

# Les contrôleurs

# Table des matières

<b>I. Comprendre le contrôleur</b>	<b>3</b>
A. Le contrôleur de base .....	3
B. Les Services .....	5
<b>II. Exercice : Quiz</b>	<b>7</b>
<b>III. Création du contrôleur</b>	<b>8</b>
A. Créer une classe contrôleur à la main .....	8
B. Utiliser MakerBundle .....	9
<b>IV. Exercice : Quiz</b>	<b>12</b>
<b>V. Essentiel</b>	<b>12</b>
<b>VI. Auto-évaluation</b>	<b>13</b>
A. Exercice .....	13
B. Test .....	13
<b>Solutions des exercices</b>	<b>14</b>

# I. Comprendre le contrôleur

**Durée :** 1 H

**Environnement de travail :** PC et connexion internet

**Pré-requis :** notion de PHP, Notion de POO

## Contexte

Les contrôleurs font partie des éléments clef de l'architecture MVC (Modèle-Vue-Contrôleur) qui est aujourd'hui omniprésente dans la POO (Programmation-Orientée-Objet) pour les applications web modernes. Cette structure architecturale a pour principe de permettre la séparation entre les données et les méthodes qui les utilisent. Ainsi, grâce au MVC, il est possible de permettre une meilleure collaboration entre les développeurs front-end et back-end au sein d'une même application.

Le contrôleur joue donc son rôle entre la Vue et le Modèle. Les « *Vues* », qui représentent l'affichage des pages, seront généralement développées en Twig pour un projet Symfony. Quant aux « *Modèles* », ils sont chargés de gérer les données et leur persistance. C'est ainsi que, souvent, un modèle sera associé à une base de données, même si cela n'est pas obligatoire, ou pourra interagir avec un API externe.

On comprend ainsi que maîtriser tout le fonctionnement du contrôleur est essentiel pour un développeur d'application web ou d'API (*Application Programming Interface* ou « *Interface de Programmation d'Application* » en français). Dans ce cours, nous répondrons aux questions suivantes : Que peut-on mettre dans un contrôleur ? Comment lui attribuer une route avec Symfony 5.4 ou supérieur ? Qu'est-ce qu'un contrôleur peut retourner ? Comment faire en sorte qu'il ne soit pas surchargé ?

## A. Le contrôleur de base

### Exemple

Voici un exemple de contrôleur classique :

```
1 <?php
2
3 namespace App\Controller;
4
5 use App\Repository\PicturesRepository;
6 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
7 use Symfony\Component\HttpFoundation\Response;
8 use Symfony\Component\Routing\Annotation\Route;
9
10 class PicturesController extends AbstractController
11 {
12     /**
13      * @Route('/', name: 'app_pictures_index', methods: ['GET'])
14      */
15     public function index(PicturesRepository $picturesRepository): Response
16     {
17         $allPictures = $picturesRepository->findAll();
18         return $this->render('pictures/index.html.twig', [
19             'pictures' => $allPictures
20         ]);
21     }
22 }
```

Sur la première ligne, vous pouvez constater l'utilisation des namespaces, espace de noms en français, fonctionnalité issue de PHP via la librairie SPL et ses fonctions d'autoload (également intégrée dans Symfony via Composer).

Il faut donc indiquer le chemin d'accès après le terme `namespace`, dans l'exemple, vous le retrouvez sous `App\Controller` qui est un alias du dossier `\src`. On retrouve cette configuration dans le fichier de configuration « `composer.json` ».

- `use` appelle les classes dont nous avons besoin pour faire fonctionner ce contrôleur, suivi du chemin d'accès pour les retrouver. Ne vous inquiétez pas, souvent, votre « *ide* » (environnement de développement intégré) est le logiciel de création d'applications que vous utilisez pour programmer peut vous faciliter les choses. Pensez donc à installer tous les plugins gratuits relatifs à PHP et Symfony pour disposer d'une complétion automatique fonctionnelle.
- La classe de ce contrôleur se nomme `PicturesController`. Notez que, par convention, toutes les classes de contrôleur doivent terminer leur nom par « *Controller* ».
- Cette classe hérite d'`AbstractController`. Cela signifie que, comme toute classe `Controller`, elle doit hériter de la classe mère et abstraite `AbstractController`, délivrée par Symfony pour disposer d'une panoplie de fonctionnalités.
- La méthode qui est déclarée avec `public function` se nomme « *index* » et appelle, grâce à ses paramètres et l'injection de dépendance (que l'on verra plus tard), la classe `PicturesRepository` et retournera un objet de type `Response` fourni par le composant mère `HttpFoundation`.
- La variable `$allPictures` récupère toutes les images enregistrées dans la base de données grâce à la méthode `findAll`, proposée par Doctrine.
- Dans le retour, on demande à Symfony d'écrire une réponse http, contenant dans son corps le HTML résultant de la vue `index.html.twig` puis on fait passer l'objet `$allPictures` que l'on pourra appeler dans Twig sous l'appellation `pictures`.

