

## 1) Frameworks et éléments sur Symfony

### a) définition d'un framework

Un framework (cadre) de développement est un ensemble d'outils logiciels permettant de bâtir l'architecture d'une application.

La programmation au sein d'un framework incite voire impose de programmer selon un ou plusieurs patrons de conception, *design patterns* en anglais.

Un des patrons de conception le plus utilisé est l'architecture MVC (*Modèle Vue Contrôleur*).

On trouve notamment :

- pour PHP le framework Symfony : développement back-end
- pour JavaScript le framework Angular : développement front-end
- pour Java le framework Hibernate : développement lourd

De manière typique, un framework est une couche logicielle supplémentaire au dessus notamment d'un langage.

### b) objectifs d'un framework PHP

- > compartimenter une application
- > pas d'appel aux fonctions natives de PHP : à la place on passe par des éléments du framework ce qui permet une meilleure évolutivité et portabilité
- > faciliter les tests unitaires et fonctionnels

### c) choix de Symfony

Symfony présente de nombreux avantages qui expliquent son choix :

- il offre une grande modularité via ses ***bundles*** : paquets de Symfony,
- il est très documenté grâce à une grande communauté de développeurs,
- il permet une bonne personnalisation de sa configuration.

### d) versions de Symfony et PHP associées

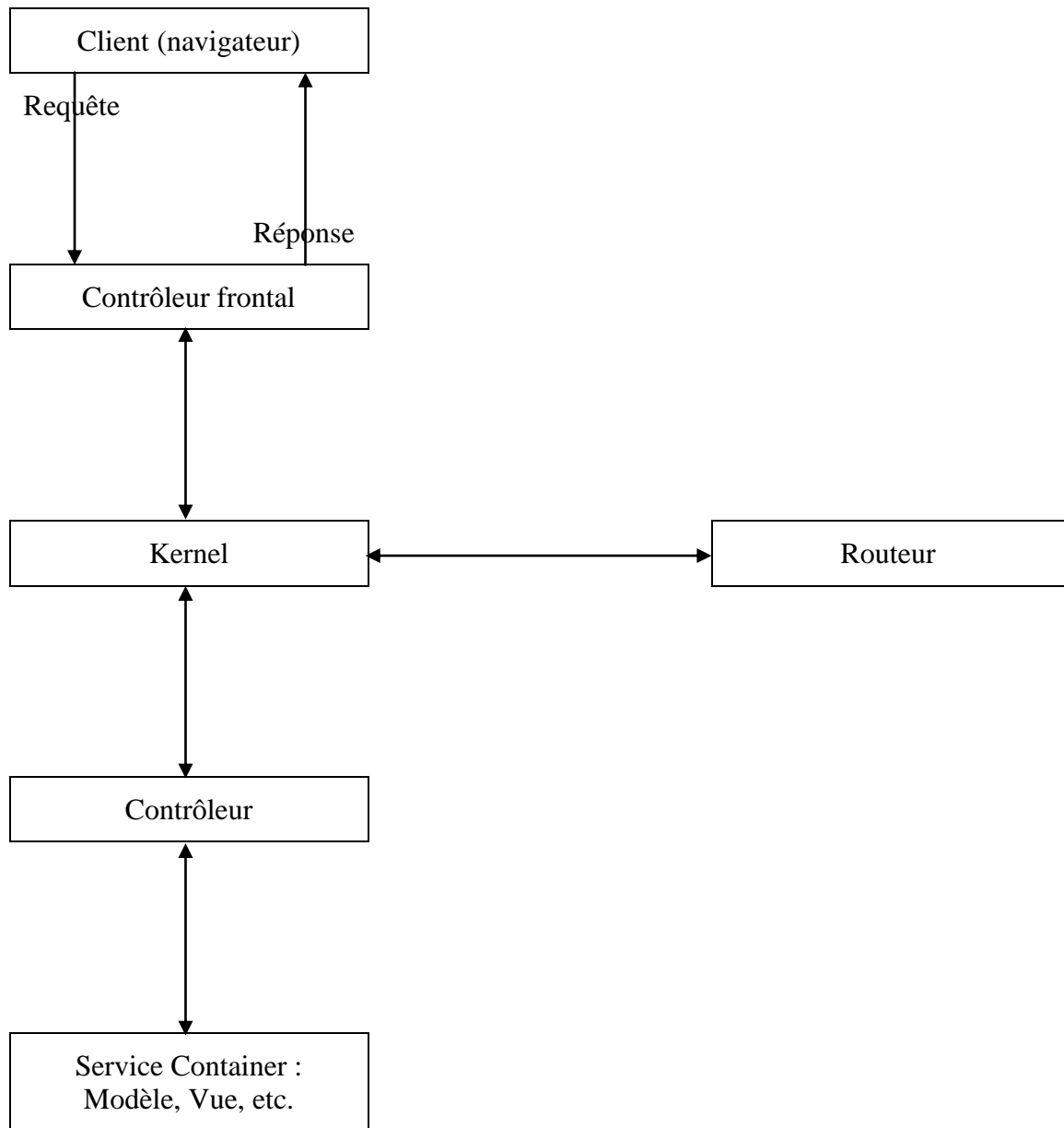
Symfony propose 6 versions.

Le tableau suivant fournit pour chaque version la version PHP minimum.

<i>Version de Symfony</i>	<i>Version PHP minimum</i>
1.x	5.2.4
2.x	5.3.3
3.x	5.5.9
4.x	7.1
5.x	7.2.5
6.x	8

Nous verrons ici plus précisément la version 5 de Symfony, sachant que les versions 5 et 6 se ressemblent.

e) architecture de base de Symfony



On voit clairement que l'esprit du framework Symfony est basé sur l'architecture MVC.

Deux éléments sont propres à Symfony :

- le Kernel (noyau) qui est donc l'élément central du framework : il va notamment être capable d'appeler le bon contrôleur pour satisfaire la requête en tant que demande de service du client (typiquement un navigateur),
- le Service Container qui va contenir un ensemble de services.  
On y retrouve notamment les services Modèle et Vue.

Voyons à présent le cheminement d'une requête jusqu'à sa réponse au client.

Le client, typiquement un navigateur, fait une requête HTTP via une URL, ce qui va invoquer une certaine page à charger depuis le serveur Apache avec donc en plus cette couche Symfony.

C'est le contrôleur frontal qui reçoit comme toujours cette requête qu'il transmet au Kernel de Symfony.

Le Kernel interroge un élément appelé routeur pour savoir quel contrôleur il doit invoquer.

Le Kernel exécute le contrôleur.

Ce contrôleur va invoquer les bons services notamment le Modèle ou la Vue, disponibles dans le Service Container.

Le contrôleur est ensuite capable de générer une réponse HTTP qu'il retourne au Kernel, réponse qui est fournie au final au client (navigateur).

## 2) Installation et configuration de Symfony

### a) prérequis

Symfony est un framework PHP.

Cela signifie que PHP doit être installé préalablement.

Comme on a dans la plupart des cas besoin d'une base de données, un SGBDR comme MySQL ou MariaDB doit également être en fonctionnement.

La solution, connue depuis longtemps, est d'installer un package comme XAMPP, LAMP, MAMP ou WAMP. Ici, nous prendrons le package WAMP.

Pour pouvoir tirer parti de Symfony 5, il faut une version de PHP supérieure ou égale à 7.2.5.

#### Remarque :

Le service Web Apache n'a pas besoin d'être démarré.

Cependant, on le lancera car on en aura besoin pour certains tests.

