Server-side
Web Development

Unit 11. Symfony. Controllers, routes and views.

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 MVC (Model - View - Controller)

MVC is acronym of **Model-View-Controller**, a software design pattern. By using this pattern, we divide our applications in three main parts:

- **Model**: is the section where we work with the data. Usually these data are extracted from databases or other information sources. Our model classes will contain the instructions to connect and work with these information sources.
- **View**: is the section that connects the application with the user. Usually the views are interfaces to show information or ask the user for information.
- **Controller**: is the section that coordinates and communicates the other two. The controllers accepts some inputs and transform them to instructions and calls to the model or the view.

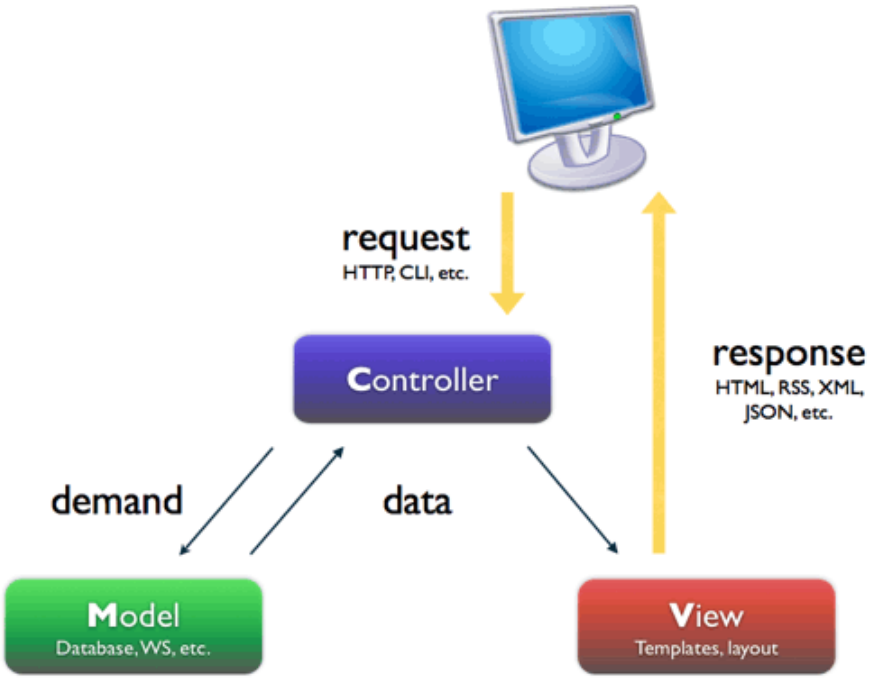

Figure 1: MVC pattern in Symfony

MVC is a very structured pattern that allows independence between the sections. It has been widely adopted as a web design pattern, and a lot of frameworks use this pattern. Because of its modular structure, is a great approach to work as a team.

2 Controllers and routes

The interaction between the user and the web application usually works as follows:

- The user sends a ***request***, mainly by writing (or calling in any other way) an URL
- In a PHP application using the MVC pattern, this request has to be redirected to a ***controller***, that will be a PHP callable piece of code (usually a class method)
- The controller receives the request, and then it has to create a ***response*** for that request. The response can be rendering a view with Twig, returning html code or json code, and so on.

A ***route*** is the link that connects the requested path (like *localhost:8000/*, or any other local URL) and a controller.

2.1 The Symfony Maker Bundle

In Symfony the controllers are implemented as classes. These classes can be created manually, but a better and faster way is to use the ***Symfony Maker Bundle***.

