

Symfony 5

Guide du cours

Table of Contents

1.	Configuration de base : Composer, Git et XDebug	7
2.	Configuration de base : Visual Studio Code	8
3.	Installation de Symfony et création d'un projet de base	9
4.	Installation de packages dans un projet Symfony Flex	10
5.	Symfony avec Apache. Configuration des Virtual Hosts	11
	Création d'un serveur virtuel (virtual host) en Windows	11
	Création d'un serveur virtuel (virtual host) en OSX	13
6.	Routing	15
5.1.	Les routes et la réécriture des URLs	15
5.2.	Les routes contenant de paramètres	19
6.1.	Les contraintes dans les routes	21
6.2.	Les valeurs par défaut pour les paramètres du routing	22
6.3.	Paramètres optionnels	23
6.4.	Les valeurs alternatives pour les paramètres	23
6.6.7.	Controllers et actions	24
	Procédure de création manuelle d'un controller	24
7.1.	L'objet Response	27
7.2.	L'objet Request	28
7.3.	Types de réponses d'un controller: render, redirect, redirectToRoute et forward	29
7.4.1.	Redirect	29
7.4.2.	RedirectToRoute	30
7.4.3.	Forward	32
7.4.4.	Render	32
8.1.	Création d'un controller avec l'assistant	32
8.2.	8. Gestion basique d'erreurs	33
8.3.	Créer une vue pour chaque erreur à gérer en utilisant les conventions de Symfony	33
9.1.	Modifier la réponse http du serveur	35
9.2.	Lancer une exception	36
9.	Les Vues. Le moteur de templates TWIG	37
	Création d'un template en utilisant Twig	38
	Les variables	39
9.2.1.	Les conditions	41
9.2.2.	Les boucles	42

9.2.3.	Les filtres	45
	Héritage de templates en TWIG (I)	46
	Création de Templates en TWIG (II).....	49
	Vider un bloc hérité d'un template.....	50
	Incruster une action du controller dans une vue.....	52
9.3.	Faire appel à une action depuis la vue.....	54
9.4.10.	Les environnements de développement et production	55
9.5.	Les fichiers .env et .env.local	56
9.6.	Le Web Profiler	57
9.7.	Afficher le contenu des variables avec dump	57
10.1.	11. Les Services	58
10.2.	Utilisations des services inclus dans Symfony	58
10.3.	Création de nos propres services.....	61
11.1.	Injecter les services dans le controller.....	62
11.2.	Injection de paramètres dans le service (I).....	64
11.3.	Injection de paramètres dans le service (II).....	66
11.4.	Utiliser un service dans un autre service	66
11.5.	Injection de paramètres dans le service (II).....	68
11.6.	12. Cookies.....	71
12.	13. Session	72
13.	14. Obtenir de données du GET et POST	74
15.1.	15. Le modèle : création d'entités et de la BD.....	75
15.2.	Présentation de Doctrine.....	75
15.3.	Configuration de Doctrine et des paramètres de la BD.....	76
15.4.	Création des entités et mise à jour de la BD.....	77
16.1.	Rajouter/effacer des propriétés d'une entité.....	78
16.2.	16. Le modèle : les relations	80
16.3.	Relation Many-To-One.....	80
16.4.	Explication du code généré par l'assistant.....	84
16.5.	Relation Many-To-Many	87
	Relation One-To-One	89
	Relation récursive (self-association)	90
16.5.1.	Relation récursive d'un à plusieurs	90
16.5.2.	Relation récursive de plusieurs à plusieurs.....	91
17.	17. Le modèle : accès à la BD avec Doctrine.....	93

	SELECT	93
	INSERT et UPDATE.....	96
17.2.1.	INSERT	96
17.2.2.	UPDATE	97
17.2.3.	DELETE.....	97
17.1. 18.1. 17.2.	Le modèle : persistance	98
	Transitivité en Cascade	101
	Encapsulation.....	104
19.	Héritage de classes et implémentation dans la BD.....	106
18.1.	Single Table Inheritance.....	107
18.2.	Class Table Inheritance	109
19.20. 19.1.	Accès à la BD avec DQL	110
19.2.	SELECT	111
20.1.1.	Requête qui renvoi un array d'arrays	111
20.1.2.	Requête qui renvoi un array d'objets	112
20.1.3.	Regular Joins et Fetch Joins	113
20.2.	UPDATE	116
20.3.	Exercices DQL.....	116
21.	Accès à la BD avec DQL en utilisant les classes Repositoires.....	118
22.	Accès à la BD avec Query Builder.....	120
23.23. 23.1.	Formulaires en Symfony	125
23.2.	Création d'un formulaire indépendant.....	125
23.3.	Création une classe de formulaire	125
23.4.	Création d'un formulaire associé à une entité existante.....	129
23.5.	Types des champs du formulaire	130
23.6.	Propriétés des champs du formulaire.....	131
23.1.	Méthode et Action	133
23.2.	Boutons dans les formulaires (bonnes pratiques)	136
23.3.	Rendu du formulaire dans la vue	137
23.4.	Résumé : création et personnalisation de base d'un formulaire.....	138
23.5.	Traitement des données du formulaire (explication de base).....	139
23.6.	Bonnes pratiques pour créer de formulaires en Symfony.....	143
23.7.	Style des formulaires.....	144
	Formulaires concernant plusieurs entités	145

	Formulaire contenant une liste déroulante d'entités filtrés	147
24.	Upload de fichiers en utilisant un formulaire	152
	Stockage dans le serveur d'une seule image pour chaque entité	152
	Problèmes dans l'upload	155
23.8. 25.	AJAX en Symfony avec Axios	157
	Exemple d'appel AJAX avec un formulaire	157
24.1.	Utilisation de blocs dans twig avec AJAX	159
24.2.	Ajax et Axios avec script externe au Twig (sans Webpack)	163
25.1. 26.	AJAX en Symfony (Vanilla JS)	164
25.2.	Exemple d'appel AJAX avec un formulaire	164
25.3.	Utilisation de blocs dans twig avec AJAX	167
26.1. 27.	Renvoi d'un array d'objets en JSON	170
26.2.	Renvoi d'un array d'objets en JSON (repo)	170
	Renvoi d'un array d'objets en JSON (DQL)	170
27.1. 28.	Doctrine Fixtures	171
27.2.		
29.	Authentification : inscription et login/password	174
	Configuration de la sécurité et création d'un formulaire de login	174
29.1.		
29.2.	(En cours, cette doc. appartient à Symfony 4) Traduction des messages de succès/erreur	183
29.3.	Création d'un formulaire d'inscription	184
30.	Logout	185
31.	Accès à l'objet app.user	187
32.1. 32.	Authentication et Rôles	188
32.2.	Gestion de rôles	188
	Contrôle d'accès par rôles	191
32.2.1.	Dans security.yaml	191
32.3. 32.2.2.	Dans le controller	193
33.1. 32.2.3.	Dans les vues	194
33.2.	Gestion de l'erreur "Access Denied" (exception) en utilisant une classe propre	195
33.3. 33.	Fenêtre modale pour le login	197
	Description générale	197
	Adaptation à Ajax et fenêtre modale	198
	Formulaire d'enregistrement	202
34.	Envoi du mail (Swift Mailer)	203
35.	Pagination	205

36.	JS et CSS avec Webpack encore	208
	Installation de Webpack Encore et de node_modules	208
	Configurer Webpack Encore	209
	Lancer Webpack.....	209
	Utiliser le code dans les vues	210
36.1.	37. Encore et Bootstrap	211
36.3.	38. Intégration de boutons de paiement Paypal	212
36.4.		

1. Configuration de base : Composer, Git et XDebug

- Installez **Composer** (voir guide)
- Installez **Git**
- Installez **XDebug** (voir guide)
- Changez la configuration dans php.ini pour activer XDebug :

```
[XDebug]
zend_extension = "C:\xampp\php\ext\php_xdebug-2.5.5-7.1-vc14.dll"
xdebug.remote_host = "127.0.0.1"
xdebug.remote_enable = 1
xdebug.remote_handler = "dbgp"
xdebug.idekey = netbeans-xdebug
xdebug.profiler_append = 0
xdebug.profiler_enable = 0
xdebug.profiler_enable_trigger = 0
xdebug.profiler_output_dir = "c:/xampp/tmp/xdebug"
xdebug.profiler_output_name = "cachegrind.out.%t-%s"
xdebug.remote_autostart = 0
xdebug.remote_connect_back = 0
xdebug.remote_host = "127.0.0.1"
xdebug.remote_port = 9000
xdebug.remote_handler = "dbgp"
xdebug.remote_mode = req
xdebug.remote_log = "c:/xampp/tmp/xdebug/xdebug_remot.log"
xdebug.show_local_vars = 9
xdebug.trace_output_dir = "c:/xampp/tmp"
```

2. Configuration de base : Visual Studio Code

1. Installez [Visual Studio Code](#)
2. Installez ces extensions :
 - [Intelephense](#)
 - [PHP NameSpace Resolver](#)

3. Installation de Symfony et création d'un projet de base

- Téléchargez l'exécutable **d'installation** de Symfony :

<https://symfony.com/download>

Lancez le fichier que vous venez de télécharger et qui installera l'exécutable **symfony.exe** dans le dossier que vous-même pouvez spécifier. Ce fichier exécutable permet de lancer un certain ensemble d'actions concernant le framework (ex : créer un projet Symfony). Voyons un exemple :

- Créez le **squelette** d'une application web. Allez dans votre dossier htdocs (ou le dossier de votre choix) et lancez l'exécutable de Symfony (qui se trouve dans le path) avec les paramètres suivants :

```
symfony new --full Projet1Symfony5
```

Cette ligne créera un dossier contenant la structure d'un projet de base Symfony.

Note : si vous ne voulez pas installer Symfony vous pouvez toujours créer vos projets en utilisant composer directement

```
composer create-project symfony/website-skeleton Projet1Symfony5
```

- Lancez le **serveur web interne** de Symfony depuis la console (on utilisera Apache plus tard). Allez dans le dossier de votre projet et tapez :

```
symfony server:start
```

Vous pouvez toujours utiliser Apache comme serveur, mais pour la phase de développement c'est plus simple d'utiliser le serveur déjà inclus dans Symfony. Testez le bon fonctionnement du serveur en regardant les messages sur la console et en tapant l'adresse suivante dans le navigateur :

<http://localhost:8000/>

4. Installation de packages dans un projet Symfony Flex

Symfony Flex est un plug-in Composer installé **par défaut lors de la création d'une nouvelle application Symfony** et automatisant les tâches les plus courantes des applications Symfony. Flex modifie le comportement de base de *require*, *update* et *remove*. Beaucoup de packages créés pour la 1 de Symfony ont une *recipe* (« recette ») : **un ensemble d'instructions pour installer et activer un package dans un projet Symfony**. On peut, en plus, rajouter Flex dans un projet Symfony existant d'une version précédente (qui n'aie pas Flex). Plus de documentation ici :

<https://symfony.com/doc/current/setup.html#creating-symfony-applications>

5. Symfony avec Apache. Configuration des Virtual Hosts

Considérez qu'on a une application web qui se trouve dans le dossier

```
C:/xampp/htdocs/Symfony5/projet1symfony/web
```

Normalement on devrait saisir cette URL pour y accéder :

```
localhost/Symfony4/projet1symfony/web
```

Nous devons savoir qu'Apache permet d'utiliser la technique de réécriture d'URL. Cela nous permettra, par exemple, d'avoir un projet qui se trouve dans

```
C:/xampp/htdocs/Symfony4/projet1symfony/web
```

Et en accéder en utilisant tout simplement cette URL :

```
projet1symfony.localhost
```

Nous devons configurer cette correspondance dans le fichier `/xampp/apache/conf/extra/httpd-vhosts.conf`. Le nom `vhosts` vient du fait qu'on est en train de créer un **serveur virtuel**.

C'est Apache qui transforme une URL dans autre, mais toujours selon nos règles.

5.1. Création d'un serveur virtuel (virtual host) en Windows

Pour créer et utiliser un serveur virtuel suivez ces pas :

1. Activez d'abord la réécriture de l'URL dans la configuration d'Apache ainsi que la lecture des serveurs virtuels dans `httpd-vhosts`. Juste ouvrez le fichier `c:\xampp\apache\conf\httpd.conf` et effacez les commentaires de ces deux lignes (si elles sont commentées)

```
LoadModule rewrite_module modules/mod_rewrite.so  
Include conf/extra/httpd-vhosts.conf
```

2. Modifiez (ou créez) le fichier `c:\xampp\apache\conf\extra\httpd-vhosts.conf` en rajoutant :

```
<VirtualHost *:80>  
    ServerName projet1symfony.localhost  
    DocumentRoot "C:/xampp/htdocs/Symfony4/projet1symfony/public"  
    <Directory "C:/xampp/htdocs/Symfony4/projet1symfony/public">  
        AllowOverride All  
        Order Allow,Deny  
        Allow from All  
    </Directory>
```

```
</VirtualHost>
```

Pour chaque nouveau projet vous devez rajouter la première section en modifiant le `ServerName`, `DocumentRoot` et `Directory`.

Pour pouvoir continuer à utiliser le serveur **localhost** normalement vous devez rajouter sa **configuration** :

```
<VirtualHost *:80>
    ServerName localhost
    DocumentRoot "C:/xampp/htdocs"
    <Directory "C:/xampp/htdocs">
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

3. Installez l'apache-pack qui créera les règles d'écriture d'url pour le projet (Apache en aura besoin). Dans le dossier du projet, tapez :

```
composer require symfony/apache-pack
```

(Répondez "y" pour accepter l'installation)

4. Rajoutez dans le fichier **hosts** de `c:\Windows\System32\drivers\etc\hosts`:

```
127.0.0.1      projet1Symfony.localhost
```

(Vous devez démarrer notepad comme administrateur)

5. Redémarrez le serveur Apache et allez sur le site :

```
http://projet1symfony.localhost/
```

Une page de bienvenue devrait s'afficher

Exercice : création d'un projet contenant l'application skeleton

Créez un deuxième projet `projet2Symfony` selon la procédure précédente

Création d'un serveur virtuel (virtual host) en OSX

1. Activez la lecture de httpd-vhosts dans le fichier httpd.conf: ouvrez **xampp** et cliquez sur le bouton Config pour ouvrir ce fichier de configuration d'Apache.

5.2. Note: Le fichier se trouve dans Applications/xampp/xamppfiles/etc

Une fois le fichier ouvert, effacez les commentaires de ces deux lignes (si elles sont commentées)

```
Include conf/extra/httpd-vhosts.conf
```

Activez aussi la réécriture de l'URL dans la configuration d'Apache e

```
LoadModule rewrite_module modules/mod_rewrite.so
```

2. Modifiez (ou créez) le fichier **/Applications/XAMPP/xamppfiles/etc/extra/httpd-vhosts.conf** en rajoutant :

```
<VirtualHost *:80>
    ServerName projet1Symfony.localhost
    DocumentRoot "/Applications/XAMPP/xamppfiles/htdocs/Symfony4/projet1Symfony/public"
    <Directory "/Applications/XAMPP/xamppfiles/htdocs/Symfony4/projet1Symfony/public">
        AllowOverride All
        Order Allow,Deny
        Allow from All
    </Directory>
</VirtualHost>
```

Pour chaque nouveau projet vous devez rajouter la première section en modifiant le ServerName, DocumentRoot et Directory.

Pour pouvoir continuer à utiliser le serveur **localhost** normalement vous devez rajouter sa **configuration** :

```
<VirtualHost *:80>
    ServerName localhost
    DocumentRoot "/Applications/XAMPP/xamppfiles/htdocs"
    <Directory "/Applications/XAMPP/xamppfiles/htdocs">
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

3. Dans le dossier de votre projet, installez l'apache-pack qui créera les règles d'écriture d'url pour le projet (Apache en aura besoin). Dans le dossier du projet, tapez :

```
php composer.phar require symfony/apache-pack
```

(Répondez "y" pour accepter l'installation)

4. Rajoutez dans cette ligne dans le fichier **hosts** :

```
127.0.0.1      projet1Symfony.localhost localhost
```

Pour éditer le fichier hosts :

1. Ouvrez la console
2. Tapez `cd /`
3. Tapez `sudo nano etc/hosts`
4. Tapez votre mot de passe
5. Rajoutez la ligne indiquée
6. Enregistrez le fichier avec `CONTROL-O` et puis `Enter`, sortez du logiciel avec `CONTROL-X` et puis `Enter`

6. Routing

Objectif : comprendre l'utilité et le fonctionnement de base des routes dans un framework (Symfony) et être capable de créer des routes vers les actions des controllers

Les routes et la réécriture des URLs

Une **route** nous **sert à accéder à une ressource** de l'application (une action, une page, une image). Pour accéder aux ressources d'une application web de base **nous sommes habitués à saisir une page PHP dans l'URL** (avec de paramètres, éventuellement) **qui corresponde à un fichier qui existe dans le serveur.**

Exemples :

`http://<serveur>/<dossier>/<une page>.php`

`http://localhost/appAgenda/pages/afficherContacts.php`

`http://localhost/videoClub/films/rechercherFilm.php?nom=Alien&genre=Scifi&tutu=blablabla`

Dans de vraies applications, et pas seulement pour soigner l'esthétique mais aussi pour faciliter la saisie et pour la sécurité, nous avons besoin de créer des URLs plus simples. En général nous voulons taper une URL sur le navigateur et charger un fichier qui ne corresponde pas forcément à ce qu'on tape sur l'URL.

Exemple : pour charger la page

<http://netflox.localhost/recherche.php?prix=5000&category=four>

Nous voulons juste taper :

<http://netflox.localhost/recherche/5000/4>

Pour ce faire, on utilise une technique qui s'appelle **réécriture d'URL**.

Nous allons définir toutes les routes nous-mêmes. Une route indiquera la correspondance entre ce qu'on tape dans le navigateur et le code qui sera lancé (dans notre cas une action d'un controller) .

Pour définir une route nous pouvons utiliser quatre systèmes : **annotations, YAML, XML ou du code PHP.**

Ici on suivra les conseils des bonnes pratiques de Symfony et on utilisera des **annotations**. Pour utiliser des annotations on doit d'abord rajouter au projet le package qui permet leur manipulation :

```
composer require annotations
```

Nous allons créer maintenant un **controller** à la main contenant une route créée avec une annotation :

```
// src/Controller/ExemplesRoutingController.php

<?php

namespace App\Controller;

// importation de quelques librairies dont on a besoin dans le controller
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;

// classe contenant le code du controller (les actions)
class ExemplesRoutingController extends AbstractController{
    /**
     * @Route("/exemples/accueil");
     */
    public function afficherMessageAccueil ()
    {
        return new Response(
            '<html><body>Je suis une action du controller
ExemplesRoutingController.</body></html>'
        );
    }
}
```

C'est l'annotation **@Route** qui détermine la correspondance entre ce qu'on tape dans l'url :

`http://projet1symfony.localhost/exemples/accueil`

et l'action à lancer (la méthode qui se trouve juste après l'annotation)

`afficherMessageAccueil`

Une route indiquera alors une **action à lancer (méthode d'une classe) qui se trouve dans un controller (la classe, dans un fichier .php)**

Vous pouvez afficher toutes les routes de votre projet en tapant dans la console cette ligne :

```
php bin/console debug:route
```

L'outil **bin/console** vous permette de générer des bases de données à partir des entités, d'obtenir information du debugging et de plein d'autres opérations

Note : observez que le contenu de la route ne doit pas forcément contenir le nom du controller. Ici le controller s'appelle ExemplesRoutesController mais dans la route on indique uniquement "exemples"

Concernant le nom de l'action, dans les exemples successifs la route correspondra au nom de l'action et le camel case sera transformé selon le critère ci-dessous :

```
public function monAction1() → "/exemples/mon/action1"
```

Dans le controller ci-dessus on a juste envoyé du code HTML directement au client en utilisant l'objet **Response** (en détail plus tard) avec un string contenant du HTML, mais... et si on veut créer toute une page HTML? On ne va pas mettre tout le code dans l'action du controller ! on utilisera alors les **vues**.

Chaque action dans un controller a souvent une vue associé (le "rendu" de la page) qui contiendra le code HTML, JavaScript, CSS ou même du PHP). On étudiera les vues plus tard.

La structure complète du controller, ainsi que l'objet **Response**, seront expliqués plus tard, il faut juste savoir que le premier nous sert à envoyer une réponse directement au navigateur **sans passer par une vue**.

Exercices

1. *Créez une nouvelle action `monAction1` qui affiche le message "Ce controller est en charge du répertoire de l'application et je suis juste une action à l'intérieur"*
2. *Créez une nouvelle action `monAction2` qui affiche la date actuelle. Le path pour la route peut (ou pas) correspondre au nom de l'action, c'est à vous de choisir selon les circonstances et votre critère*

Les routes contenant de paramètres

Objectif : créer une action qui puisse accéder aux paramètres envoyés dans l'URL de la même façon qu'on fait en utilisant `$_GET` dans un code PHP normal

Nous avons déjà mentionné que, grâce à la réécriture des URLs nous n'utiliserons plus le format habituel de passage de paramètres dans l'URL :

<http://netflox.localhost/recherche.php?prix=5000&category=four>

Cette route aura ce format à partir de maintenant :

<http://netflox.localhost/recherche/5000/four>

On va configurer de routes pour pouvoir accéder facilement à ces paramètres.

En général le format de routes sera :

<http://<serveur>/<unController>/<uneAction>/<valeur param1>/<valeur param2>>

Voici un autre exemple :

<http://amazon.localhost/produits/add/5000/galaxy-s25>

Cette ligne appelle l'action **add** du controller **produits** et lui envoie les valeurs des deux variables. Voici un exemple pratique.

Exemple : Créer une action qui reçoit deux paramètres pour pouvoir l'appeler dans la route.

Le but est de créer une action "afficherContact" qui reçoit un prénom et une profession

<http://projet1symfony.localhost/exemples/afficherContact/Marie/formatrice>

et affiche un message de "Je suis <nom>,<profession>". Pour ce faire, l'action doit obtenir les valeurs reçues dans l'URL en utilisant un nouvel objet **Request** (au lieu de l'array `$_GET` ou `$_POST` qu'on utilise si on n'est pas dans le contexte d'un framework)

Procédure :

1. Créez une route qui permette à l'action de recevoir deux paramètres

```
/**
 * @Route("/exemples/afficher/contact/{prenom}/{profession}")
 */
```

2. Créez l'action afficherAction dans le controller **Exemples**

```
// rajoutez cette ligne pour importer l'objet Request
use Symfony\Component\HttpFoundation\Request;

// L'objet Request rajouté dans la signature de l'action contiendra les données
// de la requête faite au serveur. En ce qui nous concerne maintenant, il
// contiendra les valeurs des paramètres de l'URL. L'objet Request sera étudié
// plus tard.

/**
 * @Route("/exemples/afficher/contact/{prenom}/{profession}")
 */
public function afficherContact(Request $objetRequest){
    echo "Je suis dans le controller, action 'afficher';";
    // on obtient les valeurs des paramètres de l'url,
    // on fait appel à la méthode get de l'objet Request
    $lePrenom = $objetRequest->get ("prenom");
    $laProfession = $objetRequest->get ("profession");
    return new Response ("<br>Le prénom dans l'URL est: ".$lePrenom."<br>La
profession dans l'URL est: ".$laProfession);
}
```

3. Créez l'a vue associée à l'action

Dans ce cas nous n'avons pas une vue, on renvoie la réponse HTML directement au client en utilisant l'objet Response

4. Lancez l'action en tapant l'URL

<http://projet1symfony.localhost/exemples/afficher/Marie/formatrice>

Exercices : création d'actions contenant de paramètres

1. Créez une nouvelle action afficheTVAC qui reçoit un prix et affiche le prix TVAC
2. Créez une autre action qui reçoit trois valeurs et affiche la moyenne

Les contraintes dans les routes

Nous pouvons limiter les caractères qui peuvent apparaître dans les paramètres d'une route (pour limiter la saisie dans la barre d'adresse du navigateur, par exemple).

6.3.

Exemple : dans l'action suivant nous voulons forcer le paramètre **âge** à contenir uniquement un ou plusieurs caractères uniquement numériques et pas de lettres

```
/**
 * @Route("/exemples/bienvenue/age/{age}", requirements={"age"="\d+"})
 */
public function bienvenueAge (Request $objRequest){
    return new Response ("Bienvenue au site, vous avez ".$objRequest->get ("age")."
ans");
}
```

On a utilisé la balise **requirements** et on rajoute une expression régulière (ici **\d+**). Si vous voulez commencer à créer des expressions régulières pour vos propres routes, vous pouvez utiliser ce résumé :

<http://jkorpela.fi/perl/regexp.html>

Nous pouvons aussi choisir un autre format ou on n'utilisera pas la balise **requirements**. On rajouter l'expression régulière dans de balises **<>** :

```
// Requirements sans balise "requirements"

/**
 * @Route("/exemples/bienvenue/age/no/balise/{age<\d+>}")
 */
public function bienvenueAgeNoBalise (Request $objRequest){
    return new Response ("Bienvenue au site, vous avez ".$objRequest->get ("age")
." ans");
}
```

Si vous ne respectez le format, Symfony affichera une erreur de page introuvable. Pour le moment nous sommes en mode de **développement (dev)** et Symfony nous montrera l'erreur dans le **Symfony Profiler** (outil de Symfony pour faciliter le debugging, expliqué plus tard). Quand on changera au mode **production (prod)** on verra juste un simple message d'erreur 404 – page introuvable.

Exercices : création de contraintes dans les paramètres des routes

En utilisant la documentation sur les expressions régulières :

1. Créez une action "afficheVille" qui reçoit le nom d'une ville. Le nom doit contenir uniquement de lettres ou de chiffres mais pas d'autres caractères (voir \$, %, ^ etc...)
2. Modifiez l'action précédant pour que le paramètre "ville" puisse avoir une taille maximale de 15 caractères
3. Créez une action de votre choix contenant de restrictions pour le/les paramètres

Les valeurs par défaut pour les paramètres du routing

6.4.

Nous pouvons facilement établir une valeur par défaut pour nos paramètres en rajoutant ? et puis la valeur dans l'annotation.

Exemple : donner une valeur par défaut au paramètre TVA de l'action qui calcule la TVA

```
/**
 * @Route ("/exemples/affiche/prix/default/tvac/{prix}/{tauxTVA?21}")
 */
public function affichePrixDefaultTvac(Request $objetRequest)
{
    $prix = $objetRequest->get("prix");
    $tauxTVA = $objetRequest->get("tauxTVA");
    return new Response("<br>Le produit coûte ".$prix. " euros, "
        .($prix * (1 + $tauxTVA / 100)). " TVAC");
}
```

Exercice : utilisation de valeurs par défaut

Créez une action "afficheMessage" qui affiche un message un certain nombre de fois à l'utilisateur. Le message est reçu dans l'url, ainsi que le nombre de fois. Si le nombre de fois n'est pas indiqué, l'action l'affichera 10 fois. Le paramètre qui contient le nombre de fois doit être numérique

Pour avoir plus d'information sur la gestion de routes, allez sur la documentation de Symfony :

<https://symfony.com/doc/current/routing.html>

Paramètres optionnels

Vous pouvez créer une route contenant des paramètres optionnels. Il **suffit de rajouter '?' après le nom du paramètre**.

Exemple :

6.5.

```
/**
 * @Route ("/exemples/affiche/prix/opt/tvac/{prix}/{tauxTVA?}")
 */
public function afficheOptPrixTaux (Request $objetRequest){
    $prix = $objetRequest->get ("prix");
    $tauxTVA = $objetRequest->get ("tauxTVA");
    if (!isset ($tauxTVA)){
        $tauxTVA = 21;
    }
    return new Response ("<br>Le produit coûte ".$prix. " euros, "
        .($prix * (1 + $tauxTVA / 100)). " TVAC");
}
```

Les valeurs alternatives pour les paramètres

6.6.

Vous pouvez obliger à un paramètre à avoir une valeur parmi un ensemble de valeurs de votre choix en utilisant l'opérateur OR dans la route.

Exemple : le paramètre "disponibilité" peut prendre trois valeurs (et en plus il est optionnel !)

```
/**
 * @Route ("/exemples/affiche/disponibilite/{disponibilite<oui|non|en attente>?}")
 */
public function afficheDisponibilite(Request $objRequest)
{
    return new Response("Le produit est " . $objRequest->get("disponibilite"));
}
```

7. Controllers et actions

Le controller contient une méthode pour chaque action à effectuer dans l'application.

Une action peut faire, en gros, deux tâches différentes :

- a) **Générer un contenu et l'envoyer au client en utilisant une vue** (ex. : afficher la météo de Bruxelles en se connectant à un serveur de météo et envoyer les données obtenues à une vue qui les affichera)
- b) **Rediriger vers une autre action**

Procédure de création manuelle d'un controller

1. **Créez la classe vide du controller dans le Namespace App\Controller**

Exemple : ContactsController. La classe doit se trouver dans le NameSpace qui lui correspond

```
namespace App\Controller;
```

```
class ContactsController extends AbstractController{  
    // ici les actions  
}
```

2. **Importez la classe AbstractController car votre controller héritera cette classe (extends).** Si vous allez utiliser les classes **Response** et **Request** importez-les aussi. Importez la classe **Annotations** pour pouvoir créer les routes

```
namespace App\Controller;
```

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

```
class ContactsController extends AbstractController{  
    // ici les actions  
}
```


3. Créez l'action de votre choix dans le controller (dans la classe)

```
public function afficherTous (){
    $noms = ['Lucy', 'Juan', 'Salima', 'Mar', 'Lupita', 'Marie'];
    $strNoms = implode ("", $noms);
    return new Response ($strNoms);
}
```

4. Rajoutez la route vers l'action que vous venez de créer dans une annotation

```
/**
 * @Route ("/contacts/afficher/tous")
 */
```

5. Lancez l'action

<http://localhost:8000/contacts/afficherTous>

Le code final du controller sera :

```
<?php

namespace App\Controller;

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

class ContactsController extends AbstractController{
    /**
     * @Route ("/contacts/afficher/tous")
     */
    public function afficherTous (){
        $noms = ['Lucy', 'Juan', 'Salima', 'Mar', 'Lupita', 'Marie'];
        $strNoms = implode ("", $noms);
        return new Response ($strNoms);
    }
}
```

Exercice : Créez deux controllers Exercice1Controller et Exercice2Controller. Créez deux actions pour chaque controller. Créez aussi des actions qui contiennent de paramètres.

L'objet Response

Dans les exemples précédents, les actions **généraient un contenu HTML manuellement et l'envoyaient au navigateur en utilisant un objet de la classe Response**

Un objet **Response** contient toutes les propriétés et les méthodes pour **définir la réponse au client** (en plus du contenu HTML), mais ici on va se concentrer sur les plus basiques et indispensables :

- Le **contenu** (contenu envoyé au client : un **code HTML**, un **contenu JSON**...) et
- Les **headers** (entêtes http pour indiquer, entre autre, le type de réponse qu'on envoie au client – **HTML, JSON**...). Voici un exemple dans une action :

Exemple : création d'une action qui utilise l'objet Response (ExemplesReponsesController.php)

N'oubliez pas de rajouter la route !!

```
/**
 * @Route ("/contacts/message/response")
 */
public function messageResponse(){
    $contenu = "<h1>Je vais être le contenu d'un objet Response</h1>";
    $reponse = new Response ();
    $reponse->setExpires(new \DateTime('2025/3/8'));
    $reponse->headers->set ('Content-Type','text/html');
    $reponse->setContent($contenu);
    return $reponse;
}
```

L'objet Request

Quand on fait appel à une action du controller on est en train de **faire une requête au controller** (rien à voir avec les requêtes de BD !). Les informations concernant cette requête (ex : les valeurs des paramètres dans la URL) sont accessibles via l'objet **Request**.

L'objet **Request** nous permet d'accéder aux variables super-globales telles que **\$_POST**, **\$_GET**, **\$_SERVER**, **\$_FILES**. Regardez cette documentation pour savoir comment accéder à chaque variable :

https://symfony.com/doc/current/components/http_foundation.html

Pour qu'une action puisse accéder à l'objet Request **on doit juste rajouter un paramètre du type Request dans la signature de l'action (injecter l'objet Request)**.