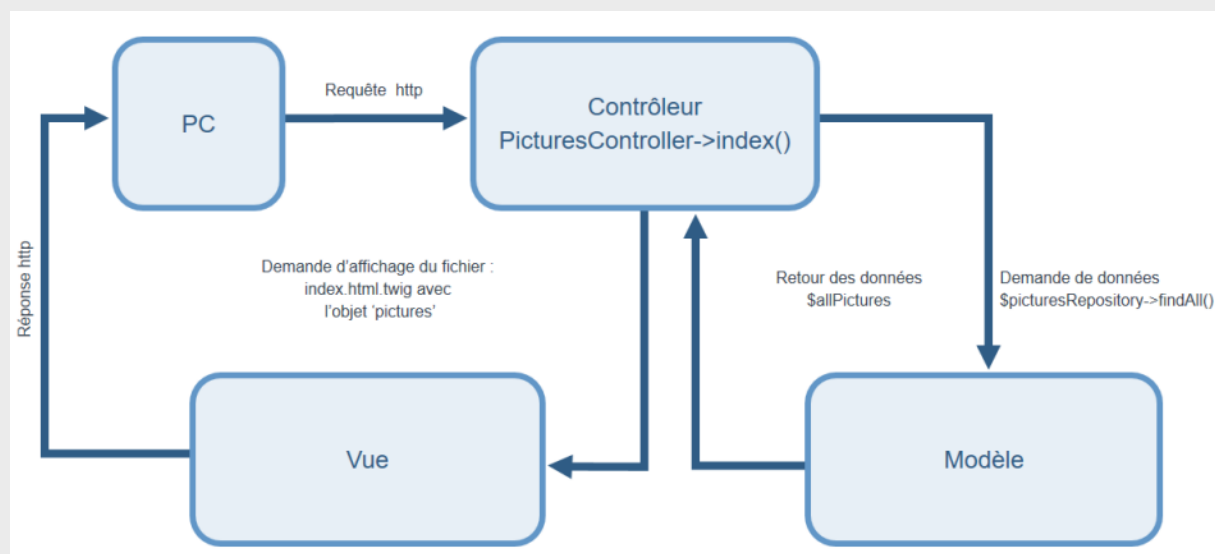


Schéma du contrôleur

Le schéma ci-dessus résume bien ce qui se passe dans notre contrôleur. Une requête HTTP est demandée, le contrôleur `PicturesController` avec la méthode `index()` interroge la base de données qui lui retourne un résultat dans la variable `$allPictures`. Ensuite, le contrôleur demande l'affichage de la page `index.html.twig` en lui passant le résultat de la requête dans la variable `$pictures`. La réponse sera alors générée par la méthode `render` qui transformera le langage Twig en pure HTML et sera insérée dans le body de la réponse http retournée au client.

## Les routes

Notez également que dans notre code, nous avons mappé une URL avec le contrôleur. Cela s'est effectué avec l'annotation : `@Route('/', name: 'app_pictures_index', methods: ['GET'])`.

Récemment, via PHP 8, une nouvelle notion est née, le PHP Attribute, que l'on retrouve notamment à partir de la version 5.4 et 6 de Symfony. Cela permet de déclarer de la métadonnée directement compris par PHP, sous cette forme `#[Route('/', name: 'app_pictures_index', methods: ['GET'])]` sans avoir besoin d'utiliser les astérisques \*. Le routing est un sujet à part entière.

## AbstractController

Symfony propose par défaut d'étendre le contrôleur avec `AbstractController`. Cette classe va fournir diverses méthodes qui permettront de rendre un template, d'effectuer une redirection, de gérer des erreurs et bien d'autres choses encore. `AbstractController`, sous Symfony 5.4 et PHP 8.1, propose ces différentes méthodes :

- `setContainer()`
- `getParameter()`
- `getSubscribedservices()`
- `generateUrl()`
- `forward()`
- `redirect()`
- `redirectToRoute()`
- `json()`
- `file()`
- `addFlash`
- `isGranted`
- `denyAccessUnlessGranted`
- `renderView()`
- `render()`
- `renderForm()`
- `stream()`
- `createAccessDeniedException()`
- `createForm()`
- `createFormBuilder`
- `getUser()`
- `isCsrfTokenValid()`
- `addLink()`

## B. Les Services

Les Services sont des classes qui vont pouvoir être appelées partout dans notre code. Ils sont très utiles pour les contrôleurs, car cela leur permet d'ajouter de nombreuses fonctionnalités sans pour autant les surcharger. Certains services sont proposés par Symfony, mais vous pouvez en créer autant que vous avez besoin.

