



| | | | | | |
|---|---|-----------------------------|-------------------------------|-----------------------------|-----------------------|
|  SymfonyCon |  Live | PARIS March 28-29 | SÃO PAULO May 16-17 | LONDON Sep. 13 | NEW YORK Q4 |
| AMSTERDAM Nov. 21-23 | LILLE March 1 | TUNIS April 27 | WARSZAWA June 13-14 | BERLIN Sep. 24-27 | |

Table of Contents

- [Templates](#)
 - [Twig Template Caching](#)
- [Template Inheritance and Layouts](#)
- [Template Naming and Locations](#)
 - [Referencing Templates in a Bundle](#)
 - [Template Suffix](#)
- [Tags and Helpers](#)
 - [Including other Templates](#)
 - [Linking to Pages](#)
 - [Linking to Assets](#)

[Home](#) / [Documentation](#)

You are browsing the **Symfony 4 documentation**, which changes significantly from Symfony 3.x. If your app doesn't use Symfony 4 yet, browse the [Symfony 3.4 documentation](#).

Creating and Using Templates

4.2 version ▼

edit this page

As explained in [the previous article](#), controllers are responsible for handling each request that comes into a Symfony application and they usually end up rendering a template to generate the response contents.

In reality, the controller delegates most of the heavy work to other places so that code can be tested and reused. When a controller needs to generate HTML, CSS or any other content, it hands the work off to the templating engine.

In this article, you'll learn how to write powerful templates that can be used to return content to the user, populate email bodies, and more. You'll learn shortcuts, clever ways to extend templates and how to reuse template code.

Templates ¶

A template is a text file that can generate any text-based format (HTML, XML, CSV, LaTeX ...). The most familiar type of template is a *PHP* template - a text file parsed by PHP that contains a mix of text and PHP code:

```
1 <!DOCTYPE html>
```

```

2  <html>
3      <head>
4          <title>Welcome to Symfony!</title>
5      </head>
6      <body>
7          <h1><?= $page_title ?></h1>
8
9          <ul id="navigation">
10             <?php foreach ($navigation as $item): ?>
11                 <li>
12                     <a href="<?= $item->getHref() ?>">
13                         <?= $item->getCaption() ?>
14                     </a>
15                 </li>
16             <?php endforeach ?>
17         </ul>
18     </body>
19 </html>

```

But Symfony packages an even more powerful templating language called [Twig](#). Twig allows you to write concise, readable templates that are more friendly to web designers and, in several ways, more powerful than PHP templates:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Welcome to Symfony!</title>
5      </head>
6      <body>
7          <h1>{{ page_title }}</h1>
8
9          <ul id="navigation">
10             {% for item in navigation %}
11                 <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
12             {% endfor %}
13         </ul>
14     </body>
15 </html>

```

Twig defines three types of special syntax:

`{{ ... }}`

"Says something": prints a variable or the result of an expression to the template.

`{% ... %}`

"Does something": a **tag** that controls the logic of the template; it is used to execute statements such as for-loops for example.

`{# ... #}`

"Comment something": it's the equivalent of the PHP `/* comment */` syntax. It's used to add single or multi-line comments. The content of the comments isn't included in the rendered pages.

Twig also contains **filters**, which modify content before being rendered. The following makes the `title` variable all uppercase before rendering it:

```
1  {{ title|upper }}
```

Twig comes with a long list of [tags](#), [filters](#) and [functions](#) that are available by default. You can even add your own *custom* filters, functions (and more) via a [Twig Extension](#). Run the following command to list them all:

```
$ php bin/console debug:twig
```

Twig code will look similar to PHP code, with subtle, nice differences. The following example uses a standard `for` tag and the `cycle()` function to print ten `div` tags, with alternating `odd`, `even` classes:

```
1  {% for i in 1..10 %}
2      <div class="{{ cycle(['even', 'odd'], i) }}">
3          <!-- some HTML here -->
4      </div>
5  {% endfor %}
```

Throughout this article, template examples will be shown in both Twig and PHP.

Why Twig?

Twig templates are meant to be simple and won't process PHP tags. This is by design: the Twig template system is meant to express presentation, not program logic. The more you use Twig, the more you'll appreciate and benefit from this distinction. And of course, you'll be loved by web designers everywhere.

Twig can also do things that PHP can't, such as whitespace control, sandboxing, automatic HTML escaping, manual contextual output escaping, and the inclusion of custom functions and filters that only affect templates. Twig contains a lot of features that make writing templates easier and more concise. Take the following example, which combines a loop with a logical `if` statement:

