

Mise en place d'une application symfony

Ibrahim ALAME

14/02/2023

1 Création d'une application

1.1 Création d'une application

Nous allons utiliser le CLI pour créer notre première application. Ouvrez un terminal et commencez par vous placez dans le dossier où vous souhaitez créer l'application, par exemple sur le bureau, puis tapez la commande suivante :

```
symfony new dymaproject
```

Vous aurez ensuite quelque chose comme cela :

```
17:33:36 ✓ erwan:~/Bureau$ symfony new dymaproject
* Creating a new Symfony project with Composer
  (running /usr/bin/composer create-project symfony/skeleton /home/erwan/Bureau/
dymaproject --no-interaction)

* Setting up the project under Git version control
  (running git init /home/erwan/Bureau/dymaproject)

[OK] Your project is now ready in /home/erwan/Bureau/dymaproject
```

Ouvrez ensuite le projet dans Visual Studio Code.

1.2 Git par défaut

Par défaut, Symfony utilise Git et crée un premier commit avec les fichiers initiaux. Vous pouvez le voir en faisant :

```
git log
```

Vous aurez par exemple :

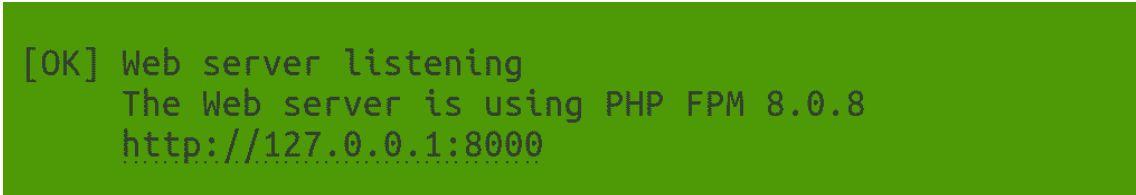
```
Commit 138afe4431c602a0c4fc86bbfa4f2965d875309f (HEAD -> master)
Author: dynafr <contact@dyma.fr>
Date: Tue Jul 6 17:34:11 2021 +0200

Add initial set of files
```

Lancer le serveur de développement. Pour lancer le serveur de développement local, il suffit d'ouvrir un terminal dans le dossier du projet et de faire :

```
symfony serve
```

Le terminal vous indiquera l'adresse où votre application sera disponible localement :



```
[OK] Web server listening
     The Web server is using PHP FPM 8.0.8
     http://127.0.0.1:8000
```

Vous pouvez couper le serveur à tout moment en faisant **Ctrl + C**.

1.3 Le fichier `composer.json`

Le fichier `composer.json` contient toutes les dépendances du projet. Par défaut, **Symfony** en installe plusieurs que nous allons voir ensemble.

- `symfony/console` : permet de créer des interfaces en ligne de commande plus facilement. C'est le composant utilisé par le CLI **Symfony**.
- `symfony/dotenv` : permet d'automatiquement parser les fichiers avec l'extension `.env` pour charger les variables d'environnement et les rendre disponibles sur `$_SERVER` ou `$_ENV`.
- `symfony/flex` : permet de gérer l'architecture des dossiers et fichiers des projets **Symfony**. Il est utilisé notamment pour générer l'architecture de base lorsque nous avons fait `symfony new`. Il permet également d'installer de nouvelles dépendances.
- `symfony/framework-bundle` : librairie qui permet d'intégrer les composants **Symfony** avec le `framework`. Nous verrons que nous utiliserons de nombreux composants.
- `symfony/runtime` : gère le lancement de **Symfony** quel que soit l'environnement (PHP-FPM, ReactPHP, Swoole etc).
- `symfony/yaml` : permet de parser des fichiers contenant du `YAML` et de les convertir en tableaux associatifs PHP.

1.4 Le fichier `.gitignore`

Nous ne passerons pas trop de temps sur **Git**, nous vous invitons à faire la formation **Git** disponible.

Ce fichier est déjà configuré pour les projets **Symfony** pour ignorer les bons dossiers et fichiers lors des `commits`.

1.5 Le fichier `.env`

Le fichier `.env` contient les variables d'environnement.

Ce sont des variables contenant des valeurs nécessaires à l'application pour fonctionner (par exemple des mots de passe ou des secrets pour se connecter à des API tierces etc).

1.6 Le dossier vendor

Ce dossier contient toutes les dépendances installées et leurs dépendances en fonction du fichier `composer.json`.

1.7 Le dossier var

- Le dossier `var/cache` contient le système de cache de Symfony. Toute la configuration de l'application est parsée une seule fois et ensuite mise en cache dans ce dossier après le lancement. Cela permet de gagner en performance.
- Le dossier `var/logs` contient les logs de l'application. Ce sont des fichiers contenant toutes les informations sur les requêtes et les éventuelles erreurs. C'est très utile pour le débogage.

