones de prueba:

```
class Project_Twig_Extension extends Twig_Extension
{
    public function getTests()
    {
        return array(
            'even' => new Twig_Test_Function('twig_test_even'),
        );
    }

    // ...
}
```

CAPÍTULO 6

Mejorando *Twig*

Twig es muy extensible y lo puedes mejorar fácilmente. Ten en cuenta que probablemente deberías tratar de crear una extensión antes de sumergirte en el núcleo, puesto que la mayoría de las características y mejoras se pueden hacer con extensiones. Este capítulo también es útil para personas que quieren entender cómo funciona *Twig* debajo del capó.

6.1 ¿Cómo funciona *Twig*?

La reproducción de una plantilla *Twig* se puede resumir en cuatro pasos fundamentales:

- **Cargar** la plantilla: Si la plantilla ya está compilada, la carga y va al paso *evaluación*, de lo contrario:
  - En primer lugar, el **analizador léxico** reduce el código fuente de la plantilla a pequeñas piezas para facilitar su procesamiento;
  - A continuación, el **analizador** convierte el flujo del segmento en un árbol de nodos significativo (el árbol de sintaxis abstracta);
  - Eventualmente, el *compilador* transforma el árbol de sintaxis abstracta en código *PHP*;
- **Evaluar** la plantilla: Básicamente significa llamar al método `display()` de la plantilla compilada adjuntando el contexto.

6.2 El analizador léxico

El objetivo del analizador léxico de *Twig* es dividir el código fuente en un flujo de segmentos (donde cada segmento es de la clase `token`, y el flujo es una instancia de `Twig_TokenStream`). El analizador léxico predeterminado reconoce nueve diferentes tipos de segmentos:

- `Twig_Token::TEXT_TYPE`
- `Twig_Token::BLOCK_START_TYPE`
- `Twig_Token::VAR_START_TYPE`
- `Twig_Token::BLOCK_END_TYPE`
- `Twig_Token::VAR_END_TYPE`
- `Twig_Token::NAME_TYPE`

- Twig_Token::NUMBER_TYPE
- Twig_Token::STRING_TYPE
- Twig_Token::OPERATOR_TYPE
- Twig_Token::EOF_TYPE

Puedes convertir manualmente un código fuente en un flujo de segmentos llamando al método tokenize() de un entorno:

```
$stream = $twig->tokenize($source, $identifier);
```

Dado que el flujo tiene un método __toString(), puedes tener una representación textual del mismo haciendo eco del objeto:

```
echo $stream."\n";
```

Aquí está la salida para la plantilla Hello {{ name }}:

```
TEXT_TYPE(Hello )
VAR_START_TYPE()
NAME_TYPE(name)
VAR_END_TYPE()
EOF_TYPE()
```

Puedes cambiar el analizador léxico predeterminado que usa *Twig* (Twig_Lexer) llamando al método setLexer():

```
$twig->setLexer($lexer);
```

Las clases Lexer deben implementar a Twig_LexerInterface:

```
interface Twig_LexerInterface
{
    /**
     * Segmenta el código fuente.
     *
     * @param  string $code     El código fuente
     * @param  string $filename Un identificador único para el código fuente
     *
     * @return Twig_TokenStream Una muestra de la instancia del flujo
     */
    function tokenize($code, $filename = 'n/a');
}
```

6.3 El analizador sintáctico

El analizador convierte el flujo de segmentos en un ASA (árbol de sintaxis abstracta), o un árbol de nodos (de clase Twig_Node_Module). La extensión del núcleo define los nodos básicos como: for, if, ... y la expresión nodos.

Puedes convertir manualmente un flujo de segmentos en un nodo del árbol llamando al método parse() de un entorno:

```
$nodes = $twig->parse($stream);
```

Al hacer eco del objeto nodo te da una buena representación del árbol:

```
echo $nodes."\n";
```

Aquí está la salida para la plantilla `Hello {{ name }}`:

```
Twig_Node_Module(
  Twig_Node_Text(Hello )
  Twig_Node_Print(
    Twig_Node_Expression_Name(name)
  )
)
```

El analizador predeterminado (`Twig_TokenParser`) también se puede cambiar mediante una llamada al método `setParser()`:

```
$twig->setParser($analizador);
```

Todos los analizadores de *Twig* deben implementar a `Twig_ParserInterface`:

```
interface Twig_ParserInterface
{
    /**
     * Convierte un flujo de segmentos en un árbol de nodos.
     *
     * @param  Twig_TokenStream $stream Una instancia de una muestra del flujo
     *
     * @return Twig_Node_Module Un nodo del árbol
     */
    function parser(Twig_TokenStream $code);
}
```

6.4 El compilador

El último paso lo lleva a cabo el compilador. Este necesita un árbol de nodos como entrada y genera código *PHP* que se puede emplear para ejecutar las plantillas en tiempo de ejecución. El compilador predeterminado genera las clases *PHP* para facilitar la implementación de la herencia de plantillas.

Puedes llamar al compilador manualmente con el método `compile()` de un entorno:

```
$php = $twig->compile($nodes);
```

El método `compile()` devuelve el código fuente *PHP* que representa el nodo.

La plantilla generada por un patrón `Hello {{ name }}` es la siguiente:

```
/* Hello {{ name }} */
class __TwigTemplate_1121b6f109fe93ebe8c6e22e3712bceb extends Twig_Template
{
    public function display($context)
    {
        $this->env->initRuntime();

        // line 1
        echo "Hello ";
        echo (isset($context['name']) ? $context['name'] : null);
    }
}
```

En cuanto a los analizadores léxico y sintáctico, el compilador predeterminado (`Twig_Compiler`) se puede cambiar mediante una llamada al método `setCompiler()`:

```
$twig->setCompiler($compilador);
```

Todos los compiladores de *Twig* deben implementar a `Twig_CompilerInterface`:

```
interface Twig_CompilerInterface
{
    /**
     * Compila un nodo.
     *
     * @param  Twig_Node $node El nodo a compilar
     *
     * @return Twig_Compiler La instancia actual del compilador
     */
    function compile(Twig_Node $node);

    /**
     * Obtiene el código PHP actual después de la compilación.
     *
     * @return string The PHP code
     */
    function getSource();
}
```

CAPÍTULO 7

Recetas

7.1 Haciendo un diseño condicional

Trabajar con *Ajax* significa que el mismo contenido a veces se muestra tal cual, y a veces se decora con un diseño. Dado que el nombre del diseño de las plantillas *Twig* puede ser cualquier expresión válida, puedes pasar una variable que evalúe a `true` cuando se hace la petición a través de *Ajax* y elegir el diseño en consecuencia:

```
{% extends request.ajax ? "base_ajax.html" : "base.html" %}

{% block content %}
    Este es el contenido a mostrar.
{% endblock %}
```

7.2 Haciendo una inclusión dinámica

Cuando incluyes una plantilla, su nombre no tiene por qué ser una cadena. Por ejemplo, el nombre puede depender del valor de una variable:

```
{% include var ~ '_foo.html' %}
```

Si `var` evalúa como `index`, se reproducirá la plantilla `index_foo.html`.