Exemple : Utilisation de l'objet Request

Créez l'action suivante dans le controller ContactsController (rajoutez la route !) et lancez-la dans le navigateur

```
// L'objet Request rajouté dans la signature de l'action contiendra les données
// de la requête faite au serveur. En ce qui nous concerne maintenant, il
// contiendra les valeurs des paramètres de l'URL.

/**
 * @Route ("/contacts/message/request/{prenom}/{profession}")
 */
public function messageRequest (Request $objetRequest){
    echo "Je suis dans le controller, action 'afficher'";
    // on obtient les valeurs des paramètres de l'url,
    // on fait appel à la méthode get de l'objet Request
    $lePrenom = $objetRequest->get ("prenom");
    $laProfession = $objetRequest->get ("profession");
    return new Response ("<br>Le prénom dans l'URL est: ".$lePrenom. "<br>La
profession dans l'URL est: ".$laProfession);
}
```

Par la suite on verra comment créer de vues qui reçoivent de données depuis le controller.

Types de réponses d'un controller: render, redirect, redirectToRoute et forward

Une action dans un controller peut générer du HTML, JSON, XML, le téléchargement d'un fichier, une redirection vers une autre action, une erreur 404 ou plein d'autres résultats : tout dépend de nos besoins. Parmi les actions principales on a :

render permet d'afficher une vue

redirect permet de rediriger le navigateur vers une autre adresse, sans ou avec de paramètres

redirectToRoute permet de lancer une action mais en utilisant le nom (**name**) d'une route. On va voir des exemples dans ce chapitre

forward permet de déléguer l'action (pas d'erreur http)

Note : Pour comprendre le fonctionnement de ces réponses, on va créer des actions dans un nouveau controller ExemplesReponsesController

7.4.1. Redirect

La méthode `redirect` nous permet d'appeler une autre action dans un autre controller ou rediriger vers une autre adresse web

Exemple : redirection vers une autre adresse avec de paramètres

```
// cette action reçoit un titre de film et réalise une redirection vers imdb
// pour chercher le film

/**
 * @Route ("/exemples/reponses/exemple/redirect/{titreFilm}")
 */
public function exempleRedirect(Request $req)
{
    $titreFilm = $req->get("titreFilm");
    $url = "http://www.imdb.com/find?ref=nv_sr_fn&q=".$titreFilm;

    // maintenant on appelle une autre action
    return $this->redirect($url);
}
```

7.4.2. RedirectToRoute

Pareil que "Redirect" mais au lieu de spécifier la route complète on utilisera sa propriété **name**. Nous pouvons envoyer de paramètres à la nouvelle route.

Exemple : Redirection vers une nouvelle sans paramètres

```
// Cette action est juste un exemple qui montre comment une action peut
// rediriger vers une autre en utilisant la propriété "name"

/**
 * @Route ("/exemples/reponses/redirection/avec/name")
 */
public function redirectionAvecName(Request $req)
{
    // faire quoi qui ce soit ici....

    // et rediriger après vers une autre route en lui envoyant de paramètres
    return $this->redirectToRoute("spaghettiCarbonara");
}

/**
 * @Route ("/exemples/reponses/action/avec/name", name="spaghettiCarbonara");
 */
public function actionAvecName(Request $req)
{
    return new Response("Je suis une action qui a été appelée par une "
        . "autre, je porte un nom");
}
```

Exemple : Redirection vers une nouvelle route avec de paramètres (l'action reçoit de paramètres dans l'URL)

```
// Cette action est juste un exemple qui montre comment une action peut
// rediriger vers une autre en utilisant sa propriété "name".
// Cette action envoie paramètres
// sous la forme d'un array à la route cible
// Cette action est juste un exemple qui montre comment une action peut
// rediriger vers une autre en utilisant sa propriété "name".
// Cette action envoie paramètres
// sous la forme d'un array à la route cible

/**
 * @Route ("/exemples/reponses/redirection/avec/name/params")
 */
public function redirectionAvecNameParams(Request $req)
{
    // faire quoi qui ce soit ici....

    // et rediriger après vers une autre route
    return $this->redirectToRoute("spaghettiBolognese", ['type'=>'bio',
                                                         'prix'=>'10']);
}

/**
 * @Route ("/exemples/reponses/action/avec/name/params/{type}/{prix}",
name="spaghettiBolognese");
 */

public function actionAvecNameParams(Request $req)
{
    $type = $req->get("type");
    $prix = $req->get("prix");
    return new Response("Je suis une action qui a été appelée par une "
        . "autre, je porte un nom et je reçoit des valeurs: ".$type." ".$prix);
}
```

7.4.3. Forward

Forward nous permet d'appeler une autre action d'un autre controller

Exemple : Appel à une autre action d'un autre controller

Ici on a créé l'action "forwardExemple" (n'oubliez pas la route !) dans le controller

```
// Exemple de forward
// utilisation de forward pour appeler une action du controller Contacts
/**
 * @Route ("/exemples/reponses/forward/exemple")
 */
public function forwardExemple()
{
    return $this->forward(
        'App\Controller\ContactsController:afficherTous',
        ['prix' => 400, 'tauxTva' => 10]
    );
}
```

7.4.4. Render

La méthode **render** prend le nom de la vue (.twig) et optionnellement une variable contenant de données. L'idée est de passer de données du controller à la vue.

Par exemple : le controller accède à une BD, obtient un array ou des objets contenant les données d'un tableau et l'envoie à la vue pour qu'elle puisse les afficher.

On va étudier cette méthode plus tard dans la section [Les Vues. Le moteur de templates Twig.](#)

Création d'un controller avec l'assistant

Vous pouvez utiliser la commande

```
php bin/console make:controller <nom du nouveau controller>
```

pour générer automatiquement le squelette d'un controller.

Symfony génère le fichier du controller, une action route ainsi qu'une vue associée. Testez-le par vous-mêmes.

8. Gestion basique d'erreurs

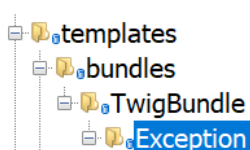
Pour gérer proprement les erreurs dans une application Web nous avons plusieurs possibilités :

- **Créer de vues personnalisées pour les erreurs de base (500, 404 etc...) en utilisant les conventions de Symfony**
- **Modifier la réponse HTTP** du serveur (ex: envoyer une réponse de 404 – Not Found quand on le souhaite, même si la page existe)
- **Lancer une exception** du système ou une personnalisée

8.1. Créer une vue pour chaque erreur à gérer en utilisant les conventions de Symfony

Cette méthode s'applique dans l'environnement de **prod**, car dans l'environnement de **dev** le Symfony Profiler est activé. Changez alors **dev** en **prod** dans le fichier **.env**. Une fois le changement est fait :

1. Créez cette **structure de dossiers** dans le dossier **templates**



2. Créez **une vue pour chaque erreur à gérer** (.html.twig) suivant la convention **errorXXX.html.twig** dans le dossier **Exception**

Exemple : error500.html.twig

3. Créez le **contenu personnalisé de la vue** pour chaque erreur que vous voulez traiter

Exemple : gestion de l'erreur 500

Voici un exemple de gestion de l'erreur 500. L'action **exempleActionProvoqueErreur** du controller **ExemplesErreursController** provoque une erreur 500 :

```
/**
 * @Route("/exemple/action/provoque/erreur")
 */
public function exempleActionProvoqueErreur()
{
    // Décommentez la ligne suivante:
    $b = aa;
}
```

Créez alors un fichier **erreur500.html.twig** dans le dossier **Exception** contenant un texte (par exemple : "Erreur interne !"). Relancez l'action pour que le nouveau message d'erreur s'affiche. Si vous obtenez l'erreur original au lieu de la vue que vous avez créé, nettoyez la cache depuis la console et relancez l'action. Pour nettoyer la cache

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

Exercice : traitez l'erreur 404 en utilisant ce système

Modifier la réponse http du serveur

Voici un exemple de comment envoyer un code d'erreur au navigateur en modifiant la réponse HTTP

```
class ExemplesErreursController extends AbstractController
{
    /**
     * @Route("/erreurs/erreur/pas/trouve", name="exemples_erreurs")
     */
    // réponse HTTP modifiée : dans cet exemple, on renvoie au
    // navigateur une réponse "404: NOT FOUND"
    // si la variable de session "login" n'existe pas. Décommentez la ligne du
    "set"
    // pour établir sa valeur une première fois, puis
    // commentez la ligne et relancez le code
    public function erreurPasTrouveAction(Request $req){
        $session = $req->getSession();
        $session->set ("login", "Marie");
        $reponse = new Response();
        if ($session->get("login") == null){
            $reponse->setStatusCode(Response::HTTP_NOT_FOUND);
            $reponse->setContent("Page non trouvée!");
            // autre exemple:
            // $reponse->setStatusCode(Response::HTTP_BAD_GATEWAY);
        }
        else {
            $reponse->setContent("Bienvenu " . $session->get("login"));
        }

        return ($reponse);
    }
}
```

Lancer une exception

Au lieu de rediriger vers une autre page du site on peut carrément lancer une exception. C'est à nous de voir qu'est-ce que nous convient selon les besoins du projet.

8.3.

```
/**
 * @Route("/erreurs/erreur/pas/trouve/avec/exception")
 */
public function errorPasTrouveAvecExceptionAction(Request $req){
    $session = $req->getSession();
    //$session->set ("login","on lance cette ligne une seule fois");
    $reponse = new Response();
    if ($session->get("login")!=null){
        throw $this->createNotFoundException ("Non trouvée");
    }
    else {
        $reponse->setContent("Bienvenu " . $session->get("login"));
    }
    return ($reponse);
}
```

9. Les Vues. Le moteur de templates TWIG

En Symfony nous pourrions générer les vues en utilisant du HTML et PHP purs, mais l'utilisation de TWIG comme **moteur de templates** nous facilitera vraiment la tâche. **Un moteur de templates sert à générer les vues à partir d'un fichier de base (template).** On part du principe que tout ce qui concerne la présentation (les vues) se trouvera dans les templates (jamais dans le controller !).

Le moteur TWIG est un **composant** de Symfony mais on pourrait l'utiliser dans n'importe quel projet ailleurs.

Les principaux avantages de TWIG sont :

1. Il masque le langage de programmation des vues : il n'y a pas du PHP dans les templates !
2. Il fournit une bonne interface avec les contrôleurs
3. Il nous permet de créer des hiérarchies de templates (en détail plus tard)
4. On peut le lier avec modèle en utilisant **Doctrine** (en détail plus tard)
5. On peut utiliser un squelette twig avec d'autres langages de programmation (ex : python)

Quand on lance la méthode **render** depuis un controller, Symfony parcourra le fichier **.twig** correspondant et générera un objet Response à partir de ce fichier. Cet objet génère le HTML qui sera renvoyé au serveur.

Attention : les fichiers **.twig** utilisent la notation snake case. Ex: Le template pour une action `mangeonsDuChocolat` sera un fichier portant le nom `mangeons_du_chocolat.html.twig`

Notes de base sur Twig :

<https://twig.symfony.com/doc/3.x/>

La documentation complete sur les vues en Symfony se trouve ici :

<https://symfony.com/doc/current/templates.html>

Création d'un template en utilisant Twig

On va créer une vue en utilisant Twig. La vue reçoit un tableau qui a été créé dans le controller et l'affichera. Normalement le controller accèdera à une base de données, mais pour simplifier notre exemple on crée juste un tableau. Faisons un exemple en partant de zéro :

9.1.

1. **Créez un controller** ExemplesTwig en utilisant

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

Notes : L'assistant créera le fichier du controller dans src/Controller, ainsi qu'un dossier exemples_twig dans le dossier **Templates**. Ce dossier contiendra toutes les vues liées aux actions de ce controller. Par défaut il y aura déjà une action **index** et une vue **index.html.twig**

Vous pouvez faire appel à cette action pour afficher la vue en tapant la route dans le navigateur :

```
http://localhost:8000/exemples/twig
```

2. Créez une **action** exemple1 dans ce controller. Dans l'action vous devez indiquer à Symfony de renvoyer au navigateur le rendu de la vue qu'on va créer :

```
/**
 * @Route ("exemplesTwig/exemple1", name="exemple1Twig")
 */
public function exemple1 (){
    return $this->render ('exemples_twig/exemple_1.html.twig');
}
```

3. Créez la vue : un **template twig** contenant portant le nom exemple_1.html.twig dans Templates

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

```
{% block body %}
```

```
Bonjour! je suis un template!
```

```
{% endblock %}
```

Conservez cette structure de blocs dans vos twigs et remplissez à chaque fois le contenu du block body. On étudiera plus tard comment utiliser les blocs pour enrichir la structure d'un template twig. Lancez la page juste pour tester si tout est en ordre, puis continuez la lecture.

Exercice : Créez une deuxième action affichePays et un template qui affiche "vive la Belgique"

Les variables

Twig peut utiliser des **variables**, ce qui nous permet d'accéder d'une façon très simple aux données générés dans le controller. Pour accéder une variable nous devons utiliser cette notation :

9.2. `{{ nom }}`

Pour accéder aux propriétés d'un objet on utilisera cette notation :

`{{ client.nom }}`

Ces variables proviennent du controller. Voyons un exemple. Créez cette action dans le controller :

```
public function afficheVille()
{
    // nous créons une structure de données
    $vars = ['nom' => 'Bruxelles',
            'population' => 1500000,
            'pays' => 'Belgique'];

    // render reçoit l'array associatif et renvoie l'objet Response
    // on accédera à cette array depuis la vue
    return $this->render ('exemples_twig/affiche_ville.html.twig', $vars);
}
```

Nous venons de rajouter un paramètre à l'appel **render**. Cela nous permet d'envoyer de valeurs à la vue. Le format est **d'un array associatif dont les clés deviennent de variables** accessibles dans le twig

Modifions maintenant notre template twig. **Notez que pour accéder aux variables on a juste utilisé les clés de l'array**

```
Le nom de la ville est:
{{ nom }}
La population est:
{{ population }}
Cette ville se trouve dans ce pays:
{{ pays }}
```

Nous avons envoyé un array associatif. Si on avait juste une valeur simple à envoyer c'est très simple :

```
return $this->render ('exemples_twig/blablabla.html.twig',
                    ['cinema'=> 'Aventura']);
```

Si on veut envoyer un array indexé il faudra lui "donner" une clé pour qu'il soit accessible dans le fichier twig. Par exemple :

```
$lesStagiaires = ['Lucie', 'Salima', 'Doris'];
```

Render enverra un array associatif contenant une clé dont la valeur correspond à cet array :

```
$vars = ['lesStagiaires' => $lesStagiaires ]
```

Dans le fichier twig on pourra accéder en utilisant la notation normale d'array :

```
{{ lesStagiaires[0] }}  
{{ lesStagiaires [1] }}  
{{ lesStagiaires [2] }}
```

Pour les objets **c'est exactement la même chose mais on utilisera la notation de "."** dans le fichier twig pour pouvoir accéder ses propriétés (pas la flèche de PHP!)

Exercices :

1. Créez une nouvelle action `afficheTvacTwig` qui reçoit une valeur d'un prix dans l'URL et une valeur de TVA et calcule le prix Tvac. Créez un twig qui affiche "Le prix TVAC est xxx"
2. Créez une nouvelle version de l'exercice précédent qui puisse afficher "Le prix xxxx euros avec le taux de TVA de xxxx % est x xxx"
3. Créez une nouvelle action qui affiche un array de trois villes belges. Les villes sont fixées et se trouvent dans la vue, ne sont pas envoyées par le controller à la vue
4. Créez une nouvelle version de l'exercice précédent où la liste de villes est envoyée depuis le controller à la vue dans un array
5. Créez une nouvelle version de l'exercice 4. Cette fois l'action reçoit la langue d'affichage dans l'URL (FR ou NL). La discrimination par langue se réalise dans le controller
6. Affichez la date actuelle dans la vue. Utilisez la classe `DateTime` et envoyez l'objet à la vue. Dans la vue, utilisez la méthode **format** de la classe `DateTime` pour afficher la date proprement

9.2.1. Les conditions

Le langage de TWIG inclut aussi les **conditions**. Voici un exemple:

```
{% if age > 18 and age < 60 %}  
    <p>Pas de réduction</p>  
{% endif %}
```

Voici la documentation pour avoir plus de détails :

<https://twig.symfony.com/doc/2.x/tags/if.html>

Exercices :

- 1) Créez une action qui reçoit un prix dans l'URL. Créez une action qui reçoit le prix et le multiplie par deux. Vérifiez dans la vue que le prix ne dépasse pas 100 euros en utilisant une condition
- 2) Trouvez dans la documentation des exemples d'utilisation des conditions dans twig. Faites deux vues qui utilisent des conditions (if et elseif)

9.2.2. Les boucles

Nous pouvons aussi créer **de boucles du type "foreach" sur des éléments itérables** (arrays, par exemple). Son utilisation est assez simple :

1. Créez l'élément à parcourir dans une action et renvoyez-le à la vue
2. Utilisez la syntaxe **for – in** dans la vue pour parcourir l'élément

Exemple : création d'un array dans le controller et affichage dans la vue en utilisant une boucle

1. Voici l'action. On a créé un array associatif

```
/**
 * @Route("/exemples/twig/boucles/exemple1")
 */
public function exemple1 (){
    $vars = [
        'ville' => [ 'nom' => 'Bruxelles',
                    'population' => 1500000,
                    'pays' => 'Belgique'
                ]
    ];
    return $this->render ('exemples_twig_boucles/exemple_1.html.twig', $vars);
}
```

2. Voici le contenu de la vue

Les données dans l'array sont :

```
<table>
{% for cle, valeur in ville %}
<tr><td>{{ cle }}</td><td>{{ valeur }}</td></tr>
{% endfor %}
</table>
```

Si nous avons besoin de **parcourir un objet on doit le transformer en array**. Cette conversion crée un array associatif dont les **clés** sont les **propriétés de l'objet** et les **valeurs** sont les **valeurs de ces propriétés**. Voici un exemple :

Exemple : parcourir un objet avec une boucle dans la vue

Nous créons un objet Film et on le renvoie à la vue. Nous devons convertir cet objet en array :

Pour lancer ce code la classe livre doit exister (juste pour cet exemple, créez la classe juste après la classe du controller :

```
class Film
{
    public $titre;
    public $auteur;

    function __construct($titre, $auteur) {
        $this->titre = $titre;
        $this->auteur = $auteur;
    }
}
```

Exercices :

- 1) Créez une nouvelle classe Film et une action qui renvoie un objet de cette classe à la vue. Affichez son contenu dans une vue dans un tableau HTML
- 2) Créez une nouvelle action contenant un array d'objets (Films). Envoyez-le à la vue et affichez le contenu de chaque Film dans un tableau HTML

9.2.3. Les filtres

Un filtre est une action sur une variable telle que mettre la variable en majuscules, enlever les espaces, enlever les caractères spéciaux, renvoyer la taille d'un string ou un array...

La syntaxe des filtres est :

```
{{ variable | filtre }}
```

Cette expression renvoie le résultat d'appliquer le filtre à la variable, par exemple :

```
{{ titre | lower }}
```

Affichera le titre en minuscules. Vous pouvez une liste **de filtres** [ici](#)

Voici un exemple de vue qu'utilise de filtres.

Exemple 1 : action qui renvoie un array, vue qu'utilise un filtre pour mettre en majuscules la première lettre de chaque propriété de l'array

```
/**
 * @Route("/exemples/twig/filtres/exemple1", name="exemples_twig_filtres")
 */
public function exemple1()
{
    $ville = ['nom' => 'Bruxelles',
              'population' => 1500000,
              'pays' => 'Belgique'
    ];

    return $this->render('exemples_twig_filtres/exemple_1.html.twig', [
        'controller_name' => 'ExemplesTwigFiltresController',
        'ville' => $ville
    ]);
}
```

Les données dans l'array sont:

```
<table>
{% for cle, valeur in ville %}
<!-- capitalize renvoie la variable en majuscules -->
<tr><td>{{ cle | capitalize }}</td><td>{{ valeur }}</td></tr>
{% endfor %}
</table>
{% if ville | length > 1 %}
Le tableau n'est pas vide
{% endif %}
```

Exemple 2 : Action qui renvoie un array à la vue. Utilisation du filtre "join"

Le filtre "join" crée une chaîne de caractères contenant tous les éléments d'un array séparés par un "séparateur" choisi par nous-mêmes. Voici un exemple où on enchaîne les filtres "join" et "upper" :

```
Les données dans l'array sont:
{% if ville | length > 1 %}
Le tableau n'est pas vide, voici son contenu
{{ ville | join ( " - " ) | upper }}
{% endif %}
```

Héritage de templates en TWIG (I)

9.3. Pour les sites qui contiennent un ensemble de pages qui partagent le même contenu (ex: header, nav et footer) le moteur de TWIG possède un **système d'héritage** qui nous permet de créer une sorte de master page qui contiendra des blocs variables.

Voici un exemple d'utilisation :

Exemple : création d'un squelette de master page et inclusion du contenu dans sa section main

1. Créez un template twig contenant le squelette d'une master page :
master_page_1.html.twig

```
<html>
<head></head>
<body>
  <header>
    <h1>Voici le header de la page</h1>
  </header>
  <main>
    <!-- on indique que un bloc "contenuPrincipal" sera incrusté ici-->
    {% block contenuPrincipal %}{% endblock %}
  </main>
  <footer>
    <nav>
      <ul>
        <li>Option footer 1</li>
        <li>Option footer 2</li>
      </ul>
    </nav>
  </footer>
</body>
```

</html>

2. Créez deux templates pour les contenus (ex: *contenu_1_master_page_1.html.twig* et *contenu_2_master_page_1.html.twig*) qui héritent de *master_page_1.html.twig* en utilisant **extends**. Délimitez le contenu de votre vue par les directives **block**.

contenu_1_master_page_1.html.twig

```
{% extends "exemples twig heritage/master_page_1.html.twig" %}

{% block contenuPrincipal %}
Je voudrais manger de champignons dans le contenu 1 !
{% endblock %}
```

contenu_2_master_page_1.html.twig

```
{% extends "exemples twig heritage/master_page_1.html.twig " %}

{% block contenuPrincipal %}
Je voudrais manger de carottes dans le contenu 2 !
{% endblock %}
```

3. Créez les **actions** (et les **routes**) qui rendront cette vue et lancez chacune dans votre navigateur. Vous observerez que le contenu de cette vue sera incrusté dans la page.

```
/**
 * @Route("/exemples/twig/heritage/contenu1/master/page1")
 */
public function contenu1MasterPage1()
{
    return $this->render('exemples twig heritage/contenu_1_master_page_1.html.twig');
}
/**
 * @Route("/exemples/twig/heritage/contenu2/master/page1")
 */
public function contenu2MasterPage1()
{
    return $this->render('exemples twig heritage/contenu_2_master_page_1.html.twig');
}
```

Important : **extends** doit être la première balise d'un template

Exercice : codez un nouveau template vous-mêmes qui hérite de la master page que vous venez de créer. Cette nouvelle vue reçoit un array contenant les infos d'un livre de son action associée et l'affiche dans le contenu de la master page

Création de Templates en TWIG (II)

Nous pouvons simplifier encore la structure de la page si on utilise la directive **include**, qui nous permet d'inclure le contenu d'autres fichiers twigs où on le souhaite.

Voici un exemple :

9.4.

Exemple : simplification de la master page en utilisant include pour importer le header et le footer

1. Créez une nouvelle master page master_page_2.html.twig

```
<html>
<head>
<body>
    <header>
        {% include "header.html.twig" %}
    </header>
    <main>
        {% block contenuPrincipal %}{% endblock %}
    </main>
    <footer>
        {% include "footer.html.twig" %}
    </footer>
</body>
</html>
```

2. Créez les twigs header.html.twig et footer.html.twig dans le dossier de la master page

header.html.twig :

```
<h1>Je suis le header</h1>
```

footer.html.twig

```
<nav>
    <ul>
        <li>Option footer 1</li>
        <li>Option footer 2</li>
    </ul>
</nav>
```

3. Créez les actions et les routes

```
/**
 * @Route("/exemples/twig/heritage/contenu1/master/page1")
 */
public function contenu1MasterPage2()
{
    return $this->render('exemples_twig_heritage/contenu_1_master_page_2.html.twig');
}
/**
 * @Route("/exemples/twig/heritage/contenu2/master/page2")
 */
public function contenu2MasterPage2()
{
    return $this->render('exemples_twig_heritage/contenu_2_master_page_2.html.twig');
}
```

Notez que les includes ne doivent pas servir uniquement à la création d'une master page. Nous pouvons utiliser cet outil dans n'importe quelle fichier twig. Cela nous permet d'inclure des fragments de la page qui se trouvent dans d'autres fichiers et qui deviennent ainsi partageables par toutes les vues.

Exercice : créez une master page en utilisant twig. La master page contiendra une barre de navigation contenant trois liens. Chaque lien appelle une action d'un controller

Aidez-vous de cette documentation :

<https://symfony.com/doc/current/templates.html#linking-to-pages>
9.5.

Vider un bloc hérité d'un template

Dans certaines pages il se peut qu'on hérite d'un template qui contienne une section qui ne nous intéresse pas (ex : la barre de navigation du contenu du site dans une page de login). La solution à ce problème est simple : on doit déclarer le bloc dans notre page et le laisser vide.

Exemple : on hérite un menu d'un template base qu'on ne souhaite pas avoir dans notre template login

Il suffira d'inclure (dans la page qui hérite le template)

```
{% block menu %}{% endblock %}
```

Et le block sera vide.

Incruster une action du controller dans une vue

Considérez le cas générique d'un site où plusieurs vues qui, en plus de son propre contenu, contiennent une section (un div, par exemple) qui charge un contenu d'une BD (ex : latest news, dernières offres, information dernière minute etc...).

Voici un exemple : Vue1 et Vue 2 ont une section dynamique commune (le code est dans *ExempleControllerDansVueController.php*, actions *afficheVue1* et *afficheVue2*)

Je suis la vue 1

Contenu de la vue 1 Bla bla bla Bla bla bla Bla bla bla Bla bla bla	<ul style="list-style-type: none">• Les chats ne sont pas si gentils que ça• Tombe, tombe, tombe la pluie• Vamos a la playa
---	---

Je suis la vue 2

Contenu de la vue 2 Coucou!!!!!!!!!!!!!!	<ul style="list-style-type: none">• Les chats ne sont pas si gentils que ça• Tombe, tombe, tombe la pluie• Vamos a la playa
---	---

Le code pour charger le contenu de cette section dans chacune de ces pages devra alors se répéter dans chaque action du controller, ce qui casse le principe du DRY. On pourrait se dire d'utiliser **include**, mais le problème est que le contenu doit être généré par une action, il ne suffit pas d'inclure un autre template !

On peut résoudre ce problème assez facilement : on peut **incruster (embed)** l'appel d'un controller dans chaque template et créer une seule action. Cette action (*genererContenuDynamique*) n'a pas de route car elle sera uniquement utilisée depuis les templates.

On aura alors :

- Une **action pour générer chaque template**
- Une **action sans route pour générer le contenu de la section commune**
- Un **appel à cette action dans chaque template** là où on veut réaliser le rendu ('embed' le controller)

L'appel au controller depuis la vue se fait de cette manière :

```
<div id='container2'>
{#on incruste le controller ici#}
{{ render (controller (
'App\\Controller\\ExempleControllerDansVueEmbedController::sectionNouvellesDynamiqu
e',{ 'nombreNouvelles':3 }
))
}}
</div>
```

Observez que si on utilise la syntaxe "::" on doit échapper les back slash. On peut aussi envoyer de paramètres à l'action.

Vous avez un exemple fonctionnel et commenté dans *ExempleControllerDansVueEmbedController.php* et les templates associés.

Exercice :

Créez un site contenant deux pages. Dans les deux pages on doit avoir une section commune qui affiche une blague aléatoire sur Chuck Norris. Utilisez cette api : <https://api.chucknorris.io/>

Pour faire appel à une API depuis Symfony, installez le client http :

```
composer require symfony/http-client
```

Si l'API n'a pas d'authentification, il suffit d'utiliser la syntaxe qui suit.

```
$client = HttpClient::create();
$response = $client->request ('GET', 'https://api.chucknorris.io/jokes/random');
```

Pour savoir plus sur l'appels aux Apis dans Symfony :

https://symfony.com/doc/current/components/http_client.html#making-requests

Regardez aussi la section "Processing réponses". Utilisez **dump** pour vérifier ce que vous obtenez.

Faire appel à une action depuis la vue

Vous pouvez créer des liens pour faire appel à une action d'un contrôleur dans votre template Twig en utilisant la méthode **path**, sans ou avec des paramètres. Cette méthode créera la URL (vous pouvez la visualiser dans le code HTML de la page). Elle utilise le **name** de la route pour générer le chemin. Vous avez un exemple pratique dans [Projet1Symfony5](#), contrôleur **ExemplesParamsTwigToController**, où un template fait appel à un autre en lui envoyant un paramètre.

Le fonctionnement est simple :

1. Sans paramètres :

```
<a href="{{ path ('action2_recoit_params') }}">lien vers action 2 sans params</a>
```

2. Avec des paramètres (tel que l'exemple dans le projet) :

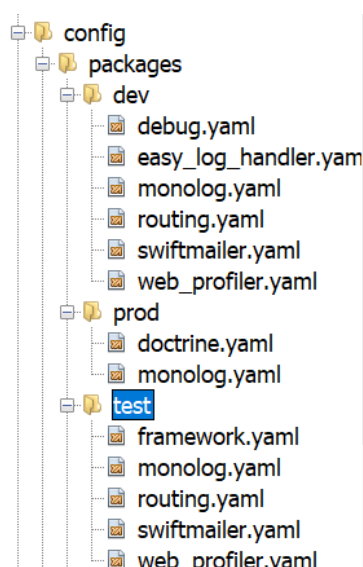
```
<a href="{{ path ('action2_recoit_params',{nom: 'Luca',ville: 'Rome'}) }}">lien vers action 2 avec params</a>
```

10. Les environnements de développement et production

Dans un projet Symfony nous pouvons choisir l'**environnement de travail**. Les choix possibles sont **prod**, **dev** et **test**. Tant Symfony que les "packages" de tiers contiennent des fichiers de configuration qui déterminent son comportement selon le mode de travail. Ces fichiers varient selon le mode de travail qu'on choisit.

Si nous choisissons le mode **dev** (par défaut) Symfony chargera le module **web_profiler**, qui facilite le debugging de l'application en affichant plein d'information sur notre application (entre autre les erreurs de toute sorte : typographie, connexion de bases de données, pages introuvables...). En mode **prod** le web profiler ne sera pas chargé car nous ne voulons pas donner au client plus d'informations que nécessaire (ex : le nom de la base de données).

Nous pouvons aussi avoir des modules (packages) de tiers ou faites par nous-mêmes dont la configuration change selon nous soyons dans le mode **dev**, **prod** ou **test** :



Observez que nous avons, par exemple, trois fichiers de configuration pour **monolog** (génération de messages de logging dans le serveur). Les logs seront générés d'une façon différente selon le mode où on se trouve. Il y a d'autres packages qui sont désactivés dans un mode et actives dans d'autres.

(La localisation de ces fichiers est définie dans `src/kernel.php`)

Vous pouvez changer d'environnement en modifiant le fichier `.env` qui se trouve dans la racine du projet (`APP_ENV` peut être `prod`, `dev` ou `test`)

```
###> symfony/framework-bundle ###  
APP_ENV=dev
```

Le **kernel** de Symfony - code qui est en charge de recevoir les requêtes de l'utilisateur et envoyer une réponse au navigateur - agira d'une façon ou l'autre selon la valeur de APP_ENV. **Le kernel chargera tous les services** qui correspondent au mode choisi, ainsi que leur **configuration** (qui varie entre dev, prod et test).

Dans la page index.php on peut observer la manière dont le kernel est créé.

Dans le fichier **config/bundles.php** vous pouvez voir quels sont les services disponibles pour chaque mode.

Les fichiers .env et .env.local

Par défaut Symfony lit le contenu du fichier **.env**. **Si vous utilisez un système de versions (GIT) ce fichier sera pris en compte**. Si on veut développer localement (avec les paramètres de BD locaux et toute le reste de la configuration) on a l'option de créer un fichier **.env.local**. Ce fichier peut être une copie modifiée du fichier **.env** ou, par exemple, on choisit la valeur **dev** pour **APP_DEV**.

Ce fichier

- Est prioritaire sur le fichier **.env** mais ...
- **est ignoré par GIT** (voir git.ignore dans le projet)

Ce mécanisme nous permet de travailler avec une configuration particulière en local qui ne sera pas copiée dans le serveur de production (car uniquement **.env** se trouvera dans le serveur).

Vous pouvez utiliser aussi :

```
composer install --no-dev --optimize-autoloader
```

qui effacera les packages qui ne sont pas nécessaires en production.