### b) installation de Composer, de Git et vérifications

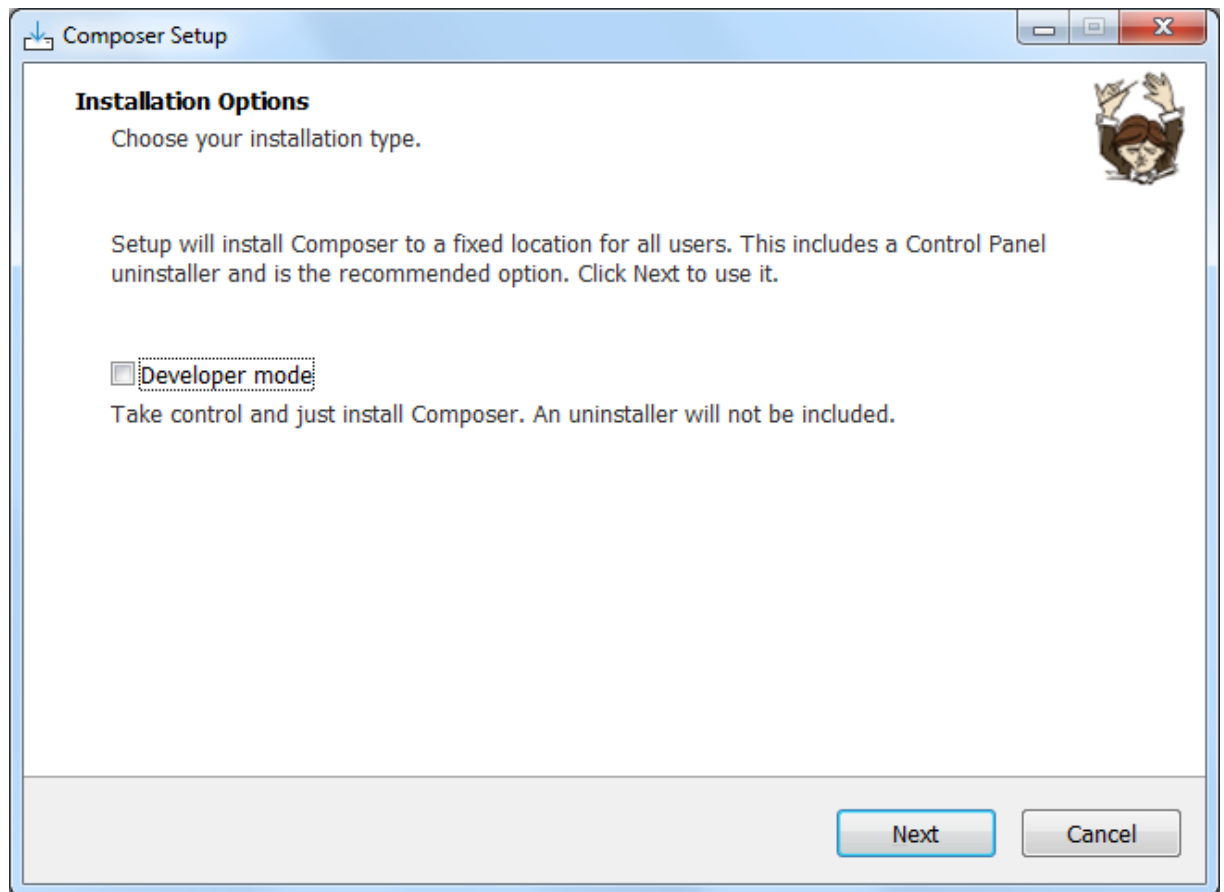
#### b-1) installation de Composer

**Composer** est un gestionnaire de dépendances Open Source. Il permet d'intégrer des bibliothèques (librairies) aux éléments déjà existants en l'occurrence ici PHP, notamment le framework Symfony en tant que couche logicielle au dessus du serveur Web Apache.

L'installateur est téléchargeable depuis l'adresse suivante [getcomposer.org/download](https://getcomposer.org/download) mais on peut le récupérer aussi depuis le NAS : fichier **Composer-Setup.exe**.

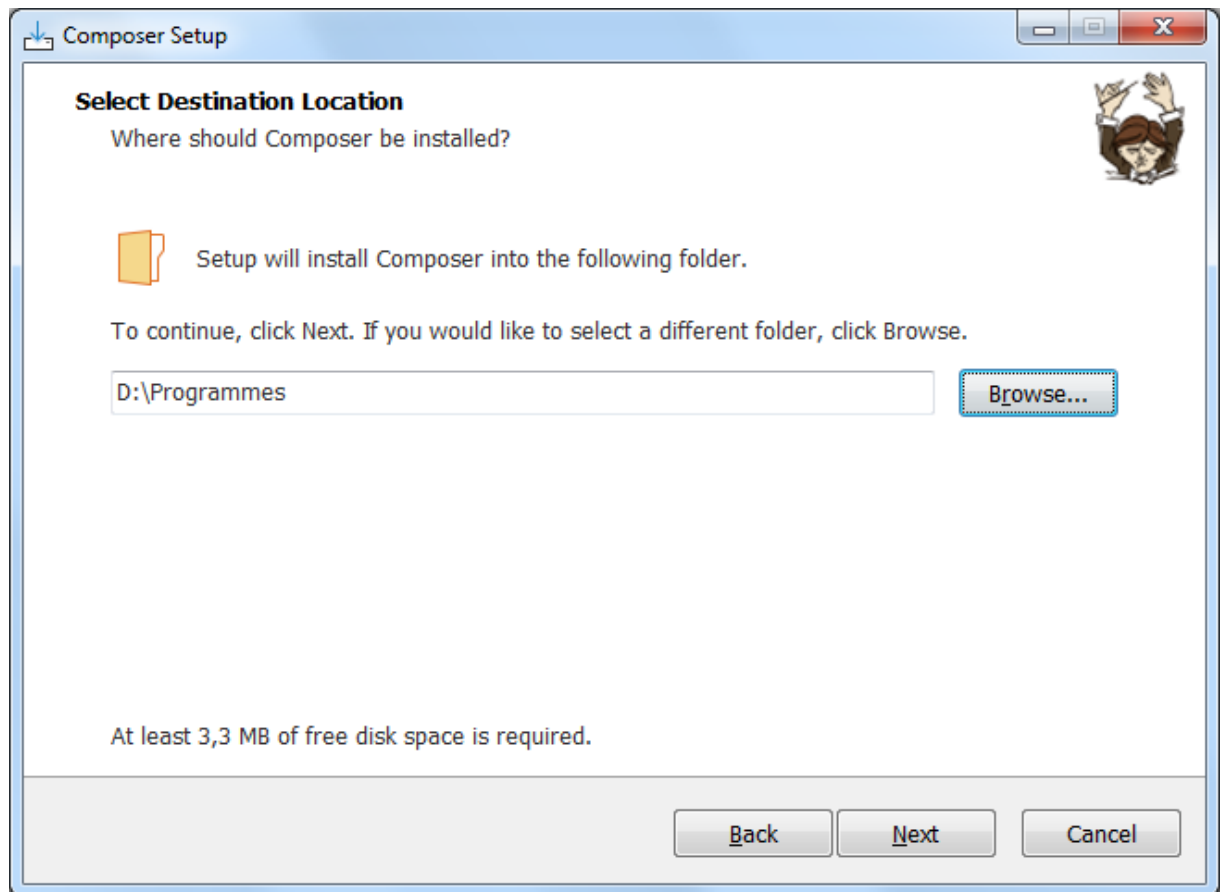
➔ exécuter le fichier donc de nom **Composer-Setup.exe**

Un assistant se lance ainsi :