1.8 Le dossier src

Le dossier `src` pour source contient le code de votre application. Par défaut, il contient la classe `Kernel` (noyau) dans le fichier `Kernel.php`. C'est le point d'entrée dans l'application : il va charger toutes les configurations de l'application.

1.9 Le dossier public

Le dossier `public` contient tous les ressources statiques ou `assets` (images, vidéos etc) et le point d'entrée `index.php`.

1.10 Le dossier config

- Le dossier `config` contient la configuration de l'application.
- Le fichier `routes.yaml` permet de configurer des routes.
- Le fichier `services.php` contient la configuration des services. Nous étudierons bien sûr les services en détail.
- Le fichier `bundles.php` permet d'activer ou de désactiver des packages. Les packages ou bundles sont des fonctionnalités utilisables directement dans l'application (c'est équivalent aux modules dans d'autres frameworks).●

1.11 Le dossier bin

Le dossier `bin` (pour binaries) contient les fichiers exécutables.

2 Les composants Symfony

Les composants Symfony sont des bibliothèques PHP utilisables dans tout environnement PHP (donc pas forcément qu'avec Symfony).

Ces composants sont utilisés dans de très nombreux frameworks et projets PHP. Quelques exemples : bien sûr Symfony, PrestaShop, Drupal,

Nous allons prendre quelques exemples :

- `Validator` : composant permettant de valider facilement les classes PHP.
- `Uid` : permet de gérer les identifiants uniques (UUIDs).

- **Serializer** : permet de transformer des entités dans un format spécifique (JSON ou YAML par exemple) et de désérialiser depuis ces formats vers des entités PHP.
HttpClient, HttpKernel et HttpFoundation : composants pour gérer les requêtes HTTP entrantes et les réponses HTTP sortantes.
 - **Form** : permet de créer et de gérer des formulaires de manière simplifiée.
- Il y a en tout une soixantaine de composants maintenus par l'équipe Symfony. Nous les verrons au fur et à mesure de nos besoins dans la formation.

3 Les bundles Symfony

Les bundles, contrairement aux composants, ne sont utilisables que dans le framework Symfony. Ils sont soit créés et maintenus par l'équipe Symfony, soit par des entités tiers. Ils sont alors appelés des bundles tiers (third-party bundles).

Un bundle permet d'ajouter une fonctionnalité à votre application. Symfony appelle les bundles également des packages, comme nous l'avons vu dans la leçon précédente.

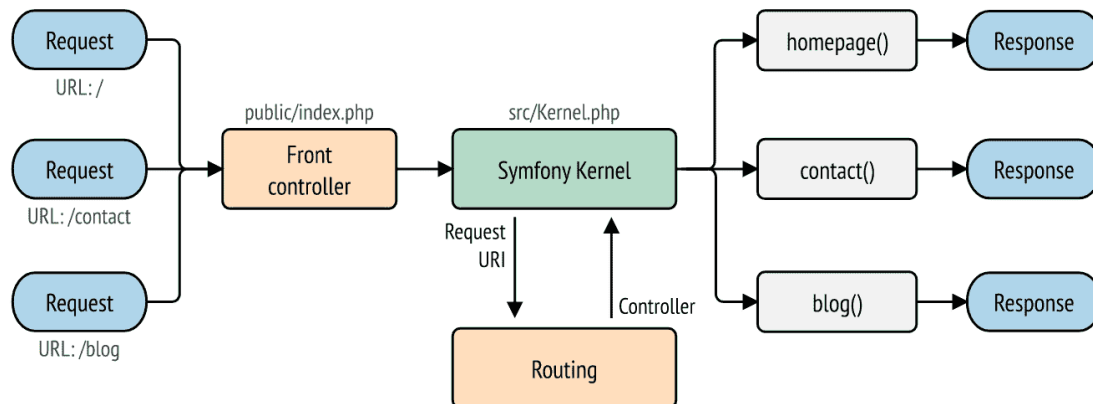
Dans d'autres frameworks, les bundles sont appelés modules ou plugins. Prenons également quelques exemples :

- **symfony/twig-bundle** : permet d'utiliser Twig de manière optimale avec Symfony. Twig est le moteur de templates pour créer les vues utilisé avec Symfony.
- **doctrine/doctrine-bundle** : même chose pour Doctrine. Doctrine est un ORM (couche d'abstraction à la base de données) pour PHP.
- **symfony/swiftmailer-bundle** : permet de gérer les envois d'emails avec les principaux services (Mandrill, SendGrid, Amazon SES etc).