De hecho, el nombre de la plantilla puede ser cualquier expresión válida, como la siguiente:

```
{% include var|default('index') ~ '_foo.html' %}
```

7.3 Sustituyendo una plantilla que además se extiende a sí misma

Puedes personalizar una plantilla de dos formas diferentes:

- *Herencia*: Una plantilla *extiende* a una plantilla padre y sustituye algunos bloques;
- *Sustitución*: Si utilizas el cargador del sistema de archivos, *Twig* carga la primera plantilla si se encuentra en una lista de directorios configurados; una plantilla que se encuentra en un directorio *sustituye* a otra de un directorio más en la lista.

Pero, ¿cómo se combinan las dos cosas?: *sustituir* una plantilla que también se extiende a sí misma (también conocida como una plantilla en un directorio más en la lista)

Digamos que tus plantillas se cargan tanto desde `.../templates/mysite` como de `.../templates/default`, en este orden. La plantilla `page.twig` almacenada en `.../templates/default` es la siguiente:

```
{# page.twig #}
{% extends "base.twig" %}

{% block content %}
{% endblock %}
```

Puedes sustituir esta plantilla poniendo un archivo con el mismo nombre en `.../templates/mysite`. Y si deseas ampliar la plantilla original, podrías tener la tentación de escribir lo siguiente:

```
{# page.twig in .../templates/mysite #}
{% extends "page.twig" %} {# from .../templates/default #}
```

Por supuesto, esto no funcionará debido a que *Twig* siempre carga la plantilla desde `.../templates/mysite`.

Resulta que es posible conseguir que esto funcione, añadiendo el directorio adecuado al final de tus directorios de plantilla, el cual es el padre de todos los otros directorios: `.../templates` en nuestro caso. Esto tiene el efecto de hacer que cada archivo de plantilla dentro de nuestro sistema sea direccionable unívocamente. La mayoría de las veces utilizarás rutas "normales", pero en el caso especial de querer extender una plantilla con una versión que se redefine a sí misma podemos referirnos a la ruta completa del padre, sin ambigüedades, en la etiqueta `extends` de la plantilla:

```
{# page.twig in .../templates/mysite #}
{% extends "default/page.twig" %} {# from .../templates #}
```

Nota: Esta receta está inspirada en la siguiente página del *wiki* de *Django*: http://code.djangoproject.com/wiki/ExtendingTemplates

---

7.4 Sintaxis personalizada

Twig te permite personalizar alguna sintaxis de los delimitadores de bloque. No se recomienda usar esta característica puesto que las plantillas serán vinculadas con tu sintaxis personalizada. Sin embargo, para proyectos específicos, puede tener sentido cambiar los valores predeterminados.

Para cambiar los delimitadores de bloque, necesitas crear tu propio objeto *lexer*:

```
$twig = new Twig_Environment();

$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('{#', '#}'),
    'tag_block'    => array('{%', '%}'),
    'tag_variable' => array('{{', '}}'),
));
$twig->setLexer($lexer);
```

Éstos son algunos ejemplos de configuración que simulan la sintaxis de algunos otros motores de plantilla:

```
// sintaxis erb de Ruby
$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('<%#', '%>'),
    'tag_block'    => array('<%', '%>'),
```

```
    'tag_variable' => array('<%=', '%>'),
));

// sintaxis de comentarios SGML
$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('<!--#', '-->'),
    'tag_block'    => array('<!--', '-->'),
    'tag_variable' => array('${', '}'),
));

// como Smarty
$lexer = new Twig_Lexer($twig, array(
    'tag_comment'  => array('{*', '*}'),
    'tag_block'    => array('{', '}'),
    'tag_variable' => array('{$', '}'),
));
```

7.5 Usando propiedades dinámicas de los objetos

Cuando *Twig* encuentra una variable como `articulo.titulo`, trata de encontrar una propiedad pública `titulo` en el objeto `articulo`.

También funciona si la propiedad no existe, pero más bien está definida de forma dinámica gracias a la magia del método `__get()`; sólo tienes que implementar también el método mágico `__isset()`, como muestra el siguiente fragmento de código:

```
class Article
{
    public function __get($name)
    {
        if ('title' == $name)
        {
            return 'The title';
        }

        // lanza algún tipo de error
    }

    public function __isset($name)
    {
        if ('title' == $name)
        {
            return true;
        }

        return false;
    }
}
```

7.6 Accediendo al contexto del padre en bucles anidados

A veces, cuando utilizas bucles anidados, necesitas acceder al contexto del padre. El contexto del padre siempre es accesible a través de la variable `loop.parent`. Por ejemplo, si tienes los siguientes datos de plantilla:

```
$datos = array(
    'temas' => array(
        'tema1' => array('Mensaje 1 del tema 1', 'Mensaje 2 del tema 1'),
        'tema2' => array('Mensaje 1 del tema 2', 'Mensaje 2 del tema 2'),
    ),
);
```

Y la siguiente plantilla para mostrar todos los mensajes en todos los temas:

```
{% for topic, messages in topics %}
    * {{ loop.index }}: {{ topic }}
  {% for message in messages %}
      - {{ loop.parent.loop.index }}.{{ loop.index }}: {{ message }}
  {% endfor %}
{% endfor %}
```

Reproducirá algo similar a:

```
* 1: topic1
  - 1.1: The message 1 of topic 1
  - 1.2: The message 2 of topic 1
* 2: topic2
  - 2.1: The message 1 of topic 2
  - 2.2: The message 2 of topic 2
```

En el bucle interno, utilizamos la variable `loop.parent` para acceder al contexto externo. Así, el índice del `tema` actual definido en el exterior del bucle es accesible a través de la variable `loop.parent.loop.index`.