➔ cliquer sur *Developer mode* puis sur *Next*

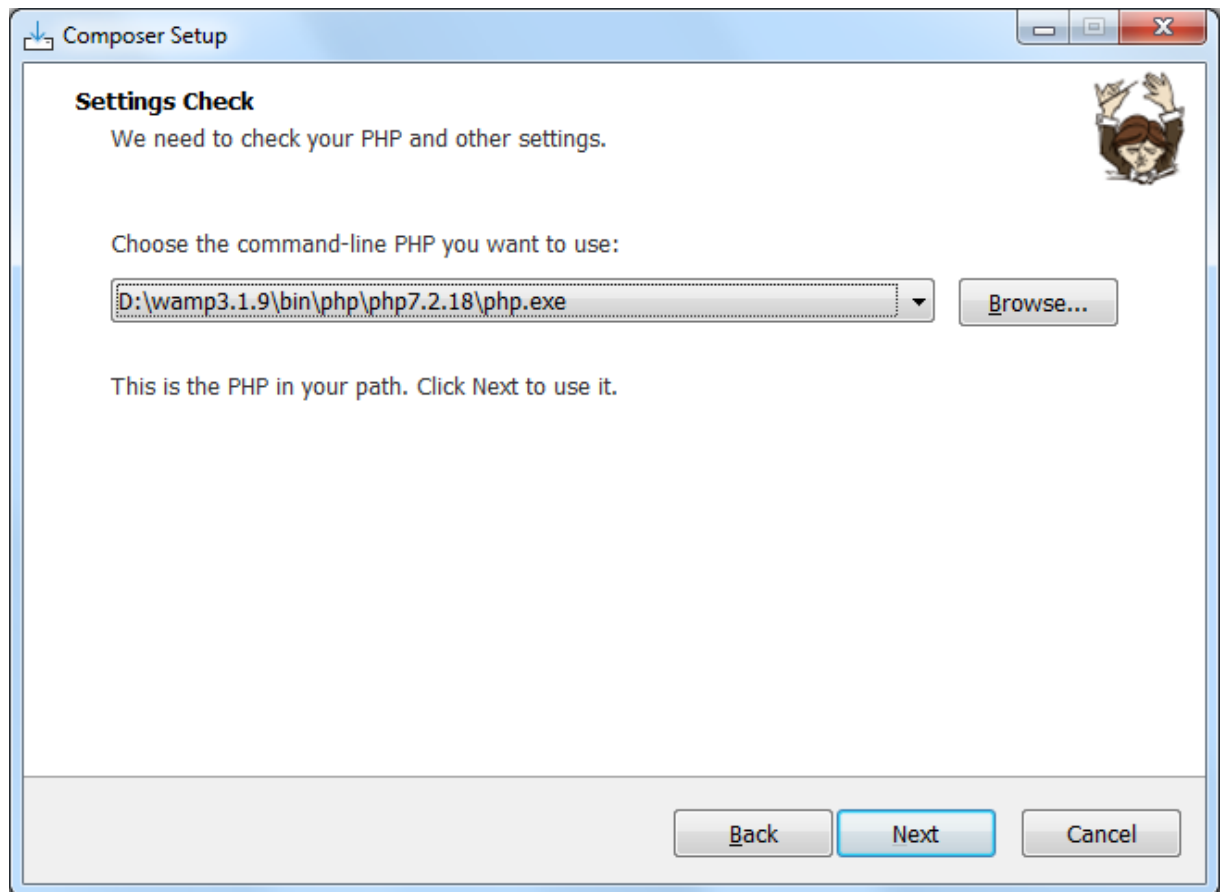
L'écran suivant apparaît que nous avons complété :



L'assistant demande l'endroit où doit être installé **Composer**.

➔ modifier éventuellement le dossier, on doit choisir un dossier où a tous les droits par exemple votre dossier d'utilisateur, ici on choisit **D:\Programmes**, et cliquer sur *Next*

L'écran suivant apparaît :

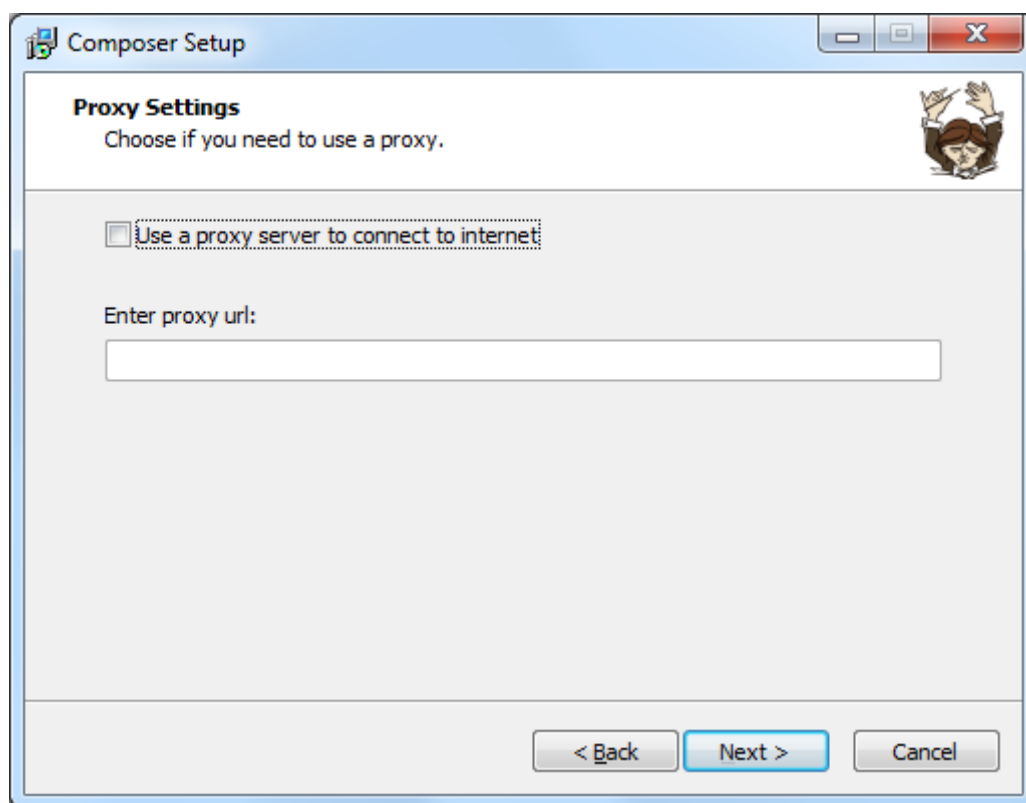


L'assistant demande de sélectionner l'exécutable PHP.

➔ sélectionner une version 7 de PHP (au moins 7.2.5), puis cliquer sur *Next*

L'écran suivant apparaît :

