

Symfony

Routage et services

Maxime Puy

31 janvier 2024

1 Routage des pages

Routage des contrôleurs

Introduction au routage Symfony

- Le traitement des requêtes implique l'appel d'une action de contrôleur pour générer des réponses.
- Le routage sert à définir quelle action exécuter pour chaque URL.
- Les URL conviviales pour le référencement/navigabilité sont obtenues grâce à un routage efficace.

Ressources

- <https://symfony.com/doc/5.4/routing.html>

Création de routes avec des annotations

- ❑ Symfony recommande l'utilisation d'attributs PHP pour une définition concise des routes.
 - ▷ Seulement en PHP8 => Pas dispo à l'IUT :(
 - ▷ Autre possibilité : les annotations

```

1 // src/Controller/BlogController.php
2 use Symfony\Component\Routing\Annotation\Route;
3
4 /**
5  * @Route('/blog', name: 'blog_list')
6  */
7 public function list(): Response
8 {
9     // ...
10 }

```

- ❑ Les attributs/annotations permettent de configurer directement les routes au sein des classes de contrôleurs.

Création de routes dans des fichiers YAML, XML ou PHP

- Les routes sont aussi définies dans des fichiers **YAML**, XML ou PHP.
- Avantage : toutes les routes dans un même fichier.

```
1 # config/routes.yaml
2 blog_list:
3     path: /blog
4     controller: App\Controller\BlogController::list
```

Correspondance des méthodes HTTP

- Possible de restreindre les méthodes HTTP pour les routes en utilisant l'option `methods`.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/api/posts", methods={"GET", "HEAD"})
4  */
5 public function show(): Response
6 {
7     // ... return a JSON response with the post
8 }
9
10 /**
11  * @Route("/api/posts", methods={"PUT"})
12  */
13 public function edit(): Response
14 {
15     // ... edit a post
16 }
```

Correspondance des méthodes HTTP

- Possible de restreindre les méthodes HTTP pour les routes en utilisant l'option `methods`.
- Exemple : deux méthodes du controller pour la même route !

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/api/posts", methods={"GET", "HEAD"})
4  */
5 public function show(): Response
6 {
7     // ... return a JSON response with the post
8 }
9
10 /**
11  * @Route("/api/posts", methods={"PUT"})
12  */
13 public function edit(): Response
14 {
15     // ... edit a post
16 }
```

Correspondance des expressions

- Possibilité de matcher des routes avec des expressions complexes avec l'option `condition`.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route(
4  *     "/contact",
5  *     name="contact",
6  *     condition="context.getMethod() in ['GET', 'HEAD'] \
7  *         and request.headers.get('User-Agent') matches '/firefox/i'"
8  * )
9  *
10 * expressions can also include configuration parameters:
11 * condition: "request.headers.get('User-Agent') matches '%app.allowed_browsers%'"
12 */
13 public function contact(): Response
14 {
15     // ...
16 }
```

Débogage des routes

- Symfony propose des commandes pour déboguer les routes :
 - ▷ `debug:router` liste toutes les routes.
 - ▷ `router:match` identifie quelle route correspond à une URL donnée.

```
1 symfony console debug:router  
2 symfony console router:match /blog/post/8
```

Paramètres de route et validation

- Possibilité de matcher des paramètres dans les routes et la validation de leur type avec l'option `requirements`.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
4  */
5 public function list(int $page): Response
6 {
7     // ...
8 }
```

Paramètres de route et validation

- Possibilité de matcher des paramètres dans les routes et la validation de leur type avec l'option `requirements`.
- Exemple : validation du paramètre `page` en tant que chiffres.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
4  */
5 public function list(int $page): Response
6 {
7     // ...
8 }
```

Paramètres optionnels et valeurs par défaut

- Possibilité de rendre les paramètres optionnels avec des valeurs par défaut.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
4  */
5 public function list(int $page = 1): Response
6 {
7     // ...
8 }
```

- Il est possible d'avoir plus d'un paramètre optionnel, mais dans ce cas, tout ce qui les paramètres après un paramètre optionnel sont optionnels.