Il y a bien sûr beaucoup d'autres bundles Symfony que nous utiliserons au fur et à mesure.

4 Fonctionnement d'une application Symfony

Nous allons brièvement voir dans l'ordre les éléments qui gèrent une requête HTTP entrante d'un client :



4.1 Le script index.php

Lorsqu'une requête HTTP est reçue par le serveur Web (par exemple NGINX) puis transmise à PHP-FPM (grâce au protocole FastCGI), le premier script PHP à être exécuté est `public/index.php`. On l'appelle point d'entrée ou Front controller.

Le front controller est le contrôleur qui gère l'ensemble des requêtes d'une application. Autrement dit, toutes les requêtes HTTP entraînent obligatoirement l'exécution de ce script.

Si vous avez suivi le cours PHP vous noterez que c'est déjà une différence avec une application PHP basique qui exécute un script différent suivant la requête HTTP et qui n'a donc pas un seul point d'entrée pour l'ensemble des requêtes.

Voici son contenu :

```
1  <?php
2
3  use App\Kernel;
4
5  require_once dirname(__DIR__) . '/vendor/autoload_runtime.php';
6
7  return function (array $context) {
8      return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
9  };
```

Ce script commence par initialiser la fonctionnalité du chargement automatique en fonction de l'environnement (autoload).

Ce script crée ensuite une instance de la classe Kernel.

4.2 La classe Kernel

La classe Kernel est chargée par le script index.php, elle se trouve dans `src/kernel.php`.

Voici son contenu :

```
1  <?php
2
3  namespace App;
4
5  use Symfony\Bundle\FrameworkBundle\Kernel\MicroKernelTrait;
6  use Symfony\Component\HttpKernel\Kernel as BaseKernel;
7
8  class Kernel extends BaseKernel
9  {
10     use MicroKernelTrait;
11 }
```

Symfony
Component
HttpKernel

Kernel qui est ici renommée en `BaseKernel` est le cœur de Symfony. Elle va initialiser tous les bundles Symfony avec la configuration de votre application.

Vous pouvez aller regarder les fonctions d'initialisation des bundles dans `dymaproject/vendor/symfony/http-kernel`

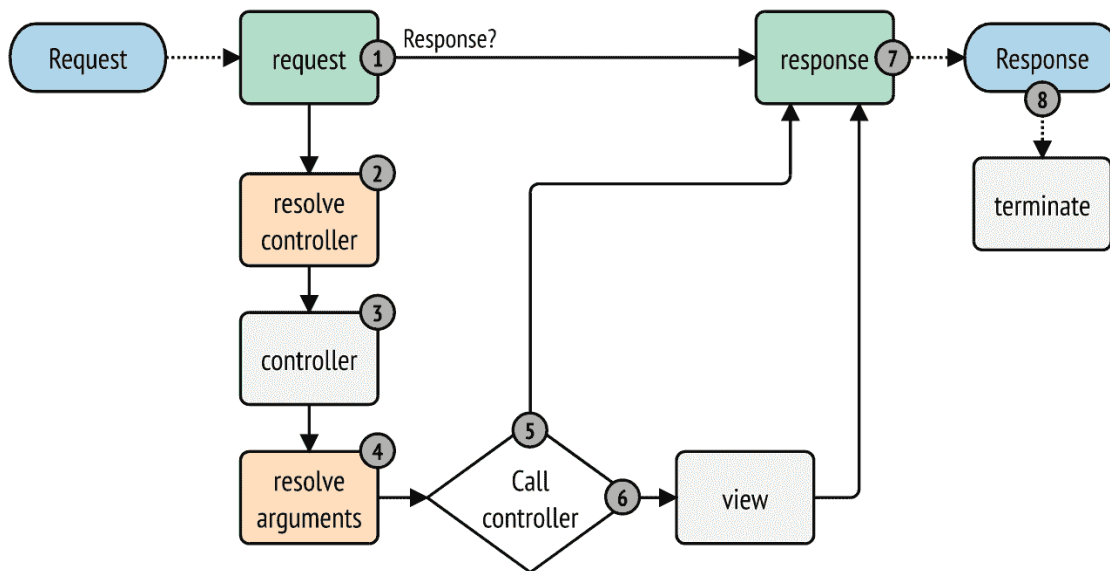
Vous y trouverez notamment une fonction `initializeBundles()` qui est chargée de cette initialisation.

4.3 La fonction `handle()`

Une fois toutes les configurations, le routing et les bundles chargés, la requête est gérée par la fonction `handle()` de la classe `HttpKernel`.

Cette fonction a pour objectif de partir d'une requête HTTP et de la transformer en la réponse HTTP attendue.

C'est donc une fonction très importante dont le schéma suivant résume son fonctionnement :



Cette fonction va envoyer des événements qui sont ensuite gérés par des gestionnaires d'événements.

