Symfony

Routage et services

Maxime Puys

12 janvier 2025

Plan

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

```php
// src/Controller/BlogController.php
use Symfony\Component\Routing\Attribute\Route;

#[Route('/blog', name: 'blog_list')]
public function list(): Response
{
    // ...
}
```

- 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.

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

Correspondance des méthodes HTTP

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

```php
// src/Controller/BlogController.php
#[Route('/api/posts', name: 'api_posts', methods: ['GET','HEAD'])]
public function show(): Response
{
    // ... return a JSON response with the post
}

//Can be same path, but name may be different
#[Route('/api/posts', name: 'api_posts_edit', methods: ['PUT'])]
public function edit(): Response
{
    // ... edit a post
}
```

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 !

```php
// src/Controller/BlogController.php
#[Route('/api/posts', name: 'api_posts', methods: ['GET','HEAD'])]
public function show(): Response
{
    // ... return a JSON response with the post
}

//Can be same path, but name may be different
#[Route('/api/posts', name: 'api_posts_edit', methods: ['PUT'])]
public function edit(): Response
{
    // ... edit a post
}
```

Correspondance des expressions

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

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

Conditions de routing approfondies

- Fonctionne aussi avec de l'AJAX :

```php
// src/Controller/BlogController.php
#[Route(
        '/contact/ajax',
        name: 'contact_ajax',
        options: ['expose' => true], // Needed to expose as an AJAX route
                                     // (using a specific bundle)
        condition: "request.isXmlHttpRequest()
        and service('auth_checker').isAuthorizedToContact()",
        // Check for AJAX request and then using a custom service
        // for specific verifications using AsRoutingConditionalServ
    )]
public function contact(): Response
{
  // ...
}
```

Conditions de routing approfondies

- Et le service en question :

```php
#[AsRoutingConditionService(alias: 'auth_checker')]
class AuthChecker
{
    public function check(Request $request): bool
    {
        // Some custom verification here
    }
}
```

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`.

```php
// src/Controller/BlogController.php
#[Route(
        '/blog/{page}',
        name:'blog_list',
        requirements: ['page' => '\d+'],
        methods: ['GET']
    )]
public function list(int $page): Response
{
    // ...
}
```

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.

```php
// src/Controller/BlogController.php
#[Route(
        '/blog/{page}',
        name:'blog_list',
        requirements: ['page' => '\d+'],
        methods: ['GET']
    )]
public function list(int $page): Response
{
    // ...
}
```

Paramètres optionnels et valeurs par défaut

- Possibilité de rendre les paramètres optionnels.

```php
// src/Controller/BlogController.php
#[Route(
        '/blog/{page}',
        name:'blog_list',
        requirements: ['page' => '\d+'],
        defaults: ['page' => 1],
        methods: ['GET']
    )]
public function list(int $page = 1): Response
{
    // ...
}
```

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

```php
// src/Controller/BlogController.php
#[Route(
        '/blog/{page}',
        name:'blog_list',
        requirements: ['page' => '\d+'],
        defaults: ['page' => 1],
        methods: ['GET']
    )]
public function list(int $page = 1): Response
{
    // ...
}
```

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

Priorité entre les routes

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

```php
// src/Controller/BlogController.php
#[Route('/blog/{slug}', name:'blog_show')]
public function show(string $slug): Response
{
    // Default action
}

#[Route('/blog/special/{slug}', name:'blog_special_show', priority: 2)]
public function list(): Response
{
    // Special action
}
```

- **Critique** : Dans un respect des bonnes pratiques, les priorités sont à éviter. En géneral, 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).

```php
// src/Controller/BlogController.php
// This route has a greedy pattern and is defined first.
#[Route('/blog/{slug}', name:'blog_show')]
public function show(string $slug): Response
{
    // Default action
}

// This route could not be matched without defining a higher priority than 0.
#[Route('/blog/special/{slug}', name:'blog_special_show', priority: 2)]
public function list(): Response
{
    // Special action
}
```

Conversion des paramètres

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

```php
// src/Controller/BlogController.php

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

Conversion des paramètres

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

```php
// src/Controller/BlogController.php

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

- 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.

