

Symfony

Routage et services

Maxime Puy

22 mars 2025



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
Permission is explicitly granted to copy, distribute and/or modify this document
for educational purposes under the terms of the CC BY-NC-SA license.

Plan

1 Routage des pages

2 Services et Injection de Dépendances

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 URLs optimisées pour le référencement/navigabilité sont obtenues grâce à un routage efficace.

Ressources

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

Création de routes avec des annotations

- Les bonnes pratiques Symfony recommande l'utilisation des attributs PHP pour une définition concise des routes.
 - ▷ Seulement disponible en PHP8
 - ▷ Autre possibilité : les annotations, routes.yaml, fichier xml

```
1 // src/Controller/BlogController.php
2 use Symfony\Component\Routing\Attribute\Route;
3 #[Route('/blog', name: 'blog_list')]
4 public function list(): Response
5 {
6     // ...
7 }
```

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

Création de routes dans des fichiers YAML, XML ou PHP

- Les routes peuvent aussi être 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 #[Route('/api/posts', name: 'api_posts', methods: ['GET', 'HEAD'])]
3 public function show(): Response
4 {
5     // ... return a JSON response with the post
6 }
7
8 //Can be same path, but name may be different
9 #[Route('/api/posts', name: 'api_posts_edit', methods: ['POST'])]
10 public function edit(): Response
11 {
12     // ... edit a post
13 }
```


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 #[Route('/api/posts', name: 'api_posts', methods: ['GET', 'HEAD'])]
3 public function show(): Response
4 {
5     // ... return a JSON response with the post
6 }
7
8 //Can be same path, but name may be different
9 #[Route('/api/posts', name: 'api_posts_edit', methods: ['POST'])]
10 public function edit(): Response
11 {
12     // ... edit a post
13 }
```

Correspondance des expressions

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

```
1 // src/Controller/BlogController.php
2 #[Route(
3     '/contact',
4     name: 'contact',
5     condition: "context.getMethod() in ['GET', 'HEAD']
6     and request.headers.get('User-Agent') matches '/firefox/i'",
7     // expressions can also include config parameters:
8     // condition: "request.headers.get('User-Agent') matches '%app.allowed_brow
9 )]
0 public function contact(): Response
1 {
2     // ...
3 }
```

Conditions de routing approfondies

- Fonctionne aussi avec de l'AJAX :

```
1 // src/Controller/BlogController.php
2 #[Route(
3     '/contact/ajax',
4     name: 'contact_ajax',
5     options: ['expose' => true], // Needed to expose as an AJAX route
6                                     // (using a specific bundle)
7     condition: "request.isXmlHttpRequest()
8     and service('auth_checker').isAuthorizedToContact()",
9     // Check for AJAX request and then using a custom service
10    // for specific verifications using AsRoutingConditionalServ
11 )]
12 public function contact(): Response
13 {
14     // ...
15 }
```

Conditions de routing approfondies

- Et le service en question :

```
1 #[AsRoutingConditionService(alias: 'auth_checker')]
2 class AuthChecker
3 {
4     public function check(Request $request): bool
5     {
6         // Some custom verification here
7     }
8 }
```

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 #[Route(
3     '/blog/{page}',
4     name: 'blog_list',
5     requirements: ['page' => '\d+'],
6     methods: ['GET']
7 )]
8 public function list(int $page): Response
9 {
10     // ...
11 }
```

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 #[Route(
3     '/blog/{page}',
4     name: 'blog_list',
5     requirements: ['page' => '\d+'],
6     methods: ['GET']
7 )]
8 public function list(int $page): Response
9 {
10     // ...
11 }
```

Paramètres optionnels et valeurs par défaut

- Possibilité de rendre les paramètres optionnels.

```
1 // src/Controller/BlogController.php
2 #[Route(
3     '/blog/{page}',
4     name: 'blog_list',
5     requirements: ['page' => '\d+'],
6     defaults: ['page' => 1],
7     methods: ['GET']
8 )]
9 public function list(int $page = 1): Response
10 {
11     // ...
12 }
```

- Il est possible d'avoir plus d'un paramètre optionnel, mais alors, tous les paramètres après un paramètre optionnel sont optionnels.

