Symfony - cours

Laurent BOUCHET

Mis à jour le 10/11/2022

(Symfony 5.4)

1 - Introduction

1.1 - Qu’est-ce que Symfony

Symfony est un ensemble de composants PHP, un framework d’application web et une méthodologie de développement.

Il peut être utilisé de manière modulaire en se servant uniquement des composants nécessaires.

Il peut être utilisé sous la forme d’un framework (lui-même basé sur les composants Symfony).

Un framework fournit un cadre de développement permettant de créer des applications structurées, maintenables et évolutives. Il permet aussi de gagner du temps en se servant de librairies PHP déjà existantes, fonctionnelles et éprouvées, afin de se concentrer sur le code métier.

1.2 - Historique

Symfony a été créé par l'agence web française SensioLabs par Fabien Potencier.

Il l'a créé au départ pour les besoins de son agence web et l'a ensuite partagé avec la communauté.

La première version de Symfony voit le jour en 2005.

La version 2 sortie en 2011 casse totalement la compatibilité avec la version précédente. Le framework est en effet entièrement revu.

La version 3 fait son entrée en 2015.

Symfony 4 sort en 2017, la version 5 en 2019 et la version 6 en 2021.

1.3 - Gestion des versions

Symfony suit le semantic versionning :

- une nouvelle version "patch" sort tous les mois (par expl : v2.8.15, v4.1.7), elle ne contient que des bug fixes (des corrections de bogues)
- une nouvelle version mineure sort tous les 6 mois (par expl : v2.8, v3.2, v4.1) en mai et en novembre, elle contient des bug fixes et des nouvelles fonctionnalités, mais ne casse pas la compatibilité
- une nouvelle version majeure sort tous les 2 ans (par expl : v3.0, v4.0), elle peut casser la compatibilité, il faut donc parfois faire quelques changements (des consignes de mises à jour sont mises à disposition)

Le site de symfony propose une roadmap : https://symfony.com/roadmap.

A partir de la version 3 de Symfony, le nombre de versions mineures est limitée à 5 par branche (par expl : v3.0, v3.1, v3.2, v3.3, v3.4).

La dernière version mineure d'une branche est considérée comme une version LTS (Long Term Support).
Symfony assure la correction de bugs pendant 3 ans et la correction des failles de sécurité pendant 4 ans.

Les autres versions sont considérées comme des versions standards.
Symfony assure la correction de bugs pendant 8 mois et la correction des failles de sécurité pendant 14 mois.

1.4 - Quelques chiffres

Plus de 3000 contributeurs.

Une communauté de plus de 600000 développeurs répartis dans plus de 120 pays.

Plus de 48 millions de téléchargements par mois.

Plus de 2,5 milliards de téléchargements au total.

De nombreux projets utilisent Symfony :

- CMS (Drupal, Joomla…)
- eCommerce (Magento, Prestashop…)
- frameworks (Laravel, CakePHP…)
- APIs (Facebook, Google…)
- autres...

2 - Installation

2.1 - Pré-requis

Chaque version de Symfony nécessite une version de PHP minimale pour fonctionner.
Par expl, la version Symfony 4 de Symfony nécessite PHP 7.1 ou une version ultérieure.

Il faut aussi que certaines extensions PHP soient installées et activées (Ctype, iconv, JSON, …).

Certains dossiers du projet doivent aussi être accessibles en écriture.

Il existe un outil permettant de vérifier cela automatiquement.

Voir cette page web pour plus d'informations sur les pré-requis.

2.2 - Installer Composer

Composer est un gestionnaire de dépendances PHP.

Il permet d'installer des librairies PHP pour un projet spécifique ou de manière globale.

Symfony a besoin de Composer. Il faut donc procéder à son installation (voir la procédure sur le site officiel).

Une fois l'installation effectuée, il faut vérifier qu'il est bien installé.

Taper la commande suivante afin de retourner la version de composer :

```
composer --version
```

(si composer n'est pas installé, un message d'erreur sera retourné)

2.3 - Installer un nouveau projet Symfony

Il existe plusieurs façons d'installer Symfony (Symfony dispose aussi d'un "installateur").

Nous utiliserons ici l'installation via Composer.

La commande générique est la suivante :

```
php composer create-project distribution_de_symfony nom_du_projet
```

Dans cette commande :
- "distribution de symfony" correspond à la distribution que l'on souhaite installer (voir plus bas)
- "nom_du_projet" correspond au nom du dossier dans le lequel le projet sera installé

Il existe plusieurs distributions de Symfony :

- symfony/skeleton : correspond à une version minimaliste (recommandé par exemple pour des applications en ligne de commande)
- symfony/website-skeleton : correspond à une version "standard" (recommandé par exemple pour des sites internet)

2.4 - Installer un projet Symfony existant

Récupérer le code du projet via git ou en le téléchargeant.

Ouvrez un terminal, allez dans le répertoire du projet et lancer la commande suivante afin d'installer le code Symfony et celui des dépendances :

```
php composer install
```

Si il y a une base de données, il faudra :

- La configurer via le paramètre "DATABASE_URL" situé dans fichier ".env" (ou ".env.local") (voir Les environnements);
- La créer, créer sa structure et éventuellement charger les fixtures (voir Doctrine) - ou bien - il faudra importer directement un export de la base de données.

3 - Avant de commencer

3.1 - Les répertoires

L’arborescence par défaut est la suivante :

```
├── bin/          (contient tous les exécutables du framework / des dépendances)
├── config/       (contient la configuration du framework, des dépendances et le routing)
├── migrations/   (contient les migrations permettant de mettre la base de données à jour)
├── public/       (contient le "front controller" et les assets, seul dossier accessible de l’extérieur)
├── src/          (contient votre code !)
├── templates/    (contient les templates...)
├── tests/        (contient les tests unitaires / fonctionnels)
├── translations/ (contient les traductions)
├── var/          (contient le cache, les logs et les sessions)
└── vendor/       (contient toutes les dépendances PHP)
```

3.2 - La console

Symfony intègre une console accessible via l'invite de commandes (Windows) / le terminal (Linux / Mac).

Elle facilite la vie du développeur en automatisant certaines tâches comme par exemple :

- afficher la version de Symfony
- lister toutes les commandes disponibles
- générer du code
- vider le cache
- déboguer
- ...

Il est possible de créer de nouvelles commandes qui apparaîtront alors dans la liste des commandes de la console.

Pour afficher la liste de toutes les commandes disponibles, entrer la commande suivante :

```
php bin/console
```

3.3 - Les bundles

Un bundle est une sorte de plugin pour Symfony représentant une dépendance externe.
Il en existe une multitude listés par expl à ces adresses :
The 30 Most Useful Symfony Bundles (and making them even better)
Symfony Bundles Documentation
Knp Bundles

Pour installer un bundle, il faut utiliser la commande suivante : composer require nom_du_bundle

Jusqu'à la version 3.x de Symfony, il était recommandé d'organiser le code de son application en utilisant des bundles. Depuis Symfony 4.0, cela n'est plus recommandé sauf si vous souhaitez créer un code réutilisable afin de le partager.

Symfony est lui-même construit sur des composants qui sont eux-mêmes des bundles.

Symfony 4.0 utilise Flex par défaut. Flex est un plugin pour composer qui permet d'installer le bundle et de le configurer grâce à un système de recettes (recipes).

Avant cette version, il fallait configurer manuellement chaque bundle installé.

3.4 - Les environnements

Une application Symfony dispose par défaut de 3 environnements.

L’environnement "dev" est utilisé lors du développement de l’application (en local). Il propose des outils de débogage et enregistre dans les logs un maximum d’informations.

L’environnement "prod" est utilisé sur le serveur de production (le site en ligne). L’application est alors optimisée en terme de rapidité et n’enregistre que les erreurs dans les logs.

L’environnement "test" est utilisé pour tests.

Il est possible de spécifier l’environnement dans le fichier ".env" (à la racine du projet) en modifiant la ligne :

```
APP_ENV=environnement_souhaite
```

Il est conseillé de créer un fichier ".env.local" (ignoré par git).
Ce fichier écrasera les valeurs renseignées dans le fichier ".env" et sera propre à la machine sur laquelle il a été créé.
Cela permet donc d’avoir une configuration variant en fonction de l’environnement.

3.5 - Le cache

Symfony utilise un cache pour son fonctionnement.

Il est parfois nécessaire de vider le cache de Symfony (en particulier dans un environnement de production).

La commande est la suivante :

```
php bin/console cache:clear
```

Il est possible de vider le cache manuellement (en supprimant le contenu du dossier "cache") mais il est conseillé d'utiliser la commande proposée.

En effet, la commande en plus de vider le cache, fait aussi un "warm-up".
Cela consiste à "préparer" le cache et donc d'optimiser les prochaines requêtes.

4 - Créer une première page

Pour créer une nouvelle page, il faut en général suivre les 3 étapes suivantes.

Par exemple, nous souhaitons créer une page situé à l'URL "/hello-world" affichant le message "Hello World !!!".

4.1- Créer une route

Ouvrir le fichier "config/routes.yaml" et y ajouter le contenu suivant :

```
hello_world:
    path:         /hello-world
    controller: App\Controller\HelloWorldController::index
```

4.2- Créer un contrôleur

Créer le fichier "src/controller/HelloWorldController.php" et y ajouter le contenu suivant :

```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class HelloWorldController extends AbstractController
{
    public function index()
    {
        return $this->render('hello_world/index.html.twig');
    }
}
```

4.3- Créer un template

Créer le fichier "templates/hello_world/index.html.twig" et y ajouter contenu suivant :

```
<h1>Hello World !!!</h1>
```

Afin d’afficher cette première page, dans le navigateur, taper l’URL correspondant à votre projet Symfony suivie de "index.php/hello-world".

Une page affichant un titre "Hello World !!!" devrait apparaître !

Le routing, les contrôleurs et les templates seront détaillés dans les chapitres suivants.

5 - Le routing

5.1 - Introduction

Le routing Symfony permet d'avoir de belles URLs de la forme "/blog/post/13" (à la place d'une URL du style "index.php?post_id=13" par exemple) grâce à l'URL rewriting.

Une route consiste en général à associer une URL à un contrôleur.

Les routes peuvent être configurées via un fichier de configuration (comme vu précédemment).

Elle peuvent aussi être configurées via des annotations (méthode recommandée).

Il est possible d'afficher toutes les routes de l'application grâce à la ligne de commande :

```
php bin/console debug:router
```

5.2 - Routes simples

Reprenons l'exemple précédent et modifions le afin d'utiliser des annotations :

- supprimer la route ajoutée dans le fichier "config/routes.yaml"
- adapter le code dans le contrôleur en ajoutant les nouvelles lignes :

```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class HelloWorldController extends AbstractController
{
    /**
     * @Route("/hello-world", name="hello_world")
     */
    public function index()
    {
        return $this->render('hello_world/index.html.twig');
    }
}
```

5.2 - Routes avec wildcard

5.2.1 - Introduction

Il est possible de créer des routes avec un ou plusieurs wildcards (joker).

Prenons par exemple la route suivante censée afficher une page de contenu dynamique :

```php
namespace App\Controller;

//...

class PageController
{
    /**
     * @Route("/page/{slug}", name="page_show")
     */
    public function show($slug)
    {
        // ...
    }
```

Le wildcard "slug" est - dans le cas présent - le slug de la page en base de données.