```
1  <ul>
2      {% for user in users if user.active %}
3          <li>{{ user.username }}</li>
4      {% else %}
5          <li>No users found</li>
6      {% endfor %}
7  </ul>
```

Twig Template Caching ¶

Twig is fast because each template is compiled to a native PHP class and cached. But don't worry: this happens automatically and doesn't require *you* to do anything. And while you're developing, Twig is smart enough to re-compile your templates after you make any changes. That means Twig is fast in production, but convenient to use while developing.

Template Inheritance and Layouts ¶

More often than not, templates in a project share common elements, like the header, footer, sidebar or more. In Symfony, this problem is thought about differently: a template can be decorated by another one. This works exactly the same as PHP classes: template inheritance allows you to build a base "layout" template that contains all the common elements of your site defined as **blocks** (think "PHP class with base methods"). A child template can extend the base layout and override any of its blocks (think "PHP subclass that overrides certain methods of its parent class").

First, build a base layout file:

```
1  {% templates/base.html.twig %}
2  <!DOCTYPE html>
3  <html>
4      <head>
5          <meta charset="UTF-8">
6          <title>{% block title %}Test Application{% endblock %}</title>
7      </head>
8      <body>
9          <div id="sidebar">
10             {% block sidebar %}
11                 <ul>
12                     <li><a href="/">Home</a></li>
13                     <li><a href="/blog">Blog</a></li>
14                 </ul>
15             {% endblock %}
16          </div>
17
18          <div id="content">
19              {% block body %}{% endblock %}
20          </div>
21      </body>
22  </html>
```



Note

Though the discussion about template inheritance will be in terms of Twig, the philosophy is the same between Twig and PHP templates.

This template defines the base HTML skeleton document of a two-column page. In this example, three `{% block %}` areas are defined (`title`, `sidebar` and `body`). Each block may be overridden by a child template or left with its default implementation. This template could also be rendered directly. In that case the `title`, `sidebar` and `body` blocks would retain the default values used in this template.

A child template might look like this:

```
1  {% templates/blog/index.html.twig %}
2  {% extends 'base.html.twig' %}
3
4  {% block title %}My cool blog posts{% endblock %}
5
6  {% block body %}
```

```

7      {% for entry in blog_entries %}
8          <h2>{{ entry.title }}</h2>
9          <p>{{ entry.body }}</p>
10     {% endfor %}
11 {% endblock %}

```

Note

The parent template is stored in `templates/`, so its path is `base.html.twig`. The template naming conventions are explained fully in [Template Naming and Locations](#).

The key to template inheritance is the `{% extends %}` tag. This tells the templating engine to first evaluate the base template, which sets up the layout and defines several blocks. The child template is then rendered, at which point the `title` and `body` blocks of the parent are replaced by those from the child. Depending on the value of `blog_entries`, the output might look like this:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="UTF-8">
5          <title>My cool blog posts</title>
6      </head>
7      <body>
8          <div id="sidebar">
9              <ul>
10                 <li><a href="/">Home</a></li>
11                 <li><a href="/blog">Blog</a></li>
12             </ul>
13         </div>
14
15         <div id="content">
16             <h2>My first post</h2>
17             <p>The body of the first post.</p>
18
19             <h2>Another post</h2>
20             <p>The body of the second post.</p>
21         </div>
22     </body>
23 </html>

```

Notice that since the child template didn't define a `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used by default.

Tip

You can use as many levels of inheritance as you want! See [How to Organize Your Twig Templates Using Inheritance](#) for more info.

When working with template inheritance, here are some tips to keep in mind:

- If you use `{% extends %}` in a template, it must be the first tag in that template;
- The more `{% block %}` tags you have in your base templates, the better. Remember, child templates don't have to define all parent blocks, so create as many blocks in your base templates as you want and give each a sensible default. The more blocks your base templates have, the more flexible your layout will be;
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template. In some cases, a better solution may be to move the content to a new template and `include` it (see [Including other Templates](#));
- If you need to get the content of a block from the parent template, you can use the `{{ parent() }}` function. This is useful if you want to add to the contents of a parent block instead of completely overriding it:

```

1  {% block sidebar %}
2      <h3>Table of Contents</h3>
3
4      {# ... #}
5
6      {{ parent() }}
7  {% endblock %}

```

Template Naming and Locations ¶

By default, templates can live in two different locations:

`templates/`

The application's `views` directory can contain application-wide base templates (i.e. your application's layouts and templates of the application bundle) as well as templates that [override third party bundle templates](#).

`vendor/path/to/CoolBundle/Resources/views/`

Each third party bundle houses its templates in its `Resources/views/` directory (and subdirectories). When you plan to share your bundle, you should put the templates in the bundle instead of the `templates/` directory.