Paramètres optionnels et valeurs par défaut

- Possibilité de rendre les paramètres optionnels.
- Exemple : rendre `page` facultatif avec une valeur par défaut de 1.
- On peut soit l'indiquer dans l'attribute, soit en paramètre.

```

1 // src/Controller/BlogController.php
2 #[Route(
3     '/blog/{page}',
4     name: 'blog_list',
5     requirements: ['page' => '\d+'],
6     defaults: ['page' => 1],
7     methods: ['GET']
8 )]
9 public function list(int $page = 1): Response
10 {
11     // ...
12 }

```

- Il est possible d'avoir plus d'un paramètre optionnel, mais alors, tous 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 #[Route('/blog/{slug}', name: 'blog_show')]
3 public function show(string $slug): Response
4 {
5     // Default action
6 }
7
8 #[Route('/blog/special/{slug}', name: 'blog_special_show', priority: 2)]
9 public function specialShow(string $slug): Response
10 {
11     // Special action
12 }
```

- **Critique** : Dans un respect des bonnes pratiques, les priorités sont à éviter. En général, cela traduit plutôt un mauvais nommage/découpage des routes.

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 // This route has a greedy pattern and is defined first.
3 #[Route('/blog/{slug}', name: 'blog_show')]
4 public function show(string $slug): Response
5 {
6     // Default action
7 }
8
9 // This route could not be matched without defining a higher priority than 0.
10 #[Route('/blog/special/{slug}', name: 'blog_special_show', priority: 2)]
11 public function specialShow(string $slug): Response
12 {
13     // Special action
14 }
```

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 #[Route('/blog/{slug}', name:'blog_show')]
3 public function show(BlogPost $post): Response
4 {
5     // $post est l'objet dont le slug correspond au paramètre de routage
6     // ...
7 }
```

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 #[Route('/blog/{slug}', name:'blog_show')]
3 public function show(BlogPost $post): Response
4 {
5     // $post est l'objet dont le slug correspond au paramètre de routage
6     // ...
7 }
```

- 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 la propriété `slug` de l'entité `BlogPost`).
- Erreur 404 si l'entité n'est pas trouvée.

Routage avec Alias

- Les alias de route permettent d'avoir plusieurs noms pour une même route.

```
1 # config/routes.yaml
2 new_route_name:
3     alias: original_route_name
```

Déprécier les Alias de Route

- Les alias de route peuvent être dépréciés pour indiquer qu'ils ne doivent plus être utilisés.
- Ajoutez la section `deprecated` pour déprécier une route avec un message personnalisé.

```
1 # config/routes.yaml
2 new_route_name:
3   alias: original_route_name
4   deprecated:
5     package: 'acme/package'
6     version: '1.2'
7     message: 'Le chemin "%alias_id%" est déprécié. Ne l'utilisez plus.'
```

Groupes et préfixes de route

- Les groupes de routes partagent des options communes.

```
1 #[Route('/blog', name: 'blog_', requirements:['_locale' => 'en|es|fr'])]
2 class BlogController extends AbstractController
3 {
4     #[Route('/{_locale}', name: 'index')]
5     public function index(): Response {}
6
7     #[Route('/{_locale}/posts/{slug}', name: 'show')]
8     public function show(string $slug): Response {}
9 }
```


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 #[Route('/blog', name: 'blog_', requirements:['_locale' => 'en|es|fr'])]
2 class BlogController extends AbstractController
3 {
4     #[Route('/{_locale}', name: 'index')]
5     public function index(): Response {}
6
7     #[Route('/{_locale}/posts/{slug}', name: 'show')]
8     public function show(string $slug): Response {}
9 }
```

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 #[Route('/blog', name: 'blog_', requirements:['_locale' => 'en|es|fr'])]
2 class BlogController extends AbstractController
3 {
4     #[Route('/{_locale}', name: 'index')]
5     public function index(): Response {}
6
7     #[Route('/{_locale}/posts/{slug}', name: 'show')]
8     public function show(string $slug): Response {}
9 }
```

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 #[Route('/blog/{page}', name: 'index')]
3 public function list(Request $request, int page): Response
4 {
5     $routeName = $request->attributes->get('_route');
6     $routeParameters = $request->attributes->get('_route_params');
7 }
```

- En principe pas utile car le front controller fait le travail de valider la route et les paramètres.

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
```

- Sans appeler de contrôleur ...