The maker bundle not only generates controllers but a lot of different and useful classes. When using the maker we will need to specify which kind of element we want to generate. If we run:

```
symfony console list make
```

> Alternatively, if you don't have the Symfony-CLI installed, you can use `bin/console` inside a Symfony project to execute Symfony commands:

```
bin/console list make
```

We will see on screen all the generators provided by the maker.

```
eljust@fidel-VirtualBox:~/prova$ symfony console list make
Symfony 5.4.2 (env: dev, debug: true)

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display help for the given command. When no command is given display help for the list command
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi|--no-ansi  Force (or disable --no-ansi) ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -e, --env=ENV         The Environment name. [default: "dev"]
      --no-debug        Switch off debug mode.
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands for the "make" namespace:
  make:auth                   Creates a Guard authenticator of different flavors
  make:command                Creates a new console command class
  make:controller             Creates a new controller class
  make:crud                   Creates CRUD for Doctrine entity class
  make:docker:database        Adds a database container to your docker-compose.yaml file
  make:entity                 Creates or updates a Doctrine entity class, and optionally an API Platform resource
  make:fixtures               Creates a new class to load Doctrine fixtures
  make:form                   Creates a new form class
  make:functional-test        Creates a new test class
  make:message                Creates a new message and handler
  make:messenger-middleware   Creates a new messenger middleware
  make:migration              Creates a new migration based on database changes
  make:registration-form      Creates a new registration form system
  make:reset-password         Create controller, entity, and repositories for use with symfonycasts/reset-password-bundle
```

Figure 2: List of elements of Symfony Maker Bundle

If you don't have the maker bundle installed (the usual if you are in a microservice project), you'll get an error. Install the Symfony maker bundle with:

```
composer require --dev symfony/maker-bundle
```

2.2 Routes with attributes

From PHP 8.0 on, we can define routes and required dependencies using attributes instead of annotations.

With annotations:

```
/**
* @Route("/path", name="action")
*/
```

With attributes (PHP 8.0 or higher):

```
#[Route('/path', name: 'action')]
```

Let's see an example:

```
class MainController extends AbstractController {

    #[Route('/main', name: 'main')]
    public function index() {
        return $this->render('main/index.html.twig');
    }
}
```

The content of the *index* function is not important. The main thing is learning how to connect our route (*/main*) with our controller (the *index* method in the *MainController* class). We will learn more about annotations, routes and controllers later.

Symfony recommends the use of attributes.

> Symfony also supports another methods to make routes, using YAML, XML or PHP configuration files in the `config` dir: https://symfony.com/doc/current/routing.html#creating-routes-in-yaml-xml-or-php-files

2.3 Creating a controller

Now we can create our first controller. The Maker offers us an easy way to create a controller:

```
symfony console make:controller HelloController
```

This command will create a class named ***HelloController*** in the *src/Controller* folder of our project which serves a Json response:

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Routing\Annotation\Route;

class HelloController extends AbstractController
{
    #[Route('/hello', name: 'app_hello')]
    public function index(): JsonResponse
    {
```

```
        return $this->json([
            'message' => 'Welcome to your new controller!',
            'path' => 'src/Controller/HelloController.php',
        ]);
    }
}
```

Again, it's not important what the index function does. By now we only need to check if the connection between the route (/hello) and the controller (index) is working fine.

Let's go to our browser and write the URL `http://localhost:8000/hello` (the route begins in our local server root page, which is `http://localhost:port`)

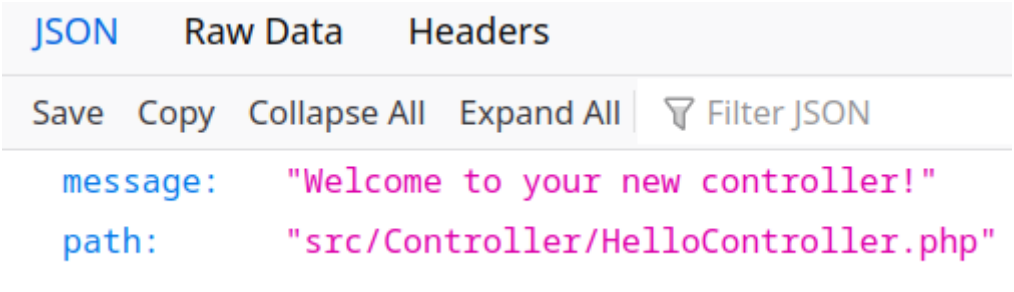

Figure 3: Our first controller

As you can see, the route `/hello` returns a Json response.

This is fine if we are making a web service, but if we want to show a webpage instead of Json data, we need a template system. Symfony uses **Twig** as template engine.

To install Twig, write in the command line:

```
symfony composer require twig
```

Now, delete the `HelloController.php` and create again the same controller:

```
symfony console make:controller HelloController
```

It generates again the controller:

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class HelloController extends AbstractController
{
    #[Route('/hello', name: 'app_hello')]
    public function index(): Response
    {
        return $this->render('hello/index.html.twig', [
            'controller_name' => 'HelloController',
        ]);
    }
}
```

Figure 4: Our first view

Now, we have a different code in the body of `index()`. Note how the function calls the render method and pass to it the parameter `controller_name` whit the value `'HelloController'`.This string is shown in the view title.

This is how the whole thing works: the user writes a ***route***, this route leads to a certain method in a ***controller***, and, then, the controller does something and leads to the ***view*** that we can see on the screen.

As we don't know about views yet, we can change the content of the method so we can show a different

message:

```
public function index(): Response
    {
        /*return $this->render('hello/index.html.twig', [
            'controller_name' => 'HelloController',
        ]);*/
        return new Response("<h2>Hola!</h2>");
    }
```

Now we can refresh the page in our browser:

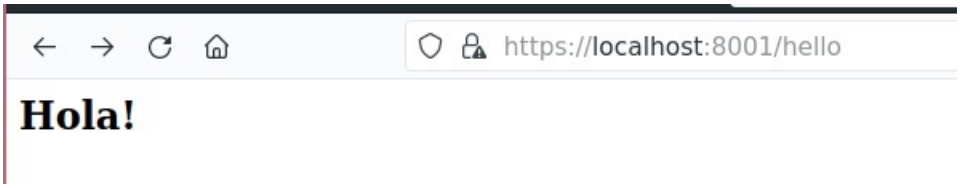

Figure 5: Changing the output

We can create our own homepage by adding a new method to the class with the route "/". We don't need the template, so you can create and write the class directly or use the `symfony maker` and then delete the view.

```
#[Route('/', name: 'homepage')]
public function main(): Response {
    return new Response("
        <html>
        <body>
            <div style='text-align:center;'>
            <h2>This is my first Symfony web</h2>
            <h3>Welcome!</h3>
            <img src='/images/under-construction.gif'>
            </div>
        </body>
        </html>"
    );
}
```

Now if we write on the browser `http://localhost:8000/` we will get our new homepage:

Figure 6: Our new homepage

> **Note**: To show the image correctly, you need to make a new folder inside the `public` folder named `images` and then save the image inside.

2.4 Passing parameters to the controller

We can send parameters to a method by adding GET arguments in the URL.

For instance: let's personalize our welcome message.

First, we need to attach the argument to the route:

```
#[Route('/{name}', name: 'homepage')]
```

Then, we pass the parameter with the same name to the function related to the route:

```
#[Route('/{name}', name: 'homepage')]
public function main($name=''): Response
{
```

Now we can use the *$name* variable in the function to personalize our welcome message:

```php
public function main($name=''): Response
    {
        $message="Welcome";
        if($name!='') {
            $message=$message." ".$name."!";
        }
        return new Response("
            <html>
                <body>
                    <div style='text-align:center;'>
                    <h2>This is my first Symfony web</h2>
                    <h3>$message</h3>
                    <img src='/images/under-construction.gif'/>
                    </div>
                </body>
            </html>"
        );
    }
    }
```

Finally, call the URL with a GET parameter attached:

Figure 7: A controller with parameters

If we call the same URL without the GET parameter, *$name* will get is default value `main($name='')` and therefore the message will be different (just Welcome! without a name):

We can pass two or more parameters attached to the URL. In this case, we need to define the same

Figure 8: The same controller without passing the parameter

parameters in the route and in the method.

```
#[Route('/{name}/{lastname}', name: 'homepage')]
public function main($name='', $lastname=''): Response
```

Figure 9: Controller method with 2 parameters

More about controllers: https://symfony.com/doc/current/controller.html

2.5 Troubleshooting with the controller parameters

What happens if we have this two pairs route-controller?

```
#[Route('/{name}', name: 'homepage')]
public function main($name=''): Response
```

```
#[Route('/hello', name: 'app_hello')]
public function index(): Response
```

When we write the URL `http://localhost:8000/hello` the function index() should be called, but…

Figure 10: Wrong method

The router thinks that “hello” is the value of the *name* argument in the URL, so the function main() is invoked with the parameter *name* equals to “hello”.

To know which route is taken by Symfony, we can write in the console the command:

```
symfony console debug:route
```

```
 ------------------------- -------- -------- ------ ---------
  Name                     Method   Scheme   Host   Path
↪
 ------------------------- -------- -------- ------ ---------
  ...
  homepage                 ANY      ANY      ANY    /{name}
  app_hello                ANY      ANY      ANY    /hello
↪

 ------------------------- -------- -------- ------ ---------
```

We can see that Symfony is taking the first method in our `HelloController` file.

If we have two routes with the same name and we want them to lead to different methods, we need to differentiate between them in some way. It might be:

- Using the Priority parameter:

```
#[Route('/{name}', name: 'homepage')]
public function main($name=''): Response
```

```
#[Route('/hello', name: 'app_hello', priority: 2)]
public function index(): Response
```

The priority parameter expects an integer value. Routes with higher priority are sorted before routes with lower priority. The default value when it is not defined is 0.

- By having a different number of parameters

```
#[Route('/{name}', name: 'homepage')]
public function main01...
```

```
#[Route('/{name}/{lastname}', name: 'homepage')]
public function main02...
```

- By having the same number of parameters but of different type:

```
#[Route('/{customer}/{id<\d+>}', name: 'customer01')]
public function customer01...
```

```
#[Route('/{customer}/{name}', name: 'customer02')]
public function customer02...
```

If the parameter is a number, the function *customer01* will be executed. If the parameter is not a number, then the function *customer02* will be called instead.

> \d+ is a **regular expression** that matches a digit of any length.

- By changing the route adding levels to it:

```
#[Route('/{customer}/id/{id}', name: 'customer01')]
public function customer01...
```

```
#[Route('/{customer}/name/{name}', name: 'customer01')]
public function customer02...
```

The route `http://localhost:8000/id/1` leads to function *customer01*, while the route `http://localhost:8000/name/1` would lead to function *customer02*.

More about routes: https://symfony.com/doc/current/routing.html

3 Views

As we have seen in previous sections, **MVC** is acronym of **Model-View-Controller**, a software design pattern. The idea of MVC is to separate the **model** (data) from the **view** (user interface). The controller coordinates the communication between the model and the view.

We want our views, controllers and models to be quite independent of each other. That's the reason why we don't want to implement interfaces in the controllers. We must be able to change our interfaces without modifying neither the controllers nor the models.

On the other hand, our interfaces might be too complex to handle them in a `return new Response(...)` clause. We need to implement interfaces that interact with the user, with styles, forms and a lot of features. Therefore, we are going to completely separate our interfaces from our controllers.

Our goal will be to separate, as much as possible, our PHP code from the HTML code. Basically, the **controller** (PHP code) obtains the data that we want to show and send them to the **views**. The result is what the user will see on the screen.

Symfony offers a tool to build templates for our interfaces: **Twig**.