## Récupération des services

Symfony propose de nombreux services qu'il vous est possible d'utiliser. Prenons l'exemple de `loggerInterface` qui permet de stocker par défaut des informations dans le fichier « *log* » situé dans le dossier « *var* ».

Grâce à l'injection de dépendances, on ajoute la classe `LoggerInterface` et on lui donne un nom de variable dans les paramètres de la fonction. Bien sûr, il ne faut pas oublier de rajouter le namespace dans les « *use* ». Ensuite, on appelle `$log` et lui ajoute la méthode ici `info()`, mais sachez que `LoggerInterface` propose aussi les méthodes: `debug`, `warning`, `notice`, `error`, `critical`, `alert`, `emergency`.

## Injection de dépendances

L'injection de dépendances est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle. Injecter, c'est créer dynamiquement la dépendance.

Pour faire simple, c'est un peu comme si on avait une grosse valise pleine de services déjà instanciés et qui sont prêts à être utilisés à la demande.

Ainsi, grâce à ce concept, il n'est plus nécessaire de créer ses dépendances manuellement. Dans le framework Symfony, l'injection de dépendances est réalisée via le *Container* de services, qui est construit par le *ContainerBuilder*. Celui-ci est initialisé par le *Kernel*.

```
1 class PicturesController extends AbstractController
2 {
3     /**
4      * @Route('/', name: 'app_pictures_index', methods: ['GET'])
5      */
6     public function index(PicturesRepository $picturesRepository, LoggerInterface $log):
7         Response
8     {
9         $allPictures = $picturesRepository->findAll();
10        $log->info("j'ai récupéré toutes les images");
11
12        return $this->render('pictures/index.html.twig', [
13            'pictures' => $allPictures
14        ]);
15    }
16 }
```

Si vous souhaitez connaître tous les services que propose Symfony, tapez la ligne de commande suivante :

```
1 $ php bin/console debug:autowiring
```

## Création de services

Mais vous pouvez aussi créer vos propres Services. Pour cela, vous devez créer un fichier qui contiendra le nom « *services* », par exemple « *UserServices* », le même nom que votre classe. À l'intérieur de cette classe, vous pourrez mettre autant de méthode que vous le souhaitez. Après, grâce à l'injection de dépendances et au namespace, vous pourrez l'utiliser dans votre contrôleur.

Toutefois, pour être certain que le chargement de service se fasse automatiquement, vérifiez le fichier « *services.yaml* » et assurez-vous d'avoir les deux variables `autowire` et `autoconfigure` à `true`. Ce sont elles qui permettent à Symfony d'instancier vos classes de Service automatiquement et d'aller chercher d'autres dépendances.

Voici comment ce fichier doit être par défaut sous Symfony 5.4 :

```
1 # config/services.yaml
2 services:
3     # default configuration for services in *this* file
4     _defaults:
5         autowire: true      # Automatically injects dependencies in your services.
```

```

6         autoconfigure: true # Automatically registers your services as commands, event
subscribers, etc.
7
8     # makes classes in src/ available to be used as services
9     # this creates a service per class whose id is the fully-qualified class name
10    App\:
11        resource: '../src/'
12        exclude:
13            - '../src/DependencyInjection/'
14            - '../src/Entity/'
15            - '../src/Kernel.php'
16
17    # order is important in this file because service definitions
18    # always *replace* previous ones; add your own service configuration below
19
20    # ...

```

### Les composants HTTPFoundation

Les accès à une application web se font par une requête HTTP. Cette requête contient un en-tête et un body. Ces éléments contiennent de nombreuses informations. La réponse du serveur renverra aussi une syntaxe similaire. Grâce aux classes `Request` et `Response` de `HttpFoundation`, il est possible de récupérer ces informations dans le contrôleur. Pour utiliser ces classes, il faudra procéder de la même manière que pour les Services, c'est-à-dire insérer dans les paramètres de la méthode du contrôleur `Request` et `Response` avec un nom de variable, souvent, vous retrouverez `$request` et `$response` (pensez toujours à l'ajouter au namespace utilisé).