Le Web Profiler

Comme nous avons mentionné ci-dessus, le **web profiler est un outil de debugging** de Symfony. Si vous activez le mode **dev** et vous chargez une page, la barre du debugger du **web profiler** sera affichée **en bas de la fenêtre du navigateur** :

10.2.



Le profiler vous permet d'afficher beaucoup d'information concernant la requête qui a été faite au serveur. Pour que le profiler soit visible on doit envoyer une page complète HTML depuis le controller. Il ne suffit de faire "return Response". Le twig devra hériter du template de base ou avoir ses propres balises html et body.

Afficher le contenu des variables avec dump

10.3.

Vous pouvez afficher le contenu des variables dans les vues et les controllers grâce à la fonction **dump**.

Exemple :

dump (\$livre)

Cette fonction affiche le contenu complet de la variable d'une façon très complète. Utilisez-la au lieu de **var_dump** à partir de maintenant.

```
ContactsController.php on line 17:
array:6 [▼
  0 => "Alice"
  1 => "Viviana"
  2 => "Diana"
  3 => "Golnar"
  4 => "Salima"
  5 => "Marie Ange"
]
```

11. Les Services

Une application WEB utilise une **énorme quantité d'objets pour réaliser plein de fonctionnalités : connecter à une BD, envoyer un mail, connecter avec une API à un autre site, écrire/lire de fichiers dans le disque, etc...**

Certains parmi ces objets seront définis par nous (en créant de classes), certains se trouvent déjà dans la structure du framework et d'autres seront importés dans le projet.

Très **souvent on veut accéder à un de ces objets (qui réalise une certaine fonctionnalité) depuis de différents lieux dans notre projet**. On veut, par exemple, connecter à une BD depuis plein d'actions qui ne se trouvent même pas dans le même controller. Cela implique qu'on devrait avoir une sorte de structure de "includes" ou de "autoload" qui nous permettent de réaliser ces actions facilement.

Symfony (et plein d'autres frameworks) **fournit un système qui nous permet d'accéder facilement à ces objets qu'on utilise dans plusieurs emplacements de notre projet : le Service Container**.

En fait, un objet qui sera utilisé de manière générale depuis n'importe quelle localisation dans notre projet portera le nom de **Service**. Pour cette raison, On va parler du "service mailer", du "service logger", du "service DB" ou du "service maps".

11.1. Utilisations des services inclus dans Symfony

Symfony contient plein de services par défaut même si pour le moment nous ne les avons pas vraiment utilisés (au moins consciemment !). Allez dans la console (dans le dossier de votre projet actuel) pour afficher tous les services actifs ainsi que la description de leur fonctionnalité :

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

(note : la commande `|more` permet d'arrêter l'affichage à chaque page. Ça n'a rien à voir avec Symfony, c'est un vieil outil de la console. Tapez sur enter pour continuer à afficher le reste de services)

Nous pouvons utiliser ces services ainsi que rajouter nos propres services (nous devons faire le code, bien sûr !)

La documentation sur les services et leur utilisation se trouve ici :

https://symfony.com/doc/current/service_container.html

Pour passer à la pratique, utilisons déjà un des services fournis par Symfony. Commençons par utiliser un service qui nous permet de créer de fichiers de **logs**.

*Exemple : Utilisation du **service** Logger dans un controller*

1. **Créez un controller** ExemplesServicesController avec l'assistant
2. **Rajoutez une action utiliseLogger** (ignorez ou effacez l'action index et sa vue)
3. **Dans le prototype de l'action, rajoutez un paramètre de la classe LoggerInterface** (n'oubliez pas le **use**). Ceci indiquera au **Service Container** qu'il doit injecter un **objet Logger** dans notre **action** (rappelez-vous de l'injection de dépendances !!) pour qu'il soit utilisable à l'intérieur
4. **Utilisez normalement le service**. Il n'y a pas besoin de créer une instance, car on la reçoit en paramètre ! Ici on montre deux exemples de base: info et error

Voici le code final du controller :

```
namespace App\Controller;

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

class ExemplesServicesController extends AbstractController
{
    /**
     * @Route ("/exemples/services/utilise/logger")
     */
    public function utiliseLogger(LoggerInterface $monLogger){
        $monLogger->info ("On utilise le logger, c'est super!");
        $monLogger->error ("Hey! une erreur s'est produite!");
        return new Response ("J'ai fait mon boulot de logger ce qu'il faut!");
    }
}
```

Pour vérifier que le log a été effectué, ouvrez le fichier `/var/log/dev.log` de votre projet

On aurait pu rajouter d'autres services si on avait eu besoin dans le prototype de l'action. Il suffit de les séparer par virgules. Cette "magie" est réalisé grâce au système de **autowire** de Symfony, dont la configuration se trouve dans **config/services.yaml**

Si vous avez besoin de réaliser du logging dans votre application, vous avez plus d'info sur le sujet ici :

<https://symfony.com/doc/current/logging.html>

Exercice :

Symfony contient un service qui nous permet de gérer la session, **SessionInterface**. Appliquez la même procédure que dans l'exemple précédant pour l'obtenir. Dans votre action, créez une variable de session et affichez sa valeur. Le mécanisme de base de la session est simple : pour créer une variable de session on utilise la méthode set (clé, valeur) et pour la lire on utilise la méthode get (clé) de l'objet Session

Création de nos propres services

Nous avons mentionné **qu'on peut transformer nos propres objets en services**. Pour ce faire, on doit juste créer notre classe dans le dossier **/src/Services** (à créer s'il n'existe pas encore) et suivre la même procédure que dans les exemples précédents.

11.2.

Exemple : Nous voulons un service permettant d'obtenir toutes les permutations possibles des éléments d'un array de noms (permutations = combinaisons où l'ordre compte)

1. **Créez le dossier /src/Services** (s'il n'existe pas déjà) qui contiendra tous nos services

/src/Services

2. **Créez la classe qui réalise la fonctionnalité à l'intérieur de ce dossier (le service!)**

Voici un code possible, où la méthode *permutations* reçoit un array et renvoie un autre contenant le résultat :

```
// src/Services/Statistiques.php
namespace App\Services;

class Statistiques {

    // calcule toutes les permutations possibles de valeurs d'un array
    /**
     *
     * @param type $items
     * @param type $perms
     * @return type array (toutes les permutations - combinaisons
     * ou l'ordre compte)
     */
    function permutations($items, $perms = array( )) {
        if (empty($items)) {
            $return = array($perms);
        } else {
            $return = array();
            for ($i = count($items) - 1; $i >= 0; --$i) {
                $newitems = $items;
                $newperms = $perms;
                list($foo) = array_splice($newitems, $i, 1);
                array_unshift($newperms, $foo);
                $return = array_merge($return, $this->permutations($newitems,
                    $newperms));
            }
        }
        return $return;
    }
}
```

```
}
```

3. Créez une action dans un controller (ici *ExemplesPropreService*) pour utiliser votre service, tel qu'on a fait dans les sections précédentes. Symfony le reconnaitra directement !

```
// src/Controller/ExemplesPropreServiceController
use App\Services\Statistiques;

class ExemplesPropreServiceController extends AbstractController
{
    /**
     * @Route ("/exemples/propreservice/utilise/statistiques");
     */
    public function utiliseStatistiques (Statistiques $mesStats){

        $arrayNoms = ['Lucas','Jean','Norah'];
        $permutationsNoms = $mesStats->permutations($arrayNoms);
        return $this->render
        ('exemples_propre_service/utilise_statistiques.html.twig', ['permutationsNoms'=>
        $permutationsNoms]);
    }
}
```

Exercice : Créez à partir de zéro un service qui dit bonjour et une action pour montrer son fonctionnement

11.3.

Injecter les services dans le controller

Nous pouvons injecter de services directement dans le controller en utilisant son constructeur. Voici un exemple équivalent au code précédant.

Exemple : injection d'un service dans le constructeur

```
// src/Controller/ExemplesPropreServiceInjectionController.php
class ExemplesPropreServiceInjectionController extends AbstractController
{
    private $mesStats;

    // on injecte le service directement dans le constructeur du controller
    public function __construct (Statistiques $mesStats){
        $this->mesStats = $mesStats;
    }

    /**
```

```

        * @Route ("/exemples/propreservice/injection/utilise/statistiques");
        */
    public function utiliseStatistiques (){

        $arrayNoms = ['Lucas','Jean','Norah'];
        $permutationsNoms = $this->mesStats->permutations($arrayNoms);
        return $this->render
('exemples_propreservice_injection/utilise_statistiques.html.twig',
['permutationsNoms'=> $permutationsNoms]);
    }
}

```

Cette méthode devient indispensable quand **on veut utiliser un service dans un autre service, car la seule méthode où on pourra l'injecter sera dans le constructeur** ! On montre un exemple ci-dessous de ce cas de figure.

Injection de paramètres dans le service (I)

Nous pouvons rajouter de paramètres aux services en utilisant *services.yaml*. Dans ce fichier on configure la manière dont les services deviendront accessibles dans nos controllers.

11.4.

Exemple : un service Bonjour qui affiche "bonjour à tous" dans la langue paramétrée dans services.yaml

D'abord on doit créer le service. Si on veut que le service soit paramétré **on rajoute un paramètre dans le constructeur et on le stocke dans une propriété** (on peut avoir autant de paramètres qu'on veut) :

```
// App/Services/Bonjour.php
namespace App\Services;

class Bonjour {
    private $langue;
    public function __construct ($langue){
        $this->langue = $langue;
    }
    // service contenant un paramètre
    public function obtenirMessage (){
        // array fake... juste pour essayer le service
        $messages = ['fr' => 'Bonjour à tous!!',
                     'en' => 'Hello everybody!!'
                    ];
        // on obtient le paramètre du propre service
        $langue = $this->langue;
        return ($messages[$langue]);
    }
}
```

On doit définir le parametre dans *services.yaml* :

```
App\Services\Bonjour:
    arguments:
        $langue: 'fr' namespace App\Services;
```

Finalement on peut utiliser le service dans un controller. Ici on a injecté le service dans le controller en suivant la méthode de la section precedente :


```
// src/Controller/ExemplesServicesParamsController.php
class ExemplesServicesParamsController extends AbstractController
{
    private $bonjour;

    // on utilise la méthode d'injection du service dans le controller
    public function __construct (Bonjour $bonjour){
        $this->bonjour = $bonjour;
    }

    /**
     * @Route ("/exemples/propres/service/params");
     */
    public function utiliseBonjour(){
        return new Response ($this->bonjour->obtenirMessage());
    }
}
```

Si on veut qu'un paramètre soit utilisé par tous les services on a qu'à le rajouter à la section **parameters** qui se trouve tout en haut du fichier *services.yaml*

Utiliser un service dans un autre service

Si nous voulons utiliser un service dans un autre service on peut uniquement les injecter dans le constructeur (on ne peut pas injecter l'objet Logger dans une autre méthode, tel que "permutations"). **Si on veut utiliser un service dans un autre on est obligé de les injecter dans le constructeur. Ce n'est pas possible de les injecter dans une autre méthode.** C'est uniquement dans le controller où on peut injecter les services dans n'importe quelle méthode.

Exemple : utiliser Logger dans le service Statistiques

```
// src/Services/StatistiquesLog.php
namespace App\Services;

use Psr\Log\LoggerInterface;

class StatistiquesLog {

    private $logger;

    // injection dans le constructeur
    function __construct (LoggerInterface $logger){
        $this->logger = $logger;
    }

    function permutations($items, $perms = array( )) {
        if (empty($items)) {
            $res = array($perms);
        } else {
            $res = array();
            for ($i = count($items) - 1; $i >= 0; --$i) {
                $newitems = $items;
                $newperms = $perms;
                list($foo) = array_splice($newitems, $i, 1);
                array_unshift($newperms, $foo);
                $res = array_merge($res, $this->permutations($newitems,
                    $newperms));
            }
        }
        // on utilise le service de log
        $this->logger->info ("De permutations ont été calculées");
        return $res;
    }
}
```

```
// src/Controller/ExemplesServiceUtiliseService.php
class ExemplesServiceUtiliseService extends AbstractController
{
    private $mesStats;
    // Le service StatistiquesLog utilise Logger
    public function __construct (StatistiquesLog $mesStats){
        $this->mesStats = $mesStats;
    }

    /**
     * @Route ("/exemples/propres/service/utilise/service");
     */
    public function utiliseStatistiques (){

        $arrayNoms = ['Lucas','Jean','Norah'];
        // calculera les permutations et créera une ligne de log
        $permutationsNoms = $this->mesStats->permutations($arrayNoms);

        return $this->render
        ('exemples_service_utilise_service/utilise_statistiques.html.twig',
        ['permutationsNoms'=> $permutationsNoms]);
    }
}
```

Injection de paramètres dans le service (II)

Dans certains cas **nous utilisons un service dans un autre et le premier doit être paramétré.**

Exemple : rajouter l'envoi d'un mail dans notre service de Statistiques

Le service StatistiquesLogMail envoie un mail en plus de créer une ligne de log quand on fait appel à la fonction de permutations.

Installez d'abord le service de mail (on le configura plus tard) :

```
composer require symfony/swiftmailer-bundle
```

Dans ce cas, le service a besoin d'un ou plusieurs paramètres pour être configuré (ici on va considérer le paramètre **\$adresse** le destinataire du mail qui sera, par défaut, "yoyo@touloulou.com")

Voici notre service, qui inclut maintenant l'envoi d'un mail. On a dû injecter le Mailer dans le constructeur et on a décidé d'envoyer l'adresse mail en paramètre.

```
// src/Services/StatistiquesLogMail.php
namespace App\Services;

use Psr\Log\LoggerInterface;
use Swift_Mailer;

class StatistiquesLogMail {

    private $logger;
    private $mailer;
    private $adresse;

    function __construct (LoggerInterface $logger, Swift_Mailer $mailer, $adresse){
        $this->logger = $logger;
        $this->mailer = $mailer;
        $this->adresse = $adresse; // spécifiée dans services.yaml, porte le même
nom
    }

    function permutations($items, $perms = array( )) {
        .
        .
        .
        // on utilise le service de log
        $this->logger->info ("De permutations ont été calculées");
        // on envoie un mail
        $message = (new \Swift_Message)
            ->setTo ($this->adresse);
        // on doit envoyer ici le mail après avoir configuré le service
    }
}
```

```

        // https://symfony.com/doc/current/email.html#configuration
        // dump ($message);
        // die();
        return $res;
    }
}

```

mais cette solution ne suffit pas. Inclure le paramètre parmi les paramètres du constructeur provoque cette erreur :



C'est tout à fait logique, car quand on fait appel à notre service depuis le controller on n'indique pas ce paramètre extra. Le service principal est juste injecté dans le constructeur sans paramètres :

```

// /src/controller/ExemplesPropreServiceInjectionParamsController.php
class ExemplesPropreServiceInjectionParamsController extends AbstractController
{
    private $mesStats;
    // on injecte le service directement dans le constructeur du controller,
    // sans paramètres!
    public function __construct (StatistiquesLogMail $mesStats){
        $this->mesStats = $mesStats;
    }
    .
    .
    .
}

```

Comment fixer alors l'adresse du mail si on ne peut pas l'envoyer en paramètre au service lors sa création ?

Nous pouvons configurer de paramètres particulières de nos services dans le fichier **services.yaml** :

```

# add more service definitions when explicit configuration is needed
# please note that last definitions always *replace* previous ones
App\Services\StatistiquesLogMail:
    arguments:
        $adresse: "yoyo@gmail.com"

```

Le paramètre doit porter le même nom que celui qu'on a rajouté dans le constructeur, autrement on obtient une erreur.

Si on voulait changer l'adresse on peut toujours créer de méthodes pour ce faire dans notre service, rien nous empêche de créer une méthode pour ce faire dans notre service. On vient d'arranger le problème d'avoir la valeur lors la création du service. En plus, le code reste plus propre car les paramètres de tous nos services seront centralisés dans **services.yaml**

12. Cookies

Créer de cookies dans Symfony est très simple avec la classe `Cookie`. On doit :

- 1) Créer l'objet **Cookie** dans une action (spécifier son nom, sa valeur et le moment d'expiration)
- 2) Créer un objet **Response**
- 3) Attacher le cookie à l'objet Réponse avec **setCookie**
- 4) Envoyer la réponse au client avec **send**

Voici un exemple complet : la première action crée le cookie. La deuxième action lit la valeur du cookie et l'efface. Juste pour pouvoir afficher sa valeur on l'envoie à une vue. Vous pouvez aussi afficher la valeur de la cookie en utilisant la console du navigateur (onglet Stockage dans Firefox).

```
// src/Controller/CookiesController.php
class CookiesController extends AbstractController
{
    /**
     * @Route("/cookies/creer/cookie")
     */
    public function creerCookie()
    {
        $cookie = new Cookie(
            'unCookie',
            'chocolat',
            time() + (50 * 24 * 60 * 60)); // le temps est exprimé en
            // secondes, ici 50 jours à partir de cet instant

        $reponse = new Response();
        $reponse->headers->setCookie($cookie); // rattache la cookie
            //aux en-têtes http

        $reponse->send();
        // return $reponse;
        return $this->render('cookies/creer_cookie.html.twig');
    }

    /**
     * @Route("/cookies/afficher/effacer/cookie")
     */
    public function afficherEffacerCookie(Request $req){
        $valeur = $req->cookies->get('unCookie'); // on stocke avant d'effacer
            // pour l'envoyer à la vue

        $res = new Response();
        $res->headers->clearCookie('unCookie');
        $res->send();

        return $this->render('cookies/afficher_effacer_cookie.html.twig',
            ['valeurCookie' => $valeur]);
    }
}
```

13. Session

Nous pouvons créer des variables de session juste en injectant un objet `SessionInterface` dans une action grâce aux méthodes **set** et **get**. On peut effacer la valeur d'une variable de session en utilisant **remove** ou la totalité de la session en utilisant **clear**. La méthode **has** nous permet de savoir si une variable existe dans la session.

Voici un exemple complet :

```
// src/Controller/SessionController.php
class SessionController extends AbstractController
{
    /**
     * @Route("/session/creer/session")
     */
    public function creerSession(SessionInterface $s)
    {
        $s->set('nom', 'Pepita');
        return $this->render ('session/creer_session.html.twig');
    }

    /**
     * @Route("/session/afficher/session")
     */
    public function afficherSession (SessionInterface $s){

        if ($s->has('nom')){
            // on peut l'obtenir et l'envoyer à la vue
            $val = $s->get ('nom');
            return $this->render ('session/afficher_session.html.twig',
['valeurSession'=>$val]);
        }
        else {
            return new Response ("Le nom n'a pas été stocké dans la session");
        }
    }

    /**
     * @Route("/session/effacer/variable/session")
     */
    public function effacerVariableSession (SessionInterface $s){

        $s->remove('nom'); // efface la variable de session
        // $s->clear(); efface la totalité de la session!
        return $this->render ('session/effacer_variable_session.html.twig');
    }
}
```


}

14. Obtenir de données du GET et POST

Pour obtenir de données d'une requête **GET** ou **POST** on utilise :

```
$request->query->get ("cleParametre");
```

Exemple : obtention de valeurs des paramètres de l'URL ou requête GET

Cette action lira les valeurs d'une requête GET contenant de paramètres dans la URL :

<http://localhost:8000/exemple/get?nom=frigo&categorie=cuisine>

```
// src/Controller/GetPostController.php
class GetPostController extends AbstractController
{
    /**
     * @Route("/exemple/get", name="get")
     */
    public function exempleGet(Request $req)
    {
        $nom = $req->query->get ("nom");
        $categorie = $req->query->get ("categorie");

        // dump ($req->query);
        // die();
        return new Response ("nom : " . $nom . "<br>categorie: " . $categorie .
"<br>");
    }
}
```

Observez la différence entre obtenir un paramètre tu GET et obtenir un paramètre du router : dans les cas du GET on obtient d'abord le "query" (ParameterBag) contenant tous les paramètres.

Pour obtenir les données directement du POST on utilise :

```
$request->request->get ("cleParametre");
```

15. Le modèle : création d'entités et de la BD

Dans une application web, les classes qui contiennent les données de l'application s'appellent **entités** (les classes de notre diagramme de classes) et on doit les définir.

Présentation de Doctrine

Pour créer la structure de données d'une application web on va utiliser un ensemble de librairies de PHP qui s'appelle **Doctrine**. Doctrine est constitué d'un **ORM** (Objet Relational Mapping) et d'un **DBAL** (Database Abstraction Layer).

L'ORM permet de créer une **correspondance entre les classes** de notre application (qui est orientée objet) et **une base de données** (constitué de tableaux)

Ex: notre classe *Evenement* sera représentée dans la BD avec un tableau *evenement*

Le **DBAL** est un ensemble de librairies basées sur PDO qui facilite l'accès à la BD, tant à sa structure comme aux données.

Ex: on peut utiliser une méthode *select()* qui nous renverra des données de la BD sous la forme d'objets et pas d'array. On ne doit plus faire une requête à la main "SELECT * FROM" dans les requêtes les plus habituelles

Documentation de Doctrine : <https://symfony.com/doc/current/doctrine.html>

L'utilisation de base de Doctrine est simple. On doit réaliser trois étapes :

1. Définir les propriétés et les relations de **chaque entité** en utilisant des annotations, XML, YAML ou PHP

Ex. : définir les propriétés des entité *Realisateur* et *Film* et le type de lien existant entre elles (un à plusieurs, plusieurs à plusieurs, un à un...)

2. Demander à Doctrine de **créer le code PHP de base de ces entités** (nos classes, set et get inclus)
3. Demander à Doctrine de **créer la base de données contenant ces entités** et ses relations

La bonne nouvelle est que les pas 2 et 3 se font automatiquement ! Doctrine génère le code des classes, les tableaux dans la BD, les relations... et en plus on peut tout mettre à jour juste en changeant les fichiers de définition des entités !

Une fois créé le code des entités et le schéma de la BD, nous pouvons accéder en utilisant un objet gestionnaire de Doctrine (entity manager). Doctrine fournit les méthodes pour pouvoir **obtenir les données de la BD sous forme d'objets** (au lieu d'arrays associatifs) et **stocker des objets dans la BD** (qui seront stockés sous la forme d'un enregistrement, une ligne dans un tableau).

On va voir comment faire tout ça par la suite dans un nouveau projet **ProjetModelSymfony**

```
symfony new --full ProjetModelSymfony
```

Créez un contrôleur `ExemplesModele` vous-mêmes et une action de bienvenue. Suivez le guide rapide du création d'un projet (Annexe 1) pour le faire.

Documentation de Doctrine liée à Symfony :

<https://symfony.com/doc/current/doctrine.html>

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/working-with-objects.html>

Configuration de Doctrine et des paramètres de la BD

15.2.

1. **Rajoutez** les librairies de **Doctrine** à votre projet

```
composer require symfony/orm-pack  
composer require symfony/maker-bundle --dev
```

2. **Adaptez les paramètres de la BD** dans le fichier `.env` à votre serveur et nom de base de données

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

On va créer une base de données **bibliotheque**. Dans notre cas le password est vide, alors on obtient :

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

Si vous engagez les services d'un hébergeur vous devez mettre ici ses paramètres de connexion

3. **Créez la BD vide** depuis la console (dans le dossier de votre projet)

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

Création des entités et mise à jour de la BD

On va créer une première entité et générer son code.

1. Lancez l'assistant de création d'entités :

15.3.

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

L'assistant nous demandera le **nom de l'entité** (*Livre*) et **ses propriétés** (*titre, prix, description, datePublication*). D'abord il demande pour le nom de la propriété et puis pour le type. Les types de base (affichables en tapant "?") sont *string (255)*, *decimal (8,2)*, *text (400)*, *datetime*. Pour certains types il nous demande aussi la taille.

L'assistant créera les fichiers */Entity/Livre.php* (la classe !) et */Repository/LivreRepository.php*. **Si vous ratez la création de l'entité** sortez de l'assistant (CTRL-C), effacez les fichiers */Entity/Livre.php* et */Repository/LivreRepository.php* et recommencez.

Ouvrez le fichier *Livre.php* et observez qu'il s'agit d'une classe normale qui représente un livre et qui **contient des annotations qui décrivent certains de caractéristiques des champs**. Ces annotations seront utilisées par Doctrine pour générer automatiquement la base de données selon les types de notre choix. Pour plus d'information sur les types de doctrine, allez sur :

<http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/types.html>

L'assistant est juste un outil pour nous faciliter la tâche mais vous pouvez très bien créer/éditer les fichiers par vous-mêmes (rajouter/effacer des champs)

2. Réaliser la migration de la BD

La BD est actuellement vide. Nous voulons que Doctrine crée un tableau "Livre" à partir de notre entité. De manière plus générale, **nous voulons que Doctrine mette à jour la BD pour qu'elle reflète les changements dans nos entités (nom des entités, types des champs, relations, etc...)**. Ces mises à jours, assez habituelles dans un projet, s'appellent migrations.

Nous allons réaliser cette procédure en deux étapes très simples :

- 1) **Créer ou mettre à jour un historique de migrations** (tableau dans la BD) et un fichier qui contient le code pour mettre à jour la BD (dans *src/Migrations*)

```
php bin/console make:migration
```

- 2) **Mettre à jour la BD en lançant ce code** (ceci créera déjà notre tableau "Livre")

```
php bin/console doctrine:migrations:migrate
```

Note : Si vous le souhaitez, vous pouvez aussi créer les entités par vous-mêmes à la main, ou utiliser d'autres systèmes de notations au lieu des annotations tel que XML

<https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html>

Rajouter/effacer des propriétés d'une entité

Nous pouvons modifier nos entités facilement. Nous pouvons éditer le fichier à notre aise ou lancer à nouveau `make:entity` si on veut juste rajouter de nouvelles propriétés. Si on édite le fichier de l'entité (ici Livre.php) **à la main, on doit absolument créer les getters, setters et annotations pour les nouvelles propriétés.**

Dans tous les cas vous devez toujours **migrer la BD** en lançant les deux commandes déjà mentionnées :

```
php bin/console make:migration
```

```
php bin/console doctrine:migrations:migrate
```

Si on crée une propriété à la main, cette commande créera les getters et les setters pour nous

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

Cette action crée les classes Repository si elles n'existent pas (appuyez sur Enter quand on vous demande un namespace et Symfony créera tous les Repository)

Important : n'oubliez pas de rajouter le namespace au nom de la classe (ex : `App\Entity\Livre`)

Si on efface une propriété il ne faut pas oublier d'effacer aussi ses get et set, Symfony ne le fera pas automatiquement

On peut utiliser aussi la commande :

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

si on veut recréer tous les getters et les setters pour toutes les propriétés!

Exercices :

- 1) Créez une nouvelle propriété *isbn* et migrez la BD
- 2) Créez une nouvelle propriété *dateEdition* et migrez la BD. Supprimez-le et assurez-vous que la BD est toujours cohérente avec votre code
- 3) Créez l'entité Client (nom, prénom, email) et Exemple (état)
- 4) Créez l'entité Auteur (nom, nationalité)

16. Le modèle : les relations

Nous sommes capables maintenant de générer les tableaux qui correspondent à nos entités. Cette section est consacrée à l'implémentation des relations d'un schéma.

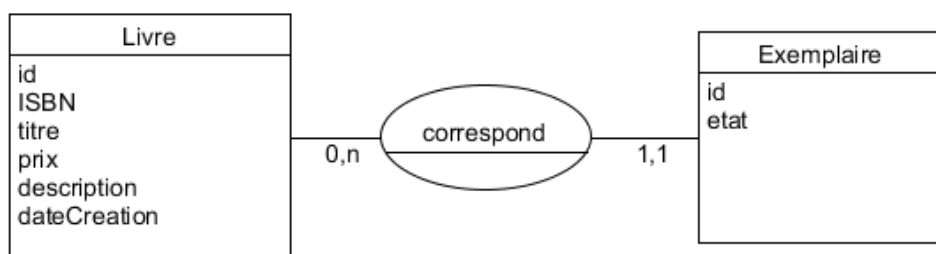
Vous trouverez les exemples décrits ci-dessous dans les projets **projetModelSymfony** et **projetRelationsSymfony** (regardez le code des entités !)

Documentation: <https://symfony.com/doc/current/doctrine/associations.html>

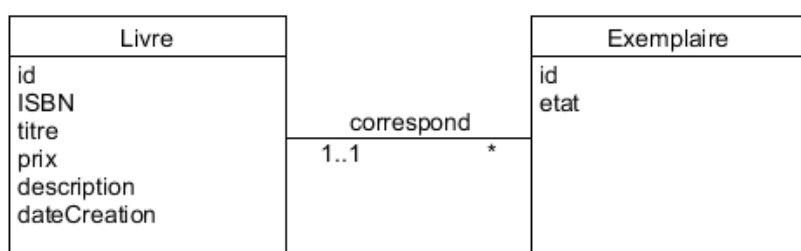
Relation Many-To-One

Considérons ce schéma Merise (MCD - base de données relationnelles) :

16.1.



Qui équivaut à ce schéma UML (POO) :



La transformation de ce modèle nous donne, si on la réalise à la main :

- Dans la BD : Un tableau Livre qui ne change pas + un tableau Exempleire qui contient une colonne id (qu'on va appeler *livre_id* pour faciliter la lecture)

- Dans le code : Une classe Livre qui ne change pas (ou qui a un array d'Exemplaires si la relation est bidirectionnelle) + une classe Exemple qui contient un objet Livre

Nous pouvons créer le code et modifier la BD à la main mais Doctrine va le faire pour nous.

Objectif :

Implémenter la partie correspondante aux associations dans le code des entités et créer les relations dans la BD

Procédure :

- 1. Rajoutez une propriété Many-to-One dans l'entité correspondante en utilisant l'assistant** (ici c'est l'entité *Exemple*)

La propriété est du type **relation, ManyToOne**.

Dans ce cas, la propriété s'appellera **livre** et sera un objet de la classe Livre. On souhaite aussi rajouter une propriété et de méthodes dans Livre pour créer une association bidirectionnelle (pas seulement pouvoir obtenir le Livre qui correspond à un exemple mais aussi tous les exemples d'un livre !). Cette propriété s'appellera **exemples** et sera une collection qui contient tous les exemples d'un livre.

```
C:\xampp\htdocs\Symfony4\projetModelSymfony>php bin/console make:entity Exempleaire

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> livre

Field type (enter ? to see all types) [string]:
> relation

What class should this entity be related to?:
> Livre

What type of relationship is this?
-----
Type          Description
-----
ManyToOne     Each Exempleaire relates to (has) one Livre.
               Each Livre can relate to (can have) many Exempleaire objects

OneToMany     Each Exempleaire can relate to (can have) many Livre objects.
               Each Livre relates to (has) one Exempleaire

ManyToMany     Each Exempleaire can relate to (can have) many Livre objects.
               Each Livre can also relate to (can also have) many Exempleaire objects

OneToOne      Each Exempleaire relates to (has) exactly one Livre.
               Each Livre also relates to (has) exactly one Exempleaire.
-----

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> ManyToOne
```

```
Is the Exempleaire.livre property allowed to be null (nullable)? (yes/no) [yes]:
>

Do you want to add a new property to Livre so that you can access/update Exempleaire objects?
>

A new property will also be added to the Livre class so that you can access the related Exempleaire objects.

New field name inside Livre [exemples]:
> exemples

updated: src/Entity/Exempleaire.php
updated: src/Entity/Livre.php
```

C'est fait ! Nos entités sont mises à jour (explication dans la section suivante)

2. Migrez la BD pour que les relations soient créées dans la BD

```
php bin/console make:migration
```

```
php bin/console doctrine:migrations:migrate
```

```
-> ALTER TABLE exemplaire ADD le_livre_id INT DEFAULT NULL
-> ALTER TABLE exemplaire ADD CONSTRAINT FK_5EF83C92F9970E22 FOREIGN KEY (le_livre_id) REFERENCES livre (id)
-> CREATE INDEX IDX_5EF83C92F9970E22 ON exemplaire (le_livre_id)
```

3. Vérifiez que la BD a été mise à jour

Observez les changements dans le/les tableaux !

