Twig Documentation

Release latest

Sep 27, 2017

Contents

CHAPTER 1

Introduction

This is the documentation for Twig, the flexible, fast, and secure template engine for PHP.

If you have any exposure to other text-based template languages, such as Smarty, Django, or Jinja, you should feel right at home with Twig. It's both designer and developer friendly by sticking to PHP's principles and adding functionality useful for templating environments.

The key-features are...

- *Fast*: Twig compiles templates down to plain optimized PHP code. The overhead compared to regular PHP code was reduced to the very minimum.
- *Secure*: Twig has a sandbox mode to evaluate untrusted template code. This allows Twig to be used as a template language for applications where users may modify the template design.
- *Flexible*: Twig is powered by a flexible lexer and parser. This allows the developer to define their own custom tags and filters, and to create their own DSL.

Twig is used by many Open-Source projects like Symfony, Drupal8, eZPublish, phpBB, Piwik, OroCRM; and many frameworks have support for it as well like Slim, Yii, Laravel, Codeigniter and Kohana — just to name a few.

Prerequisites

Twig needs at least **PHP 7.0.0** to run.

Installation

The recommended way to install Twig is via Composer:

```
composer require "twig/twig:^2.0"
```

Note: To learn more about the other installation methods, read the *installation* chapter; it also explains how to install the Twig C extension.

Basic API Usage

This section gives you a brief introduction to the PHP API for Twig.

```
require_once '/path/to/vendor/autoload.php';

$loader = new Twig_Loader_Array(array(
    'index' => 'Hello {{ name }}!',
));
$twig = new Twig_Environment($loader);

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

Twig uses a loader (`Twig_Loader_Array`) to locate templates, and an environment (`Twig_Environment`) to store the configuration.

The `render()` method loads the template passed as a first argument and renders it with the variables passed as a second argument.

As templates are generally stored on the filesystem, Twig also comes with a filesystem loader:

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

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

CHAPTER 2

Installation

You have multiple ways to install Twig.

Installing the Twig PHP package

Installing via Composer (recommended)

Install Composer and run the following command to get the latest version:

```
composer require twig/twig:~3.0
```

Installing from the tarball release

1. Download the most recent tarball from the download page
2. Verify the integrity of the tarball http://fabien.potencier.org/article/73/signing-project-releases
3. Unpack the tarball
4. Move the files somewhere in your project

Installing the development version

```
git clone git://github.com/twigphp/Twig.git
```

CHAPTER 3

Twig for Template Designers

This document describes the syntax and semantics of the template engine and will be most useful as reference to those creating Twig templates.

Synopsis

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, LaTeX, etc.). It doesn't have a specific extension, `.html` or `.xml` are just fine.

A template contains **variables** or **expressions**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. We will cover further details later on:

```
<!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>
```

There are two kinds of delimiters: `{% ... %}` and `{{ ... }}`. The first one is used to execute statements such as for-loops, the latter prints the result of an expression to the template.

IDEs Integration

Many IDEs support syntax highlighting and auto-completion for Twig:

- *Textmate* via the Twig bundle
- *Vim* via the Jinja syntax plugin or the vim-twig plugin
- *Netbeans* via the Twig syntax plugin (until 7.1, native as of 7.2)
- *PhpStorm* (native as of 2.1)
- *Eclipse* via the Twig plugin
- *Sublime Text* via the Twig bundle
- *GtkSourceView* via the Twig language definition (used by gedit and other projects)
- *Coda* and *SubEthaEdit* via the Twig syntax mode
- *Coda 2* via the other Twig syntax mode
- *Komodo* and *Komodo Edit* via the Twig highlight/syntax check mode
- *Notepad++* via the Notepad++ Twig Highlighter
- *Emacs* via web-mode.el
- *Atom* via the PHP-twig for atom
- *Visual Studio Code* via the Twig pack

Also, TwigFiddle is an online service that allows you to execute Twig templates from a browser; it supports all versions of Twig.

Variables

The application passes variables to the templates for manipulation in the template. Variables may have attributes or elements you can access, too. The visual representation of a variable depends heavily on the application providing it.

You can use a dot (`.`) to access attributes of a variable (methods or properties of a PHP object, or items of a PHP array), or the so-called "subscript" syntax (`[]`):

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

When the attribute contains special characters (like `-` that would be interpreted as the minus operator), use the `attribute` function instead to access the variable attribute:

```
{# equivalent to the non-working foo.data-foo #}
{{ attribute(foo, 'data-foo') }}
```

Note: It's important to know that the curly braces are *not* part of the variable but the print statement. When accessing variables inside tags, don't put the braces around them.

If a variable or attribute does not exist, you will receive a `null` value when the `strict_variables` option is set to `false`; alternatively, if `strict_variables` is set, Twig will throw an error (see *environment options*).

Implementation

For convenience's sake `foo.bar` does the following things on the PHP layer:

- check if `foo` is an array and `bar` a valid element;
- if not, and if `foo` is an object, check that `bar` is a valid property;
- if not, and if `foo` is an object, check that `bar` is a valid method (even if `bar` is the constructor - use `__construct()` instead);
- if not, and if `foo` is an object, check that `getBar` is a valid method;
- if not, and if `foo` is an object, check that `isBar` is a valid method;
- if not, and if `foo` is an object, check that `hasBar` is a valid method;
- if not, return a `null` value.

`foo['bar']` on the other hand only works with PHP arrays:

- check if `foo` is an array and `bar` a valid element;
- if not, return a `null` value.

Note: If you want to access a dynamic attribute of a variable, use the *attribute* function instead.

Global Variables

The following variables are always available in templates:

- `_self`: references the current template name;
- `_context`: references the current context;
- `_charset`: references the current charset.

Setting Variables

You can assign values to variables inside code blocks. Assignments use the *set* tag:

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

Filters

Variables can be modified by **filters**. Filters are separated from the variable by a pipe symbol (`|`) and may have optional arguments in parentheses. Multiple filters can be chained. The output of one filter is applied to the next.

The following example removes all HTML tags from the `name` and title-cases it:

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

Filters that accept arguments have parentheses around the arguments. This example will join a list by commas:

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

To apply a filter on a section of code, wrap it in the *filter* tag:

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

Go to the *filters* page to learn more about built-in filters.

Functions

Functions can be called to generate content. Functions are called by their name followed by parentheses (`()`) and may have arguments.

For instance, the `range` function returns a list containing an arithmetic progression of integers:

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

Go to the *functions* page to learn more about the built-in functions.

Named Arguments

```
{% for i in range(low=1, high=10, step=2) %}
    {{ i }},
{% endfor %}
```

Using named arguments makes your templates more explicit about the meaning of the values you pass as arguments:

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

{# versus #}

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

Named arguments also allow you to skip some arguments for which you don't want to change the default value:

```
{# the first argument is the date format, which defaults to the global date format if␣
↪null is passed #}
{{ "now"|date(null, "Europe/Paris") }}

{# or skip the format value by using a named argument for the time zone #}
{{ "now"|date(timezone="Europe/Paris") }}
```

You can also use both positional and named arguments in one call, in which case positional arguments must always come before named arguments:

```
{{ "now"|date('d/m/Y H:i', timezone="Europe/Paris") }}
```

Tip: Each function and filter documentation page has a section where the names of all arguments are listed when supported.

Control Structure

A control structure refers to all those things that control the flow of a program - conditionals (i.e. `if`/`elseif`/`else`), `for`-loops, as well as things like blocks. Control structures appear inside `{% ... %}` blocks.

For example, to display a list of users provided in a variable called `users`, use the *for* tag:

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

The *if* tag can be used to test an expression:

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

Go to the *tags* page to learn more about the built-in tags.

Comments

To comment-out part of a line in a template, use the comment syntax `{# ... #}`. This is useful for debugging or to add information for other template designers or yourself:

```
{# note: disabled template because we no longer use this
    {% for user in users %}
        ...
    {% endfor %}
#}
```

Including other Templates

The *include* function is useful to include a template and return the rendered content of that template into the current one:

```
{{ include('sidebar.html') }}
```

By default, included templates have access to the same context as the template which includes them. This means that any variable defined in the main template will be available in the included template too:

```
{% for box in boxes %}
    {{ include('render_box.html') }}
{% endfor %}
```

The included template `render_box.html` is able to access the `box` variable.

The filename of the template depends on the template loader. For instance, the `Twig_Loader_Filesystem` allows you to access other templates by giving the filename. You can access templates in subdirectories with a slash:

```
{{ include('sections/articles/sidebar.html') }}
```

This behavior depends on the application embedding Twig.

Template Inheritance

The most powerful part of Twig is template inheritance. Template inheritance allows you to build a base "skeleton" template that contains all the common elements of your site and defines **blocks** that child templates can override.

Sounds complicated but it is very basic. It's easier to understand it by starting with an example.

Let's define a base template, `base.html`, which defines a simple HTML skeleton document that you might use for a simple two-column page:

```
<!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>
```

In this example, the *block* tags define four blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

```
{% 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 to my awesome homepage.
    </p>
{% endblock %}
```

The *extends* tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent. The extends tag should be the first tag in the template.

Note that since the child template doesn't define the `footer` block, the value from the parent template is used instead.

It's possible to render the contents of the parent block by using the *parent* function. This gives back the results of the parent block:

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

Tip: The documentation page for the *extends* tag describes more advanced features like block nesting, scope, dynamic inheritance, and conditional inheritance.

Note: Twig also supports multiple inheritance with the so called horizontal reuse with the help of the *use* tag. This is an advanced feature hardly ever needed in regular templates.

HTML Escaping

When generating HTML from templates, there's always a risk that a variable will include characters that affect the resulting HTML. There are two approaches: manually escaping each variable or automatically escaping everything by default.

Twig supports both, automatic escaping is enabled by default.

The automatic escaping strategy can be configured via the *autoescape* option and defaults to `html`.

Working with Manual Escaping

If manual escaping is enabled, it is **your** responsibility to escape variables if needed. What to escape? Any variable you don't trust.

Escaping works by piping the variable through the *escape* or `e` filter:

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

By default, the `escape` filter uses the `html` strategy, but depending on the escaping context, you might want to explicitly use any other available strategies:

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

Working with Automatic Escaping

Whether automatic escaping is enabled or not, you can mark a section of a template to be escaped or not by using the *autoescape* tag:

```
{% autoescape %}
    Everything will be automatically escaped in this block (using the HTML strategy)
{% endautoescape %}
```

By default, auto-escaping uses the `html` escaping strategy. If you output variables in other contexts, you need to explicitly escape them with the appropriate escaping strategy:

```
{% autoescape 'js' %}
    Everything will be automatically escaped in this block (using the JS strategy)
{% endautoescape %}
```

Escaping

It is sometimes desirable or even necessary to have Twig ignore parts it would otherwise handle as variables or blocks. For example if the default syntax is used and you want to use `{{` as raw string in the template and not start a variable you have to use a trick.

The easiest way is to output the variable delimiter (`{{`) by using a variable expression:

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

For bigger sections it makes sense to mark a block *verbatim*.

Macros

Macros are comparable with functions in regular programming languages. They are useful to reuse often used HTML fragments to not repeat yourself.

A macro is defined via the *macro* tag. Here is a small example (subsequently called `forms.html`) of a macro that renders a form element:

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

Macros can be defined in any template, and need to be "imported" via the *import* tag before being used:

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

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

Alternatively, you can import individual macro names from a template into the current namespace via the *from* tag and optionally alias them:

```
{% from 'forms.html' import input as input_field %}

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

A default value can also be defined for macro arguments when not provided in a macro call:

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

If extra positional arguments are passed to a macro call, they end up in the special `varargs` variable as a list of values.

Expressions

Twig allows expressions everywhere. These work very similar to regular PHP and even if you're not working with PHP you should feel comfortable with it.

Note: The operator precedence is as follows, with the lowest-precedence operators listed first: `b-and`, `b-xor`, `b-or`, `or`, `and`, `==`, `!=`, `<`, `>`, `>=`, `<=`, `in`, `matches`, `starts with`, `ends with`, `..`, `+`, `-`, `~`, `*`, `/`, `//`, `%`, `is`, `**`, `|`, `[]`, and `.`:

```
{% set greeting = 'Hello ' %}
{% set name = 'Fabien' %}

{{ greeting ~ name|lower }}   {# Hello fabien #}

{# use parenthesis to change precedence #}
{{ (greeting ~ name)|lower }} {# hello fabien #}
```

Literals

The simplest form of expressions are literals. Literals are representations for PHP types such as strings, numbers, and arrays. The following literals exist:

- `"Hello World"`: Everything between two double or single quotes is a string. They are useful whenever you need a string in the template (for example as arguments to function calls, filters or just to extend or include a template). A string can contain a delimiter if it is preceded by a backslash (`\`) – like in `'It\'s good'`. If the string contains a backslash (e.g. `'c:\Program Files'`) escape it by doubling it (e.g. `'c:\\Program Files'`).
- `42` / `42.23`: Integers and floating point numbers are created by just writing the number down. If a dot is present the number is a float, otherwise an integer.
- `["foo", "bar"]`: Arrays are defined by a sequence of expressions separated by a comma (`,`) and wrapped with squared brackets (`[]`).

- `{"foo": "bar"}`: Hashes are defined by a list of keys and values separated by a comma (`,`) and wrapped with curly braces (`{}`):

```
{# keys as string #}
{ 'foo': 'foo', 'bar': 'bar' }

{# keys as names (equivalent to the previous hash) #}
{ foo: 'foo', bar: 'bar' }

{# keys as integer #}
{ 2: 'foo', 4: 'bar' }

{# keys as expressions (the expression must be enclosed into parentheses) #}
{% set foo = 'foo' %}
{ (foo): 'foo', (1 + 1): 'bar', (foo ~ 'b'): 'baz' }
```

- `true` / `false`: `true` represents the true value, `false` represents the false value.
- `null`: `null` represents no specific value. This is the value returned when a variable does not exist. `none` is an alias for `null`.

Arrays and hashes can be nested:

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

Tip: Using double-quoted or single-quoted strings has no impact on performance but string interpolation is only supported in double-quoted strings.

Math

Twig allows you to calculate with values. This is rarely useful in templates but exists for completeness' sake. The following operators are supported:

- `+`: Adds two objects together (the operands are casted to numbers). `{{ 1 + 1 }}` is `2`.
- `-`: Subtracts the second number from the first one. `{{ 3 - 2 }}` is `1`.
- `/`: Divides two numbers. The returned value will be a floating point number. `{{ 1 / 2 }}` is `{{ 0.5 }}`.
- `%`: Calculates the remainder of an integer division. `{{ 11 % 7 }}` is `4`.
- `//`: Divides two numbers and returns the floored integer result. `{{ 20 // 7 }}` is `2`, `{{ -20 // 7 }}` is `-3` (this is just syntactic sugar for the *round* filter).
- `*`: Multiplies the left operand with the right one. `{{ 2 * 2 }}` would return `4`.
- `**`: Raises the left operand to the power of the right operand. `{{ 2 ** 3 }}` would return `8`.