### Le retour du contrôleur

Au début de ce chapitre, nous avons parlé du MVC, qui est l'architecture la plus utilisée pour une application web, mais dans le cas d'une API (Application Programming Interface). Par exemple, le contrôleur ne retournera pas une vue, mais des données. Ainsi, un objet `Responses` peut renvoyer une page HTML, JSON, XML, un téléchargement de fichier, une redirection, une erreur ou autre. Pour associer des données, il sera nécessaire de rajouter en paramètres à la méthode un nom de variable et de définir à quoi elle correspond.

```

1 public function listAction(EntityManagerInterface $em)
2 {
3     return $this->render('task/list.html.twig', ['tasks' => $em->getRepository(Task::class)-
>findAll()]);
4 }

```

Dans l'exemple ci-dessus, la méthode `render()` issue d'`AbstractController` retourne la vue ainsi que la variable `tasks` qui est un tableau de toutes les `tasks` (tâche, en français).

Arrêtons-nous sur le JSON, qui est largement utilisé et donc retourné. Prenons un exemple très simple où la méthode de notre contrôleur va nous retourner une adresse email en JSON. Il faudra cette fois appeler la classe `JsonResponse` ainsi que la méthode d'`AbstractController` `json` avec `$this`, préciser que le type de retour est du `JsonResponse` et enfin le retourner avec un `return $this->json()` comme dans l'exemple ci-dessous.

```

1 use Symfony\Component\HttpFoundation\JsonResponse;
2 // ...
3 public function index(): JsonResponse
4 {
5     return $this->json(['email' => 'daniel.martin@free.fr']);
6 }

```

## Exercice : Quiz

[solution n°1 p.15]

### Question 1

L'injection de dépendances permet d'utiliser dynamiquement les classes qui sont injectées :

- ☐ Vrai
- ☐ Faux

#### Question 2

Les services sont des classes PHP.

- ☐ Vrai
- ☐ Faux

#### Question 3

Un contrôleur retourne toujours vers un fichier twig.

- ☐ Vrai
- ☐ Faux

#### Question 4

Pour récupérer des informations d'une requête, il faut appeler `Request` `HTTPFoundation` par le namespace.

- ☐ Vrai
- ☐ Faux

#### Question 5

Le contrôleur ne peut pas retourner d'objet pour un template.

- ☐ Vrai
- ☐ Faux

## III. Création du contrôleur

Il existe plusieurs façons de créer un contrôleur : en créant un fichier PHP dédié à une classe contrôleur ou en utilisant les lignes de commandes.

### A. Créer une classe contrôleur à la main

Symfony est un framework très structuré et qui possède de nombreuses conventions. Celles-ci permettent aux développeurs de développer en utilisant de bonnes pratiques.

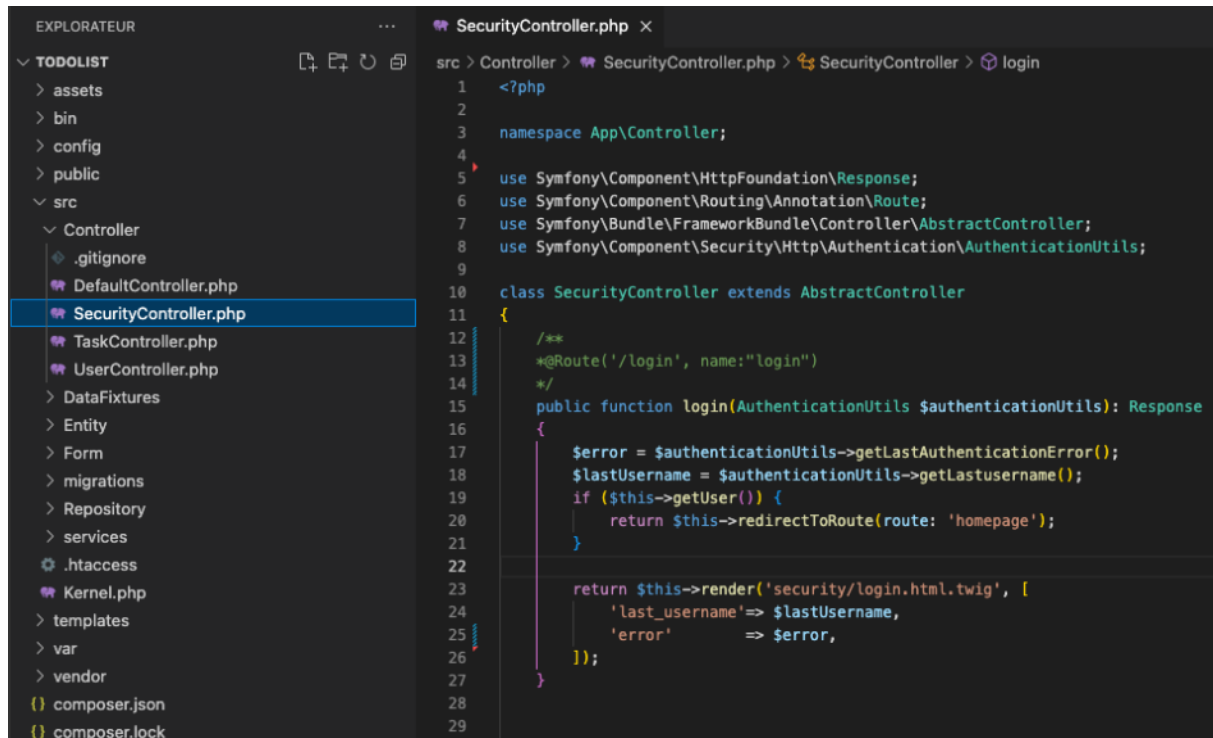
Ainsi, si l'on souhaite créer une classe contrôleur « à la main » dans un fichier PHP, il conviendra de le placer dans le dossier « *src* » puis dans un dossier « *controller* ».