Paramètres optionnels et valeurs par défaut

- Possibilité de rendre les paramètres optionnels avec des valeurs par défaut.
- Exemple : rendre `page` facultatif avec une valeur par défaut de 1.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{page}", name="blog_list", requirements={"page"="\d+"})
4  */
5 public function list(int $page = 1): Response
6 {
7     // ...
8 }
```

- Il est possible d'avoir plus d'un paramètre optionnel, mais dans ce cas, tout ce qui les paramètres après un paramètre optionnel sont optionnels.

Priorité entre les routes

- Par défaut, les routes sont matchées dans l'ordre de définition, on peut les prioriser (0 = moins prioritaire).

```
1 // src/Controller/BlogController.php
2 /**
3  *
4  * @Route("/blog/{slug}", name="blog_show")
5  */
6 public function show(string $slug): Response
7 {
8     // ...
9 }
10
11 /**
12  *
13  * @Route("/blog/list", name="blog_list", priority=2)
14  */
15 public function list(): Response
16 {
17     // ...
18 }
```

Priorité entre les routes

- Par défaut, les routes sont matchées dans l'ordre de définition, on peut les prioriser (0 = moins prioritaire).

```
1 // src/Controller/BlogController.php
2 /**
3  * This route has a greedy pattern and is defined first.
4  * @Route("/blog/{slug}", name="blog_show")
5  */
6 public function show(string $slug): Response
7 {
8     // ...
9 }
10
11 /**
12  * This route could not be matched without defining a higher priority than 0.
13  * @Route("/blog/list", name="blog_list", priority=2)
14  */
15 public function list(): Response
16 {
17     // ...
18 }
```

Conversion des paramètres

- Possibilité de convertir les types des paramètres
- Exemple : la conversion d'un slug en un objet `BlogPost`.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{slug}", name="blog_show")
4  */
5 public function show(BlogPost $post): Response
6 {
7     // $post est l'objet dont le slug correspond au paramètre de routage
8     // ...
9 }
```

Conversion des paramètres

- Possibilité de convertir les types des paramètres
- Exemple : la conversion d'un slug en un objet `BlogPost`.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{slug}", name="blog_show")
4  */
5 public function show(BlogPost $post): Response
6 {
7     // $post est l'objet dont le slug correspond au paramètre de routage
8     // ...
9 }
```

- Si le controller mentionne un paramètre dont le type est une entité (type de donnée mise en BDD), `Symfony` fera une requête en BDD utilisant les paramètres de la route (ici le champ `slug` de la table).
- Erreur 404 si l'entité n'est pas trouvée.

Paramètres supplémentaires

- Exemple : un constructeur générique personnalisé avec un paramètre qui n'est pas dans la route.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{page}",
4  *       name="blog_index",
5  *       defaults={"page": 1, configFile: "config.txt"})
6  */
7
8 public function index(int $page, string $configFile): Response
9 {
10     // ...
11 }
```

Groupes et préfixes de route

- Les groupes de routes partagent des options communes.

```
1 /**
2  * @Route("/blog", requirements={"_locale": "en/es/fr"}, name="blog_")
3  */
4 class BlogController extends AbstractController
5 {
6     /**
7      * @Route("/{_locale}", name="index")
8      */
9     public function index(): Response {}
10
11     /**
12      * @Route("/{_locale}/posts/{slug}", name="show")
13      */
14     public function show(string $slug): Response {}
15 }
```

Groupes et préfixes de route

- ❑ Les groupes de routes partagent des options communes.
- ❑ Exemple : requirements commun à toutes les routes, prefix “/blog” et mangling du nom.

```
1 /**
2  * @Route("/blog", requirements={"_locale": "en/es/fr"}, name="blog_")
3  */
4 class BlogController extends AbstractController
5 {
6     /**
7      * @Route("/{_locale}", name="index")
8      */
9     public function index(): Response {}
10
11     /**
12      * @Route("/{_locale}/posts/{slug}", name="show")
13      */
14     public function show(string $slug): Response {}
15 }
```