Logic

You can combine multiple expressions with the following operators:

- `and`: Returns true if the left and the right operands are both true.
- `or`: Returns true if the left or the right operand is true.
- `not`: Negates a statement.

- `(expr)`: Groups an expression.

Note: Twig also support bitwise operators (`b-and`, `b-xor`, and `b-or`).

Note: Operators are case sensitive.

Comparisons

The following comparison operators are supported in any expression: ==, !=, <, >, >=, and <=.

You can also check if a string `starts with` or `ends with` another string:

```
{% if 'Fabien' starts with 'F' %}
{% endif %}

{% if 'Fabien' ends with 'n' %}
{% endif %}
```

Note: For complex string comparisons, the `matches` operator allows you to use regular expressions:

```
{% if phone matches '/^[\\d\\.]+$/' %}
{% endif %}
```

Containment Operator

The `in` operator performs containment test.

It returns `true` if the left operand is contained in the right:

```
{# returns true #}

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

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

Tip: You can use this filter to perform a containment test on strings, arrays, or objects implementing the `Traversable` interface.

To perform a negative test, use the `not in` operator:

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

{# is equivalent to #}
{% if not (1 in [1, 2, 3]) %}
```

Test Operator

The `is` operator performs tests. Tests can be used to test a variable against a common expression. The right operand is name of the test:

```
{# find out if a variable is odd #}

{{ name is odd }}
```

Tests can accept arguments too:

```
{% if post.status is constant('Post::PUBLISHED') %}
```

Tests can be negated by using the `is not` operator:

```
{% if post.status is not constant('Post::PUBLISHED') %}

{# is equivalent to #}
{% if not (post.status is constant('Post::PUBLISHED')) %}
```

Go to the *tests* page to learn more about the built-in tests.

Other Operators

The following operators don't fit into any of the other categories:

- `|`: Applies a filter.
- `..`: Creates a sequence based on the operand before and after the operator (this is just syntactic sugar for the *range* function):

  ```
  {{ 1..5 }}

  {# equivalent to #}
  {{ range(1, 5) }}
  ```

  Note that you must use parentheses when combining it with the filter operator due to the *operator precedence rules*:

  ```
  (1..5)|join(', ')
  ```

- `~`: Converts all operands into strings and concatenates them. `{{ "Hello " ~ name ~ "!" }}` would return (assuming `name` is `'John'`) `Hello John!`.
- `.`, `[]`: Gets an attribute of an object.
- `?:`: The ternary operator:

  ```
  {{ foo ? 'yes' : 'no' }}
  {{ foo ?: 'no' }} is the same as {{ foo ? foo : 'no' }}
  {{ foo ? 'yes' }} is the same as {{ foo ? 'yes' : '' }}
  ```

- `??`: The null-coalescing operator:

  ```
  {# returns the value of foo if it is defined and not null, 'no' otherwise #}
  {{ foo ?? 'no' }}
  ```

String Interpolation

String interpolation (`#{expression}`) allows any valid expression to appear within a *double-quoted string*. The result of evaluating that expression is inserted into the string:

```
{{ "foo #{bar} baz" }}
{{ "foo #{1 + 2} baz" }}
```

Whitespace Control

The first newline after a template tag is removed automatically (like in PHP.) Whitespace is not further modified by the template engine, so each whitespace (spaces, tabs, newlines etc.) is returned unchanged.

Use the `spaceless` tag to remove whitespace *between HTML tags*:

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

{# output will be <div><strong>foo bar</strong></div> #}
```

In addition to the spaceless tag you can also control whitespace on a per tag level. By using the whitespace control modifier on your tags, you can trim leading and or trailing whitespace:

```
{% set value = 'no spaces' %}
{#- No leading/trailing whitespace -#}
{%- if true -%}
    {{- value -}}
{%- endif -%}

{# output 'no spaces' #}
```

The above sample shows the default whitespace control modifier, and how you can use it to remove whitespace around tags. Trimming space will consume all whitespace for that side of the tag. It is possible to use whitespace trimming on one side of a tag:

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

{# outputs '<li>no spaces    </li>' #}
```

Extensions

Twig can be easily extended.

If you are looking for new tags, filters, or functions, have a look at the Twig official extension repository.

If you want to create your own, read the *Creating an Extension* chapter.

CHAPTER 4

Twig for Developers

This chapter describes the API to Twig and not the template language. It will be most useful as reference to those implementing the template interface to the application and not those who are creating Twig templates.

Basics

Twig uses a central object called the **environment** (of class `Twig_Environment`). Instances of this class are used to store the configuration and extensions, and are used to load templates from the file system or other locations.

Most applications will create one `Twig_Environment` object on application initialization and use that to load templates. In some cases it's however useful to have multiple environments side by side, if different configurations are in use.

The simplest way to configure Twig to load templates for your application looks roughly like this:

```
require_once '/path/to/vendor/autoload.php';

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

This will create a template environment with the default settings and a loader that looks up the templates in the `/path/to/templates/` folder. Different loaders are available and you can also write your own if you want to load templates from a database or other resources.

Note: Notice that the second argument of the environment is an array of options. The `cache` option is a compilation cache directory, where Twig caches the compiled templates to avoid the parsing phase for sub-sequent requests. It is very different from the cache you might want to add for the evaluated templates. For such a need, you can use any available PHP cache library.

Rendering Templates

To load a template from a Twig environment, call the `load()` method which returns a `Twig_TemplateWrapper` instance:

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

To render the template with some variables, call the `render()` method:

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

Note: The `display()` method is a shortcut to output the template directly.

You can also load and render the template in one fell swoop:

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

If a template defines blocks, they can be rendered individually via the `renderBlock()` call:

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

Environment Options

When creating a new `Twig_Environment` instance, you can pass an array of options as the constructor second argument:

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

The following options are available:

- `debug` *boolean*

  When set to `true`, the generated templates have a `__toString()` method that you can use to display the generated nodes (default to `false`).

- `charset` *string* (defaults to `utf-8`)

  The charset used by the templates.

- `base_template_class` *string* (defaults to `Twig_Template`)

  The base template class to use for generated templates.

- `cache` *string* or `false`

  An absolute path where to store the compiled templates, or `false` to disable caching (which is the default).

- `auto_reload` *boolean*

  When developing with Twig, it's useful to recompile the template whenever the source code changes. If you don't provide a value for the `auto_reload` option, it will be determined automatically based on the `debug` value.

- `strict_variables` *boolean*

If set to `false`, Twig will silently ignore invalid variables (variables and or attributes/methods that do not exist) and replace them with a `null` value. When set to `true`, Twig throws an exception instead (default to `false`).

- `autoescape` *string*

  Sets the default auto-escaping strategy (`name`, `html`, `js`, `css`, `url`, `html_attr`, or a PHP callback that takes the template "filename" and returns the escaping strategy to use – the callback cannot be a function name to avoid collision with built-in escaping strategies); set it to `false` to disable auto-escaping. The `name` escaping strategy determines the escaping strategy to use for a template based on the template filename extension (this strategy does not incur any overhead at runtime as auto-escaping is done at compilation time.)

- `optimizations` *integer*

  A flag that indicates which optimizations to apply (default to `-1` – all optimizations are enabled; set it to `0` to disable).

Loaders

Loaders are responsible for loading templates from a resource such as the file system.

Compilation Cache

All template loaders can cache the compiled templates on the filesystem for future reuse. It speeds up Twig a lot as templates are only compiled once; and the performance boost is even larger if you use a PHP accelerator such as APC. See the `cache` and `auto_reload` options of `Twig_Environment` above for more information.

Built-in Loaders

Here is a list of the built-in loaders Twig provides:

`Twig_Loader_Filesystem`

`Twig_Loader_Filesystem` loads templates from the file system. This loader can find templates in folders on the file system and is the preferred way to load them:

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

It can also look for templates in an array of directories:

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

With such a configuration, Twig will first look for templates in `$templateDir1` and if they do not exist, it will fallback to look for them in the `$templateDir2`.

You can add or prepend paths via the `addPath()` and `prependPath()` methods:

```
$loader->addPath($templateDir3);
$loader->prependPath($templateDir4);
```

The filesystem loader also supports namespaced templates. This allows to group your templates under different namespaces which have their own template paths.

When using the `setPaths()`, `addPath()`, and `prependPath()` methods, specify the namespace as the second argument (when not specified, these methods act on the "main" namespace):

```
$loader->addPath($templateDir, 'admin');
```

Namespaced templates can be accessed via the special `@namespace_name/template_path` notation:

```
$twig->render('@admin/index.html', array());
```

`Twig_Loader_Filesystem` support absolute and relative paths. Using relative paths is preferred as it makes the cache keys independent of the project root directory (for instance, it allows warming the cache from a build server where the directory might be different from the one used on production servers):

```
$loader = new Twig_Loader_Filesystem('templates', getcwd().'/..');
```

Note: When not passing the root path as a second argument, Twig uses `getcwd()` for relative paths.

Twig_Loader_Array

`Twig_Loader_Array` loads a template from a PHP array. It's passed an array of strings bound to template names:

```
$loader = new Twig_Loader_Array(array(
    'index.html' => 'Hello {{ name }}!',
));
$twig = new Twig_Environment($loader);

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

This loader is very useful for unit testing. It can also be used for small projects where storing all templates in a single PHP file might make sense.

Tip: When using the `Array` loader with a cache mechanism, you should know that a new cache key is generated each time a template content "changes" (the cache key being the source code of the template). If you don't want to see your cache grows out of control, you need to take care of clearing the old cache file by yourself.

Twig_Loader_Chain

`Twig_Loader_Chain` delegates the loading of templates to other loaders:

```
$loader1 = new Twig_Loader_Array(array(
    'base.html' => '{% block content %}{% endblock %}',
));
$loader2 = new Twig_Loader_Array(array(
    'index.html' => '{% extends "base.html" %}{% block content %}Hello {{ name }}{%␣
↪endblock %}',
    'base.html'  => 'Will never be loaded',
));

$loader = new Twig_Loader_Chain(array($loader1, $loader2));

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

When looking for a template, Twig will try each loader in turn and it will return as soon as the template is found. When rendering the `index.html` template from the above example, Twig will load it with `$loader2` but the `base.html` template will be loaded from `$loader1`.

`Twig_Loader_Chain` accepts any loader that implements `Twig_LoaderInterface`.

Note: You can also add loaders via the `addLoader()` method.

Create your own Loader

All loaders implement the `Twig_LoaderInterface`:

```
interface Twig_LoaderInterface
{
    /**
     * Returns the source context for a given template logical name.
     *
     * @param string $name The template logical name
     *
     * @return Twig_Source
     *
     * @throws Twig_Error_Loader When $name is not found
     */
    public function getSourceContext($name);

    /**
     * Gets the cache key to use for the cache for a given template name.
     *
     * @param string $name The name of the template to load
     *
     * @return string The cache key
     *
     * @throws Twig_Error_Loader When $name is not found
     */
    public function getCacheKey($name);

    /**
     * Returns true if the template is still fresh.
     *
     * @param string    $name The template name
     * @param timestamp $time The last modification time of the cached template
     *
     * @return bool    true if the template is fresh, false otherwise
     *
     * @throws Twig_Error_Loader When $name is not found
     */
    public function isFresh($name, $time);

    /**
     * Check if we have the source code of a template, given its name.
     *
     * @param string $name The name of the template to check if we can load
     *
     * @return bool    If the template source code is handled by this loader or not
     */
```

```
    public function exists($name);
}
```

The `isFresh()` method must return `true` if the current cached template is still fresh, given the last modification time, or `false` otherwise.

The `getSourceContext()` method must return an instance of `Twig_Source`.

Using Extensions

Twig extensions are packages that add new features to Twig. Using an extension is as simple as using the `addExtension()` method:

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

Twig comes bundled with the following extensions:

- *Twig_Extension_Core*: Defines all the core features of Twig.
- *Twig_Extension_Escaper*: Adds automatic output-escaping and the possibility to escape/unescape blocks of code.
- *Twig_Extension_Sandbox*: Adds a sandbox mode to the default Twig environment, making it safe to evaluate untrusted code.
- *Twig_Extension_Profiler*: Enabled the built-in Twig profiler.
- *Twig_Extension_Optimizer*: Optimizes the node tree before compilation.

The core, escaper, and optimizer extensions do not need to be added to the Twig environment, as they are registered by default.

Built-in Extensions

This section describes the features added by the built-in extensions.

Tip: Read the chapter about extending Twig to learn how to create your own extensions.

Core Extension

The `core` extension defines all the core features of Twig:

- *Tags*;
- *Filters*;
- *Functions*;
- *Tests*.

Escaper Extension

The `escaper` extension adds automatic output escaping to Twig. It defines a tag, `autoescape`, and a filter, `raw`.

When creating the escaper extension, you can switch on or off the global output escaping strategy:

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

If set to `html`, all variables in templates are escaped (using the `html` escaping strategy), except those using the `raw` filter:

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

You can also change the escaping mode locally by using the `autoescape` tag:

```
{% autoescape 'html' %}
    {{ var }}
    {{ var|raw }}      {# var won't be escaped #}
    {{ var|escape }}   {# var won't be double-escaped #}
{% endautoescape %}
```

Warning: The `autoescape` tag has no effect on included files.

The escaping rules are implemented as follows:

- Literals (integers, booleans, arrays, ...) used in the template directly as variables or filter arguments are never automatically escaped:

  ```
  {{ "Twig<br />" }} {# won't be escaped #}

  {% set text = "Twig<br />" %}
  {{ text }} {# will be escaped #}
  ```

- Expressions which the result is always a literal or a variable marked safe are never automatically escaped:

  ```
  {{ foo ? "Twig<br />" : "<br />Twig" }} {# won't be escaped #}

  {% set text = "Twig<br />" %}
  {{ foo ? text : "<br />Twig" }} {# will be escaped #}

  {% set text = "Twig<br />" %}
  {{ foo ? text|raw : "<br />Twig" }} {# won't be escaped #}

  {% set text = "Twig<br />" %}
  {{ foo ? text|escape : "<br />Twig" }} {# the result of the expression won't be␣
  ↪escaped #}
  ```

- Escaping is applied before printing, after any other filter is applied:

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

- The *raw* filter should only be used at the end of the filter chain:

  ```
  {{ var|raw|upper }} {# will be escaped #}

  {{ var|upper|raw }} {# won't be escaped #}
  ```

- Automatic escaping is not applied if the last filter in the chain is marked safe for the current context (e.g. `html` or `js`). `escape` and `escape('html')` are marked safe for HTML, `escape('js')` is marked safe for JavaScript, `raw` is marked safe for everything.