Nous allons voir les grandes étapes du processus :

1. Un événement `kernel.request` est émis. Plusieurs gestionnaires sont appelés à ce niveau. Par exemple, le gestionnaire du bundle Security qui va déterminer si la requête est autorisée ou non. La requête peut être transformée en réponse 403 (pour forbidden) et retournée à ce moment. Un autre gestionnaire important pour cet événement est le Router, il va déterminer quel contrôleur doit être appelé en fonction de la requête.
2. Le Contrôleur est résolu. La fonction `handle()` va appeler une fonction `getController()` (située dans `dymaproject/vendor/symfony/http-kernel/Controller/ControllerResolver.php`) qui va être chargée de récupérer et d'exécuter le bon contrôleur en fonction de la propriété `_controller` qui a été placé par le Router sur le tableau associatif `Request`.

Vous pouvez voir cette mécanique au début de la fonction `getController()` :

```
1 <?php
2 public function getController(Request $request) {
3     if (!$controller = $request->attributes->get('_controller')) {
```

```

4     if (null !== $this->logger) {
5         $this->logger->warning('Unable to look for the controller as the "_controller" parameter is missing.');
```

- 6 }
 - 7 return false;
 - 8 }
 - 9 }
3. Un événement `kernel.controller` est émis. Il permet d'exécuter des gestionnaires d'événement juste avant que le contrôleur récupéré ne soit exécuté. C'est un design pattern commun appelé *hooks* en programmation. Cela permet d'exécuter du code lors d'événements clés.
 4. Récupération des arguments pour le contrôleur. Avant l'exécution du contrôleur, une fonction `getArguments()` va récupérer les arguments à passer au contrôleur en fonction de l'environnement et de la configuration.
 5. Exécution du contrôleur. Le contrôleur, qui est chargé de construire la réponse à renvoyer au client, est exécuté. Il va créer une réponse qui peut être une page HTML, une réponse au format JSON ou toute autre réponse HTTP valide. C'est à cette étape que sera exécuté votre code : votre contrôleur, qui va éventuellement appeler vos modèles et vos vues.
 6. Un événement `kernel.view` est émis. Permet de gérer les cas où aucune réponse n'est retournée par le contrôleur. Par défaut, Symfony n'a aucune gestionnaire pour cet événement et vos contrôleurs doivent obligatoirement retourner une réponse. Certains bundles utilisent cet événement, comme par exemple `FOSRestBundle`.
 7. Un événement `kernel.response` est émis. Permet d'exécuter des gestionnaires juste avant que la réponse ne soit envoyée au client. Des exemples d'utilisation sont la modification des headers de la réponse, l'ajout de cookies etc.
 8. Un événement `kernel.terminate` est émis. La réponse a été envoyée au client. Cet événement permet de procéder aux nettoyages ou à des tâches pouvant être réalisées après la réponse HTTP (par exemple, enregistrement de données d'analyse, envoi d'emails etc).

5 Création d'une première page

5.1 Utiliser HTTPS en local

Lorsque vous exécutez `symfony serve`, vous aurez l'avertissement suivant :

```

17:51:13 ✓ erwan:(master)~/Bureau/dynaproject$ symfony serve

[WARNING] run "symfony server:ca:install" first if you want to run the web se
rver with TLS support, or use "--no-tls" to avoid this warning
```

Il signifie que par défaut, votre serveur de développement local utilisera le protocole HTTP et non sa version sécurisée, le protocole HTTPS (utilisant le protocole TLS).

Ouvrez donc un terminal dans votre projet et entrez :

```
symfony server:ca:install
```

Le CLI Symfony créera un certificat local pour pouvoir utiliser localement HTTPS. Il l'ajoutera à vos navigateurs automatiquement :

```
The local CA is now installed in the system trust store!
The local CA is now installed in the Firefox and/or Chrome/Chromium trust store (requires browser re
```

Il faudra les redémarrer pour que le certificat soit pris en compte. Ce certificat n'est évidemment valide que lors du développement en local. Vous pouvez ensuite relancer votre serveur de développement :

```
symfony serve
```

Il sera cette fois disponible par défaut à l'adresse `https://127.0.0.1:8000/`.
Notez bien l'utilisation du protocole HTTPS dans l'URL.

5.2 Le langage YAML

Le YAML (pour YAML Ain't Markup Language ou "YAML n'est pas un langage de balisage") est un langage permettant de représenter des informations élaborées tout en conservant une grande lisibilité. Il repose principalement sur l'indentation.

C'est un langage utilisé le plus souvent pour des fichiers de configuration.