Groupes et préfixes de route

- Les groupes de routes partagent des options communes.
- Exemple : requirements commun à toutes les routes, prefix “/blog” et mangling du nom.
- Exemple de route pour la méthode `show` : `/blog/fr/posts/bonjour`

```

1 /**
2  * @Route("/blog", requirements={"_locale": "en/es/fr"}, name="blog_")
3  */
4 class BlogController extends AbstractController
5 {
6     /**
7      * @Route("/{_locale}", name="index")
8      */
9     public function index(): Response {}
10
11     /**
12      * @Route("/{_locale}/posts/{slug}", name="show")
13      */
14     public function show(string $slug): Response {}
15 }

```

Obtenir le nom et les paramètres de la route

- Les informations de la route, comme le nom et les paramètres, sont stockées dans la classe `Request`.

```
1 // src/Controller/BlogController.php
2 /**
3  * @Route("/blog/{page}", name="index")
4  */
5 public function list(Request $request): Response
6 {
7     $routeName = $request->attributes->get('_route');
8     $routeParameters = $request->attributes->get('_route_params');
9 }
```

Redirections au niveau des routes

- Possibilité de gérer les redirections (302, etc) au niveau des routes.

```
1 # config/routes.yaml
2 legacy_doc:
3     path: /legacy/doc
4     controller: Symfony\Bundle\FrameworkBundle\Controller\RedirectController
5     defaults:
6         # this value can be another route
7         route: 'doc_page'
8         # or an absolute path or an absolute URL
9         path: 'https://legacy.example.com/doc'
10
11     # sets if redirection is permanent
12     permanent: true
```

- Pas avec des annotations car controller spécifique à la redirection.

Redirection au niveau des contrôleur

```
1 // src/Controller/BlogController.php
2 public function index(): RedirectResponse // Attention au type de retour!
3 {
4     // redirects to the "homepage" route
5     return $this->redirectToRoute('homepage');
6
7     // does a permanent HTTP 301 redirect
8     return $this->redirectToRoute('homepage', [], Response::HTTP_MOVED_PERMANENTLY);
9
10    // redirect to a route with parameters
11    return $this->redirectToRoute('app_lucky_number', ['max' => 10]);
12
13    // redirects to a route and maintains the original query string parameters
14    return $this->redirectToRoute('blog_show', $request->query->all());
15
16    // redirects to the current route (e.g. for Post/Redirect/Get pattern):
17    return $this->redirectToRoute($request->attributes->get('_route'));
18
19    // redirects externally
20    return $this->redirect('http://symfony.com/doc');
21 }
```

Routes localisées (i18n)

- Pour les applications traduites, chaque route peut définir une URL différente pour chaque locale.

```
1 /**
2  * @Route({
3  *     "en": "/about-us",
4  *     "fr": "/a-propos"
5  * }, name="about_us")
6  */
7 public function about(): Response
8 {
9     // ...
10 }
```

Routes localisées (i18n)

- Une bonne utilisation pour les préfixes de route !

```
1 # config/routes/annotations.yaml
2 controllers:
3     resource: '../src/Controller/'
4     type: annotation
5     prefix:
6         en: '' # pas de préfixe pour la langue principale
7         fr: '/fr'
```

Génération d'URLs

- Les systèmes de routage Symfony permettent de générer des URLs à partir des noms de route et des paramètres.
- Utilisez `generateUrl()` pour générer des URLs dans les contrôleurs.

```
1 // src/Controller/BlogController.php
2 $signupPage = $this->generateUrl('sign_up');
3 $userProfilePage = $this->generateUrl(
4     'user_profile',
5     ['username' => $user->getUserIdentifier()]
6 );
```

- Utile pour générer des liens vers les autres pages!
 - ▶ Plus de détails dans le CM sur Twig