Il convient également de créer une classe contrôleur par fichier, et celui-ci portera le même nom que la classe. À l'intérieur de la classe, vous pouvez utiliser autant de méthode que vous le souhaitez.

Nous vous conseillons d'utiliser Codacy, SymfonyInsight ou encore CodeClimate pour auditer la qualité de votre code. En effet, ils permettent de détecter une fonction surchargée de votre contrôleur.

Votre contrôleur devra posséder une route, qui sera décrite par un système d'annotation, et renvoyer une réponse que vous pouvez typer après les paramètres de la méthode. Dans les paramètres justement, vous pouvez ajouter des arguments, comme dans n'importe quelle méthode, ainsi que faire de l'injection de dépendances.





Exemple de contrôleur avec son arborescence

## B. Utiliser MakerBundle

Symfony Maker est un pack qui vous aide à créer des contrôleurs, des formulaires, des tests et bien plus encore.

Tout d'abord, assurez-vous que le bundle soit installé en consultant le fichier « *composer.json* ». Si ce n'est pas le cas, tapez la ligne de commande suivante :

```
1 $ composer require --dev symfony/maker-bundle
```

Si vous souhaitez connaître toutes les fonctionnalités qu'apporte Maker-bundle, tapez la ligne de commande suivante :

```
1 $ php bin/console list make
```

Maintenant créons un contrôleur avec la ligne de commande suivante :

```
1 $ php bin/console make:controller NewController
```

Qu'est-ce qu'il se passe ? Maker vient de créer un controller de base dans le dossier controller qui se nomme « *NewController* » puisque nous l'avons nommé ainsi. Bien sûr, vous devez modifier son nom dans la ligne de commande pour pouvoir le personnaliser suivant vos besoins.

Notez que MakerBundle génère les routes avec PHP attributes et non pas les annotations. Voici ci-dessous le contrôleur généré que vous devrez personnaliser.

```
1 <?php
2
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Routing\Annotation\Route;
8
9 class NewController extends AbstractController
10 {
```

```

11  #[Route('/new', name:'app_new')]
12  public function index(): Response
13  {
14      return $this->render('new/index.html.twig', [
15          'controller_name' => 'NewController',
16      ]);
17  }
18 }

```

Maintenant, allons encore plus loin en créant un contrôleur à partir d'une Entité. Cet ORM de Doctrine permet de gérer tous les éléments de la base de données que nous avons créé à partir d'une Entité.

Le but va être de créer un CRUD (*Create, Read, Update, Delete*), c'est-à-dire les fonctions de base qui permettent de créer, de récupérer, de mettre à jour ou de supprimer des objets.

Pour cela, utilisons la ligne de commande suivante qui va utiliser une Entité déjà créé qui se nomme Figure :

```
1 $ php bin/console make:crud Figure
```

Maker va vous demander comment vous voulez nommer le nouveau contrôleur. Il vous proposera par défaut de rajouter « *Controller* » au nom de votre Entité. Puis, il vous demandera si vous souhaitez générer des tests pour votre contrôleur. Vous pouvez répondre « *no* », surtout si vous ne maîtrisez pas encore les tests, mais tester ses contrôleurs est une excellente pratique à adopter.

