Mise en place d'une API REST

La mise en place d'une API REST avec Symfony est une chose relativement aisée dans la mesure où le framework propose nativement tous les outils nécessaires. Nous avons déjà vu au cours de cet ouvrage le développement des contrôleurs et de leurs actions et comment les associer à des URL ; cela permet de structurer l'API.

De plus, la prise en charge du format JSON pour l'échange des données nous permet de facilement gérer ce format dans la requête comme dans la réponse.

1. Le service serializer

Avant de nous permettre de manipuler des données en JSON, il est nécessaire d'installer le sérialiseur Symfony. Ce sérialiseur sera ensuite disponible sous forme d'un service injectable dans les classes de votre application.

Pour installer ce service nommé **serializer**, il faut exécuter une recette Flex avec la commande suivante :

composer require symfony/serializer-pack

Une fois installé, le service est injectable dans les actions de contrôleur via l'interface **Symfony\Component\Serializer\SerializerInterface** :

```
namespace App\Controller;


use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Serializer\SerializerInterface;


class DefaultController extends AbstractController
{
  public function index(SerializerInterface $serializer)
  {
     ...
  }
}
```

a. Sérialiser des données

Pour sérialiser des données en JSON, nous allons utiliser la méthode **serialize()**du service **serializer**. Cette méthode prend en premier paramètre la variable représentant

les données à sérialiser et en second paramètre le format de sérialisation. On pourra ainsi facilement sérialiser en JSON une entité Doctrine récupérée en amont, comme dans l'exemple de code ci-dessous :

```
public function index(SerializerInterface $serializer)
{
    $article = // Récupération d'une entité Article avec Doctrine...
    $jsonArticle = $serializer->serialize($article, 'json');
}
```

b. Désérialiser des données

Pour réaliser l'opération inverse, on utilise la méthode **deserialize()** du même service. Cette méthode prend trois paramètres :

- le flux de données à désérialiser ;
- le type d'objet dans lequel désérialiser les données ;
- le format de données reçu.

Par exemple, en supposant que le flux de données JSON entrant soit représentatif d'une entité **Article**, nous pouvons désérialiser le flux de la manière suivante :

```
public function index(SerializerInterface $serializer)
{
  $json = // Récupération du flux JSON entrant...
  $article = $serializer->deserialize(
                $json,
                Article::class,
                'json'
              );
}
```

La variable **$article** est alors une référence sur un objet de type **Article**.

2. Adaptation des contrôleurs

Les contrôleurs que nous avons développés jusqu'à présent avaient tous pour objectif final de faire le rendu d'une vue. Dans le contexte d'une API REST, ils vont devoir retourner un flux de données au travers d'un

objet **Symfony\Component\HttpFoundation\Response**, il n'y aura donc pas d'usage de la méthode **render()** pour la génération d'un template Twig.

Concernant la mise en place de l'API, nous allons ici évidemment utiliser l'attribut **#[Route]** afin d'associer les actions à des URL. L'attribut **methods** sera systématiquement présent afin de spécifier quelle méthode HTTP devra être utilisée pour solliciter l'action.

```
#[Route(
  '/api/articles',
  name: 'api_article_tous',
  methods: ['GET']
)]
public function tous(): Response
{
  ...
}
```

Tout le principe d'exposition de fonctionnalités dans une API REST repose donc sur l'usage de cet attribut.