7.7 Definiendo al vuelo funciones indefinidas y filtros

Cuando una función (o un filtro) no está definido, de manera predeterminada *Twig* lanza una excepción `Twig_Error_Syntax`. Sin embargo, también puede invocar una retrollamada (cualquier *PHP* válido que se pueda ejecutar) la cual debe devolver una función (o un filtro).

Para filtros, registra las retrollamadas con `registerUndefinedFilterCallback()`. Para funciones, usa `registerUndefinedFunctionCallback()`:

```
// Autoregistra todas las funciones nativas de PHP como funciones Twig
// no intentes esto en casa, ¡ya que no es seguro en absoluto!
$twig->registerUndefinedFunctionCallback(function ($name) {
    if (function_exists($name)) {
        return new Twig_Function_Function($name);
    }

    return false;
});
```

Si el ejecutable no es capaz de devolver una función válida (o filtro), deberá devolver `false`.

Si registras más de una retrollamada, *Twig* la llamará a su vez hasta que una no devuelva `false`.

Truco: Debido a que la resolución de funciones y filtros se realiza durante la compilación, no hay ninguna sobrecarga cuando registras estas retrollamadas.

7.8 Validando la sintaxis de la plantilla

Cuando el código de plantilla lo proporciona un tercero (a través de una interfaz web, por ejemplo), podría ser interesante validar la sintaxis de la plantilla antes de guardarla. Si el código de la plantilla se almacena en una variable `$template`, así es cómo lo puedes hacer:

```
try {
    $twig->parse($twig->tokenize($template));

    // $template es válida
} catch (Twig_Error_Syntax $e) {
    // $template contiene uno o más errores de sintaxis
}
```

7.9 Actualizando plantillas modificadas cuando APC está habilitado y apc.stat=0

Cuando utilizas APC con `apc.stat` establecido en `0` y está habilitada la memorización en caché de *Twig*, borra la caché de la plantilla que no va a actualizar la memoria caché *APC*. Para evitar esto, puedes extender `Twig_Environment` y forzar la actualización de la caché *APC* cuando *Twig* reescriba la memoria caché:

```
class Twig_Environment_APC extends Twig_Environment
{
    protected function writeCacheFile($file, $content)
    {
        parent::writeCacheFile($file, $content);

        // Archivo memorizado y compilado a bytecode
        apc_compile_file($file);
    }
}
```

Etiquetas

8.1 for

Recorre cada elemento de una secuencia. Por ejemplo, para mostrar una lista de usuarios provista en una variable llamada `usuarios`:

```
<h1>Members</h1>
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

Nota: Una secuencia puede ser una matriz o un objeto que implementa la interfaz `Traversable`.

Si necesitas iterar en una secuencia de números, puedes utilizar el operador `..`:

```
{% for i in 0..10 %}
    * {{ i }}
{% endfor %}
```

El fragmento de código anterior debería imprimir todos los números del 0 al 10.

También puede ser útil con letras:

```
{% for letter in 'a'..'z' %}
    * {{ letter }}
{% endfor %}
```

El operador `..` puede tomar cualquier expresión en ambos lados:

```
{% for letter in 'a'|upper..'z'|upper %}
    * {{ letter }}
{% endfor %}
```

Dentro de un bloque de bucle `for` puedes acceder a algunas variables especiales:

Variable	Descripción
`loop.index`	La iteración actual del bucle. (indexada en 1)
`loop.index0`	La iteración actual del bucle. (indexada en 0)
`loop.revindex`	El número de iteraciones a partir del final del bucle (indexadas en 1)
`loop.revindex0`	El número de iteraciones a partir del final del bucle (indexadas en 0)
`loop.first`	`True` si es la primera iteración
`loop.last`	`True` si es la última iteración
`loop.length`	El número de elementos en la secuencia
`loop.parent`	El contexto del padre

Nota: Las variables `loop.length`, `loop.revindex`, `loop.revindex0` y `loop.last` únicamente están disponibles para matrices *PHP*, u objetos que implementen la interfaz `Countable`.

Nuevo en la versión 1.2: La compatibilidad con el modificador `if` se añadió en *Twig* 1.2. A diferencia de *PHP*, en un bucle no es posible usar `break` ni `continue`. Sin embargo, puedes filtrar la secuencia durante la iteración, lo cual te permite omitir elementos. En el siguiente ejemplo se omiten todos los usuarios que no están activos:

```
<ul>
    {% for user in users if user.active %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

La ventaja es que la variable especial `loop` contará correctamente, es decir, sin contar a los usuarios inactivos en la iteración.

Si no se llevó a cabo iteración debido a que la secuencia está vacía, puedes reproducir un bloque sustituto utilizando `else`:

```
<ul>
    {% for user in users %}
        <li>{{ user.username|e }}</li>
    {% else %}
        <li><em>no user found</em></li>
    {% endfor %}
</ul>
```

De forma predeterminada, un bucle itera en los valores de la secuencia. Puedes iterar en las claves con el filtro `keys`:

```
<h1>Members</h1>
<ul>
    {% for key in users|keys %}
        <li>{{ key }}</li>
    {% endfor %}
</ul>
```

También puedes acceder tanto a las claves como a los valores:

```
<h1>Members</h1>
<ul>
    {% for key, user in users %}
        <li>{{ key }}: {{ user.username|e }}</li>
    {% endfor %}
</ul>
```

8.2 if

La declaración `if` en *Twig* es comparable con las declaraciones `if` de *PHP*. En la forma más simple la puedes usar para probar si una variable no está vacía:

```
{% if users %}
    <ul>
        {% for user in users %}
            <li>{{ user.username|e }}</li>
        {% endfor %}
    </ul>
{% endif %}
```

Nota: Si deseas probar si una variable está definida, usa `if usuarios is defined` en su lugar.