Most of the templates you'll use live in the `templates/` directory. The path you'll use will be relative to this directory. For example, to render/extend `templates/base.html.twig`, you'll use the `base.html.twig` path and to render/extend `templates/blog/index.html.twig`, you'll use the `blog/index.html.twig` path.

Referencing Templates in a Bundle ¶

If you need to refer to a template that lives in a bundle, Symfony uses the Twig namespaced syntax (`@BundleName/directory/filename.html.twig`). This allows for several types of templates, each which lives in a specific location:

- `@AcmeBlog/Blog/index.html.twig`: This syntax is used to specify a template for a specific page. The three parts of the string, each separated by a slash (`/`), mean the following:
 - `@AcmeBlog`: is the bundle name without the `Bundle` suffix. This template lives in the `AcmeBlogBundle` (e.g. `src/Acme/BlogBundle`);
 - `Blog`: (*directory*) indicates that the template lives inside the `Blog` subdirectory of `Resources/views/`;
 - `index.html.twig`: (*filename*) the actual name of the file is `index.html.twig`.

Assuming that the `AcmeBlogBundle` lives at `src/Acme/BlogBundle`, the final path to the layout would be `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.

- `@AcmeBlog/layout.html.twig`: This syntax refers to a base template that's specific to the `AcmeBlogBundle`. Since the middle, "directory", portion is missing (e.g. `Blog`), the template lives at `Resources/views/layout.html.twig` inside `AcmeBlogBundle`.

Using this namespaced syntax instead of the real file paths allows applications to [override templates that live inside any bundle](#).

Template Suffix ¶

Every template name also has two extensions that specify the *format* and *engine* for that template.

| Filename | Format | Engine |
|-----------------------------------|--------|--------|
| <code>blog/index.html.twig</code> | HTML | Twig |
| <code>blog/index.html.php</code> | HTML | PHP |
| <code>blog/index.css.twig</code> | CSS | Twig |

By default, any Symfony template can be written in either Twig or PHP, and the last part of the extension (e.g. `.twig` or `.php`) specifies which of these two *engines* should be used. The first part of the extension, (e.g. `.html`, `.css`, etc) is the final format that the template will generate. Unlike the engine, which determines how Symfony parses the template, this is an organizational tactic used in case the same resource needs to be rendered as HTML (`index.html.twig`), XML (`index.xml.twig`), or any other format. For more information, read the [How to Work with Different Output Formats in Templates](#) section.

Tags and Helpers ¶

You already understand the basics of templates, how they're named and how to use template inheritance. The hardest parts are already behind you. In this section, you'll learn about a large group of tools available to help perform the most common template tasks such as including other templates, linking to pages and including images.

Symfony comes bundled with several specialized Twig tags and functions that ease the work of the template designer. In PHP, the templating system provides an extensible *helper* system that provides useful features in a template context.

You've already seen a few built-in Twig tags like `{% block %}` and `{% extends %}`. Here you will learn a few more.

Including other Templates ¶

You'll often want to include the same template or code fragment on several pages. For example, in an application with "news articles", the template code displaying an article might be used on the article detail page, on a page displaying the most popular articles, or in a list of the latest articles.

When you need to reuse a chunk of PHP code, you typically move the code to a new PHP class or function. The same is true for templates. By moving the reused template code into its own template, it can be included from any other template. First, create the template that you'll need to reuse.

```
1  {# templates/article/article_details.html.twig #}  
2  <h2>{{ article.title }}</h2>
```

```

3  <h3 class="byline">by {{ article.authorName }}</h3>
4
5  <p>
6      {{ article.body }}
7  </p>

```

Including this template from any other template is achieved with the `{{ include() }}` function:

```

1  {# templates/article/list.html.twig #}
2  {% extends 'layout.html.twig' %}
3
4  {% block body %}
5      <h1>Recent Articles</h1>
6
7      {% for article in articles %}
8          {{ include('article/article_details.html.twig', { 'article': article }) }}
9      {% endfor %}
10 {% endblock %}

```

Notice that the template name follows the same typical convention. The `article_details.html.twig` template uses an `article` variable, which we pass to it. In this case, you could avoid doing this entirely, as all of the variables available in `list.html.twig` are also available in `article_details.html.twig` (unless you set `with_context` to false).



Tip

The `{'article': article}` syntax is the standard Twig syntax for hash maps (i.e. an array with named keys). If you needed to pass in multiple elements, it would look like this: `{'foo': foo, 'bar': bar}`.