Le slug est une sorte d'URL partielle identifiant de manière unique chaque page.

Les URLS suivantes correspondent bien à la route définie :

- page/**debuter-avec-symfony**
- page/**les-meilleurs-bundles**
- page/**le-composant-router**

Nous pouvons constater que la méthode "show($slug)" contient un argument "$slug".

Symfony stocke automatiquement le contenu du wildcard dans la variable correspondante.

Autrement dit, la variable "$slug" contient le slug renseigné dans l'URL.

Si l'URL est "page/debuter-avec-symfony", la variable "$slug" aura pour valeur "debuter-avec-symfony".

5.2.2 - Valeur par défaut d’un wildcard

Prenons un autre exemple : la route "page_list" est censée afficher plusieurs posts d’un blog correspondant à un numéro de page représenté par le wildcard "pageNumber".

```php
namespace App\Controller;

//...

class PageController
{
    /**
     * @Route("/blog/{pageNumber}", name="page_list")
     */
    public function list($pageNumber)
    {
        // ...
    }
```

Nous souhaitons afficher la page 1 dans le cas où la page n’est pas spécifiée dans l’URL (/blog).

Pour que cela soit possible, on peut spécifier une valeur par défaut au wildcard "pageNumber" de la manière suivante :

```php
namespace App\Controller;

//...

class PageController
{
    /**
     * @Route("/blog/{pageNumber}", name="page_list")
     */
    public function list($pageNumber = 1)
    {
        // ...
    }
```

5.2.3 - Requirements

Reprenons l'exemple précédent et imaginons que l'internaute souhaite accéder à l'URL suivante : /blog/nimportequoi

Le numéro de page va récupérer la valeur "nimportequoi" et un problème va se poser car ce n'est pas un numéro de page valide…

Pour pallier à cela, il est possible d'utiliser l'option "requirements" dans la route afin de spécifier le format du wildcard :

```
/**
 * @Route("/blog/{pageNumber}", name="page_list", requirements={"pageNumber"="\d+"})
 */
```

NB : \d+ est une expression régulière signifiant que le wildcard "pageNumber" doit être un nombre.

5.3 - Groupes de routes et préfixes

Il est courant qu'un groupe de routes partage certaines options (par exemple, toutes les routes liées au blog commencent par /blog). C'est pourquoi Symfony inclut une fonctionnalité permettant de partager la configuration des routes.

Il est possible de définir une configuration commune en utilisant des options lors de l'importation des routes au niveau du fichier de configuration "config/routes/annotations.yaml" :

```yaml
back:
    resource: ../../src/Controller/Back/
    type: annotation
    name_prefix: back_
    prefix: /back
    trailing_slash_on_root: false
```

Dans l'exemple ci-dessus, Symfony va importer toutes les routes situées dans les classes contrôleur du dossier "/src/Controller/Back" sous forme d'annotation.
Chacune de ces routes aura son nom préfixé par "back_" et son URL préfixée par "/back".

L'option "trailing_slash_on_root" permet d'éviter que Symfony ajoute automatiquement un slash de fin à certaines URLs (par exemple cela permet d'éviter que l'URL "/back" ne donne l'URL "/back/".

Il est possible de définir une annotation "@Route" directement au niveau de la classe du contrôleur :

```php
namespace App\Controller;
// ...

/**
 * @Route("/blog", name="blog_")
 */
class BlogController extends AbstractController
{
    /**
     * @Route("/voir/{id}", name="show")
     */
    public function show(int $id): Response
    {
        // ...
    }
}
```

Dans cet exemple, la route de l'action show() aura :

- Pour nom final "blog_show";
- Pour URL finale "/blog/voir/15".

6 - Les contrôleurs

6.1 - Introduction

Un contrôleur est une méthode PHP qui a pour but de :

- lire les informations de l'objet Request (objet créé par Symfony correspondant à la requête HTTP)
- créer puis retourner un objet Response (objet correspondant à la réponse HTTP).

La réponse retournée par le contrôleur peut être :

- une page HTML
- du JSON
- du XML
- un téléchargement de fichier
- une redirection
- ...

Le contrôleur peut effectuer toutes sortes d’actions comme par exemple :

- récupérer des enregistrements en base de données
- ajouter / modifier / supprimer un enregistrement en base de données
- envoyer des mails
- retourner une erreur 404...
- ...

6.2 - Créer un contrôleur

Il est possible d'utiliser la console afin de générer une classe contrôleur.
Pour cela, taper la commande suivante :

```
php bin/console make:controller
```

Le nom d'une classe contrôleur doit toujours être suffixé par "Controller".

Si vous indiquez que votre contrôleur s'appelle TestController, cela va créer :

- Un contrôleur "src/Controller/TestController.php";
- Un template "template/test/index.html.twig" (voir Twig et le templating).

Vous pouvez utiliser la commande suivante afin de générer un contrôleur "CRUD" :

```
php bin/console make:crud
```

L'action sera la même que la précédente mis à part que le contrôleur créé contiendra toutes les actions possibles : créer, lire, mettre à jour et supprimer une entité (voir Doctrine).

6.3 - La classe AbstractController

6.3.1 - Étendre la classe

Symfony propose un contrôleur de base appelé AbstractController.

Si vous étendez cette classe, vous aurez accès à certaines méthodes dites "helpers" bien pratiques.

```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class MonController extends AbstractController
{
    // ...
}
```

6.3.2 - Créer des URLs

La méthode generateUrl() permet de générer des URLs à partir d'une route.

Voici sa syntaxe :

```
$this->generateUrl('nom_de_la_route');
```

Si la route prend des paramètres, il faut alors spécifier un 2ème argument sous forme de tableau.
Voici la syntaxe :

```
$this->generateUrl('nom_de_la_route', [
    'param1' => $param1,
    'param2' => $param2,
]);
```

Par expl, pour la route "blog_post" ("/blog/15" pour afficher l'article d'un blog ayant l'id 15) :

```
$this->generateUrl('blog_post', ['id' => 15]);
```

6.3.3 - Effectuer une redirection

Il est fréquent d'effectuer des redirections, par expl :

- après soumission d'un formulaire
- après s'être connecté via un formulaire de login
- après avoir supprimé un élément dans une liste

Il est possible d'effectuer une redirection à partir d'une route via le helper "redirectToRoute()".

Par exemple :

```
$this->redirectToRoute('blog_post', ['id' => 15]);
```

Il est aussi possible d'effectuer une redirection à partir d'une URL via le helper "redirect()" (utilisé bien souvent pour les URLs externes).

Par exemple :

```
$this->redirect('https://symfony.com');
```

6.3.4 - Rendre un template

Pour afficher du HTML, on "rend" en général un template.

Cela consiste à retourner une réponse (un objet Response) contenant le rendu du template passé en argument.

On utilise pour cela le helper "render()" en spécifiant le chemin vers le template.

Voici sa syntaxe :

```
$this->render('chemin/vers/le/template.html.twig');
```

Si on souhaite transmettre des données au template, on peut passer un 2ème argument à cette méthode correspondant à un tableau contenant tous les paramètres.

```
$this->render('chemin/vers/le/template.html.twig', [
    'param1' => $param1,
    'param2' => $param2,
]);
```

6.3.5 - Gérer les pages 404 et les erreurs

Si une ressource n’est pas trouvée (en base de données par expl), il faut retourner une réponse 404.

On peut utiliser pour cela le helper "createNotFoundException($msg)" :

```
$post = ...;       // récupération d'un post en base de données (explications dans le chapitre Doctrine)

if (!$post) {      // si le post n'est pas trouvé
    // on retourne une erreur 404
    throw $this->createNotFoundException("Le post n'existe pas !");
}
```

Cette méthode est un raccourci permettant de créer une exception de type "NotFoundHttpException" (correspondant à une erreur 404) et de la retourner en tant que réponse.

Lorsqu’une erreur est rencontrée, on peut retourner une erreur de type 500.

Pour cela, il faut lancer une exception en utilisant la classe Exception ou une classe héritant de la classe Exception.

Par expl :

```
throw new \Exception('Une erreur s'est produite !');
```

Le fait de lancer une exception aura pour effet de provoquer une 500.

6.4 - Le paramconverter

Le paramconverter est une annotation optionnelle permettant de convertir les paramètres de la requête en objets qui seront injectés dans les arguments du contrôleur.

Par expl :

```
/**
 * @Route("/blog/{id}")
 * @ParamConverter("post", class="App:Post")       // Cette ligne est optionnelle
 */
public function show(Post $post)
{
}
```

- le paramconverter récupère automatiquement l'objet en base de données à partir du paramètre de la requête défini par le "wildcard" dans la route, dans le cas présent, l'id
- si l'objet n'est pas trouvé, une réponse de type 404 est automatiquement retournée
- sinon, l'objet ("$post" dans notre exemple) est accessible dans la signature du contrôleur grâce au type hinting

6.5 - L’objet Request

Pour utiliser l’objet request (qui est un service), il suffit d’ajouter un argument "$request" au contrôleur et de le "type-hinter" avec la classe "Request".

```php
<?php
use Symfony\Component\HttpFoundation\Request;

public function index(Request $request)
{
    // accès à l'objet $request ...
}
```

L’objet Request permet d’effectuer différentes actions.

Vérifier que la requête est une requête AJAX :

```
$request->isXmlHttpRequest();
```

Récupérer un paramètre passé dans l’URL ($_GET) :

```
$request->query->get('page');
```

Récupérer la valeur d’un champ de formulaire venant d’être soumis ($_POST) :

```
$request->request->get('page');
```

Récupérer une instance d’un fichier uploadé ($_FILES) :

```
$request->files->get('photo');
```

Récupérer le nom de la route courante et ses paramètres :

```
$routeName = $request->attributes->get('_route');
$routeParameters = $request->attributes->get('_route_params');
```

La liste n’est pas exhaustive, voir la documentation officielle pour plus d’informations.

7 - Les templates et Twig

7.1 - Introduction

Un template est la meilleure façon d'organiser et de rendre du HTML à l'intérieur de votre application, que vous ayez besoin de rendre du HTML à partir d'un contrôleur ou de générer le contenu d'un e-mail. Les templates dans Symfony sont créés avec Twig : un moteur de template flexible, rapide et sécurisé (protection par défaut des failles XSS).

Le langage de modélisation Twig vous permet d'écrire des templates concis et lisibles, plus conviviaux pour les concepteurs Web et, à plusieurs égards, plus puissants que les templates PHP.

Vous ne pouvez pas exécuter de code PHP dans les templates Twig !

Twig est rapide dans l'environnement prod (car les templates sont compilés en PHP et mis en cache automatiquement), mais pratique à utiliser dans l'environnement dev (car les templates sont recompilés automatiquement lorsque vous les modifiez).

7.2 - Configuration

Twig dispose de plusieurs options de configuration pour définir des éléments tels que le format utilisé pour afficher les nombres et les dates, la mise en cache des templates, etc.

Lisez la référence sur la configuration de Twig pour en savoir plus sur ces options.

7.3 - La syntaxe

7.3.1 - La syntaxe de base

Voici un exemple de template Twig :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        {% if user.isLoggedIn %}
            Hello {{ user.name }}!
        {% endif %}

        {# ... #}
    </body>
</html>
```

Twig définit 3 éléments syntaxiques de base :

```
{{ ... }}
```

"Afficher quelque chose": le contenu d'une variable ou le résultat de l'évaluation d'une expression.