3.1 HTML templates with Twig

Twig is a template engine for PHP. Twig makes the code faster by compiling the templates to create optimized and secure PHP code. By using Twig templates, we will avoid the *echo* instruction and functions like *htmlspecialchars* and others. Twig offers elements like blocks, automatic escaping, shortcuts for common structures, multiple inheritance and much more.

Usually Twig will be incorporated into our Symfony full web apps projects. Anyway, if we want to install Twig in a project that doesn't has the bundle, we can do:

```
symfony composer req twig
```

That adds Twig as a required dependency for our project.

When we create a project with Twig installed on it, a ***templates*** directory will be created. Inside that folder, we can find a sample layout template named *base.html.twig*. Twig templates have the extension *.html.twig*. The template *base.html.twig* is the basic structure from which the rest of our templates will inherit.

For every controller, we will get a subfolder in the `templates` directory. And for every method in our controller, we will get a new template. In our example (*HelloController*) we created a method called *index*, and automatically we got the Twig template *index.html.twig* as you can see below.

Figure 11: Twig templates structure

The *base* template defines some sections called *blocks*. We will get at least one block for the title, one block for styles, one block for the body and one block for javascript.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg
        ↪  xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128
        ↪  128%22><text y=%221.2em%22 font-size=%2296%22>·</text></svg>">
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Our own templates will be an extension of *base.html.twig*. By doing that, we just need to fill the blocks in the child templates with our HTML code. As we will see later, we can add variables and some Twig code as well.

Let's try first something simple in order to learn how to connect the controllers with the views.

3.2 Our first view

Let's create a new route `'/main'` with a new method name, `main2`:

```php
#[Route('/main', name: 'main')]
public function main2(): Response {
    return new Response("
        <html>
        <body>
        <div style='text-align:center;'>
        <h2>This is my first Symfony web</h2>
        <h3>Welcome!</h3>
        <img src='/images/under-construction.gif'/>
        </div>
        </body>
        </html>"
    );
}
```

The **`main2()`** function is associated to the route `/main`. So, if we write *`http://localhost:8000/main`* in our browser we will get:

Figure 12: Basic welcome page

Now, we will try this using a view.

First, we need to create a view. Let's call it **main.html.twig**. As usual, the view will extend the **base.html.twig**. We will need at least two blocks: one for the **title** and one for the **body**. At first, the views should be like follows:

```
{% extends 'base.html.twig' %}
{% block title %} {% endblock %}
{% block body %}
{% endblock %}
```

Of course, the view doesn't show anything on the screen. Let's fill the blocks with out content:

```
{% extends 'base.html.twig' %}
{% block title %} My first Symfony page{% endblock %}
{% block body %}
<div style='text-align:center;'>
    <h2>This is my first Symfony page</h2>
    <h3>Welcome!</h3>
    <img src='/images/under-construction.gif'/>
</div>
{% endblock %}
```

We have included in our *block body* the same content that we were including in our response.

Next step is to render the view from the controller. Modify the `main2` function:

```
public function main2(): Response {
    return $this->render('hello/main.html.twig');
}
```

The method **render** is inherited from the class **AbstractController** which we extended in our **HelloController** class.

Now, if we go to `http://localhost:8000/main` we will get the same result, but without including any HTML code in our controller.

Figure 13: The result of our view

3.3 Passing parameters to the view

We can send an **array of parameters** as the second argument of the *render* function.

```
$name = "Fidel";
return $this->render('hello/main.html.twig', [
        'name' => $name
    ]);
```

Now we can show the value of the variable *name* in our view, but without using an *echo* instruction.

In the Twig templates we can use two basic notations to include dynamic content:

- The {% %} notation to indicate an action or a structure.
- The {{ }} notation to display something.

In this case, we want our view to receive the variable *name* and then to show its value on the screen. We just need to make a little change to the view to include the *name* variable and display it.

```
...
<h3>Welcome, {{ name }}!</h3>
...
```

> **ATTENTION!** It's recommended to include a blank space after and before the content of the **{{ }}** variable.