Nous allons voir les principaux éléments de la syntaxe de ce langage car tout ce que nous verrons à partir de maintenant utilisera du YAML :

- # permet de commenter une ligne s'il est placé au début d'une ligne. S'il est avant une chaîne de caractère, il signifie nombre littéral.

- ~ signifie valeur nulle.

- permet de créer des listes. Chaque élément d'une liste doit être précédé par - puis un espace.

Il doit y avoir un élément par ligne.

- clé : valeur permet de déclarer un map clé / valeur.

L'indentation par des espaces permet de créer une hiérarchie, par exemple :

```
1 paul:
2   nom: Paul Dupont
3   emploi: Developer
```

Une structure plus complexe :

```
1 - paul:
2   nom: Paul Dupont
3   emploi: Developer
4   languages:
5     - javascript
6     - css
7     - html
8     - php
```

5.3 Le fichier config/routes/routes.yaml

Dans **Symfony**, vous pouvez écrire des routes directement dans les contrôleurs, comme nous le verrons, ou dans des fichiers YAML, XML ou PHP.

Depuis **Symfony 5.1** seules les routes dans des fichiers écrits en YAML sont chargées par défaut. Nous ne verrons donc pas les routes écrites en XML et PHP qui ne sont plus recommandées.

Par défaut, le fichier de route chargé est `config/routes/routes.yaml` :


```

1 controllers:
2     resource:
3         path: ../src/Controller/
4         namespace: App\Controller
5     type: attribute

```

- **controllers** indique le début de la configuration des contrôleurs.
- **resource** est utilisée pour déterminer la source des contrôleurs, c'est-à-dire où les fichiers contenant les contrôleurs sont situés.
 path spécifie le chemin du répertoire où les fichiers des contrôleurs sont situés. Dans ce cas, les contrôleurs sont stockés dans le dossier Controller du répertoire src.
- **namespace** spécifie le namespace PHP des contrôleurs. Les namespaces PHP sont utilisés pour organiser et regrouper les classes de manière logique, évitant ainsi les conflits de noms entre les classes. Ici, le namespace des contrôleurs est **App** **Controller**.
- **type: attribute** indique que les contrôleurs sont définis en utilisant des attributs PHP (disponibles à partir de PHP 8.0). Les attributs permettent d'ajouter des métadonnées à une classe, une méthode ou une propriété. Dans le contexte de Symfony, les attributs sont utilisés pour définir des informations sur le routage, telles que l'URL d'une route et la méthode HTTP associée.

Modifiez le fichier `config/routes/routes.yaml` pour ajouter une route pour notre premier contrôleur :

```

1 controllers:
2     resource:
3         path: ../src/Controller/
4         namespace: App\Controller
5     type: attribute
6
7 index:
8     path: /
9     controller: App\Controller\DefaultController::index

```

- **index** est le nom de la route.
- **path** est le chemin de la route qui va matcher avec l'URI de la requête HTTP.
- **controller** est le chemin vers le contrôleur et l'action à exécuter pour la route. Par exemple ici, la classe contrôleur à exécuter est `DefaultController`, qui se situe dans `dymaproject/src/Controller`. L'action est située après les doubles deux points `::`. Elle correspond à la méthode sur la classe contrôleur qui va être exécutée à chaque requête qui correspond à la route.

5.4 Création du contrôleur

Pour le moment, le contrôleur n'existe pas et vous aurez une page d'erreur lorsque vous irez sur `https://127.0.0.1:8000/`.

Symfony Exception Symfony Docs Symfony Support

Error > InvalidArgumentException > InvalidArgumentException HTTP 500 Internal Server Error

The controller for URI "/" is not callable: Controller "App\Controller\DefaultController" does neither exist as service nor as class.

Exceptions 3 Stack Traces 3

InvalidArgumentException

- in vendor/symfony/http-kernel/Controller/ControllerResolver.php (line 88)
- in vendor/symfony/http-kernel/HttpKernel.php -> **getController** (line 140)
- in vendor/symfony/http-kernel/HttpKernel.php -> **handleRaw** (line 79)
- in vendor/symfony/http-kernel/Kernel.php -> **handle** (line 199)
- in vendor/symfony/runtime/Runner/Symfony/HttpKernelRunner.php -> **handle** (line 37)
- in vendor/autoload_runtime.php -> **run** (line 35)
- require_once** ('/home/erwan/Bureau/dymaproject/vendor/autoload_runtime.php')
in public/index.php (line 5)

```

1. <?php
2.
3. use App\Kernel;
4.
5. require_once dirname(__DIR__).'/vendor/autoload_runtime.php';
6.
7. return function (array $context) {
8.     return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
9. };
10.

```

InvalidArgumentException

Controller "App\Controller\DefaultController" does neither exist as service nor as class.