```
{% ... %}
```

"Faire quelque chose": utilisé pour exécuter une certaine logique, telle qu'une condition ou une boucle.

```
{# ... #}
```

"Commenter quelque chose": permet de commenter une partie de code sur une ou plusieurs lignes (la syntaxe est la même). Ces commentaires n'apparaissent pas dans le code source (contrairement aux commentaires HTML).

7.3.2 - Les autres éléments syntaxiques

7.3.2.1 - Introduction

Twig fournit par défaut une longue liste de tags, filtres, fonctions, tests et opérateurs.

Symfony propose aussi ces éléments supplémentaires.

Il existe aussi les extensions officielles de Twig liées aux chaînes de caractères, au HTML, au Markdown, à l'internationalisation, etc.

7.3.2.2 - Les tags

Ce sont des éléments de langage spécifiques à Twig. Les tags les plus utilisés sont listés ci-dessous.

if / elseif / else : structures conditionnelles traditionnelles.

Exemple avec if / else :

```
{% if value == 1 %}
    ...
{% else %}
    ...
{% endif %}
```

Autre exemple avec if / elseif / else :

```
{% if value == 1 %}
    ...
{% elseif value == 2 %}
    ...
{% else %}
    ...
{% endif %}
```

for : boucle (style foreach en PHP) permettant de boucler sur un tableau.

Exemple avec une boucle ne récupérant que la valeur de chaque ligne du tableau :

```
{% for valeur in monTableau %}
{% endfor %}
```

Autre exemple avec une boucle récupérant la clé et la valeur de chaque ligne du tableau :

```
{% for cle, valeur in monTableau %}
{% endfor %}
```

Autre exemple avec une condition "else" dans la boucle for :

```
{% for valeur in monTableau %}
{% else %}
{% endfor %}
```

set : permet de définir une (ou plusieurs) variable(s).

```
{% set maVariable = 'hello' %}
```

block : permet de définir des emplacements qui pourront être de nouveau définis (ou non) dans les templates enfants (voir le chapitre sur l'héritage).

7.3.2.3 - Les fonctions

constant() : appelle une constante PHP.

```
{{ constant('Namespace\\Classname::CONSTANT_NAME') }}
```

include() : retourne le rendu d'un template.

```
{{ include('template.html') }}
```

dump() : affiche le contenu d'une ou plusieurs variables.

```
{{ dump(maVariable) }}
{{ dump(maVariable1, maVariable2) }}
```

`{{ dump() }}` => affiche le contenu de toutes les variables du contexte

parent() : affiche le contenu du bloc parent (voir plus bas concernant l'héritage)

```
{% block monBloc %}
    {{ parent() }}
{% endblock %}
```

7.3.2.4 - Les filtres

Un filtre sert à modifier un élément. Pour appliquer un filtre sur un élément, il faut le faire suivre du caractère "|" puis du nom du filtre.

Il en existe énormément, ils sont très pratiques, en voici quelques uns :

`{{ maDate|date('d-m-Y') }}` formate une date

`{{ monTableau|first }}` affiche le premier élément d'un tableau ou d'une chaîne de caractères

`{{ monTableau|last }}` affiche le dernier élément d'un tableau ou d'une chaîne de caractères

`{{ monTableau|length }}` affiche la longueur d'un tableau ou d'une chaîne de caractères

`{{ maVariable|raw }}` : permet de ne pas protéger une valeur (elles protégées par défaut avec Twig)

7.3.2.5 - Les tests

defined : vérifie si une variable est définie dans le contexte actuel.

```
{% if foo is defined and foo.bar is defined %}
    ...
{% endif %}
```

empty : vérifie si une variable est une chaîne vide, un tableau vide, un hachage vide, exactement faux, ou exactement nul.

```
{% if foo is empty %}
    ...
{% endif %}
```

null : renvoie *true* si la variable est nulle.

```
{% if foo is null %}
    ...
{% endif %}
```

7.3.2.6 - Les opérateurs

Twig propose de nombreux opérateurs, dont voici une liste non exhaustive.

Les opérateurs mathématiques

+ : addition => `{{ 1 + 1 }}`
- : soustraction => `{{ 3 - 2 }}`
/ : division => `{{ 1 / 2 }}`
% : modulo => `{{ 11 % 7 }}`
***** : multiplication => `{{ 2 * 2 }}`
...

Les opérateurs logiques

"**and**" correspond à "et" ("&&" en PHP)
"**or**" correspond à "ou" ("||" en PHP)
"**not**" correspond à la négation ("!" en PHP)
...

Les opérateurs de comparaison

égal à : **==**
différent de : **!=**
inférieur à : **<**
inférieur ou égal à : **<=**
supérieur à : **>**
supérieur ou égal à : **>=**

L'opérateur "containment"

Cet opérateur représenté par "**in**" fonctionne avec les tableaux et les chaînes de caractères.

Par exemple :

`{{ 1 in [1, 2, 3] }}` permet de vérifier si la valeur 1 existe dans le tableau [1, 2, 3].

`{{ 'cd' in 'abcde' }}` permet de vérifier si la valeur "cd" existe dans la chaîne "abcde".

7.3.3 - Ajouter ses propres éléments syntaxiques

Les extensions Twig permettent de créer des fonctions, des filtres et d'autres éléments personnalisés à utiliser dans vos templates Twig.

Vous pouvez créer une extension Twig grâce à la commande suivante :

```
php bin/console make:twig-extension
```

Cela va générer une classe similaire à celle-ci.

```
namespace App\Twig;

use Twig\Extension\AbstractExtension;

class AppExtension extends AbstractExtension
{
    public function getFilters()
    {
        return [];
    }
    public function getFunctions()
    {
        return [];
    }
}
```

Afin de créer un nouveau filtre "intials", il faut :

- Ajouter la ligne suivante au tableau retourné par la méthode "getFilters()";
- Implémenter la nouvelle méthode "getInitials()".

```php
use Twig\TwigFilter;

class AppExtension extends AbstractExtension
{
    public function getFilters()
    {
        return [
            new TwigFilter('intials', [$this, 'getInitials']),
        ];
    }

    public function getInitials(string $lastname, string $firstname): string
    {
        return strtoupper($lastname[0] . $firstname[0]);
    }
}
```

Afin de créer une nouvelle fonction "area", il faut :

- Ajouter la ligne suivante au tableau retourné par la méthode "getFunctions()";
- Implémenter la nouvelle méthode "calculateArea()".

```php
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFunction;

class AppExtension extends AbstractExtension
{
    public function getFunctions()
    {
        return [
            new TwigFunction('area', [$this, 'calculateArea']),
        ];
    }

    public function calculateArea(int $width, int $length): int
    {
        return $width * $length;
    }
}
```

7.4 - Créer des templates

7.4.1 - Emplacement et nommage

Les templates se situent par défaut dans le répertoire "templates/".

Utilisez la casse "snake_case" pour les noms de fichiers et les répertoires.
Par exemple : blog_posts.html.twig, admin/default_theme/blog/index.html.twig, etc.

Définissez deux extensions pour les noms de fichiers.
Par exemple : index.html.twig ou blog_posts.xml.twig
La première extension (html, xml, etc.) étant le format final que le template va générer.

Bien que les templates génèrent généralement des contenus HTML, ils peuvent générer n'importe quel format textuel (par exemple du CSV, du XML, du TXT, etc.).

7.4.2 - Les variables

Un besoin courant des templates est d'afficher des objets et des tableaux plutôt que des chaînes de caractères, des nombres et des valeurs booléennes. C'est pourquoi Twig offre un accès rapide aux variables PHP complexes.

```
<p>{{ user.name }} added this comment on {{ comment.publishedAt|date }}</p>
```

Dans l'exemple ci-dessus, la notation "user.name" signifie que vous voulez afficher une information "le nom" stockée dans la variable "user".
User est-il un tableau ou un objet ? Est-ce que name est une propriété ou une méthode ?
Dans Twig, cela n'a pas d'importance !

En utilisant la notation foo.bar, Twig essaie de récupérer la valeur de la variable dans l'ordre suivant :
$foo['bar'] (tableau et élément) ;
$foo->bar (objet et propriété publique) ;
$foo->bar() (objet et méthode publique) ;
$foo->getBar() (objet et méthode getter) ;
$foo->isBar() (objet et méthode isser) ;
$foo->hasBar() (objet et méthode hasser) ;
Si aucun des éléments ci-dessus n'existe, null est utilisé (ou une exception Twig\Error\RuntimeError est lancée si l'option "strict_variables" est activée).

7.4.3 - Liens vers des assets

Les assets représentent les CSS, JavaScript et images.

La fonction "asset()" de Twig permet créer un lien vers un asset en générant l'URL.
Cela est possible grâce au bundle Asset (intégré par défaut dans le framework web Symfony).

Voici quelques exemples :

```
{# the image lives at "public/images/logo.png" #}
<img src="{{ asset('images/logo.png') }}" alt="Symfony!"/>

{# the CSS file lives at "public/css/blog.css" #}
<link href="{{ asset('css/blog.css') }}" rel="stylesheet"/>

{# the JS file lives at "public/bundles/acme/js/loader.js" #}
<script src="{{ asset('bundles/acme/js/loader.js') }}"></script>
```

L'objectif principal de la fonction asset() est de rendre votre application plus portable. Elle se charge de générer des chemins corrects (en prenant en compte les éventuels sous-répertoires).

Pour vous aider à construire, versionner et réduire vos ressources JavaScript et CSS d'une manière moderne, il est possible d'utiliser Webpack Encore de Symfony.

7.4.5 - La variable globale "app"

Une variable nommée "app" est disponible dans tous les templates Twig.

Elle donne accès à certaines informations sur l'application :
app.user : retourne l'objet correspondant à l'utilisateur actuellement connecté (ou null si ce dernier n'est pas authentifié).

app.request : retourne l'objet "Request" contenant toutes les données de la requête.

app.session : retourne "Session" contenant toutes les données de la session actuelle de l'utilisateur (ou null si il n'y en a pas)

app.flashes : retourne un tableau de toutes les messages flash (voir la documentation pour plus d'informations).

app.environment : retourne le nom de l'environnement de configuration actuel (dev, prod, etc).

app.debug : retourne *true* si le mode "debug" est activé, sinon retourne *false*.

7.5 - Réutilisation du contenu des templates

7.5.1 - Inclure un template

Si un certain code Twig est répété dans plusieurs templates, vous pouvez l'extraire dans un seul "fragment de template" et l'inclure dans d'autres templates.

Un fragment est un template Twig dont le nom sera préfixé par un "_" (le préfixe _ est facultatif, mais il s'agit d'une convention utilisée pour mieux différencier les templates complets des fragments de templates).

Le fragment peut alors être inclus de la manière suivante :

```
{{ include('blog/_user_profile.html.twig') }}
```

Le template inclus a accès à toutes les variables du template qui l'inclut.

La fonction include() Twig prend comme argument le chemin du template à inclure.

Vous pouvez également passer des variables au template inclus. Ceci est utile par exemple pour renommer des variables :

```
{{ include('blog/_user_profile.html.twig', {user: blog_post.author}) }}
```

7.5.2 - Intégrer un contrôleur

L'inclusion de fragments de template est utile pour réutiliser le même contenu sur plusieurs pages.

Mais lorsque le fragment inclus doit afficher les résultats d'une requête dans une BDD (dans ce cas, la même requête de BDD doit être effectuée dans chaque page qui inclut le fragment), il est possible d'intégrer le résultat de l'exécution d'un contrôleur avec les fonctions Twig render() et controller() :