```
{% autoescape 'js' %}
    {{ var|escape('html') }} {# will be escaped for HTML and JavaScript #}
    {{ var }} {# will be escaped for JavaScript #}
    {{ var|escape('js') }} {# won't be double-escaped #}
{% endautoescape %}
```

Note: Note that autoescaping has some limitations as escaping is applied on expressions after evaluation. For instance, when working with concatenation, `{{ foo|raw ~ bar }}` won't give the expected result as escaping is applied on the result of the concatenation, not on the individual variables (so, the `raw` filter won't have any effect here).

Sandbox Extension

The `sandbox` extension can be used to evaluate untrusted code. Access to unsafe attributes and methods is prohibited. The sandbox security is managed by a policy instance. By default, Twig comes with one policy class: `Twig_Sandbox_SecurityPolicy`. This class allows you to white-list some tags, filters, properties, and methods:

```
$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);
```

With the previous configuration, the security policy will only allow usage of the `if` tag, and the `upper` filter. Moreover, the templates will only be able to call the `getTitle()` and `getBody()` methods on `Article` objects, and the `title` and `body` public properties. Everything else won't be allowed and will generate a `Twig_Sandbox_SecurityError` exception.

The policy object is the first argument of the sandbox constructor:

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

By default, the sandbox mode is disabled and should be enabled when including untrusted template code by using the `sandbox` tag:

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

You can sandbox all templates by passing `true` as the second argument of the extension constructor:

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

Profiler Extension

The `profiler` extension enables a profiler for Twig templates; it should only be used on your development machines as it adds some overhead:

```
$profile = new Twig_Profiler_Profile();
$twig->addExtension(new Twig_Extension_Profiler($profile));

$dumper = new Twig_Profiler_Dumper_Text();
echo $dumper->dump($profile);
```

A profile contains information about time and memory consumption for template, block, and macro executions.

You can also dump the data in a Blackfire.io compatible format:

```
$dumper = new Twig_Profiler_Dumper_Blackfire();
file_put_contents('/path/to/profile.prof', $dumper->dump($profile));
```

Upload the profile to visualize it (create a free account first):

```
blackfire --slot=7 upload /path/to/profile.prof
```

Optimizer Extension

The `optimizer` extension optimizes the node tree before compilation:

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

By default, all optimizations are turned on. You can select the ones you want to enable by passing them to the constructor:

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

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

Twig supports the following optimizations:

- `Twig_NodeVisitor_Optimizer::OPTIMIZE_ALL`, enables all optimizations (this is the default value).
- `Twig_NodeVisitor_Optimizer::OPTIMIZE_NONE`, disables all optimizations. This reduces the compilation time, but it can increase the execution time and the consumed memory.
- `Twig_NodeVisitor_Optimizer::OPTIMIZE_FOR`, optimizes the `for` tag by removing the `loop` variable creation whenever possible.
- `Twig_NodeVisitor_Optimizer::OPTIMIZE_RAW_FILTER`, removes the `raw` filter whenever possible.
- `Twig_NodeVisitor_Optimizer::OPTIMIZE_VAR_ACCESS`, simplifies the creation and access of variables in the compiled templates whenever possible.

Exceptions

Twig can throw exceptions:

- `Twig_Error`: The base exception for all errors.
- `Twig_Error_Syntax`: Thrown to tell the user that there is a problem with the template syntax.
- `Twig_Error_Runtime`: Thrown when an error occurs at runtime (when a filter does not exist for instance).
- `Twig_Error_Loader`: Thrown when an error occurs during template loading.
- `Twig_Sandbox_SecurityError`: Thrown when an unallowed tag, filter, or method is called in a sandboxed template.

CHAPTER 5

Extending Twig

Twig can be extended in many ways; you can add extra tags, filters, tests, operators, global variables, and functions. You can even extend the parser itself with node visitors.

Note: The first section of this chapter describes how to extend Twig easily. If you want to reuse your changes in different projects or if you want to share them with others, you should then create an extension as described in the following section.

Caution: When extending Twig without creating an extension, Twig won't be able to recompile your templates when the PHP code is updated. To see your changes in real-time, either disable template caching or package your code into an extension (see the next section of this chapter).

Before extending Twig, you must understand the differences between all the different possible extension points and when to use them.

First, remember that Twig has two main language constructs:

- `{{ }}`: used to print the result of an expression evaluation;
- `{% %}`: used to execute statements.

To understand why Twig exposes so many extension points, let's see how to implement a *Lorem ipsum* generator (it needs to know the number of words to generate).

You can use a `lipsum` *tag*:

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

That works, but using a tag for `lipsum` is not a good idea for at least three main reasons:

- `lipsum` is not a language construct;
- The tag outputs something;

- The tag is not flexible as you cannot use it in an expression:

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

In fact, you rarely need to create tags; and that's good news because tags are the most complex extension point of Twig.

Now, let's use a `lipsum` *filter*:

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

Again, it works, but it looks weird. A filter transforms the passed value to something else but here we use the value to indicate the number of words to generate (so, `40` is an argument of the filter, not the value we want to transform).

Next, let's use a `lipsum` *function*:

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

Here we go. For this specific example, the creation of a function is the extension point to use. And you can use it anywhere an expression is accepted:

```
{{ 'some text' ~ lipsum(40) ~ 'some more text' }}

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

Last but not the least, you can also use a *global* object with a method able to generate lorem ipsum text:

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

As a rule of thumb, use functions for frequently used features and global objects for everything else.

Keep in mind the following when you want to extend Twig:

What?	Implementation difficulty?	How often?	When?
macro	trivial	frequent	Content generation
global	trivial	frequent	Helper object
function	trivial	frequent	Content generation
filter	trivial	frequent	Value transformation
tag	complex	rare	DSL language construct
test	trivial	rare	Boolean decision
operator	trivial	rare	Values transformation

Globals

A global variable is like any other template variable, except that it's available in all templates and macros:

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

You can then use the `text` variable anywhere in a template:

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

Filters

Creating a filter is as simple as associating a name with a PHP callable:

```
// an anonymous function
$filter = new Twig_Filter('rot13', function ($string) {
    return str_rot13($string);
});

// or a simple PHP function
$filter = new Twig_Filter('rot13', 'str_rot13');

// or a class static method
$filter = new Twig_Filter('rot13', array('SomeClass', 'rot13Filter'));
$filter = new Twig_Filter('rot13', 'SomeClass::rot13Filter');

// or a class method
$filter = new Twig_Filter('rot13', array($this, 'rot13Filter'));
// the one below needs a runtime implementation (see below for more information)
$filter = new Twig_Filter('rot13', array('SomeClass', 'rot13Filter'));
```

The first argument passed to the `Twig_Filter` constructor is the name of the filter you will use in templates and the second one is the PHP callable to associate with it.

Then, add the filter to your Twig environment:

```
$twig = new Twig_Environment($loader);
$twig->addFilter($filter);
```

And here is how to use it in a template:

```
{{ 'Twig'|rot13 }}

{# will output Gjvt #}
```

When called by Twig, the PHP callable receives the left side of the filter (before the pipe `|`) as the first argument and the extra arguments passed to the filter (within parentheses `()`) as extra arguments.

For instance, the following code:

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

is compiled to something like the following:

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

The `Twig_Filter` class takes an array of options as its last argument:

```
$filter = new Twig_Filter('rot13', 'str_rot13', $options);
```

Environment-aware Filters

If you want to access the current environment instance in your filter, set the `needs_environment` option to `true`; Twig will pass the current environment as the first argument to the filter call:

```
$filter = new Twig_Filter('rot13', function (Twig_Environment $env, $string) {
    // get the current charset for instance
    $charset = $env->getCharset();

    return str_rot13($string);
}, array('needs_environment' => true));
```

Context-aware Filters

If you want to access the current context in your filter, set the `needs_context` option to `true`; Twig will pass the current context as the first argument to the filter call (or the second one if `needs_environment` is also set to `true`):

```
$filter = new Twig_Filter('rot13', function ($context, $string) {
    // ...
}, array('needs_context' => true));

$filter = new Twig_Filter('rot13', function (Twig_Environment $env, $context,
↪$string) {
    // ...
}, array('needs_context' => true, 'needs_environment' => true));
```

Automatic Escaping

If automatic escaping is enabled, the output of the filter may be escaped before printing. If your filter acts as an escaper (or explicitly outputs HTML or JavaScript code), you will want the raw output to be printed. In such a case, set the `is_safe` option:

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

Some filters may need to work on input that is already escaped or safe, for example when adding (safe) HTML tags to originally unsafe output. In such a case, set the `pre_escape` option to escape the input data before it is run through your filter:

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

Variadic Filters

When a filter should accept an arbitrary number of arguments, set the `is_variadic` option to `true`; Twig will pass the extra arguments as the last argument to the filter call as an array:

```
$filter = new Twig_Filter('thumbnail', function ($file, array $options = array()) {
    // ...
}, array('is_variadic' => true));
```

Be warned that named arguments passed to a variadic filter cannot be checked for validity as they will automatically end up in the option array.

Dynamic Filters

A filter name containing the special `*` character is a dynamic filter as the `*` can be any string:

```
$filter = new Twig_Filter('*_path', function ($name, $arguments) {
    // ...
});
```

The following filters will be matched by the above defined dynamic filter:

- `product_path`
- `category_path`

A dynamic filter can define more than one dynamic parts:

```
$filter = new Twig_Filter('*_path_*', function ($name, $suffix, $arguments) {
    // ...
});
```

The filter will receive all dynamic part values before the normal filter arguments, but after the environment and the context. For instance, a call to `'foo'|a_path_b()` will result in the following arguments to be passed to the filter: `('a', 'b', 'foo')`.

Deprecated Filters

You can mark a filter as being deprecated by setting the `deprecated` option to `true`. You can also give an alternative filter that replaces the deprecated one when that makes sense:

```
$filter = new Twig_Filter('obsolete', function () {
    // ...
}, array('deprecated' => true, 'alternative' => 'new_one'));
```

When a filter is deprecated, Twig emits a deprecation notice when compiling a template using it. See *Displaying Deprecation Notices* for more information.

Functions

Functions are defined in the exact same way as filters, but you need to create an instance of `Twig_Function`:

```
$twig = new Twig_Environment($loader);
$function = new Twig_Function('function_name', function () {
    // ...
});
$twig->addFunction($function);
```

Functions support the same features as filters, except for the `pre_escape` and `preserves_safety` options.

Tests

Tests are defined in the exact same way as filters and functions, but you need to create an instance of `Twig_Test`:

```
$twig = new Twig_Environment($loader);
$test = new Twig_Test('test_name', function () {
    // ...
});
$twig->addTest($test);
```

Tests allow you to create custom application specific logic for evaluating boolean conditions. As a simple example, let's create a Twig test that checks if objects are 'red':

```
$twig = new Twig_Environment($loader);
$test = new Twig_Test('red', function ($value) {
    if (isset($value->color) && $value->color == 'red') {
        return true;
    }
    if (isset($value->paint) && $value->paint == 'red') {
        return true;
    }
    return false;
});
$twig->addTest($test);
```

Test functions should always return true/false.

When creating tests you can use the `node_class` option to provide custom test compilation. This is useful if your test can be compiled into PHP primitives. This is used by many of the tests built into Twig:

```
$twig = new Twig_Environment($loader);
$test = new Twig_Test(
    'odd',
    null,
    array('node_class' => 'Twig_Node_Expression_Test_Odd'));
$twig->addTest($test);

class Twig_Node_Expression_Test_Odd extends Twig_Node_Expression_Test
{
    public function compile(Twig_Compiler $compiler)
    {
        $compiler
            ->raw('(')
            ->subcompile($this->getNode('node'))
            ->raw(' % 2 == 1')
            ->raw(')')
        ;
    }
}
```

The above example shows how you can create tests that use a node class. The node class has access to one sub-node called 'node'. This sub-node contains the value that is being tested. When the `odd` filter is used in code such as:

```
{% if my_value is odd %}
```

The `node` sub-node will contain an expression of `my_value`. Node-based tests also have access to the `arguments` node. This node will contain the various other arguments that have been provided to your test.

If you want to pass a variable number of positional or named arguments to the test, set the `is_variadic` option to `true`. Tests also support dynamic name feature as filters and functions.

Tags

One of the most exciting features of a template engine like Twig is the possibility to define new language constructs. This is also the most complex feature as you need to understand how Twig's internals work.

Let's create a simple `set` tag that allows the definition of simple variables from within a template. The tag can be used like follows:

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

{{ name }}

{# should output value #}
```

Note: The `set` tag is part of the Core extension and as such is always available. The built-in version is slightly more powerful and supports multiple assignments by default (cf. the template designers chapter for more information).

Three steps are needed to define a new tag:

- Defining a Token Parser class (responsible for parsing the template code);
- Defining a Node class (responsible for converting the parsed code to PHP);
- Registering the tag.

Registering a new tag

Adding a tag is as simple as calling the `addTokenParser` method on the `Twig_Environment` instance:

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

Defining a Token Parser

Now, let's see the actual code of this class:

```
class Project_Set_TokenParser extends Twig_TokenParser
{
    public function parse(Twig_Token $token)
    {
        $parser = $this->parser;
        $stream = $parser->getStream();

        $name = $stream->expect(Twig_Token::NAME_TYPE)->getValue();
        $stream->expect(Twig_Token::OPERATOR_TYPE, '=');
        $value = $parser->getExpressionParser()->parseExpression();
        $stream->expect(Twig_Token::BLOCK_END_TYPE);

        return new Project_Set_Node($name, $value, $token->getLine(), $this->
↪getTag());
    }

    public function getTag()
    {
```

```
        return 'set';
    }
}
```

The `getTag()` method must return the tag we want to parse, here `set`.

The `parse()` method is invoked whenever the parser encounters a `set` tag. It should return a `Twig_Node` instance that represents the node (the `Project_Set_Node` calls creating is explained in the next section).

The parsing process is simplified thanks to a bunch of methods you can call from the token stream (`$this->parser->getStream()`):

- `getCurrent()`: Gets the current token in the stream.
- `next()`: Moves to the next token in the stream, *but returns the old one*.
- `test($type)`, `test($value)` or `test($type, $value)`: Determines whether the current token is of a particular type or value (or both). The value may be an array of several possible values.
- `expect($type[, $value[, $message]])`: If the current token isn't of the given type/value a syntax error is thrown. Otherwise, if the type and value are correct, the token is returned and the stream moves to the next token.
- `look()`: Looks at the next token without consuming it.

Parsing expressions is done by calling the `parseExpression()` like we did for the `set` tag.

Tip: Reading the existing `TokenParser` classes is the best way to learn all the nitty-gritty details of the parsing process.

Defining a Node

The `Project_Set_Node` class itself is rather simple:

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

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

The compiler implements a fluid interface and provides methods that helps the developer generate beautiful and readable PHP code:

- `subcompile()`: Compiles a node.

- `raw()`: Writes the given string as is.
- `write()`: Writes the given string by adding indentation at the beginning of each line.
- `string()`: Writes a quoted string.
- `repr()`: Writes a PHP representation of a given value (see `Twig_Node_For` for a usage example).
- `addDebugInfo()`: Adds the line of the original template file related to the current node as a comment.
- `indent()`: Indents the generated code (see `Twig_Node_Block` for a usage example).
- `outdent()`: Outdents the generated code (see `Twig_Node_Block` for a usage example).

Creating an Extension

The main motivation for writing an extension is to move often used code into a reusable class like adding support for internationalization. An extension can define tags, filters, tests, operators, global variables, functions, and node visitors.

Most of the time, it is useful to create a single extension for your project, to host all the specific tags and filters you want to add to Twig.

Tip: When packaging your code into an extension, Twig is smart enough to recompile your templates whenever you make a change to it (when `auto_reload` is enabled).

Note: Before writing your own extensions, have a look at the Twig official extension repository: http://github.com/twigphp/Twig-extensions.

An extension is a class that implements the following interface:

```
interface Twig_ExtensionInterface
{
    /**
     * Returns the token parser instances to add to the existing list.
     *
     * @return Twig_TokenParserInterface[]
     */
    public function getTokenParsers();

    /**
     * Returns the node visitor instances to add to the existing list.
     *
     * @return Twig_NodeVisitorInterface[]
     */
    public function getNodeVisitors();

    /**
     * Returns a list of filters to add to the existing list.
     *
     * @return Twig_Filter[]
     */
    public function getFilters();

    /**
```

```
     * Returns a list of tests to add to the existing list.
     *
     * @return Twig_Test[]
     */
    public function getTests();

    /**
     * Returns a list of functions to add to the existing list.
     *
     * @return Twig_Function[]
     */
    public function getFunctions();

    /**
     * Returns a list of operators to add to the existing list.
     *
     * @return array<array> First array of unary operators, second array of binary␣
↪operators
     */
    public function getOperators();
}
```

To keep your extension class clean and lean, inherit from the built-in `Twig_Extension` class instead of implementing the interface as it provides empty implementations for all methods:

```
class Project_Twig_Extension extends Twig_Extension { }
```

Of course, this extension does nothing for now. We will customize it in the next sections.

Twig does not care where you save your extension on the filesystem, as all extensions must be registered explicitly to be available in your templates.

You can register an extension by using the `addExtension()` method on your main `Environment` object:

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

Tip: The Twig core extensions are great examples of how extensions work.

Globals

Global variables can be registered in an extension via the `getGlobals()` method:

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

    // ...
}
```

Functions

Functions can be registered in an extension via the `getFunctions()` method:

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

    // ...
}
```

Filters

To add a filter to an extension, you need to override the `getFilters()` method. This method must return an array of filters to add to the Twig environment:

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

    // ...
}
```

Tags

Adding a tag in an extension can be done by overriding the `getTokenParsers()` method. This method must return an array of tags to add to the Twig environment:

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

    // ...
}
```

In the above code, we have added a single new tag, defined by the `Project_Set_TokenParser` class. The `Project_Set_TokenParser` class is responsible for parsing the tag and compiling it to PHP.

Operators

The `getOperators()` methods lets you add new operators. Here is how to add `!`, `||`, and `&&` operators:

```
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),
            ),
        );
    }

    // ...
}
```

Tests

The `getTests()` method lets you add new test functions:

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

    // ...
}
```

Definition vs Runtime

Twig filters, functions, and tests runtime implementations can be defined as any valid PHP callable:

- **functions/static methods**: Simple to implement and fast (used by all Twig core extensions); but it is hard for the runtime to depend on external objects;
- **closures**: Simple to implement;
- **object methods**: More flexible and required if your runtime code depends on external objects.

The simplest way to use methods is to define them on the extension itself:

```
class Project_Twig_Extension extends Twig_Extension
{
    private $rot13Provider;

    public function __construct($rot13Provider)
    {
```

```
        $this->rot13Provider = $rot13Provider;
    }

    public function getFunctions()
    {
        return array(
            new Twig_Function('rot13', array($this, 'rot13')),
        );
    }

    public function rot13($value)
    {
        return $rot13Provider->rot13($value);
    }
}
```

This is very convenient but not recommended as it makes template compilation depend on runtime dependencies even if they are not needed (think for instance as a dependency that connects to a database engine).

You can easily decouple the extension definitions from their runtime implementations by registering a `Twig_RuntimeLoaderInterface` instance on the environment that knows how to instantiate such runtime classes (runtime classes must be autoload-able):

```
class RuntimeLoader implements Twig_RuntimeLoaderInterface
{
    public function load($class)
    {
        // implement the logic to create an instance of $class
        // and inject its dependencies
        // most of the time, it means using your dependency injection container
        if ('Project_Twig_RuntimeExtension' === $class) {
            return new $class(new Rot13Provider());
        } else {
            // ...
        }
    }
}

$twig->addRuntimeLoader(new RuntimeLoader());
```

Note: Twig comes with a PSR-11 compatible runtime loader (`Twig_ContainerRuntimeLoader`).

It is now possible to move the runtime logic to a new `Project_Twig_RuntimeExtension` class and use it directly in the extension:

```
class Project_Twig_RuntimeExtension
{
    private $rot13Provider;

    public function __construct($rot13Provider)
    {
        $this->rot13Provider = $rot13Provider;
    }

    public function rot13($value)
    {
```

```
        return $rot13Provider->rot13($value);
    }
}

class Project_Twig_Extension extends Twig_Extension
{
    public function getFunctions()
    {
        return array(
            new Twig_Function('rot13', array('Project_Twig_RuntimeExtension', 'rot13
↪')),
            // or
            new Twig_Function('rot13', 'Project_Twig_RuntimeExtension::rot13'),
        );
    }
}
```

Overloading

To overload an already defined filter, test, operator, global variable, or function, re-define it in an extension and register it **as late as possible** (order matters):

```
class MyCoreExtension extends Twig_Extension
{
    public function getFilters()
    {
        return array(
            new Twig_Filter('date', array($this, 'dateFilter')),
        );
    }

    public function dateFilter($timestamp, $format = 'F j, Y H:i')
    {
        // do something different from the built-in date filter
    }
}

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

Here, we have overloaded the built-in `date` filter with a custom one.

If you do the same on the `Twig_Environment` itself, beware that it takes precedence over any other registered extensions:

```
$twig = new Twig_Environment($loader);
$twig->addFilter(new Twig_Filter('date', function ($timestamp, $format = 'F j, Y H:i
↪') {
    // do something different from the built-in date filter
}));
// the date filter will come from the above registration, not
// from the registered extension below
$twig->addExtension(new MyCoreExtension());
```

Caution: Note that overloading the built-in Twig elements is not recommended as it might be confusing.

Testing an Extension

Functional Tests

You can create functional tests for extensions simply by creating the following file structure in your test directory:

```
Fixtures/
    filters/
        foo.test
        bar.test
    functions/
        foo.test
        bar.test
    tags/
        foo.test
        bar.test
IntegrationTest.php
```

The `IntegrationTest.php` file should look like this:

```
class Project_Tests_IntegrationTest extends Twig_Test_IntegrationTestCase
{
    public function getExtensions()
    {
        return array(
            new Project_Twig_Extension1(),
            new Project_Twig_Extension2(),
        );
    }

    public function getFixturesDir()
    {
        return dirname(__FILE__).'/Fixtures/';
    }
}
```

Fixtures examples can be found within the Twig repository tests/Twig/Fixtures directory.

Node Tests

Testing the node visitors can be complex, so extend your test cases from `Twig_Test_NodeTestCase`. Examples can be found in the Twig repository tests/Twig/Node directory.

CHAPTER 6

Twig Internals

Twig is very extensible and you can easily hack it. Keep in mind that you should probably try to create an extension before hacking the core, as most features and enhancements can be handled with extensions. This chapter is also useful for people who want to understand how Twig works under the hood.

How does Twig work?

The rendering of a Twig template can be summarized into four key steps:

- **Load** the template: If the template is already compiled, load it and go to the *evaluation* step, otherwise:
  - First, the **lexer** tokenizes the template source code into small pieces for easier processing;
  - Then, the **parser** converts the token stream into a meaningful tree of nodes (the Abstract Syntax Tree);
  - Eventually, the *compiler* transforms the AST into PHP code.
- **Evaluate** the template: It basically means calling the `display()` method of the compiled template and passing it the context.

The Lexer

The lexer tokenizes a template source code into a token stream (each token is an instance of `Twig_Token`, and the stream is an instance of `Twig_TokenStream`). The default lexer recognizes 13 different token types:

- `Twig_Token::BLOCK_START_TYPE`, `Twig_Token::BLOCK_END_TYPE`: Delimiters for blocks (`{% %}`)
- `Twig_Token::VAR_START_TYPE`, `Twig_Token::VAR_END_TYPE`: Delimiters for variables (`{{ }}`)
- `Twig_Token::TEXT_TYPE`: A text outside an expression;
- `Twig_Token::NAME_TYPE`: A name in an expression;
- `Twig_Token::NUMBER_TYPE`: A number in an expression;

- `Twig_Token::STRING_TYPE`: A string in an expression;
- `Twig_Token::OPERATOR_TYPE`: An operator;
- `Twig_Token::PUNCTUATION_TYPE`: A punctuation sign;
- `Twig_Token::INTERPOLATION_START_TYPE`, `Twig_Token::INTERPOLATION_END_TYPE`: Delimiters for string interpolation;
- `Twig_Token::EOF_TYPE`: Ends of template.

You can manually convert a source code into a token stream by calling the `tokenize()` method of an environment:

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

As the stream has a `__toString()` method, you can have a textual representation of it by echoing the object:

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

Here is the output for the `Hello {{ name }}` template:

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

Note: The default lexer (`Twig_Lexer`) can be changed by calling the `setLexer()` method:

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

The Parser

The parser converts the token stream into an AST (Abstract Syntax Tree), or a node tree (an instance of `Twig_Node_Module`). The core extension defines the basic nodes like: `for`, `if`, ... and the expression nodes.

You can manually convert a token stream into a node tree by calling the `parse()` method of an environment:

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

Echoing the node object gives you a nice representation of the tree:

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

Here is the output for the `Hello {{ name }}` template:

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

Note: The default parser (`Twig_TokenParser`) can be changed by calling the `setParser()` method:

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

The Compiler

The last step is done by the compiler. It takes a node tree as an input and generates PHP code usable for runtime execution of the template.

You can manually compile a node tree to PHP code with the `compile()` method of an environment:

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

The generated template for a `Hello {{ name }}` template reads as follows (the actual output can differ depending on the version of Twig you are using):

```
/* Hello {{ name }} */
class __TwigTemplate_1121b6f109fe93ebe8c6e22e3712bceb extends Twig_Template
{
    protected function doDisplay(array $context, array $blocks = array())
    {
        // line 1
        echo "Hello ";
        echo twig_escape_filter($this->env, (isset($context["name"]) ? $context["name
↪"] : null), "html", null, true);
    }

    // some more code
}
```

Note: The default compiler (`Twig_Compiler`) can be changed by calling the `setCompiler()` method:

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

CHAPTER 7

Recipes

Displaying Deprecation Notices

Deprecated features generate deprecation notices (via a call to the `trigger_error()` PHP function). By default, they are silenced and never displayed nor logged.

To easily remove all deprecated feature usages from your templates, write and run a script along the lines of the following:

```
require_once __DIR__.'/vendor/autoload.php';

$twig = create_your_twig_env();

$deprecations = new Twig_Util_DeprecationCollector($twig);

print_r($deprecations->collectDir(__DIR__.'/templates'));
```

The `collectDir()` method compiles all templates found in a directory, catches deprecation notices, and return them.

Tip: If your templates are not stored on the filesystem, use the `collect()` method instead. `collect()` takes a `Traversable` which must return template names as keys and template contents as values (as done by `Twig_Util_TemplateDirIterator`).

However, this code won't find all deprecations (like using deprecated some Twig classes). To catch all notices, register a custom error handler like the one below:

```
$deprecations = array();
set_error_handler(function ($type, $msg) use (&$deprecations) {
    if (E_USER_DEPRECATED === $type) {
        $deprecations[] = $msg;
    }
});
```

```
// run your application

print_r($deprecations);
```

Note that most deprecation notices are triggered during **compilation**, so they won't be generated when templates are already cached.

Tip: If you want to manage the deprecation notices from your PHPUnit tests, have a look at the symfony/phpunit-bridge package, which eases the process a lot.

Making a Layout conditional

Working with Ajax means that the same content is sometimes displayed as is, and sometimes decorated with a layout. As Twig layout template names can be any valid expression, you can pass a variable that evaluates to `true` when the request is made via Ajax and choose the layout accordingly:

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

{% block content %}
    This is the content to be displayed.
{% endblock %}
```

Making an Include dynamic

When including a template, its name does not need to be a string. For instance, the name can depend on the value of a variable:

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

If `var` evaluates to `index`, the `index_foo.html` template will be rendered.

As a matter of fact, the template name can be any valid expression, such as the following:

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

Overriding a Template that also extends itself

A template can be customized in two different ways:

- *Inheritance*: A template *extends* a parent template and overrides some blocks;
- *Replacement*: If you use the filesystem loader, Twig loads the first template it finds in a list of configured directories; a template found in a directory *replaces* another one from a directory further in the list.

But how do you combine both: *replace* a template that also extends itself (aka a template in a directory further in the list)?

Let's say that your templates are loaded from both `.../templates/mysite` and `.../templates/default` in this order. The `page.twig` template, stored in `.../templates/default` reads as follows:

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

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

You can replace this template by putting a file with the same name in `.../templates/mysite`. And if you want to extend the original template, you might be tempted to write the following:

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

Of course, this will not work as Twig will always load the template from `.../templates/mysite`.

It turns out it is possible to get this to work, by adding a directory right at the end of your template directories, which is the parent of all of the other directories: `.../templates` in our case. This has the effect of making every template file within our system uniquely addressable. Most of the time you will use the "normal" paths, but in the special case of wanting to extend a template with an overriding version of itself we can reference its parent's full, unambiguous template path in the extends tag:

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

Note: This recipe was inspired by the following Django wiki page: http://code.djangoproject.com/wiki/ExtendingTemplates

Customizing the Syntax

Twig allows some syntax customization for the block delimiters. It's not recommended to use this feature as templates will be tied with your custom syntax. But for specific projects, it can make sense to change the defaults.