Para ramificación múltiple puedes utilizar `elseif` y `else` como en *PHP*. Allí también puedes utilizar `expresiones` más complejas:

```
{% if kenny.sick %}
    Kenny is sick.
{% elseif kenny.dead %}
    You killed Kenny!  You bastard!!!
{% else %}
    Kenny looks okay --- so far
{% endif %}
```

8.3 macro

Las macros son comparables con funciones en lenguajes de programación regulares. Son útiles para poner modismos *HTML* utilizados frecuentemente en elementos reutilizables para no repetirlos.

He aquí un pequeño ejemplo de una macro que reproduce un elemento de formulario:

```
{% macro input(name, value, type, size) %}
    <input type="{{ type|default('text') }}"
           name="{{ name }}"
           value="{{ value|e }}"
           size="{{ size|default(20) }}" />
{% endmacro %}
```

Las macros se diferencian de las funciones *PHP* nativas en varias formas:

- Los valores predeterminados de los argumentos se definen usando el filtro `default` en el cuerpo de la macro;
- Los argumentos de una macro siempre son opcionales.

Pero como las funciones de *PHP*, las macros no tienen acceso a las variables de la plantilla actual.

Truco: Puedes pasar todo el contexto como un argumento usando la variable especial `_context`.

Las macros se pueden definir en cualquier plantilla, y es necesario "importarlas", antes de utilizarlas (consulta la etiqueta *import* (Página 55) para más información):

```
{% import "formularios.html" as forms %}
```

La llamada a `import` anterior importa el archivo "formularios.html" (el cual puede contener macros solamente, o una plantilla y algunas macros), e importa las funciones como elementos de la variable `forms`.

Entonces puedes llamar a la macro a voluntad:

```
<p>{{ forms.input('username') }}</p>
<p>{{ forms.input('password', null, 'password') }}</p>
```

Si defines macros y las utilizadas en la misma plantilla, puedes utilizar la variable especial `_self`, sin necesidad de importarlas:

```
<p>{{ _self.input('nombreusuario') }}</p>
```

Cuando desees utilizar una macro en otra en el mismo archivo, utiliza la variable `_self`:

```
{% macro input(name, value, type, size) %}
  <input type="{{ type|default('text') }}"
             name="{{ name }}"
             value="{{ value|e }}"
             size="{{ size|default(20) }}" />
{% endmacro %}

{% macro wrapped_input(name, value, type, size) %}
    <div class="field">
        {{ _self.input(name, value, type, size) }}
    </div>
{% endmacro %}
```

Cuando la macro está definida en otro archivo, necesitas importarla:

```
{# formularios.html #}

{% macro input(name, value, type, size) %}
  <input type="{{ type|default('text') }}"
             name="{{ name }}"
             value="{{ value|e }}"
             size="{{ size|default(20) }}" />
{% endmacro %}

{# shortcuts.html #}

{% macro wrapped_input(name, value, type, size) %}
    {% import "formularios.html" as forms %}
    <div class="field">
        {{ forms.input(name, value, type, size) }}
    </div>
{% endmacro %}
```

Ver También:

from (Página 57), *import* (Página 55)

8.4 filter

Filtrar secciones te permite aplicar filtros *Twig* regulares en un bloque de datos de la plantilla. Simplemente envuelve el código en el bloque especial `filter`:

```
{% filter upper %}
    Este texto cambia a mayúsculas
{% endfilter %}
```

También puedes encadenar filtros:

```
{% filter lower|escape %}
    <strong>ALGÚN TEXTO</strong>
{% endfilter %}

{# produce "&lt;strong&gt;some text&lt;/strong&gt;" #}
```

8.5 set

Dentro del código de los bloques también puedes asignar valores a variables. Las asignaciones utilizan la etiqueta `set` y puedes tener múltiples destinos:

```
{% set foo = 'foo' %}

{% set foo = [1, 2] %}

{% set foo = {'foo': 'bar'} %}

{% set foo = 'foo' ~ 'bar' %}

{% set foo, bar = 'foo', 'bar' %}
```

La etiqueta `set` también se puede usar para "capturar" trozos de texto:

```
{% set foo %}
  <div id="pagination">
    ...
  </div>
{% endset %}
```

Prudencia: Si habilitas el escape automático, *Twig* sólo tendrá en cuenta el contenido seguro al capturar fragmentos de texto.

8.6 extends

Puedes utilizar la etiqueta `extends` para extender una plantilla a partir de otra.

Nota: Al igual que *PHP*, *Twig* no admite la herencia múltiple. Por lo tanto sólo puedes tener una etiqueta `extends` por reproducción. Sin embargo, *Twig* apoya el *reuso* (Página 57) horizontal.

Vamos a definir una plantilla base, `base.html`, la cual define el esqueleto de un documento *HTML* simple:

```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
```

```
            <title>{% block title %}{% endblock %} - My Webpage</title>
        {% endblock %}
    </head>
    <body>
        <div id="content">{% block content %}{% endblock %}</div>
        <div id="footer">
            {% block footer %}
                &copy; Copyright 2011 by <a href="http://domain.invalid/">you</a>.
            {% endblock %}
        </div>
    </body>
</html>
```

En este ejemplo, las etiquetas *{ % block %}* (Página 54) definen cuatro bloques que las plantillas descendientes pueden rellenar. Todas las etiquetas `bloque` le dicen al motor de plantillas que una plantilla heredera puede sustituir esas porciones de la plantilla.

8.6.1 Plantilla descendiente

Una plantilla hija podría tener este aspecto:

```
{% extends "base.html" %}

{% block title %}Index{% endblock %}
{% block head %}
    {{ parent() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
        Welcome on my awesome homepage.
    </p>
{% endblock %}
```

Aquí, la clave es la etiqueta `{ % extends %}`. Esta le dice al motor de plantillas que esta plantilla "extiende" otra plantilla. Cuando el sistema de plantillas evalúa esta plantilla, en primer lugar busca a la plantilla padre. La etiqueta `extends` debe ser la primera etiqueta de la plantilla.

Ten en cuenta que debido a que la plantilla heredera no define el bloque `footer`, en su lugar se utiliza el valor de la plantilla padre.