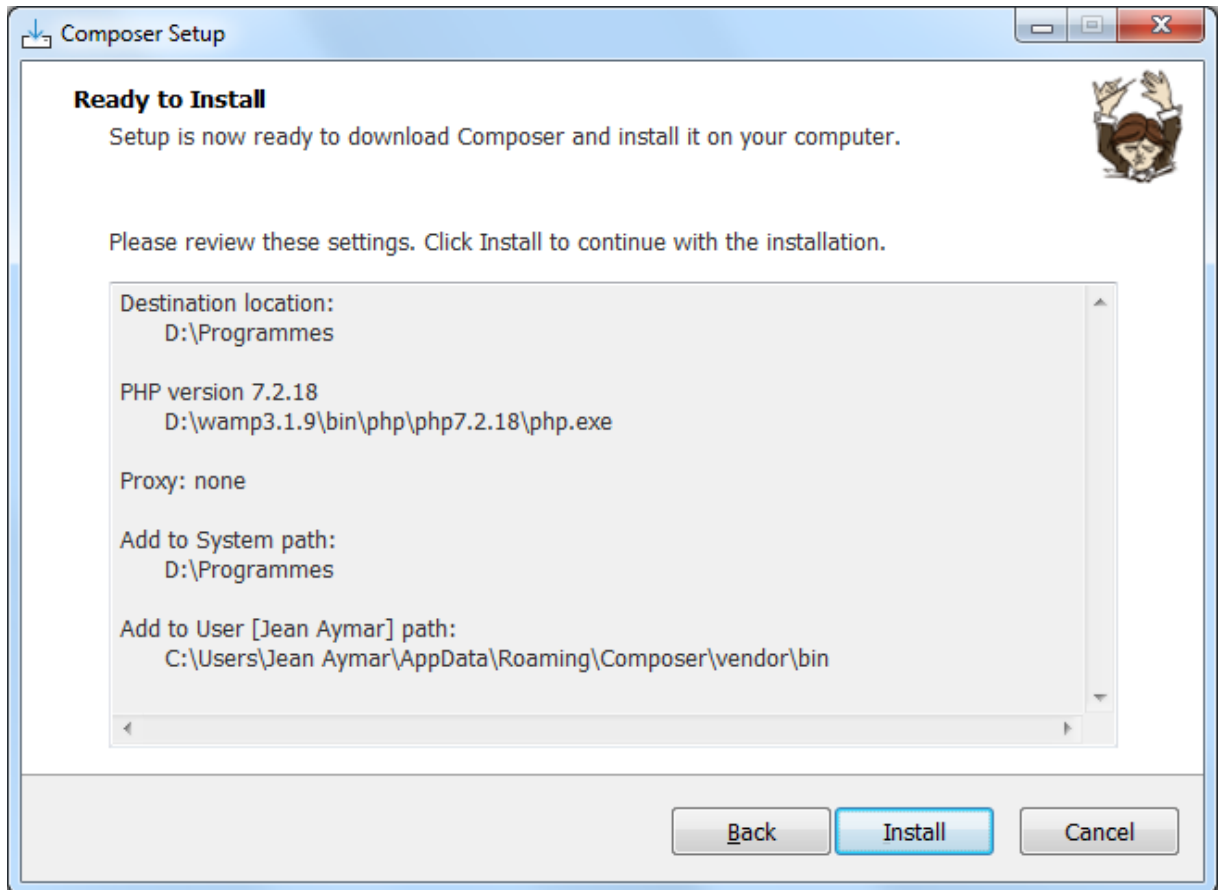




L'assistant demande de fournir l'URL d'un éventuel proxy.

➔ saisir éventuellement cette URL puis cliquer sur *Next*

L'écran suivant apparaît :



L'assistant est prêt pour l'installation.

➔ cliquer sur *Install* pour lancer l'installation

A la fin, l'écran suivant indiquant que l'installation de Composer s'est bien achevée apparaît :



➔ cliquer sur *Finish*

## b-2) installation de **Git**

Pour utiliser Symfony, il est vivement recommandé d'installer le logiciel **Git**, sinon on ne pourra pas lancer certaines commandes.

**Git** est un logiciel de gestion de versions.

Il va permettre de gérer un dépôt, *repository* en anglais, où seront stockés les différents fichiers d'un projet avec leur historique.

Chaque développeur du projet aura sur son propre ordinateur une copie de ce dépôt.

Parallèlement, il existe **GitHub** qui est un service web d'hébergement et de gestion de développement de logiciels, exploitant ce logiciel **Git**.

L'installateur de **Git** est téléchargeable depuis l'adresse suivante [git-scm.com/downloads](https://git-scm.com/downloads) mais on peut le récupérer aussi depuis le NAS : fichier dont la syntaxe est **Git-version.exe**.

➔ exécuter le fichier donc de nom **Git-version.exe**

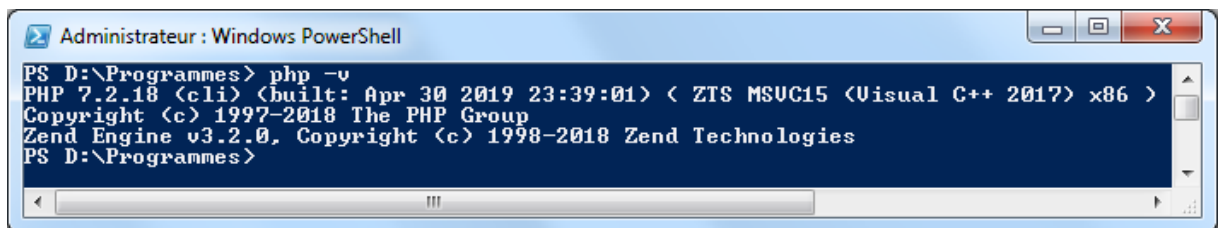
### b-3) vérifications

Les commandes seront à taper depuis une fenêtre console par exemple sous l'invite de commandes (**cmd.exe**) ou PowerShell *qu'on doit exécuter en tant qu'administrateur*.

On se placera dans le dossier d'installation ici **D:\Programmes**.

Tout d'abord pour vérifier quelle version de PHP est utilisée la commande est :  
**php -v**

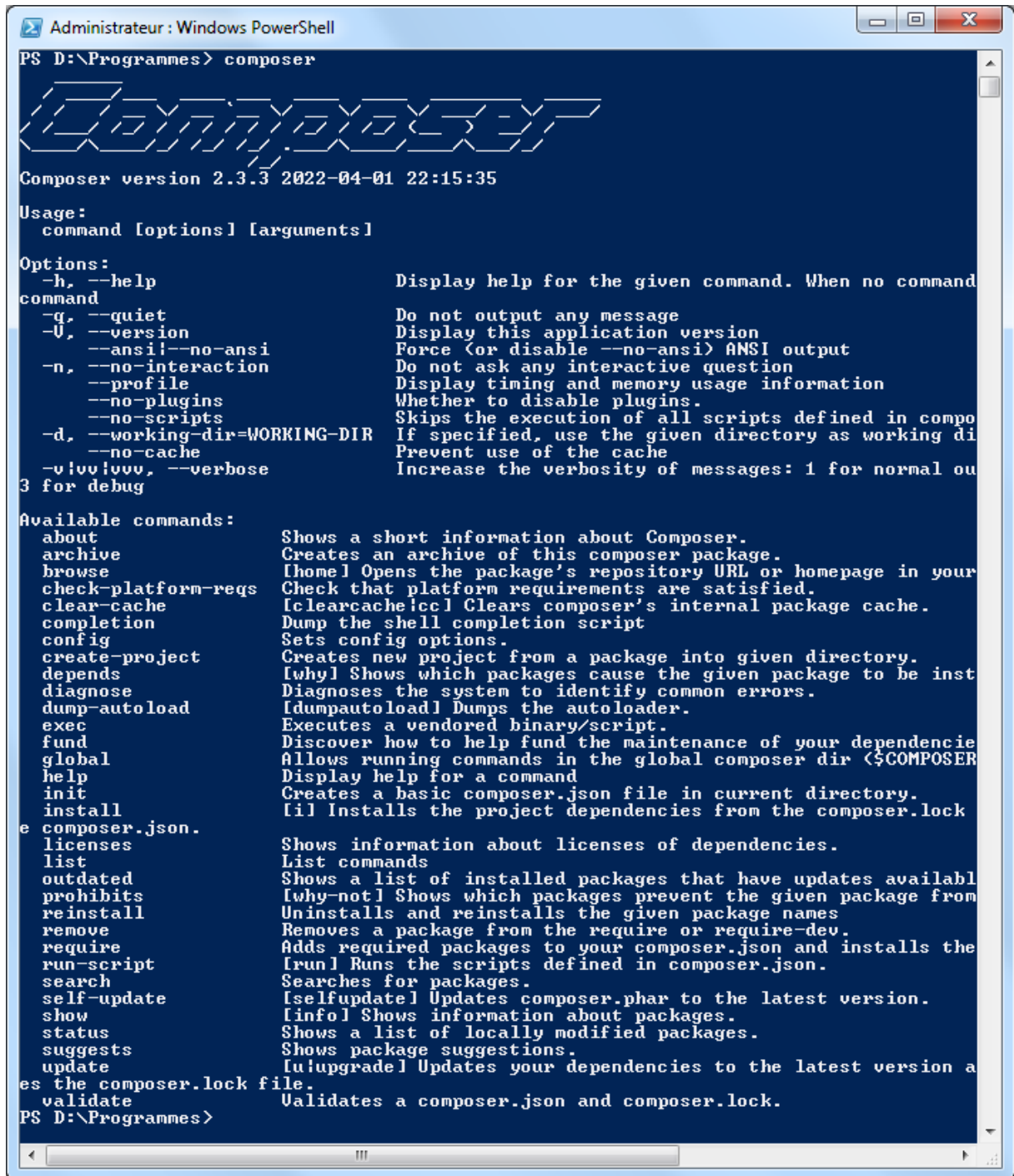
On obtient :



```
Administrateur : Windows PowerShell
PS D:\Programmes> php -v
PHP 7.2.18 (cli) (built: Apr 30 2019 23:39:01) < ZTS MSUC15 (Visual C++ 2017) x86 >
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
PS D:\Programmes>
```