Let's refresh our main page:

Figure 14: Our view with a *name* parameter

A better way to show a name, is to include it in the route. To do that, add the `name` parameter to the route and to the function's parameters:

```
#[Route('/main/{name}', name: 'main', priority: 2)]
public function main2($name = ''): Response {
    return $this->render('hello/main.html.twig', [
        'name' => $name
    ]);
}
```

Check the route adding a name to the route, for instance `localhost:8000/main/John`.

3.4 Accessing to properties and methods from Twig

In Twig we have an universal method to access to properties of a given object. That way is:

```
object.property
```

It works for **associative arrays**, for **object properties** and for **object methods**. For instance, if we send a parameter `contact` to a Twig template, if we write:

```
contact.name
```

Twig will search for one of these elements, in this order:

1. **$contact['name']** (a property of the $contact array)
2. **$contact->name** (a public attribute of the $contact object)
3. **$contact->name()** (a method of the $contact object)
4. **$contact->getName()** (a getter method of the $contact object)
5. **$contact->isName()** (object and isser method);
6. **$contact->hasName()** (object and hasser method);
7. If none of the above exists, use `null` (or throw a `Twig\Error\RuntimeError` exception if the `strict_variables` option is enabled).

3.5 Control structures in Twig

We can use the **{% %}** blocks to create control structures like iterations and alternatives.

3.5.1 If in Twig

We can use the **if** structure like in other programming languages.

```
{% if condition %}
    ...
{% elseif condition %}
    ...
{% else %}
    ...
{% endif %}
```

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

Example:

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

You can also use **not** (equivalent to ! in PHP) 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 product.stock > 10 %}
   Available
{% elseif product.stock > 0 %}
   Only {{ product.stock }} left!
{% else %}
   Sold-out!
{% endif %}
```

3.5.2 For in Twig

We can use the **for** structure to loop over each item in a given sequence, like an array or a explicit list of values.

```
{% for i in 1..10 %}
    ...
{% endfor %}
```

If we send an array named *contacts* to a Twig view:

```
{% for contact in contacts %}
    <li> Name:  {{ contact.name }} </li>
{% endfor %}
```

By default the **for** loop iterates over the values. We can iterate over the keys, too.

```
{% for key in contacts|keys %}
    <li>{{ key }}</li>
{% endfor %}
```

We can iterate over the keys and values like in `foreach($array as $key=>$value)` in php:

```
{% for key, contact in contacts %}
    <li>{{ key }}: {{ contact.name }}</li>
{% endfor %}
```

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>
```

Inside a *for* we can access to some **loop properties** like:

Loop property	Description
`loop.index`	The current iteration beginning with 1
`loop.index0`	The current iteration beginning with 0
`loop.first`	True if it's the first iteration
`loop.last`	True if it's the last iteration
`loop.length`	Total number of iterations

```
{# Prints a list of usernames with a number beginning with 1 #}
{% for user in users %}
    {{ loop.index }} - {{ user.username }}
{% endfor %}
```

3.6 Filters

The **filters** modify the content before rendering the page. The syntax is:

```
{{ variable|filter }}
```

For instance, we can change a string to uppercase doing:

```
{{ variable|upper }}
```

To find out the length of an array:

```
{{ arrayname|length }}
```

To display a date in a specific format:

```
{{ date|format_datetime('date format','hour format', locale='...') }}
```

Example:

```
{{ birthDate|date("d-m-Y", "Europe/Madrid") }}
```

To escape characters:

```
{{ value|escape }} or {{ value|e }}
```

By default, the escaping strategy consists in applying the `htmlspecialchars()` php function.

There are a lot of filters. Some of them are Twig natives, and some others are provided by Symfony. You can check them out at these pages:

Twig filters

Filters and functions defined by Symfony

3.7 Linking to other pages

Although we can create links to other pages in our web in the usual HTML way, linking to the URL associated to the page, is a good practice to use the route name instead. By doing this, we avoid problems if a URL route changes (as long as the name of the route doesn't change).