Voici le code que Symfony nous a généré dans le nouveau contrôleur qui se nomme « *FigureController* » :

```

1 <?php
2
3 namespace App\Controller;
4
5 use App\Entity\Figure;
6 use App\Form\FigureType;
7 use App\Repository\FigureRepository;
8 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
9 use Symfony\Component\HttpFoundation\Request;
10 use Symfony\Component\HttpFoundation\Response;
11 use Symfony\Component\Routing\Annotation\Route;
12
13 #[Route('/figure')]
14 class FigureController extends AbstractController
15 {
16     #[Route('/', name: 'app_figure_index', methods: ['GET'])]
17     public function index(FigureRepository $figureRepository): Response
18     {
19         return $this->render('figure/index.html.twig', [
20             'figures' => $figureRepository->findAll(),
21         ]);
22     }
23
24     #[Route('/new', name: 'app_figure_new', methods: ['GET', 'POST'])]
25     public function new(Request $request, FigureRepository $figureRepository): Response
26     {
27         $figure = new Figure();
28         $form = $this->createForm(FigureType::class, $figure);
29         $form->handleRequest($request);
30
31         if ($form->isSubmitted() && $form->isValid()) {
32             $figureRepository->add($figure, true);
33
34             return $this->redirectToRoute('app_figure_index', [], Response::HTTP_SEE_OTHER);
35         }

```

```

36
37     return $this->renderForm('figure/new.html.twig', [
38         'figure' => $figure,
39         'form' => $form,
40     ]);
41 }
42
43 #[Route('/{id}', name: 'app_figure_show', methods: ['GET'])]
44 public function show(Figure $figure): Response
45 {
46     return $this->render('figure/show.html.twig', [
47         'figure' => $figure,
48     ]);
49 }
50
51 #[Route('/{id}/edit', name: 'app_figure_edit', methods: ['GET', 'POST'])]
52 public function edit(Request $request, Figure $figure, FigureRepository $figureRepository):
    Response
53 {
54     $form = $this->createForm(FigureType::class, $figure);
55     $form->handleRequest($request);
56
57     if ($form->isSubmitted() && $form->isValid()) {
58         $figureRepository->add($figure, true);
59
60         return $this->redirectToRoute('app_figure_index', [], Response::HTTP_SEE_OTHER);
61     }
62
63     return $this->renderForm('figure/edit.html.twig', [
64         'figure' => $figure,
65         'form' => $form,
66     ]);
67 }
68
69 #[Route('/{id}', name: 'app_figure_delete', methods: ['POST'])]
70 public function delete(Request $request, Figure $figure, FigureRepository
    $figureRepository): Response
71 {
72     if ($this->isCsrfTokenValid('delete.'.$figure->getId(), $request->request-
    >get('_token')) {
73         $figureRepository->remove($figure, true);
74     }
75
76     return $this->redirectToRoute('app_figure_index', [], Response::HTTP_SEE_OTHER);
77 }
78 }

```

Nous n'allons pas tout décrire, mais notez que cinq méthodes ont été créées :

- `index()`, dont le but est de retourner dans la vue aussi créée (`figure/index.html.twig`) toutes les figures de la base de données.
- `new()`, qui utilise la méthode `Post` et donc permet de créer une nouvelle `Figure` à partir d'un formulaire. Formulaire qui a d'ailleurs été ajouté dans le dossier « *Form* » sous le nom de « *FigureType* ».
- `show()`, dont le but est de retourner une figure en détail grâce à l'id passé en paramètre de l'url.
- `edit()`, qui va permettre de modifier une figure précise grâce à l'id.
- `delete()`, qui, comme son nom l'indique, va permettre au contrôleur de supprimer une figure.

Pour résumer, avec la ligne de commande `php bin/console make:crud Figure`, Symfony a créé un nouveau contrôleur « *FigureController* » avec cinq méthodes essentielles et tout ce qui est nécessaire pour fonctionner.

## Exercice : Quiz

[solution n°2 p.15]

### Question 1

Que signifie CRUD ?

- ☐ Create, Request, Update, Delete
- ☐ Create, Reset, Update, Delete
- ☐ Create, Read, Update, Delete

### Question 2

Les contrôleurs doivent être stockés par convention dans un dossier controller qui est lui-même dans le dossier « *src* ».