```
{# Si le contrôleur est associé à une route, utilisez les fonctions path() ou url() #}
{{ render(path('latest_articles', {max: 3})) }}
{{ render(url('latest_articles', {max: 3})) }}

{# Si vous souhaitez utiliser une URL non publique, utilisez la fonction controller() #}
{{ render(controller(
    'App\\Controller\\BlogController::recentArticles', {max: 3}
)) }}
```

L'intégration de contrôleurs nécessite l'envoi de requêtes à ces contrôleurs et le rendu de certains templates en conséquence. Cela peut avoir un impact significatif sur les performances de l'application si vous intégrez beaucoup de contrôleurs. Si possible, mettez en cache le fragment de template.

7.5.3 - Héritage des templates et layouts

7.5.3.1 - Introduction

Au fur et à mesure que votre application se développe, vous trouverez de plus en plus d'éléments répétés entre les pages, tels que des en-têtes, des pieds de page, des barres latérales, etc. L'inclusion de templates et l'intégration de contrôleurs peuvent vous aider, mais lorsque les pages partagent une structure commune, il est préférable d'utiliser l'héritage.

Le concept d'héritage des templates Twig est similaire à l'héritage des classes PHP. Vous définissez un template parent à partir duquel les autres templates peuvent s'étendre et les templates enfants peuvent remplacer certaines parties du template parent.

Nous allons voir ici l'utilisation de l'héritage avec le mot-clé "extends".
Twig propose d'autres tags tel que "use" (héritage horizontal) et "embed".
Cette présentation permet d'en apprendre plus et de savoir quels tags utiliser en fonction du contexte.

7.5.3.2 - Organisation des templates

Symfony recommande l'héritage de templates à trois niveaux suivant pour les applications moyennes et complexes.

Niveau 1 : templates/base.html.twig, définit les éléments communs à tous les templates d'application, tels que <head>, <header>, <footer>, etc ;

Niveau 2 : templates/layout.html.twig, s'étend à partir de "base.html.twig" et définit la structure de contenu utilisée dans toutes ou la plupart des pages, comme une disposition de contenu en deux colonnes + barre latérale.
Certaines sections de l'application peuvent définir leurs propres mises en page (par exemple, **templates/blog/layout.html.twig**) ;

Niveau 3 : templates/*.html.twig, les pages de l'application qui s'étendent à partir du template principal layout.html.twig ou de tout autre template de section (**templates/blog/*.html.twig**).

7.5.3.3 - Le template de niveau 1

Voici un exemple de template de base (niveau 1) :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}My Application{% endblock %}</title>
        {% block stylesheets %}
            <link rel="stylesheet" type="text/css" href="/css/base.css"/>
        {% endblock %}
    </head>
    <body>
        {% block body %}
            <div id="sidebar">
                {% block sidebar %}
                    <ul>
                        <li><a href="{{ path('homepage') }}">Home</a></li>
                        <li><a href="{{ path('blog_index') }}">Blog</a></li>
                    </ul>
                {% endblock %}
            </div>
```

```
            <div id="content">
                {% block content %}{% endblock %}
            </div>
        {% endblock %}
    </body>
</html>
```

La balise Twig "block" définit les sections de la page qui peuvent être remplacées dans les templates enfants.

Elles peuvent être vides, comme le bloc "content", ou définir un contenu par défaut, comme le bloc "title", qui s'affiche lorsque les templates enfants ne les remplacent pas.

7.5.3.4 - Les templates de niveau 2

Voici un exemple de template "layout" (niveau 2) :

```
{% extends 'base.html.twig' %}

{% block content %}
    <h1>Blog</h1>

    {% block page_contents %}{% endblock %}
{% endblock %}
```

Le template s'étend à partir du template "base.html.twig" et définit uniquement le contenu du bloc "content". Les autres blocs du template parent afficheront leur contenu par défaut.

Remarque importante : lors de l'utilisation de "extends", il est interdit à un template enfant de définir des parties de template en dehors d'un bloc. Le code suivant génère une erreur de syntaxe :

```
{% extends 'base.html.twig' %}

<h1>Blog</h1>}
```

7.5.3.5 - Les templates de niveau 3

Voici un exemple d'un template de page (niveau 3) :

```
{% extends 'blog/layout.html.twig' %}

{% block title %}Blog Index{% endblock %}

{% block page_contents %}
    {% for article in articles %}
        <h2>{{ article.title }}</h2>
        <p>{{ article.body }}</p>
    {% endfor %}
{% endblock %}
```

Ce template s'étend à partir du template "layout" de deuxième niveau mais remplace les blocs de différents templates parents : "page_contents" du template de "layout" et "title" du template de "base".

Lorsque vous effectuez le rendu du template de niveau 3, Symfony utilise trois templates différents pour créer le contenu final. Ce mécanisme d'héritage augmente votre productivité car chaque template n'inclut que son contenu unique et laisse le contenu répété et la structure HTML à certains templates parents.

7.5.3.6 - Exemples

Soit un template "parent" nommé "base.html.twig" ayant pour contenu un bloc "titre".

```
{% block titre %}Mon site{% enblock %}
```

Soit un template "enfant" héritant simplement du template "parent".
=> Le template enfant hérite du bloc "titre" du template parent, la page aura pour titre "Mon site".

Soit un autre template "enfant" héritant du template "parent" et re-définissant le bloc "titre" parent :

```
{% block titre %}page de contenu{% endblock %}
```

=> Le bloc "titre" du template enfant remplace le bloc titre du template parent, la page aura pour titre "page de contenu".

Soit un autre template "enfant" héritant du template "parent", re-définissant le bloc "titre" parent tout en appelant le contenu du bloc "titre" parent :

```
{% block titre %}
    {{ parent() }} - page de contenu
{% endblock %}
```

=> Le bloc "titre" du template enfant remplace le bloc titre du template parent, mais appelle explicitement le bloc "titre" original du template parent, la page aura pour titre "Mon site - page de contenu".

8 - La BDD et l’ORM Doctrine

8.1 - Introduction

Doctrine est un ORM (Object Relationnal Mapping). Doctrine est intégré par défaut avec Symfony,

Un ORM permet d’établir une correspondance entre des objets et des tables d’une base de données (BDD ou DB pour DataBase en anglais) relationnelle et de communiquer avec la base de données directement avec des objets.

Doctrine s’occupe en effet d’hydrater les objets avec les données en provenance de la base (hydrater un objet signifie le "remplir" avec les données).

Doctrine propose aussi un DQL (Doctrine Query Language) qui est langage orienté objet permettant d’effectuer des requêtes SQL

Doctrine permet aussi d’effectuer des requêtes SQL traditionnelles sans passer par le DQL.

8.2 - Configuration et création de la BDD

Il faut avant tout configurer la BDD via la clé "DATABASE_URL" du fichier ".env.local" :

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
```

Où :

- "mysql" est le Système de Gestion de Base de Données (SGBD)
- "db_user" est le login
- "db_password" est le mot de passe
- "127.0.0.1" est le serveur hôte
- "3306" est le port utilisé
- "db_name" est le nom de la base de données

Par exemple, si il n'y pas de mot de passe (dev local) et si la base de données s'appelle "voiture" :

```
DATABASE_URL=mysql://root@127.0.0.1:3306/voiture
```

Une fois la base de données configurée, il faut la créer via la commande suivante :

```
bin/console doctrine:database:create
```

La commande suivante permet de supprimer la base de données (nécessaire si on souhaite la recréer) :

```
bin/console doctrine:database:drop
```

8.3 - Les entités et la structure de la BDD

8.3.1 - Créer une entité simple

Comme précisé plus haut, pour communiquer avec la base de données, on se sert d'objets. Ces objets sont appelés des entités.

Pour relier ces objets à la base de données, il y a plusieurs moyens (comme pour le routing) :

- les annotations dans les entités
- un fichier de configuration en YAML
- un fichier de configuration en XML
- un fichier de configuration en PHP

Nous utiliserons ici les annotations qui est la méthode recommandée.

Prenons l'exemple suivant :
Nous souhaitons créer une table "Article" en base de données avec les champs suivants :

- un id (id)
- un titre (title)
- ...

Il faut créer la classe Entité correspondante.

Pour pouvoir se servir des annotations dans les entités, il faut ajouter en haut du fichier la ligne suivante :

```
use Doctrine\ORM\Mapping as ORM;
```

Au dessus du nom de classe, il faut utiliser l'annotation suivante :

```
/**
 * @ORM\Entity()
 */
class Article
{
...
```

Cela permet de spécifier à Doctrine que cette classe est une entité.

La classe devra contenir autant de propriétés privées qu'il y a de champ dans la table en base de données (les clés étrangères ne sont pas concernées ici, voir les chapitres sur les relations).

La classe contiendra des getters et setters pour chaque propriété (sauf pour le champ id qui n'est pas censé être modifié, puisque c'est la base de données qui s'en charge).

Par exemple pour le champ "id" :

```php
/**
 * @ORM\Id
 * @ORM\GeneratedValue
 * @ORM\Column(type="integer")
 */
private $id;

public function getId(): int
{
    return $this->id;
}
```

L'identifiant unique "id" étant auto-généré par le système de gestion de base de données :

- il faut spécifier les deux annotations supplémentaires suivantes :

@ORM\Id => indique qu'il s'agit d'un id et donc d'une clé primaire
@ORM\GeneratedValue => indique que la valeur est auto-générée (par la base de données)

- pas besoin de créer de setter, l'id n'est pas censé être modifié

Par exemple pour le champ "title" :

```php
/**
 * @ORM\Column(type="string")
 */
private $title;

public function getTitle(): string
{
    return $this->title;
}

public function setTitle(string $title): self
{
    $this->title = $title;

    return $this;
}
```

Pour information, il est possible de la générer une entité via la commande interactive de la console :

```
php bin/console make:entity
```

Un assistant vous posera plusieurs questions et va créer la classe entité correspondante (ainsi que la classe repository associée).

Si vous spécifiez le nom d'une entité existante, vous pouvez lui ajouter des champs supplémentaires.

Si vous ajoutez l'argument "--regenerate" :

```
php bin/console make:entity --regenerate
```

Doctrine va créer les méthodes (getters/setters) manquants et créer le repository s'il est indiqué dans l'entité et s'il n'existe pas déjà.

Si vous ajoutez en plus l'argument " --overwrite"

```
php bin/console make:entity --regenerate --overwrite
```

Doctrine va recréer toutes les méthodes (getters/setters) et écraser les anciennes et créer le repository s'il est indiqué dans l'entité et s'il n'existe pas déjà.

8.3.2 - Créer la structure de la BDD

Une fois toutes les entités créées, il faut créer le schéma dans la base de données.

La console symfony propose la commande suivante :

```
php bin/console doctrine:schema:create
```

Si le schéma a déjà été créé et si on souhaite le mettre à jour, il faut se servir de la commande suivante :