For instance, if we have this route:

```
#[Route('/main/{name}', name: 'main')]
```

We can link to the page from Twig doing:

```
<a href="/main/{{ name_value }}">Link to the main page</a>
```

But it is better to use the function **path** with the name of the route and the parameters:

```
<a href="{{ path('main', { name: name_value }) }}">Link to the main page</a>
```

3.8 Linking to static content (CSS, JavaScript, images)

The static content always should be placed in the **public** folder or a subfolder inside *public*. So, the route of the content will be rooted on the *public* folder.

We can create a css folder inside *public* with a *styles.css* file inside it. Then, we can do:

```
<link href="/css/styles.css" rel="stylesheet" />
```

or

```
<link href="{{ asset('css/styles.css') }}" rel="stylesheet" />
```

Before using the `asset` function, you probably will need to install the `asset` package on your project. Run:

```
composer require symfony/asset
```

The advantage of using the **asset** function is that it references the root in our host even if we change that host. In addition, we can define specific assets in our configuration files to make our links shorter.

More information in this link:

More about assets in Twig

It works the same way for javascript content and images.

```
<img src="{{ asset('images/image.png') }}" />
```

```
<script src="{{ asset('js/scriptname.js') }}"></script>
```

3.9 Inheritance, extending blocks and including templates

As you know, we can extend a Twig template by doing:

```
{% extends 'name_of_the_template.html.twig' %}
```

And then, fill in the blocks defined in the parent template.

We can extend a block from our parent template by using the *parent()* function. By doing so, we include the content of the given block in the parent template and we complete the block with some new code.

For instance, if we have a block for a css file in a template, and we want to add a new css file in the inherited template, we could do:

```
{% extends name_of_parent_twig.html.twig %}
{% block stylesheets %}
    {{ parent() }}  //the content of the parent stylesheets block
    <link href="{{ asset('new_css_file.css') }} rel="stylesheet" /> // the
     ↪  added content
{% endblock %}
```

We can include a template in other templates in order to reuse its code. It works like the `include` in php.

```
{{ include('other_template.html.twig') }}
```

If the template requires parameters:

```
{{ include ('other_template.html.twig', { 'parameter': value }) }}
```

3.10 Twig official documentation

Twig has a lot more features that you can find in the official Twig website.

Twig official documentation

4 Services

In Symfony, we have several utilities, (logger, mailer, etc) that are called **services**. Each service lives inside a very special object called the **service container**. The container allows you to centralize the way objects are constructed.

4.1 Using services

If you want to use a service you only need to import it with the **use** keyword:

```
use Psr\Log\LoggerInterface;
...
class ProductController extends AbstractController
{
    #[Route('/products')]
    public function list(LoggerInterface $logger): Response
    {
        $logger->info('Look, I just used a service!');

        // ...
    }
}
```

If you want to know what services are available just run:

```
symfony console debug:autowiring
```

4.2 Creating services

You can create your own services. It's the better way to share functionalities between controllers.

To create a new service, create a new class in the src/Service folder (create the folder if needed). for instance, the next class is used to generate a welcome message:

```
// src/Service/HelloGenerator.php

namespace App\Service;

class HelloGenerator
{
    public static function getWelcome(): string {
        return "Welcome to my Symfony App!";
    }
}
```

You can use it immediately inside your controller including a `use` directive:

```
use App\Service\HelloGenerator;
...

class HelloController extends AbstractController
{
    #[Route('/hello', name: 'app_hello')]
    public function index(): Response
    {
        $welcomMsg = HelloGenerator::getWelcome();
        return new Response("
            <html>
            <body>
                <h1>$welcomMsg</h1>
                </body>
            </html>"
        );
}
```

More about services: https://symfony.com/doc/current/service_container.html