To change the block delimiters, you need to create your own lexer object:

```
$twig = new Twig_Environment(...);

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

Here are some configuration example that simulates some other template engines syntax:

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

// SGML Comment Syntax
$lexer = new Twig_Lexer($twig, array(
```

```
    'tag_comment'  => array('<!--#', '-->'),
    'tag_block'    => array('<!--', '-->'),
    'tag_variable' => array('${', '}'),
));

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

Using dynamic Object Properties

When Twig encounters a variable like `article.title`, it tries to find a `title` public property in the `article` object.

It also works if the property does not exist but is rather defined dynamically thanks to the magic `__get()` method; you just need to also implement the `__isset()` magic method like shown in the following snippet of code:

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

        // throw some kind of error
    }

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

        return false;
    }
}
```

Accessing the parent Context in Nested Loops

Sometimes, when using nested loops, you need to access the parent context. The parent context is always accessible via the `loop.parent` variable. For instance, if you have the following template data:

```
$data = array(
    'topics' => array(
        'topic1' => array('Message 1 of topic 1', 'Message 2 of topic 1'),
        'topic2' => array('Message 1 of topic 2', 'Message 2 of topic 2'),
    ),
);
```

And the following template to display all messages in all topics:

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

The output will be similar to:

```
* 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
```

In the inner loop, the `loop.parent` variable is used to access the outer context. So, the index of the current `topic` defined in the outer for loop is accessible via the `loop.parent.loop.index` variable.

Defining undefined Functions and Filters on the Fly

When a function (or a filter) is not defined, Twig defaults to throw a `Twig_Error_Syntax` exception. However, it can also call a callback (any valid PHP callable) which should return a function (or a filter).

For filters, register callbacks with `registerUndefinedFilterCallback()`. For functions, use `registerUndefinedFunctionCallback()`:

```
// auto-register all native PHP functions as Twig functions
// don't try this at home as it's not secure at all!
$twig->registerUndefinedFunctionCallback(function ($name) {
    if (function_exists($name)) {
        return new Twig_Function($name, $name);
    }

    return false;
});
```

If the callable is not able to return a valid function (or filter), it must return `false`.

If you register more than one callback, Twig will call them in turn until one does not return `false`.

Tip: As the resolution of functions and filters is done during compilation, there is no overhead when registering these callbacks.

Validating the Template Syntax

When template code is provided by a third-party (through a web interface for instance), it might be interesting to validate the template syntax before saving it. If the template code is stored in a *$template* variable, here is how you can do it:

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

    // the $template is valid
} catch (Twig_Error_Syntax $e) {
    // $template contains one or more syntax errors
}
```

If you iterate over a set of files, you can pass the filename to the `tokenize()` method to get the filename in the exception message:

```
foreach ($files as $file) {
    try {
        $twig->parse($twig->tokenize(new Twig_Source($template, $file->getFilename(),
↪$file)));

        // the $template is valid
    } catch (Twig_Error_Syntax $e) {
        // $template contains one or more syntax errors
    }
}
```

Note: This method won't catch any sandbox policy violations because the policy is enforced during template rendering (as Twig needs the context for some checks like allowed methods on objects).

Refreshing modified Templates when OPcache or APC is enabled

When using OPcache with `opcache.validate_timestamps` set to `0` or APC with `apc.stat` set to `0` and Twig cache enabled, clearing the template cache won't update the cache.

To get around this, force Twig to invalidate the bytecode cache:

```
$twig = new Twig_Environment($loader, array(
    'cache' => new Twig_Cache_Filesystem('/some/cache/path', Twig_Cache_
↪Filesystem::FORCE_BYTECODE_INVALIDATION),
    // ...
));
```

Reusing a stateful Node Visitor

When attaching a visitor to a `Twig_Environment` instance, Twig uses it to visit *all* templates it compiles. If you need to keep some state information around, you probably want to reset it when visiting a new template.

This can be easily achieved with the following code:

```
protected $someTemplateState = array();

public function enterNode(Twig_Node $node, Twig_Environment $env)
{
    if ($node instanceof Twig_Node_Module) {
        // reset the state as we are entering a new template
```

```
        $this->someTemplateState = array();
    }

    // ...

    return $node;
}
```

Using a Database to store Templates

If you are developing a CMS, templates are usually stored in a database. This recipe gives you a simple PDO template loader you can use as a starting point for your own.

First, let's create a temporary in-memory SQLite3 database to work with:

```
$dbh = new PDO('sqlite::memory:');
$dbh->exec('CREATE TABLE templates (name STRING, source STRING, last_modified INTEGER)
↪');
$base = '{% block content %}{% endblock %}';
$index = '
{% extends "base.twig" %}
{% block content %}Hello {{ name }}{% endblock %}
';
$now = time();
$dbh->exec("INSERT INTO templates (name, source, last_modified) VALUES ('base.twig', '
↪$base', $now)");
$dbh->exec("INSERT INTO templates (name, source, last_modified) VALUES ('index.twig',
↪'$index', $now)");
```

We have created a simple `templates` table that hosts two templates: `base.twig` and `index.twig`.

Now, let's define a loader able to use this database:

```
class DatabaseTwigLoader implements Twig_LoaderInterface
{
    protected $dbh;

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

    public function getSourceContext($name)
    {
        if (false === $source = $this->getValue('source', $name)) {
            throw new Twig_Error_Loader(sprintf('Template "%s" does not exist.',
↪$name));
        }

        return new Twig_Source($source, $name);
    }

    public function exists($name)
    {
        return $name === $this->getValue('name', $name);
    }
```

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

    public function isFresh($name, $time)
    {
        if (false === $lastModified = $this->getValue('last_modified', $name)) {
            return false;
        }

        return $lastModified <= $time;
    }

    protected function getValue($column, $name)
    {
        $sth = $this->dbh->prepare('SELECT '.$column.' FROM templates WHERE name =␣
↪:name');
        $sth->execute(array(':name' => (string) $name));

        return $sth->fetchColumn();
    }
}
```

Finally, here is an example on how you can use it:

```
$loader = new DatabaseTwigLoader($dbh);
$twig = new Twig_Environment($loader);

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

Using different Template Sources

This recipe is the continuation of the previous one. Even if you store the contributed templates in a database, you might want to keep the original/base templates on the filesystem. When templates can be loaded from different sources, you need to use the `Twig_Loader_Chain` loader.

As you can see in the previous recipe, we reference the template in the exact same way as we would have done it with a regular filesystem loader. This is the key to be able to mix and match templates coming from the database, the filesystem, or any other loader for that matter: the template name should be a logical name, and not the path from the filesystem:

```
$loader1 = new DatabaseTwigLoader($dbh);
$loader2 = new Twig_Loader_Array(array(
    'base.twig' => '{% block content %}{% endblock %}',
));
$loader = new Twig_Loader_Chain(array($loader1, $loader2));

$twig = new Twig_Environment($loader);

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

Now that the `base.twig` templates is defined in an array loader, you can remove it from the database, and everything else will still work as before.

Loading a Template from a String

From a template, you can easily load a template stored in a string via the `template_from_string` function (via the `Twig_Extension_StringLoader` extension):

```
{{ include(template_from_string("Hello {{ name }}")) }}
```

From PHP, it's also possible to load a template stored in a string via `Twig_Environment::createTemplate()`:

```
$template = $twig->createTemplate('hello {{ name }}');
echo $template->render(array('name' => 'Fabien'));
```

Using Twig and AngularJS in the same Templates

Mixing different template syntaxes in the same file is not a recommended practice as both AngularJS and Twig use the same delimiters in their syntax: `{{` and `}}`.

Still, if you want to use AngularJS and Twig in the same template, there are two ways to make it work depending on the amount of AngularJS you need to include in your templates:

- Escaping the AngularJS delimiters by wrapping AngularJS sections with the `{% verbatim %}` tag or by escaping each delimiter via `{{ '{{' }}` and `{{ '}}' }}`;
- Changing the delimiters of one of the template engines (depending on which engine you introduced last):
  - For AngularJS, change the interpolation tags using the `interpolateProvider` service, for instance at the module initialization time:

    ```
    angular.module('myApp', []).config(function($interpolateProvider) {
        $interpolateProvider.startSymbol('{[').endSymbol(']}');
    });
    ```

  - For Twig, change the delimiters via the `tag_variable` Lexer option:

    ```
    $env->setLexer(new Twig_Lexer($env, array(
        'tag_variable' => array('{[', ']}'),
    )));
    ```

CHAPTER 8

Coding Standards

When writing Twig templates, we recommend you to follow these official coding standards:

- Put one (and only one) space after the start of a delimiter (`{{`, `{%`, and `{#`) and before the end of a delimiter (`}}`, `%}`, and `#}`):

```
{{ foo }}
{# comment #}
{% if foo %}{% endif %}
```

  When using the whitespace control character, do not put any spaces between it and the delimiter:

```
{{- foo -}}
{#- comment -#}
{%- if foo -%}{%- endif -%}
```

- Put one (and only one) space before and after the following operators: comparison operators (`==`, `!=`, `<`, `>`, `>=`, `<=`), math operators (`+`, `-`, `/`, `*`, `%`, `//`, `**`), logic operators (`not`, `and`, `or`), `~`, `is`, `in`, and the ternary operator (`?:`):

```
{{ 1 + 2 }}
{{ foo ~ bar }}
{{ true ? true : false }}
```

- Put one (and only one) space after the `:` sign in hashes and `,` in arrays and hashes:

```
{{ [1, 2, 3] }}
{{ {'foo': 'bar'} }}
```

- Do not put any spaces after an opening parenthesis and before a closing parenthesis in expressions:

```
{{ 1 + (2 * 3) }}
```

- Do not put any spaces before and after string delimiters:

```
{{ 'foo' }}
{{ "foo" }}
```

- Do not put any spaces before and after the following operators: `|`, `.`, `..`, `[]`:

```
{{ foo|upper|lower }}
{{ user.name }}
{{ user[name] }}
{% for i in 1..12 %}{% endfor %}
```

- Do not put any spaces before and after the parenthesis used for filter and function calls:

```
{{ foo|default('foo') }}
{{ range(1..10) }}
```

- Do not put any spaces before and after the opening and the closing of arrays and hashes:

```
{{ [1, 2, 3] }}
{{ {'foo': 'bar'} }}
```

- Use lower cased and underscored variable names:

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

- Indent your code inside tags (use the same indentation as the one used for the target language of the rendered template):

```
{% block foo %}
    {% if true %}
        true
    {% endif %}
{% endblock %}
```

CHAPTER 9

Tags

autoescape

Whether automatic escaping is enabled or not, you can mark a section of a template to be escaped or not by using the `autoescape` tag:

```
{% autoescape %}
    Everything will be automatically escaped in this block
    using the HTML strategy
{% endautoescape %}

{% autoescape 'html' %}
    Everything will be automatically escaped in this block
    using the HTML strategy
{% endautoescape %}

{% autoescape 'js' %}
    Everything will be automatically escaped in this block
    using the js escaping strategy
{% endautoescape %}

{% autoescape false %}
    Everything will be outputted as is in this block
{% endautoescape %}
```

When automatic escaping is enabled everything is escaped by default except for values explicitly marked as safe. Those can be marked in the template by using the *raw* filter:

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

Functions returning template data (like *macros* and *parent*) always return safe markup.

Note: Twig is smart enough to not escape an already escaped value by the *escape* filter.

Note: Twig does not escape static expressions:

```
{% set hello = "<strong>Hello</strong>" %}
{{ hello }}
{{ "<strong>world</strong>" }}
```

Will be rendered "Hello **world**".

Note: The chapter *Twig for Developers* gives more information about when and how automatic escaping is applied.

block

Blocks are used for inheritance and act as placeholders and replacements at the same time. They are documented in detail in the documentation for the *extends* tag.

Block names should consist of alphanumeric characters, and underscores. Dashes are not permitted.

See also:

block, *parent*, *use*, *extends*

do

The `do` tag works exactly like the regular variable expression (`{{ ... }}`) just that it doesn't print anything:

```
{% do 1 + 2 %}
```

embed

The `embed` tag combines the behaviour of *include* and *extends*. It allows you to include another template's contents, just like `include` does. But it also allows you to override any block defined inside the included template, like when extending a template.

Think of an embedded template as a "micro layout skeleton".

```
{% embed "teasers_skeleton.twig" %}
    {# These blocks are defined in "teasers_skeleton.twig" #}
    {# and we override them right here:                    #}
    {% block left_teaser %}
        Some content for the left teaser box
    {% endblock %}
    {% block right_teaser %}
        Some content for the right teaser box
    {% endblock %}
{% endembed %}
```

The `embed` tag takes the idea of template inheritance to the level of content fragments. While template inheritance allows for "document skeletons", which are filled with life by child templates, the `embed` tag allows you to create "skeletons" for smaller units of content and re-use and fill them anywhere you like.

Since the use case may not be obvious, let's look at a simplified example. Imagine a base template shared by multiple HTML pages, defining a single block named "content":

```
-- page layout --------------
|                                  |
|           - block "content" -    |
|           |                  |   |
|           |                  |   |
|           | (child template to   |   |
|           |  put content here)   |   |
|           |                  |   |
|           |                  |   |
|           --------------    |
|                                  |
--------------------------
```

Some pages ("foo" and "bar") share the same content structure - two vertically stacked boxes:

```
-- page layout --------------
|                                  |
|           - block "content" -    |
|           | - block "top" -- |   |
|           | |                | | |   |
|           | ----------- |    |
|           | - block "bottom"  |   |
|           | |                | | |   |
|           | ----------- |    |
|           --------------    |
|                                  |
--------------------------
```

While other pages ("boom" and "baz") share a different content structure - two boxes side by side:

```
-- page layout --------------
|                                  |
|           - block "content" -    |
|           |                  |   |
|           |  block   block   |   |
|           | |"left" | |"right"| |   |
|           | |       | |       | |   |
|           | |       | |       | |   |
|           | ----- ----- |    |
|           --------------    |
|                                  |
--------------------------
```

Without the `embed` tag, you have two ways to design your templates:

- Create two "intermediate" base templates that extend the master layout template: one with vertically stacked boxes to be used by the "foo" and "bar" pages and another one with side-by-side boxes for the "boom" and "baz" pages.
- Embed the markup for the top/bottom and left/right boxes into each page template directly.

These two solutions do not scale well because they each have a major drawback:

- The first solution may indeed work for this simplified example. But imagine we add a sidebar, which may again contain different, recurring structures of content. Now we would need to create intermediate base templates for all occurring combinations of content structure and sidebar structure... and so on.
- The second solution involves duplication of common code with all its negative consequences: any change involves finding and editing all affected copies of the structure, correctness has to be verified for each copy, copies may go out of sync by careless modifications etc.