```
php bin/console doctrine:schema:update
```

Cette commande seule ne fonctionne pas, elle affiche des informations vous précisant qu'il faut ajouter un paramètre à la fin de commande :

- le paramètre "--dump-sql" : permet de simuler les mises à jour et d'afficher les requêtes SQL nécessaires
- le paramètre "--force" : permet de forcer la mise à jour du schéma, c'est un sécurité, afin d'éviter les erreurs et la perte de données

Lorsque le site est déjà en ligne, il est fortement recommandé d'utiliser des migrations.

8.3.3 - Correspondance entre les types Doctrine/PHP/MySQL

Voici une correspondance Doctrine/PHP/MySQL pour les types les plus courants.

Type Doctrine	Type PHP	Type en base de données (MySQL)
string	string	VARCHAR
integer	integer	INT
boolean	boolean	BOOLEAN
date	\DateTime	DATETIME
datetime	\DateTime	TIMESTAMP
blob	stream resource	BLOB

Pour avoir la liste exhaustive de tous les types Doctrine, se référer à la documentation.

8.3.4 - Créer des relations

8.3.4.1 - La relation One-To-One

La relation doctrine One-To-One correspondant à une d'une relation 1-1.
C'est à dire qu'un enregistrement dans une table A peut être lié à un seul enregistrement dans une table B.

Soit l'exemple suivant :
Une voiture a un seul volant.
Un volant appartient à une seule voiture.

Dans la classe "Voiture", il faut ajouter le code suivant :

```
/**
 * @ORM\OneToOne(targetEntity="Volant", mappedBy="voiture") // Entité correspondant au volant
 */
private $volant;      // la voiture possède un volant
```

Cela permettra de récupérer le volant à partir de la voiture :

```
$voiture->getVolant()
```

Dans la classe "Volant", il faut ajouter le code suivant :

```
/**
 * @ORM\OneToOne(targetEntity="Voiture", inversedBy="volant")    // Entité correspondant à la voiture
 */
private $voiture;          // le volant appartient à une voiture
```

Cela permettra de récupérer la voiture à partir du volant :

```
$volant->getVoiture()
```

La classe possédant "inversedBy" dans son annotation générera une clé étrangère dans la table correspondante.

Il faudra aussi ajouter les getters / setters correspondant dans chaque classe.

Il est possible de générer automatiquement ces méthodes grâce à la console.

La commande est la suivante :

```
php bin/console make:entity --regenerate
```

8.3.4.2 - La relation One-To-Many / Many-To-One

La relation doctrine One-To-Many / Many-To-One correspondant à une relation 1-n.
C'est à dire qu'un enregistrement dans une table A peut être lié à plusieurs enregistrements dans une table B. A l'inverse un enregistrement de la table B peut être lié qu'à un seul enregistrement dans la table A.

Soit l'exemple suivant :
Une voiture a plusieurs portes.
Une porte appartient à une seule voiture.

Dans la classe "Voiture", il faut ajouter le code suivant :

```
/**
 * @ORM\OneToMany(targetEntity="Porte", mappedBy="voiture") // Entité correspondant à une porte
 */
private $portes; // la voiture possède des portes
```

Cela permettra de récupérer les portes à partir de la voiture :

```
$voiture->getPortes()
```

Dans la classe "Porte", il faut ajouter le code suivant :

```
/**
 * @ORM\ManyToOne(targetEntity="Voiture", inversedBy="portes")
 */
private $voiture;          // le volant appartient à une voiture
```

Cela permettra de récupérer la voiture à partir de la porte :

```
$porte->getVoiture()
```

La classe contenant l'annotation "ManyToOne" et la mention "InversedBy" générera une clé étrangère dans la table correspondante.

Il faudra aussi mettre à jour le constructeur de la classe contenant la relation "OneToMany" afin d'initialiser un tableau (destiné à contenir les portes dans notre exemple), et ajouter les getters / setters correspondant dans chaque classe.

Dans la classe "Voiture", il y aura donc dans le constructeur la ligne suivante :

```
$this->portes = new ArrayCollection();
```

La commande cité précédemment s'occupera de tout cela :

```
php bin/console make:entity --regenerate
```

8.3.4.3 - La relation Many-To-Many

La relation Many-To-Many correspond à une relation n-m, autrement dit plusieurs entités liées à plusieurs entités. C'est à dire qu'un enregistrement dans une table A peut être lié à plusieurs enregistrements dans une table B. A l'inverse un enregistrement de la table B peut aussi être à plusieurs enregistrements dans la table A.

Soit l'exemple suivant :
Une voiture peut être conduite par plusieurs personnes.
Une personne peut conduire plusieurs voitures.

Dans la classe "Voiture", il faut ajouter le code suivant :

```
/**
  * @ORM\ManyToMany(targetEntity="Personne", mappedBy="voitures") // Entité correspondant à une personne
  */
private $personnes;   // la voiture peut être conduite par plusieurs personnes
```

Cela permettra de récupérer les personnes à partir de la voiture :

```
$voiture->getPersonnes()
```

Dans la classe "Personne", il faut ajouter le code suivant :

```
/**
  * @ORM\ManyToMany(targetEntity="Voiture", inversedBy="personnes")
  */
private $voitures;          // la personne peut conduire plusieurs voitures
```

Cela permettra de récupérer les voitures à partir de la personne :

```
$personne->getVoitures()
```

Il faudra aussi mettre à jour le constructeur de chaque classe afin d'initialiser un tableau, et ajouter les getters / setters correspondant dans chaque classe.

Dans la classe "Voiture", il y aura donc dans le constructeur la ligne suivante :

```
$this->personnes = new ArrayCollection();
```

Idem dans la classe "Personne" :

```
$this->voitures = new ArrayCollection();
```

Utilisez la console pour mettre tout cela à jour !

8.4 - Manipuler les données de la BDD via les entités

8.4.1 - L’entity manager

Afin de manipuler les entités, on se sert de l’entity manager.

Il s’agit d’un objet permettant - entre autres - de créer, modifier et supprimer les entités (et par conséquent des enregistrements en base de données).

Dans un contrôleur, il est possible de le récupérer grâce au helper suivant :

```
$em = $this->getDoctrine()->getManager();
```

Il propose entre autres les fonctions suivantes :

- la fonction "persist" qui permet de "marquer" une entité afin qu’elle soit enregistrée
- la fonction "remove" pour "marquer" une entité afin qu’elle soit supprimée
- la fonction "flush", pour ajouter / mettre à jour / supprimer tous les objets "marqués"

8.4.2 - Créer un enregistrement

Afin d'enregistrer une nouvelle entité en base de données, il faut procéder aux étapes suivantes :

Créer un objet entité :

```
$voiture = new Voiture();
```

Remplir l'entité :

```
$voiture->setVolant($volant);
$voiture->setCouleur('bleu');
```

…

Persister l'entité :

```
$em = $this->getDoctrine()->getManager();
$em->persist($voiture);
$em->flush();
```

8.4.3 - Mettre un enregistrement

Afin d'enregistrer en base de données une entité récupéré via doctrine et ensuite mise à jour, il faut procéder aux étapes suivantes :

Récupération de l'entité : voir plus loin comment récupérer des entités à partir de la base de données

```
$voiture = ...
```

Mettre à jour l'entité :

```
$voiture->setVolant($nouveauVolant);
$voiture->setCouleur('vert');
```

…

Enregistrer l'entité :

```
$em = $this->getDoctrine()->getManager();
$em->flush();
```

Il n'y pas besoin de persister l'entité étant donné que Doctrine la connaît déjà.

8.4.4 - Supprimer un enregistrement

Afin de supprimer en base de données une entité récupéré via doctrine, il faut procéder aux étapes suivantes :

Récupération de l'entité : voir plus loin comment récupérer des entités à partir de la base de données

```
$car = ...
```

Suppression de l'entité :

```
$em = $this->getDoctrine()->getManager();
$em->remove($car);
$em->flush();
```

La méthode "remove()" permet de marquer une entité comme étant à supprimer par Doctrine.

8.5 - Récupérer les données via les repositories

8.5.1 - Introduction

Les repositories sont des objets permettant de récupérer des entités à partir de la base de données.
Ils proposent des méthodes permettant d'effectuer des requêtes SQL et donc de récupérer des enregistrements en BDD.

Voici un exemple de classe repository :

```php
namespace App\Repository;

use App\Entity\Product;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

class ProductRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Product::class);
    }
}
```

8.5.2 - Créer un repository

Le repository est lié son entité, cela est fait dans la classe entité grâce à l'annotation suivante :

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\VoitureRepository")
 */
```

Normalement, le repository a déjà été créé en même temps que l'entité si vous avez utilisé la commande :

```
php bin/console make:entity
```

Si tel n'est pas le cas, vous pouvez utiliser la commande ci-dessous afin de créer le repository :

```
php bin/console make:entity --regenerate
```

8.5.3 - Récupérer le repository

Dans un contrôleur, il est possible de le récupérer un repository grâce au helper suivant :

```
class CarController extends AbstractController
{
    public function index(): Response
    {
        /** @var CarRepository $repository Repository */
        $repository = $this->getDoctrine()->getRepository(Car::class);
        $cars = $repository->findAll();
```

Il est aussi possible de récupérer directement le repository via le type-hinting et l'injection de dépendance (dans un contrôleur ou un service PHP) :

```
// ...
use App\Repository\CarRepository;

class CarController extends AbstractController
{
    public function index(CarRepository $repository): Response
    {
        $cars = $repository->findAll();
```

8.5.4 - Requêter avec les méthodes magiques

Le repository propose par défaut des méthodes bien pratiques.

Pour récupérer une entité à partir de son id :

```
$voiture = $repository->find($id);
```

Pour récupérer une entité à partir d'une ou plusieurs propriétés :

```
$voiture = $repository->findOneBy([
    'couleur' => 'bleu',
    'annee' => '2010',
    ...
]);
```

Pour récupérer toutes les entités :

```
$voitures = $repository->findAll();
```

Pour récupérer des entités à partir d'une ou plusieurs propriétés :

```
$voitures = $repository->findBy([
    'couleur' => 'vert',
    'annee' => '2015',
]);
```

8.5.3 - Requêter avec ses propres méthodes

8.5.3.1 - Introduction

Il est possible de créer ses propres méthodes, cela permet d'effectuer des requêtes plus complexes.

Il suffit d'ajouter une méthode dans le repository :

```
namespace App\Repository;
// ...

class ProductRepository extends ServiceEntityRepository
{
    public function maMethode()
    {
        // ...
    }
}
```

Vous pouvez alors l'appeler de la manière suivante :

```
$repository->maMethode();
```

8.5.3.1 - Requêter en SQL

Doctrine permet d'effectuer des requêtes directement avec du SQL :

```php
public function findAllGreaterThanPrice(int $price): array
{
    $conn = $this->getEntityManager()->getConnection();

    $sql = 'SELECT * FROM product p
        WHERE p.price > :price
        ORDER BY p.price ASC';

    $stmt = $conn->prepare($sql);
    $resultSet = $stmt->executeQuery(['price' => $price]);

    // returns an array of arrays (i.e. a raw data set)
    return $resultSet->fetchAllAssociative();
}
```

Les résultats retournés seront des données brutes, pas des objets.

8.5.3.2 - Requêter en DQL

Le Doctrine Query Language (DQL) ressemble au SQL mais permet de taper des requêtes en utilisant un langage de requête commun, en référençant des objets PHP à la place (par exemple, dans l'instruction FROM) :

```php
public function findAllGreaterThanPrice(int $price): array
{
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery(
        'SELECT p
        FROM App\Entity\Product p
        WHERE p.price > :price
        ORDER BY p.price ASC'
    )->setParameter('price', $price);

    // returns an array of Product objects
    return $query->getResult();
}
```

Il est possible d’utiliser du DQL grâce à la méthode "createQuery()".