Explication du code généré par l'assistant

Les opérations réalisées ont généré ce code dans les entités *Exemplaire* et *Livre*. Dans **Exemplaire.php** :

```
.
.
.

/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Livre", inversedBy="exemplaires")
 */
private $livre;

.
.
.

public function getLivre(): ?Livre
{
    return $this->livre;
}

public function setLivre(?Livre $livre): self
{
    $this->livre = $livre;

    return $this;
}
```

et dans **Livre.php** :

```
/**
 * @ORM\OneToMany(targetEntity="App\Entity\Exemplaire", mappedBy="livre")
 */
private $exemplaires;

.
.
.

/**
 * @return Collection|Exemplaire[]
 */
public function getExemplaires(): Collection
{
    return $this->exemplaires;
}

public function addExemplaire(Exemplaire $exemplaire): self
{

```

```

        if (!$this->exemplaires->contains($exemplaire)) {
            $this->exemplaires[] = $exemplaire;
            $exemplaire->setLivre($this);
        }

        return $this;
    }

    public function removeExemplaire(Exemplaire $exemplaire): self
    {
        if ($this->exemplaires->contains($exemplaire)) {
            $this->exemplaires->removeElement($exemplaire);
            // set the owning side to null (unless already changed)
            if ($exemplaire->getLivre() === $this) {
                $exemplaire->setLivre(null);
            }
        }

        return $this;
    }
}

```

Note importante : ce code est la seule chose qui change dans la classe originale. Si vous **avez fait des erreurs** pendant la création des associations avec l'assistant, vous devez **effacer ces lignes et recommencez la création** des associations (pas créer les entités depuis zéro !)

On a créé une association bidirectionnelle de Many-to-One. Observez que dans Livre il y aura une collection d'exemplaires et que dans Exemplaire aura un objet Livre. **Doctrine a généré** :

- Les **annotations** pour indiquer le type d'association entre les deux classes (on utilise *inversedBy* du côté *plusieurs* et *mappedBy* du côté *un*)
- Les **méthodes** get et set pour accéder aux objets des deux côtés de l'association. Dans le cas de la collection, de méthodes pour rajouter un élément à la collection et pour l'effacer de la collection.

Notez que Doctrine efface la "s" quand il crée les méthodes ! Notre propriété s'appelle **exemplaires** mais les méthodes créées s'appellent par exemple removeExemplaire au lieu de removeExemplaires !

Observez aussi que **les méthodes spécifient les types de retour**. **Collection** est une interface de PHP qui assure que les objets qui l'implémentent soient Countables, Transversables et qu'on puisse les encoder en JSON avec json_encode.

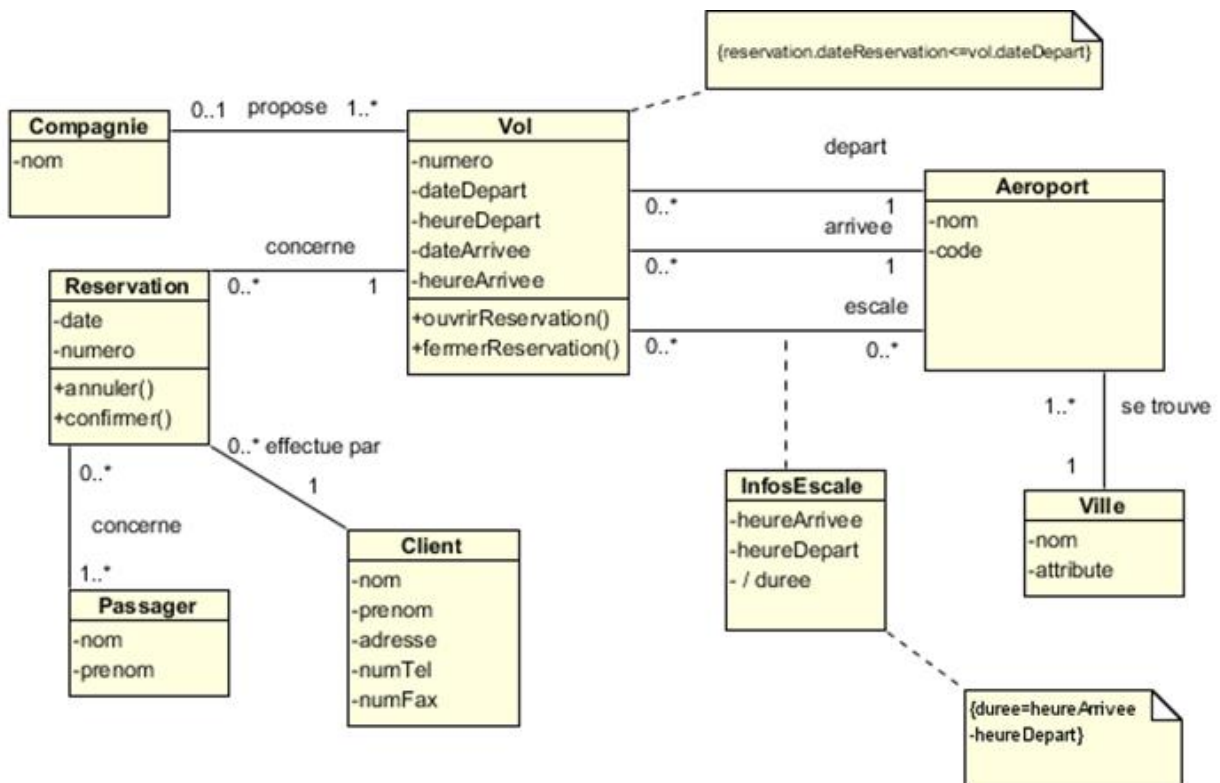
self est une manière d'accéder au nom de la classe (ça aurait pu être remplacé par le nom de la classe dans chaque cas où il apparaît dans le code)

Le code est généré, ainsi que les tableaux et les relations.

Dans la base de données il y aura une colonne *livre_id* dans Exemplaire. Le tableau Livre par contre ne changera pas car on ne peut pas mettre une collection d'exemplaires dans une BD relationnelle ! ;)

Exercices :

- 1) On va considérer qu'un Client de la bibliothèque a une Adresse (rue, numero, codePostal, ville, pays) et qu'une adresse peut correspondre à plusieurs clients. Créez la classe Client pour représenter les clients et la classe Adresse pour représenter l'adresse de chaque client. Implémentez-la en utilisant Doctrine tel qu'on vient de faire
- 2) Implementez ce diagramme

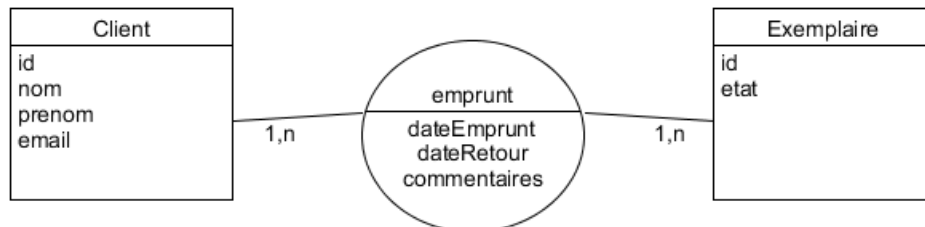


- 3) Prenez des exemples du cours d'UML et implémentez les entités avec Doctrine dans un nouveau projet. Si vous êtes en train de planifier un projet pour vous, prenez plusieurs entités de votre schéma et implémentez-les avec Doctrine. Implémentez les associations aussi

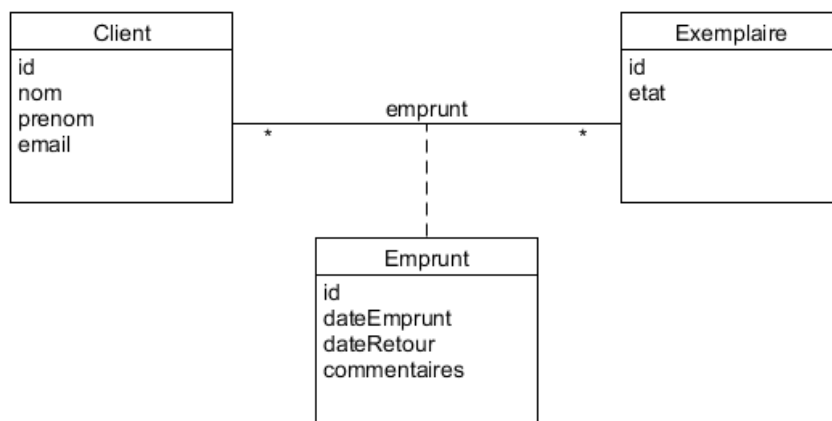
Relation Many-To-Many

Considérez le schéma MCD suivant :

16.3.

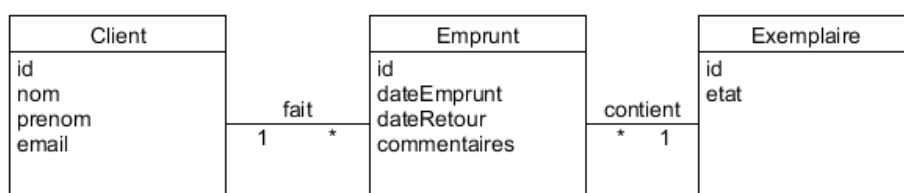


Équivalent à ce schéma UML :



Nous sommes dans une association de plusieurs à plusieurs qui contient d'attributs d'association. Symfony a son propre mécanisme pour implémenter ces associations (<many-to-many>), toutefois on ne l'utilisera pas ce mécanisme **parce qu'il ne permet pas d'inclure d'attributs dans l'association !**

Mais la solution est simple : on peut transformer l'association dans deux associations de one-to-many :



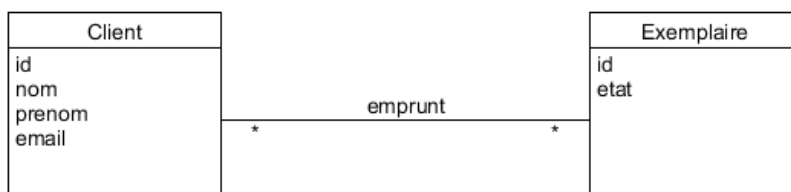
Exercice : implémentez vous-même ce modèle pour avoir la relation entre les Clients et les exemplaires !

Si vous êtes toujours intéressé à implémenter une association de plusieurs à plusieurs **sans attributs**, suivez les exemples de la documentation de Doctrine :

<https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html#many-to-many-bidirectional>

Voici un exemple :

Imaginons qu'on a juste besoin de connaître qui sont les clients qui ont emprunté un exemplaire et vice-versa. Nous aurions ce schéma :



1. Créez les entités **Client** et **Exempleire** **dans un autre projet** pour ne pas détruire votre schéma !
2. Suivez la même procédure que pour les relations Many-to-One mais choisissez Many-to-Many

Observez les changements dans le code des entités ainsi que dans le schéma de la BD

Relation One-To-One

Ces types de relations sont moins fréquentes, mais si vous avez besoin de les réaliser la documentation se trouve ici :

16.4.
<https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html#one-to-many-bidirectional>

L'implémentation en soit est facile. Voici un exemple :

Considérez par exemple qu'un client peut avoir un avatar (un fichier d'image) et qu'un avatar appartient à un seul client. On ne veut pas stocker les fichiers dans le tableau client, on veut carrément une autre entité.

1. Créez l'entité Avatar
2. Suivez la même procédure que pour les relations Many-to-One mais choisissez One-To-One

Observez les changements dans le code des entités ainsi que dans le schéma de la BD

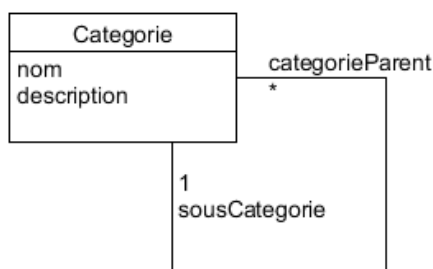
Relation récursive (self-association)

16.5.1. Relation récursive d'un à plusieurs

16.5.

Nous pouvons aussi implémenter une entité qui a une référence à soi-même. L'exemple le plus typique est celui des parents-enfants ou chef-employé. On peut modéliser ce cas en utilisant une relation one-to-many dans la même entité.

Considérons un magasin qui organise les produits en catégories. Une catégorie peut avoir des sous-catégories mais une sous-catégorie appartient uniquement à une catégorie.



1. Créez l'entité **Categorie**
2. Créez une propriété **sousCategories** du type relation (OneToMany). La propriété complémentaire sera **categorieParent**

Observez les changements dans le code des entités ainsi que dans le schéma de la BD

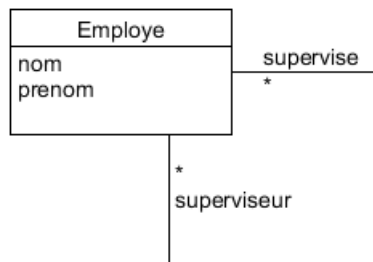
relations categorie	
id : int(11)	
categorie_parent_id : int(11)	
nom : varchar(255)	
description : varchar(255)	

On obtient un seul tableau dans la BD et un champ extra qui indique la relation (à cause du one-to-many).

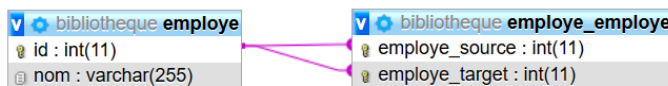
Dans le code PHP (Habitant.php) on obtient deux listes, une pour les sous-catégories et une autre pour les catégories-parent. Aucun tableau extra sera généré.

16.5.2. Relation récursive de plusieurs à plusieurs

Si la relation est de **plusieurs à plusieurs sans attributs** (ex. : une Personne supervise plusieurs Personnes et elle est à son tour Supervisé par d'autres Personnes), on peut utiliser une **relation many-to-many** (si on n'a pas d'attributs dans la relation).



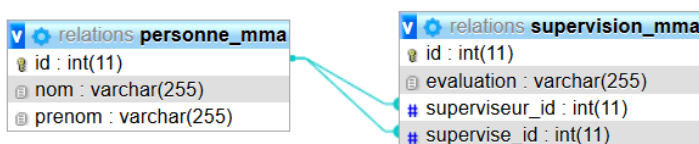
1. Créez l'entité `Employe`
2. Créez une propriété **`supervises`** (attention au pluriel) du type relation (`ManyToOne`). La propriété complémentaire sera **`superviseurs`**
3. Observez le code et le schéma généré



Si la relation avait eu des attributs, on aurait créé deux entités (Personne et Supervision) nous-mêmes et on aurait appliqué la méthode de many-to-one.

Voici un exemple (on rajoute le suffixe MMA pour ne pas écraser les autres entités).

1. Créez l'entité **PersonneMMA**
2. Créez l'entité **SupervisionMMA**
3. Créez une propriété dans **SupervisionMMA** du type relation qui porte le nom **superviseur** et pointe vers l'entité **PersonneMMA** (ManyToOne). La propriété complémentaire sera **supervisionsSuperviseur**
4. Créez une propriété dans **SupervisionMMA** du type relation qui porte le nom **supervise** et pointe vers l'entité **PersonneMMA** (ManyToOne). La propriété complémentaire sera **supervisionsSupervise**
5. Observez le code et le schéma généré



Sur les associations récursives :

<http://www.tomjewett.com/dbdesign/dbdesign.php?page=recursive.php>

17. Le modèle : accès à la BD avec Doctrine

Documentation: <https://symfony.com/doc/3.3/doctrine.html>

SELECT

Créez à la main quelques enregistrements dans la base de données (Bibliotheque, table Livres) car nous allons réaliser quelques requêtes de sélection.

17.1.

Pour pouvoir réaliser un Select dans la BD nous pouvons utiliser les **méthodes de base de Doctrine** associés à notre Entité. Quand on crée une entité, sa classe **Repository** est créée aussi (ex : Client.php et ClientRepository.php sont générées).

Cette classe contient **de méthodes capables de réaliser de requêtes hérités de \Doctrine\ORM\EntityRepository**.

Par défaut il existe les méthodes suivantes (hérités) :

- find : sélection par id. Renvoie un objet
- findBy : sélection avec de critères (dans un array associatif). Renvoie un array d'objets
- findOneBy : pareil que findBy mais elle renvoie que le premier enregistrement. Renvoie un objet
- findAll : Renvoie un array d'objets contenant tous les éléments du tableau

Voici un exemple d'utilisation de chaque méthode (actions dans le controller). Créez vous-mêmes les vues et les routes.

```
// SELECT: findOneBy
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use App\Entity\Livre; // nous permet d'écrire Livre::class

class ExemplesModeleController extends AbstractController
{
    /**
     * @Route ("/exemples/modele/exemple/find/one/by")
     */
    public function exempleFindOneBy (){
        // obtenir le entity manager
        $em = $this->getDoctrine()->getManager();
        // obtenir le repository
        $rep = $em->getRepository(Livre::class);

        // on obtient l'objet, le filtre est envoyé sous la forme d'un array
        $livre = $rep->findOneBy (array('titre'=>'Life and Fate'));
```

```

        // on stocke le résultat dans un array associatif
        // pour l'envoyer à la vue comme d'habitude
        $vars = ['unLivre' => $livre];

        // on renvoie l'objet à la vue, rien ne change ici
        return $this->render ("exemples_modele/exemple_find_one_by.html.twig",
        $vars);
    }
}

```

L'objet `$em` est un objet de la classe `EntityManager`. Le **entity manager** gère le **CRUD des entités** dans la BD. Tel qu'on l'a mentionné ci-dessus, l'objet repository (ici `$rep`) possède de méthodes qui nous facilitent ce CRUD. Par défaut nous disposons de l'ensemble de méthodes de base mentionné mais **on peut rajouter d'autres méthodes dans nos repository pour réaliser des requêtes plus complexes.**

```

// SELECT: find (chercher par id)

/**
 * @Route ("exemples/modele/exemple/find")
 */
public function exempleFind (){
    $em = $this->getDoctrine()->getManager();
    $rep = $em->getRepository(Livre::class);

    $livre = $rep->find(2);
    $vars = ['unLivre' => $livre];
    return $this->render("exemples_modele/exemple_find.html.twig",$vars);
}

```

```

// SELECT: findBy (chercher par un ou plusieurs champs, filtre array)

/**
 * @Route ("exemples/modele/exemple/find/by")
 */
public function exempleFindBy (){
    $em = $this->getDoctrine()->getManager();
    $rep = $em->getRepository(Livre::class);

    // notez que findBy renverra toujours un array même s'il trouve
    // qu'un objet
    $livres = $rep->findBy(array ('prix'=>'18',
                                'datePublication'=>new \DateTime('1960-1-1')));

    $vars = ['livres' => $livres];
    return $this->render("exemples_modele/exemple_find_by.html.twig",$vars);
}

```

```

// SELECT: findAll (chercher par un ou plusieurs champs, filtre array)

/**
 * @Route ("exemples/modele/exemple/find/all")

```

```

*/
public function exempleFindAll (){
    $em = $this->getDoctrine()->getManager();
    $rep = $em->getRepository(Livre::class);

    // notez que findBy renverra toujours un array même s'il trouve
    // qu'un objet
    $livres = $rep->findAll();
    $vars = ['livres' => $livres];
    return $this->render("exemples_modele/exemple_find_all.html.twig",$vars);
}

```

Créez de vues qui reçoivent ces objets en sachant que pour accéder aux propriétés des objets depuis la vue vous pouvez utiliser la notation d'objet (.)

```
{{ livre.titre }}
```

Exercices :

- 1) Créez une méthode qui obtient la liste de tous les clients (remplissez la BD d'abord ;))
- 2) Créez une méthode qui obtient tous les clients qui s'appellent Marie Dupont
- 3) Créez une méthode qui obtient le client qui porte l'id numéro 3 dans la BD

INSERT et UPDATE

Quand on crée un objet (ex : Livre) il n'a aucune relation avec la BD: il est dans les **domaine des objets**. Si on lance la méthode **persist** sur cet objet, **l'objet sera lié au domaine de la BD** mais aucune requête sera lancée pour le moment, l'objet ne sera pas encore dans la BD. Nous pouvons alors :

1. Créer un objet
2. Lancer **persist** pour lier l'objet avec la BD (on ne modifie pas la BD encore!)
3. Modifier le contenu de l'objet
4. Lancer **flush** pour l'enregistrer/mettre à jour dans la BD

L'opération **flush** **applique dans la BD les changements qu'on a réalisés dans le domaine des objets** (ex : obtenir un livre de la BD, le modifier dans le domaine des objets et le réenregistrer dans la BD).

Les méthodes **persist** et **flush** n'appartiennent pas à la classe Repository, ils font partie du Entity Manager de Doctrine. On discutera ces deux méthodes dans un contexte plus global (unité de travail) dans la section "Persistence" plus bas dans ce tutoriel.

Voyons quelques exemples d'insertion et mise à jour à continuation.

17.2.1. INSERT

```
// INSERT
/**
 * @Route ("exemples/modele/exemple/insert")
 */
public function exempleInsert()
{
    $em = $this->getDoctrine()->getManager();
    // créer l'objet
    $livre = new Livre();
    $livre->setTitre("Guerre et paix");
    $livre->setPrix(20);
    $livre->setDescription(" l'histoire de la Russie à l'époque de Napoléon Ier, notamment la campagne de Russie en 1812. Léon Tolstoï y développe une théorie fataliste de l'histoire, où le libre arbitre n'a qu'une importance mineure et où tous les événements n'obéissent qu'à un déterminisme historique inéluctable. ");

    // lier l'objet avec la BD
    $em->persist($livre);
    // écrire l'objet dans la BD
    $em->flush();
    return $this->render("exemples_modele/exemple_insert.html.twig");
}
```


17.2.2. UPDATE

```
// UPDATE
/**
 * @Route ("exemples/modele/exemple/update")
 */
public function exempleUpdate (){
    $em = $this->getDoctrine()->getManager();
    // on obtient d'abord un livre
    $unLivre = $em->getRepository(Livre::class)->findOneBy(array("titre"=>"Vie et
destin"));
    $unLivre->setTitre ("Toto est content");
    // pas besoin de persist
    // quand on obtient un objet de la BD
    // $em->persist ($unLivre);
    $em->flush();
    return $this->render ("exemples_modele/exemple_update.html.twig");
}
```

17.2.3. DELETE

```
// DELETE
/**
 * @Route ("exemples/modele/exemple/delete")
 */
public function exempleDelete (){
    $em = $this->getDoctrine()->getManager();
    $unLivre = $em->getRepository(Livre::class)->findOneBy(array("titre"=>"1989"));
    // pas besoin de persist
    // quand on obtient un objet de la BD
    // $em->persist ($unLivre);
    $em->remove ($unLivre);
    $em->flush();
    return $this->render ("exemples_modele/exemple_delete.html.twig");
}
```

Exercices :

1. Créez une méthode qui efface un client de la BD
2. Créez une méthode qui insère deux Livres
3. Pour nous faciliter la création des entités et ne pas devoir affecter les propriétés avec les méthodes Set, modifiez les constructeurs de vos entités pour qu'il puissent recevoir un array contenant les paramètres pour construire chaque entité. Créez une méthode hydrate pour chaque entité. Cette méthode sera appelée si le constructeur reçoit un array de couples propriété-valeur (voir le cours d'hydratation).

18. Le modèle : persistance

Les actions qu'on réalise sur les entités (modification, mise à jour...) sont regroupées dans une **unité de travail (Unit of Work)**.

L'unité de travail contient l'état de tous les objets et réalise de requêtes à la BD. Elle est en charge de maintenir la **cohérence entre le modèle et la BD**

Documentation :

<http://doctrine-orm.readthedocs.io/en/latest/reference/unitofwork.html>

L'unité de travail fonctionne de la manière suivante :

- La méthode **persist** (\$objet) **rajoute une entité à l'unité de travail** (si elle n'y était pas présente déjà).

Ex : quand on vient de créer un Livre avec **new**.

L'entité devient "gérée" ("**managed**")

- Quand **on fait un select de la BD, les entités obtenues se trouvent directement dans l'unité de travail** et on n'a pas besoin de faire **persist**. Par contre, si on crée une entité juste avec **new**, elle fera partie de l'unité de travail uniquement quand on lancera **persist**
- Si on veut **enlever une entité de l'unité de travail** et la rendre indépendante, on utilise **detach**. L'objet devient indépendant de l'entity manager et les modifications n'auront pas d'effet sur la BD même si on lance **flush**
- **persist** rajoute une entité à l'unité de travail et rend cette instance "gérée" (c'est-à-dire que les futures mises à jour de l'entité seront suivies et la BD sera modifiée quand on fera **flush**).
- **merge** crée une **nouvelle entité**. L'entité que vous transmettez en paramètre ne sera pas gérée, mais uniquement la nouvelle. On n'a pas besoin de l'appliquer **persist** pour que ses modifications soient appliquées à la BD (il suffira de faire **flush**) car elle est déjà gérée
- **refresh** recharge le contenu de l'entité de la BD
- Chaque fois qu'on appelle **flush**, Doctrine vérifie l'unité de travail et synchronise les objets avec la BD (change la BD en fonction du contenu des objets qui se trouvent dans l'unité de travail).

// DETACH

```

/**
 * @Route ("/exemples/modele/exemple/detach")
 */
public function exempleDetach (){
    $em = $this->getDoctrine()->getManager();
    $livre = $em->getRepository(Livre::class)->findOneBy(array("titre"=>"Life and
Fate"));
    $livre->setTitre("Totorito");
    $em->detach($livre);
    // ce flush ne fera rien, l'entité a été détachée de l'unité du travail
    $em->flush();
    dump ($livre);
    die();
    return $this->render ("exemplesModele/exemple_detach.html.twig");
}

// MERGE

/**
 * @Route ("/exemples/modele/exemple/merge")
 */
public function exempleMergeEntite (){
    $em = $this->getDoctrine()->getManager();
    $livre = new Livre ();
    $livre->setTitre ("Pizza pizza");
    // on crée une copie de l'entité. Cette copie sera gérée
    // dans l'unité mais pas le livre original
    $nouveauLivre = $em->merge ($livre);
    $livre->setTitre ("Tururu"); // ne changera pas dans la BD
    //quand on fera flush plus bas

    // ce livre changera, cette copie est déjà dans l'unité de travail.
    // observez qu'il n'y aura pas un "persist"
    $nouveauLivre->setTitre ("Nonono");
    $nouveauLivre->setPrix (40);
    $nouveauLivre->setDescription ("Super");
    $nouveauLivre->setDatePublication (new \DateTime);
    // ce flush mettra à jour la BD pour "nouveauLivre"
    $em->flush();
    return $this->render ("exemples_modele/exemple_merge.html.twig");
}

// REFRESH

/**
 * @Route ("/exemples/modele/exemple/refresh")
 */
public function exempleRefresh (){
    $em = $this->getDoctrine()->getManager();
    $unLivre = $em->getRepository(Livre::class)->findOneBy(array("titre"=>"Life and
fate"));
    $unLivre->setTitre ("La vie est belle");
    // dump ($unLivre);

    // recharge le livre de la BD, il y aura le titre original

```

```
$em->refresh($unLivre);  
// dump ($unLivre);  
// die();  
$em->persist ($unLivre);  
// rien ne change dans la BD  
$em->flush();  
return $this->render ("exemples_modele/exemple_refresh.html.twig");  
}
```

Transitivité en Cascade

Quand nous avons des associations entre les entités, nous avons la possibilité d'indiquer à Symfony **de propager l'opération réalisé sur une entité en cascade sur les entités associées**.

Exemple : on efface un livre et on provoque l'effacement de tous ses exemplaires en cascade
18.1.

Nous avons plusieurs possibilités :

cascade-persist : Si on a une entité qui contient de références à d'autres entités, et nous modifions/rajoutons ces dernières, nous allons devoir faire uniquement **persist** sur la première et Doctrine fera persist sur toutes les entités associées.

Exemple : nous obtenons un Livre auquel on rajoute des exemplaires. Si nous faisons **persist** sur le Livre, l'opération sera transmise en cascade à tous les exemplaires. Autrement on devrait lancer **persist** sur chaque exemplaire.

cascade-remove : Si on efface une entité, l'effacement sera propagé en cascade. Si l'entité n'a pas eu un **persist**, elle n'est pas effacée de la BD mais ses entités associées le seront. Si on a enlevé l'entité de l'unité de travail (clear), remove enverra une exception.

cascade-detach: le détachement se transmet en cascade

cascade-merge: pareil que la précédente mais pour **merge**

cascade-refresh : pareil que la précédente mais pour **refresh**

cascade-all : Implique toutes les opérations précédentes. Peut dégrader la performance.

Exemple : Réalisation d'un INSERT des objets d'une relation One-to-Many sans cascade-persist

Observez cet exemple où on crée un Livre et plusieurs Exemplaires, et on stocke le tout dans la BD (créez un nouveau controller et importez les classes Livre et Exemplaires) :

```
class ExemplesCascadeController extends AbstractController
{
    /**
     * @Route("/exemples/cascade/exemple/sans/encapsulation")
     */
    public function exempleSansEncapsulation (){

        $em = $this->getDoctrine()->getManager();
        // on crée un livre
```

```

        $livre = new Livre();
        $livre->setTitre("Confesión de un asesino");
        $livre->setPrix (20);
        $livre->setDescription("Roman");
        $livre->setDatePublication(new \DateTime("1968:10:10 00:00:00"));
        // on crée deux exemplaires de ce Livre
        $exemplaire1 = new Exemple();
        $exemplaire1->setEtat("tache de chocolat");
        $exemplaire2 = new Exemple();
        $exemplaire2->setEtat("très vieux");
        $livre->addExemple($exemplaire1);
        $livre->addExemple($exemplaire2);

        // Observez que l'exemple fait référence à son livre
        // grâce au code généré par l'assistant dans "addExemple"
        // car on a choisi de créer une association bidirectionnelle!
        //dump ($exemplaire1->getLivre());
        //die();

        // nous n'avons pas besoin d'indiquer nous-mêmes qui est
        // le livre de chaque exemple!
        // $exemplaire1->setLivre($livre);    // pas besoin
        // $exemplaire2->setLivre($livre);    // pas besoin

        $em->persist($livre);
        $em->flush();

        return $this->render
("exemples_cascade/exemple_sans_encapsulation.html.twig");
    }

}

```

Normalement le code devrait insérer un Livre et deux Exemples dans la BD, et puis mettre à jour la clé étrangère (livre_id) de chaque exemple.

Mais si vous le lancez-vous obtenez :

Multiple non-persisted new entities were found through the given association graph:

* A new entity was found through the relationship 'App\Entity\Livre#exemplaires' that was not configured to cascade persist operations for entity: App\Entity\Exemplaire@0000000048bcd3dd0000000009129465. To solve this issue: Either explicitly call EntityManager#persist() on this unknown entity or configure cascade persist this association in the mapping for example @ManyToOne(..., cascade={"persist"}). If you cannot find out which entity causes the problem implement 'App\Entity\Exemplaire#__toString()' to get a clue.

* A new entity was found through the relationship 'App\Entity\Livre#exemplaires' that was not configured to cascade persist operations for entity: App\Entity\Exemplaire@0000000048bcd3f90000000009129465. To solve this issue: Either explicitly call EntityManager#persist() on this unknown entity or configure cascade persist this association in the mapping for example @ManyToOne(..., cascade={"persist"}). If you cannot find out which entity causes the problem implement 'App\Entity\Exemplaire#__toString()' to get a clue.

Symfony remarque qu'on n'a pas fait **persist** des objets associés (les Exemplaires du Livre qu'on vient de créer). Pour que le mécanisme fonctionne, nous avons deux possibilités :

- Lancer **persist** pour les exemplaires avant de lancer le persist du Livre (pas pratique)

```
$em->persist($exemplaire1);
$em->persist($exemplaire2);
$em->persist($livre);
.
.
.
```

- Spécifier que la **persistance doit se réaliser en cascade** dans l'annotation de l'association (**fichier de l'entité**). Modifiez le fichier de l'entité Livre.php

```
/**
 * @ORM\OneToMany(targetEntity="App\Entity\Exemplaire", mappedBy="livre",
 *                cascade={"persist"})
 */
private $exemplaires;
```

IMPORTANT : Observez que les deux côtes de la relation sont mises à jour dans la méthode addExemplaire :

1. L'exemplaire est rajouté à la liste d'exemplaires dans Livre
2. La propriété Livre est affecté dans l'objet exemplaire. Cela permet que quand on réalisera la migration dans la BD, le livre_id sera mis à jour sans devoir le faire à la main !