Ensuite on vérifie que **composer** est bien installé avec la commande :  
**composer**

Les différentes commandes disponibles relatives à Composer doivent s'afficher ainsi :



```

Administrateur : Windows PowerShell
PS D:\Programmes> composer

Composer version 2.3.3 2022-04-01 22:15:35

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display help for the given command. When no command
                             command
  -q, --quiet               Do not output any message
  -V, --version             Display this application version
                             Force (or disable --no-ansi) ANSI output
  -n, --no-interaction      Do not ask any interactive question
  --profile                Display timing and memory usage information
  --no-plugins              Whether to disable plugins.
  --no-scripts              Skips the execution of all scripts defined in compo
                             If specified, use the given directory as working di
                             Prevent use of the cache
  -v|vv|vvv, --verbose     Increase the verbosity of messages: 1 for normal ou
                             3 for debug

Available commands:
  about                    Shows a short information about Composer.
  archive                  Creates an archive of this composer package.
  browse                   [home] Opens the package's repository URL or homepage in your
  check-platform-reqs      Check that platform requirements are satisfied.
  clear-cache              [clearcache|cc] Clears composer's internal package cache.
  completion               Dump the shell completion script
  config                   Sets config options.
  create-project            Creates new project from a package into given directory.
  depends                  [why] Shows which packages cause the given package to be inst
  diagnose                 Diagnoses the system to identify common errors.
  dump-autoload            [dumpautoload] Dumps the autoloader.
  exec                     Executes a vendored binary/script.
  fund                     Discover how to help fund the maintenance of your dependencie
  global                   Allows running commands in the global composer dir ($COMPOSER
  help                     Display help for a command
  init                     Creates a basic composer.json file in current directory.
  install                  [i] Installs the project dependencies from the composer.lock
  composer.json.
  licenses                 Shows information about licenses of dependencies.
  list                     List commands
  outdated                 Shows a list of installed packages that have updates availabl
  prohibits                [why-not] Shows which packages prevent the given package from
  reinstall                Uninstalls and reinstalls the given package names
  remove                   Removes a package from the require or require-dev.
  require                  Adds required packages to your composer.json and installs the
  run-script               [run] Runs the scripts defined in composer.json.
  search                   Searches for packages.
  self-update              [selfupdate] Updates composer.phar to the latest version.
  show                     [info] Shows information about packages.
  status                   Shows a list of locally modified packages.
  suggests                 Shows package suggestions.
  update                   [u|upgradel] Updates your dependencies to the latest version a
  es the composer.lock file.
  validate                 Validates a composer.json and composer.lock.
PS D:\Programmes>

```

A présent on va installer Symfony et créer un projet Web Symfony.

### c) installation de Symfony et création d'un projet Web

L'exécutable est téléchargeable depuis l'adresse suivante [symfony.com/download](https://symfony.com/download) mais on peut le récupérer aussi depuis le NAS : fichier **symfony.exe**.

➔ mettre ce fichier **symfony.exe** dans votre dossier de départ ici **D:\Programmes**

➔ taper la commande **symfony**

On obtient :

```

PS D:\Programmes> symfony
Symfony CLI version 5.4.5 (c) 2017-2022 Symfony SAS #StandWithUkraine Support Ukrain
Symfony CLI helps developers manage projects, from local code to remote infrastru

These are common commands used in various situations:

Work on a project locally

  new                                Create a new Symfon
  server:start                       Run a local web ser
  server:stop                        Stop the local web
  security:check                    Check security issu
  composer                          Runs Composer witho
  console                          Runs the Symfony Co

ect
  php, pecl, pear, php-fpm, php-cgi, php-config, phpdbg, phpize  Runs the named bina

Manage a project on Cloud

  init                                Initialize a new project using templates
  cloud:domains                      Get a list of all domains
  cloud:branch                       Branch an environment
  cloud:environments                 Get a list of environments
  cloud:push                         Push code to an environment
  cloud:ssh                          SSH to the current environment
  cloud:projects                    Get a list of all active projects
  cloud:tunnel:open                 Open SSH tunnels to an app's relationships
  cloud:user:add                    Add a user to the project
  cloud:variables                   List variables

Show all commands with symfony.exe help.
Get help for a specific command with symfony.exe help COMMAND.
PS D:\Programmes>

```

On trouve le client Symfony CLI qui va nous permettre de taper différentes commandes relatives à Symfony.

Ces différentes commandes s'affichent.

Parmi ces commandes, on trouve **new** qui permet de créer un projet Symfony.

Notre application Web finale permettra de gérer l'encadrement de missions par des employés. Le projet correspondant sera appelé **encmis**.

La syntaxe de la commande pour créer un projet Web Symfony est :  
`symfony new --full nomProjet`

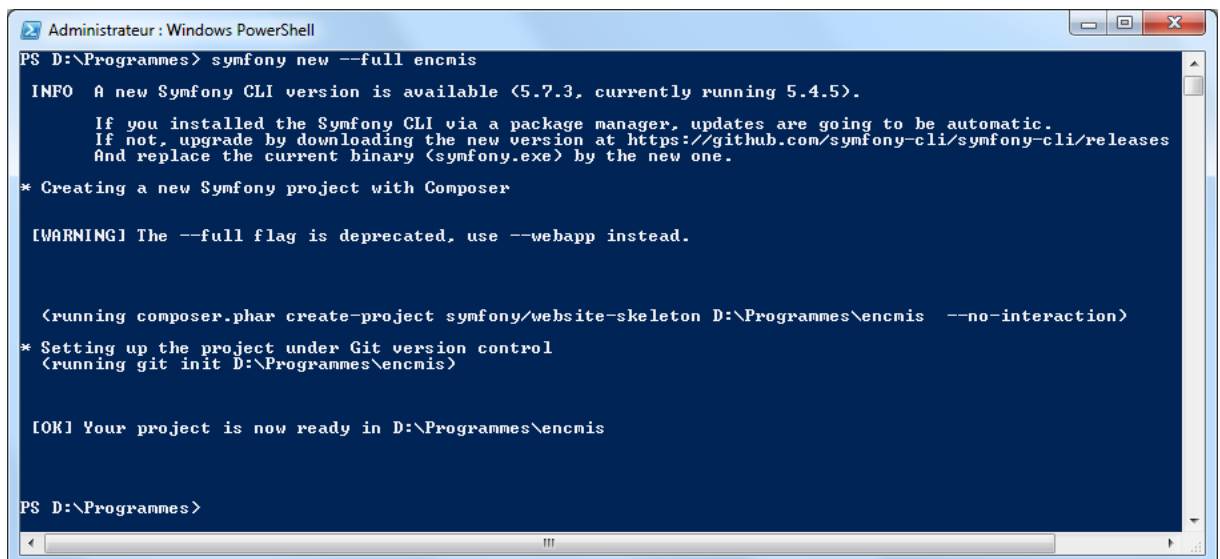
*Ne pas oublier l'espace avant --full.*

#### Remarque :

A la place de `--full`, on devrait mettre `--webapp` d'après l'installateur mais cela ne fonctionne pas sur certaines versions de PHP.

