Module 06 - Routage et contrôleurs

Dans ce module, nous allons apprendre à mettre en place

- Le **routage** (les moyens d'accéder aux ressources);
- Les **contrôleurs** (les moyens de produire et retourner une réponse contenant la représentation de la ressource)

dans une application Symfony.

Ce sont les deux premiers composants essentiels de notre application web MVC.

Pré-requis

- Avoir installé Symfony CLI;
- Avoir crée un projet Symfony (par exemple `symfony new app --version="7.2" --webapp`);
- Être à la racine du projet;
- Avoir servi le projet en local (`symfony server:start -d`).

Les contrôleurs de Symfony

Un Contrôleur est une classe qui prend en entrée une requête HTTP (`Request`) et **retourne une réponse HTTP** (objet de type `Response`). On désignera également les **méthodes** d'une classe "contrôleur" comme des contrôleurs.

> Les middlewares d'Express (fonctions anonymes) ne sont rien d'autre que des contrôleurs...

Routing/Routage : **chaque méthode d'une classe contrôleur peut être associée à une/plusieurs URL(s) + Méthode(s) HTTP**.

Le composant de routage de Symfony se charge d'appeler le bon contrôleur et la bonne méthode en fonction de 1) l'URL demandée par le client 2) l'intention sur cette ressource (méthode HTTP) 3) toute logique custom et propriété de la requête.

Les contrôleurs de Symfony

Pour créer un contrôleur, créer une classe dans le dossier `src/Controller` :

```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

class SomeController extends AbstractController
{
    //Notre code...
}
```

- On place généralement le suffixe `Controller` dans le nom de la classe ;
- On étend généralement la classe par la classe abstraite `AbstractController`, pour bénéficier de données et de méthodes utiles.
- Un contrôleur a pour seule obligation de retourner un objet de type `Response` ;

Routage via les attributs PHP

Dans le fichier `config/routes.yaml`, renseigner les informations suivantes :

```
controllers:
    resource:
        path: ../src/Controller/
        namespace: App\Controller
    type: attribute
kernel:
    resource: App\Kernel
    type: attribute
```

Routage via les attributs PHP

Placer un attribut PHP sur un contrôleur ou l'une de ses méthodes (`src/Controller`) pour configurer le routage. Chaque Route a un argument `name` et possède un nom unique dans l'application (si non renseigné, symfony en fournit un par défaut).

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;

//Utiliser la classe Route du module Annotation
use Symfony\Component\Routing\Attribute\Route;

class SomeController extends AbstractController
{
    #[Route('/hi', name: 'hi')] // Routing par annotation
    public function index(): Response
    {
        return new Response("hello world");
    }
}
```

Un mot sur les attributs PHP

Les attributs permettent d'ajouter des informations de métadonnées structurées et lisibles *par la machine* sur les déclarations dans le code (classes, méthodes, fonctions, etc.).

Les métadonnées définies par les attributs peuvent ensuite être inspectées au moment de l'exécution à l'aide de l'API de Réflexion, (API "reflexive", qui permet d'inspecter le code *depuis* le code PHP).

Les attributs peuvent donc être considérés comme un langage de configuration intégré directement dans le code.

Un mot sur les attributs PHP

Une déclaration d'attribut est toujours entourée d'un `#[` et d'une terminaison correspondante `]`;

```php
<?php

#[Attribut, UnAutreAttribut(1,2)]
class Foo{
    //Ma classe marqué par les attributs Attribut et UnAutreAttribut (métadonnées)
}
```

> En savoir plus: La documentation officielle n'est pas encore à la hauteur sur l'usage des attributs PHP. Cet article peut être utile.

Sur les attributs

- Un attribut *peut être résolu* à une classe (obligatoire seulement si on veut une instance de cet attribut);
- Les arguments de l'attribut sont facultatifs, mais ils sont placés entre les parenthèses habituelles `()`;
- Les noms d'attributs et leurs arguments *sont* **résolus à une classe** et les **arguments sont transmis à son constructeur** lorsqu'une instance de l'attribut est demandée par l'intermédiaire de l'API de réflexion ;

Créer ses attributs

Un attribut PHP est une classe standard, déclarée elle-même avec l'attribut `#[Attribute]`

```php
//Déclaration de mon attribut
#[Attribute]
class FooAttribute{
    public function __construct(public ?string $param1 = null) {}
}
//Utilisation de mon attribut et son argument nommé
#[FooAttribute(param1: 'some value for this class')]
class Foo{}
```

Exemple d'usage d'attribut

Il est ensuite possible d'utiliser les attributs via l'API Reflection :