Exercices :

1. Effacez un livre et provoquez que les exemplaires soient effacés automatiquement. Modifiez la configuration de cascade pour que l'opération soit réalisée correctement (exerciceCascadeRemove)
2. Créez une méthode qui rajoute deux clients et une adresse. Faites bien attention au sens de cette association (côté Many et côté One). Modifiez les attributs de cascade pour pouvoir faire le persist
3. Créez une méthode qui cherche un client dans la BD et puis l'efface, y incluses toutes ses adresses

Encapsulation

18.2. Le code de l'exemple précédant crée un Livre, puis crée deux exemplaires et pour finir stocke le tout dans la BD.

La méthode utilisée fonctionne mais on peut l'améliorer :

Pour le moment on doit toujours créer les objets Livre et Exemple dans l'action principale. On doit alors **connaître l'existence et importer toutes ces classes dans le controller !**

On peut aller plus loin en utilisant le concept d'**encapsulation**. L'encapsulation est un principe de la programmation orienté objet qui, entre autre, propose de **cacher la visibilité des objets ou des parties d'un objet**. Dans ce cas on veut permettre au développeur de rajouter des exemplaires sans que le controller aie besoin de connaître l'existence de la classe Exemple.

Le mécanisme est très simple : **au lieu de créer l'objet exemple dans l'action du controller et l'envoyer à la méthode qui le rajoute au livre (addExemple dans Livre), on créera l'exemple à l'intérieur d'une nouvelle méthode de Livre**. Cette méthode du livre sera **appelée depuis l'action, qui l'enverra les données nécessaires pour créer l'objet exemple mais pas l'objet en soi**. Voici un exemple, observez les différences avec l'exemple précédant :

Nouvelle méthode dans Livre.php :

```
// rajouté pour permettre l'encapsulation
public function addExempleNoClass ($etat){
    $exemple = new \App\Entity\Exemple();
    $exemple->setEtat($etat);
    $this->addExemple($exemple);
}
```

Nous n'avons plus besoin de la classe Exemple dans l'action principale et en plus son code sera simplifié :


```

/**
 * @Route("/exemples/encapsulation/rajouter/livre/exemplaires/encapsulation")
 */
public function rajouterLivreExemplairesEncapsulation (){
    $em = $this->getDoctrine()->getManager();
    // on crée un livre
    $livre = new Livre();
    $livre->setTitre("Curucucu Paloma");
    $livre->setPrix (20);
    $livre->setDescription("Roman");
    $livre->setDatePublication(new \DateTime("1968:10:10 00:00:00"));

    // on ne crée pas ici les exemplaires. On envoie les données
    // nécessaires pour créer les objets Exemple à la nouvelle méthode
    // de l'entité Livre
    // Cette méthode mettra à jour les deux côtés de la relation
    // car elle fait appel à addExemple
    $livre->addExempleNoClass("tache de chocolat", "15A");
    $livre->addExempleNoClass("très vieux", "13B");

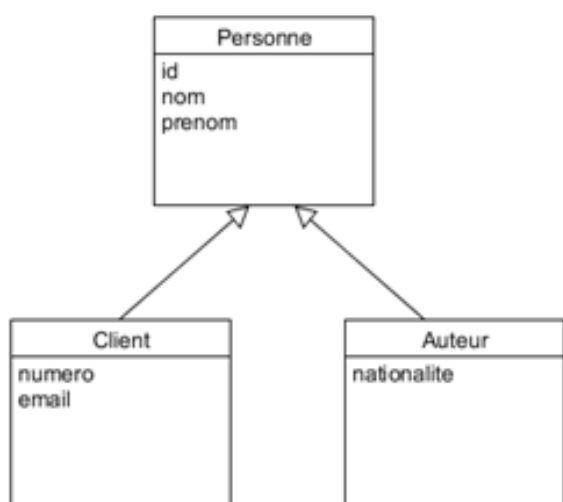
    $em->persist($livre);
    $em->flush();
    return $this->render
    ("exemples_encapsulation/rajouter_livre_exemplaires_encapsulation.html.twig");
}

```

Comparez ce code avec celui de "rajouterSansEncapsulation"...

19. Héritage de classes et implémentation dans la BD

Exemple : Les clients et les auteurs d'une application sont tous de personnes. Implementons ce modèle en code et dans la BD



Note : cr

Nous pouvons approcher ce problème de deux formes différentes :

1. **Single Table Inheritance** : On crée un seul tableau contenant les propriétés des trois entités. Dans le code il y a trois entités mais dans la BD il y a qu'une. Pour savoir si une ligne dans le tableau correspond à une entité ou une autre on rajoutera une colonne "discriminatrice" qui indiquera le type de la ligne. Simple, rapide et sans jointures.
2. **Class Table Inheritance** : On crée un tableau pour chaque entité. Plus lourd, pas toujours stable. Chaque query, même les très simples, demanderont la réalisation d'une jointure.

Single Table Inheritance

L'héritage de table unique (Single Table inheritance) est une stratégie de mappage d'héritage dans laquelle toutes les classes d'une hiérarchie sont mappées vers une seule table de base de données. Afin de distinguer quelle ligne du tableau représente quel type dans la hiérarchie, une colonne dite "discriminator" est utilisée.

- 1) **Créez les entités** enfants et parent : ClientH, AuteurH et PersonneH
- 2) Rajoutez **extends** *PersonneH* dans les définitions des classes filles pour indiquer à Doctrine la présence d'un héritage
- 3) **Rajoutez les annotations InheritanceType, DiscriminatorColumn et DiscriminatorMap** à la classe **parent**

InheritanceType indique le type d'héritage. Ici c'est Single Table

DiscriminatorColumn indique le nom de la colonne qui contiendra la valeur qui nous indique à quelle classe fille correspond la ligne (ici "auteurH" ou "clientH")

DiscriminatorMap indique les valeurs concretes de la colonne indiquée dans DiscriminatorColumn

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneHRepository")
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr",type="string")
 *
 * @DiscriminatorMap({"personneH"="PersonneH", "auteurH"="AuteurH", "clientH"="ClientH"})
 */
class PersonneH
{
    .
    .
    .
}
```

- 4) **Migrez la BD** et observez le résultat dans PHPMysqlAdmin

id	nom	prenom	discr	nationalite	numero	email
----	-----	--------	-------	-------------	--------	-------

Bien que nous avons trois entités au total, la méthode de Single Table crée une seule table contenant une colonne (discr) qui indiquera à quelle classe fille correspond la ligne (dans notre cas le colonne contient "auteurH" ou "personneH")

Les **régles** à suivre sont :

- @InheritanceType et @DiscriminatorColumn doivent être spécifiés dans la classe la plus haute appartenant à la hiérarchie des entités mappées
- @DiscriminatorMap indique le type d'une ligne. Ici, une valeur de "personneH" identifie une ligne comme étant de type PersonneH et "auteurH" identifie une ligne comme étant de type AuteurH.

On peut maintenant faire le CRUD de nos entités ...


Exemple : insérer un Client et un Auteur dans la base de données

```
class ExemplesHeritageController extends AbstractController
{
    /**
     * @Route("/exemples/heritage/insérer/client/auteur")
     */
    public function insererClientAuteur(){
        $em = $this->getDoctrine()->getManager();
        // créer l'objet
        $client = new ClientH();
        $client->setNom("López");
        $client->setPrenom("Jean");
        $client->setEmail ("jean.lopez@lala.de");
        $client->setNumero(200);
        $auteur = new AuteurH();
        $auteur->setNom("Lucas");
        $auteur->setPrenom("George");
        $auteur->setNationalite("USA");

        // lier les objets avec la BD
        $em->persist($client);
        $em->persist($auteur);

        // écrire les objets dans la BD
        $em->flush();
        return new Response ("Ok, objets insérés");
    }
}
```

Nous devons uniquement créer un Client et l'insérer, Doctrine remplira tant le tableau parent avec la colonne discriminateur ! Les valeurs qui ne concernent pas chaque entité respective seront *NULL* bien évidemment

<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	5	Lucas	George	auteurH	USA	NULL	NULL
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	6	López	Jean	clientH	NULL	200	jean.lopez@lala.de

Class Table Inheritance

Nous n'allons pas développer cette méthode maintenant mais vous avez la documentation ici :

¹⁹²
<https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/inheritance-mapping.html#class-table-inheritance>

20. Accès à la BD avec DQL

Nous avons vu comment réaliser de requêtes CRUD simples, mais dans un projet réel nous allons devoir lancer de requêtes assez plus complexes, tels que de regroupements (GROUP BY), de jointures de tableaux (JOIN) ou même de sous-requêtes.

Pour ce faire, on peut utiliser :

1. Du **SQL pur en PHP** (tel qu'on a fait jusqu'à maintenant)
2. **DQL**: Doctrine Query Language, similaire à SQL
3. **QueryBuilder**: une API qui nous permet de créer la requête en utilisant uniquement la POO

On va nous concentrer sur les méthodes 2 et 3. Dans cette section concrètement on va étudier la méthode numéro 2: DQL.

Documentation de DQL:

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html>

Exemples : projetDQLSymfony

DQL utilise des objets, pas de tableaux. Nos requêtes doivent être basées sur notre modèle de classes. Ça implique qu'on ne peut pas, par exemple, faire une jointure de deux tableaux qui n'ont pas de relation dans le modèle de classes.

Nous pouvons réaliser des requêtes de SELECT, UPDATE et DELETE. Pour les INSERTS on doit utiliser la méthode déjà expliquée de persistance (créer l'objet, le rendre persistant et le stocker dans la BD en lançant flush).

Passons aux exemples d'utilisation pour mieux comprendre.

SELECT

20.1.1. Requête qui renvoi un array d'arrays

```
20.1.
// Exemple de SELECT uniquement des titres des livres
// qui coutent plus de 15 euros en DQL,
// on obtient un array de strings, pas d'objets

/**
 * @Route ("/exemples/dql/exemple/select/array/arrays")
 */
public function exempleSelectArrayArrays (){
    $em = $this->getDoctrine()->getManager();
    $query = $em->createQuery ("SELECT livre.titre, livre.prix FROM
App\Entity\Livre livre WHERE livre.prix>15");
    $resultat = $query->getResult();
    $vars = ['livres'=> $resultat];
    return $this->render ("exemples_dql/exemple_select_array_arrays.html.twig",
$vars);
}
```

- "livre" est un **alias** pour la classe Livre. Toutes les **entités de cette classe** qui **satisfont la requête** seront **incluses** dans le résultat de la requête.
- **FROM** est toujours suivi d'une **classe** d'entité (chemin complet)
- L'expression "livre.titre" est juste un "chemin" qui **permet d'atteindre des objets et de propriétés** dans la requête

```
array:6 [▼
  0 => array:2 [▼
    "titre" => "The Aleph"
    "prix" => 20.0
  ]
  1 => array:2 [▼
    "titre" => "1984"
    "prix" => 15.0
  ]
  2 => array:2 [▼
    "titre" => "Life and Fate"
    "prix" => 18.0
  ]
  3 => array:2 [▼
    "titre" => "Vie et destin"
    "prix" => 18.0
  ]
  4 => array:2 [▼
    "titre" => "Nonono"
    "prix" => 40.0
  ]
  5 => array:2 [▼
    "titre" => "Currucucu Paloma"
    "prix" => 20.0
  ]
]
```

20.1.2. Requête qui renvoi un array d'objets

```
// SELECT des Livres complets en DQL,  
// on obtient un array d'objets!  
  
/**  
 * @Route ("/exemples/d/q/l/exemple/select/array/objets")  
 */  
public function exempleSelectArrayObjets (){  
    $em = $this->getDoctrine()->getManager();  
    // avec cette requête on obtient un array d'objets  
    $query = $em->createQuery ('SELECT livre FROM App\Entity\Livre livre WHERE  
livre.prix >15');  
    $resultat = $query->getResult();  
    $vars = ['livres'=> $resultat];  
    return $this->render ("exemples_dql/exemple_select_array_arrays.html.twig",  
$vars);  
}
```

```
ExemplesModeleController.php on line 201:  
array:4 [▼  
    0 => Livre {#615 ▶}  
    1 => Livre {#429 ▶}  
    2 => Livre {#425 ▶}  
    3 => Livre {#421 ▶}  
]
```


20.1.3. Regular Joins et Fetch Joins

Nous pouvons naviguer dans la hiérarchie d'objets de Doctrine tel qu'on l'a fait jusqu'à maintenant...

Exemple : obtenir un entité Livre de la BD et, une fois on l'a dans une variable, obtenir les Exemplaies de ce Livre pour après obtenir les Emprunts.

Tel qu'on a déjà vérifié, Doctrine utilise une technique qui porte le nom de **lazy-loading**. Pour résumer son fonctionnement : **si une entité** (ex. : Livre) **est associée à d'entités d'une autre classe** (ex. : Exemplaies dans Livre), **Doctrine réalisera les requêtes à la BD uniquement quand on accèdera au contenu de ces dernières entités en PHP** (accéder aux exemplaies du Livre pour les afficher, par exemple). Autrement l'objet (ou liste d'objets) contenu dans l'entité (ex. : Exemplaies dans Livre) apparaîtra vide (ou contenant un id, mais jamais complet)

Ce comportement est très logique car si à chaque fois qu'on accède à une entité on doit charger toutes ses entités associés la surcharge du système peut être énorme (ex. : obtenir un Livre et devoir charger tous ses Exemplaies, Emprunts, Clients etc...)

Quand on utilise du DQL contenant de jointures nous allons avoir deux possibilités : faire la requête pour qu'elle utilise le lazy-loading ou forcer la charge des entités associées.

Voyons les deux cas de figure :

Regular Join (collection d'Exemplaies vide) :

```
// Regular Join
/**
 * @Route ("/exemples/d/q/l/exemple/regular/join")
 */
public function exempleRegularJoin(){
    $em = $this->getDoctrine()->getManager();
    $query = $em->createQuery ('SELECT livre FROM App\Entity\Livre livre JOIN
livre.exemplaies exemplaies');

    // observez que les exemplaies sont vides
    $resultat = $query->getResult();
    // observez que les exemplaies sont remplis dans le dump de la vue
    $vars = ['livres'=> $resultat];
    return $this->render ("exemples_dql/exemple_regular_join.html.twig", $vars);
}
```

```

array:6 [▼
  0 => Livre {#486 ▼
    -id: 1
    -titre: "The Aleph"
    -prix: 20.0
    -description: "In Borges' story, the Aleph
    -datePublication: DateTime @-788922000 {#4
    -exemplaires: PersistentCollection {#610 ▼
      -snapshot: []
      -owner: Livre {#486}
      -association: array:15 [ ...15]
      -em: EntityManager {#385 ...11}
      -backRefFieldName: "livre"
      -typeClass: ClassMetadata {#548 ...}
      -isDirty: false
      #collection: ArrayCollection {#410 ▼
        -elements: []
      }
      #initialized: false
    }
  ]
}

```

Fetch Join (collection d'Exemplaires remplie) :

```

// Fetch Join
/**
 * @Route ("/exemples/d/q/l/exemple/fetch/join")
 */
public function exempleFetchJoin(){
    $em = $this->getDoctrine()->getManager();
    // si on indique juste "SELECT livre" on obtient les objets de cette entité
    $query = $em->createQuery ('SELECT livre, exemplaires FROM App\Entity\Livre
livre JOIN livre.exemplaires exemplaires');
    $resultat = $query->getResult();
    // observez que les exemplaires sont remplis dans le dump de la vue
    $vars = ['livres'=> $resultat];
    return $this->render ("exemples_dql/exemple_fetch_join.html.twig", $vars);
}

```

```

array:2 [▼
  0 => Livre {#679 ▼
    -id: 23
    -titre: "Currucucu Paloma"
    -prix: 20.0
    -description: "Roman"
    -datePublication: DateTime @-38710800 {#571 ►}
    -exemplaires: PersistentCollection {#567 ▼
      -snapshot: array:2 [ ...2]
      -owner: Livre {#679}
      -association: array:15 [ ...15]
      -em: EntityManager {#537 ...11}
      -backRefFieldName: "livre"
      -typeClass: ClassMetadata {#680 ...}
      -isDirty: false
      #collection: ArrayCollection {#676 ▼
        -elements: array:2 [▼
          0 => Exempleaire {#643 ►}
          1 => Exempleaire {#667 ►}
        ]
      }
      #initialized: true
    }
  }
  1 => Livre {#658 ►}
]

```

UPDATE

Exemple de UPDATE en DQL : réduire le prix d'un livre

```
// UPDATE
/**
 * @Route ("/exemples/d/q/l/exemple/update")
 */
public function exempleUpdate (){

    $em = $this->getDoctrine()->getManager();
    $query = $em->createQuery ('UPDATE App\Entity\Livre l SET l.prix = l.prix -
:montant WHERE l.titre = :titre');

    // pour simplifier on fixe de valeurs pour le montant à déduire et le livre à
    changer (ISBN)
    $montant = 0.5;
    $ISBN = "The Aleph";

    $query->setParameter ('montant',$montant);
    $query->setParameter ('titre',$ISBN);
    $query->execute(); // pas getResult!
    return $this->render ("exemples_dql/exemple_update.html.twig");

}
```

Important : Les instructions DQL UPDATE sont portées directement dans une simple instruction UPDATE de la BD. Ça implique que les entités qui sont déjà chargées dans le contexte de persistance ne seront PAS synchronisées avec le nouvel état de la base de données mise à jour. Dans certains cas, quand vous utilisez du DQL il est recommandé d'appeler la méthode **clear** du EntityManager pour d'extraire les nouvelles instances de toute entité affectée.

20.3.

Exercices DQL

En utilisant DQL :

- 1) Obtenez les clients
- 2) Obtenez les emprunts (isolés)
- 3) Obtenez les emprunts de tous les clients (seulement le nom du client et les dates de retour)
- 4) Obtenez l'état de tous les emprunts de tous les clients (affichez le nom, prénom du client ainsi que l'état de l'exemplaire)
- 5) Obtenez la liste de livres empruntés par tous les clients : nom du client, prénom du client et titre du livre
- 6) Obtenez la liste des livres empruntés par un client de votre choix : nom du client, prénom du client et titre du livre

- 7) Obtenez la liste de livres qui coutent plus qu'une valeur reçue en paramètre dans l'URL
- 8) Obtenez tous les livres qui contient un texte dans le titre reçu comme paramètre dans l'URL
- 9) Obtenez tous les emprunts réalisés pendant la première quinzaine de février en utilisant DQL. On veut afficher le titre du livre, la date de l'emprunt et le nom et le prénom du client

Note : Les fonctions DAY, MONTH et YEAR ne sont pas acceptées par défaut dans DQL pour pouvoir garder le langage independant du type de BD. Vous allez avoir besoin d'enregistrer de fonctions de date dans symfony en rajoutant des extensions de doctrine. Essayez de suivre par vous-mêmes la documentation !

<https://github.com/oro-inc/doctrine-extensions>

- 10) **Exercez-vous** en réalisant toute sorte de requêtes, essayez les possibilités de Doctrine :

<https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/dql-doctrine-query-language.html>

- 11) **Extra** : Créez de vues pour afficher convenablement tous ces résultats. Vous allez mieux apprendre comment parcourir les structures de données

21. Accès à la BD avec DQL en utilisant les classes Repositoryes

Tel qu'on a déjà mentionné dans la section "Selection", quand on crée une entité sa classe Repository est créée aussi. Cette classe contient les méthodes par défaut qu'on a déjà utilisés (find, findBy, findOneBy, etc...). On va maintenant rajouter **de méthodes faits par nous capables de réaliser de requêtes plus complexes**.

Le but est de simplifier les actions du controller qui, au lieu de devoir contenir la logique de requêtes complexes, appelleront aux actions des repositoryes.

Exemple : Créez une méthode dans la classe Repository de Livre et l'utiliser depuis une action du controller (au lieu d'utiliser DQL depuis le controller lui-même)

1. Créez la méthode du repository pour nous faciliter la tâche

Observez que :

- La méthode renvoie le résultat de la requête, pas de vue bien évidemment
- **Pour obtenir l'Entity Manager dans les classes Repository** on utilise **`$this->getEntityManager()`**. Nous ne sommes pas dans le controller !

```
// obtenir les livres entre deux prix
public function livresEntreDeuxPrixDQL ($pmin, $pmax){
    $em = $this->getEntityManager();
    // avec cette requête on obtient un array
    $query = $em->createQuery ('SELECT livre FROM App\Entity\Livre livre WHERE
livre.prix >= :pmin AND ' .
        'livre.prix <= :pmax');
    $query->setParameter ('pmin', $pmin);
    $query->setParameter ('pmax', $pmax);
    $resultat = $query->getResult();
    // cette méthode renvoie le résultat de la requête
    return $resultat;
}
```

2. Utilisez la méthode depuis le controller

Observez qu'il n'y a pratiquement rien à faire dans l'action...

```
/**
 * @Route
 * ("/exemples/d/q/l/repositories/utilise/repo/livres/entre/deux/prix/{prixMin}/{prixMax}")
 */
function utiliseRepoLivresEntreDeuxPrix (Request $req){

    $prixMin = $req->get("prixMin");
    $prixMax = $req->get("prixMax");

    $em = $this->getDoctrine()->getManager();
    $livresRepo = $em->getRepository(Livre::class);
    $livres = $livresRepo->livresEntreDeuxPrixDQL($prixMin, $prixMax);
    dump ($livres);
    die();

    // return new Response .....
}
```

Exercices :

1. Faites une action où vous créez une adresse et plusieurs clients. Le tout sera stocké dans la BD
2. Rajoutez une adresse à un client existant
3. Créez une méthode qui utilise DQL dans la classe Repository de l'entité Adresse pour vous faciliter la tâche d'obtenir les adresses d'une certaine ville

22. Accès à la BD avec Query Builder

Query Builder est une API qui permet de générer des requêtes de sélection complexes qui renvoient des objets (requêtes de regroupement, jointures, sous-requêtes...) et pas juste des arrays. En fait Query Builder est un générateur de DQL pour faciliter la création de requêtes, mais DQL est préféré.

On pourra réaliser les fonctions de DQL mais en utilisant une notation complètement orientée objet (avec ses avantages et ses inconvénients)

Un Objet **QueryBuilder** est accessible depuis une classe **Repository**.

QueryBuilder fournit les méthodes suivantes qu'on combinera selon nos besoins concrètes :

- join
- innerJoin
- leftJoin
- where
- orWhere
- andWhere
- groupBy
- addGroupBy
- having
- andHaving
- orHaving
- orderBy
- addOrderBy

La documentation de QueryBuilder se trouve ici :

<http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/query-builder.html#>

<https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/query-builder.html>

À continuation on va réaliser un exemple pratique, on commencera par une requête simple.

Commencez par la création d'un controller portant le nom `UtiliseQueryBuilderController`

Exemple : utiliser QueryBuilder pour construire une requête capable d'obtenir le nombre de Livres dont le prix se trouve entre un minimum et un maximum

1. Créez la méthode du repository (LivreRepository.php) capable de réaliser la requête avec Query Builder

```
// QUERYBUILDER: obtenir les livres entre deux prix
// obtenir les livres entre deux prix, version QueryBuilder
public function getEntreDeuxPrix ($min, $max){
    $qb = $this->createQueryBuilder("u"); // u est un nom générique
    $query = $qb->select('u')
        ->where('u.prix >= :min')
        ->andWhere('u.prix <= :max')
        ->setParameter('min', $min)
        ->setParameter('max', $max)
        ->getQuery();
    $res = $query->getResult();
    //var_dump ($res);

    return $res;
}
```

Notez que l'API nous permet de réaliser l'ensemble de la requête sans utiliser ni du SQL pur ni du DQL. Sachez quand-même que QueryBuilder utilise le langage DQL comme langage sous-jacent.

2. Créez une action dans le controller qu'utilise cette méthode et envoyez la réponse au client (new Response) pour qu'il l'affiche

```
/**
 * @Route
 * ("/exemples/query/builder/utilise/repo/livres/entre/deux/prix/{prixMin}/{prixMax}")
 */
function utiliseRepoLivresEntreDeuxPrix (Request $req){

    $prixMin = $req->get("prixMin");
    $prixMax = $req->get("prixMax");

    $em = $this->getDoctrine()->getManager();
    $livresRepo = $em->getRepository(Livre::class);
    $livres = $livresRepo->livresEntreDeuxPrixDQL($prixMin, $prixMax);
    dump ($livres);
    die();

    // return new Response .....
}
```


Exemple : obtenir un Client dont on connaît l'email de la BD avec QueryBuilder

1. Créez des données dans la BD

- Créez une nouvelle Personne dans la BD. Pour la "discrimination-column", tapez "client"
- Créez un nouveau Client qui sera cette Personne (utilisez l'id de la Personne qui vous venez d'encoder)
- Créez la méthode getByEmail dans le repository de l'entité Client (ClientRepository.php) :

2. Créez la méthode du repository capable de réaliser la requête avec Query Builder

```
// QUERYBUILDER: obtenir les clients par mail, version QueryBuilder
public function getParEmail ($email){
    $qb = $this->createQueryBuilder("u");
    $query = $qb->select('u')
        ->where ('u.email = :email')
        ->setParameter('email', $email)
        ->getQuery();
    $resultat = $query->getSingleResult();
    return $resultat;
}
```

- ### 3. Créez une action dans le controller qu'utilise cette méthode et envoyez la réponse au client (new Response) pour qu'il l'affiche

```
/**
 * @Route ("/exemples/query/builder/trouver/client/par/mail/{email}")
 */
public function trouverClientParMail(Request $req){
    $em = $this->getDoctrine()->getManager();
    $rep = $em->getRepository(Client::class);
    // on fait appel à la méthode du Repository
    $objetClient = $rep->getParEmail($req->get ("email"));
    // on affiche les données du Client, on a obtenu un objet
    dump ($objetClient);
    die ();
    // return new Response .....
}
```

Note : vous pouvez toujours afficher le SQL crée par queryBuilder en utilisant de méthodes de cette classe. Par exemple :

```
dd ($repo->createQueryBuilder('g')->getQuery()->getSql())
```

ou

```
$qb = $this->createQueryBuilder("u");
$query = $qb->select('u')
```

```
->where ('u.email = :email')  
->setParameter('email', $email)  
->getQuery();  
dd($query->getSql());
```

23. Formulaires en Symfony

Un formulaire est un ensemble d'éléments HTML dont leur contenu est envoyé au serveur (**action**) en exécutant un **submit**. Le serveur reçoit une requête **POST** qui contient les contenus des éléments du formulaire. En PHP, ces contenus sont accessibles à partir de la variable `$_POST`.

En Symfony nous avons deux options pour créer un formulaire :

1. Créer un formulaire HTML indépendant et obtenir les données dans une action du controller avec l'objet Request
2. Créer un formulaire qui est associé à une classe du modèle. Quand on fait submit on obtient une entité dans le controller

Exemple : Si on crée un formulaire pour insérer un Client, il sera associé à l'entité Client

On va mieux comprendre avec des exemple pratique.

Création d'un formulaire indépendant

23.1.

Vous pouvez créer un formulaire HTML dans votre twig sans aucun problème. Pour obtenir les données du formulaire dans une action vous allez utiliser l'objet **Request**.

Pour accéder au contenu du form vous allez utiliser l'objet Request :

https://symfony.com/doc/current/components/http_foundation.html#accessing-request-data

Vous avez des exemples dans le projet `ProjetFormulairesSymfony5`, controller `ExemplesFormulaireController`.

23.2.

Création une classe de formulaire

Si vous voulez que vos formulaires aient une correspondance directe avec vos entités (ex : un formulaire `Evenement` que dans le controller est directement transformé en objet `Evenement`, sans passer par **query** ni **request**), vous devez créer une classe qui represente ce formulaire.

Si on a une classe formulaire pour une entité, quand on fait submit on obtient une entité dans le controller qu'on pourra, par exemple, insérer directement dans une BD

Nous allons faire un exemple :

Créez d'abord un **nouveau projet** (ex : `projetFormulaires`) contenant un controller (ex : **FormulairesController**). Rajoutez une entité `Aeroport` (nom, code) et créez la BD (ex: `formulairesbd`) et saisissez quelques données.

Exemple : création d'une classe de formulaire

On va créer un formulaire pour l'entité Aeroport qui contiendra deux champs de texte (nom et code). Le bouton de submit sera rajouté à posteriori.

1. Rajoutez les **classes** qui gèrent les **formulaires** en Symfony **dans le projet** (cette action doit se faire une seule fois para projet!)

```
composer require symfony/form
```

2. **Créez le dossier src/Form et un fichier qui contiendra la classe qui définira le formulaire** (pour l'entité Aeroport on crée le fichier AeroportType.php)

Cette définition est une classe, une représentation abstraite de notre formulaire, mais il n'y a pas du HTML à l'intérieur.

Voici le code :

```
<?php

namespace App\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

use Symfony\Component\Form\Extension\Core\Type\TextType;

class AeroportType extends AbstractType {
    public function buildForm (FormBuilderInterface $builder, array $options){
        $builder->add ('nom', TextType::class)
            ->add ('code', TextType::class);
    }
}
```

La classe qui représente le formulaire doit hériter de **AbstractType**, et possède une méthode **buildForm** qui est en charge de générer l'objet formulaire. Cette méthode reçoit un objet qui implémente la classe **FormBuilderInterface** (il est injecté par Symfony, nous ne créons pas cet objet par nous-mêmes), en plus d'un array d'options qui nous permettrait de personnaliser le formulaire.

En général, vous allez utiliser la méthode **add** de cet objet pour rajouter les champs du formulaire. Vous devez, pour chaque champ, indiquer le **name** (premier paramètre) ainsi que le **type**. Symfony possède un vaste nombre de types déjà définis qui correspondent aux type HTML, mais on peut en plus définir nos propres types pour atteindre un niveau de complexité assez élevé.

Voici la liste de types inclus dans Symfony :

<https://symfony.com/doc/current/reference/forms/types.html>

Étudiez par vous-mêmes les types et son fonctionnement, dans ce cours c'est impossible de les parcourir tous vu le temps dont on dispose.

3. Dans une nouvelle action, créez une instance du formulaire en utilisant la méthode **createForm** du controller.

Dans cette action on utilisera la méthode **createForm** pour créer une instance du formulaire (on indique le type qu'on vient de définir). Puis on utilise la méthode **createView** pour générer le code HTML qui sera envoyé à la vue. Voici le code :

```
namespace App\Controller;

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

```
// la classe du Formulaire
use App\Form\AeroportType;

class ExemplesFormulairesController extends AbstractController
{
    /**
     * @Route ("/exemples/formulaires/exemple/aeroport");
     */
    public function exempleAeroport (){
        // on crée le formulaire du type souhaité
        $formulaireAeroport = $this->createForm (AeroportType::class);

        // on envoie un objet FormView à la vue pour qu'elle puisse
        // faire le rendu, pas le formulaire en soi
        $vars = ['unFormulaire'=>$formulaireAeroport->createView()];

        return $this->render
        ('/exemples_formulaires/exemple_aeroport.html.twig',$vars);
    }
}
```

4. Créez la vue et appelez la fonction **form de twig** en lui envoyant l'objet **FormView** qu'on vient de recevoir du controller

Il y a plusieurs manières de rendre le formulaire :

<https://symfony.com/doc/current/forms.html#rendering-the-form>

On peut rendre le formulaire complète ou par parties, en utilisant un thème (Bootstrap, Foundation, etc...) ou pas.

```
{{ form_start (unFormulaire) }}
    {{ form_widget (unFormulaire) }}
{{ form_end (unFormulaire) }}
```

`form_start` et `form_end` génèrent les balises du début et fin du formulaire et `form_widget` génère tous les contrôles.

Création d'un formulaire associé à une entité existante

Vous pouvez créer un formulaire pré-rempli avec les données d'une entité. Pour ce faire, il suffit de créer l'entité avant et l'envoyer comme deuxième paramètre à la méthode **createForm**. Voici un exemple :

23.3.

```
/**
 * @Route ("/exemples/formulaires/exemple/aeroport/rempli");
 */
public function exempleAeroportRempli (){
    $unAeroport = new Aeroport ();
    $unAeroport->setNom("Sevilla Santa Justa");
    $unAeroport->setCode("SVQ");
    // etc....

    // on crée le formulaire du type souhaité
    $formulaireAeroport = $this->createForm (AeroportType::class, $unAeroport);

    // on envoie un objet FormView à la vue pour qu'elle puisse
    // faire le rendu, pas le formulaire en soi
    $vars = ['unFormulaire' => $formulaireAeroport->createView()];

    return $this->render
    ('/exemples_formulaires/exemple_aeroport.html.twig',$vars);
}
```

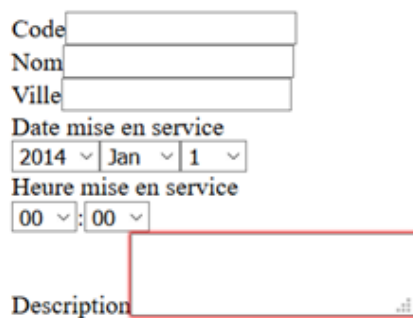
Types des champs du formulaire

On va modifier le formulaire en rajoutant le type de chaque widget. Rajoutez les propriétés **dateMiseEnService**, **heureMiseEnService** et **description** dans l'entité et dans le formulaire (n'oubliez pas les **use**). Faites aussi la migration !