In such a situation, the `embed` tag comes in handy. The common layout code can live in a single base template, and the two different content structures, let's call them "micro layouts" go into separate templates which are embedded as necessary:

Page template `foo.twig`:

```
{% extends "layout_skeleton.twig" %}

{% block content %}
    {% embed "vertical_boxes_skeleton.twig" %}
        {% block top %}
            Some content for the top box
        {% endblock %}

        {% block bottom %}
            Some content for the bottom box
        {% endblock %}
    {% endembed %}
{% endblock %}
```

And here is the code for `vertical_boxes_skeleton.twig`:

```
<div class="top_box">
    {% block top %}
        Top box default content
    {% endblock %}
</div>

<div class="bottom_box">
    {% block bottom %}
        Bottom box default content
    {% endblock %}
</div>
```

The goal of the `vertical_boxes_skeleton.twig` template being to factor out the HTML markup for the boxes.

The `embed` tag takes the exact same arguments as the `include` tag:

```
{% embed "base" with {'foo': 'bar'} %}
    ...
{% endembed %}

{% embed "base" with {'foo': 'bar'} only %}
    ...
{% endembed %}

{% embed "base" ignore missing %}
    ...
{% endembed %}
```

Warning: As embedded templates do not have "names", auto-escaping strategies based on the template name won't work as expected if you change the context (for instance, if you embed a CSS/JavaScript template into an HTML one). In that case, explicitly set the default auto-escaping strategy with the `autoescape` tag.

See also:

include

extends

The `extends` tag can be used to extend a template from another one.

Note: Like PHP, Twig does not support multiple inheritance. So you can only have one extends tag called per rendering. However, Twig supports horizontal *reuse*.

Let's define a base template, `base.html`, which defines a simple HTML skeleton document:

```
<!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>
```

In this example, the *block* tags define four blocks that child templates can fill in.

All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

Child Template

A child template might look like this:

```
{% 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 %}
```

The `extends` tag is the key here. It tells the template engine that this template "extends" another template. When the template system evaluates this template, first it locates the parent. The extends tag should be the first tag in the template.

Note that since the child template doesn't define the `footer` block, the value from the parent template is used instead.

You can't define multiple `block` tags with the same name in the same template. This limitation exists because a block tag works in "both" directions. That is, a block tag doesn't just provide a hole to fill - it also defines the content that fills the hole in the *parent*. If there were two similarly-named `block` tags in a template, that template's parent wouldn't know which one of the blocks' content to use.

If you want to print a block multiple times you can however use the `block` function:

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

Parent Blocks

It's possible to render the contents of the parent block by using the *parent* function. This gives back the results of the parent block:

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

Named Block End-Tags

Twig allows you to put the name of the block after the end tag for better readability:

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

Of course, the name after the `endblock` word must match the block name.

Block Nesting and Scope

Blocks can be nested for more complex layouts. Per default, blocks have access to variables from outer scopes:

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

Block Shortcuts

For blocks with little content, it's possible to use a shortcut syntax. The following constructs do the same thing:

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

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

Dynamic Inheritance

Twig supports dynamic inheritance by using a variable as the base template:

```
{% extends some_var %}
```

If the variable evaluates to a `Twig_Template` or a `Twig_TemplateWrapper` instance, Twig will use it as the parent template:

```
// {% extends layout %}

$layout = $twig->load('some_layout_template.twig');

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

You can also provide a list of templates that are checked for existence. The first template that exists will be used as a parent:

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

Conditional Inheritance

As the template name for the parent can be any valid Twig expression, it's possible to make the inheritance mechanism conditional:

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

In this example, the template will extend the "minimum.html" layout template if the `standalone` variable evaluates to `true`, and "base.html" otherwise.

How do blocks work?

A block provides a way to change how a certain part of a template is rendered but it does not interfere in any way with the logic around it.

Let's take the following example to illustrate how a block works and more importantly, how it does not work:

```
{# base.twig #}

{% for post in posts %}
    {% block post %}
        <h1>{{ post.title }}</h1>
        <p>{{ post.body }}</p>
```

```
    {% endblock %}
{% endfor %}
```

If you render this template, the result would be exactly the same with or without the `block` tag. The `block` inside the `for` loop is just a way to make it overridable by a child template:

```
{# child.twig #}

{% extends "base.twig" %}

{% block post %}
    <article>
        <header>{{ post.title }}</header>
        <section>{{ post.text }}</section>
    </article>
{% endblock %}
```

Now, when rendering the child template, the loop is going to use the block defined in the child template instead of the one defined in the base one; the executed template is then equivalent to the following one:

```
{% for post in posts %}
    <article>
        <header>{{ post.title }}</header>
        <section>{{ post.text }}</section>
    </article>
{% endfor %}
```

Let's take another example: a block included within an `if` statement:

```
{% if posts is empty %}
    {% block head %}
        {{ parent() }}

        <meta name="robots" content="noindex, follow">
    {% endblock head %}
{% endif %}
```

Contrary to what you might think, this template does not define a block conditionally; it just makes overridable by a child template the output of what will be rendered when the condition is `true`.

If you want the output to be displayed conditionally, use the following instead:

```
{% block head %}
    {{ parent() }}

    {% if posts is empty %}
        <meta name="robots" content="noindex, follow">
    {% endif %}
{% endblock head %}
```

See also:

block, *block*, *parent*, *use*

filter

Filter sections allow you to apply regular Twig filters on a block of template data. Just wrap the code in the special `filter` section:

```
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

You can also chain filters:

```
{% filter lower|escape %}
    <strong>SOME TEXT</strong>
{% endfilter %}

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

flush

The `flush` tag tells Twig to flush the output buffer:

```
{% flush %}
```

Note: Internally, Twig uses the PHP flush function.

for

Loop over each item in a sequence. For example, to display a list of users provided in a variable called `users`:

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

Note: A sequence can be either an array or an object implementing the `Traversable` interface.

If you do need to iterate over a sequence of numbers, you can use the `..` operator:

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

The above snippet of code would print all numbers from 0 to 10.

It can be also useful with letters:

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

The `..` operator can take any expression at both sides:

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

The *loop* variable

Inside of a `for` loop block you can access some special variables:

Variable	Description
`loop.index`	The current iteration of the loop. (1 indexed)
`loop.index0`	The current iteration of the loop. (0 indexed)
`loop.revindex`	The number of iterations from the end of the loop (1 indexed)
`loop.revindex0`	The number of iterations from the end of the loop (0 indexed)
`loop.first`	True if first iteration
`loop.last`	True if last iteration
`loop.length`	The number of items in the sequence
`loop.parent`	The parent context

```
{% for user in users %}
    {{ loop.index }} - {{ user.username }}
{% endfor %}
```

Note: The `loop.length`, `loop.revindex`, `loop.revindex0`, and `loop.last` variables are only available for PHP arrays, or objects that implement the `Countable` interface. They are also not available when looping with a condition.

Adding a condition

Unlike in PHP, it's not possible to `break` or `continue` in a loop. You can however filter the sequence during iteration which allows you to skip items. The following example skips all the users which are not active:

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

The advantage is that the special loop variable will count correctly thus not counting the users not iterated over. Keep in mind that properties like `loop.last` will not be defined when using loop conditions.

Note: Using the `loop` variable within the condition is not recommended as it will probably not be doing what you expect it to. For instance, adding a condition like `loop.index > 4` won't work as the index is only incremented when the condition is true (so the condition will never match).

The *else* Clause

If no iteration took place because the sequence was empty, you can render a replacement block by using `else`:

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

Iterating over Keys

By default, a loop iterates over the values of the sequence. You can iterate on keys by using the `keys` filter:

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

Iterating over Keys and Values

You can also access both keys and values:

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

Iterating over a Subset

You might want to iterate over a subset of values. This can be achieved using the *slice* filter:

```
<h1>Top Ten Members</h1>
<ul>
    {% for user in users|slice(0, 10) %}
        <li>{{ user.username|e }}</li>
    {% endfor %}
</ul>
```

`from`

The `from` tag imports *macro* names into the current namespace. The tag is documented in detail in the documentation for the *import* tag.

See also:

macro, *import*

if

The `if` statement in Twig is comparable with the if statements of PHP.

In the simplest form you can use it to test if an expression evaluates to `true`:

```
{% if online == false %}
    <p>Our website is in maintenance mode. Please, come back later.</p>
{% endif %}
```

You can also test if an array is not empty:

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

Note: If you want to test if the variable is defined, use `if users is defined` instead.

You can also use `not` to check for values that evaluate to `false`:

```
{% if not user.subscribed %}
    <p>You are not subscribed to our mailing list.</p>
{% endif %}
```

For multiple conditions, `and` and `or` can be used:

```
{% if temperature > 18 and temperature < 27 %}
    <p>It's a nice day for a walk in the park.</p>
{% endif %}
```

For multiple branches `elseif` and `else` can be used like in PHP. You can use more complex `expressions` there too:

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

Note: The rules to determine if an expression is `true` or `false` are the same as in PHP; here are the edge cases rules:

Value	Boolean evaluation
empty string	false
numeric zero	false
whitespace-only string	true
empty array	false
null	false
non-empty array	true
object	true

import

Twig supports putting often used code into *macros*. These macros can go into different templates and get imported from there.

There are two ways to import templates. You can import the complete template into a variable or request specific macros from it.

Imagine we have a helper module that renders forms (called `forms.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, cols) %}
    <textarea name="{{ name }}" rows="{{ rows|default(10) }}" cols="{{␣
↪cols|default(40) }}">{{ value|e }}</textarea>
{% endmacro %}
```

The easiest and most flexible is importing the whole module into a variable. That way you can access the attributes:

```
{% import 'forms.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('comment') }}</p>
```

Alternatively you can import names from the template into the current namespace:

```
{% from 'forms.html' import input as input_field, textarea %}

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

Tip: To import macros from the current file, use the special `_self` variable for the source.

See also:

macro, *from*

include

The `include` statement includes a template and returns the rendered content of that file into the current namespace:

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

Included templates have access to the variables of the active context.

If you are using the filesystem loader, the templates are looked for in the paths defined by it.

You can add additional variables by passing them after the `with` keyword:

```
{# template.html will have access to the variables from the current context and the␣
↪additional ones provided #}
{% include 'template.html' with {'foo': 'bar'} %}

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

You can disable access to the context by appending the `only` keyword:

```
{# only the foo variable will be accessible #}
{% include 'template.html' with {'foo': 'bar'} only %}
```

```
{# no variables will be accessible #}
{% include 'template.html' only %}
```

Tip: When including a template created by an end user, you should consider sandboxing it. More information in the *Twig for Developers* chapter and in the *sandbox* tag documentation.

The template name can be any valid Twig expression:

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

And if the expression evaluates to a `Twig_Template` or a `Twig_TemplateWrapper` instance, Twig will use it directly:

```
// {% include template %}

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

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

You can mark an include with `ignore missing` in which case Twig will ignore the statement if the template to be included does not exist. It has to be placed just after the template name. Here some valid examples:

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

You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be included:

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

If `ignore missing` is given, it will fall back to rendering nothing if none of the templates exist, otherwise it will throw an exception.

macro

Macros are comparable with functions in regular programming languages. They are useful to put often used HTML idioms into reusable elements to not repeat yourself.

Here is a small example of a macro that renders a form element:

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

Macros differ from native PHP functions in a few ways:

- Default argument values are defined by using the `default` filter in the macro body;
- Arguments of a macro are always optional.
- If extra positional arguments are passed to a macro, they end up in the special `varargs` variable as a list of values.

But as with PHP functions, macros don't have access to the current template variables.

Tip: You can pass the whole context as an argument by using the special `_context` variable.

Import

Macros can be defined in any template, and need to be "imported" before being used (see the documentation for the *import* tag for more information):

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

The above `import` call imports the "forms.html" file (which can contain only macros, or a template and some macros), and import the functions as items of the `forms` variable.

The macro can then be called at will:

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

If macros are defined and used in the same template, you can use the special `_self` variable to import them:

```
{% import _self as forms %}

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

When you want to use a macro in another macro from the same file, you need to import it locally:

```
{% 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) %}
    {% import _self as forms %}

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

Named Macro End-Tags

Twig allows you to put the name of the macro after the end tag for better readability:

```
{% macro input() %}
    ...
{% endmacro input %}
```

Of course, the name after the `endmacro` word must match the macro name.

See also:

from, *import*

sandbox

The `sandbox` tag can be used to enable the sandboxing mode for an included template, when sandboxing is not enabled globally for the Twig environment:

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

> **Warning:** The `sandbox` tag is only available when the sandbox extension is enabled (see the *Twig for Developers* chapter).

Note: The `sandbox` tag can only be used to sandbox an include tag and it cannot be used to sandbox a section of a template. The following example won't work:

```
{% sandbox %}
    {% for i in 1..2 %}
        {{ i }}
    {% endfor %}
{% endsandbox %}
```

set

Inside code blocks you can also assign values to variables. Assignments use the `set` tag and can have multiple targets.

Here is how you can assign the `bar` value to the `foo` variable:

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

After the `set` call, the `foo` variable is available in the template like any other ones:

```
{# displays bar #}
{{ foo }}
```

The assigned value can be any valid *Twig expressions*:

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

Several variables can be assigned in one block:

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

{# is equivalent to #}

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

The `set` tag can also be used to 'capture' chunks of text:

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

Caution: If you enable automatic output escaping, Twig will only consider the content to be safe when capturing chunks of text.

Note: Note that loops are scoped in Twig; therefore a variable declared inside a `for` loop is not accessible outside the loop itself:

```
{% for item in list %}
    {% set foo = item %}
{% endfor %}
```

```
{# foo is NOT available #}
```

If you want to access the variable, just declare it before the loop:

```
{% set foo = "" %}
{% for item in list %}
    {% set foo = item %}
{% endfor %}

{# foo is available #}
```

spaceless

Use the `spaceless` tag to remove whitespace *between HTML tags*, not whitespace within HTML tags or whitespace in plain text:

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

{# output will be <div><strong>foo</strong></div> #}
```

This tag is not meant to "optimize" the size of the generated HTML content but merely to avoid extra whitespace between HTML tags to avoid browser rendering quirks under some circumstances.

Tip: If you want to optimize the size of the generated HTML content, gzip compress the output instead.

Tip: If you want to create a tag that actually removes all extra whitespace in an HTML string, be warned that this is not as easy as it seems to be (think of `textarea` or `pre` tags for instance). Using a third-party library like Tidy is probably a better idea.

Tip: For more information on whitespace control, read the *dedicated section* of the documentation and learn how you can also use the whitespace control modifier on your tags.

use

Note: Horizontal reuse is an advanced Twig feature that is hardly ever needed in regular templates. It is mainly used by projects that need to make template blocks reusable without using inheritance.

Template inheritance is one of the most powerful features of Twig but it is limited to single inheritance; a template can only extend one other template. This limitation makes template inheritance simple to understand and easy to debug:

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

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

Horizontal reuse is a way to achieve the same goal as multiple inheritance, but without the associated complexity:

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

{% use "blocks.html" %}

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

The `use` statement tells Twig to import the blocks defined in `blocks.html` into the current template (it's like macros, but for blocks):

```
{# blocks.html #}

{% block sidebar %}{% endblock %}
```

In this example, the `use` statement imports the `sidebar` block into the main template. The code is mostly equivalent to the following one (the imported blocks are not outputted automatically):

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

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

Note: The `use` tag only imports a template if it does not extend another template, if it does not define macros, and if the body is empty. But it can *use* other templates.