Les résultats retournés seront sous la forme d’objets ou de tableaux d’objets.

8.5.4 - Le QueryBuilder

8.5.4.1 - Introduction

Doctrine fournit également un constructeur de requêtes - le QueryBuilder - une manière orientée objet d'écrire des requêtes. Il est recommandé de l'utiliser surtout lorsque les requêtes sont construites dynamiquement :

```
public function findOnline(int $maxresults = null): array
{
    $qb = $this->createQueryBuilder('p')
        ->andWhere('c.isOnline = :isOnline')
        ->setParameter('isOnline', true)
    ;

    if (null !== $maxResults) {
        $qb->setMaxResults(10):
    }

    return $qb
        ->getQuery()
        ->getResult()
    ;
}
```

Par exemple, la requête ci-dessous retourne 10 voitures de couleur bleue, triées par ordre d'année de construction :

```
$voitures = $this->createQueryBuilder('c')
    ->leftJoin('c.person', 'p')             // jointure avec la table "person" (avec un alias "p")
    ->andWhere('c.couleur = :couleur')      // définition d'un placeholder intitulé "couleur"
    ->setParameter('couleur', 'bleu')       // affectation d'une valeur au placeholder "couleur"
    ->addOrderBy('c.annee', 'ASC')          // tri croissant par année
    ->setMaxResults(10)                     // limitation à 10 résultats
    ->getQuery()                            // récupération de la requête
    ->getResult()                           // récupération des résultats
;
```

8.5.4.2 - Construire la requête

Voici les principales méthodes disponibles pour construire la requête SQL.

"select" / "addSelect" pour spécifier / ajouter une clause SQL de type "select".

"join" / "leftJoin" / "innerJoin" pour créer une jointure SQL.

"where" / "andWhere" pour spécifier / ajouter une clause SQL de type "where".
"setParameter" fonctionnant de paire avec "where" / "andWhere" pour spécifier la valeur définie par le placeholder (voir expl ci-dessus).

"orderBy" / "addOrderBy" pour spécifier / ajouter une clause SQL de type "order by" (les valeurs sont respectivement pour les tris croissants et décroissants, "ASC" et "DESC").

"setFirstResult" pour spécifier le premier argument de la clause SQL "limit" ("index" du premier résultat à retourner).

"setMaxResults" pour spécifier le second argument de la clause SQL "limit" (nombre de résultats à retourner).

8.5.4.3 - Construire une requête avec un "OR"

Il est possible de procéder de la manière suivante :

```
$qb
    $this->createQueryBuilder('p')
        ->andWhere(
            'p.code = :departmentCode'
            .' OR '.'p.name LIKE :departmentName'
            .' OR '.'p.name LIKE :regionName'
        )
        ->setParameter('departmentCode', $searchValue)
        ->setParameter('departmentName', $searchValue)
        ->setParameter('regionName', $searchValue)
;
```

Dans l'exemple ci-dessus, utilisé pour un moteur de recherche, le mot-clé recherché (stocké dans la variable $searchValue) est utilisé pour les 3 champs "departmentCode", "departmentName" et "regionName".
Il y a autant de méthode "setParameter" qu'il y a de placeholder définis dans la requête.

8.5.4.4 - Retourner les résultats

Les résultats peuvent être retournés sous différents formats, grâce aux méthodes suivantes.

"getResult()" retourne les résultats sous forme d'un tableau d'objets.

"getSingleResult()" retourne un résultat sous la forme d'un objet ou une erreur s'il n'y a pas de résultat.

"getOneOrNullResult()" retourne un résultat sous la forme d'un objet ou null s'il n'y a pas de résultat.

"getSingleScalarResult()" retourne un résultat sous la forme d'une valeur simple (= scalaire).

"getArrayResult" retourne les résultats sous la forme d'un tableau de tableaux (et non d'objets).

8.6 - Les fixtures et les données de tests

8.6.1 - Installer le bundle

Il est courant d'utiliser lors du développement d'une application des données de test.

Ces données de test sont appelées des fixtures.
Il existe un bundle permettant de créer des fixtures appelé DoctrineFixturesBundle.

Pour installer ce bundle, exécuter la commande suivante :

```
composer require --dev doctrine/doctrine-fixtures-bundle
```

ou celle-ci (orm-fixtures est un alias disponible grâce à Flex):

```
composer require --dev orm-fixtures
```

La création de fixtures étant limitée au développement de l'application et n'étant pas utilisée en production, le paramètre "--dev" est passé à la commande afin de spécifier que ce bundle ne servira que dans l'environnement "dev".

8.6.2 - Créer des fixtures

Une fois le bundle installé, vous pouvez implémenter vos fixtures dans la classe "src/DataFixtures/AppFixtures.php".

Il est possible de créer des fixtures dans plusieurs fichiers séparés mais il est recommandé de n'utiliser qu'une seule classe.

Vous pouvez utiliser la commande suivante afin de générer des classes de fixtures :

```
php bin/console make:fixtures
```

Voici un exemple de fixtures permettant d'insérer 10 nouveaux posts en base de données :

```php
namespace App\DataFixtures;

use App\Entity\Post;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 10; $i++) {
            $post = new Post();
            $post->setTitle('Titre du post n°' . $i);
            $manager->persist($post);
        }

        $manager->flush();
    }
}
```

8.6.3 - Charger des fixtures dans la BDD

Une fois les fixtures créées, il est possible de les charger dans la base de données via la commande suivante :

```
php bin/console doctrine:fixtures:load
```

Cette commande va supprimer toutes les données de la BDD et recharger les fixtures. Les identifiants (id) dans les tables de la BDD ne seront pas remis à zéro.

Si vous souhaitez que les identifiants soient remis à zéro, vous pouvez utiliser cette commande :

```
php bin/console doctrine:fixtures:load --purge-with-truncate
```

Si vous souhaitez que les données ne soient pas effacées mais incrémentées, utilisez la commande suivante :

```
php bin/console doctrine:fixtures:load --append
```

8.6.4 - Accéder à des services à partir des fixtures

Vous pouvez accéder aux autres services à partir de votre classe de fixtures (qui est elle-même un service) en utilisant l'injection de dépendances.

Voici un exemple bien pratique permettant d'encoder le mot de passe d'un utilisateur.

```
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class AppFixtures extends Fixture
{
    private UserPasswordHasherInterface $hasher;

    public function __construct(UserPasswordHasherInterface $hasher)
    {
        $this->hasher = $hasher;
    }
```

L'encodeur est récupéré grâce au type-hinting dans le constructeur.

L’encoder est alors accessible partout dans la classe.

```
// ...
    public function load(ObjectManager $manager)
    {
        $user = new User();
        $user->setUsername('admin');

        $password = $this->hasher->hashPassword($user, 'pass_1234');
        $user->setPassword($password);

        $manager->persist($user);
        $manager->flush();
    }
}
```

9 - Les formulaires

9.1 - Introduction

Le composant Form de Symfony permet de gérer les formulaires afin qu'ils soient maintenables et réutilisables.

Le formulaire Symfony a besoin d'un objet, qui sera ensuite hydraté avec les valeurs renseignées dans le formulaire (hydrater un objet consiste à le remplir avec les valeurs).

Si il s'agit d'un formulaire de création, on passera un nouvel objet (vide) au formulaire.
Si il s'agit d'un formulaire de modification, on passera un objet existant au formulaire, le formulaire sera pré-rempli avec les valeurs de cet objet.

En général, cet objet est une entité, mais ça n'est pas obligatoire.

L'objet "FormBuilder" permet de construire des formulaires.

Cela peut être fait dans le contrôleur via le helper "$this->createFormBuilder()" ou directement dans une classe dédiée.

9.2 - Créer un formulaire

La classe "FormBuilder" évoquée plus haut permet de construire les formulaires directement dans le contrôleur :

```
$form = $this->createFormBuilder($objet)
            ->add('titre', TextType::class)
            ->add('date_creation', DateType::class)
            ->getForm()
;
```

Le helper "createFormBuilder()" utilisé ci-dessus retourne une instance de cette classe "FormBuilder".

Cette dernière dispose d'une méthode add() permettant d'ajouter des champs au formulaire.

Elle prend plusieurs arguments :

- le nom du champ (devant correspondre au nom du champ dans l'objet)
- le type de champ à afficher (texte simple, zone de texte, date, liste déroulante …)
- un tableau de paramètres

La méthode "getForm()" retourne une instance du formulaire créé.

9.3 - Créer un formulaire dans une classe dédiée

L’avantage de procéder de la sorte (méthode recommandée) est :

- d’alléger le contrôleur
- de rendre le formulaire réutilisable
- d’isoler le code du formulaire

Cela consiste à créer un fichier dans le dossier "src/Form" intitulé "NomDuFormulaireType.php". Il faut donc suffixer la classe par "Type".

Le principe est le même que dans un contrôleur.

Pour récupérer une instance du formulaire dans le contrôleur, il faut alors utiliser le helper suivant

```
$this->createForm(NomDuFormulaireType::class, $objet);
```

La console propose une commande permettant de générer une classe de type Form :

```
php bin/console make:form
```

Voici un exemple :

```php
namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('titre')
            ->add('date_creation', TextType::class)
            ->add('save', SubmitType::class)
        ;
    }
}
```

Si le deuxième argument de la méthode "add()" n’est pas spécifié, Symfony déterminera automatiquement le type à utiliser.

9.4 - Les types

Comme vu précédemment, la méthode "add()" permettant de construire un formulaire prend en deuxième argument un "type".

Symfony propose de nombreux types par défaut.

En voici quelques uns dans la catégorie "texte" :

- TextType
- TextareaType
- EmailType
- IntegerType
- MoneyType
- NumberType
- PasswordType
- ...

En voici quelques uns dans la catégorie "choix" :

- ChoiceType
- EntityType
- …

En voici d’autres dans la catégorie "date" :

- DateType
- DateIntervalType
- DateTimeType
- TimeType
- …

Il en existe de nombreux autres, vous pouvez les consulter sur la documentation officielle.

Vous pouvez aussi créer vos propres types.

