a Symfony Project

# Twig

The flexible, fast, and secure template engine for PHP



You are reading the documentation for Twig 2.x. Switch to the documentation for Twig 1.x.

## 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)

Table of Contents



Questions & Feedback

**Found a typo or an error?**
Want to improve this document? Edit it.

**Need support or have a technical question?**
Ask support on Stack Overflow.

License

- *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') }}
```

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.

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") }}
```

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:

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

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

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

```twig
{{ 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:

```twig
<!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:

```twig
{% 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:

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

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

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(
{% 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.

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

```twig
{# 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:

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

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.

Twig also supports bitwise operators (`b-and`, `b-xor`, and `b-or`).

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 %}
```

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' }}
```

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.

« Installation | Twig for Developers »