Symfony Features

Symfony offers many convenient features for our controllers. Let's go back to our "**HomeController.php**" file.

Route parameters

One practical feature of routers is being able to avoid using GET parameters by retrieving data directly from the URL. Obviously, Symfony allows this.

Let's create a new route:

```
\#[Route("/bonjour/{nom}/{prenom}", name: "app_bonjour")]
public function bonjour($nom, $prenom, Request $request): Response
{
    dump($request);
    // dd($request);
    return $this->render("home/bonjour.html.twig", [
        "nom" => $nom,
        "prenom"=> $prenom
    ]);
}
```

Make sure that the classes used, such as "Request" here, have their corresponding "use" statements:

```
use Symfony\Component\HttpFoundation\Request;
```

Braces in the route allow us to create parameters that can be retrieved by method arguments with the same names.

We will see later that route parameters can go much further.

We'll also create the corresponding template.
It should at least contain:

```
{% extends 'base.html.twig' %}
{% block body %}
    <h2>Hello {{prenom ~ " " ~ nom}}</h2>
{% endblock %}
```

The tilde "~" in Twig is the concatenation symbol.

Essential Functions and Classes

Now let's look at the third argument called request.

When we indicate a class for an argument, Symfony will automatically instantiate it and assign it to the parameter. In this case, we used:

```
use Symfony\Component\HttpFoundation\Request;
```

dump and dd

Using Symfony's *dump()* function lets us inspect what's inside this object.
However, when we load the page, nothing shows up... Where did our dump go? Something has changed—at least while we're in development mode.

In dev mode, Symfony displays a toolbar at the bottom of the screen. This toolbar contains a lot of useful development information. Let's focus on the target icon that appears to the right of the first section. Hover over it to see your dump—colorful and interactive.

Try dumping different types: strings, numbers, arrays—you'll see that *dump()* is much more convenient than PHP's classic `var_dump()`.

There's also the *dd()* function, which means "dump and die"—it dumps and then stops execution. In this case, the dump appears directly on the page.

Twig also supports **dump()**.
The result will be displayed directly in the browser.

Request

Let's explore the contents of this Request object we dumped earlier:

- `attributes`: contains information related to our route, including its parameters.
- `request`: contains all `$_POST` data.
- `query`: contains all `$_GET` data.
- `server`: contains all `$_SERVER` data.
- `files`: contains all `$_FILES` data.
- `cookies`: contains all `$_COOKIE` data.
- `session`: contains all `$_SESSION` data.
- and much more like method, URI, language, etc.

So Request groups all PHP superglobals along with Symfony-specific information.

| PHP Superglobal | Symfony Request | |----------|----------| | `$_GET` | `$request->query` | | `$_POST` | `$request->request` | | `$_SERVER` | `$request->server` | | `$_FILES` | `$request->files` | | `$_COOKIE` | `$request->cookies` | | `$_SESSION` | `$request->getSession()` |

Specifics of Parameter Routes

Route order

WARNING: If I now want to create a route "/bonjour/anglais/{username}" and place it **after** the "bonjour" route, it will become inaccessible.

```
\#[Route("/bonjour/anglais/{username}", name: "app_hello")]
public function hello($username): RedirectResponse
{
    dd("Hello ".$username);
    return $this->redirectToRoute("app_bonjour");
}
```

The *redirectToRoute()* method lets us redirect to another route by name.

Symfony reads routes in the order they are defined.
So if I try to access "/bonjour/anglais/pierre", Symfony will match it to "bonjour" with "anglais" as `nom` and "pierre" as `prenom`.

If I place the more specific route above the general one, both will work—unless the name "anglais" is used as a parameter.

You can view the route matching details in the Symfony toolbar (bottom left > "Routing").