Redirection au niveau des contrôleur

```
1 // src/Controller/BlogController.php
2 public function index(Request $request): RedirectResponse
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 }
```

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 );
```

- Beaucoup de méthodes très utiles fournies par le composant `UrlGeneratorInterface` de Symfony
 - ▶ 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

Rappel : Type hinting en PHP

- PHP est un langage dynamique, pas besoin de déclarer les types des paramètres.

```
1 <?php
2 function add($x, $y)
3 {
4     return $x + $y;
5 }
6 $result = add(1,2);
7 echo $result; // 3
```

Rappel : Type hinting en PHP

- Génial, ça marche aussi avec les flottants !

```
1 <?php
2 function add($x, $y)
3 {
4     return $x + $y;
5 }
6
6 $result = add(1.0,2.5);
7 echo $result; // 3.5
```

Rappel : Type hinting en PHP

- Et même avec des strings !

```
1 <?php
2 function add($x, $y)
3 {
4     return $x + $y;
5 }
6 $result = add('Hi', 'There');
7 echo $result;
```

Rappel : Type hinting en PHP

- Et même avec des strings !

```
1 <?php
2 function add($x, $y)
3 {
4     return $x + $y;
5 }
6 $result = add('Hi', 'There');
7 echo $result;
```

- Ah toute compte fait non ...

```
1 Fatal error: Uncaught TypeError: Unsupported operand types: string + string in ...
```

Rappel : Type hinting en PHP

- Tout compte fait c'est bien de mettre les types !

```
1 <?php
2 function add(int $x, int $y)
3 {
4     return $x + $y;
5 }
6
6 $result = add(1,2);
7 echo $result; // 3
```

Rappel : Type hinting en PHP

- Bien sur, ça marche aussi pour le type de retour.

```
1 <?php
2 function add(int $x, int $y): int
3 {
4     return $x + $y;
5 }
6
6 $result = add(1,2);
7 echo $result; // 3
```

Rappel : Type hinting en PHP

- Bien sur, ça marche aussi pour le type de retour.

```
1 <?php
2 function add(int $x, int $y): int
3 {
4     return $x + $y;
5 }
6
6 $result = add(1,2);
7 echo $result; // 3
```

- Utilisez le type hinting, votre “vous” du futur vous dira merci !

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 destruction des instances de services.
- **Architecture et Performance** : Architecture modulaire et 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 (inutil ..)** : `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

- **Demande de Service** : Utilisez le `type-hinting` dans les contrôleurs pour accéder aux services.
- **Démarrage automatique du service par Symfony** : Grâce au `type-hinting`, Symfony crée l'objet `LoggerInterface` tout seul, pas besoin de le faire à la main !

```
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** : Activation de 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 constructeur

```
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     public function __construct(
10         private readonly MessageGenerator $messageGenerator,
11         private readonly MailerInterface $mailer,
12     ) { }
13 }
```

- Toutes les dépendances seront accessibles dans les autres méthodes avec `$this`.

Paramètres pour les services

- Rappel : Autowiring = Symfony créer les objets des services lui-même

Paramètres pour les services

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

```
1 // src/Service/SiteUpdateManager.php
2 class SiteUpdateManager
3 {
4     public function __construct(
5         private readonly MessageGenerator $messageGenerator,
6         private readonly 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 lui-même des paramètres ?

```

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

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

```

- Si paramètres “autowirables”, OK, Symfony suit toute la chaîne de dépendances !

Paramètres pour les services

- Rappel : Autowiring = Symfony créer les objets des services lui-même
- Si paramètres pas "autowirables" ??

```
1 // src/Service/SiteUpdateManager.php
2 class SiteUpdateManager
3 {
4     public function __construct(
5         private readonly 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.

Autre possibilité : Les paramètres du container

- ❑ Et si plusieurs services ont le même paramètre pas autowirable ?
- ❑ On peut fait des paramètres du container.

```
1 # config/services.yaml
2 parameters:
3     app.admin.email: 'admin.email@symfony.com'
```

```
1 class SiteUpdateManager
2 {
3     public function __construct(
4         private readonly MailerInterface $mailer,
5         private readonly ContainerBagInterface $containerBag
6     ) {
7         // ...
8     }
9
10    public function someServiceMethod(): void
11    {
12        $adminEmail = $this->containerBag->get('app.admin.email');
13    }
14 }
```

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.
```