➔ taper donc pour notre cas `symfony new --full encmis`

Voici ce que cela donne dans la console (la commande risque d'être longue à s'exécuter car Symfony va générer beaucoup d'éléments) :



```

Administrateur : Windows PowerShell
PS D:\Programmes> symfony new --full encmis
INFO  A new Symfony CLI version is available (5.7.3, currently running 5.4.5).
       If you installed the Symfony CLI via a package manager, updates are going to be automatic.
       If not, upgrade by downloading the new version at https://github.com/symfony-cli/symfony-cli/releases
       And replace the current binary (symfony.exe) by the new one.

* Creating a new Symfony project with Composer

[WARNING] The --full flag is deprecated, use --webapp instead.

<running composer.phar create-project symfony/website-skeleton D:\Programmes\encmis --no-interaction>
* Setting up the project under Git version control
  <running git init D:\Programmes\encmis>

[OK] Your project is now ready in D:\Programmes\encmis

PS D:\Programmes>

```

Un dossier est créé qui porte le nom du projet ici **encmis**, dans notre exemple au niveau du dossier **D:\Programmes** avec les paquets Symfony et les dépendances associées, concernant notamment le moteur de templates Twig pour les vues et la librairie Doctrine pour la gestion de bases de données.

➔ aller donc dans ce dossier **encmis**

Voici ce que donne le contenu :

```

Administrateur : Windows PowerShell
PS D:\Programmes\encmis> dir

Répertoire : D:\Programmes\encmis

Mode                LastWriteTime         Length Name
----                -
d-----         04/04/2022        16:42         bin
d-----         04/04/2022        16:42         config
d-----         04/04/2022        16:42     migrations
d-----         04/04/2022        16:42         public
d-----         04/04/2022        16:42         src
d-----         04/04/2022        16:42     templates
d-----         04/04/2022        16:42         tests
d-----         04/04/2022        16:42    translations
d-----         04/04/2022        16:43         var
d-----         04/04/2022        16:42        vendor
-a---         04/04/2022        16:42         1533 .env
-a---         04/04/2022        16:42         215 .env.test
-a---         04/04/2022        16:42         375 .gitignore
-a---         04/04/2022        16:44        3075 composer.json
-a---         04/04/2022        16:44     366853 composer.lock
-a---         04/04/2022        16:42         247 docker-compose.override.yml
-a---         04/04/2022        16:42         717 docker-compose.yml
-a---         04/04/2022        16:42        1367 phpunit.xml.dist
-a---         04/04/2022        16:43        14501 symfony.lock

PS D:\Programmes\encmis>

```

#### d) démarrage du serveur Web de Symfony et test dans le navigateur

Pour lancer le service Web du framework Symfony relativement au projet créé, il faut lancer la commande suivante depuis le dossier du projet :

**symfony serve**

Cela donne :



```

PS D:\Programmes\encmis> symfony serve

[WARNING] run "symfony.exe server:ca:install" first if you want to run the web
server with TLS support, or use "--p12" or "--no-tls" to avoid this
warning

Tailing PHP-CGI log file <C:\Users\Jean Aymar\.symfony5\log\6a7881bfee6f09c60cf92
55a33ea6a41ec5e854698.log>
Tailing Web Server log file <C:\Users\Jean Aymar\.symfony5\log\6a7881bfee6f09c60c

[WARNING] The local web server is optimized for local development and MUST nev
er be used in a production setup.

[OK] Web server listening
The Web server is using PHP CGI 7.2.18
http://127.0.0.1:8000

[Web Server ] Apr  4 16:46:52 !DEBUG ! PHP   Reloading PHP versions
[Web Server ] Apr  4 16:46:52 !DEBUG ! PHP   Using PHP version 7.2.18 <from def
[Web Server ] Apr  4 16:46:52 !INFO  ! PHP   listening path="D:\wamp3.1.9\bin
.18" port=50533

```

Le serveur Web local est démarré : adresse **127.0.0.1** (localhost).

Comme indiqué, Symfony utilise le port **8000**.

Pour arrêter le serveur, le raccourci est <Ctrl>+<C> comme pour toute commande.

A présent, allons dans le navigateur et tapons donc :

**127.0.0.1:8000**

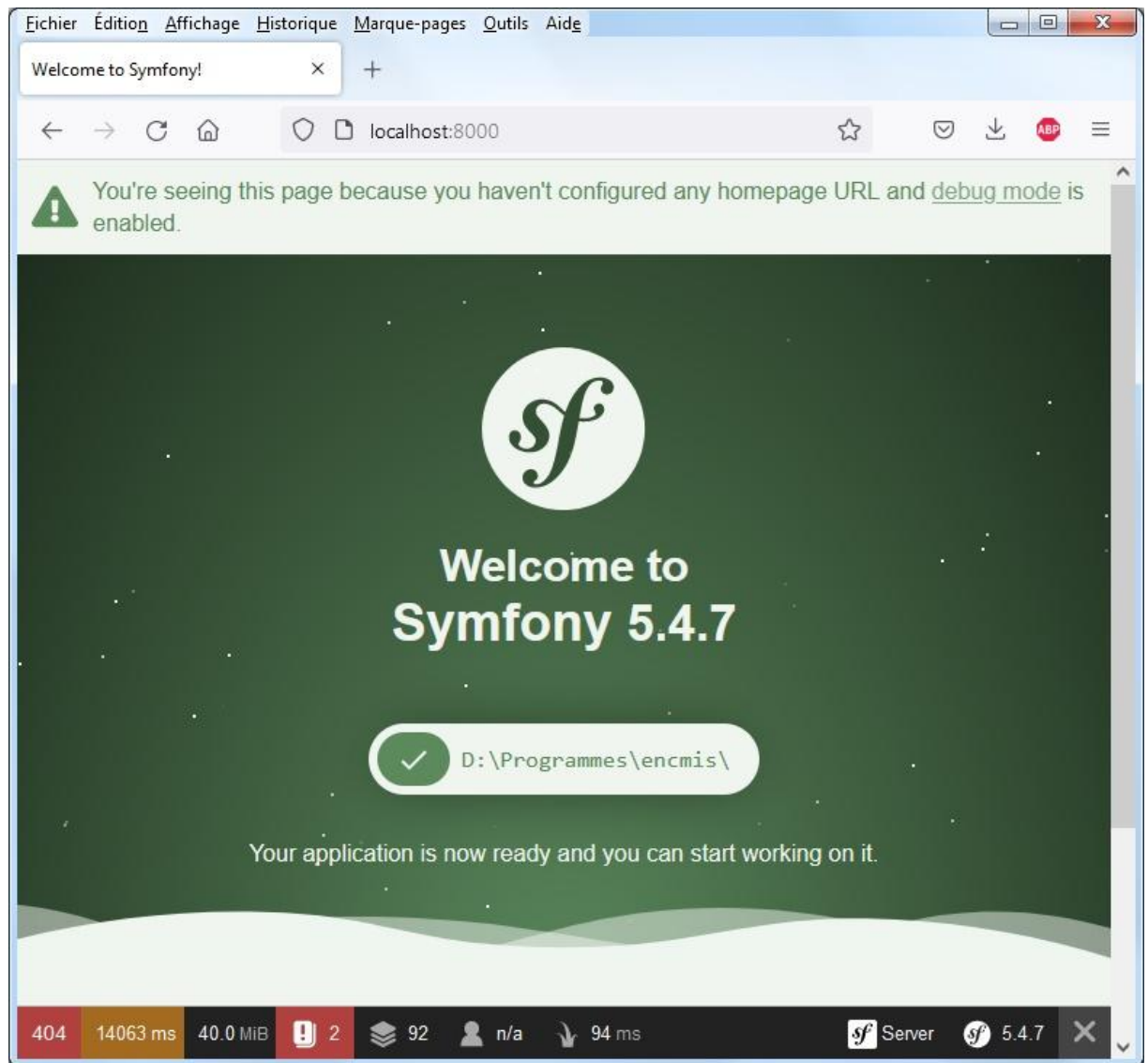
(raccourci de **http://127.0.0.1:8000**)

ou

**localhost:8000**

(raccourci de **http://localhost:8000**)

Cela donne :



Le projet Web est bien accessible depuis le navigateur.

A noter qu'un point d'exclamation en rouge apparaît dans la barre d'état en bas avec le nombre 2 : cela signifie qu'il y a deux alertes dont au moins une erreur (provenant d'un problème de route). Nous y reviendrons...

