Symfony

Routage et services

Maxime Puys

29 avril 2024

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 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

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

/**
 * @Route('/blog', name: 'blog_list')
 */
public function list(): Response
{
    // ...
}
```

- 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 aussie 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", methods={"GET","HEAD"})
 */
public function show(): Response
{
    // ... return a JSON response with the post
}

/**
 * @Route("/api/posts", 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", methods={"GET","HEAD"})
 */
public function show(): Response
{
    // ... return a JSON response with the post
}

/**
 * @Route("/api/posts", 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 configuration parameters:
 * condition: "request.headers.get('User-Agent') matches '%app.allowed_browsers%'"
 */
public function contact(): Response
{
    // ...
}
```

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+"})
 */
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+"})
 */
public function list(int $page): Response
{
    // ...
}
```

Paramètres optionnels et valeurs par défaut

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

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

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

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

- Il est possible d'avoir plus d'un paramètre optionnel, mais dand ce cas, tout ce qui 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
{
    // ...
}

/**
 *
 * @Route("/blog/list", name="blog_list", priority=2)
 */
public function list(): Response
{
    // ...
}
```

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
{
    // ...
}

/**
 * This route could not be matched without defining a higher priority than 0.
 * @Route("/blog/list", name="blog_list", priority=2)
 */
public function list(): Response
{
    // ...
}
```

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 le champ `slug` de la table).
- Erreur 404 si l'entité n'est pas trouvée.

Groupes et préfixes de route

- Les groupes de routes partagent des options communes.

```php
/**
 * @Route("/blog", requirements={"_locale": "en|es|fr"}, name="blog_")
 */
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", requirements={"_locale": "en|es|fr"}, name="blog_")
 */
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`

```
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 {}

0      /**
1       * @Route("/{_locale}/posts/{slug}", name="show")
2       */
3      public function show(string $slug): Response {}
4  }
```

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');
}
```

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
```

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

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');
}
```

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      // ...
0  }
```

Routes localisées (i18n)

- Une bonne utilisation pour les préfixes de route !

```yaml
# config/routes/annotations.yaml
controllers:
    resource: '../../src/Controller/'
    type: annotation
    prefix:
        en: '' # pas de prefixe pour la langue principale
        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.

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

- 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.

```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

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** : Architecture modulair 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** : `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

- **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.

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

- 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
    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
{
    private $messageGenerator;
    private $mailer;

    public function __construct(
        MessageGenerator $messageGenerator,
        MailerInterface $mailer,
    ) {
        $this->messageGenerator = $messageGenerator;  // Remove in PHP8
        $this->mailer = $mailer;                      // Remove in PHP8
    }
}
```

Paramètres pour les services

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

```php
// src/Service/SiteUpdateManager.php
class SiteUpdateManager
{
    public function __construct(
        MessageGenerator $messageGenerator,
        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 des paramètres ?

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

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

- Si paramètres “autowirables”, OK !

Paramètres pour les services

- Si paramètres pas “autowirables” ??

```php
// src/Service/SiteUpdateManager.php
class SiteUpdateManager
{
    public function __construct(
        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.

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.
```