9.5 - Gestion du formulaire

Voici une exemple "complet" de la gestion d'un formulaire dans un contrôleur :

```php
use App\Entity\MonObjet;
use App\Form\MonFormulaireType;
use Symfony\Component\HttpFoundation\Request;

public function new(Request $request) {
    $objet = new MonObjet();
    $form = $this->createForm(MonFormulaireType::class, $objet);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($objet);
        $em->flush();

        return $this->redirectToRoute('ma_route');
    }

    return $this->render('chemin_vers_le_template/new.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

Explications :

- création de l'objet correspondant au formulaire (il s'agit ici d'un formulaire de création), dans le cas où il s'agit d'un formulaire de modification, le principe est le même mis à part le fait que l'on utilise un objet pré-rempli (bien souvent récupéré à partir de la base de données)

- création d'une instance du formulaire (en lui passant l'objet précédemment créé)

- si le formulaire est soumis et valide :
    - récupération de l'Entity Manager de Doctrine
    - enregistrement de l'objet ayant récupéré les valeurs du formulaire en base de données via l'Entity Manager
    - redirection vers une autre page (de confirmation par exemple)

- passage de l'objet formulaire au template (afin de lui permettre de l'afficher)

9.6 - Affichage du formulaire

Twig propose plein de helpers permettant d'afficher le formulaire de différentes manières.

Il est possible d'afficher le formulaire via une simple ligne de la manière suivante :

```
{{ form(form) }}
```

Il est aussi possible d'avoir plus de contrôle sur les différents champs du formulaire, par expl :

```
{{ form_start(form) }}
    {{ form_errors(form) }}

    {{ form_row(form.titre) }}
    {{ form_row(form.date_creation) }}

    {{ form_row(form.submit, { 'label': 'Enregistrer' }) }}
{{ form_end(form) }}
```

"form_row" permet d'afficher une ligne de formulaire, c'est-à-dire le libellé, le widget, l'erreur et l'aide correspondant à un champ de formulaire d'une traite.

Il est possible d'avoir encore plus de contrôle sur l'affichage des erreurs, par expl :

```
{{ form_start(form) }}
    {{ form_widget(form.titre) }}
    {% if form.titre.vars.errors|length %}
    <div class="error">
        {{ form_errors(form.titre) }}
    </div>
    {% endif %}
    ...
{{ form_end(form) }}
```

Il est aussi possible de créer des thèmes de formulaires appelés "form themes" afin de personnaliser les formulaires et de les rendre réutilisables.

Se référer à la documentation officielle pour en savoir plus.

9.7 - La validation

Afin de valider un formulaire, il faut en fait "valider l'objet". La validation d'un formulaire Symfony repose en effet sur la validité des données de l'objet lié au formulaire.

Il est possible de spécifier ces "contraintes" directement au dessus des différents champs de l'entité grâce à des annotations sur le même principe que celui utilisé pour spécifier les annotations Doctrine liées à la base de données.

Il existe de nombreuses contraintes de validation proposées par défaut par Symfony, elles peuvent être consultées ici.

Si vous souhaitez en savoir plus sur la validation, vous pouvez consulter cette page.

Voici un exemple simple avec une entité Post contenant deux champs :

- un champ titre obligatoire
- un champ dateCreation de type date obligatoire lui aussi

```php
namespace App\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Post
{
    /**
     * @Assert\NotBlank              // le champ ne peut pas être vide
     */
    public $titre;

    /**
     * @Assert\NotBlank              // le champ ne peut pas être vide
     * @Assert\Type("\DateTime")     // la valeur du champ doit être de type DateTime
     */
    protected $dateCreation;
}
```

9.8 - Désactiver la validation HTML5

La validation HTML5 est activée par défaut dans Symfony. Il est plus facile de déboguer si la validation HTML5 est désactivée.

Il est possible de la désactiver de deux manières.

Soit en ajoutant à votre formulaire un attribut novalidate dans votre le template :

```
{{ form_start(form, {attr: {'novalidate': 'novalidate'}}) }}
```

Soit en déclarant cet attribut directement dans la classe type du formulaire :

```
// ...
public function configureOptions(OptionsResolver $resolver): void
{
    $resolver->setDefaults([
        'attr' => [
            'novalidate' => 'novalidate',
        ]
    ]);
}
```

10 - La sécurité

10.1 - Introduction

Symfony fournit par défaut certains outils de sécurité liés à HTTP, comme les cookies de session sécurisés et la protection CSRF. Ces fonctionnalités d'authentification et d'autorisation sont fournies par le SecurityBundle (normalement installé par défaut dans le framework Symfony).

Ce bundle fournit un fichier de configuration "config/packages/security.yaml" déjà pré-rempli.

Ce fichier est divisé en plusieurs parties.

```yaml
security:
    enable_authenticator_manager: true
```

La clé "enable_authenticator_manager" permet de spécifier à Symfony d'utiliser le nouveau système basé sur l'authentification (lorsque la valeur est *true*) existant depuis la version 5.3.

```yaml
security:
    # ...

    #
https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

La section "password_hashers" configure quel "encodeur" de mot de passe doit être utilisé.

```yaml
security:
    # ...

    #
https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        users_in_memory: { memory: null }
```

La section "providers" permet de définir un ou plusieurs fournisseurs d'utilisateurs.

"users_in_memory" présent par défaut, permet de charger les utilisateurs à partir d’un fichier de configuration.

```yaml
security:
    # ...

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-authentication

            # https://symfony.com/doc/current/security/impersonating_user.html
            # switch_user: true
```

La section "firewalls" correspond au pare-feu et est la plus importante.

Le pare-feu "dev" s'assure que tous que les outils de développement de Symfony ne soient pas bloqués.
Le pare-feu "main" définit les parties de l’application qui sont sécurisées et la façon dont les utilisateurs pourront s'authentifier.

```
security:
    # ...

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

La section "access_control" permet de contrôler les autorisations requises pour effectuer une action spécifique ou visiter une URL spécifique.

Nous reviendrons sur ces différents points dans les chapitres suivants.

10.2 - Création des utilisateurs

Les autorisations dans Symfony sont toujours liées à un objet utilisateur.

Il faut donc créer une classe utilisateur (il s'agit souvent d'une entité Doctrine).

Cette classe devra implémenter les interfaces UserInterface et PasswordAuthenticatedUserInterface.

La manière la plus simple de créer cette classe utilisateur est d'utiliser la commande suivante :

```
php bin/console make:user
```

Cette commande est intéractive et va vous demander de renseigner certaines informations.

```
 The name of the security user class (e.g. User) [User]:
 > BackUser

 Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
 > yes

 Enter a property name that will be the unique "display" name for the user (e.g. email,
username, uuid) [email]:
 > email

 Will this app need to hash/check user passwords? Choose No if passwords are not needed
or will be checked/hashed by some other system (e.g. a single sign-on server).

 Does this app need to hash/check user passwords? (yes/no) [yes]:
 > yes
```

Il vous est demandé :

- le nom de classe utilisateur (ici "BackUser" par exemple)
- si cette classe est une entité Doctrine (dans la majorité des cas "oui")
- quelle propriété de l'entité sera utilisée comme login (l'email en général)
- si les mots de passe doivent être encodés (recommandé)

Cette commande va créer une entité utilisateur contenant les méthodes nécessaires à la gestion de la sécurité ainsi que le repository associé.

De plus, le fichier configuration "config/packages/security.yaml" sera mis à jour en conséquence.

```yaml
security:
    # ...

    providers:
        app_user_provider:
            entity:
                class: App\Entity\BackUser
                property: email

    # ...

    main:
        lazy: true
        provider: app_user_provider
```

Doctrine est utilisé pour charger l'entité BackUser (spécifiée lors de la création de la classe utilisateur) en utilisant la propriété email comme identifiant.

Pensez ensuite à mettre à jour la structure de la base de données via la commande :

```
php bin/console doctrine:schema:update --force
```

Maintenant que la table des utilisateurs est créée en base de données, il faut créer au moins un utilisateur.
Il est possible de le faire directement dans la base de données ou en utilisant les fixtures. Voici un exemple de fixtures de création d'un utilisateur "BackUser"

```php
namespace App\DataFixtures;

use App\Entity\BackUser;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        $backUser = new BackUser();
        $backUser->setEmail('admin@gmail.com');
        $backUser->setPassword('mot_de_passe_encodé');
        $backUser->setRoles(['ROLE_ADMIN']);
        $manager->persist($backUser);
```

```
        $manager->flush();
    }
}
```

Attention, le mot de passe fourni doit être encodé !

Vous pouvez encoder un mot de passe en utilisant la console de la manière suivante :

```
php bin/console security:hash-password
```

Vous obtiendrez la sortie suivante :

```
--------------- ------------------------------------------------------------
 Key             Value
--------------- ------------------------------------------------------------
 Hasher used     Symfony\Component\PasswordHasher\Hasher\MigratingPasswordHasher
 Password hash   $2y$13$xU7JpR/4NupGCn0NSxbjNenD3Wk7pkTMcaAoBknt7pubhR5HGbPK2
--------------- ------------------------------------------------------------
```

Où "Password hash" représente le mot de passe encodé.

Une fois les fixtures créées, pensez à mettre à jour les données de la base de données via la commande :

```
php bin/console doctrine:fixtures:load
```

10.3 - Création de l’authentification

Il va maintenant falloir mettre l’authentification en place. Il existe différents systèmes tels qu’un formulaire de connexion, un jeton API, etc…

Nous utiliserons dans le cas présent un formulaire de login, ce qui est le cas de la plupart des sites web. Cette fonctionnalité est fournie par le "form login authenticator".

Utiliser la commande intéractive suivante afin de vous simplifier la vie :

```
php bin/console make:controller
```

Cette commande va :

- créer un un contrôleur "src/Controller/LoginController"
- créer le template correspondant "templates/login/index.html.twig"

```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class LoginController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function index(): Response
    {
        return $this->render('login/index.html.twig', [
            'controller_name' => 'LoginController',
        ]);
    }
}
```

Il faut ensuite activer le "form login authenticator" dans le fichier "config/packages/security.yaml".

```yaml
security:
    # ...

    firewalls:
        main:
            # ...
            form_login:
                # "app_login" is the name of the route created previously
                login_path: app_login
                check_path: app_login
                enable_csrf: true
            logout:
                path: app_logout
                target: app_any_route
```

Une fois activé, le système de sécurité redirige les visiteurs non authentifiés vers le l'URL correspondant à la route "login_path" lorsqu'ils essaient d'accéder à un endroit sécurisé. "enable_csrf" permet d'activer la protection CSRF sur le formulaire de login.

Dans la section "logout", la clé "path" permet de spécifier la route permettant de se déconnecter. La clé "target" permet de rediriger l'utilisateur vers une route après la déconnexion.

Modifier le "LoginController" pour rendre le formulaire de login :

```php
// ...
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;
class LoginController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function index(AuthenticationUtils $authenticationUtils): Response
    {
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();
        return $this->render('login/index.html.twig', [
            'controller_name' => 'LoginController',
            'last_username' => $lastUsername,
            'error'         => $error,
        ]);
    }

    /**
     * @Route("/logout", name="app_logout", methods={"GET"})
     */
    public function logout(): void
```

```
    {
        // controller can be blank: it will never be called!
        throw new \Exception('Don\'t forget to activate logout in security.yaml');
    }
}
```

Pour finir, modifier le template "templates/login/index.html.twig" :

```
{% extends 'base.html.twig' %}
{# ... #}

{% block body %}
    {% if error %}
        <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

    <form action="{{ path('app_login') }}" method="post">
        <label for="username">Email:</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}"/>

        <label for="password">Password:</label>
        <input type="password" id="password" name="_password"/>

        {# If you want to control the URL the user is redirected to on success
        <input type="hidden" name="_target_path" value="/account"/> #}
        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">
```

```
        <button type="submit">login</button>
    </form>
{% endblock %}
```

Pour résumer, voici l'ensemble du processus du système d'authentification pour formulaire de login :
1- L'utilisateur tente d'accéder à une ressource qui est protégée;
2- Le pare-feu lance le processus d'authentification en redirigeant l'utilisateur vers le formulaire de connexion;
3- La page de login rend le formulaire de connexion;
4- L'utilisateur soumet le formulaire de connexion;
5- Le système de sécurité authentifie l'utilisateur ou le renvoie vers le formulaire de connexion si l'authentification échoue.

Remarque : lorsque vous utilisez plusieurs zones sécurisées (par exemple une pour le front et une pour le back, vous devez créer plusieurs pares-feu.
Afin que Symfony puisse associer le bon pare-feu à la bonne zone sécurisée, vous pouvez utiliser le paramètre pattern dans le fichier "config/packages/security.yaml".

10.4 - Récupérer l’objet utilisateur

10.4.1 - À partir d’un contrôleur

Il est possible de le récupérer grâce à la méthode "getUser()" de la classe parente "AbstractController".