Notez l'erreur : "Le contrôleur `DefaultController` n'existe pas".

Nous devons donc le créer dans `src/Controller/DefaultController.php` :

```

1  <?php
2
3  namespace App\Controller;
4
5  use Symfony\Component\HttpFoundation\Response;
6
7  class DefaultController
8  {
9      public function index()
10     {
11         return new Response('<h1>Hello World !</h1>');
12     }
13 }

```

Expliquons en détail ce contrôleur.

`namespace App`

`Controller;` : permet à la classe d'être importée automatiquement par l'`autoload`. Il faut impérativement commencer par `App` puis respecter le chemin depuis `src` jusqu'à la classe. Comme notre classe est dans le dossier `src/Controller` il faut donc indiquer `App Controller`.

Si vous voulez en savoir plus revoyez les chapitres namespace et Composer et autoload dans la formation PHP.

Nous devons ensuite déclarer la classe en respectant le nom donné dans la route sur la propriété `controller`. Dans notre cas, il faut donc obligatoirement appeler la classe `DefaultController`.

Il faut ensuite créer une méthode sur la classe qui correspond au nom de l'action déclarée sur la route. Dans notre cas l'action est `index`. Nous devons donc créer une méthode `index()` qui sera exécutée.

Dans cette méthode, nous devons obligatoirement retourner une réponse. Comme nous l'avons vu, l'objectif d'un contrôleur est de recevoir une requête HTTP et de retourner une réponse HTTP.

Pour créer un objet `Response`, commencez à taper `Resp` et vous aurez l'autocomplétion et l'auto-importation de la classe par VS Code.

6 L'objet Request du composant HttpFoundation

6.1 Le composant HttpFoundation

En PHP, vous savez qu'il existe un ensemble de variables globales. Par exemple : `$_SERVER`, `$_GET` ou `$_COOKIE`. Le composant `Symfony HttpFoundation` permet de remplacer ces variables globales par des objets, et notamment par `Request` et `Response`.

6.2 L'objet Request

Pour créer un objet `Request`, il suffit de faire :

```
1 <?php
2 use Symfony\Component\HttpFoundation\Request;
3
4 $request = Request::createFromGlobals();
```

La méthode statique `createFromGlobals()` permet de créer l'objet en passant la plupart des variables globales PHP à une méthode `createRequestFromFactory()` :

```
1 public static function createFromGlobals()
2 {
3     $request = self::createRequestFromFactory($_GET, $_POST, [], $_COOKIE, $_FILES, $_SERVER);
4
5     if (0 === strpos($request->headers->get('CONTENT_TYPE', ''), 'application/x-www-form-urlencoded')
6         && \in_array(strtoupper($request->server->get('REQUEST_METHOD', 'GET')), ['PUT', 'DELETE', 'PATCH']))
7     {
8         parse_str($request->getContent(), $data);
9         $request->request = new InputBag($data);
10    }
```

```

11
12     return $request;
13 }

```

Sur l'objet `Request`, nous retrouvons les propriétés suivantes :

- `request` qui contient les informations de la variable globale `$_POST`.
- `query` qui contient les informations de la variable globale `$_GET`.
- `cookies` qui contient les informations de la variable globale `$_COOKIE`.
- `attributes` qui contient des propriétés propres à l'application.
- `files` qui contient les informations de la variable globale `$_FILES`.
- `server` qui contient les informations de la variable globale `$_SERVER`.
- `headers` qui contient les informations sur les en-têtes qui se trouvent pour la plupart sur la variable globale `$_SERVER`.

Tous ces objets sont des instances des classes `ParameterBag`, `InputBag`, `FileBag`, `ServerBag` ou `HeaderBag`.

Ces classes contiennent des méthodes permettant d'accéder aux propriétés et de les modifier. Nous verrons toutes ces méthodes en détail au cours de la formation.

Pour donner un exemple, si nous avons une requête HTTP GET avec une `query string`, par exemple `?name=paul` :

```

1  <?php
2
3  $request->query->get('name');

```

Permet d'accéder à la valeur de la `query string` `name` et donc ici de retourner `paul`.

6.3 Code de la vidéo

Créer un nouveau dossier `php`. Dans ce dossier, installez `symfony/var-dumper` et `symfony/http-foundation` :

```
composer require symfony/http-foundation symfony/var-dumper
```

Créez un fichier `index.php` :

```

1  <?php
2
3  use Symfony\Component\HttpFoundation\Request;
4
5  require __DIR__ . '/vendor/autoload.php';
6
7  $request = Request::createFromGlobals();
8
9  dump($request);
10 dd($request->query->get('name'));

```

Lancez ensuite le serveur de développement :

```
php -S localhost:3000
```

Rendez-vous sur `http://localhost:3000/`.