Generate parameter route with Twig

We've seen we can use `path()` to generate a URL by route name. If the route has parameters, you pass them as an array:

```
<a href="{{ path("app_bonjour", {nom: "Fontaine", prenom: "Jean"}) }}">Jean
Fontaine</a>
<a href="{{ path("app_hello", {username: "John"}) }}">John Doe</a>
```

Same idea for redirects in controllers:

```
// dd("Hello ".$username);
return $this->redirectToRoute("app_bonjour", ["nom"=>"smith",
"prenom"=>$username]);
```

Route prefix

Symfony allows us to prefix all routes within a controller class:

```
\#[Route('/user')]
class UserController
{
  \#[Route('/profil', name:"profil")]
  public function profil(): response{}
}
```

In this example, `/user/profil` is the full path to the "profil" route.
All other routes in this class will start with "/user".
We'll use this when we build the user CRUD.

Default route values

We can assign default values to route parameters just like we do for function parameters:

```
\#[Route("/bonjour/anglais/{username}", name: "app_hello", defaults:
["username"=>"John"])]
```

Or use the short syntax:

```
\#[Route("/bonjour/{nom}/{prenom?Jean}", name: "app_bonjour")]
```

Important: If there are multiple parameters, only those on the right can have default values.

Regex and routes

Currently, our parameters accept anything—even "55 22". Let's restrict that using regex:

```
\#[Route("/bonjour/anglais/{username}",
    name: "app_hello",
    defaults: ["username"=>"John"],
    requirements: ["username"=>"^[a-zA-Z]+$"])]
```

Now if we enter digits, Symfony will match the "bonjour" route instead.

You can also use the short version:

```
\#[Route("/bonjour/{nom<^[a-zA-Z]+$>}/{prenom<^[a-zA-Z]+$>?Jean}", name:
"app_bonjour")]
```

This ensures the route only accepts letters—ideal for slugs or IDs.

Symfony Developer Toolbar

We've already seen it can show dumps and routing—but it also shows:

- Request status and matched route.
- Page load time.
- Memory usage.
- Logged-in user (we'll come back to this later).
- Twig rendering details.

- Any dumps.

On the right:

- Server details.
- Symfony version and info.
- A toggle button to hide the toolbar.

This bar disappears in production mode.
Don’t forget to switch environments when deploying.

Session Management

Symfony has its own session management system.
Let’s add a visit counter to the "bonjour" page:

```
$sess = $request->getSession();

if($sess->has('nbVisite')) $nb = $sess->get("nbVisite")+1;
else $nb = 1;

$sess->set("nbVisite", $nb);
```

- `has()` checks if a session key exists.
- `get()` retrieves a session value (you can pass a default).
- `set()` sets a session value.
- `remove()` deletes a session key.

In Twig, to display the counter:

```
<h3>Already {{ app.session.get("nbVisite") }} visit(s)!</h3>
```

Flash messages

Symfony provides an easy-to-use flash message system.
Let’s add one to our `bonjour` controller:

```
$this->addFlash("bonjour", "Hello $prenom $nom!");
```

In the `hello` controller:

```
$this->addFlash("redirect", "You have been redirected!");
$this->addFlash("bonjour", "Hello $username");
```

In the template:

```
{% for message in app.flashes('bonjour') %}
    <div>{{ message }}</div>
{% endfor %}
```

Flash messages disappear after display.

You can retrieve multiple categories:

```
{% for categorie, messages in app.flashes(['bonjour', "redirect"]) %}
  {% for message in messages %}
    <div>
      <strong>{{ categorie }} :</strong>
      <br>
      {{ message }}
    </div>
  {% endfor %}
{% endfor %}
```

Or retrieve all:

```
{% for categorie, messages in app.flashes %}
```

Once done, let's simplify life and move this flash block into `base.html.twig`:

```
{% for label, messages in app.flashes %}
  {% for message in messages %}
    <div class="alert alert-{{ label }}">
      {{ message }}
    </div>
  {% endfor %}
{% endfor %}
```

We'll style it later with Bootstrap.