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('code', TextType::class)
        ->add('nom', TextType::class)
        ->add('dateMiseEnService', DateType::class)
        ->add('heureMiseEnService', TimeType::class)
        ->add('description', TextareaType::class)
}
```

Nous n'allons pas rajouter un bouton de submit dans la classe du formulaire **car ce n'est pas une bonne pratique**.

Affichez à nouveau la vue et observez le résultat :



Exercice : créez l'action et la vue nécessaires pour afficher ce formulaire!

Propriétés des champs du formulaire

Chaque type de données correspond à une classe qui hérite de la classe **FormType**. Chaque champ d'un formulaire à un **objet (widget) associé qui générera son code HTML (selon son type)**. Chaque champ a un **ensemble de propriétés HTML héritées de ses parents** (ex: "label" ou "placeholder") **ainsi qu'un ensemble d'options propres** (ex: "preferred_choices" pour le type `LanguageType`).

Exemple : rajout des options dans les champs du formulaire

On va créer un formulaire plus personnalisé que le précédent pour l'entité Livre. Rajoutez les entités Exemple et Livre (vous pouvez les copier d'un autre projet, mais effacez les relations avec les autres entités tel que Catégorie). Pour Livre, rajoutez les champs **nombrePages**, **langue** et **format**

Créez un formulaire basé sur Livre (prenez comme exemple celui de l'entité Aeroport) et **rajoutez les types pour chaque champ** qui puissent vous convenir le plus.

Toutes les informations nécessaires sur les types se trouvent ici :

<https://symfony.com/doc/current/reference/forms/types.html>

Voici le code d'exemple:

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('ISBN', TextType::class)
        ->add('titre', TextareaType::class)
        ->add('prix', MoneyType::class)
        ->add('description', TextareaType::class)
        ->add('datePublication', DateType::class, [
            'label' => 'Date de publication'
        ])
        ->add('nombrePages', IntegerType::class, [
            'label' => 'Nombre de pages',
            'required' => false
        ])
        ->add('langue', LanguageType::class, [
            'label' => 'Langue du livre',
            'preferred_choices' => ['es', 'fr', 'it']
        ])
        ->add('format', ChoiceType::class, [
            'choices' => [
                'eBook' => 'ebook',
                'papier' => 'papier'
            ],
            // les combinaisons de ces paramètres détermineront le type de
            // liste de choix : liste, liste déroulante, checkbox ou
```

```
        // radiobuttons
        'expanded' => false,
        'multiple' => true
    });
}
```

Avant de créer une action pour générer ce formulaire on va rajouter la méthode et l'action dans la section suivante.

Méthode et Action

Pour finir le formulaire, on peut spécifier l'action à réaliser pour le submit (même avant créer le bouton) ainsi que la méthode (GET ou POST). Nous avons deux possibilités :

- 1) Définir l'action **dans la classe du formulaire**. C'est facile mais on ne pourra utiliser le formulaire pour exécuter d'autres actions !

Important : Cette méthode est à éviter mais elle facilite la compréhension des bonnes pratiques

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('ISBN', TextType::class)
        ->add('titre', TextareaType::class)
        ->add('prix', MoneyType::class)
        ->add('description', TextareaType::class)
        ->add('datePublication', DateType::class, [
            'label' => 'Date de publication'
        ])
        ->add('nombrePages', IntegerType::class, [
            'label' => 'Nombre de pages',
            'required' => false
        ])
        ->add('langue', LanguageType::class, [
            'label' => 'Langue du livre',
            'preferred_choices' => ['es', 'fr', 'it']
        ])
        ->add('format', ChoiceType::class, [
            'choices' => [
                'eBook' => 'ebook',
                'papier' => 'papier'
            ],
            // les combinaisons de ces paramètres détermineront le type de
            // liste de choix : liste, liste déroulante, checkbox ou
            // radiobuttons
            'expanded' => false,
            'multiple' => false
        ])
        ->setMethod('POST')
        ->setAction('traitementFormulaireLivre');
}
```

- 2) Définir l'action et la méthode dans le controller lors de la création de l'objet formulaire. Cette option est **plus souple** car elle nous permet de réutiliser le formulaire pour lancer d'autres actions :

Dans la classe du formulaire il n'y a pas d'action ni de méthode :

```
class LivreType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('ISBN', TextType::class)
            ->add('titre', TextareaType::class)
            ->add('prix', MoneyType::class)
            ->add('description', TextareaType::class)
            ->add('datePublication', DateType::class, [
                'label' => 'Date de publication'
            ])
            ->add('nombrePages', IntegerType::class, [
                'label' => 'Nombre de pages',
                'required' => false
            ])
            ->add('langue', LanguageType::class, [
                'label' => 'Langue du livre',
                'preferred_choices' => ['es', 'fr', 'it']
            ])
            ->add('format', ChoiceType::class, [
                'choices' => [
                    'eBook' => 'ebook',
                    'papier' => 'papier'
                ],
                'expanded' => false,
                'multiple' => true
            ])
    }
}
```

```
/**
 * @Route ("/exemples/formulaires/exemple/livre");
 */
public function exempleLivre (){
    $livre = new Livre();
    $formulaireLivre = $this->createForm (LivreType::class, $livre, array(
        'action' => $this->generateUrl("rajouter_livre"), // name de la route!
        // si on n'utilise pas le name d'une route
        // 'action' => "/exemples/formulaires/livre/rajouter",
        'method' => 'POST'
    ));
    $vars = ['unFormulaire'=>$formulaireLivre->createView()];

    return $this->render ('/exemples_formulaires/exemple_livre.html.twig', $vars);
}
```

Nous utiliserons un array de paramètres et la méthode **generateUrl** pour générer le code HTML qui correspond à une route qui porte un "name". Si la route n'a pas de "name" on peut juste mettre un path, mais c'est moins souple car la modification d'un path dans le routing impliquera modifier une par une toutes les appels à cette action.

Voici le code complet du controller, ici on a une route avec "name" (rajouter_livre) :

```
<?php

namespace App\Controller;

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

// les entités de base
use App\Entity\Aeroport;
use App\Entity\Livre;
// les classes des Formulaires
use App\Form\AeroportType;
use App\Form\LivreType;

class ExemplesFormulairesController extends AbstractController
{
    /**
     * @Route ("/exemples/formulaires/exemple/aeroport");
     */
    public function exempleAeroport (){
        // on crée le formulaire du type souhaité
        $formulaireAeroport = $this->createForm (AeroportType::class);

        // on envoie un objet FormView à la vue pour qu'elle puisse
        // faire le rendu, pas le formulaire en soi
        $vars = ['unFormulaire'=>$formulaireAeroport->createView()];

        return $this->render
        ('/exemples_formulaires/exemple_aeroport.html.twig',$vars);
    }

    /**
     * @Route ("/exemples/formulaires/exemple/livre");
     */
    public function exempleLivre (){
        $livre = new Livre();
        $formulaireLivre = $this->createForm (LivreType::class, $livre, array(
            'action' => $this->generateUrl("rajouter_livre"), // name de la route!
            // si on n'utilise pas le name d'une route
            // 'action' => "/exemples/formulaires/livre/rajouter",
            'method' => 'POST'
        ));
        $vars = ['unFormulaire'=>$formulaireLivre->createView()];
    }
}
```

```

        return $this->render
        ('/exemples_formulaires/exemple_livre.html.twig', $vars);
    }

}

```

Boutons dans les formulaires (bonnes pratiques)

Si on veut réutiliser un même formulaire pour réaliser plusieurs actions (ex : mettre à jour un livre ou rajouter un livre) on ne doit pas créer les boutons dans la classe du formulaire mais dans la vue correspondante. Si on le crée dans la classe on sera coincés car l'étiquette du bouton sera fixée (on casse le principe de réutilisation du code!).

On ne doit pas non plus rajouter le bouton dans le controller car on serait en train de mélanger présentation (ex : la classe du bouton) avec la logique (on casse le principe de "separation of concerns!").

La **meilleure option est de créer le bouton de submit en HTML dans la vue**. Voici un exemple :

```

{{ form_start (formulaireLivre) }}
{{ form_widget (formulaireLivre) }}
<input type="submit" class="btn" value="Envoyer" />
{{ form_end (formulaireLivre) }}

```

Cette méthode nous permet de réutiliser le formulaire pour plein d'actions, on devra juste créer les boutons dans chaque vue.

Exercice : Créez des boutons de submit dans les vues qui rendent les formulaires des exemples précédents

Rendu du formulaire dans la vue

Au moment de générer un formulaire dans un fichier twig on peut utiliser :

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

Pour rendre la totalité du formulaire d'un coup. Mais on peut contrôler plus la génération du formulaire en utilisant :

```
{{ form_start (nomDuFormulaire) }}
```

Rend la balise de début du formulaire, y compris l'attribut enctype correct lors de l'utilisation des téléchargements de fichiers.

```
{{ form_widget (nomChampFormulaire) }}
```

Rend un champ, ce qui inclut l'élément du champ lui-même, une étiquette et tous les messages d'erreur de validation pour le champ (si on a défini des validations)

```
{{ form_end (nomDuFormulaire) }}
```

Rend l'étiquette de fin du formulaire et tous les champs qui n'ont pas encore été rendus, dans le cas où vous avez rendu chaque champ vous-même. Ceci est utile pour ne pas devoir rendre à la main les champs cachés.

Exemple :

```
{{ form_start (formulaireAeroport) }}
    {{ form_widget (formulaireAeroport.nom) }}
    {{ form_widget (formulaireAeroport.description) }}
{{ form_end (formulaireAeroport) }}
```

Si vous ne voulez pas afficher un champ d'un formulaire c'est simple : effacez la ligne **form_widget** qui correspond à ce champ

```
{{ form_end(form, {'render_rest': false}) }}
```

. Le contrôleur recevra alors une valeur **null** pour ce champ de l'entité associée.

Par défaut Symfony rend les champs qui ne sont pas spécifiés. Pour éviter le rendu automatique du reste des champs il faut rajouter :

Résumé : création et personnalisation de base d'un formulaire

Pour créer un formulaire et le traiter :

1. Créez le **squelette** du formulaire (la **classe**)
2. Dans cette classe, **rajoutez les champs ("widgets") et leurs types** selon vos besoins (TextType, 23.3. ChoiceType, etc...). Personnalisez le widget avec des **propriétés** (taille, required, etc...)
3. **Rajouter les boutons de submit dans la vue qui affiche ce formulaire**
4. **Définissez le nom de l'action qui affichera et traitera ce formulaire, ainsi que la méthode (GET, POST)**
5. Créez une **action qui génère et traite le formulaire**

Traitement des données du formulaire (explication de base)

Pour **recevoir et traiter** les données introduites dans un formulaire nous devons créer une action dans un controller. L'action traitera la requête (reçoit un objet **Request**).

Voici un exemple complet et son explication.

23.4.

Exemple : Rendu et réception d'un formulaire

Cette action peut être appelé dans deux cas de figure :

- a) On appelle l'action une première fois, le formulaire doit être rendu (affiché dans la vue)
- b) On appelle l'action en cliquant sur un bouton submit et on doit traiter le contenu reçu (dans ce cas on va juste afficher un objet créé à partir du contenu du formulaire)

Dans les deux cas de figure on crée une entité et un objet formulaire. Pour savoir si on se trouve dans le cas a) ou b) on doit **vérifier l'objet Request**. Dans le cas a) l'objet sera vide car on n'a pas fait submit et dans le cas b) l'objet Request contiendra les données du formulaire, on doit prendre ces données et remplir l'entité vide qu'on a créé tout au début pour, par exemple, faire un CRUD

Pour analyser l'objet Request on utilise la méthode **handleRequest**, auquel on envoie l'objet Request injecté dans l'action. Les méthodes **isSubmitted** et **isValid** nous indiqueront si le formulaire a été submitted (on a cliqué sur le bouton) et si les données du formulaire sont valides (en principe on recevra toujours **true** car on n'a fait aucune règle de validation).

Il faut remarquer qu'on aura normalement au moins **deux templates** : un pour le rendu du formulaire et l'autre pour afficher le résultat du traitement du formulaire.

Voici le code qui implémente ce qu'on vient de décrire : le controller, les templates et la classe du formulaire.

1. Controller :

```
/**
 * @Route ("/exemples/formulaires/traitement/exemple/livre",name =
 "exemple_livre");
 */

// dans la même action on réalise le rendu et la réception
public function exempleLivre (Request $req){
    // 1. Création d'une entité vide
    $livre = new Livre();
    // 2. Création du formulaire du type souhaité
    $formulaireLivre = $this->createForm (LivreType::class, $livre,
        ['action'=> $this->generateUrl ("exemple_livre"),
        'method'=>'POST']);
}
```

```

// 3. Analyse de l'objet Request
$formulaireLivre->handleRequest($req);

// 4. Vérification: on vient d'un submit ou pas?

// si oui, on traite le formulaire et on remplit l'entité
if ($formulaireLivre->isSubmitted() && $formulaireLivre->isValid()){
    // l'entité sera remplie avec les données du formulaire pendant le submit,
    pas besoin de getData

    // Rendu d'une vue où on affiche les données
    // Normalement on faire CRUD ici ou une autre opération...
    return $this->render
('/exemples_formulaires_traitement/traitement_formulaire_livre.html.twig',
    ['livre'=> $livre]);
}
// si non, on doit juste faire le rendu du formulaire
else {
    return $this->render
('/exemples_formulaires_traitement/affichage_formulaire_livre.html.twig',
    ['formulaireLivre'=> $formulaireLivre->createView()]);
}
}

```

2. Templates (un pour afficher le formulaire et l'autre pour afficher le résultat du traitement)

```

{# affichage_formulaire_livre.html.twig #}
{{ form_start (formulaireLivre) }}
{{ form_widget (formulaireLivre) }}
<input type="submit" class="btn" value="Envoyer" />
{{ form_end (formulaireLivre) }}

{# traitement_formulaire_livre.html.twig #}
{{ dump (livre) }}

```

3. Classe du formulaire

```

class LivreType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('ISBN', TextType::class)
            ->add('titre', TextareaType::class)
            ->add('prix', MoneyType::class)
            ->add('description', TextareaType::class)
            ->add('datePublication', DateType::class, [
                'label' => 'Date de publication'
            ])
    }
}

```

```

        ->add('nombrePages', IntegerType::class, [
            'label' => 'Nombre de pages',
            'required' => false
        ])
        ->add('langue', LanguageType::class, [
            'label' => 'Langue du livre',
            'preferred_choices' => ['es', 'fr', 'it']
        ])
        ->add('format', ChoiceType::class, [
            'choices' => [
                'eBook' => 'ebook',
                'papier' => 'papier'
            ],
            // les combinaisons de ces paramètres détermineront
            // le type de
            // liste de choix : liste, liste déroulante, checkbox ou
            // radiobuttons
            'expanded' => false,
            'multiple' => true
        ]);
        //->add('Envoyer', SubmitType::class);
    }
}

```

Résumé : Traitement de données d'un formulaire dans une action d'un controller

1. Créez une **entité vide**
2. Créez une **instance du formulaire**
3. Faites appel à **handleRequest pour traiter la requête et faire submit** du Form. HandleRequest fait appel au submit si on a fait submit et l'entité « data » qu'on a envoyé dans la création du formulaire sera remplie. On peut aussi obtenir les données du form avec **getData**.
On utilisera getData pour les champs du formulaire qui n'existent pas dans l'entité (et dans le traitement des champs de certains types particuliers).

Ex : formulaire pour Aeroport (nom, code) qui contient aussi un champ « description ». Pour obtenir les données de la description dans le traitement du formulaire on utilise `getData` :

```
$form->get('description')->getData();
```

Pour le reste on utilise un objet, l'entité Aeroport (`$aeroport`) qui sera remplie automatiquement dans le submit

Une fois l'entité a été créé à partir des données on peut faire n'importe quelle action (CRUD ou une autre).

Si on arrive à l'action sans avoir fait un submit (exemple : tapez l'URL de l'action dans le navigateur) ou les données ne sont pas valides (`isSubmitted` ou `isValid` renvoient faux), on doit juste envoyer le formulaire à la vue pour réaliser le rendu.

Exercices : traitez toutes les données du formulaire que vous avez créés dans l'exercice précédent

Bonnes pratiques pour créer de formulaires en Symfony

1. **Ne rajoutez de boutons aux formulaires dans les classes des formulaires ni dans les controllers,** mais dans les templates.

23.5.

Exemple : Si vous créez un formulaire pour insérer un client dans la BD et vous créez un bouton "insérer" dans la classe du formulaire, ce formulaire ne pourra plus être utilisé pour par exemple mettre à jour le client... bien qu'il s'agit du même formulaire pour les deux actions ! Rajouter les boutons dans les controllers est aussi une mauvaise idée car vous allez mélanger logique et présentation ("vues"). Il nous reste alors que les rajouter dans les fichiers twigs (en HTML)

2. Utilisez **une même action pour créer le formulaire et le traiter**
3. Pour définir l'action et la méthode (différente, par exemple, pour un update et un delete), vous pouvez envoyer de paramètres à **form_start** dans le fichier twig

```
{{ form_start(form, {'action': path('rajouter'), 'method': 'POST'}) }}
```

Attention : Le **path** sera le **name** d'une route

Style des formulaires

Vous pouvez appliquer du style aux formulaires en utilisant du CSS. Symfony inclut plusieurs templates. Plus d'information ici :

23.6. https://symfony.com/doc/current/form/form_themes.html#symfony-builtin-forms

Formulaires concernant plusieurs entités

Considérons que les Genres sont aussi des entités de la BD (un Genre a un nom et une description). Comment faire si on veut créer un formulaire pour insérer un livre et choisir au même temps son genre dans le formulaire ? Le genre est un objet (entité) !

La solution est **d'utiliser le type EntityType**, qui nous permettra **d'envoyer une entité** de notre choix **dans le formulaire**.

<https://symfony.com/doc/current/reference/forms/types/entity.html>

Créez une nouvelle entité Genre (contenant **nom** et **description**) et une **association d'un a plusieurs avec Livre** (on va considérer qu'un livre à juste un genre). **Ici on veut pouvoir envoyer une entité Genre pour pouvoir l'incruster dans l'entité Livre.**

Créons une classe de formulaire **LivreGenreType** à partir de **LivreType** et faisons les modifications nécessaires (explication après le code) :

```
class LivreGenreType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('ISBN', TextType::class)
            ->add('titre', TextareaType::class)
            ->add('prix', MoneyType::class)
            ->add('description', TextareaType::class)
            ->add('datePublication', DateType::class, [
                'label' => 'Date de publication'
            ])
            ->add('nombrePages', IntegerType::class, [
                'label' => 'Nombre de pages',
                'required' => false
            ])
            ->add('langue', LanguageType::class, [
                'label' => 'Langue du livre',
                'preferred_choices' => ['es', 'fr', 'it']
            ])
            ->add('format', ChoiceType::class, [
                'choices' => [
                    'eBook' => 'ebook',
                    'papier' => 'papier'
                ],
                'expanded' => false,
                'multiple' => true
            ])
            ->add('genre', EntityType::class, [
                'class' => Genre::class,
                'query_builder' => function (GenreRepository $er){
                    return
                        $er->createQueryBuilder('u')->orderBy ('u.nom', 'DESC');
                },
                // on affiche ici les noms et les descriptions en majuscules,
                // mais c'est à vous de choisir la façon de l'afficher
                'choice_label' => function ($x){
```

```

        return strtoupper($x->getNom() . "-" . $x-
>getDescription());
    }
    });
}
}

```

Explication :

1. On rajoute un champ du type `EntityType` qui portera le nom du champ de l'association
2. On spécifie la classe de cet Entité (ici, "Genre")
3. Dans la clé **query_builder**, on crée une fonction qui, en utilisant un `QueryBuilder`, **renvoie** un ensemble d'objets "Genre" (voir la section consacrée au `QueryBuilder`)
4. Chaque objet contenu dans la requête sera passé à fonction spécifiée dans **choice_label**. Le contenu renvoyé par cette fonction s'affichera comme option dans la liste déroulante.

Observez le code de **ExemplesFormulairesObjetsController**.

Voici le code des **templates** :

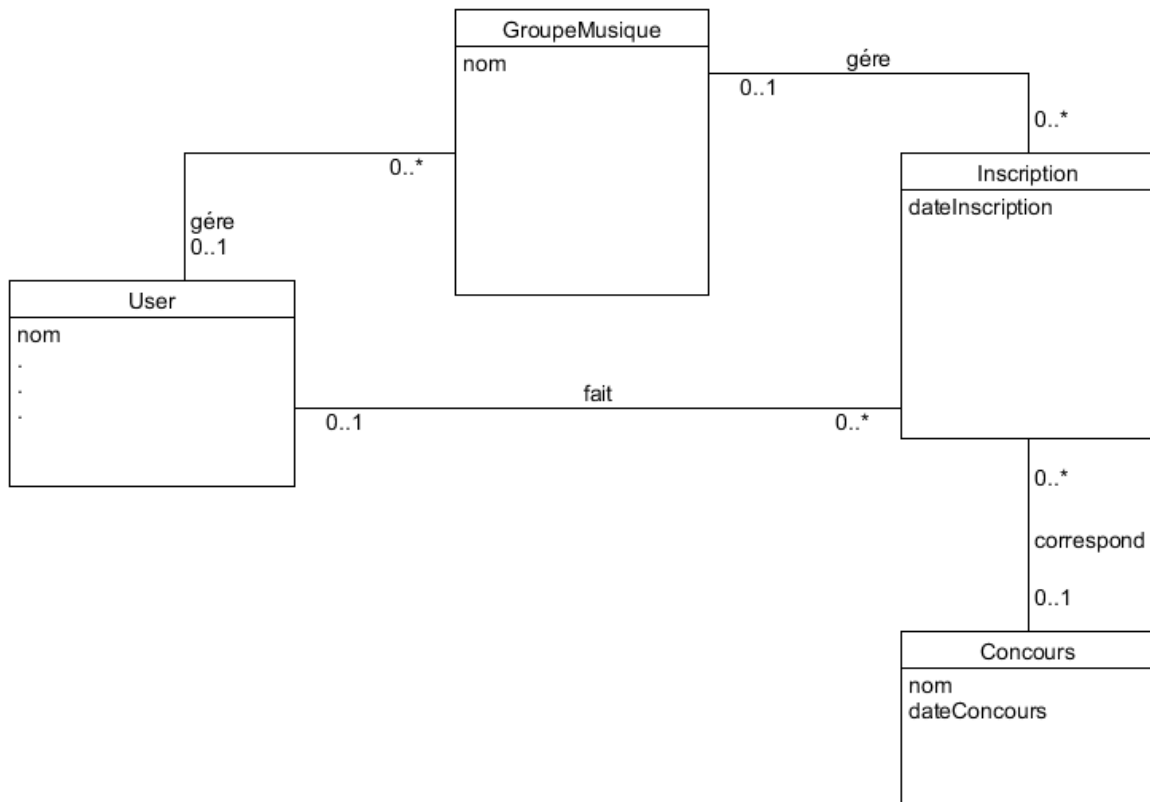
```

{# affichage_formulaire_livre.html.twig #}
{{ form_start (formulaireLivre) }}
{{ form_widget (formulaireLivre) }}
<input type="submit" class="btn" value="Envoyer" />
{{ form_end (formulaireLivre) }}
{# traitement_formulaire_livre.html.twig #}
{{ dump (livre) }}

```

Formulaire contenant une liste déroulante d'entités filtrés

Considérez ce modèle :



Un **User** inscrit à un de ses **GroupeMusique** à un concours.

Quand on affiche le formulaire **d'Inscription** on veut pouvoir choisir le **GroupeMusique** à inscrire dans le **Concours**, mais on veut avoir uniquement les groupes qui ont été créés par cet User. On doit créer la liste de **GroupeMusique** en filtrant par **User**.

On se trouve dans une situation similaire à celle de l'exemple précédant, mais la requête qui renvoie les entités de la liste (avant Genre et maintenant GroupeMusique) doit filtrer par User. **Mais on ne peut pas obtenir l'User dans le code du formulaire!**

Nous avons deux solutions :

- Envoyer l'User comme option (array associatif) pendant la création du formulaire (méthode **createForm**) quand on crée le formulaire dans le Controller. Cet array **\$options** sera disponible dans la méthode **buildForm** de la classe formulaire.

- b) Enregistrer le formulaire comme Service dans **services.yaml**. Créer un paramètre contenant le token de l'User et l'envoyer lors de la création du Formulaire

Cette solution est expliquée ici :

<https://stackoverflow.com/questions/38199882/filter-entitytype-by-owner-current-user>

Dans les deux cas, il faut adapter la requête (QueryBuilder) dans la création de la liste.

Réalisons la première méthode (envoyer l'User dans la création du form) :

- 1) D'abord on crée une fixture capable de créer des users et de groupes et de les lier (**RajouterGroupesUsers**) :

```
<?php

namespace App\DataFixtures;

use App\Entity\User;

use App\Entity\GroupeMusique;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

class RajouterGroupesUsers extends Fixture
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }

    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 3 ; $i++){

            $user = new User();
            $user->setNom("autre user" . $i);
            $user->setEmail("user".$i."liegroupe@gmx.com");
            $user->setPassword($this->passwordEncoder->encodePassword(
                $user,
                'unpass'
```

```

    ));
    $manager->persist ($user);

    for ($j = 0; $j < 5; $j++){
        $groupeMusique = new GroupeMusique();
        $groupeMusique->setNom("le groupe ".$j." de ".$user->getNom());

        $user->addGroupeGere($groupeMusique);
        $manager->persist ($groupeMusique);
    }
    // dans ce cas on aurait pu faire aussi $groupeMusique-
    >setUser($user);
    }
    $manager->flush();
}
}

```

2) Créez l'action du controller, qui envoie l'user dans la création du form

```

/**
 * @Route("/exemple/filtre/form/user", name="exemple_filtre_form_user")
 */
public function exempleFiltreFormUser(Request $request)
{
    // On utilise l'array options pour envoyer l'User
    $inscription = new Inscription();
    // La date est créé dans le constructeur de l'entité

    $form = $this->
        createForm(InscriptionFiltreType::class, $inscription,
            ['user'=>$this->getUser()]);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($inscription);
        $entityManager->flush();
        return $this
            ->render('exemple_filtre_form_user/enregistrement_succes.html.twig');
    }
    return $this
        ->render('exemple_filtre_form_user/affichage.html.twig',
            ['form'=> $form->createView()]);
}

```

3) Créez votre form (InscriptionFiltreType) :

```
<?php

namespace App\Form;

use App\Entity\Concours;
use App\Entity\Inscription;
use App\Entity\GroupeMusique;
use App\Repository\ConcoursRepository;
use Symfony\Component\Form\AbstractType;
use App\Repository\GroupeMusiqueRepository;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;

class InscriptionFiltreType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        //dd($options['user']);
        $builder
            ->add('dateInscription', DateType::class, [
                'required' => false,
            ])
            ->add('groupeMusique', EntityType::class, [
                'class' => GroupeMusique::class,
                'query_builder' => function (GroupeMusiqueRepository $repo) use ($options) {
                    // dd ($options['user']); // attention au "use" car fonction anonyme
                    // afficher le SQL
                    // dd ($repo->createQueryBuilder('g')->getQuery()->getSql());
                    return $repo->createQueryBuilder('g')
                        ->select ('g')
                        ->innerJoin('g.user', 'u', 'WITH', 'u.id=:idUser')
                        ->setParameter ('idUser', $options['user']);
                    // doc exemples join : https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/query-builder.html
                },
                'choice_label' => function ($groupeMusique) {
```

```

        return $groupeMusique->getNom();
    },

    ])
->add('concours', EntityType::class, [
    'class' => Concours::class,
    'query_builder' => function (ConcoursRepository $repo) {

        return $repo->createQueryBuilder('c');
    },
    'choice_label' => function ($concours) {
        return $concours->getNom();
    },

    ]));
}

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults([
        'user' => null, // valeur de l'option par défaut, car c'est optionnel
        'data_class' => Inscription::class,
    ]);
}
}

```

Pour le tester, lancez les fixtures et **faites login** avec un parmi les users qui se trouvent dans la fixture **RajouterGroupesUsers.php** . Puis lancez l'action **exempleFiltreFormUser** du controller **ExempleFiltreFormUser** (tapez-la dans l'url). La liste de groupes doit contenir uniquement les groupes auxquels l'user qui vient de faire login appartient.

24. Upload de fichiers en utilisant un formulaire

Dans cette section on propose une méthode pour pouvoir faire upload de fichiers du client au serveur en utilisant un formulaire créé par Symfony.

La documentation pour ce faire se trouve ici :

https://symfony.com/doc/current/controller/upload_file.html

Mais nous allons développer nos propres exemples.

Stockage dans le serveur d'une seule image pour chaque entité

24.1. Objectif : Pouvoir faire upload d'une image pour chaque entité dans la BD.

On va créer une entité (Pays) et un formulaire qui nous permettra de faire upload d'une image associée à cette entité (une image pour chaque pays). Notre action stockera le nom du pays et le lien vers l'image dans la BD, ainsi que le fichier en soi dans un dossier du serveur.

Procédure :

1. **Créez l'entité** (Pays, contenant le **nom** du pays et un champ **lienImage** pour stocker le lien de l'image. Les deux sont du type string)

Important: effacez la spécification des types (paramètres et retour) dans les méthodes set et get de lienImage

2. Générez les entités et mettez à jour le schéma de la BD
3. **Créez la classe du formulaire** pour cette entité (PaysType.php). Pour le champ **lienImage**, choisissez **FileType**, et rajoutez un bouton de submit.