No puedes definir múltiples etiquetas `{ % block %}` con el mismo nombre en la misma plantilla. Esta limitación existe porque una etiqueta de bloque trabaja en "ambas" direcciones. Es decir, una etiqueta de bloque no sólo proporciona un hueco para rellenar - sino que también define en el *padre* el contenido que rellena el hueco. Si en una plantilla hubiera dos etiquetas `{ % block %}` con nombres similares, el padre de esa plantilla, no sabría cual contenido de entre los bloques usar.

No obstante, si deseas imprimir un bloque varias veces, puedes utilizar la función `block`:

```
<title>{% block title %}{% endblock %}</title>
<h1>{{ block('title') }}</h1>
{% block body %}{% endblock %}
```

Bloques padre

Es posible reproducir el contenido del bloque padre usando la función *parent* (Página 68). Esta devuelve el resultado del bloque padre:

```
{% block sidebar %}
    <h3>Table Of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

Etiquetas de cierre de bloque nombradas

Twig te permite poner el nombre del bloque después de la etiqueta para facilitar su lectura:

```
{% block sidebar %}
    {% block inner_sidebar %}
        ...
    {% endblock inner_sidebar %}
{% endblock sidebar %}
```

Por supuesto, el nombre después de la palabra `endblock` debe coincidir con el nombre del bloque.

Bloques anidados y ámbito

Los bloques se pueden anidar para diseños más complejos. Por omisión, los bloques tienen acceso a las variables del ámbito externo:

```
{% for item in seq %}
    <li>{% block loop_item %}{{ item }}{% endblock %}</li>
{% endfor %}
```

Atajos de bloque

Para bloques con poco contenido, es posible utilizar una sintaxis abreviada. Las siguientes construcciones hacen exactamente lo mismo:

```
{% block title %}
    {{ page_title|title }}
{% endblock %}

{% block title page_title|title %}
```

Herencia dinámica

Twig es compatible con la herencia dinámica usando una variable como la plantilla base:

```
{% extends alguna_var %}
```

Si la variable se evalúa como un objeto `Twig_Template`, *Twig* la utilizará como la plantilla padre:

```
// {% extends base %}

$base = $twig->loadTemplate('some_layout_template.twig');

$twig->display('template.twig', array('base' => $base));
```

Nuevo en la versión 1.2: La posibilidad de pasar un arreglo de plantillas se añadió en *Twig* 1.2. También puedes proporcionar una lista de plantillas que comprueben su existencia. La primer plantilla existente se utilizará como el padre:

```
{% extends ['base.html', 'base_layout.html'] %}
```

Herencia condicional

Gracias a que el nombre para la plantilla padre puede ser cualquier expresión *Twig*, es posible el mecanismo de herencia condicional:

```
{% extends standalone ? "minimum.html" : "base.html" %}
```

En este ejemplo, la plantilla debe extender a la plantilla base "minimum.html" si la variable `standalone` evalúa a `true`, o de otra manera extiende a "base.html".

Ver También:

block (Página 68), *block* (Página 54), *parent* (Página 68), *use* (Página 57)

8.7 block

Los bloques se utilizan para la herencia y actúan como marcadores de posición y reemplazo al mismo tiempo. Estos están documentados en detalle en la documentación de la etiqueta *extends* (Página 51).

Los nombres de bloque deben consistir de caracteres alfanuméricos y guiones bajos. Los guiones no están permitidos.

Ver También:

block (Página 68), *parent* (Página 68), *use* (Página 57), *extends* (Página 51)

8.8 include

La declaración `include` inserta una plantilla y devuelve el contenido presentado por ese archivo en el espacio de nombres actual:

```
{% include 'header.html' %}
    Body
{% include 'footer.html' %}
```

Las plantillas incluidas tienen acceso a las variables del contexto activo.

Puedes añadir variables adicionales pasándolas después de la palabra clave `with`:

```
{# la plantilla foo tendrá acceso a las variables del contexto actual y al de foo #}
{% include 'foo' with {'foo': 'bar'} %}

{% set vars = {'foo': 'bar'} %}
{% include 'foo' with vars %}
```

Puedes desactivar el acceso al contexto añadiendo la palabra clave `only`:

```
{# únicamente la variable foo será accesible #}
{% include 'foo' with {'foo': 'bar'} only %}

{# ninguna variable será accesible #}
{% include 'foo' only %}
```

Truco: Cuando incluyes una plantilla creada por un usuario final, debes considerar supervisarla. Más información en el capítulo *Twig para Desarrolladores* (Página 13).

El nombre de la plantilla puede ser cualquier expresión *Twig* válida:

```
{% include some_var %}
{% include ajax ? 'ajax.html' : 'not_ajax.html' %}
```

Y si la expresión evalúa como un objeto `Twig_Template`, *Twig* la usará directamente:

```
// {% include template %}

$template = $twig->loadTemplate('some_template.twig');

$twig->loadTemplate('template.twig')->display(array('template' => $template));
```

Nuevo en la versión 1.2: La característica `ignore missing` se añadió en *Twig* 1.2. Puedes marcar un `include` con `ignore missing` en cuyo caso *Twig* omitirá la declaración si la plantilla a ignorar no existe. Se tiene que colocar justo después del nombre de la plantilla. He aquí algunos ejemplos válidos:

```
{% include "sidebar.html" ignore missing %}
{% include "sidebar.html" ignore missing with {'foo': 'bar} %}
{% include "sidebar.html" ignore missing only %}
```

Nuevo en la versión 1.2: La posibilidad de pasar un arreglo de plantillas se añadió en *Twig* 1.2. También puedes proporcionar una lista de plantillas para comprobar su existencia antes de la inclusión. La primer plantilla existente será incluida:

```
{% include ['page_detailed.html', 'page.html'] %}
```

Si se le da `ignore missing`, caerá de nuevo en reproducir nada si ninguna de las plantillas existe, de lo contrario se producirá una excepción.

8.9 import

Twig apoya poner en macros el código usado frecuentemente *macros* (Página 49). Estas macros pueden estar en diferentes plantillas y se importan desde allí.

Hay dos formas de importar plantillas. Puedes importar la plantilla completa en una variable o solicitar macros específicas de ella.

Imaginemos que tienes un módulo auxiliar que reproduce formularios (llamado `formularios.html`):

```
{% macro input(name, value, type, size) %}
    <input type="{{ type|default('text') }}"
           name="{{ name }}"
           value="{{ value|e }}"
           size="{{ size|default(20) }}" />
{% endmacro %}
```