```php
$reflector = new \ReflectionClass(Foo::class);
$attributes = $reflector->getAttributes();
foreach($attributes as $attribute){

    $attribute->getName(); // "FooAttribute"
    $attribute->getArguments(); // ["some value for this class"]
    $instance = $attribute->newInstance(); // object(FooAttribute)#3 (1) {...}
    var_dump($instance->param1); // "some value for this class"

    //Faire quelque chose en fonction des attributs, valeurs des arguments, etc.
}
```

Exemple de l'attribut Route de Symfony

Inspecter le fichier `vendor/symfony/routing/Attribute/Route.php`. L'attribut Route est résolu à cette classe.

On peut voir tous les arguments nommés de cet attribut, que l'on peut et va utiliser. Le premier argument est `path` pour définir l'URL.

```php
#[\Attribute(\Attribute::IS_REPEATABLE | \Attribute::TARGET_CLASS | \Attribute::TARGET_METHOD)]
class Route
{
    private ?string $path = null;
    private array $localizedPaths = [];
    private array $methods;
    private array $schemes;

    /**
     * @param string|array<string,string>|null $path         The route path (i.e. "/user/login")
     * @param string|null                      $name         The route name (i.e. "app_user_login")
     * @param array<string|\Stringable>        $requirements Requirements for the route attributes, @see https://symfony.com/doc/current/routing.html#parameters-validation
     * @param array<string, mixed>             $options      Options for the route (i.e. ['prefix' => '/api'])
     * @param array<string, mixed>             $defaults     Default values for the route attributes and query parameters
     * @param string|null                      $host         The host for which this route should be active (i.e. "localhost")
     * @param string|string[]                  $methods      The list of HTTP methods allowed by this route
     * @param string|string[]                  $schemes      The list of schemes allowed by this route (i.e. "https")
     * @param string|null                      $condition    An expression that must evaluate to true for the route to be matched, @see https://symfony.com/doc/current/routing.html#matching-expressions
     * @param int|null                         $priority     The priority of the route if multiple ones are defined for the same path
     * @param string|null                      $locale       The locale accepted by the route
     * @param string|null                      $format       The format returned by the route (i.e. "json", "xml")
     * @param bool|null                        $utf8         Whether the route accepts UTF-8 in its parameters
     * @param bool|null                        $stateless    Whether the route is defined as stateless or stateful, @see https://symfony.com/doc/current/routing.html#stateless-routes
     * @param string|null                      $env          The env in which the route is defined (i.e. "dev", "test", "prod")
     */
      public function __construct(
        string|array|null $path = null,
        ...
    )
```

Routes avec paramètres d'URL et validation

- Il est possible de créer des routes *paramétrées*, *templatées* où l'URL contient un *pattern* à vérifier afin de **gérer une collection de routes avec une seule déclaration** ;
- Les **paramètres d'URL** s'écrivent entre accolades (`{}`). Par ex: `/route/{parametre}` ;
- On peut ajouter n'importe quelle règle de **validation** avec les arguments `requirements` et `condition` de l'attribut `Route` pour valider la route

> Validation = faire correspondre requête (URL, méthode HTTP, headers) à une route (traitement par contrôleur)

> En savoir plus sur le routing

Routes avec paramètres d'URL et validation

Tout paramètre d'URL est injecté comme argument du contrôleur.

```
    #[Route(
        "/hello/{firstName}/{lastName}",
        name: 'say_hello'
    )]
    public function sayHello(string $firstName, string $lastName): Response
    {
        return new Response("Hello {$firstName} {$lastName} !");
    }
```

Test: `curl -k localhost:8000/hello/foo/bar`

Routes avec paramètres d'URL et validation

Tout paramètre d'URL est injecté comme argument du contrôleur, **avec le bon type demandé** (conversion).

```
//On impose une contrainte sur les paramètres :
//Seules les valeurs 'foo' et 'bar' sont acceptées pour firstName et lastName respectivement.
//Seule l'URL /hello/foo/bar existe, sinon 404.
    #[Route(
        "/hello/{firstName}/{lastName}",
        name: 'say_hello',
        requirements: ['firstName' => "foo", 'lastName' => 'bar']
    )]
    public function sayHello(string $firstName, string $lastName): Response
    {
        //On peut récupérer la valeur du paramètre directement sur l'objet Request
        return new Response("Hello {$firstName} {$lastName} !");
    }
```

Routes avec paramètres d'URL et validation

Validation via une *regex* pour associer les bonnes URL aux bons contrôleurs.