### e) création d'un domaine local : pour information

De manière professionnelle, on crée un nom de domaine pour le site web en production.

Le nom choisi sera ici **encmis.lan** (**encmis** pour notre site d'encadrement de missions, par des employés).

Cette création de domaine se fait en modifiant le fichier **hosts** se trouvant sous Windows dans le dossier **C:\Windows\System32\Drivers\etc** et sous Linux dans le dossier **/etc**.

On retrouve notamment la ligne :

**127.0.0.1      localhost**

qui permet de taper dans la barre d'adresse **localhost** en plus de l'adresse IP de la machine elle-même **127.0.0.1** (adresse de **loopback**).

On va rajouter pour notre domaine la ligne suivante :

**127.0.0.1      encmis.lan**

A présent, vous pouvez saisir dans la barre d'adresses :

**encmis.lan**

(raccourci de **http://encmis.lan**)

pour vérifier que votre serveur Web apparaît bien.

Et une fois démarré Symfony, vous pouvez corrélativement taper dans la barre d'adresses :

**encmis.lan:8000**

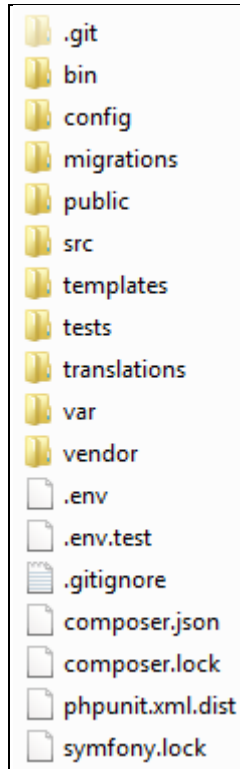
(raccourci de **http://encmis.lan:8000**)

pour vérifier que Symfony se lance bien.

A présent, voyons les éléments générés lors de la création d'un projet Symfony.

## f) structure d'un projet Symfony

Le contenu du dossier du projet ici **enemis** donne :



Voyons les éléments essentiels.

### f-1) dossier **bin**

Ce dossier contient les fichiers exécutables utiles pour le projet notamment la console Symfony.

### f-2) dossier **config**

Ce dossier contient les fichiers de configuration de l'application concernant le framework mais aussi les dépendances (notamment Twig pour les templates et Doctrine pour les bases de données) et les routes.

### f-3) dossier **public**

Il contient au départ le seul fichier **index.php** : contrôleur frontal qui va traiter toutes les requêtes utilisateur. *Symfony est donc basé plus précisément sur MVC2.*

C'est ce fichier qui est directement accessible depuis le navigateur de l'utilisateur, dans notre cas via l'adresse **encmis.lan:8000** ou **localhost:8000** si on teste en local.

Il va stocker toutes les ressources publiques accessibles aux utilisateurs.

### f-4) dossiers **src** et **templates**

Le dossier **src** contient tous les fichiers de l'application ainsi que le noyau : fichier **Kernel.php**.

Symfony étant basé sur l'architecture MVC, on retrouve le sous-dossier **Entity** pour la couche *Modèle* et le sous-dossier **Controller** pour la couche *Contrôleur*.

Pour la couche *Vue*, tout est stocké dans le dossier **templates**.

Les fichiers HTML auront pour extension **.html.twig** : Twig est un moteur de templates (on emploie aussi à la place de template le mot gabarit ou disposition, *layout* en anglais).

### f-5) dossier **tests**

Il contient tous les fichiers relatifs aux tests que ce soit unitaires ou d'intégration.

### f-6) dossier **translations**

Il contient tous les fichiers pour tout ce qui est traduction du site.

### f-7) dossier **var**

Il contient notamment les fichiers de cache dans le sous-dossier **cache** et les fichiers de log (journaux) dans le sous-dossier **log**.

f-8) dossier **vendor**

Il contient toutes les librairies dont notre application (projet) dépend : dépendances du projet.

Elles ont été installées au départ.

On va notamment retrouver le sous-dossier **symfony** pour le framework proprement dit, le sous-dossier **twig** pour le moteur de templates (vues) et le sous-dossier **doctrine** pour le moteur de bases de données.

Il y aussi notamment le fichier **autoload.php** qui est le chargeur de dépendances.

f-9) fichier **.env**

Il contient les variables d'environnement pour la configuration de divers éléments. Par exemple la variable d'environnement **DATABASE\_URL** est relative à la base de données utilisée dans le cadre du projet.

f-10) fichier **composer.json**

Il contient différentes informations en particulier la liste des librairies utilisées.

g) couches de l'architecture MVC et dossiers correspondants

Le tableau suivant rappelle pour chaque couche de l'architecture MVC/MVC2 le dossier correspondant dans un projet Symfony.

<i>Couche</i>	<i>Dossier projet Symfony</i>
Modèle	<b>src/Entity</b>
Vue	<b>templates</b>
Contrôleur	<b>src/Controller</b>

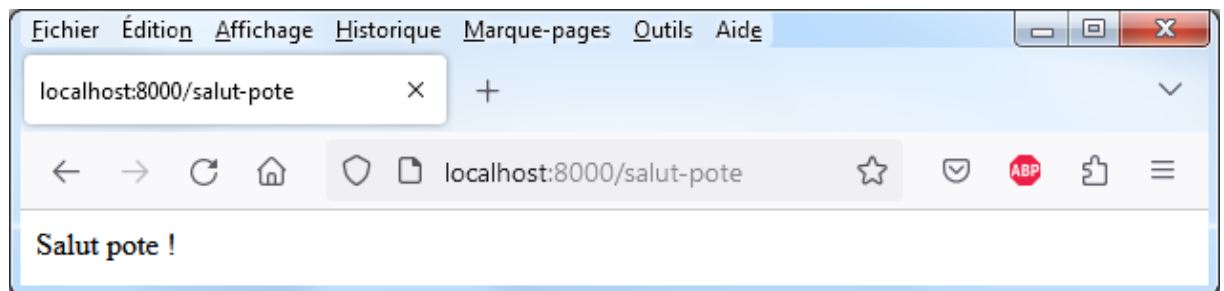
### 3) Un premier exemple de mise en place

Nous allons afficher une première page via Symfony afin de voir les différentes étapes inhérentes au framework.

L'affichage d'une page dans le navigateur va nécessiter deux étapes :

- création d'un contrôleur avec route et méthode à lancer,
- création d'un template.

Voici la page qu'on doit afficher dans un premier temps : on utilisera ici **localhost** comme nom de domaine car on veut tester en local :



Dans l'URL, on voit la route à mettre en place qui est **/salut-pote**.

Remarque préliminaire :

*Tous les fichiers doivent être encodés UTF-8 sans BOM sous votre éditeur de texte.*

#### a) création du contrôleur avec routage

##### a-1) dossier **Controller** et code complet

Lors de la création du projet, Symfony a créé un dossier **Controller** : c'est ici qu'il ira chercher les contrôleurs via le kernel.

Le contrôleur sera appelé ici **AvisController**.

Voici son code complet :

```

<?php

// src/Controller/AvisController.php
namespace App\Controller;

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

class AvisController extends AbstractController
{
    /**
     * @Route("/salut-pote", name="salut_mon_pote")
     */
    public function avis()
    {
        return new Response("Salut pote !");
    }
}

```

### Remarques :

- > Pour chaque fichier de code, on indiquera au début son chemin complet en commentaire.
- > Ici le chemin complet est **src/Controller/AvisController.php**.
- > L'espace de noms **App** correspond au dossier **src**. On met ensuite le sous-dossier éventuel ici **Controller**.

### a-2) explications

On importe ici tout d'abord plusieurs classes :

- **AbstractController** permet de définir notre contrôleur via un héritage
- **Route** permet de mettre en place une route
- **Response** permet d'écrire dans le navigateur

Le routage se met ici en place via une annotation.

Une annotation permet de générer des éléments.

Elle se met dans ce qu'on appelle un *docblock* : il s'agit d'un commentaire de documentation commençant par `/**` et se finissant par `*/`.