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

Groupes et préfixes de route

- Les groupes de routes partagent des options communes.

```php
#[Route('/blog', name: 'blog_', requirements:['_locale' => 'en|es|fr'])]
class BlogController extends AbstractController
{
    #[Route('/{_locale}', name: 'index')]
    public function index(): Response {}

    #[Route('/{_locale}/posts/{slug}', name: 'show')]
    public function show(string $slug): Response {}
}
```

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.

```php
#[Route('/blog', name: 'blog_', requirements:['_locale' => 'en|es|fr'])]
class BlogController extends AbstractController
{
    #[Route('/{_locale}', name: 'index')]
    public function index(): Response {}

    #[Route('/{_locale}/posts/{slug}', name: 'show')]
    public function show(string $slug): Response {}
}
```

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`

```php
#[Route('/blog', name: 'blog_', requirements:['_locale' => 'en|es|fr'])]
class BlogController extends AbstractController
{
    #[Route('/{_locale}', name: 'index')]
    public function index(): Response {}

    #[Route('/{_locale}/posts/{slug}', name: 'show')]
    public function show(string $slug): Response {}
}
```

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`.

```php
// src/Controller/BlogController.php

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

- 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.

```yaml
# config/routes.yaml
legacy_doc:
    path: /legacy/doc
    controller: Symfony\Bundle\FrameworkBundle\Controller\RedirectController
    defaults:
        # this value can be another route
        route: 'doc_page'
        # or an absolute path or an absolute URL
        path: 'https://legacy.example.com/doc'

        # sets if redirection is permanent
        permanent: true
```

- Sans appeller de controlleur ...

Redirection au niveau des controlleur

```php
// src/Controller/BlogController.php
public function index(): RedirectResponse  // Attention au type de retour!
{
    // redirects to the "homepage" route
    return $this->redirectToRoute('homepage');

    // does a permanent HTTP 301 redirect
    return $this->redirectToRoute('homepage', [], Response::HTTP_MOVED_PERMANENTLY);

    // redirect to a route with parameters
    return $this->redirectToRoute('app_lucky_number', ['max' => 10]);

    // redirects to a route and maintains the original query string parameters
    return $this->redirectToRoute('blog_show', $request->query->all());

    // redirects to the current route (e.g. for Post/Redirect/Get pattern):
    return $this->redirectToRoute($request->attributes->get('_route'));

    // redirects externally
    return $this->redirect('http://symfony.com/doc');
}
```

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.

```php
// src/Controller/BlogController.php
$signUpPage = $this->generateUrl('sign_up');
$userProfilePage = $this->generateUrl(
    'user_profile',
    ['username' => $user->getUserIdentifier()]
);
```

- 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.

```php
use Symfony\Component\Routing\Exception\RouteNotFoundException;

try {
    $url = $this->router->generate($routeName, $routeParameters);
} catch (RouteNotFoundException $e) {
    // 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.

```php
<?php

function add($x, $y)
{
    return $x + $y;
}

$result = add(1,2);
echo $result; // 3
```

Rappel : Type hinting en PHP

- Génial, ça marche aussi avec les flottants !

```php
<?php

function add($x, $y)
{
    return $x + $y;
}

$result = add(1.0,2.5);
echo $result; // 3.5
```

Rappel : Type hinting en PHP

- Et même avec des strings !

```php
<?php

function add($x, $y)
{
    return $x + $y;
}

$result = add('Hi','There');
echo $result;
```

Rappel : Type hinting en PHP

- Et même avec des strings !

```php
<?php

function add($x, $y)
{
    return $x + $y;
}

$result = add('Hi','There');
echo $result;
```

- Ah toute compte fait non ...

```
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 !