```
{% macro textarea(name, value, rows) %}
    <textarea name="{{ name }}"
          rows="{{ rows|default(10) }}"
          cols="{{ cols|default(40) }}">
          {{ value|e }}
</textarea>
{% endmacro %}
```

La forma más fácil y flexible es importar todo el módulo en una variable. De esa manera puedes acceder a los atributos:

```
{% import 'formularios.html' as forms %}

<dl>
    <dt>Username</dt>
    <dd>{{ forms.input('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ forms.input('password', null, 'password') }}</dd>
</dl>
<p>{{ forms.textarea('comentario') }}</p>
```

Alternativamente, puedes importar nombres desde la plantilla al espacio de nombres actual:

```
{% from 'formularios.html' import input as campo_input, textarea %}

<dl>
    <dt>Username</dt>
    <dd>{{ input_field('username') }}</dd>
    <dt>Password</dt>
    <dd>{{ input_field('password', '', 'password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

La importación no es necesaria si las macros y la plantilla están definidas en el mismo archivo; en su lugar usa la variable especial `_self`:

```
{# plantilla index.html #}

{% macro textarea(name, value, rows) %}
    <textarea name="{{ name }}"
          rows="{{ rows|default(10) }}"
          cols="{{ cols|default(40) }}">
          {{ value|e }}
</textarea>
{% endmacro %}

<p>{{ _self.textarea('comentario') }}</p>
```

Pero sí puedes crear un alias importando la variable `_self`:

```
{# plantilla index.html #}

{% macro textarea(name, value, rows) %}
    <textarea name="{{ name }}"
          rows="{{ rows|default(10) }}"
          cols="{{ cols|default(40) }}">
          {{ value|e }}
</textarea>
{% endmacro %}
```

```
{% import _self as forms %}

<p>{{ forms.textarea('comentario') }}</p>
```

Ver También:

macro (Página 49), *from* (Página 57)

8.10 from

Las etiquetas `from` importan nombres de *macro* (Página 49) al espacio de nombres actual. La etiqueta está documentada en detalle en la documentación de la etiqueta *import* (Página 55).

Ver También:

macro (Página 49), *import* (Página 55)

8.11 use

Nuevo en la versión 1.1: La reutilización horizontal se añadió en *Twig* 1.1.

Nota: La reutilización horizontal es una característica avanzada de *Twig* que casi nunca es necesaria en plantillas regulares. La utilizan principalmente proyectos que tienen que reutilizar bloques de plantilla sin utilizar herencia.

La herencia de plantillas es una de las más poderosas características de *Twig*, pero está limitada a herencia simple; una plantilla sólo puede extender a una plantilla más. Esta limitación facilita el entendimiento y depuración de la herencia de plantillas:

```
{% extends "base.html" %}

{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

La reutilización horizontal es una forma de conseguir el mismo objetivo que la herencia múltiple, pero sin la complejidad asociada:

```
{% extends "base.html" %}

{% use "bloques.html" %}

{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

La declaración `use` dice a *Twig* que importe los bloques definidos en `bloques.html` a la plantilla actual (es como las macros, pero para bloques):

```
{# bloques.html #}
{% block sidebar %}{% endblock %}
```

En este ejemplo, la declaración `use` importa la declaración del bloque `sidebar` en la plantilla principal. El código —en su mayoría— es equivalente a lo siguiente (los bloques importados no se generan automáticamente):

```
{% extends "base.html" %}

{% block sidebar %}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

Nota:

La etiqueta `use` sólo importa una plantilla si esta:

- no extiende a otra plantilla
- no define macros, y
- si el cuerpo está vacío. Pero puedes *usar* otras plantillas.

Nota: Debido a que las declaraciones `use` se resuelven independientemente del contexto pasado a la plantilla, la referencia de la plantilla no puede ser una expresión.

La plantilla principal también puede sustituir cualquier bloque importado. Si la plantilla ya define el bloque `sidebar`, entonces, se ignora el definido en `bloques.html`. Para evitar conflictos de nombre, puedes cambiar el nombre de los bloques importados:

```
{% extends "base.html" %}

{% use "bloques.html" with sidebar as base_sidebar %}

{% block sidebar %}{% endblock %}
{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

Nuevo en la versión 1.3: El apoyo a `parent()` se añadió en *Twig 1.3*. La función `parent()` determina automáticamente el árbol de herencia correcto, por lo tanto lo puedes utilizar cuando reemplaces un bloque definido en una plantilla importada:

```
{% extends "base.html" %}

{% use "bloques.html" %}

{% block sidebar %}
    {{ parent() }}
{% endblock %}

{% block title %}{% endblock %}
{% block content %}{% endblock %}
```

En este ejemplo, el `parent()` correctamente llama al bloque `sidebar` de la plantilla `blocks.html`.

Truco: En *Twig 1.2*, el cambio de nombre te permite simular la herencia llamando al bloque "padre":

```
{% extends "base.html" %}

{% use "bloques.html" with sidebar as parent_sidebar %}

{% block sidebar %}
    {{ block('parent_sidebar') }}
{% endblock %}
```

Nota: Puedes utilizar tantas instrucciones `use` como quieras en cualquier plantilla determinada. Si dos plantillas importadas definen el mismo bloque, la última gana.

8.12 spaceless

Utiliza la etiqueta `spaceless` para quitar los espacios en blanco entre las etiquetas *HTML*:

```
{% spaceless %}
    <div>
        <strong>foo</strong>
    </div>
{% endspaceless %}

{# Producirá <div><strong>foo</strong></div> #}
```

8.13 autoescape

Ya sea que el escape automático esté habilitado o no, puedes marcar una sección de una plantilla para que sea escapada o no utilizando la etiqueta `autoescape`:

```
{% autoescape true %}
    Todo en este bloque se va a escapar automáticamente
{% endautoescape %}

{% autoescape false %}
    Todo en este bloque se reproducirá tal cual
{% endautoescape %}

{% autoescape true js %}
    Todo en este bloque se escapará automáticamente con la estrategia de escape js
{% endautoescape %}
```

Cuando se activa el escape automático, de manera predeterminada todo será escapado, salvo los valores marcados explícitamente como seguros. Estos se pueden marcar en la plantilla usando el filtro *raw* (Página 65):

```
{% autoescape true %}
    {{ safe_value|raw }}
{% endautoescape %}
```

Las funciones que devuelven datos de la plantilla (como *macros* (Página 49) y *parent* (Página 68)) siempre devuelven marcado seguro.