Vérification de l'existence d'une route

- ❑ Évitez d'utiliser `getRouteCollection()` pour vérifier l'existence d'une route, car cela régénère le cache de routage.
- ❑ Essayez de générer l'URL et capturez `RouteNotFoundException` si la route n'existe pas.

```
1 use Symfony\Component\Routing\Exception\RouteNotFoundException;
2
3 try {
4     $url = $this->router->generate($routeName, $routeParameters);
5 } catch (RouteNotFoundException $e) {
6     // La route n'est pas définie...
```

[RAPPEL !] Débogage des routes

- Symfony propose des commandes pour déboguer les routes :
 - ▷ `debug:router` liste toutes les routes.
 - ▷ `router:match` identifie quelle route correspond à une URL donnée.

```
1 symfony console debug:router  
2 symfony console router:match /blog/post/8
```

Services et Injections de Dépendances

Introduction au Conteneur de Services Symfony

- **Concept de Service** : Objets PHP effectuant des tâches spécifiques dans votre application.
- **Le Conteneur de Services** : Centralise la gestion des services, facilite la construction et la maintenance des objets.
- **Architecture et Performance** : Favorise une architecture robuste et offre des performances optimisées.

Création et Configuration de Services

- **Organisation du Code en Services** : Transformez vos fonctionnalités en services pour une meilleure réutilisabilité.
- **Exemple de Service** : `MessageGenerator` pour générer des messages aléatoires.

```
1 // src/Service/MessageGenerator.php
2 class MessageGenerator {
3     public function getHappyMessage(): string {
4         // Retourne un message aléatoire
5     }
6 }
```

Accéder et Utiliser les Services

- **Démarrage avec Symfony** : Le conteneur contient plusieurs services prêts à l'emploi dès le lancement de l'application.
- **Demande de Service** : Utilisez le `type-hinting` dans les contrôleurs pour accéder aux services.

```
1 // src/Controller/BlogController.php
2 public function list(LoggerInterface $logger): Response {
3     $logger->info('Utilisation d\'un service Symfony!');
4     // ...
5 }
```

- L'appel des services via le `type-hinting` se nomme `autowiring` en Symfony

Configuration pour l'autowiring

- **Configuration dans services.yaml** : Paramètres par défaut pour l'autowiring et l'autoconfiguration.
- **Simplification de la déclaration** : Les classes dans `src/` deviennent automatiquement des services.

```
1 # config/services.yaml
2 services:
3     _defaults:
4         autowire: true
5         autoconfigure: true
6     App\:
7         resource: '../src/'
8         // Exclusions et autres paramètres
```

Types Autowirable

- Pour découvrir quels types sont accessibles avec l'autowiring :

```
1 $ symfony console debug:autowiring
```

```
2 Autowirable Types
```

```
3 =====
```

```
4 The following classes & interfaces can be used as type-hints when autowiring:
```

```
5 App\Kernel (kernel)
```

```
6 Interface for annotation readers.
```

```
7 Doctrine\Common\Annotations\Reader (annotations.cached_reader)
```

```
8 Doctrine\Common\Persistence\ManagerRegistry (doctrine)
```

```
9 [...]
```

```
0 App\Controller\BlogController
```

```
1 App\Repository\BlogArticleRepository
```

```
2 App\Service\MessageGenerator
```

Injection de Dépendances

□ Type-hinting au niveau du controller

```

1 // src/Service/SiteUpdateManager.php
2 namespace App\Service;
3
4 use App\Service\MessageGenerator;
5 use Symfony\Component\Mailer\MailerInterface;
6
7 class SiteUpdateManager
8 {
9     private $messageGenerator;
10    private $mailer;
11
12    public function __construct(
13        MessageGenerator $messageGenerator,
14        MailerInterface $mailer,
15    ) {
16        $this->messageGenerator = $messageGenerator; // Remove in PHP8
17        $this->mailer = $mailer; // Remove in PHP8
18    }
19 }