```php
class BlogController extends AbstractController
{
    //Match /blog/0, /blog/1, etc. : URL d'une page d'articles (pagination)
    //L'expression régulière '\d+' match n'importe quel nombre entier
    #[Route('/blog/{page}', name: 'blog_list', requirements: ['page' => '\d+'])]
    public function list(int $page): Response
    {
        // ...
    }

    //Match /blog/my-first-post, /blog/some-slug : URL d'un article
    #[Route('/blog/{slug}', name: 'blog_show')]
    public function show($slug): Response
    {
        // ...
    }
}
```

Routes avec paramètres d'URL et validation

Une liste de *regex* universelles (nombres, slug ASCII, UUID, etc.) et communes est fournie via l'enum `Requirement`.

```php
class BlogController extends AbstractController
{
    //Même que précédent mais avec l'enum Requirement fourni par Symfony
    #[Route('/blog/{page}', name: 'blog_list', requirements: ['page' => Requirement::DIGITS])]
    public function list(int $page): Response
    {
        // ...
    }
}
```

Validation d'URL basée sur une logique *custom*

Utiliser l'argument `condition` si vous avez besoin de valider la requête avec *une logique custom*:

```
#[Route(
       '/contact',
       name: 'contact',
       condition: "context.getMethod() in ['GET', 'HEAD'] and request.headers.get('User-Agent') matches '/firefox/i'",
       // les expressions peuvent inclure des valeurs définies dans la config.
       // condition: "request.headers.get('User-Agent') matches '%app.allowed_browsers%'"
   )]
   public function contact(): Response
   {...}

    #[Route(
       '/posts/{id}',
       name: 'post_show',
       //l'expression peut accéder aux paramètres de route via la variable "params"
       condition: "params['id'] < 1000"
   )]
   public function showPost(int $id): Response
   {...}
```

La valeur de `condition` est une expression utilisant toute "expression language syntax" valide, une syntaxe texte fournie par le composant Expression Language.

> On retrouvera cette syntaxe dans le moteur de templates Twig.

Paramètres d'URL optionnels

Si paramètre est optionnel (par exemple, `/blog` et `/blog/1` existent), définir sa valeur par défaut dans l'argument du contrôleur.

```php
class BlogController extends AbstractController
{
    //Match /blog et /blog/1. /blog sera matchée avec $page égal à 1 par défaut
    #[Route('/blog/{page}', name: 'blog_list', requirements: ['page' => '\d+'])]
    public function list(int $page = 1): Response
    {
        // ...
    }
}
```

Définir la priorité des routes

Symfony (comme Express.js!) évalue les routes **dans l'ordre dans lequel elles sont déclarées** (ordre de déclaration des méthodes).

Au besoin, on peut définir la **priorité** d'une route avec l'argument `priority` (vaut par défaut `0`).
Une route avec une priorité plus élevée sera exécutée (son contrôleur) en premier.

```php
class BlogController extends AbstractController
{
    /**
     * Cette route, du fait de son pattern, intercepte l'URL /blog/list gérée par le contrôleur ci-dessous.
     */
    #[Route('/blog/{slug}', name: 'blog_show')]
    public function show(string $slug): Response
    {
        // ...
    }
    /**
     * Cette route ne pourrait jamais être empruntée sans la déclarer avant ou mettre sa priorité à une valeur plus élevée
     */
    #[Route('/blog/list', name: 'blog_list', priority: 2)]
    public function list(): Response
    {
        // ...
    }
}
```

Paramètres supplémentaires

Via l'argument `defaults` d'une Route, on peut définir des paramètres supplémentaires non inclus dans la configuration de la route. Pratique pour passer des arguments supplémentaires aux contrôleurs.

```
class BlogController extends AbstractController
{
    #[Route('/blog/{page}', name: 'blog_index', defaults: ['page' => 1, 'title' => 'Hello world!'])]
    public function index(int $page, string $title): Response
    {
        // ...
    }
```

Implémentation de l'interface uniforme : Autoriser des méthodes HTTP, en interdire d'autres

Par défaut, **une route accepte toutes les méthodes HTTP**. Vous pouvez restreindre les méthodes autorisées avec l'argument `methods` de l'attribut `Route`.

