Views with Twig

By default, Symfony uses a rendering engine called Twig.
It is an HTML template engine that allows you to easily integrate variables and control structures into HTML without writing PHP.

Good to know: Twig can also be used without Symfony, in any PHP project.

It adds features to our HTML files.
You will notice that our files are named **name.html.twig**.

These twig files will not contain any PHP. When we want to insert data, we will use twig's own features.

base.html.twig

Before looking at the file called by our controller, let's focus on **base.html.twig**.

It starts like a classic HTML file until we reach curly braces {}. These braces are twig's opening and closing tags, and there are 3 types:

- {{ }} the double curly braces correspond to a PHP "**echo**".
- {# #} corresponds to a comment in twig.
- {% %} corresponds to code without output.

Blocks

So in this base file, we find the following twig:

- In the title tag, we have twig with the keyword "**block**". Blocks are named content areas that can be overridden in other templates.
  By default, a block displays the content it contains. But as soon as a child template redefines this block, the original content is replaced.
- Then we find a block reserved for stylesheets (css) and one for javascript.
  - The latter contains itself a block reserved for "importmap" which we will talk about later.
  - These blocks can for example allow adding or replacing scripts and stylesheets that might differ on another page.
- Finally, in the body, we find an empty block body.

Before leaving, let's change the default title to "Symfony Course".

home/index.html.twig

When we create a controller, Symfony automatically generates a corresponding "index" view in a folder named after the controller.

Let's see what is written by default:

- We have the keyword "**extends**" which indicates that this file inherits from "base.html.twig". Most page templates will contain this.

Important: When a template inherits from another ({% extends "base.html.twig" %}), all its content must be inside blocks.
Otherwise, it will be ignored.

- Then we have the title block again, if we look at the page title, it corresponds to this tag. But if we comment out this tag, we will see the default title we defined.
  We can also choose not to erase the default value but just add ours thanks to the "**parent()**" function.
  Let's replace the current title with:

```
{{ parent() }} - Home
```

The title is now a merge of the default one and the new one.

- The next element is the body block. It was empty in the "base" file but here we have some base elements indicating the location of the controller and the template, which we can remove.
  Everything here will replace the empty body of the base file.
- There is nothing else here but if we needed a css or js file specific to this page, we could call the corresponding blocks.
- The block names are completely arbitrary; we can name them as we want and create as many as we want. If we needed a footer that varies from one page to another, we could very well create a footer block and call it when it needs to change.

VS Code configuration

Note that **HTML emmet** no longer works.

Because in VS Code we are in a **twig** language file, not HTML anymore.
It's okay, we can fix this:

1. Go to VS Code settings and search for `emmet.includeLanguages`
2. Add an item.
3. Set the key to `twig`.
4. Set the value to `html`.

Now VS Code understands that when editing twig, it should continue to include HTML emmet.

Display

As I said earlier, the render method allows us to provide variables to our view. If I write:

```
{{ controller_name }}
```

We will see the value of the variable displayed.

But when displaying a variable, we can also apply filters to modify the variable.

```
{{ controller_name | upper }}
```

The value of our variable is now in uppercase.

Many filters and functions are available; we can even create our own. Meanwhile, the full list is here:

https://twig.symfony.com/doc/3.x/#reference

When our variable corresponds to an associative array or an object, we can access the value directly with:

"variable.property"

For example, if we take our variable "pays":

```
{{ pays.france }}
<br>
{{ pays.england | reverse }}
```

WARNING: twig cannot directly display a complete array.

Conditions

Twig also allows handling if, elseif and else conditions:

```
<strong>
{% if chiffre > 5 %}
    {{ chiffre }} is greater than 5
{% elseif chiffre < 5 %}
    {{ chiffre }} is less than 5
{% else %}
    Your number is 5.
{% endif %}
</strong>
```

Arrays

And of course, we can loop through an array with twig:

```
<ul>
    {% for f in fruits %}
        <li>{{ f }}</li>
    {% endfor %}
</ul>
```

Filters can also be used with our for loop; for example, add "**| reverse**" after our array. The array will then be reversed.

One nice thing about twig is that it can handle empty arrays such as those returned by a SQL query with no results. For this, we just need to include an "**else**" in our "**for**".

```
{% for v in vide %}
    The array is empty, so this will not be displayed.
{% else %}
    Your array is empty.
{% endfor %}
```

XSS

By default, Symfony’s twig library is configured to filter all displayed data to prevent XSS attacks:

```
{{ xss }}
```

If we want to add HTML or JS via a variable to display, we must add the `raw` filter;

header and footer

Let's step away from this file for a moment to create in our templates a "**layout**" folder containing the files "**_header.html.twig**" and "**_footer.html.twig**" and look at the first one.

In the h1 we placed two variables that will allow us to change it if we want a different header from page to page:

- header will allow changing the entire content of our header.
- title will allow changing just the title inside the h1.

```
<header>
    {% if header is defined %}
        {{ header|raw }}
    {% else %}
    <h1>
        {{ title ?? "Symfony Course" }}
    </h1>
    {% endif %}
</header>
```

In the footer we find a link to our homepage. One difficulty when developing a site is to plan in advance how all the routes of our site will be constructed.
If we change our mind later, it forces us to go through the entire site to change each link. This is not the case with Symfony and Twig.

```
<footer>
    <a href="">Back to homepage</a>
```

```
</footer>
```

path()

If you remember, we called the route to our homepage "**app_home**". Well, I will use twig's "path()" function with the route name as argument.

```
<a href="{{ path("app_home") }}">Back to homepage</a>
```

This function will return the absolute link corresponding to this name. So as long as I don't change the name, the route may change, it will automatically be changed wherever I use this function.

Note that we can also use `url("routeName")` for example in an email, which will return the link including the domain name.

include

Now let's place in our "base.html.twig" file the following lines around our body block:

```
{% include "layout/_header.html.twig" %}
<main class="{{ mainClass ?? '' }}">
    {% block body %}{% endblock %}
</main>
{% include "layout/_footer.html.twig" %}
```

As the name suggests, the keyword **include** allows including other files.

These twig files will not contain PHP. When we want to insert data, we will use twig’s own features.

Note that I wrapped our pages in a "main" tag that has an optional class that can be added directly in our pages by setting the variable "mainClass".

Including CSS

We will talk more about how CSS and JS work with Symfony later.
For now, we will just copy and paste the included CSS from the course resources into the following file:

`/assets/styles/app.css`

The global "app" variable

This global variable is available anywhere in twig.

It allows many things such as:

- Retrieving information about the connected user.
- Retrieving flash messages.

- Retrieving information from the session.
- Retrieving information about the current request.
- And so on (see the official documentation for full details).

Here we will use it to get the current route name.
Let's add this to our header:

```
<nav class="header-navigation">
    <a href="{{ path('app_home') }}" class="{{ app.current_route == 'app_home' ?
'active' : '' }}">Home</a>
    <a href="" class="{{ app.current_route == 'app_user_list' ? 'active' : ''
}}">User List</a>
</nav>
```

Now if we are on a route named "app_home" then the corresponding link will get a class.
The same for the second link with "app_user_list" which we will build later.

Including a controller

Imagine we want to have on the right side of our site a list of recent articles. We are not going to make the query for these in each of our controllers.
Symfony and twig allow us to include not another view but an entire controller.

For that, the concerned view must not contain "**extends**" or other blocks but only the content that should be displayed.

Then we place ourselves where we want to display our element and write:

```
{{ render(controller("App\\Controller\\ControllerName::MethodName")) }}
```