```

Paramètres pour les services

- ❑ Rappel : Autowiring = Symfony créer les objets des services lui-même
- ❑ Quid si mon service prend des paramètres ?

```
1 // src/Service/SiteUpdateManager.php
2 class SiteUpdateManager
3 {
4     public function __construct(
5         MessageGenerator $messageGenerator,
6         MailerInterface $mailer,
7     ) {
8         // ...
9     }
10 }

1 // src/Controller/BlogController.php
2 public function list(SiteUpdateManager $manager): Response {
3     // ...
4 }
```

Paramètres pour les services

- ❑ Rappel : Autowiring = Symfony créer les objets des services lui-même
- ❑ Quid si mon service prend des paramètres ?

```

1 // src/Service/SiteUpdateManager.php
2 class SiteUpdateManager
3 {
4     public function __construct(
5         MessageGenerator $messageGenerator,
6         MailerInterface $mailer,
7     ) {
8         // ...
9     }
10 }

1 // src/Controller/BlogController.php
2 public function list(SiteUpdateManager $manager): Response {
3     // ...
4 }

```

- ❑ Si paramètres “autowirables”, OK !

Paramètres pour les services

- Si paramètres pas “autowirables” ??

```

1 // src/Service/SiteUpdateManager.php
2 class SiteUpdateManager
3 {
4     public function __construct(
5         MailerInterface $mailer,
6         string $adminEmail // D'où Symfony sort l'adresse email ?
7     ) {
8         // ...
9     }
10 }

1 // src/Controller/BlogController.php
2 public function list(SiteUpdateManager $manager): Response {
3     // ...
4 }

```

Configuration Manuelle des Arguments

□ Configuration Explicite :

- ▷ Définissez explicitement les arguments non autowirables dans `config/services.yaml`.
- ▷ Exemple pour `SiteUpdateManager` :

```
1 # config/services.yaml
2 services:
3     App\Service\SiteUpdateManager:
4         arguments:
5             $adminEmail: 'manager@example.com'
```

□ Gestion des Erreurs et Flexibilité :

- ▷ En cas de modification du nom d'argument (ex : `$mainEmail`), Symfony lève une exception claire.
- ▷ Cela garantit une configuration robuste et évite les erreurs silencieuses.

Sélection d'un Service Spécifique

- ❑ Choix parmi plusieurs implémentations de la même interface.
- ❑ Exemple : Utilisation d'un logger particulier pour `LoggerInterface`.

```
1 // src/Service/MessageGenerator.php
2 use Psr\Log\LoggerInterface;
3 class MessageGenerator
4 {
5     private $logger;
6     public function __construct(LoggerInterface $logger)
7     {
8         $this->logger = $logger;
9     }
10 }
```

```
1 # config/services.yaml
2 services:
3     App\Service\MessageGenerator:
4         arguments:
5             $logger: '@monolog.logger.request'
```

Binding des Paramètres

- Possibilité de binder les paramètres des services par nom/type :

```
1 # config/services.yaml
2 services:
3     _defaults:
4         bind:
5             # valeur pour tout paramètre nommé $adminEmail
6             $adminEmail: 'manager@example.com'
7
8             # type pour tout paramètre implémentant LoggerInterface
9             Psr\Log\LoggerInterface: '@monolog.logger.request'
10
11             # ou les deux
12             string $adminEmail: 'manager@example.com'
13             Psr\Log\LoggerInterface $requestLogger: '@monolog.logger.request'
```

Paramètres Abstraits

- Comme les classes abstraites !
- L'argument doit explicitement être passé à l'appel.

```
1 # config/services.yaml
2 services:
3     App\Service\MyService:
4         arguments:
5             $rootNamespace: !abstract 'should be defined by Pass'
```

- **RuntimeException** : Argument “\$rootNamespace” of service “App\Service\MyService” is abstract : should be defined by Pass.

Linting des Services

- Possibilité de valider la capacité de Symfony à instancer les services via la configuration :

```
1 $ symfony console lint:container
2 In DefinitionErrorExceptionPass.php line 49:
3 Cannot autowire service "App\Service\TestService": argument "$a" of method
4 "__construct()" is type-hinted "int", you should configure its value
5 explicitly.
```