- ☐ Vrai
- ☐ Faux

### Question 3

Avec une seule ligne de commande, il est possible dans Symfony de créer un contrôleur qui est totalement opérationnel avec de nombreuses méthodes.

- ☐ Vrai
- ☐ Faux

### Question 4

Il est possible d'utiliser la méthode PUT dans un contrôleur.

- ☐ Vrai
- ☐ Faux

### Question 5

Je n'ai pas besoin de Doctrine pour créer un CRUD avec MakerBundle.

- ☐ Vrai
- ☐ Faux

## V. Essentiel

MVC (Modèle-Vue-Contrôleur) est un modèle de conception de logiciel qui permet la séparation entre la logique métier et l'affichage de l'application. Symfony est taillé pour cela et le contrôleur est le cœur, le moteur de l'application.

Grâce à ce cours, vous avez pu découvrir quel est le rôle d'un contrôleur, comment celui-ci fonctionne et s'articule, quelles sont les conventions à respecter et quels outils peuvent nous faciliter la tâche.

Mais nous n'avons abordé que le fonctionnement essentiel de ce moteur : plus vous travaillerez avec et plus vous découvrirez ses performances et pourquoi il est si utile.

Ce qui est important à retenir, c'est que le contrôleur utilise un routing d'où va lui venir une requête. Pour y répondre, il va se référer au Modèle, qui est chargé de gérer la base de données et sa persistance, à l'aide d'outils que l'on nomme la plupart du temps sous l'appellation Service, qu'ils viennent de PHP, Symfony ou d'autres classes. La réponse pourra être un template auquel il est possible de joindre un ou des objets, une redirection, du JSON, etc. Mais son travail est de donner une réponse.

N'oubliez pas qu'il existe des bundles qui peuvent véritablement vous faciliter le travail et vous faire gagner du temps, parce qu'il vont exploiter pleinement la puissance de Symfony et de l'ORM Doctrine, comme avec MarkerBundle que nous avons décrit dans ce cours.

## VI. Auto-évaluation

### A. Exercice

Vous êtes un développeur back-end expérimenté dans une agence web où vous gérez de nombreux projets. On vous demande de créer une application web simple avec Symfony 5.4 dans de très brefs délais.

#### Question 1

[solution n°3 p.17]

Comment est-ce que vous allez vous y prendre pour créer les contrôleurs et pourquoi ?

#### Question 2

[solution n°4 p.17]

Expliquez pourquoi il est important de créer des services et décrivez techniquement comment vous allez vous y prendre.

### B. Test

#### Exercice 1 : Quiz

[solution n°5 p.17]

##### Question 1

Le contrôleur s'occupe de tout, il est donc forcément volumineux.

- ☐ Vrai
- ☐ Faux

##### Question 2

Le contrôleur est effectivement au cœur de l'action mais pour plus de lisibilité, il doit être le plus concis possible. On peut faire cela en créant des services.

- ☐ Vrai
- ☐ Faux

##### Question 3

Il est obligatoire d'ajouter `extends AbstractController` au contrôleur.

- ☐ Vrai
- ☐ Faux

##### Question 4

Un objet et une classe sont la même chose.

- ☐ Vrai
- ☐ Faux

##### Question 5

Quel nom est le plus approprié pour un contrôleur ?

- ☐ Controllerproduct
- ☐ controllerproduct
- ☐ ControllerProduct
- ☐ ProductController


### **Solutions des exercices**

**Exercice p. 7 Solution n°1****Question 1**

L'injection de dépendances permet d'utiliser dynamiquement les classes qui sont injectées :

☒ Vrai

☐ Faux


 L'injection de dépendance permet de résoudre la problématique de communication entre les classes. PHP a intégré cette technologie, mais elle est courante dans la programmation informatique.

**Question 2**

Les services sont des classes PHP.

☒ Vrai

☐ Faux


 En POO, les classes sont des comme des briques et les services qui sont des classes sont des briques qui sont installées dans notre projet, par PHP, Symfony, un bundle ou un développeur.

**Question 3**

Un contrôleur retourne toujours vers un fichier twig.

☐ Vrai

☒ Faux


 Le retour d'un contrôleur peut être un template, un fichier PDF, du JSON, XML ou autre.

**Question 4**

Pour récupérer des informations d'une requête, il faut appeler `Request::HTTPFoundation` par le namespace.

☒ Vrai

☐ Faux


 Vrai, il faudra aussi créer une variable pour l'appeler.

**Question 5**

Le contrôleur ne peut pas retourner d'objet pour un template.