```php
//Utiliser la classe Route du module Annotation
namespace App\Controller;
use Symfony\Component\Routing\Annotation\Route;

# Cette route n'autorise que la méthode HTTP GET. Impossible de POST par exemple ici.
#[Route("hi", methods: ['GET'])]
public function sayHello(): Response{
    return new Response("Hello World!");
}
```

Implémentation de l'interface uniforme : autoriser des méthodes HTTP, en interdire d'autres

Vous pouvez facilement tester ces restrictions avec cURL:

```
# Exemples, observez le réponse HTTP
# Requete POST
curl -k -X POST https://127.0.0.1:8001/hi
# Requete GET
curl -k -X GET https://127.0.0.1:8001/hi
```

Préfixer les routes

Regrouper, organiser les routes et éviter les conflits de nom. Pour cela, ajouter l'attribut Route directement sur la classe Contrôleur.

```php
namespace App\Controller;

# On préfixe tous les contrôleurs (méthodes) par 'foo'
#[Route("/foo")]
class FooController extends AbstractController
{
    #[Route("/bar")]
    public function index(): Response{
        //URL routée : /foo/bar
    }
}
```

Manipuler la requête HTTP

Chaque méthode du contrôleur se voit injecté un objet de type `Symfony\Component\HttpFoundation\Request` contenant toutes les données sur la requête entrante (crée à partir de l'analyse des *variables superglobales* `$_SERVER`, `$_REQUEST`, `$_GET`, `$_POST`, etc.)

> Cette injection dans la méthode est possible grâce au Service Container, discuté au module 7.

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

 #[Route("/some-url/{param1}")]
 //Je peux injecter l'objet Request dans le contrôleur
 public function mirror(Request $request): Response
 {...}
```

Mapping automatique du contenu de la requête : query parameters

Depuis Symfony 6, il est possible de mapper/valider automatiquement le payload (contenu du body) ou les paramètres de requête (?) aux arguments du contrôleur avec l'attribut `MapQueryParameter`.

Par exemple l'URL `https://example.com/dashboard?firstName=John&lastName=Smith&age=27` est validée par la route dashboard et les paramètres d'URL sont extraites et mappées automatiquement aux arguments `$firstName`, `$lastName` et `$age`.

```php
use Symfony\Component\HttpKernel\Attribute\MapQueryParameter;

#[Route('/dashboard', name: 'dashboard')]
public function dashboard(
    #[MapQueryParameter] string $firstName,
    #[MapQueryParameter] string $lastName,
    #[MapQueryParameter] int $age,
): Response
{
    // Code du contrôleur
}
```

Mapping automatique du contenu de la requête : query parameters

On peut également y ajouter des règles de validation avec l'argument `filter` de l'attribut `MapQueryParameter`

```php
use Symfony\Component\HttpKernel\Attribute\MapQueryParameter;

public function dashboard(
    #[MapQueryParameter(filter: \FILTER_VALIDATE_REGEXP, options: ['regexp' => '/^\w+$/'])] string $firstName,
    #[MapQueryParameter] string $lastName,
    #[MapQueryParameter(filter: \FILTER_VALIDATE_INT)] int $age,
): Response
{
    // ...
}
```

Mapping automatique du contenu de la requête : body

Mapping du body de la requête (*payload*). Imaginons qu'une route accepte la méthode `POST` et demande les données suivantes :

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 28
}
```

Supposons qu'on possède une classe `UserDto` avec ces attributs, symfony va instancier pour nous un objet de ce type dont les champs sont initialisés avec le contenu du payload.

```
#[Route('/dashboard', name: 'dashboard')]
public function dashboard(
    #[MapRequestPayload] UserDto $userDto
): Response
{
    // ...
}
```

Manipuler la requête HTTP

L'objet Request contient toutes les informations sur la requête du client (méthode, headers, body, query, etc.).

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

//...Dans une classe Contrôleur

 #[Route("/some-url/{param1}")]
 public function mirror(Request $request, string $param1): Response
 {
    //Récupérer les données envoyées en query d'URL (derrière le ?)
    $request->query->get('foo');
    //Récupérer le paramètre d'URL
    $request->get('param1');
    //Récupérer les données dans le body (cas d'une requête POST)
    $request->request;
    etc...
 }
```

Test : `https://localhost:8000/some-url/me?foo=bar`

Request Object

Tout contrôleur a accès à l'objet `Request` qui fournit toutes les informations utiles sur la requête à traiter. Pour y avoir accès, il suffit de l'injecter dans le contrôleur avec le type hinting `Request`.