Note: Because `use` statements are resolved independently of the context passed to the template, the template reference cannot be an expression.

The main template can also override any imported block. If the template already defines the `sidebar` block, then the one defined in `blocks.html` is ignored. To avoid name conflicts, you can rename imported blocks:

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

{% use "blocks.html" with sidebar as base_sidebar, title as base_title %}

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

The `parent()` function automatically determines the correct inheritance tree, so it can be used when overriding a block defined in an imported template:

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

{% use "blocks.html" %}
```

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

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

In this example, `parent()` will correctly call the `sidebar` block from the `blocks.html` template.

Tip: Renaming allows you to simulate inheritance by calling the "parent" block:

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

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

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

Note: You can use as many `use` statements as you want in any given template. If two imported templates define the same block, the latest one wins.

verbatim

The `verbatim` tag marks sections as being raw text that should not be parsed. For example to put Twig syntax as example into a template you can use this snippet:

```
{% verbatim %}
    <ul>
    {% for item in seq %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>
{% endverbatim %}
```

with

Use the `with` tag to create a new inner scope. Variables set within this scope are not visible outside of the scope:

```
{% with %}
    {% set foo = 42 %}
    {{ foo }}           foo is 42 here
{% endwith %}
foo is not visible here any longer
```

Instead of defining variables at the beginning of the scope, you can pass a hash of variables you want to define in the `with` tag; the previous example is equivalent to the following one:

```
{% with { foo: 42 } %}
    {{ foo }}           foo is 42 here
{% endwith %}
foo is not visible here any longer

{# it works with any expression that resolves to a hash #}
{% set vars = { foo: 42 } %}
{% with vars %}
    ...
{% endwith %}
```

By default, the inner scope has access to the outer scope context; you can disable this behavior by appending the `only` keyword:

```
{% set bar = 'bar' %}
{% with { foo: 42 } only %}
    {# only foo is defined #}
    {# bar is not defined #}
{% endwith %}
```

CHAPTER 10

Filters

abs

The `abs` filter returns the absolute value.

```
{# number = -5 #}

{{ number|abs }}

{# outputs 5 #}
```

Note: Internally, Twig uses the PHP abs function.

batch

The `batch` filter "batches" items by returning a list of lists with the given number of items. A second parameter can be provided and used to fill in missing items:

```
{% set items = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] %}

<table>
{% for row in items|batch(3, 'No item') %}
    <tr>
        {% for column in row %}
            <td>{{ column }}</td>
        {% endfor %}
    </tr>
{% endfor %}
</table>
```

The above example will be rendered as:

```
<table>
    <tr>
        <td>a</td>
        <td>b</td>
        <td>c</td>
    </tr>
    <tr>
        <td>d</td>
        <td>e</td>
        <td>f</td>
    </tr>
    <tr>
        <td>g</td>
        <td>No item</td>
        <td>No item</td>
    </tr>
</table>
```

Arguments

- `size`: The size of the batch; fractional numbers will be rounded up
- `fill`: Used to fill in missing items

capitalize

The `capitalize` filter capitalizes a value. The first character will be uppercase, all others lowercase:

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

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

convert_encoding

The `convert_encoding` filter converts a string from one encoding to another. The first argument is the expected output charset and the second one is the input charset:

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

Note: This filter relies on the iconv or mbstring extension, so one of them must be installed. In case both are installed, mbstring is used by default.

Arguments

- `to`: The output charset
- `from`: The input charset

date

The `date` filter formats a date to a given format:

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

The format specifier is the same as supported by date, except when the filtered data is of type DateInterval, when the format must conform to DateInterval::format instead.

The `date` filter accepts strings (it must be in a format supported by the strtotime function), DateTime instances, or DateInterval instances. For instance, to display the current date, filter the word "now":

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

To escape words and characters in the date format use `\\` in front of each character:

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

If the value passed to the `date` filter is `null`, it will return the current date by default. If an empty string is desired instead of the current date, use a ternary operator:

```
{{ post.published_at is empty ? "" : post.published_at|date("m/d/Y") }}
```

If no format is provided, Twig will use the default one: `F j, Y H:i`. This default can be easily changed by calling the `setDateFormat()` method on the `core` extension instance. The first argument is the default format for dates and the second one is the default format for date intervals:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('Twig_Extension_Core')->setDateFormat('d/m/Y', '%d days');
```

Timezone

By default, the date is displayed by applying the default timezone (the one specified in php.ini or declared in Twig – see below), but you can override it by explicitly specifying a timezone:

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

If the date is already a DateTime object, and if you want to keep its current timezone, pass `false` as the timezone value:

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

The default timezone can also be set globally by calling `setTimezone()`:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('Twig_Extension_Core')->setTimezone('Europe/Paris');
```

Arguments

- `format`: The date format
- `timezone`: The date timezone

date_modify

The `date_modify` filter modifies a date with a given modifier string:

```
{{ post.published_at|date_modify("+1 day")|date("m/d/Y") }}
```

The `date_modify` filter accepts strings (it must be in a format supported by the strtotime function) or DateTime instances. You can easily combine it with the *date* filter for formatting.

Arguments

- `modifier`: The modifier

default

The `default` filter returns the passed default value if the value is undefined or empty, otherwise the value of the variable:

```
{{ var|default('var is not defined') }}

{{ var.foo|default('foo item on var is not defined') }}

{{ var['foo']|default('foo item on var is not defined') }}

{{ ''|default('passed var is empty')  }}
```

When using the `default` filter on an expression that uses variables in some method calls, be sure to use the `default` filter whenever a variable can be undefined:

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

Note: Read the documentation for the *defined* and *empty* tests to learn more about their semantics.

Arguments

- `default`: The default value

escape

The `escape` filter escapes a string for safe insertion into the final output. It supports different escaping strategies depending on the template context.

By default, it uses the HTML escaping strategy:

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

For convenience, the `e` filter is defined as an alias:

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

The `escape` filter can also be used in other contexts than HTML thanks to an optional argument which defines the escaping strategy to use:

```
{{ user.username|e }}
{# is equivalent to #}
{{ user.username|e('html') }}
```

And here is how to escape variables included in JavaScript code:

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

The `escape` filter supports the following escaping strategies:

- `html`: escapes a string for the **HTML body** context.
- `js`: escapes a string for the **JavaScript context**.
- `css`: escapes a string for the **CSS context**. CSS escaping can be applied to any string being inserted into CSS and escapes everything except alphanumerics.
- `url`: escapes a string for the **URI or parameter contexts**. This should not be used to escape an entire URI; only a subcomponent being inserted.
- `html_attr`: escapes a string for the **HTML attribute** context.

Note: Internally, `escape` uses the PHP native htmlspecialchars function for the HTML escaping strategy.

Caution: When using automatic escaping, Twig tries to not double-escape a variable when the automatic escaping strategy is the same as the one applied by the escape filter; but that does not work when using a variable as the escaping strategy:

```
{% set strategy = 'html' %}

{% autoescape 'html' %}
    {{ var|escape('html') }}    {# won't be double-escaped #}
    {{ var|escape(strategy) }} {# will be double-escaped #}
{% endautoescape %}
```

When using a variable as the escaping strategy, you should disable automatic escaping:

```
{% set strategy = 'html' %}

{% autoescape 'html' %}
    {{ var|escape(strategy)|raw }} {# won't be double-escaped #}
{% endautoescape %}
```

Custom Escapers

You can define custom escapers by calling the `setEscaper()` method on the `core` extension instance. The first argument is the escaper name (to be used in the `escape` call) and the second one must be a valid PHP callable:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('Twig_Extension_Core')->setEscaper('csv', 'csv_escaper');
```

When called by Twig, the callable receives the Twig environment instance, the string to escape, and the charset.

Note: Built-in escapers cannot be overridden mainly they should be considered as the final implementation and also for better performance.

Arguments

- `strategy`: The escaping strategy
- `charset`: The string charset

first

The `first` filter returns the first "element" of a sequence, a mapping, or a string:

```
{{ [1, 2, 3, 4]|first }}
{# outputs 1 #}

{{ { a: 1, b: 2, c: 3, d: 4 }|first }}
{# outputs 1 #}

{{ '1234'|first }}
{# outputs 1 #}
```

Note: It also works with objects implementing the Traversable interface.

format

The `format` filter formats a given string by replacing the placeholders (placeholders follows the sprintf notation):

```
{{ "I like %s and %s."|format(foo, "bar") }}

{# outputs I like foo and bar
   if the foo parameter equals to the foo string. #}
```

See also:

replace

join

The `join` filter returns a string which is the concatenation of the items of a sequence:

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

The separator between elements is an empty string per default, but you can define it with the optional first parameter:

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

Arguments

- `glue`: The separator

json_encode

The `json_encode` filter returns the JSON representation of a value:

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

Note: Internally, Twig uses the PHP json_encode function.

Arguments

- `options`: A bitmask of json_encode options (`{{ data|json_encode(constant('JSON_PRETTY_PRINT')) }}`)

keys

The `keys` filter returns the keys of an array. It is useful when you want to iterate over the keys of an array:

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

last

The `last` filter returns the last "element" of a sequence, a mapping, or a string:

```
{{ [1, 2, 3, 4]|last }}
{# outputs 4 #}

{{ { a: 1, b: 2, c: 3, d: 4 }|last }}
{# outputs 4 #}

{{ '1234'|last }}
{# outputs 4 #}
```

Note: It also works with objects implementing the Traversable interface.

length

New in version 2.3: Support for the `__toString()` magic method has been added in Twig 2.3.

The `length` filter returns the number of items of a sequence or mapping, or the length of a string.

For objects that implement the `Countable` interface, `length` will use the return value of the `count()` method.

For objects that implement the `__toString()` magic method (and not `Countable`), it will return the length of the string provided by that method.

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

lower

The `lower` filter converts a value to lowercase:

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

{# outputs 'welcome' #}
```

merge

The `merge` filter merges an array with another array:

```
{% set values = [1, 2] %}

{% set values = values|merge(['apple', 'orange']) %}

{# values now contains [1, 2, 'apple', 'orange'] #}
```

New values are added at the end of the existing ones.

The `merge` filter also works on hashes:

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

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

{# items now contains { 'apple': 'fruit', 'orange': 'fruit', 'peugeot': 'car',
↪'renault': 'car' } #}
```

For hashes, the merging process occurs on the keys: if the key does not already exist, it is added but if the key already exists, its value is overridden.

Tip: If you want to ensure that some values are defined in an array (by given default values), reverse the two elements in the call:

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

{% set items = { 'apple': 'unknown' }|merge(items) %}

{# items now contains { 'apple': 'fruit', 'orange': 'fruit' } #}
```

Note: Internally, Twig uses the PHP array_merge function. It supports Traversable objects by transforming those to arrays.

nl2br

The `nl2br` filter inserts HTML line breaks before all newlines in a string:

```
{{ "I like Twig.\nYou will like it too."|nl2br }}
{# outputs

    I like Twig.<br />
    You will like it too.

#}
```

Note: The `nl2br` filter pre-escapes the input before applying the transformation.

number_format

The `number_format` filter formats numbers. It is a wrapper around PHP's number_format function:

```
{{ 200.35|number_format }}
```

You can control the number of decimal places, decimal point, and thousands separator using the additional arguments:

```
{{ 9800.333|number_format(2, '.', ',') }}
```

To format negative numbers, wrap the number with parentheses (needed because of Twig's *precedence of operators*:

```
{{ -9800.333|number_format(2, '.', ',') }} {# outputs : -9 #}
{{ (-9800.333)|number_format(2, '.', ',') }} {# outputs : -9,800.33 #}
```

If no formatting options are provided then Twig will use the default formatting options of:

- 0 decimal places.
- `.` as the decimal point.
- `,` as the thousands separator.

These defaults can be easily changed through the core extension:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('Twig_Extension_Core')->setNumberFormat(3, '.', ',');
```

The defaults set for `number_format` can be over-ridden upon each call using the additional parameters.

Arguments

- `decimal`: The number of decimal points to display
- `decimal_point`: The character(s) to use for the decimal point
- `thousand_sep`: The character(s) to use for the thousands separator

raw

The `raw` filter marks the value as being "safe", which means that in an environment with automatic escaping enabled this variable will not be escaped if `raw` is the last filter applied to it:

```
{% autoescape %}
    {{ var|raw }} {# var won't be escaped #}
{% endautoescape %}
```

Note: Be careful when using the `raw` filter inside expressions:

```
{% autoescape %}
    {% set hello = '<strong>Hello</strong>' %}
    {% set hola = '<strong>Hola</strong>' %}

    {{ false ? '<strong>Hola</strong>' : hello|raw }}
    does not render the same as
    {{ false ? hola : hello|raw }}
    but renders the same as
    {{ (false ? hola : hello)|raw }}
{% endautoescape %}
```

The first ternary statement is not escaped: `hello` is marked as being safe and Twig does not escape static values (see *escape*). In the second ternary statement, even if `hello` is marked as safe, `hola` remains unsafe and so is the whole expression. The third ternary statement is marked as safe and the result is not escaped.