```

namespace App\Form;

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

class PaysType extends AbstractType
{
    public function buildForm (FormBuilderInterface $builder, array $options)
    {
        $builder->add('nom', TextType::class)
            ->add('lienImage', FileType::class, array ('label'=>"Sélectionner
l'image du pays"));
    }
}

```

4. Créez un fichier twig capable d'afficher ce formulaire

```

{# affichage_formulaire upload.html.twig #}
{{ form_start (formulaire) }}
{{ form_widget (formulaire) }}
<input type="submit" class="btn" value="Envoyer" />
{{ form_end (formulaire) }}

```

5. Créez une action qui traite les données envoyées par le formulaire

Cette action doit :

- **Créer un objet formulaire** (PaysType) **associé à une entité vide** (\$pays de la classe Pays)
- **Gérer la requête** : HandleRequest remplira les propriétés de l'entité (hydrate)
- **Vérifier que le formulaire a été envoyé** (isSubmitted) **et si les données sont valables** (isValid).
- **Obtenir le fichier** (objet UploadedFile, pas un string) **de l'entité** associée au formulaire
 - o **Obtenir un nom de fichier unique** pour le stocker dans le serveur (si on utilise le nom original il pourrait y avoir plein de doublons !)
 - o **Stocker le fichier dans le serveur**
- **Affecter la propriété contenant le fichier dans l'entité et lui donner le nom unique qu'on vient d'obtenir**
- **Stocker l'objet dans la BD**

Voici le code de l'action :

```

namespace App\Controller;

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

```

```

use Symfony\Component\HttpFoundation\Request;
use App\Entity\Pays;
use App\Form\PaysType;
use Symfony\Component\HttpFoundation\Response;

class ExemplesFormulaireUploadController extends AbstractController
{
    /**
     * @Route ("/exemples/formulaire/upload/exemple");
     */
    public function exemple (Request $request){
        // créer une nouvelle entité vide
        $pays = new Pays();
        // créer un formulaire associé à cette entité
        $formulairePays = $this->createForm (PaysType::class, $pays);
        // gérer la requête (et hydrater l'entité)
        $formulairePays->handleRequest($request);
        // vérifier que le formulaire a été envoyé (isSubmitted)
        // et que les données sont valides
        if ($formulairePays->isSubmitted() && $formulairePays->isValid()){
            // obtenir le fichier (pas un "string" mais un
            // objet de la class UploadedFile)
            $fichier = $pays->getLienImage();
            // obtenir un nom de fichier unique
            // pour éviter les doublons dans le dossier
            $nomFichierServeur = md5(uniqid()).".".$fichier->guessExtension();
            // stocker le fichier dans le serveur (on peut indiquer un dossier)
            $fichier->move ("dossierFichiers", $nomFichierServeur);
            // affecter le nom du fichier de l'entité. Ça sera le nom qu'on
            // aura dans la BD (un string, pas un objet UploadedFile cette fois)
            $pays->setLienImage($nomFichierServeur);

            // stocker l'objet dans la BD, ou faire update
            $em = $this->getDoctrine()->getManager();
            $em->persist($pays);
            $em->flush();
            return new Response ("fichier uploaded et BD mise à jour!");
        }
        else {
            return $this->render
            ("/exemples_formulaires_upload/affichage.html.twig",
                ['formulaire'=> $formulairePays->createView()]);
        }
    }
}

```

Problèmes dans l'upload

Nous pouvons avoir de problèmes liés à certaines limites concernant la taille des fichiers qu'on peut charger dans le serveur.

24.2.

1. Dans **php.ini**, **upload_max_filesize** spécifie la taille maximale acceptée par le module de php

```
; Maximum allowed size for uploaded files.  
; http://php.net/upload-max-filesize  
upload_max_filesize=20M
```

2. Dans **php.ini**, **post_max_size** indique la taille maximale d'un formulaire envoyé en POST (avec ou sans le champ d'upload)

```
; Maximum size of POST data that PHP will accept.  
; Its value may be 0 to disable the limit. It is ignored if POST data reading  
; is disabled through enable_post_data_reading.  
; http://php.net/post-max-size  
post_max_size=20M
```

Notez que, en ce qui concerne l'upload d'un fichier, ça ne vous sert à rien de changer le premier paramètre sans changer le deuxième car il faut que le serveur admette un post contenant un fichier d'au moins la taille permise par **upload_max_filesize**.

Si on a un formulaire avec un champ d'upload, la taille du POST sera, en gros, celle du fichier envoyé plus celle de tous les autres champs du formulaire.

Après avoir augmenté la valeur de ces deux paramètres on ne doit plus avoir de problèmes, mais si ce n'est pas le cas il faut considérer aussi les paramètres suivants :

3. Dans certains cas il faut considérer aussi la limite pour la taille du fichier **.php** qu'on peut charger (en **php.ini**)

```
; Maximum amount of memory a script may consume (128MB)  
; http://php.net/memory-limit  
memory_limit=128M
```

4. Il peut avoir aussi un problème si la connexion du client est lente et l'upload prend plus du temps spécifié dans **max_input_time (php.ini)**. Ce paramètre indique le temps maximum permis pour analyser les données du POST ou GET: c'est le temps qui passe entre l'appel au script PHP et le début de son exécution. Dans la configuration de XAMPP la valeur est -1, il n'y a pas de limite de temps.

25. AJAX en Symfony avec Axios

Objectif : utiliser AJAX dans un template Twig avec Axios

Axios est une librairie que nous simplifie les appels AJAX. Vous pouvez parfaitement faire du AJAX sans cette librairie mais ici on l'utilise pour nous faciliter la tâche.

Créez un contrôleur **ExemplesAjaxFormDataController** (code original dans le projet **projetFormulaires**). Ce contrôleur contiendra uniquement quelques exemples d'appel Ajax. Plus tard on réalisera des exemples plus pratiques basés sur la BD du projet.

Exemple d'appel AJAX avec un formulaire

25.1. Dans cet exemple on envoie de données en utilisant AJAX **sans utiliser un formulaire**. Nous avons juste les contrôles. Dans la section suivante on utilisera un formulaire complet.

1. Créez une vue contenant un formulaire. Cette vue contiendra aussi le code AJAX

Exemple : créez un formulaire contenant un input (nom). Quand on clique sur le bouton, un message de bienvenue sera affiché dans le div. Analysez vous-même le code.

Attention aux **names** des contrôles car on les utilisera dans le traitement de l'action dans le contrôleur!!

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

{% block body %}
<!-- formulaire à envoyer -->
<form id="leFormulaire">
    <input type="text" name="nom" />
    <input type="submit" value="Envoyer" />
</form>
<div id="divMessage"></div>
{% endblock %}

{% block javascripts %}
<!-- AJAX - AXIOS -->
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    envoyerNom.addEventListener("click", function (event) {
        event.preventDefault();

        console.log (document.getElementById("leFormulaire"));
```

```

    axios({
        url: '{{path ("exemple1_traitement')}}',
        method: 'POST',
        headers: { 'Content-Type': 'multipart/form-data' },
        data: new FormData(document.getElementById("leFormulaire"))
    })
    .then(function (response) {
        // response.data est un objet qui correspond à l'array associatif envoyé dans le controller
        // JsonResponse a transformé l'array en JSON. Axios transforme le JSON en objet JS
        // (et on utilise ici la clé "leMessage")
        document.getElementById("divMessage").innerHTML = response.data.leMessage;
    })
    .catch(function (error) {
        console.log(error);
    });
});
</script>
{% endblock %}

```

Dans l'appel AXIOS on envoie un objet JS contenant :

- Le **nom de l'action** qui traitera les données envoyées
- La **méthode** (POST)
- Les « **headers** » de la requête, pour indiquer qu'on envoie un formulaire (dans ce cas)
- Les **données (data)** : un objet JS contenant des clés et des valeurs. Ici on envoie un objet FormData (classe de JS) construit à partir du formulaire qui se trouve dans la page web

Créez l'action qui affiche la vue exemple1_affichage.html

```

/**
 * @Route ("/exemples/ajax/axios/exemple1/affichage" );
 */
public function exemple1Affichage()
{
    return $this->render("/exemples_ajax_axios/exemple1_affichage.html.twig");
}

```

2. Créez l'action qui traite la pétition AJAX

```
/**
 * @Route ("/exemples/ajax/axios/exemple1/traitement",name="exemple1_traitement" );
 */
// action qui traite la commande AJAX, elle n'a pas une vue associée
public function exemple1Traitement(Request $requeteAjax)
{

    $valeurNom = $requeteAjax->get('nom');
    $arrayReponse = [ 'leMessage' => 'Bienvenu, ' . $valeurNom];
    return new JsonResponse($arrayReponse);
}
```

Cette action reçoit un objet Request. On peut accéder aux éléments du formulaire en utilisant get. Dans cet exemple, l'action renvoie un array à traiter dans le code JS. Pour envoyer des arrays ou des objets à JS depuis PHP on doit les transformer en **JSON**. On verra d'autres exemples (envoyer des objets) par la suite.

Utilisation de blocs dans twig avec AJAX

25.2.

Il s'agit juste d'une combinaison de master page + AJAX, rien de nouveau.

1. Créez un template master_page.html.twig contenant une section pour nos vues. Créez un block pour le contenu et un autre pour le JS

```
<html>
  <body>
    <header>
      Voici la section header
    </header>
    <main>
      Voici la section main
      {% block contenuMain %}{% endblock %}
    </main>
    <footer>
      Voici la section footer
    </footer>
  </body>
  {% block javascripts %}{% endblock %}
</html>
```

2. Créez une vue exemple1_affichage_master_page.html.twig qui hérite du template master_page.html.twig

```
{% extends '/exemples_ajax/master_page.html.twig' %}

{% block contenuMain %}
<!-- on mettra cet script dans un block -->

<!-- formulaire à envoyer -->
<form id="leFormulaire" method="POST">
    <input type="text" name="nom" />
    <input type="submit" id="envoyerNom" value="Envoyer" />
    <div id="divMessage"></div>
</form>
{% endblock %}
```

3. Rajoutez le code Ajax dans un bloc **javascripts** dans la même vue, le code doit faire appel à une action dans le controller qui gère la petition Ajax.

```
{% block javascripts %}
<!-- AJAX - AXIOS dans la page, sans avoir un script externe -->

<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    envoyerNom.addEventListener("click", function (event) {
        event.preventDefault();

        console.log(document.getElementById("leFormulaire"));

        axios({
            url: '{{path("exemple1_traitement")}}',
            method: 'POST',
            headers: { 'Content-Type': 'multipart/form-data' },
            data: new FormData(document.getElementById("leFormulaire"))
        })
        .then(function (response) {
            // response.data est un objet qui correspond à l'array associatif e
nvoyé dans le controller
            // JsonResponse a transformé l'array en JSON. Axios transforme le J
SON en objet JS
            // (et on utilise ici la clé "leMessage")
            document.getElementById("divMessage").innerHTML = response.data.leM
essage;
            console.log(response);
        })
    })
}
```



```
        .catch(function (error) {  
            console.log(error);  
        });  
    });  
</script>  
{% endblock %}
```

Notez que dans le code Ajax on doit réaliser l'opération pertinente avec les données reçues du serveur (ex : afficher dans un div)

4. Créez l'action qui affiche la vue qu'on vient de créer

```
// exemple d'utilisation d'AJAX avec de blocs ("master page")

/**
 * @Route ("/exemples/ajax/axios/exemple1/affichage/master/page");
 */
public function exemple1AffichageMasterPage()
{
    return $this-
>render("/exemples_ajax_axios/exemple1_affichage_master_page.html.twig");
}
```

5. Créez l'action qui traite la commande AJAX

Dans cette action, renvoyez votre réponse JSON. Pour ce faire, au lieu d'envoyer un objet Response ou le rendu d'une vue, vous allez utiliser un objet JsonResponse. Par exemple :

```
/**
 * @Route ("/exemples/ajax/axios/exemple1/traitemement/master/page");
 */
// action qui traite la commande AJAX, elle n'a pas une vue associée
public function exemple1TraitementMasterPage(Request $requeteAjax)
{
    $valeurNom = $requeteAjax->get('nom');
    $arrayReponse = ['message' => 'Bienvenu, ' . $valeurNom];
    return new JsonResponse($arrayReponse);
}
```

Exercice 1 : faites un jeu de deviner un chiffre en utilisant Ajax en Symfony (utilisez le controller AjaxExemples)

Exercice 2 : créez une autre master page et deux vues qui en héritent. La première contient le jeu que vous venez de réaliser et la deuxième contient trois boutons. Chaque bouton affiche la photo d'un animal sans recharger la page.

Ajax et Axios avec script externe au Twig (sans Webpack)

Si on veut utiliser un script externe JS dans une vue, le script ne pourra pas utiliser la fonction « **path** » pour générer les routes de cible AJAX. Les fonctions de twig telles que **path** fonctionnent uniquement **dans les fichiers TWIG**. Ceci est un problème typique qu'on peut résoudre en utilisant le module **FOSJsRoutingBundle**.

Un exemple pratique est réalisé dans le projet **projetFormulairesSymfony5**, dans le controller ExemplesAjaxAxiosController :

Actions :

- exemple1AffichageMasterPageScriptExterne
- exemple1TraitementMasterPageScriptExterne

Vue :

- exemple1_affichage_master_page_script_externe.html

Important : dans les routes qui seront accédées par ce bundle (regardez le code dans le controller) vous devez rajouter le paramètre **{"expose"=true}**. Le code du projet inclut déjà cette option.

```
/**
 * @Route ("/exemples/ajax/axios/exemple1/traitement/master/page/script/externe",
 *         options={"expose"=true}, name="exemple1_traitement_externe");
 */
```

26. AJAX en Symfony (Vanilla JS)

Objectif : utiliser AJAX dans un template Twig

Créez un controller **ExemplesAjaxFormDataController** (code original dans le projet **projetFormulaires**). Ce controller contiendra uniquement quelques exemples d'appel Ajax. Plus tard on réalisera des exemples plus pratiques basés sur la BD du projet.

Exemple d'appel AJAX avec un formulaire

Dans cet exemple on envoie de données en utilisant AJAX **sans utiliser un formulaire**. Nous avons juste les contrôles. Dans la section suivante on utilisera un formulaire complet

1. Créez une vue contenant du code AJAX

Exemple : on tapera un nom dans l'input et, quand on clique sur le bouton, un message de bienvenue sera affiché dans le div. Analysez vous-même le code.

Attention aux **names** des contrôles car on les utilisera dans le traitement de l'action dans le controller!!

```
<form id="leFormulaire" method="POST">
<input type="text" id="inputNom" name="nom" />
<input type="submit" id="envoyerNom" value="Envoyer"/>
</form>

<div id="divMessage"></div>

<script>

envoyerNom.addEventListener ("click", function (event){
    // on annule l'effet du submit
    event.preventDefault();

    var xhr = new XMLHttpRequest ();

    // on crée un formulaire à partir de celui du DOM
    var formulaire = new FormData (leFormulaire); // accès direct, on peut aussi
    utiliser getElementById

    xhr.onreadystatechange = function (){
        console.log (xhr.status);
        if (xhr.readyState == 4){
            if (xhr.status == 200){
                // transformer le string JSON envoyé par le serveur
                // comme réponse en objet JavaScript
                var reponse = JSON.parse (xhr.responseText);
```

```

        divMessage.innerHTML = reponse.message;
        console.log (reponse);
        console.log (typeof(reponse));
    }
    // s'il y a une erreur:
    else {
        // effacer en production!
        console.log (xhr.reponseText);
    }
}

}

}

xhr.open ('POST', '/exemples/ajax/form/data/exemple1/traitement');
// on envoie l'objet formulaire
xhr.send (formulaire);

// xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
//xhr.send ("nom=" + inputNom.value);

});
</script>

```

2. Créez l'action **exemple1Affichage**, qui renvoie le rendu de la vue `exemple1_affichage.html`

```

/**
 * @Route ("/exemples/ajax/form/data/exemple1/affichage");
 */
public function exemple1Affichage (){
    return $this->render ("/exemples_ajax_form_data/exemple1_affichage.html.twig");
}

```

3. Créez l'action **exempleTraitementAjax** qui traite la pétition AJAX

```

/**
 * @Route ("/exemples/ajax/form/data/exemple1/traitement");
 */
// action qui traite la commande AJAX, elle n'a pas une vue associée
public function exemple1Traitement (Request $requeteAjax){
    $valeurNom = $requeteAjax->get ('nom');
    $arrayReponse = ['message' => 'Bienvenu, ' . $valeurNom];
    return new JsonResponse ($arrayReponse);
}

```


Utilisation de blocs dans twig avec AJAX

Il s'agit juste d'une combinaison de master page + AJAX, rien de nouveau.

6. Créez un template `master_page.html.twig` contenant une section pour nos vues. Créez un `26.2.` block pour le contenu et un autre pour le JS

```
<html>
  <body>
    <header>
      Voici la section header
    </header>
    <main>
      Voici la section main
      {% block contenuMain %}{% endblock %}
    </main>
    <footer>
      Voici la section footer
    </footer>
  </body>
  {% block javascripts %}{% endblock %}
</html>
```

7. Créez une vue `exemple1_affichage_master_page.html.twig` qui hérite du template `master_page.html.twig`

```
{% extends '/exemples_ajax/master_page.html.twig' %}

{% block contenuMain %}
Nom<input type="text" id="inputNom" />
<button id="envoyerNom">Envoyer</button>
<div id="divMessage"></div>
{% endblock %}
```

8. Rajoutez le code Ajax dans un bloc `javascripts` dans la même vue, le code doit faire appel à une action dans le controller qui gère la petition Ajax.

```
{% block javascripts %}
envoyerNom.addEventListener ("click", function (event){
  var xhr = new XMLHttpRequest ();

  xhr.onreadystatechange = function (){
    if (xhr.readyState == 4){
      if (xhr.status == 200){
```

```

        // transformer le string JSON envoyé par le serveur
        // comme réponse en objet JavaScript
        var reponse = JSON.parse (xhr.responseText);
        divMessage.innerHTML = reponse.message;
        console.log (reponse);
        console.log (typeof(reponse));
    }
    // s'il y a une erreur:
    else {
        // effacer en production!
        console.log (xhr.responseText);
    }
}
}
xhr.open ('POST', '/exemples/ajax/exemple1/traitement/master/page');
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhr.send ("nom=" + inputNom.value);

});

{% endblock %}

```

Notez que dans le code Ajax on doit réaliser l'opération pertinente avec les données reçues du serveur (ex : afficher dans un div)

9. Créez l'action qui affiche la vue qu'on vient de créer

```
/**
 * @Route ("/exemples/ajax/exemple1/affichage/master/page");
 */
public function exemple1AffichageMasterPage (){
    return $this->render
    ("/exemples_ajax/exemple1_affichage_master_page.html.twig");
}
```

10. Créez l'action qui traite la demande AJAX

Dans cette action, renvoyez votre réponse JSON. Pour ce faire, au lieu d'envoyer un objet Response ou le rendu d'une vue, vous allez utiliser un objet JsonResponse. Par exemple :

```
/**
 * @Route ("/exemples/ajax/exemple1/traitement/master/page");
 */
// action qui traite la commande AJAX, elle n'a pas une vue associée
public function exemple1TraitementMasterPage (Request $requeteAjax){
    $valeurNom = $requeteAjax->get ('nom');
    $arrayReponse = ['message' => 'Bienvenu, ' . $valeurNom];
    return new JsonResponse ($arrayReponse);
}
```

Pour finir, sachez que les fichiers **.js** et **.css** sont considérés comme des "assets" en Symfony. Pour pouvoir en rajouter dans notre projet vous devez créer les dossiers **public/assets/js** et **public/assets/css** respectivement et y placer vos fichiers. Dans vos vues, inclure les fichiers est simple :

```
<script src="{ asset('/assets/js/monFichier.js') }"></script>
<link rel="stylesheet" href="{ asset('/assets/css/monCss.css') }" />
```

Vous avez des exemples dans le projet **projetFormulaires (controller ExemplesAjaxController)**

Exercice 1 : faites un jeu de deviner un chiffre en utilisant Ajax en Symfony (utilisez le controller AjaxExemples)

Exercice 2 : créez une autre master page et deux vues qui en héritent. La première contient le jeu que vous venez de réaliser et la deuxième contient trois boutons. Chaque bouton affiche la photo d'un animal sans recharger la page.

27. Renvoi d'un array d'objets en JSON

Renvoi d'un array d'objets en JSON (repo)

findBy, findAll + serialize + renvoyer un objet Response

Exemple : obtenir une liste d'aéroports et les afficher dans un div

Code :

- **Projet** projetFormulaires
- **Controller** ExemplesAjaxAxiosController, actions
 - o exempleAffichageObjetsRepo
 - o exempleAffichageObjetsTraitementRepo
- **Vue** exemple_affichage_objets_repo.html.twig

Renvoi d'un array d'objets en JSON (DQL)

27.2.

getArrayResult + renvoyer un objet JsonResponse
--

Exemple : obtenir une liste d'aéroports et les afficher dans un div

Code :

- **Projet** projetFormulaires
- **Controller** ExemplesAjaxAxiosController, actions
 - o exempleAffichageObjetsDql
 - o exempleAffichageObjetsTraitementDql
- **Vue** exemple_affichage_objets_dql.html.twig

28. Doctrine Fixtures

Doctrine fournit un outil qui nous permet **d'encoder de données dans la base de données d'une façon semi-automatique**, ce qui est très **utile pendant les périodes de développement et de test de l'application**.

Le fonctionnement est simple : **si on veut encoder de données pour une classe d'entité** existante (ex : Livres) on **demande à Doctrine de créer une class Fixture** (ex : LivresFixtures) qui contient au moins une **méthode load**. Dans cette méthode (à remplir par nous) **contiendra le code qui insère** les données dans la BD. Puis on appelle cette fonction et les données seront stockés dans la BD.

Ce système a plusieurs avantages :

- On peut appeler la méthode génératrice autant de fois qu'on veut
- Le code qui crée les données de la BD se trouve localisé
- On peut générer les données pour toutes les entités du projet avec une seule commande (si on a créé la Fixture pour chaque entité, bien sûr).

Toute la documentation sur les fixtures se trouve ici :

<https://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html>

mais on va développer un exemple simple et associé à une classe d'entité qui nous servira plus tard.

Exemple : Création d'une fixture

On va créer et lancer une fixture pour l'entité **Pays** dans le projet **projetFormulaires**. Si l'entité n'existe pas, créez la d'abord (Pays: nom et lienImage). Suivez cette procédure :

1. Installez le **support** pour les **fixtures**

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

2. **Créez la classe fixture** (nom: PaysFixture)

```
php bin/console make:fixture
```

3. **Editez la fonction load** pour qu'elle stocke de Pays. Voir cet exemple :

```

namespace App\DataFixtures;

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

class PaysFixtures extends Fixture
{
    // load créera et stockera 20 pays dans la BD
    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 20; $i++){
            $pays = new Pays ();
            $pays->setNom ("Belgique" . $i);
            $pays->setLienImage ("belgique" . $i . ".jpg");

            $manager->persist($pays);
        }
        $manager->flush();
    }
}

```

4. Lancez les fixtures

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

--append permet de lancer la fixture sans effacer les données existantes dans les tableaux. Si vous l'enlevez-vous effacerez la totalité du contenu de votre BD (Symfony vous prévient quand-même)

Ici on a qu'une fixture mais on pourrait avoir plein.

5. Vérifiez que les données sont insérées dans la BD

Note: si vous voulez générer de valeurs plus "réalistes" vous pouvez utiliser la librairie Faker.

Exercices :

1. Créez une classe Fixture qui permette de rajouter automatiquement des objets d'une classe de votre choix

29. Authentification : inscription et login/password

Objectif : créer un projet (ProjetLoginPass) contenant :

1. Un formulaire de login/password traditionnel
2. Un formulaire d'inscription pour rajouter des utilisateurs dans la BD

Important : Si vous avez une ancienne version de XAMPP assurez-vous d'avoir au moins la version 10.2 de MariaDB. Pour mettre à jour votre version de MariaDB pour xampp suivez les instructions qui se trouvent ici **dans sa totalité** :

<https://stackoverflow.com/questions/44027926/update-xampp-from-maria-db-10-1-to-10-2>

Configuration de la sécurité et création d'un formulaire de login

29.1. On va réaliser une configuration de base de la sécurité pour pouvoir créer un formulaire d'inscription/login standard. Pour des options plus avancées (ex : changez d'utilisateur sans devoir se déconnecter de l'application) consultez la documentation ici :

<https://symfony.com/doc/current/security.html>

https://symfony.com/doc/current/security/form_login_setup.html

La procédure à suivre devra :

1. **Installer le support de sécurité dans le projet**
2. **Créer une entité User** avec l'assistant
3. **Créer** (avec l'assistant)
 - Un **controller** pour le **login** et le **logout**
 - Un **template** pour **afficher le formulaire** de login
 - Un **Guard Authenticator**, classe qui **traite les informations** du formulaire de login
4. Configurez la BD dans **.env**, créez le **schéma** de la BD, créez et lancez une **migration**
5. **Encoder des utilisateurs et de passwords dans la BD**
6. **Vérifier** le bon fonctionnement en tapant un couple login/pass valable

Réalisez la procédure en détail :

1. Installer le support de sécurité dans le projet

```
composer require symfony/security-bundle
```

2. Créer une entité User avec l'assistant avec **make:user** (pas **make:entity**!)

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

Cette commande crée l'entité, qui **doit** implémenter l'interface [UserInterface](#) (Faites la migration pour que la BD soit mise à jour!)

L'assistant vous demandera :

- Le nom de la classe (on choisira User)
- Si vous voulez stocker de données dans la BD avec Doctrine (oui!)
- La propriété qu'on utilisera pour réaliser le login (on choisira email)
- Si on souhaite hasher les passwords (oui!)

Ouvrez **src/Entity/User.php** et regardez le code. **Vous pouvez par après rajouter d'autres propriétés ou méthodes si vous le souhaitez (make:entity)**

L'assistant aura modifié aussi le fichier **security.yaml** (dans **config/packages**) selon les informations qu'on vient de fournir.

Note : c'est très important de respecter l'indentation dans les fichiers .yaml

3. Créer le controller, le template et un Guard Authenticator (avec l'assistant) :

- Un **controller** pour le **login** et **une route**
- Un **template** pour afficher le **formulaire** de login
- Un **Guard Authenticator**, **classe** qui **traite les informations** du formulaire de login

Ces trois pas se font **avec une seule commande de l'assistant** :

```
php bin/console make:auth
```

Pour les questions posées par l'assistant on choisira :

- **L'option 1** pour que Symfony crée un formulaire de login de base et pas seulement le système d'authentification vide
- **FormulaireLoginAuthenticator** comme nom de la classe Guard Authenticator qui prendra en charge la requête à la BD pour réaliser **l'authentification** (créé dans le dossier **src/Security**)
- **SecuriteController** comme nom du controller (actions login et logout)
- **Oui**, car on veut que Symfony crée aussi l'URL de logout (avec l'action qui deloggera l'utilisateur, c.à.d. l'effacer de la session)

Cette action met aussi à jour le fichier de configuration **config/packages/security.yaml**.

Observez que le controller et le template ont été créés. Vous pouvez accéder à la vue contenant le formulaire de login en tapant la route de l'action **login** du controller.

4. Configurez la BD dans **.env** (projetloginpass), créez le **schéma** de la BD, créez et lancez une **migration**

```
php bin/console doctrine:database:create
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

5. Encoder des utilisateurs et de passwords dans la BD

Créez une fixture pour la classe User (voir chapitre précédant sur le Doctrine Fixtures).

```
composer require --dev doctrine/doctrine-fixtures-bundle
php bin/console make:fixture
```

La fixture portera le nom **UserFixtures**. Attention au nom car si on se trompe il n'y aura pas un message d'erreur.

Cette méthode est plus facile qu'encoder les utilisateurs à la main, **car le password doit être hashé**

Doc: <https://symfony.com/doc/current/security.html> (2c)

Dans ce cas, la fonction **load** devra créer un utilisateur, fixer ses attributs et le stocker dans la BD. Nous devons utiliser un service pour encoder le password avant d'appeler à setPassword. Le service est injecté dans le constructeur de la classe (dependency injection par le constructeur !!).

Voici un code possible pour la Fixture **UserFixtures.php** :

```
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

use App\Entity\User;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

class UserFixtures extends Fixture
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }

    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 10 ; $i++){
            $user = new User();
            $user->setEmail ("user".$i."@lala.com");
            $user->setPassword($this->passwordEncoder->encodePassword(
                $user,
                'lePassword' . $i
            ));
            $manager->persist ($user);
        }
        $manager->flush();
    }
}
```

Important : si votre entité a d'autres attributs (nom, adresse, etc...) il faudra rajouter les sets qui correspondent

N'oubliez pas de lancer la fixture avec :

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































Note : Symfony nous indique qu'il effacera la BD (purge). Choisissez **oui**.

Si vous avez besoin à un moment donné d'obtenir le hash d'un password depuis la console, tapez :

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

et puis tapez le password. Vous pouvez par après le copier-coller dans la table (colonne password)

Dans **phpmyadmin** votre tableau **User** ressemblera à :

<div><div><div>←</div><div>T</div><div>→</div></div><div></div></div>						id	email	roles (DC2Type=json)	password
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	1	user0@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	2	user1@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	3	user2@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	4	user3@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	5	user4@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	6	user5@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	7	user6@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	8	user7@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	9	user8@lala.com	[]	\$argon2id\$v=19:		
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	10	user9@lala.com	[]	\$argon2id\$v=19:		

6. Vérifier le bon fonctionnement en tapant un couple login/pass valable

Allez sur la page de login (par défaut l'action login dans SecuriteController) et tapez un couple login/pass valable. Si tout va bien vous allez obtenir une Exception car **dans votre controller Authenticator (FormulaireLoginAuthenticator** dans le dossier **src/Security**) vous n'avez pas spécifié une Response pour le serveur (méthode **onAuthenticationSuccess** de ce controller)

Exception

TODO: provide a valid redirect inside C:\xampp\htdocs\Symfony5\Tests\ProjetLoginPass\src\Security\FormulaireLoginAuthenticator.php

Vous avez juste à implémenter cette action pour indiquer quoi faire dans le cas de succès.
Voici un exemple :

```

public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
        return new RedirectResponse($targetPath);
    }

    // For example : return new RedirectResponse($this->urlGenerator->generate('some_route'));

    // nous devons charger une vue ou faire quoi que ce soit
    // ex:
    // on peut penser à : return $this->redirectToRoute('accueil'); // mais cette classe n'a pas la méthode car
    // elle n'est pas un controller! On utilise alors :
    return new RedirectResponse($this->urlGenerator->generate('accueil'));

    // throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
}

```

Dans le cas de succès, le code qui reste de l'action **login** ne sera pas lancé car on a fait un redirect. Ici vers une action qui porte le nom 'accueil', mais c'est à vous de choisir. Pour l'exemple, créez le controller AccueilController avec l'assistant et une vue contenant un message de bienvenue.

Si une erreur de login s'est produite, **nous avons deux possibilités** pour le traiter :

a) Utiliser le template login crée par Symfony et l'adapter à nos besoins (par défaut)

Dans cet exemple, si le couple login/pass n'est pas correcte l'action **onAuthenticationSuccess** ne sera pas lancé. Symfony essayera de lancer l'action **onAuthenticationFailure** mais elle n'existe pas. Le code de l'action login continuera et la variable **error** contiendra l'info de l'erreur de login.

La vue sera rechargée et affichera (voir if) un div contenant le message de l'erreur qui s'est produite (ex: mail inexistant, invalid credentials si le password n'est pas correcte...).

Dans la vue on peut choisir par nous-mêmes quoi faire s'il y a une erreur, il suffit de vérifier la valeur de cette variable et agir conséquemment (afficher un message d'erreur, rediriger vers un autre site etc...). On peut aussi juste établir une traduction pour les messages d'erreur de base de Symfony, car par défaut ils seront en anglais !

À chaque essai de login c'est conseillé de lancer l'action **logout** pour effacer le contenu de la session. On parlera du logout plus bas.

b) Définir `onAuthenticationFailure` dans `FormulaireLoginAuthenticator.php`

Cette action sera lancée quand il y aura une erreur de login, de la même manière que **`onAuthenticationSuccess`** est lancée dans le cas de succès. Elle est commentée dans le code, effacez les commentaires pour que Symfony la trouve. Le comportement expliqué dans a) sera logiquement annulé car le code de la vue ne sera plus lancé.

```
// methode faite par nous-mêmes. Enlevez les commentaires pour voir l'effet
public function onAuthenticationFailure(
    \Symfony\Component\HttpFoundation\Request $request,
    \Symfony\Component\Security\Core\Exception\AuthenticationException $exception
)
{
    throw new Exception ("error dans le login, c'est onAuthenticationFailure dans F
ormulaireLoginAuthenticator qui s'en occupe"); // rediriger, exception etc...
}
```

(En cours, cette doc. appartient à Symfony 4) Traduction des messages de succès/erreur

1. Changer la variable **locale** de **en** à **fr** dans **config/services.yaml**

29.2.

2. Créez un fichier contenant les traductions des messages selon le **locale** (voir **translations/security.fr.xlf** dans **projetLoginPass**)

Maintenant, à chaque fois qu'un service de Symfony renvoie un message il lira le fichier de traductions. Nous avons qu'à rajouter la traduction de chaque message en français.

Le service de traduction mérite d'une section à part qu'on ne traitera pas dans ce tuto.

<https://symfony.com/doc/current/translation.html> (voir la section **Basic Translation**).

Création d'un formulaire d'inscription

Vous pouvez créer un formulaire d'inscription automatiquement et le personnaliser après en suivant les instructions de cette documentation :

https://symfony.com/doc/current/doctrine/registration_form.html

29.3.

Si vous n'avez pas réalisé les opérations du chapitre précédente, suivez au moins les pas 1,2,3 pour configurer la sécurité dans Symfony, créer l'entité User et le Guard Authenticator.

Voici la continuation de la procédure, qui créera un formulaire d'inscription :

Lancez, dans la console :

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

Choisissez si vous voulez qu'on ne puisse pas avoir de doublons dans les Users et si vous voulez que les utilisateurs soient logés directement après l'inscription (comme dans la plupart de sites)

L'assistant créera :

- Une classe formulaire (**RegistrationFormType**)
- Un controller qui crée l'objet formulaire et le renvoie à la vue
- Un template qui affiche le formulaire

A ce stade de la formation vous savez comment éditer le formulaire pour l'adapter à vos besoins.

Important : si vous modifiez l'entité User pour, par exemple, rajouter une propriété **nom**, et vous voulez **générer à nouveau le formulaire d'inscription**, effacez d'abord le formulaire `RegistrationFormType.php`, le controller `RegistrationController.php` et le template `register.html.twig`.

30. Logout

Les outils de sécurité de Symfony nous permettent d'implémenter le logout très facilement :

- 1) Rajoutez dans `config/packages/security.yaml` une section qui indique le path à saisir dans l'URL quand on veut réaliser un logout (pas un name) et la route (pas un name) de l'action qui sera lancée après avoir fait le logout (route complete, pas le name !). "Faire le logout" est, en gros, effacer l'objet User de la session. Symfony s'en chargera de le faire sans votre intervention

```
main:
    anonymous: true
    guard:
        authenticators:
            - App\Security\FormulaireLoginAuthenticator
    logout:
        path: /logout
        target: /apres/logout
```

Important : Respectez l'indentation dans les fichiers .yaml. Elle est faite avec des espaces!

On doit avoir une action

- 2) Laissez vide l'action de logout (elle ne sera jamais lancée) et créez l'action à lancer après d'avoir fini le traitement du logout (effacer user, session etc...)

"ProjetLoginPass" contient cette fonctionnalité. L'action cible se trouve dans **SecuriteController**.