Nota: *Twig* es lo suficientemente inteligente como para no escapar un valor que ya fue escapado por el filtro *escape* (Página 65).

Nota: El capítulo *Twig para desarrolladores* (Página 13) proporciona más información acerca de cuándo y cómo se aplica el escape automático.

8.14 raw

La etiqueta `raw` marca secciones como texto seguro que no se deben analizar. Por ejemplo, para reproducir un segmento de la sintaxis de *Twig* en una plantilla, puedes utilizar este fragmento:

```
{% raw %}
    <ul>
    {% for item in seq %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endraw %}
```

Filtros

9.1 date

Nuevo en la versión 1.1: La compatibilidad con la zona horaria se añadió en *Twig* 1.1. El filtro `date` es capaz de formatear una fecha en un determinado formato:

```
{{ post.published_at|date("m/d/Y") }}
```

El filtro `date` acepta cualquier formato de fecha compatible con DateTime e instancias de `DateTime`. Por ejemplo, para mostrar la fecha actual, filtra la palabra `"now"`:

```
{{ "now"|date("m/d/Y") }}
```

Para escapar palabras y caracteres en el formato de fecha usa `\\` al frente de cada carácter:

```
{{ post.published_at|date("F jS \\a\\t g:ia") }}
```

También puedes especificar una zona horaria:

```
{{ post.published_at|date("m/d/Y", "Europe/Paris") }}
```

9.2 format

El filtro `format` filtra formatos de una cadena dada sustituyendo los marcadores de posición (los marcadores de posición siguen la notación de `printf`):

```
{{ "Me gustan %s y %s."|format(foo, "bar") }}

{# devuelve Me gustan foo y bar
   si el parámetro foo es igual a la cadena foo. #}
```

Ver También:

replace (Página 62)

9.3 replace

El filtro `reemplaza` formatos de una cadena dada sustituyendo los marcadores de posición (los marcadores de posición son libres):

```
{{ "Me gustan %this% y %that%."|replace({'%this%': foo, '%that%': "bar"}) }}

{# devuelve Me gustan foo y bar
   si el parámetro foo es igual a la cadena foo. #}
```

Ver También:

format (Página 61)

9.4 url_encode

El filtro `url_encode` produce una cadena *URL* codificada.

```
{{ data|url_encode() }}
```

Nota: Internamente, *Twig* utiliza la función urlencode de *PHP*.

9.5 json_encode

El filtro `json_encode` devuelve la representación *JSON* de una cadena:

```
{{ data|json_encode() }}
```

Nota: Internamente, *Twig* utiliza la función json_encode de *PHP*.

9.6 convert_encoding

Nuevo en la versión 1.4: El filtro `convert_encoding` se añadió en *Twig* 1.4. El filtro `convert_encoding` convierte una cadena de una codificación a otra. El primer argumento es el juego de caracteres esperado y el segundo es el juego de caracteres de entrada:

```
{{ data|convert_encoding('UTF-8', 'iso-2022-jp') }}
```

Nota: Este filtro está basado en la extensión iconv o mbstring. Por lo tanto una de ellas debe estar instalada.

9.7 title

El filtro `title` devuelve una versión con mayúsculas iniciales del valor. Es decir, las palabras deben empezar con letras mayúsculas, todos los caracteres restantes son minúsculas:

```
{{ 'mi primer automóvil'|title }}

{# produce 'Mi Primer Automóvil' #}
```

9.8 capitalize

El filtro `capitalize` capitaliza un valor. El primer carácter será en mayúscula, todos los demás en minúsculas:

```
{{ 'my first car'|capitalize }}

{# produce 'My first car' #}
```

9.9 upper

El filtro `upper` convierte un valor a mayúsculas:

```
{{ 'bienvenido'|upper }}

{# produce 'BIENVENIDO' #}
```

9.10 lower

El filtro `lower` convierte un valor a minúsculas:

```
{{ 'WELCOME'|lower }}

{# produce 'welcome' #}
```

9.11 striptags

El filtro `striptags` quita etiquetas *SGML/XML* y sustituye los espacios en blanco adyacentes por un espacio:

```
{% some_html|striptags %}
```

Nota: Internamente, *Twig* utiliza la función strip_tags de *PHP*.

9.12 join

El filtro `join` devuelve una cadena que es la concatenación de las cadenas de una secuencia:

```
{{ [1, 2, 3]|join }}
{# devuelve 123 #}
```

El separador predeterminado entre los elementos es una cadena vacía, lo puedes definir con el primer parámetro opcional:

```
{{ [1, 2, 3]|join('|') }}
{# devuelve 1|2|3 #}
```

9.13 reverse

El filtro `reverse` invierte una matriz (o un objeto si este implementa la interfaz `Iterator`):

```
{% for use in users|reverse %}
    ...
{% endfor %}
```

9.14 length

El filtro `length` devuelve el número de elementos de una secuencia o asignación, o la longitud de una cadena:

```
{% if users|length > 10 %}
    ...
{% endif %}
```

9.15 sort

El filtro `sort` ordena una matriz:

```
{% for use in users|sort %}
    ...
{% endfor %}
```

Nota: Internamente, *Twig* utiliza la función asort de *PHP* para mantener asociado el índice.

9.16 default

El filtro `default` devuelve el valor pasado como predeterminado si el valor no está definido o está vacío, de lo contrario devuelve el valor de la variable:

```
{{ var|default('var no está definido') }}

{{ var.foo|default('el elemento foo en var no está definido') }}

{{ var['foo']|default('el elemento foo en var no está definido') }}

{{ ''|default('la variable pasada está vacía')  }}
```

Cuando usas el filtro `default` en una expresión que usa variables en alguna llamada a método, asegúrate de usar el filtro `default` cuando no se haya definido una variable:

```
{{ var.method(foo|default('foo'))|default('foo') }}
```

Nota: Lee más adelante la documentación de las pruebas *defined* (Página 72) y *empty* (Página 73) para aprender más acerca de su semántica.