Linking to Pages ¶

Creating links to other pages in your application is one of the most common jobs for a template. Instead of hardcoding URLs in templates, use the `path` Twig function (or the `router` helper in PHP) to generate URLs based on the routing configuration. Later, if you want to modify the URL of a particular page, all you'll need to do is change the routing configuration: the templates will automatically generate the new URL.

First, link to the "welcome" page, which is accessible via the following routing configuration:

Annotations

YAML

XML

PHP

```

1  // src/Controller/WelcomeController.php
2
3  // ...
4  use Symfony\Component\Routing\Annotation\Route;
5
6  class WelcomeController extends AbstractController
7  {
8      /**
9       * @Route("/", name="welcome")
10      */
11      public function index()

```



```

12     {
13         // ...
14     }
15 }

```

To link to the page, use the `path()` Twig function and refer to the route:

```

1 <a href="{{ path('welcome') }}">Home</a>

```

As expected, this will generate the URL `/`. Now, for a more complicated route:

Annotations

YAML

XML

PHP

```

1 // src/Controller/ArticleController.php
2
3 // ...
4 use Symfony\Component\Routing\Annotation\Route;
5
6 class ArticleController extends AbstractController
7 {
8     /**
9      * @Route("/article/{slug}", name="article_show")
10     */
11     public function show($slug)
12     {
13         // ...
14     }
15 }

```

In this case, you need to specify both the route name (`article_show`) and a value for the `{slug}` parameter. Using this route, revisit the `recent_list.html.twig` template from the previous section and link to the articles correctly:

```

1 {# templates/article/recent_list.html.twig #}
2 {% for article in articles %}
3     <a href="{{ path('article_show', {'slug': article.slug}) }}">
4         {{ article.title }}
5     </a>
6 {% endfor %}

```



Tip

You can also generate an absolute URL by using the `url()` Twig function:

```

1 <a href="{{ url('welcome') }}">Home</a>

```

Linking to Assets ¶

Templates also commonly refer to images, JavaScript, stylesheets and other assets. You could hard-code the web path to these assets (e.g. `/images/logo.png`), but Symfony provides a more dynamic option via the `asset()` Twig function.

To use this function, install the `asset` package:

```
$ composer require symfony/asset
```

You can now use the `asset()` function:

```
1 
2
3 <link href="{{ asset('css/blog.css') }}" rel="stylesheet" />
```

The `asset()` function's main purpose is to make your application more portable. If your application lives at the root of your host (e.g. `http://example.com`), then the rendered paths should be `/images/logo.png`. But if your application lives in a subdirectory (e.g. `http://example.com/my_app`), each asset path should render with the subdirectory (e.g. `/my_app/images/logo.png`). The `asset()` function takes care of this by determining how your application is being used and generating the correct paths accordingly.



Tip

The `asset()` function supports various cache busting techniques via the [version](#), [version_format](#), and [json_manifest_path](#) configuration options.

If you need absolute URLs for assets, use the `absolute_url()` Twig function as follows:

```
1 
```

Including Stylesheets and JavaScripts in Twig ¶

No site would be complete without including JavaScript files and stylesheets. In Symfony, the inclusion of these assets is handled elegantly by taking advantage of Symfony's template inheritance.



Tip

This section will teach you the philosophy behind including stylesheet and JavaScript assets in Symfony. If you are interested in compiling and creating those assets, check out the [Webpack Encore documentation](#) a tool that seamlessly integrates Webpack and other modern JavaScript tools into Symfony applications.

Start by adding two blocks to your base template that will hold your assets: one called `stylesheets` inside the `head` tag and another called `javascripts` just above the closing `body` tag. These blocks will contain all of the stylesheets and JavaScripts that you'll need throughout your site:

```
1 {# templates/base.html.twig #}
2 <html>
```

```

3     <head>
4         {# ... #}
5
6         {% block stylesheets %}
7             <link href="{ asset('css/main.css') }}" rel="stylesheet" />
8         {% endblock %}
9     </head>
10    <body>
11        {# ... #}
12
13        {% block javascripts %}
14            <script src="{ asset('js/main.js') }"></script>
15        {% endblock %}
16    </body>
17 </html>

```

This looks almost like regular HTML, but with the addition of the `{% block %}`. Those are useful when you need to include an extra stylesheet or JavaScript from a child template. For example, suppose you have a contact page and you need to include a `contact.css` stylesheet *just* on that page. From inside that contact page's template, do the following:

```

1 {# templates/contact/contact.html.twig #}
2 {% extends 'base.html.twig' %}
3
4 {% block stylesheets %}
5     {{ parent() }}
6
7     <link href="{ asset('css/contact.css') }}" rel="stylesheet" />
8 {% endblock %}
9
10 {# ... #}

```

In the child template, you override the `stylesheets` block and put your new stylesheet tag inside of that block. Since you want to add to the parent block's content (and not actually *replace* it), you also use the `parent()` Twig function to include everything from the `stylesheets` block of the base template.