☐ Vrai

☒ Faux


 Le contrôleur peut retourner une vue et lui associer une variable en lui donnant un nom et l'objet, la variable ou le tableau auquel il correspond. Exemple : `return $this->render('pictures/index.html.twig', [ 'pictures' => $allPictures ]);`

**Exercice p. 12 Solution n°2**

### Question 1

Que signifie CRUD ?


- ☐ Create, Request, Update, Delete
- ☐ Create, Reset, Update, Delete
- ☒ Create, Read, Update, Delete

 CRUD est un moyen mnémotechnique pour retenir les quatre méthodes de base pour gérer la base de données.

### Question 2

Les contrôleurs doivent être stockés par convention dans un dossier controller qui est lui-même dans le dossier « *src* ».


- ☒ Vrai
- ☐ Faux

 Depuis Symfony 4, il convient par convention de regrouper tous les contrôleurs dans le dossier « *src* ». Ce dossier doit d'ailleurs contenir tout le code PHP de l'application.

### Question 3

Avec une seule ligne de commande, il est possible dans Symfony de créer un contrôleur qui est totalement opérationnel avec de nombreuses méthodes.


- ☒ Vrai
- ☐ Faux

 Effectivement, à partir d'une entité Symfony qui s'appuie sur Doctrine, Symfony est capable de générer un contrôleur totalement fonctionnel avec des formulaires et des pages de templates.

### Question 4

Il est possible d'utiliser la méthode PUT dans un contrôleur.


- ☒ Vrai
- ☐ Faux

 La méthode PUT est la plus utilisée pour la modification, mais notez que **PATCH** existe aussi et permet d'apporter des modifications partielles à une ressource existante.

### Question 5

Je n'ai pas besoin de Doctrine pour créer un CRUD avec MakerBundle.

- ☐ Vrai
- ☒ Faux

 La création d'un crud avec MakerBundle repose sur les entités que gère Doctrine.



**p. 13 Solution n°3**

Symfony est un framework très puissant et qui vous permet d'utiliser un ORM (Object-Relational Mapping) comme Doctrine qui permet de gérer la base de données. À partir de diagrammes de classes, vous allez créer vos entités, car la puissance de Doctrine est d'utiliser pour chaque table de la base de données une classe PHP que l'on nomme Entité. Une fois les entités créées avec leurs propriétés, vous pourrez, grâce à MakerBundle, créer rapidement un CRUD pour chaque entité avec simplement une ligne de commande. Bien qu'il faudra certainement apporter des modifications à vos méthodes de contrôleur en peu de temps, vous aurez la structure de base pour travailler.

**p. 13 Solution n°4**


Le rôle du contrôleur est de coordonner les travaux, pas d'effectuer le travail. Pour respecter la logique métier et pour garder un contrôleur lisible, il est indispensable de créer des services. Pour cela, vous devrez, dans un dossier nommé Services dans « src », créer des fichiers qui porteront le nom « service », par exemple « CommentService.php » ou encore « JWTService.php ». Dans ces fichiers, vous mettrez autant de méthodes que nécessaire et n'oubliez pas de rajouter le namespace que vous pourrez appeler avec `use`, suivi du chemin d'accès dans le contrôleur, ainsi que dans les paramètres des méthodes dont vous aurez besoin en lui attribuant une variable.

**Exercice p. 13 Solution n°5****Question 1**

Le contrôleur s'occupe de tout, il est donc forcément volumineux.

☐ Vrai

☒ Faux


 Le contrôleur est effectivement au cœur de l'action mais pour plus de lisibilité, il doit être le plus concis possible. On peut faire cela en créant des services.

**Question 2**

Le contrôleur est effectivement au cœur de l'action mais pour plus de lisibilité, il doit être le plus concis possible. On peut faire cela en créant des services.

☒ Vrai

☐ Faux

 Grâce au mot-clé `extends`, les contrôleurs peuvent hériter d'autre classe.

**Question 3**

Il est obligatoire d'ajouter `extends AbstractController` au contrôleur.

☐ Vrai


☒ Faux

 `AbstractController` n'est pas obligatoire, mais apporte de nombreux services.

#### Question 4

---


Un objet et une classe sont la même chose.

- ☐ Vrai
- ☒ Faux
-  En POO, les objets sont des instances de classes.

#### Question 5

---

Quel nom est le plus approprié pour un contrôleur ?

- ☐ Controllerproduct
- ☐ controllerproduct
- ☐ ControllerProduct
- ☒ ProductController
-  La convention veut que le nom d'un contrôleur finisse toujours par « *Controller* ».