9.17 keys

El filtro `keys` devuelve las claves de una matriz. Es útil cuando deseas iterar sobre las claves de una matriz:

```
{% for key in array|keys %}
    ...
{% endfor %}
```

9.18 escape

El filtro `escape` convierte los caracteres `&`, <, >, `'` y `"` de cadenas a secuencias *HTML* seguras. Utiliza esta opción si necesitas mostrar texto que puede contener tales caracteres *HTML*:

```
{{ user.username|escape }}
```

Por conveniencia, el filtro `e` está definido como un alias:

```
{{ user.username|e }}
```

El filtro `escape` también se puede utilizar fuera del contexto *HTML*; Por ejemplo, para mostrar algo en un archivo *JavaScript*, utiliza el contexto `js`:

```
{{ user.username|escape('js') }}
{{ user.username|e('js') }}
```

Nota: Internamente, `escape` utiliza la función htmlspecialchars nativa de *PHP*.

9.19 raw

El filtro `raw` marca el valor como "seguro", lo cual significa que en un entorno con escape automático activado esta variable no será escapada siempre y cuando `raw` sea el último filtro que se le aplica:

```
{% autoescape true %}
    {{ var|raw }} {# var no es escapada #}
{% endautoescape %}
```

9.20 merge

El filtro `merge` combina una matriz o un `hash` con el valor:

```
{% set items = { 'apple': 'fruit', 'orange': 'fruit' } %}

{% set items = items|merge({ 'peugeot': 'car' }) %}

{# items ahora contiene { 'apple': 'fruit', 'orange': 'fruit', 'peugeot': 'car' } #}
```

Funciones

10.1 range

Devuelve una lista conteniendo una progresión aritmética de enteros:

```
{% for i in range(0, 3) %}
    {{ i }},
{% endfor %}

{# devuelve 0, 1, 2, 3 #}
```

Cuando se da el paso (como tercer parámetro), este especifica el incremento (o decremento):

```
{% for i in range(0, 6, 2) %}
    {{ i }},
{% endfor %}

{# devuelve 0, 2, 4, 6 #}
```

El operador integrado `..` es azúcar sintáctica para la función `range` (con un paso de 1):

```
{% for i in 0..3 %}
    {{ i }},
{% endfor %}
```

Truco: La función `range` trabaja como la función range nativa de *PHP*.

10.2 cycle

Puedes utilizar la función `cycle` para recorrer un arreglo de valores:

```
{% for i in 0..10 %}
    {{ cycle(['odd', 'even'], i) }}
{% endfor %}
```

La matriz puede contener cualquier cantidad de valores:

```
{% set frutas = ['manzana', 'naranja', 'cítricos'] %}

{% for i in 0..10 %}
    {{ cycle(frutas, i) }}
{% endfor %}
```

10.3 constant

`constant` devuelve el valor constante de una determinada cadena:

```
{{ some_date|date(constant('DATE_W3C')) }}
{{ constant('Namespace\\Classname::CONSTANT_NAME') }}
```

10.4 attribute

Nuevo en la versión 1.2: La función `attribute` se añadió en *Twig* 1.2. Puedes usar `attribute` para acceder a los atributos "dinámicos" de una variable:

```
{{ attribute(object, method) }}
{{ attribute(object, method, arguments) }}
{{ attribute(array, item) }}
```

Nota: El algoritmo de resolución es el mismo que el utilizado para la notación de punto (".."), salvo que el elemento puede ser cualquier expresión válida.

10.5 block

Cuando una plantilla utiliza herencia y si deseas imprimir un bloque varias veces, usa la función `block`:

```
<title>{% block title %}{% endblock %}</title>

<h1>{{ block('title') }}</h1>

{% block body %}{% endblock %}
```

Ver También:

extends (Página 51), *parent* (Página 68)

10.6 parent

Cuando una plantilla utiliza herencia, es posible reproducir el contenido del bloque padre cuando reemplaces un bloque usando la función `parent`:

```
{% extends "base.html" %}

{% block sidebar %}
    <h3>Table Of Contents</h3>
```

```
    ...
    {{ parent() }}
{% endblock %}
```

La llamada a `parent()` devolverá el contenido del bloque `sidebar` cómo lo definimos en la plantilla `base.html`.

Ver También:

extends (Página 51), *block* (Página 68), *block* (Página 54)

Probando

11.1 divisibleby

`divisibleby` comprueba si una variable es divisible por un número:

```
{% if loop.index is divisibleby(3) %}
    ...
{% endif %}
```

11.2 null

`null` devuelve `true` si la variable es `null`:

```
{{ var is null }}
```

Nota: `none` es un alias para `null`.

11.3 even

`even` devuelve `true` si el número dado es par:

```
{{ var is even }}
```

Ver También:

odd (Página 71)

11.4 odd

`odd` devuelve `true` si el número dado es impar:

```
{{ var is odd }}
```

Ver También:

even (Página 71)

11.5 sameas

`sameas` comprueba si una variable apunta a la misma dirección de memoria que otra variable:

```
{% if foo.attribute is sameas(false) %}
    el atributo foo en realidad es el valor ``false`` de PHP
{% endif %}
```

11.6 constant

`constant` comprueba si una variable tiene el mismo valor exacto que una constante. Puedes utilizar cualquiera de las constantes globales o constantes de clase:

```
{% if post.status is constant('Post::PUBLISHED') %}
    the status attribute is exactly the same as Post::PUBLISHED
{% endif %}
```

11.7 defined

`defined` comprueba si una variable está definida en el contexto actual. Esto es muy útil si utilizas la opción `strict_variables`:

```
{# defined trabaja con nombres de variable #}
{% if foo is defined %}
    ...
{% endif %}

{# y atributos en nombres de variables #}
{% if foo.bar is defined %}
    ...
{% endif %}

{% if foo['bar'] is defined %}
    ...
{% endif %}
```

Cuando uses la prueba `defined` en una expresión que usa variables en alguna llamada a método, primero asegúrate de haberlas definido:

```
{% if var is defined and foo.method(var) is defined %}
    ...
{% endif %}
```

11.8 empty

`empty` comprueba si una variable está vacía:

```
{# evalúa a true si la variable foo es null, false o la cadena vacía #}
{% if foo is empty %}
    ...
{% endif %}
```