```
// La route "logout" sera utilisé par security.yaml
// Le name "logout" sera utilisé dans le path de la vue (le lien de logout)
// le code de cette action ne se lancera jamais
/**
 * @Route("/logout", name="logout")
 */
public function logout()
{
    // ce code ne se lance jamais, cette action peut être vide
}
// target: la route à lancer APRÈS le logout
/**
 * @Route("/apres/logout")
 */
public function apresLogout()
{
    dd("Hasta la vista, baby");
}
```

}

31. Accès à l'objet app.user

Une fois l'utilisateur est logué vous pouvez obtenir son objet **User** associé :

1. Dans le controller

```
$this->getUser()
```

2. Dans la vue

```
app.user
```

Les deux méthodes renvoient l'objet User représentant l'utilisateur qui es connecté ou **null** si personne n'a fait login.

L'objet User contient **toutes les propriétés et on peut les accéder en utilisant les gets et sets**.

Par exemple :

```
{{ dump (app.user) }}
```

Ces deux instructions donnent les même résultat car app.user.username est juste un raccourci de Symfony pour user.getUsername()

```
{{ dump (app.user.getUsername()) }}  
{{ dump (app.user.username) }}
```

Des actions d'exemple se trouvent dans le projet "ProjetLoginPass", controller SecurityController

32. Authentication et Rôles

Gestion de rôles

Nous allons traiter la gestion de rôles en Symfony et on va utiliser comme base le projet qu'on vient de créer, **ProjetLoginPass**. Nous voulons profiter de toute la partie d'authentification qui reste la même et qu'on ne veut pas refaire.

- 1) Clonez le projet **ProjetLoginPass** de github dans un dossier **ProjetLoginPassRoles** avec cette ligne :

```
git clone https://github.com/choquitofrito/ProjetLoginPass.git ProjetLoginPassRoles
composer install
```

- 2) Effacez les migrations (dans le dossier Migrations, elles correspondent à l'autre projet), modifiez votre fichier **.env** pour pointer vers une autre base de données **projetloginpassroles**.
- 3) Juste pour enrichir le projet et montrer que c'est faisable, rajouter une propriété **nom** à l'entité User

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

- 4) Rajoutez un champ **TextType** (importez-le!) au formulaire et effacez la section "agree terms" dans **Form\RegistrationFormType.php** pour pouvoir saisir le nom aussi dans le form d'inscription

```
...
->add('email')
->add('nom', TextType::class)
->add('plainPassword', PasswordType::class, [
...

```

- 5) Editez le formulaire d'inscription (vue **registration/registration.html.twig**)

Rajoutez la génération du champ du formulaire **nom**

```
{{ form_row(registrationForm.nom) }}
```

Et effacez la génération du champ **agreeTerms**

Vous pouvez tester ce formulaire. Il manque uniquement la fonctionnalité de rajouter de rôles mais on ne fera pas ça ici. Vous pouvez toujours éditer les rôles plus tard à la main ou en utilisant la méthode **setRoles** de l'entité!

6) Créez la BD même manière que dans le projet précédent

```
php bin/console doctrine:database:create
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

7) Remplacez la fixture par celle-ci, qui rajoute des Users avec de rôles (comprenez le code!)

```
class UserFixtures extends Fixture
{
    private $passwordEncoder;

    public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->passwordEncoder = $passwordEncoder;
    }

    public function load(ObjectManager $manager)
    {
        // on va créer 5 admins et 5 clients+gestionnaires
        for ($i = 0; $i < 5 ; $i++){
            $user = new User();
            $user->setEmail
            ("newuser".$i."@lala.com"); // user1@lala.com, user2@lala.com etc....
            $user->setPassword
            ($this->passwordEncoder->encodePassword(
                $user,
                'lePassword'.$i // lepassword1, lepassword2, etc...
            ));
            $user->setNom("nom".$i);
            $user->setRoles(['ROLE_ADMIN']);
            $manager->persist ($user);
        }
        for ($i = 0; $i < 5 ; $i++){
            $user = new User();
```

```

        $user->
setEmail ("autreuser".$i."@lala.com"); // user1@lala.com, user2@lala.com etc....
        $user->setPassword($this->passwordEncoder->encodePassword(
            $user,
            'lePassword'.$i // lepassword1, lepassword2, etc...
        ));
        $user->setNom("nom".$i);
        $user->setRoles(['ROLE_CLIENT', 'ROLE_GESTIONNAIRE']);
        $manager->persist ($user);
    }
    $manager->flush();
}
}

```

- 8) Lancer les fixtures pour la remplir. Dans ce projet on crée plusieurs types d'user (voir **DataFixtures/UserFixtures**). Selon le système de rôles de Symfony, vous pouvez choisir vous-mêmes les noms des rôles en sachant que le nom du rôle doit commencer par **ROLE_** (ex: **ROLE_CLIENT**, **ROLE_ADMIN**, **ROLE_PARTICIPANT**...)

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

(pas --append car vous voulez carrément effacer le tableau User)

Note : Une autre option pour remplir les users (plus élaborée, possible alternative à la fixture et qui n'est pas convenable ici) est de créer un user ayant un rôle **ROLE_SUPER_ADMIN** qui puisse accéder à la gestion de tous les utilisateurs en utilisant un formulaire. Cet user aura accès à une route qui affiche un deuxième formulaire d'inscription/modification permettant le choix/modification du rôle des utilisateurs. Attention car la sécurité de votre site peut être en jeu !

Contrôle d'accès par rôles

Puis vous pouvez restreindre l'accès à certains rôles de trois manières :

1. dans **security.yaml**
2. dans un controller
- 32.3. dans une vue

On va développer ici un exemple de chaque méthode

32.2.1. Dans security.yaml

On peut restreindre l'accès à de grandes sections de notre site (ex: partie admin) avec **le control d'accès dans config/packages/security.yaml**. C'est assez simple mais il faut connaître un minimum les expressions régulières ou adapter les exemples ci-dessous à vos besoins

<https://symfony.com/doc/current/security.html#security-authorization-access-control>

1. **Créez un controller** `GestionController` et ses vues correspondantes. On utilisera ces actions pour vérifier le bon fonctionnement des restrictions qu'on fera plus tard dans `security.yaml`

```
class GestionController extends AbstractController
{
    // ces routes sont accessibles uniquement pour certains roles
    // car on l'a fixé dans security.yaml
    /**
     * @Route("/gestion/action1")
     */
    public function action1()
    {
        return $this->render('gestion/action1.html.twig');
    }
    /**
     * @Route("/gestion/action2")
     */
    public function action2()
    {
        return $this->render('gestion/action2.html.twig');
    }
}
```

(gestion/action1.html.twig, le code sera pareil pour action2. Le dump affiche les rôles)

```
{% extends 'base.html.twig' %}
{% block body %}
Voici action 1
{{ dump (app.user.getRoles()) }}
{% endblock %}
```

2. Créez les restrictions dans security.yaml

Les deux actions de ce controller doivent être uniquement par un utilisateur ayant le role `ROLE_GESTIONNAIRE`. C'est dans **security.yaml** qu'on a fixé cette restriction :

```
access_control:
- { path: ^/gestion, roles: [ROLE_GESTIONNAIRE] }
```

Faites logout. Faites login avec un user de chaque type (regardez la BD) et essayez de lancer les actions `gestion/action1` et `gestion/action2` (depuis l'URL). Observez les résultats selon l'utilisateur qui est connecté : seulement les users ayant le `ROLE_GESTIONNAIRE` pourront lancer ces actions. Les autres obtiennent une exception **Access Denied**

32.2.2. Dans le controller

Si on ne veut pas créer de restrictions par routes, on peut tout simplement vérifier si l'utilisateur qui est connecté possède le rôle demandé à l'intérieur d'une action du controller.

Ex: dans une action, permettre l'accès à l'action uniquement au role `ROLE_CLIENT`

```
$this->denyAccessUnlessGranted(["ROLE_CLIENT"]);
```

Si l'user n'a aucun de ces rôles il y aura une exception.

Créez le controller **AutreController** et ses vues, en rajoutant le code qui vérifie le rôle :

```
/**
 * @Route("/autre/action1")
 */
public function action1()
{
    // deux rôles peuvent avoir l'accès mais il y a un bug!
    // this->denyAccessUnlessGranted(['ROLE_CLIENT', 'ROLE_ADMIN']);

    $this->denyAccessUnlessGranted("ROLE_ADMIN");

    return $this->render('autre/action1.html.twig');
}

/**
 * @Route("/autre/action2")
 */
public function action2()
{
    $this->denyAccessUnlessGranted('ROLE_GESTIONNAIRE');

    // si pas d'exception...
    return $this->render('autre/action2.html.twig');
}
```

Si l'utilisateur ne possède pas le rôle fixé dans l'action, **une exception sera lancée**. Pour tester le bon fonctionnement faites d'abord logout. Faites login avec un user de chaque type (regardez la BD) et essayez de lancer les actions `autre/action1` et `autre/action2` (depuis l'URL). Observez les résultats selon l'user qui est connecté.

32.2.3. Dans les vues

Vous pouvez aussi vérifier les rôles dans les vues pour afficher/masquer le code de votre choix selon le rôle de l'utilisateur qui est connecté en utilisant **is_granted**.

Créez une nouvelle action **action 3** qui ne limite pas par rôle :

```
/**
 * @Route("/autre/action3")
 */
public function action3()
{
    // on va controller l'accès dans la vue
    return $this->render('autre/action3.html.twig');
}
```

Et controlez le rôle dans la vue :

```
{% if is_granted('ROLE_ADMIN') %}
    La vie de l'admin est dure
    {# ex: <a href="{{ path ('effacer_user') }}">effacer</a> #}
{% else %}
    La vie est belle
{% endif %}
```

Gestion de l'erreur "Access Denied" (exception) en utilisant une classe propre

Pour **personnaliser l'action à réaliser** en cas d'**erreur d'accès par rôle** vous devez utiliser une classe propre.

32.3.

1. **Créer une classe** (ici c'est **Security/ GestionnaireErreurAcces.php**) contenant une action où on fixera l'action à réaliser. Voici un exemple où, dans le cas d'une erreur d'accès, on redirige vers login (notez que la redirection se fait de manière différente quand on se trouve à l'extérieur du controller)

```
<?php
```

```
namespace App\Security;
```

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class GestionnaireErreurAcces implements AccessDeniedHandlerInterface
{
    private $router;

    public function __construct (UrlGeneratorInterface $router){
        $this->router = $router;
    }

    public function handle(Request $request, AccessDeniedException $accessDeniedException)
    {
        // choisissez la route vers laquelle y aller. Ici on a choisi app_login
        return new RedirectResponse ($this->router->generate ("app_login"));
    }
}
```

2. Rajouter la clé **access_denied_handler** dans **security.yaml** qui pointe vers la classe qu'on vient de créer (indentation !!!)

```
logout:
  path: /logout
  target: /apres/logout
  # target est le name l'action qui
  # sera lancée après l'action logout
access_denied_handler: App\Security\GestionnaireErreurAcces
```

33. Fenêtre modale pour le login

Description générale

Cette section explique plus en profondeur le comportement du système de login/pass qu'on peut créer automatiquement avec **make:auth**, ainsi que les bases pour modifier ce système en utilisant une **fenêtre modale et ajax**.

On part du principe que vous avez déjà créé votre entité user et le système de login avec **make:auth** (voir les sections précédentes).

L'action login créé par Symfony rend toujours la vue qui affiche le login (`$this->render`).

Avec la configuration par défaut de Symfony, cette action sera lancée dans deux cas de figure :

- **GET** : quand on tape **/login** dans l'URL du navigateur ou, normalement, quand on génère la route avec un href

- **POST** : quand on fait **submit** dans un formulaire dont l'action pointe vers la route de cette action.

Dans le code généré par Symfony pour le form de login il n'y a pas d'attribut **"action"**. Ça implique que quand on clique sur le bouton de submit on chargera à nouveau la même route (dans ce cas ça sera /login). Plusieurs actions de la classe de notre Authenticator seront lancées **avant de lancer le code de cette action**.

En résumé :

1. si on tape /login dans l'URL, l'action fait un rendu de la vue login.html.twig. La vue envoie deux paramètres au controller : le dernier utilisateur connecté avec succès et un message correspondant à l'erreur qui s'est produite dans le dernier essai de connexion
2. si on fait **submit** et le **login est ok** on charge **onAuthenticationSuccess** et puis le code de l'action **login**, sauf si à l'intérieur de la méthode **onAuthenticationSuccess** on redirige vers une autre action.

Note: sachez que cette action sera aussi appelée si on crée un form d'enregistrement et on choisit d'être logué automatiquement après l'enregistrement

3. si on fait **submit** et le **login n'est pas ok** on charge **onAuthenticationFailure**. Si cette action n'existe pas on charge directement le code de l'action login, qui charge à son tour la vue login. Pareil que dans **onAuthenticationSuccess** on peut rediriger, lancer une exception ou quoi que ce soit. La différence est que cette action est optionnelle, mais **onAuthenticationSuccess** est obligatoire.

Tel qu'on a déjà mentionné, dans le code généré par défaut par Symfony l'action login envoie toujours deux valeurs à la vue :

1. **lastUsername** : contient le nom du dernier utilisateur qui s'est logué correctement

Ces infos sont utilisées dans le template par défaut, mais bien évidemment vous pouvez les utiliser comment vous voulez.

Dans plein de cas on va vouloir utiliser une fenêtre modale pour le login (ou même pour d'autres actions). Considérons un cas pratique :

- Pour ce faire on a besoin d'AJAX. Pour nous faciliter la tâche on utilisera Axios. Voici l'ensemble d'actions pour créer un exemple de ce système.

- 1) Inclure la vue login.html.twig créé par Symfony dans le bon emplacement (ici, dans le header.twig.html à l'emplacement correspondant). Utilisez include (twig). On peut adapter le code selon nos besoins. Logiquement on doit remplacer l'ancien contenu du template.

(ce code ne nous sert plus à rien. Juste adaptez le css si vous avez besoin)

- 2) Dans la vue du login, **créer des id** un pour le bouton et un autre pour le form car on va utiliser Ajax et on doit rajouter un événement et créer un objet FormData

```
<form id="formLogin">    // pas d'action ni de méthode, on utilise Axios
.
<button class="btn btn-lg btn-primary" type="submit" id="btnLogin">
```

- 3) Créer un **div** pour afficher les **messages d'erreur**

```
<!-- pour afficher l'erreur -->
<div id="divMessageErreur">
<!-- vide par défaut -->
</div>
```

- 4) **Importer axios et faire un appel ajax** à l'action login **en lui envoyant le form**. La route est ("app_login"). Si vous utilisez un fichier externe .js au lieu de twig vous allez devoir utiliser le **FOSJsRoutingBundle**, mais ici on n'a pas besoin.

Notez qu'on a créé un block **customjs**. On est déjà dans un bloc **content**, car la vue qui inclut le le header.html.twig (index.html.twig) se trouve dans ce bloc déjà. Il n'y a pas de problèmes si on imbrique les blocs. Bien sûr, cette structure est juste un exemple.

Vous devez choisir quoi faire selon ce qu'on reçoit du controller. Ici on affiche un message d'erreur ou on redirige (ici on peut!) vers l'accueil :

```

<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    document.getElementById("btnLogin").addEventListener("click", function (event) {
        event.preventDefault();
        axios({
            url: '{{ path ("app_login") }}',
            method: 'POST',
            headers: { 'Content-Type': 'multipart/form-data' },
            data: new FormData(document.getElementById("formLogin"))
        })
        .then(function (response) {
            // si erreur
            if (response.data.error != undefined) {
                divMessageErreur.innerHTML = response.data.error;
            }
            // si pas d'erreur
            else {
                console.log ('connexion ok login');
                window.location.href = "{{ path ('accueil') }}"
            }
        })
        .catch(function (error) {
            console.log(error);
        });
    });
</script>

```


5) Adapter l'action de login (SecurityController dans le projet). Cette action, dans notre cas, n'est jamais appelée dans l'URL

Si une erreur s'est produite, on envoie le `lastUserName` et l'erreur pour que la vue le traite en js. S'il n'y a pas d'erreur, on envoie uniquement le `lastUserName`. Notez qu'on **ne peut pas rediriger vers une autre action ici car on doit renvoyer une réponse JSON!! (on a appelé avec Axios)** . Si on essaie une redirection, le rendu de la vue correspondante se trouvera dans la réponse du serveur mais elle ne sera pas chargée dans le navigateur.

```
/**
 * @Route("/login/modal", name="app_login")
 */
public function login(AuthenticationUtils $authenticationUtils, Request $req): Response
{
    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();
    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

    $response = new JsonResponse(['lastUsername' => $lastUsername]); // cas de base : pas d'erreur

    // si erreur, on envoie le message. Il faut choisir le message qu'on affiche selon l'erreur
    // ou tout simplement afficher login/pass incorrecte
    if (!is_null($error)) {
        $response = new JsonResponse([
            'error' => 'Utilisateur ou mot de passe incorrectes', // $error->getMessage(), // autrement on envoie tout un objet!
            'lastUsername' => $lastUsername
        ]);
    }

    return $response; // on renvoie la réponse dans tous les cas. Elle sera traitée en JS
}
```

- 6) Dans le Authenticator (FormulaireLoginAuthenticator), modifiez l'action **onAuthenticationSuccess** pour qu'elle redirige vers l'accueil. Dans le template de l'accueil vous pouvez incruster les données de l'utilisateur là où vous voulez.

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    // if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
    //     return new RedirectResponse($targetPath);
    // }
    // redirigez vers login: là on fera reponse JSON qui nous convient.
    // Contrairement à certaines docs, on ne peut pas renvoyer null ni éliminer la méthode
    return new RedirectResponse($this->urlGenerator->generate('accueil'));

    // For example : return new RedirectResponse($this->urlGenerator->
    >generate('some_route'));
    // throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
}
```

- 7) Changer la nav ou le template de base pour afficher l'utilisateur qui vient de se connecter

Ex. de base dans **header.html.twig** :

```
{% if app.user.username is defined %}
logged: {{ app.user.username }}
{% endif %}
33.3.
```

Formulaire d'enregistrement

Le plus simple est de créer une action et une vue pour réaliser l'enregistrement. Mais si vous voulez le rendre modal aussi :

Adaptez le formulaire d'enregistrement est plus simple car on peut incruster l'action du **register** de **RegistrationController** quelque part dans le template. Si l'enregistrement fonctionne, on sera logués et dans l'accueil. Autrement un message d'erreur sera affiché. Il faut faire attention de rajouter l'action dans la vue register.html.twig car il n'a pas d'action par défaut (même situation que dans le formulaire de login dans les sections précédentes).

Il faudra quand-même adapter l'action car, en cas d'erreur, l'action d'enregistrement recharge la vue ! (il faudra le remplacer par ajax etc...)

34. Envoi du mail (Swift Mailer)

Doc: <https://symfony.com/doc/current/email.html>

Doc sur Gmail et cloud: <https://symfony.com/doc/current/email.html#email-using-gmail>

Par défaut, Symfony utilise le module Swift Mailer pour envoyer des mails (**projetFormulaires**).

Exemple : envoi d'un mail en utilisant Swift Mailer :

- 1) Rajoutez **Swift Mailer** à votre projet

```
composer require symfony/swiftmailer-bundle
```

- 2) Le fichier **packages/swiftmailer.yaml** sera mis à jour mais on doit configurer les détails de la connexion dans le fichier **.env**

Le format de la connexion le plus habituel est :

```
MAILER_URL=smtp://localhost:25?encryption=ssl&auth_mode=login&username=&password=
```

Si vous ne disposez pas d'un serveur mail et vous voulez envoyer des emails en utilisant Gmail, Symfony permet de le faire en utilisant ce format :

```
MAILER_URL=gmail://user:password@localhost
```

Exemple :

```
MAILER_URL=gmail://developinterface3:Gaucheret3!@localhost
```

Note : le mot **gmail** n'est pas un protocole en soi. C'est juste un raccourci pour que Swift Mailer utilise le protocole smtp, sécurisation ssl, login comme méthode d'authentification et le serveur smtp.gmail.com

3) Créez un **controller** contenant une action qui envoie le mail et les vues :

contenu_mail.html.twig : le contenu du mail en soi

envoyer_mail.html.twig : la vue à rendre par l'action (style "mail envoyé" ou autre)

```
class MailController extends AbstractController
{
    /**
     * @Route("/mail/envoyer/mail", name="envoyer_mail")
     */
    public function envoyerMail(\Swift_Mailer $mailer)
    {
        $message = (new \Swift_Message ('Hello mail'))
            ->setFrom ('developinterface3@gmail.com')
            ->setTo ('developinterface3@gmail.com')
            ->setBody (
                $this->renderView (
                    'mail/contenu_mail.html.twig'
                ),
                'text/html'
            );
        $mailer->send($message);
        return $this->render('mail/envoyer_mail.html.twig');
    }
}
```

35. Pagination

Exemple pratique : projet **ProjetPaginatorNoWebpack**

1. Installez le module KnpPaginatorBundle

<https://github.com/KnpLabs/KnpPaginatorBundle>

```
composer require knplabs/knp-paginator-bundle
```

2. Copiez le fichier de configuration **knp_paginator.yaml** dans **config/packages**. En principe utilisez la configuration par défaut

```
knp_paginator:
    page_range: 5                # number of links showed in the pagination menu (e.g: y
                                # ou have 10 pages, a page_range of 3, on the 5th page you'll see links to page 4, 5, 6)
    default_options:
        page_name: page         # page query parameter name
        sort_field_name: sort   # sort field query parameter name
        sort_direction_name: direction # sort direction query parameter name
        distinct: true         # ensure distinct results, useful when ORM queries are
                                # using GROUP BY statements
        filter_field_name: filterField # filter field query parameter name
        filter_value_name: filterValue # filter value query parameter name
    template:
        pagination: '@KnpPaginator/Pagination/sliding.html.twig' # sliding pagination contr
        sortable: '@KnpPaginator/Pagination/sortable_link.html.twig' # sort link template
        filtration: '@KnpPaginator/Pagination/filtration.html.twig' # filters template
```

3. Dans une action de votre controller, récupérez une instance du paginator (service). Voici un exemple commenté de son utilisation

```

class ExemplePaginationController extends AbstractController
{
    /**
     * @Route("/exemple/pagination", name="exemple_pagination")
     */
    public function index(PaginatorInterface $paginator, Request $request)
    {

        $livres = $this->getDoctrine()->getRepository(Livre::class)->findAll();

        // Cette méthode est plus rapide que findAll
        // $livres = $this->getDoctrine()->getRepository(Livre::class)-
        >createQueryBuilder('l')->getQuery();

        $numeroPage = $request->query-
        >getInt('page', 1); // 1 par défaut, s'il n'y a pas de page dans l'URL

        $paginationLivres = $paginator->paginate(
            $livres,
            $numeroPage,
            5 // résultats affichés par page
        );
        return $this->render(
            'exemple_pagination/index.html.twig',
            ['paginationLivres' => $paginationLivres]
        );
    }
}

```

4. Voici la vue correspondante :

```

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

{% block title %}Hello ExemplePaginationController!{% endblock %}

{% block stylesheets %}
<link href="{{ asset ('/assets/css/bootstrap.min.css') }}" rel="stylesheet">
{% endblock %}

{% block body %}
{% for livre in paginationLivres %}
<div class="">
    {{ livre.titre }}<br>
    {{ livre.description }}
</div>

```

```
{% endfor %}

{# inclusion de la pagination #}
<div class="paginationLivres">
    {{ knp_pagination_render (paginationLivres) }}
</div>

{% endblock %}

{% block javascripts %}
<script type="module" src="{{ asset ('/assets/js/main.js') }}"></script>
{% endblock %}
```

36. JS et CSS avec Webpack encore

Si vous voulez utiliser du JS et CSS vous pourriez juste créer un dossier dans **public** et inclure vos fichiers **.js** et **.css**, mais la bonne pratique consiste à utiliser **Webpack**. Symfony possède l'extension **Webpack Encore**, qui facilite énormément l'installation et utilisation de Webpack.

On va procéder à installer Webpack dans un projet vide et réaliser quelques exemples. Créez un projet **ExemplesWebpack** et un controller **MainController** (le projet complet est disponible dans GitHub)

Le but de Webpack est de **centraliser la charge de tout notre code JS et CSS dans un seul** (ou éventuellement plusieurs si on le souhaite) **fichier .js**. Webpack permet en plus de compiler, minimiser et découper en morceaux notre code pour optimiser le chargement dans l'application.

Nous allons installer, configurer et utiliser Webpack Encore.

Installation de Webpack Encore et de node_modules

36.1.

1. Installez Webpack Encore dans votre projet

```
composer require encore
```

- Installe et habilite le module **WebpackEncoreBundle**
- Crée le dossier **assets** (à ne pas confondre avec un possible dossier **/public/assets** qu'on aurait pu créer avant d'utiliser Webpack Encore)
- Créer les fichier **assets/js/app.js** qui centralise la charge de tout le code **js** et **css**
- Crée un fichier **webpack.config.js** dans **/config/packages** qui contient la configuration du module

2. Installez les dépendances JS de Webpack Encore

```
yarn install
```

Cette ligne crée le dossier **node_modules** et le rajoute au **./gitignore**

Configurer Webpack Encore

Ouvrez le fichier **webpack.config.js** pour configurer Encore. Vous pouvez personnaliser Encore selon vos besoins :

36.2.

.setOutputPath : emplacement des fichiers compilés

.setPublicPath : le chemin utilisé par le serveur (ex: dans le code des vues) pour accéder l'OutputPath

.addEntry ('app','./assets/js/app.js') : on aura un **entry** pour chaque fichier .js qui regroupe un ensemble de code. Ici on a créé un entry portant le nom "app" qui pointe vers un fichier app.js

Ouvrez le fichier **app.js** et observez qu'on importe le **.css**! (Concrètement on importe le fichier **assets/css/app.css**)

Lancer Webpack

36.3.

Pour compiler les assets une seule fois :

yarn encore dev

Pour lancer un daemon qui recompilera à chaque fois qu'on change le .js ou le .css :

yarn encore dev --watch

Pour créer la version de production :

yarn encore production

Encore compilera le code JS et CSS dans dossier **public/build** qui contiendra un nouveau fichier **app.js** et un **app.css** qui rassembleront tout le contenu JS et CSS (ainsi que les fichiers **manifest.json**, **entrypoints.json**, **runtime.js**). Si on a créé plusieurs entries on aura plusieurs fichiers.

Utiliser le code dans les vues

Pour faciliter l'utilisation de Webpack dans les templates on a des fonctions **Helper**. Vous pouvez inclure ces appels dans vos blocs javascripts et css dans les vues.

Pour css : `{{ encore_entry_link_tags ('app') }}`
Pour js : `{{ encore_entry_script_tags ('app') }}`

La reference 'app' est configurée dans le fichier **entrypoints.json**, qui a été crée à partir de votre fichier **webpack.config.js**. Vous pouvez utiliser un autre nom et, tel qu'on a déjà mentionné, avoir plusieurs **entries** ('app', 'autre', 'main'...)

37. Encore et Bootstrap

Note: la procédure qui suit à été utilisée pour inclure bootstrap dans le projet **ProjetPaginatorWebpack**

Installez bootstrap :

```
yarn add bootstrap --dev
```

Quand on inclut Bootstrap avec une balise SCRIPT, le code attend que jQuery soit une variable global. On change le app.js pour importer bootstrap, qui se trouve dans node_modules :

```
import $ from 'jquery';  
import 'bootstrap';
```

Bootstrap a besoin de **popper.js** :

Installez le avec npm :

```
npm install --save popper.js
```

Rajoutez-le aux modules avec :

```
yarn add popper --dev
```

Pour utiliser le css de bootstrap on doit d'abord l'importer :

```
@import '~bootstrap/dist/css/bootstrap.css';
```

dans le fichier app.css

Le tilde est nécessaire pour référencer un fichier qui se trouve dans le dossier node_modules.

Si vous voulez utiliser le js de bootstrap, vous devez inclure :

```
// pour pouvoir utiliser jQuery et le JS de Bootstrap  
import $ from 'jquery';  
import 'bootstrap';
```

dans app.js. Pour les fonts :

```
yarn add font-awesome --dev
```

Si vous allez utiliser jQuery, vous devez l'installer aussi :

```
yarn add jquery --dev
```

38. Intégration de boutons de paiement Paypal

Si vous voulez utiliser Paypal dans une application en production vous devez comprendre complètement le système de paiements et savoir très bien ce que vous faites :).

La façon la plus simple d'intégrer Paypal dans votre site est d'utiliser les Smart Payment Buttons, car vous devez uniquement utiliser du code javascript et c'est vraiment simple. Paypal fournit en plus un **sandbox** pour que vous puissiez vérifier les paiements du point de vue de l'acheteur et du point de vue du vendeur. Les instructions sont très claires et il faut uniquement adapter le code à vos besoins (ex : calculer le montant à payer d'une transaction, indiquer quoi faire dans le cas de succès ou échec d'un paiement etc...)

Suivez ces instructions pour pouvoir faire un test. Créez d'abord un projet symfony contenant au moins une vue où vous allez insérer le code proposé par Paypal. Vous avez un exemple dans **ProjetPaypalSmartButtons**.

Vous devez :

1. Ouvrir compte Business en Paypal (remplir toutes les données car c'est un vrai compte!)
2. Cliquer sur Tools → All tools
3. Cliquez à gauche sur Integrate Paypal
4. Cliquez sur Developer Site
5. Cliquez sur Checkout
6. Cliquez sur Smart Payment Buttons Overview et comprenez le système
7. Suivez le tuto complet de Smart Payment Buttons
8. Cliquez en bas de la page sur "Add a Smart Payment Buttons integration to your website"
9. Suivez les instructions. Conseil : quand vous allez devoir cliquer dans **Log into the Developer Dashboard**, faites-le dans une nouvelle fenêtre (clic droit). Le code proposé par Paypal sera intégré dans la vue (voir exemple dans le projet)

A effacer :

Pour JS :

- Créer le fichier assets\get_nice_message.js

```
module.exports = function (exclamationCount){  
  return "j'aime bien l'omelette". "!" . repeat (exclamationCount);  
}
```

(Node)

ou

```
export default function (exclamationCount){  
  return "j'aime bien l'omelette". "!" . repeat (exclamationCount);  
}
```

(ECMA)

- Dans app.js, importez le module :

```
const getNiceMessage = require ('./get_nice_message');  
(Node)
```

ou

```
import getNiceMessage from './get_nice_message'  
(ECMA)
```

Changez aussi require pour import ()

- Utilisez la fonction dans app.js pour la tester

// any CSS you import will output into a single css file (app.css in this case)

```
import './css/app.css';
```

```
const getNiceMessage = require ('./get_nice_message');
```

// Need jQuery? Install it with "yarn add jquery", then uncomment to import it.

```
// import $ from 'jquery';
```

```
console.log(getNiceMessage(5));
```

```
////////////////////////////////////
```

```
// Installer de librairies avec YARN
```

```
////////////////////////////////////
```

jQuery

```
yarn add jquery --dev
```

et puis (app.js)

```
import $ from 'jquery';
```

(pas besoin de "."/" car de cette manière Webpack le cherchera dans node_modules (dossier)

Bootstrap

Quand on inclut Bootstrap avec une balise SCRIPT, le code attend que jQuery soit une variable globale. Pour inclure bootstrap avec Webpack on doit utiliser yarn :

```
yarn add bootstrap --dev
```

On change le app.js :

```
import $ from 'jquery';  
import 'bootstrap';
```

On doit arranger la dépendance de popper.js :

```
npm install --save popper.js  
(yarn add popper --dev)
```

Importer le bootstrap.css. La tilde est nécessaire pour référencer un node module à l'intérieur d'un fichier css.

```
@import '~bootstrap/dist/css/bootstrap.css';
```

Pour les fonts :

```
yarn add font-awesome --dev
```