7 L'objet Response du composant HttpFoundation

7.1 L'objet Response

L'objet `Response` permet de créer, modifier et envoyer une réponse HTTP.

7.1.1 Créer une Response

Étudions son constructeur :

```
1  <?php
2
3  public function __construct(?string $content = '', int $status = 200, array $headers = [])
4  {
5      $this->headers = new ResponseHeaderBag($headers);
6      $this->setContent($content);
7      $this->setStatusCode($status);
8      $this->setProtocolVersion('1.0');
9  }
```

Remarquez que tous les arguments sont optionnels et qu'ils ont des valeurs par défaut.

- Le premier argument est le contenu de la réponse HTTP, qui est vide par défaut.
- Le second argument est le statut de la réponse HTTP, qui 200 pour OK par défaut.
- Le troisième argument est un tableau associatif pour les en-têtes, qui est vide par défaut.

Pour créer une réponse vous pouvez donc par exemple faire :

```
1  <?php
2
3  $response = new Response(
4      'Vous n\'êtes pas autorisé à voir cette page',
5      Response::HTTP_FORBIDDEN,
6      ['content-type' => 'text/html']
7  );
```

Remarquez que nous utilisons une constante pour le statut, tous les codes de statut sont disponibles sur la classe `Response` :

```
1  <?php
2  public const HTTP_CONTINUE = 100;
3  public const HTTP_SWITCHING_PROTOCOLS = 101;
4  public const HTTP_PROCESSING = 102;           // RFC2518
5  public const HTTP_EARLY_HINTS = 103;         // RFC8297
6  public const HTTP_OK = 200;
7  public const HTTP_CREATED = 201;
8  public const HTTP_ACCEPTED = 202;
9  public const HTTP_NON_AUTHORITATIVE_INFORMATION = 203;
10 public const HTTP_NO_CONTENT = 204;
11 public const HTTP_RESET_CONTENT = 205;
12 public const HTTP_PARTIAL_CONTENT = 206;
13 public const HTTP_MULTI_STATUS = 207;        // RFC4918
```

```

14  public const HTTP_ALREADY_REPORTED = 208;      // RFC5842
15  public const HTTP_IM_USED = 226;              // RFC3229
16  public const HTTP_MULTIPLE_CHOICES = 300;
17  public const HTTP_MOVED_PERMANENTLY = 301;
18  public const HTTP_FOUND = 302;
19  public const HTTP_SEE_OTHER = 303;
20  public const HTTP_NOT_MODIFIED = 304;
21  public const HTTP_USE_PROXY = 305;
22  public const HTTP_RESERVED = 306;
23  public const HTTP_TEMPORARY_REDIRECT = 307;
24  public const HTTP_PERMANENTLY_REDIRECT = 308;  // RFC7238
25  public const HTTP_BAD_REQUEST = 400;
26  public const HTTP_UNAUTHORIZED = 401;
27  public const HTTP_PAYMENT_REQUIRED = 402;
28  public const HTTP_FORBIDDEN = 403;
29  public const HTTP_NOT_FOUND = 404;
30  public const HTTP_METHOD_NOT_ALLOWED = 405;
31  public const HTTP_NOT_ACCEPTABLE = 406;
32  public const HTTP_PROXY_AUTHENTICATION_REQUIRED = 407;
33  public const HTTP_REQUEST_TIMEOUT = 408;
34  public const HTTP_CONFLICT = 409;
35  public const HTTP_GONE = 410;
36  public const HTTP_LENGTH_REQUIRED = 411;
37  public const HTTP_PRECONDITION_FAILED = 412;
38  public const HTTP_REQUEST_ENTITY_TOO_LARGE = 413;
39  public const HTTP_REQUEST_URI_TOO_LONG = 414;
40  public const HTTP_UNSUPPORTED_MEDIA_TYPE = 415;
41  public const HTTP_REQUESTED_RANGE_NOT_SATISFIABLE = 416;
42  public const HTTP_EXPECTATION_FAILED = 417;
43  public const HTTP_I_AM_A_TEAPOT = 418;        // RFC2324
44  public const HTTP_MISDIRECTED_REQUEST = 421;  // RFC7540
45  public const HTTP_UNPROCESSABLE_ENTITY = 422;  // RFC4918
46  public const HTTP_LOCKED = 423;              // RFC4918
47  public const HTTP_FAILED_DEPENDENCY = 424;    // RFC4918
48  public const HTTP_TOO_EARLY = 425;           // RFC-ietf-httpbis-replay-04
49  public const HTTP_UPGRADE_REQUIRED = 426;     // RFC2817
50  public const HTTP_PRECONDITION_REQUIRED = 428; // RFC6585
51  public const HTTP_TOO_MANY_REQUESTS = 429;   // RFC6585
52  public const HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE = 431; // RFC6585
53  public const HTTP_UNAVAILABLE_FOR_LEGAL_REASONS = 451;
54  public const HTTP_INTERNAL_SERVER_ERROR = 500;
55  public const HTTP_NOT_IMPLEMENTED = 501;
56  public const HTTP_BAD_GATEWAY = 502;
57  public const HTTP_SERVICE_UNAVAILABLE = 503;
58  public const HTTP_GATEWAY_TIMEOUT = 504;
59  public const HTTP_VERSION_NOT_SUPPORTED = 505;
60  public const HTTP_VARIANT_ALSO_NEGOTIATES_EXPERIMENTAL = 506; // RFC2295
61  public const HTTP_INSUFFICIENT_STORAGE = 507; // RFC4918