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

//Injection de l'objet Request dans le contrôleur
public function index(Request $request): Response
{
    $request->isXmlHttpRequest(); // is it an Ajax request?

    $request->getPreferredLanguage(['en', 'fr']);

    // retrieves GET and POST variables respectively
    $request->query->get('page');
    $request->getPayload()->get('page');

    // retrieves SERVER variables
    $request->server->get('HTTP_HOST');

    // retrieves an instance of UploadedFile identified by foo
    $request->files->get('foo');

    // retrieves a COOKIE value
    $request->cookies->get('PHPSESSID');

    // retrieves an HTTP request header, with normalized, lowercase keys
    $request->headers->get('host');
    $request->headers->get('content-type');
}
```

Response Object

```php
use Symfony\Component\HttpFoundation\Response;

// Créer une simple réponse avec un code status 200 (par défaut)
$response = new Response('Hello '.$name, Response::HTTP_OK);

// Créer une réponse de type feuille de style avec un code status 200
$response = new Response('<style> ... </style>');
$response->headers->set('Content-Type', 'text/css');
```

Retourner un document JSON :

```php
public function index(): JsonResponse
{
    // Retourne '{"username":"jane.doe"}' et se charge de définir le bon header Content-Type (application/json)
    return $this->json(['username' => 'jane.doe']);
}
```

> En savoir plus

Gérer les erreurs et les pages 404

Si aucune route existe pour la ressource demandée par le client, le site doit retourner une page web 404. Symfony met à disposition une exception spéciale : `NotFoundHttpException`. Une fois lancée, **cette exception déclenche l'envoi d'une réponse HTTP 404**. Une page d'erreur est affichée au client.

```php
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

// ...
public function index(): Response
{
    // Chercher quelque chose en base de données
    $product = ...;
    if (!$product) {
        throw $this->createNotFoundException('The product does not exist');
        // the above is just a shortcut for:
        // throw new NotFoundHttpException('The product does not exist');
    }
    return $this->render(/* ... */);
}
```

> Nous verrons dans le module 8 sur le templating avec Twig comment customiser les pages d'erreur.

Débuger le routing

- L'avantage d'utiliser la configuration par Attribut PHP c'est que **la configuration du routage est au plus proche (physiquement) du code en charge de répondre à la requête**.
- L'inconvénient, c'est que votre configuration de routage est disséminée dans vos classes *Contrôleurs*.

La console `symfony` vous met à disposition des **outils pour inspecter facilement toutes les routes de votre application**.

```
# Afficher toutes les routes de l'application
php bin/console debug:router
# Afficher toutes les infos sur la route (path, params, regex, classe contrôleur en charge, etc.)
php bin/console debug:router <nom route>
```

> Vous pouvez également générer des URLs basés sur les noms des routes, lister les routes et les manipuler directement dans le code et bien d'autres choses encore !

Être productif : Générer des contrôleurs via la console

Pour accélérer le développement, on peut générer *le boilerplate code* des contrôleurs directement depuis la `console` :

```
php bin/console make:controller NomDuControleur
```

Génère automatiquement :

- Une fichier `NomDuControleurController.php` dans `src/Controller` ;
- Une Classe `NomDuControleurController` qui étend `AbstractController` ;
- Un template Twig `nom_du_controleur/index.html.twig` dans `templates` .

Être productif : créer ses propres commandes et étendre la console

Le programme `bin/console` est basé sur le composant Console, vous pouvez créer vos propres commandes pour créer vos outils personnalisés.

Références

- Documentation officielle Symfony, commencer à l'explorer par la section *Getting Started*.
- Create your First Page in Symfony, un guide officiel pour créer sa première page avec Symfony
- **Routing**, documentation officielle sur la mise en place du routing dans Symfony;
- **Controller**, tout savoir sur les Contrôleurs Symfony (AbstractController, Redirection, Injection de Services (injection de dépendances), etc.)
- Enum Requirements, la liste des contraintes de validation accessibles par défaut sur les routes (paramètre `requirements`)

Exercices

Réaliser les exercices fournis avec le module 6.

TP Fil Rouge - Système de billetterie

Poursuivre le TP file rouge (Partie 1), correspondant au module couvert.