```php
<?php

function add(int $x, int $y)
{
    return $x + $y;
}

$result = add(1,2);
echo $result;  // 3
```

Rappel : Type hinting en PHP

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

```php
<?php

function add(int $x, int $y): int
{
    return $x + $y;
}

$result = add(1,2);
echo $result;  // 3
```

Rappel : Type hinting en PHP

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

```php
<?php

function add(int $x, int $y): int
{
    return $x + $y;
}

$result = add(1,2);
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.

```php
// src/Service/MessageGenerator.php
class MessageGenerator {
    public function getHappyMessage(): string {
        // Retourne un message aléatoire
    }
}
```

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** : Grace 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.

```yaml
# config/services.yaml
services:
    _defaults:
        autowire: true
        autoconfigure: true
    App\:
        resource: '../src/'
        // Exclusions et autres paramètres
```

Types Autowirable

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

```
$ symfony console debug:autowiring

Autowirable Types
=================
 The following classes & interfaces can be used as type-hints when autowiring:
 App\Kernel (kernel)
 Interface for annotation readers.
 Doctrine\Common\Annotations\Reader (annotations.cached_reader)
 Doctrine\Common\Persistence\ManagerRegistry (doctrine)

 [...]

 App\Controller\BlogController
 App\Repository\BlogArticleRepository
 App\Service\MessageGenerator
```

Injection de Dépendances

- Type-hinting au niveau du controller

```php
// src/Service/SiteUpdateManager.php
namespace App\Service;

use App\Service\MessageGenerator;
use Symfony\Component\Mailer\MailerInterface;

class SiteUpdateManager
{
    public function __construct(
        private readonly MessageGenerator $messageGenerator,
        private readonly MailerInterface $mailer,
    ) { }
}
```

- 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 ?

```php
// src/Service/SiteUpdateManager.php
class SiteUpdateManager
{
    public function __construct(
        private readonly MessageGenerator $messageGenerator,
        private readonly MailerInterface $mailer,
    ) {
        // ...
    }
}
```

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

Paramètres pour les services

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

```php
// src/Service/SiteUpdateManager.php
class SiteUpdateManager
{
    public function __construct(
        private readonly MessageGenerator $messageGenerator,
        private readonly MailerInterface $mailer,
    ) {
        // ...
    }
}
```

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

- 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” ??

```php
// src/Service/SiteUpdateManager.php
class SiteUpdateManager
{
    public function __construct(
        private readonly MailerInterface $mailer,
        string $adminEmail  // D'où Symfony sort l'adresse email ?
    ) {
        // ...
    }
}
```

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

Configuration Manuelle des Arguments

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

```yaml
# config/services.yaml
services:
  App\Service\SiteUpdateManager:
      arguments:
          $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.

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

```php
class SiteUpdateManager
{
    public function __construct(
        private readonly MailerInterface $mailer,
        private readonly ContainerBagInterface $containerBag
    ) {
        // ...
    }

    public function someServiceMethod(): void
    {
      $adminEmail = $this->containerBag->get('app.admin.email');
    }
}
```

Sélection d'un Service Spécifique

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

```php
// src/Service/MessageGenerator.php
use Psr\Log\LoggerInterface;
class MessageGenerator
{
    private $logger;
    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
}
```

```yaml
# config/services.yaml
services:
    App\\Service\\MessageGenerator:
        arguments:
            $logger: '@monolog.logger.request'
```

Binding des Paramètres

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

```yaml
# config/services.yaml
services:
    _defaults:
        bind:
            # valeur pout tout paramètre nommé $adminEmail
            $adminEmail: 'manager@example.com'

            # type pour tout paramètre implémentant LoggerInterface
            Psr\Log\LoggerInterface: '@monolog.logger.request'

            # ou les deux
            string $adminEmail: 'manager@example.com'
            Psr\Log\LoggerInterface $requestLogger: '@monolog.logger.request'
```

Paramètres Abstraits

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

```yaml
# config/services.yaml
services:
    App\Service\MyService:
        arguments:
            $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 configuation :

```
$ symfony console lint:container

In DefinitionErrorExceptionPass.php line 49:

Cannot autowire service "App\Service\TestService": argument "$a" of method
"__construct()" is type-hinted "int", you should configure its value
explicitly.
```