```

```

62 public const HTTP_LOOP_DETECTED = 508; // RFC5842
63 public const HTTP_NOT_EXTENDED = 510; // RFC2774
64 public const HTTP_NETWORK_AUTHENTICATION_REQUIRED = 511;

```

Cela permet une meilleure lisibilité du code et de ne pas se tromper sur le statut.

7.1.2 Modifier une Response

La classe possède également des propriétés publiques qui sont modifiables avec des setters.

Il est ainsi possible de modifier un objet Response une fois que celui-ci a été créé.

Pour rappel, un setter est une méthode permettant de modifier une propriété sur un objet.

Par exemple, pour modifier le contenu de la réponse :

```

1 <?php
2 $response->setContent('Hello World !');

```

Même chose pour les en-têtes :

```

1 <?php
2 $response->headers->set('Content-Type', 'text/plain');

```

Et pour le code de statut :

```

1 <?php
2 $response->setStatusCode(Response::HTTP_NOT_FOUND);

```

7.1.3 Envoyer la Response

Une fois que l'objet Response est prêt à être envoyé au client, il suffit d'appeler la méthode `send()` :

```

1 <?php
2 $response->send();

```

7.2 Les réponses au format JSON

Bien qu'il soit possible d'utiliser l'objet Response pour envoyer une réponse au format JSON, ce n'est pas pratique car il faut manuellement définir l'entête `Content-Type` et encoder l'entité PHP au format JSON.

Le composant `HttpFoundation` a donc une classe spécifique qui permet de faire cela automatiquement pour vous : `JsonResponse`.

Vous pouvez ainsi directement faire :

```

1 <?php
2 use Symfony\Component\HttpFoundation\JsonResponse;
3
4 $response = new JsonResponse(['prenom' => 'Jean']);

```

7.3 Code de la vidéo : dossier php

Dans le fichier `index.php` mettez le code suivant :

```
1  <?php
2
3  use Symfony\Component\HttpFoundation\Response;
4
5  require __DIR__ . '/vendor/autoload.php';
6
7  $response = new Response('<h2>First !</h2><h1>Hello World ! </h1>');
8
9  $response->headers->set('salut', 'ca va');
10
11 $response->send();
```

Lancez ensuite le serveur de développement :

```
php -S localhost:3000
```

Rendez-vous sur `http://localhost:3000/`.

7.4 Code de la vidéo : dossier dymaproject

De retour sur le projet Symfony, nous modifions notre contrôleur pour récupérer l'objet `Request` créé par Symfony dans l'action `index` :

```
1  <?php
2
3  namespace App\Controller;
4
5  use Symfony\Component\HttpFoundation\Request;
6  use Symfony\Component\HttpFoundation\Response;
7
8  class DefaultController
9  {
10     public function index(Request $request)
11     {
12         dd($request);
13         return new Response('<h1>Hello World !</h1>');
14     }
15 }
```

Symfony passe automatiquement l'objet `Request`, qu'il crée automatiquement, en argument des actions.

Pour rappel, les actions sont les méthodes sur les classes des contrôleurs qui sont exécutées lorsqu'une requête HTTP correspond à une route.

Notez bien la propriété `attributes` sur l'objet requête :


```

^ Symfony\Component\HttpFoundation\Request {#52 ▼
+attributes: Symfony\Component\HttpFoundation\ParameterBag {#95 ▼
  #parameters: array:3 [▼
    "_route" => "index"
    "_controller" => "App\Controller\DefaultController::index"
    "_route_params" => []
  ]
}

```

Comme nous l'avons vu, il s'agit d'une instance de la classe `ParameterBag` qui contient des paires clé / valeur créées par l'application.

Dans ces paires nous retrouvons, comme prévu, le nom de la route dans `_route` et le chemin vers le contrôleur, et plus précisément son action, à exécuter dans `_controller`.