You can also include assets located in your bundles' `Resources/public/` folder. You will need to run the `php bin/console assets:install target [--symlink]` command, which copies (or symlinks) files into the correct location. (target is by default the "public/" directory of your application).

```

1 <link href="{ asset('bundles/acmedemo/css/contact.css') }}" rel="stylesheet" />

```

The end result is a page that includes `main.js` and both the `main.css` and `contact.css` stylesheets.

Referencing the Request, User or Session ¶

Symfony also gives you a global `app` variable in Twig that can be used to access the current user, the Request and more.

See [How to Access the User, Request, Session & more in Twig via the app Variable](#) for details.

Output Escaping ¶

Twig performs automatic "output escaping" when rendering any content in order to protect you from Cross Site Scripting (XSS) attacks.

Suppose `description` equals `I <3 this product`:

```
1  <!-- output escaping is on automatically -->
2  {{ description }} <!-- I &lt;3 this product -->
3
4  <!-- disable output escaping with the raw filter -->
5  {{ description|raw }} <!-- I <3 this product -->
```

! Caution

PHP templates do not automatically escape content.

For more details, see [How to Escape Output in Templates](#).

Final Thoughts ¶

The templating system is just *one* of the many tools in Symfony. And its job is simple: allow us to render dynamic & complex HTML output so that this can ultimately be returned to the user, sent in an email or something else.

Keep Going! ¶

Before diving into the rest of Symfony, check out the [configuration system](#).

Learn more ¶

- [How to Use PHP instead of Twig for Templates](#)
- [How to Access the User, Request, Session & more in Twig via the `app` Variable](#)
- [How to Dump Debug Information in Twig Templates](#)
- [How to Embed Controllers in a Template](#)
- [How to Escape Output in Templates](#)
- [How to Work with Different Output Formats in Templates](#)
- [How to Inject Variables into all Templates \(i.e. global Variables\)](#)
- [How to Embed Asynchronous Content with `hinclude.js`](#)
- [How to Organize Your Twig Templates Using Inheritance](#)
- [How to Use and Register Namespaced Twig Paths](#)
- [How to Render a Template without a custom Controller](#)
- [How to Check the Syntax of Your Twig Templates](#)
- [How to Write a custom Twig Extension](#)

This work, including the code samples, is licensed under a [Creative Commons BY-SA 3.0 license](#).

Latest from the Symfony Blog

[Symfony 4.2.4 released](#)

March 3, 2019

[Symfony 3.4.23 released](#)

March 3, 2019

They Help Us Make Symfony



Thanks **Jordan Hoff** for being a Symfony contributor.

1 commit · 2 lines

Get Involved in the Community

A passionate group of over 600,000 developers from more than 120 countries, all committed to helping PHP surpass the impossible.

[Getting involved](#) →

Symfony™ is a trademark of Symfony SAS. All rights reserved.

What is Symfony?

- [Symfony at a Glance](#)
- [Symfony Components](#)
- [Case Studies](#)
- [Symfony Roadmap](#)
- [Security Policy](#)
- [Logo & Screenshots](#)
- [Trademark & Licenses](#)
- [symfony1 Legacy](#)

Screencasts

- [Learn Symfony](#)
- [Learn PHP](#)
- [Learn JavaScript](#)
- [Learn Drupal](#)
- [Learn RESTful APIs](#)

Blog

- [Events & Meetups](#)
- [A week of symfony](#)

Learn Symfony

- [Getting Started](#)
- [Components](#)
- [Best Practices](#)
- [Bundles](#)
- [Reference](#)
- [Training](#)
- [Certification](#)

Community

- [SymfonyConnect](#)
- [Support](#)
- [How to be Involved](#)
- [Events & Meetups](#)
- [Projects using Symfony](#)
- [Downloads Stats](#)
- [Contributors](#)

Services

- [Our services](#)
- [Train developers](#)

[Case studies](#)

[Community](#)

[Conferences](#)

[Diversity](#)

[Documentation](#)

[Living on the edge](#)

[Releases](#)

[Security Advisories](#)

[SymfonyInsight](#)

[Manage your project quality](#)

[Improve your project performance](#)

About

[SensioLabs](#)

[Careers](#)

[Support](#)

Follow Symfony



Switch to dark theme