---

replace

The `replace` filter formats a given string by replacing the placeholders (placeholders are free-form):

```
{{ "I like %this% and %that%."|replace({'%this%': foo, '%that%': "bar"}) }}

{# outputs I like foo and bar
   if the foo parameter equals to the foo string. #}
```

Arguments

- `from`: The placeholder values

See also:

format

reverse

The `reverse` filter reverses a sequence, a mapping, or a string:

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

{{ '1234'|reverse }}

{# outputs 4321 #}
```

Tip: For sequences and mappings, numeric keys are not preserved. To reverse them as well, pass `true` as an argument to the `reverse` filter:

```
{% for key, value in {1: "a", 2: "b", 3: "c"}|reverse %}
    {{ key }}: {{ value }}
{%- endfor %}

{# output: 0: c    1: b    2: a #}

{% for key, value in {1: "a", 2: "b", 3: "c"}|reverse(true) %}
    {{ key }}: {{ value }}
{%- endfor %}

{# output: 3: c    2: b    1: a #}
```

Note: It also works with objects implementing the Traversable interface.

Arguments

- `preserve_keys`: Preserve keys when reversing a mapping or a sequence.

round

The `round` filter rounds a number to a given precision:

```
{{ 42.55|round }}
{# outputs 43 #}
```

```
{{ 42.55|round(1, 'floor') }}
{# outputs 42.5 #}
```

The `round` filter takes two optional arguments; the first one specifies the precision (default is `0`) and the second the rounding method (default is `common`):

- `common` rounds either up or down (rounds the value up to precision decimal places away from zero, when it is half way there – making 1.5 into 2 and -1.5 into -2);
- `ceil` always rounds up;
- `floor` always rounds down.

Note: The `//` operator is equivalent to `|round(0, 'floor')`.

Arguments

- `precision`: The rounding precision
- `method`: The rounding method

slice

The `slice` filter extracts a slice of a sequence, a mapping, or a string:

```
{% for i in [1, 2, 3, 4, 5]|slice(1, 2) %}
    {# will iterate over 2 and 3 #}
{% endfor %}

{{ '12345'|slice(1, 2) }}

{# outputs 23 #}
```

You can use any valid expression for both the start and the length:

```
{% for i in [1, 2, 3, 4, 5]|slice(start, length) %}
    {# ... #}
{% endfor %}
```

As syntactic sugar, you can also use the `[]` notation:

```
{% for i in [1, 2, 3, 4, 5][start:length] %}
    {# ... #}
{% endfor %}

{{ '12345'[1:2] }} {# will display "23" #}

{# you can omit the first argument -- which is the same as 0 #}
{{ '12345'[:2] }} {# will display "12" #}

{# you can omit the last argument -- which will select everything till the end #}
{{ '12345'[2:] }} {# will display "345" #}
```

The `slice` filter works as the array_slice PHP function for arrays and mb_substr for strings with a fallback to substr.

If the start is non-negative, the sequence will start at that start in the variable. If start is negative, the sequence will start that far from the end of the variable.

If length is given and is positive, then the sequence will have up to that many elements in it. If the variable is shorter than the length, then only the available variable elements will be present. If length is given and is negative then the sequence will stop that many elements from the end of the variable. If it is omitted, then the sequence will have everything from offset up until the end of the variable.

Note: It also works with objects implementing the Traversable interface.

Arguments

- `start`: The start of the slice
- `length`: The size of the slice
- `preserve_keys`: Whether to preserve key or not (when the input is an array)

sort

The `sort` filter sorts an array:

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

Note: Internally, Twig uses the PHP asort function to maintain index association. It supports Traversable objects by transforming those to arrays.

split

The `split` filter splits a string by the given delimiter and returns a list of strings:

```
{% set foo = "one,two,three"|split(',') %}
{# foo contains ['one', 'two', 'three'] #}
```

You can also pass a `limit` argument:

- If `limit` is positive, the returned array will contain a maximum of limit elements with the last element containing the rest of string;
- If `limit` is negative, all components except the last -limit are returned;
- If `limit` is zero, then this is treated as 1.

```
{% set foo = "one,two,three,four,five"|split(',', 3) %}
{# foo contains ['one', 'two', 'three,four,five'] #}
```

If the `delimiter` is an empty string, then value will be split by equal chunks. Length is set by the `limit` argument (one character by default).

```
{% set foo = "123"|split('') %}
{# foo contains ['1', '2', '3'] #}

{% set bar = "aabbcc"|split('', 2) %}
{# bar contains ['aa', 'bb', 'cc'] #}
```

Note: Internally, Twig uses the PHP explode or str_split (if delimiter is empty) functions for string splitting.

Arguments

- `delimiter`: The delimiter
- `limit`: The limit argument

striptags

The `striptags` filter strips SGML/XML tags and replace adjacent whitespace by one space:

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

You can also provide tags which should not be stripped:

```
{{ some_html|striptags('<br><p>') }}
```

In this example, the `<br/>`, `<br>`, `<p>`, and `</p>` tags won't be removed from the string.

Note: Internally, Twig uses the PHP strip_tags function.

Arguments

- `allowable_tags`: Tags which should not be stripped

title

The `title` filter returns a titlecased version of the value. Words will start with uppercase letters, all remaining characters are lowercase:

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

{# outputs 'My First Car' #}
```

trim

The `trim` filter strips whitespace (or other characters) from the beginning and end of a string:

```
{{ '  I like Twig.  '|trim }}

{# outputs 'I like Twig.' #}

{{ '  I like Twig.'|trim('.') }}

{# outputs '  I like Twig' #}

{{ '  I like Twig.  '|trim(side='left') }}

{# outputs 'I like Twig.  ' #}

{{ '  I like Twig.  '|trim(' ', 'right') }}

{# outputs '  I like Twig.' #}
```

Note: Internally, Twig uses the PHP trim, ltrim, and rtrim functions.

Arguments

- `character_mask`: The characters to strip
- `side`: The default is to strip from the left and the right (*both*) sides, but *left* and *right* will strip from either the left side or right side only

upper

The `upper` filter converts a value to uppercase:

```
{{ 'welcome'|upper }}

{# outputs 'WELCOME' #}
```

url_encode

The `url_encode` filter percent encodes a given string as URL segment or an array as query string:

```
{{ "path-seg*ment"|url_encode }}
{# outputs "path-seg%2Ament" #}

{{ "string with spaces"|url_encode }}
{# outputs "string%20with%20spaces" #}

{{ {'param': 'value', 'foo': 'bar'}|url_encode }}
{# outputs "param=value&foo=bar" #}
```

Note: Internally, Twig uses the PHP `rawurlencode`.

CHAPTER 11

Functions

attribute

The `attribute` function can be used to access a "dynamic" attribute of a variable:

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

In addition, the `defined` test can check for the existence of a dynamic attribute:

```
{{ attribute(object, method) is defined ? 'Method exists' : 'Method does not exist' }}
```

Note: The resolution algorithm is the same as the one used for the `.` notation, except that the item can be any valid expression.

block

When a template uses inheritance and if you want to print a block multiple times, use the `block` function:

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

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

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

The `block` function can also be used to display one block of another template:

```
{{ block("title", "common_blocks.twig") }}
```

Use the `defined` test to check if a block exists in the context of the current template:

```
{% if block("footer") is defined %}
    ...
{% endif %}

{% if block("footer", "common_blocks.twig") is defined %}
    ...
{% endif %}
```

See also:

extends, *parent*

constant

`constant` returns the constant value for a given string:

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

You can read constants from object instances as well:

```
{{ constant('RSS', date) }}
```

Use the `defined` test to check if a constant is defined:

```
{% if constant('SOME_CONST') is defined %}
    ...
{% endif %}
```

cycle

The `cycle` function cycles on an array of values:

```
{% set start_year = date() | date('Y') %}
{% set end_year = start_year + 5 %}

{% for year in start_year..end_year %}
    {{ cycle(['odd', 'even'], loop.index0) }}
{% endfor %}
```

The array can contain any number of values:

```
{% set fruits = ['apple', 'orange', 'citrus'] %}

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

Arguments

- `position`: The cycle position

date

Converts an argument to a date to allow date comparison:

```
{% if date(user.created_at) < date('-2days') %}
    {# do something #}
{% endif %}
```

The argument must be in one of PHP's supported date and time formats.

You can pass a timezone as the second argument:

```
{% if date(user.created_at) < date('-2days', 'Europe/Paris') %}
    {# do something #}
{% endif %}
```

If no argument is passed, the function returns the current date:

```
{% if date(user.created_at) < date() %}
    {# always! #}
{% endif %}
```

Note: You can set the default timezone globally by calling `setTimezone()` on the `core` extension instance:

```
$twig = new Twig_Environment($loader);
$twig->getExtension('Twig_Extension_Core')->setTimezone('Europe/Paris');
```

Arguments

- `date`: The date
- `timezone`: The timezone

dump

The `dump` function dumps information about a template variable. This is mostly useful to debug a template that does not behave as expected by introspecting its variables:

```
{{ dump(user) }}
```

Note: The `dump` function is not available by default. You must add the `Twig_Extension_Debug` extension explicitly when creating your Twig environment:

```
$twig = new Twig_Environment($loader, array(
    'debug' => true,
    // ...
));
$twig->addExtension(new Twig_Extension_Debug());
```

Even when enabled, the `dump` function won't display anything if the `debug` option on the environment is not enabled (to avoid leaking debug information on a production server).

In an HTML context, wrap the output with a `pre` tag to make it easier to read:

```
<pre>
    {{ dump(user) }}
</pre>
```

Tip: Using a `pre` tag is not needed when XDebug is enabled and `html_errors` is `on`; as a bonus, the output is also nicer with XDebug enabled.

You can debug several variables by passing them as additional arguments:

```
{{ dump(user, categories) }}
```

If you don't pass any value, all variables from the current context are dumped:

```
{{ dump() }}
```

Note: Internally, Twig uses the PHP var_dump function.

Arguments

- `context`: The context to dump

include

The `include` function returns the rendered content of a template:

```
{{ include('template.html') }}
{{ include(some_var) }}
```

Included templates have access to the variables of the active context.

If you are using the filesystem loader, the templates are looked for in the paths defined by it.

The context is passed by default to the template but you can also pass additional variables:

```
{# template.html will have access to the variables from the current context and the␣
↪additional ones provided #}
{{ include('template.html', {foo: 'bar'}) }}
```

You can disable access to the context by setting `with_context` to `false`:

```
{# only the foo variable will be accessible #}
{{ include('template.html', {foo: 'bar'}, with_context = false) }}
```

```
{# no variables will be accessible #}
{{ include('template.html', with_context = false) }}
```

And if the expression evaluates to a `Twig_Template` or a `Twig_TemplateWrapper` instance, Twig will use it directly:

```
// {{ include(template) }}

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

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

When you set the `ignore_missing` flag, Twig will return an empty string if the template does not exist:

```
{{ include('sidebar.html', ignore_missing = true) }}
```

You can also provide a list of templates that are checked for existence before inclusion. The first template that exists will be rendered:

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

If `ignore_missing` is set, it will fall back to rendering nothing if none of the templates exist, otherwise it will throw an exception.

When including a template created by an end user, you should consider sandboxing it:

```
{{ include('page.html', sandboxed = true) }}
```

Arguments

- `template`: The template to render
- `variables`: The variables to pass to the template
- `with_context`: Whether to pass the current context variables or not
- `ignore_missing`: Whether to ignore missing templates or not
- `sandboxed`: Whether to sandbox the template or not

max

`max` returns the biggest value of a sequence or a set of values:

```
{{ max(1, 3, 2) }}
{{ max([1, 3, 2]) }}
```

When called with a mapping, max ignores keys and only compares values:

```
{{ max({2: "e", 1: "a", 3: "b", 5: "d", 4: "c"}) }}
{# returns "e" #}
```

min

`min` returns the lowest value of a sequence or a set of values:

```
{{ min(1, 3, 2) }}
{{ min([1, 3, 2]) }}
```

When called with a mapping, min ignores keys and only compares values:

```
{{ min({2: "e", 3: "a", 1: "b", 5: "d", 4: "c"}) }}
{# returns "a" #}
```

parent

When a template uses inheritance, it's possible to render the contents of the parent block when overriding a block by using the `parent` function:

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

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

The `parent()` call will return the content of the `sidebar` block as defined in the `base.html` template.

See also:

extends, *block*, *block*

random

The `random` function returns a random value depending on the supplied parameter type:

- a random item from a sequence;
- a random character from a string;
- a random integer between 0 and the integer parameter (inclusive).

```
{{ random(['apple', 'orange', 'citrus']) }} {# example output: orange #}
{{ random('ABC') }}                         {# example output: C #}
{{ random() }}                              {# example output: 15386094 (works as the␣
↪native PHP mt_rand function) #}
{{ random(5) }}                             {# example output: 3 #}
```

Arguments

- `values`: The values

range

Returns a list containing an arithmetic progression of integers:

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

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

When step is given (as the third parameter), it specifies the increment (or decrement for negative values):

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

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

Note: Note that if the start is greater than the end, `range` assumes a step of `-1`:

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

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

The Twig built-in `..` operator is just syntactic sugar for the `range` function (with a step of `1`, or `-1` if the start is greater than the end):

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

Tip: The `range` function works as the native PHP range function.

Arguments

- `low`: The first value of the sequence.
- `high`: The highest possible value of the sequence.
- `step`: The increment between elements of the sequence.

source

The `source` function returns the content of a template without rendering it:

```
{{ source('template.html') }}
{{ source(some_var) }}
```

When you set the `ignore_missing` flag, Twig will return an empty string if the template does not exist:

```
{{ source('template.html', ignore_missing = true) }}
```

The function uses the same template loaders as the ones used to include templates. So, if you are using the filesystem loader, the templates are looked for in the paths defined by it.

Arguments

- `name`: The name of the template to read
- `ignore_missing`: Whether to ignore missing templates or not

`template_from_string`

The `template_from_string` function loads a template from a string:

```
{{ include(template_from_string("Hello {{ name }}")) }}
{{ include(template_from_string(page.template)) }}
```

Note: The `template_from_string` function is not available by default. You must add the `Twig_Extension_StringLoader` extension explicitly when creating your Twig environment:

```
$twig = new Twig_Environment(...);
$twig->addExtension(new Twig_Extension_StringLoader());
```

Note: Even if you will probably always use the `template_from_string` function with the `include` function, you can use it with any tag or function that takes a template as an argument (like the `embed` or `extends` tags).

Arguments

- `template`: The template

CHAPTER 12

Tests

constant

`constant` checks if a variable has the exact same value as a constant. You can use either global constants or class constants:

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

You can test constants from object instances as well:

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

defined

`defined` checks if a variable is defined in the current context. This is very useful if you use the `strict_variables` option:

```
{# defined works with variable names #}
{% if foo is defined %}
    ...
{% endif %}

{# and attributes on variables names #}
{% if foo.bar is defined %}
    ...
{% endif %}

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

```
    ...
{% endif %}
```

When using the `defined` test on an expression that uses variables in some method calls, be sure that they are all defined first:

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

divisible by

`divisible by` checks if a variable is divisible by a number:

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

empty

New in version 2.3: Support for the `__toString()` magic method has been added in Twig 2.3.

`empty` checks if a variable is an empty string, an empty array, an empty hash, exactly `false`, or exactly `null`.

For objects that implement the `Countable` interface, `empty` will check the return value of the `count()` method.

For objects that implement the `__toString()` magic method (and not `Countable`), it will check if an empty string is returned.

```
{% if foo is empty %}
    ...
{% endif %}
```

even

`even` returns `true` if the given number is even:

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

See also:

odd

iterable

`iterable` checks if a variable is an array or a traversable object:

```
{# evaluates to true if the foo variable is iterable #}
{% if users is iterable %}
    {% for user in users %}
        Hello {{ user }}!
    {% endfor %}
{% else %}
    {# users is probably a string #}
    Hello {{ users }}!
{% endif %}
```

null

`null` returns `true` if the variable is `null`:

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

Note: `none` is an alias for `null`.

odd

`odd` returns `true` if the given number is odd:

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

See also:

even

same as

`same as` checks if a variable is the same as another variable. This is the equivalent to === in PHP:

```
{% if foo.attribute is same as(false) %}
    the foo attribute really is the 'false' PHP value
{% endif %}
```