```php
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class ProfileController extends AbstractController
{
    public function index(): Response
    {
        // ...
        /** @var \App\Entity\User|null $user */
        $user = $this->getUser();
        // ...
    }
}
```

10.4.2 - À partir d’un service

Il est possible de le récupérer grâce au service "Security".

```php
use Symfony\Component\Security\Core\Security;

class ExampleService
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function someMethod()
    {
        /** @var \App\Entity\User|null $user */
        $user = $this->security->getUser();

        // ...
    }
}
```

10.4.3 - À partir d’un template

Il est possible de le récupérer grâce à la variable globale "app".

```
<p>Email: {{ app.user.email }}</p>
```

10.5 - Les autorisations

10.5.1 - Introduction

Les utilisateurs peuvent maintenant se connecter à l'application via le formulaire de connexion.
Les autorisations ont pour but de décider si un utilisateur peut accéder à des ressources.

Le processus d'autorisation comporte deux aspects différents :

- L'utilisateur reçoit un rôle spécifique lorsqu'il se connecte (par exemple ROLE_ADMIN).
- L'ajout de code afin de définir si une ressource (par exemple, une URL, un contrôleur) nécessite un "attribut" spécifique (par exemple, un rôle comme ROLE_ADMIN) pour être accessible.

10.5.2 - Les rôles

Lorsqu'un utilisateur se connecte, Symfony appelle la méthode getRoles() sur l'objet User afin de déterminer les rôles de cet utilisateur. Dans la classe User, les rôles sont un tableau stocké dans la base de données et chaque utilisateur se voit toujours attribuer au moins un rôle : le rôle "ROLE_USER".

```php
class User
{
    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    // ...
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }
}
```

Un rôle est juste une chaîne de caractère et doit toujours être préfixé par "ROLE_" (par exemple ROLE_EDITOR).

Il est donc possible de créer plusieurs rôles qui seront ensuite utilisés pour accorder l'accès à des sections spécifiques du site.

Plutôt que de donner plusieurs rôles à chaque utilisateur, il est possible de définir des règles d'héritage des rôles en créant une hiérarchie de rôles.

```
security:
    # ...

    role_hierarchy:
        ROLE_ADMIN:       ROLE_EDITOR
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Dans l'exemple ci-dessus, les utilisateurs ayant le rôle ROLE_ADMIN auront également le rôle ROLE_EDITOR.

Les utilisateurs ayant le rôle ROLE_SUPER_ADMIN auront automatiquement les rôles ROLE_ADMIN et ROLE_ALLOWED_TO_SWITCH ainsi que le rôle ROLE_EDITOR (hérité de ROLE_ADMIN).

10.5.3 - Sécuriser les motifs d’URLs (access control)

Sécuriser des motifs d'URLs est la façon la plus simple mais la moins flexible pour sécuriser l’application. Cela peut se faire via le fichier "config/packages/security.yaml" :

```yaml
security:
    # ...

    firewalls:
        # ...
        main:
            # ...

    access_control:
        # allow unauthenticated users to access the login form
        - { path: ^/admin/login, roles: PUBLIC_ACCESS }

        # but require authentication for all other admin routes
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Il est possible de définir autant de motifs d'URLs que l’on souhaite - chacun est une expression régulière. Mais il n'y aura qu'une seule correspondance par requête : Symfony commence en haut de la liste et s'arrête lorsqu'il trouve la première correspondance.

Chaque contrôle d'accès peut également correspondre à l'adresse IP, au nom d'hôte et aux méthodes HTTP.

Le fait de faire précéder le chemin d'accès de ^ (dans "^/admin") signifie que seules les URL commençant par le motif "/admin" sont prises en compte.
Attention : un chemin d'accès "/admin" (sans ^) correspondrait à "/admin/foo" mais aussi à des URLs comme "/foo/admin" !

Dans l’exemple ci-dessus, toutes les URLs commençant par "/admin" nécessitent une authentification.
Si l’URL du formulaire de login est "/admin/login", elle commence elle aussi par "admin" et n’est donc pas accessible ! Du coup, impossible de s’authentifier …
Pour pallier ce problème, il est possible d’utiliser l'attribut de sécurité "PUBLIC_ACCESS" afin d’exclure certaines URLs de l'accès non authentifié (et donc dans notre exemple le formulaire de login).

10.5.3 - Sécuriser les contrôleurs

Vous pouvez refuser l'accès depuis l'intérieur d'un contrôleur.

```
public function adminDashboard(): Response
{
    $this->denyAccessUnlessGranted('ROLE_ADMIN');

    // or add an optional message - seen by developers
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a page without having
ROLE_ADMIN');
}
```

Et c'est tout ! Si l'accès n'est pas accordé, une exception spéciale AccessDeniedException est levée et aucun autre code de votre contrôleur n'est appelé. Ensuite, l'une des deux choses suivantes se produira :

- Si l'utilisateur n'est pas encore connecté, il lui sera demandé de se connecter (par exemple, il sera redirigé vers la page de connexion);
- Si l'utilisateur est connecté, mais ne possède pas le rôle "ROLE_ADMIN", la page 403 "access denied" (personnalisable) s'affichera.

Grâce au SensioFrameworkExtraBundle, il est possible d'utiliser des annotations pour sécuriser un contrôleur.

```php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

/**
 * Requires ROLE_ADMIN for all the actions of this controller
 *
 * @IsGranted("ROLE_ADMIN")
 */
class AdminController extends AbstractController
{
    /**
     * Require ROLE_SUPER_ADMIN only for this action
     *
     * @IsGranted("ROLE_SUPER_ADMIN")
     */
    public function adminDashboard(): Response
    {
        // ...
    }
}
```

L'annotation "IsGranted" permet donc de restreindre un contrôleur (et du coup toutes ses routes) ou une méthode de contrôleur (et du coup la route associée) à un rôle précis.

Elle permet aussi d'utiliser des voters personnalisés pour restreindre l'accès en fonction de variables passées au contrôleur. Dans l'exemple ci-dessous, la variable "$post" sera passée au voter personnalisé grâce à l'annotation IsGranted et plus particulièrement la partie subject="post".

```
/**
 * @Route("/posts/{id}")
 *
 * @IsGranted("ROLE_ADMIN")
 * @IsGranted("POST_SHOW", subject="post")
 */
public function show(Post $post)
{
}
```

L'utilisation de voters personnalisés est un moyen puissant de gérer les autorisations et permet aussi de regrouper tout le code dans une classe dédiée.

10.5.4 - Sécuriser les templates

Si vous voulez vérifier si l'utilisateur actuel a un certain rôle, vous pouvez utiliser la fonction d'aide intégrée is_granted() dans n'importe template.

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="/lien_pour_supprimer">Delete</a>
{% endif %}
```

Si l'utilisateur n'a pas le rôle "ROLE_ADMIN", il ne verra pas le lien de suppression ci-dessus.

Ou encore :

```
{% if is_granted('display_contracts', backUser) %}
    <a href="/lien_vers_contrats">Contrats</a>
{% endif %}
```

L'utilisateur "backUser" sera passé au voter avec la clé "display_contracts". Si l'autorisation n'est pas accordée, il ne verra pas le lien vers les contrats.

10.5.5 - Sécuriser les autres services

Vous pouvez vérifier l'accès n'importe où dans votre code en injectant le service Security (en utilisant le type-hinting dans le constructeur).

```php
use Symfony\Component\Security\Core\Security;

class MaClasse
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function doSomething()
    {
        if ($this->security->isGranted('ROLE_EDITOR')) {
            // ...
        }
    }
}
```

11 - Le Service Container

11.1 - Introduction

Votre application est pleine d'objets utiles : un objet "Mailer" peut vous aider à envoyer des courriers électroniques, tandis qu'un autre objet peut vous aider à enregistrer des données dans la base de données. Presque tout ce que votre application "fait" est en fait réalisé par l'un de ces objets. Et chaque fois que vous installez un nouveau bundle, vous avez accès à encore plus d'objets !

Dans Symfony, ces objets utiles sont appelés services et chaque service vit à l'intérieur d'un objet très spécial appelé le Service Container. Ce conteneur vous permet de centraliser la façon dont les objets sont construits. Il vous facilite la vie, favorise une architecture solide et est super rapide !

11.2 - Configuration

La documentation suppose que vous utilisez la configuration de service suivante, qui est la configuration par défaut pour un nouveau projet :

```yaml
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name
    App\:
        resource: '../src/'
        exclude:
            - '../src/DependencyInjection/'
            - '../src/Entity/'
            - '../src/Kernel.php'

    # order is important in this file because service definitions
    # always *replace* previous ones; add your own service configuration below
```

11.3 - Récupérer et utiliser les services

Au moment où vous démarrez une application Symfony, votre conteneur contient déjà de nombreux services. Ce sont comme des outils : ils attendent que vous les utilisiez.

Pour "demander" un service au Service Container, il suffit de type-hinter (c'est-à-dire indiquer le nom de la classe ou de l'interface) le service, au niveau de l'argument de la méthode. Cela est possible grâce à l'auto-wiring et l'injection de dépendance.

Pour lister les services bénéficiant de l'auto-wiring, vous pouvez vous servir de la console en utilisant la commande suivante :

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

Lorsque vous utilisez ces type-hints dans vos méthodes de contrôleur ou dans vos propres services, Symfony vous passera automatiquement l'objet de service correspondant à ce type.

Voici un exemple dans une classe contrôleur :

```php
use Psr\Log\LoggerInterface;

class DefaultController
{
     public function index(LoggerInterface $logger)
     {
        $logger->info('About to find a happy message!');
        // ...
    }
}
```

Il suffit de type-hinter le service dans la méthode concernée.

Voici un autre exemple dans une classe PHP standard (et donc un service aussi !) :

```php
// ...
use Psr\Log\LoggerInterface;

class MaClasse
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function getHappyMessage(): string
    {
        $this->logger->info('About to find a happy message!');
        // ...
    }
}
```

Il suffit de type-hinter le service dans le constructeur.

12 - Liens utiles

Voici quelques liens supplémentaires :

- les symfony casts (tutoriels vidéos) : https://symfonycasts.com/courses
- les "coding standards" de Symfony : https://symfony.com/doc/current/contributing/code/standards.html
- les "best practices" de Symfony : https://symfony.com/doc/current/best_practices/index.html
- un article sur le code de qualité : https://openclassrooms.com/fr/courses/5489656-construisez-un-site-web-a-l-aide-du-framework-symfony-4/5517036-qu-est-ce-qu-un-code-de-qualite
- une application "démo" de Symfony dont vous pouvez vous inspirer : https://github.com/symfony/demo
- le site "Stack Overflow" où vous pouvez poser vos questions / trouver des réponses : https://stackoverflow.com/questions/tagged/symfony
- la version en ligne du livre "Pro Git", référence concernant le versionning avec Git : https://git-scm.com/book/fr/v2