Dans ce commentaire, on va indiquer des directives à faire qui seront exécutées ensuite via une commande.

Ici on met donc en place une route, nous verrons ultérieurement qu'on utilisera aussi les annotations pour créer une table de base de données à partir d'une entité.

Pour mettre en place une route, on met l'annotation `@Route`.

Cela donne ici :

```
@Route("/salut-pote", name="salut_mon_pote")
```

Le premier paramètre obligatoire stipule le chemin qui sera mis dans l'URL et le second paramètre facultatif qui a pour nom `name` stipule le nom de la route, qui pour l'instant ici n'a pas d'intérêt.

Derrière cette annotation, on écrit la méthode associée à ce chemin, qui sera donc exécutée quand on invoquera le chemin.

Le nom de cette méthode est ici `avis()`.

Cette méthode affiche le message dans le navigateur via ici un objet de la classe `Response`.

#### Remarque :

La méthode retourne un objet de la classe `Response`.

On peut indiquer le type de ce que la méthode retourne dans son en-tête pour être plus précis

Cela donnerait ici :

```
public function avis() : Response
```

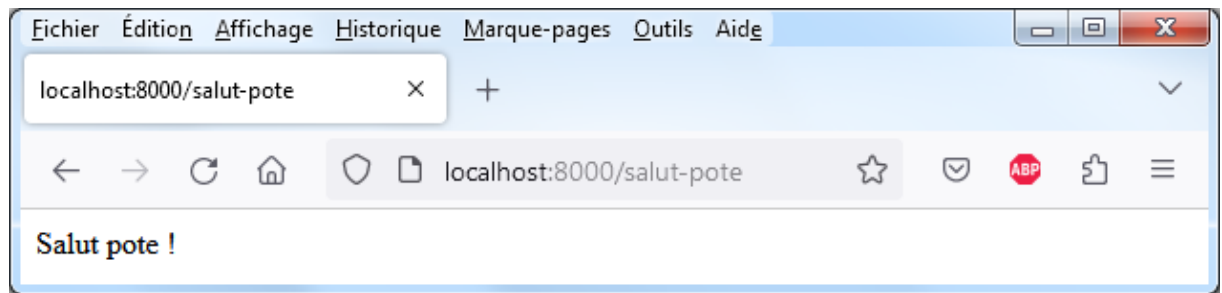
On pourra ainsi aussi trouver suivant les méthodes par exemple : `int` ou : `string`.

#### a-3) test

Taper donc l'URL suivante (on utilise ici **localhost** comme nom de domaine à la place de **encmis.lan** car on veut tester en local) :

**localhost:8000/salut-pote**

On obtient :



#### a-4) les formats de configuration sous Symfony : pour information

Nous venons de voir que pour configurer une route, on utilise l'**annotation**.

Si l'annotation est le format le plus utilisé, il en existe 3 autres : **yaml**, **xml** ou **php**.

Si les formats **xml** et **php** sont usuels, parlons du troisième **yaml**.

YAML signifie *YAML Ain't Markup Language* : ce n'est pas un langage de balise comme on peut le voir.

Dans un fichier YAML, on décrit de manière typique une suite de structures de données.

Le nom de la structure de données se termine par un deux-points : elle utilise le tiret de soulignement pour caractère de séparation.

Voici l'équivalent **yaml** pour la mise en place du routage précédent :

```
salut_mon_pote:
  path:      /salut-pote
  controller: App\Controller\AvisController::avis
```

On retrouve respectivement :

- le nom de la route ici `salut_mon_pote`
- le chemin associé ici `/salut-pote`
- la méthode à exécuter pour ce chemin ici `AvisController::avis` soit donc le contrôleur `AvisController` et dedans la méthode `avis`.

Dans un fichier YAML, il y a un élément par ligne avec ***indentation obligatoire de 4 espaces***. Les tabulations sont interdites.

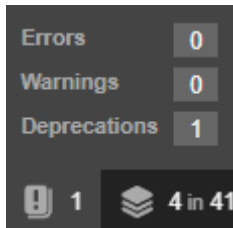
Si on ne respecte pas l'indentation de 4 en 4 espaces, Symfony signale une erreur.

On a donc réussi à mettre en place le contrôleur, avec une route, mais il n'empêche qu'il y a ici un défaut majeur : on ne respecte pas le modèle MVC vu que c'est le contrôleur qui réalise l'affichage et non la vue !

Mettons donc en place la vue : cela se fait sous Symfony en utilisant un template Twig qui est le partenaire naturel du framework au niveau de la couche *Vue*.

### a-5) accès aux logs et résolution d'une alerte relative aux dépréciations (éléments obsolètes) : pour information

Depuis qu'on a défini une route, nous n'avons plus d'erreurs apparaissant dans la barre d'état en bas dans le navigateur.



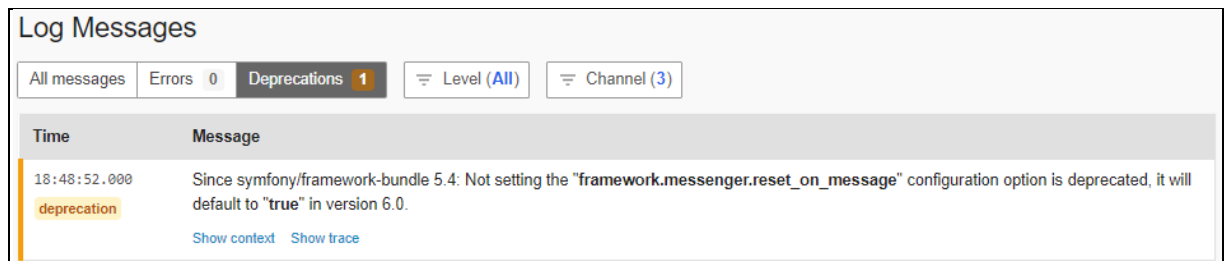
Le point d'exclamation n'est plus en rouge.

Par contre il y a ce qu'on appelle une dépréciation, *depreciation* en anglais.

Il s'agit d'un élément obsolète, le mot obsolète se traduisant en anglais technique par *depreciated*.

➔ cliquer dessus

Voici ce qui apparaît :



Même si cela n'aura pas de conséquences pour notre projet, voyons comment enlever cet élément obsolète, d'après le message de Symfony.

➔ aller dans le fichier **config/packages/messenger.yaml**

Voici ce qui apparaît :

```
framework:
    messenger:
        failure_transport: failed
```

Il faut rajouter ici la ligne :

```
reset_on_message: true
```

Cela va donner, ***en respectant bien l'indentation qui est de 4 espaces en 4 espaces***, donc ici il faut mettre 8 espaces :

```
framework:
    messenger:
        failure_transport: failed
        reset_on_message: true
```

➔ vérifier que dans le navigateur il n'y a plus d'alerte : plus de point d'exclamation

## b) création d'un template Twig

### b-1) les templates

Un template, appelé aussi gabarit ou layout, est un modèle (patron) de mise en page permettant de présenter des informations aux utilisateurs via une IHM (*Interface Homme Machine*) graphique.

Dans le contexte d'une application, elle concerne donc la couche *Vue* du modèle MVC.

En développement Web, un template est naturellement basé sur du HTML.

### b-2) moteurs de template et Twig

Lorsque la page doit afficher des informations dynamiques, il faut passer par un moteur de template qui est un script.

Nous avons vu jusque là le moteur de template PHP.

Symfony fonctionne par défaut avec le moteur Twig qui se base sur son propre langage plus simple que PHP, comme nous allons le voir.

A noter qu'au final le code Twig sera généré en PHP pour être compris du serveur Web Apache, puisque rappelons-le Symfony n'apporte qu'une couche logicielle supplémentaire.

### b-3) appel du template dans le contrôleur

Voici le code du contrôleur avec appel du template (la nouvelle ligne est en gras) :

```

<?php

// src/Controller/AvisController.php
namespace App\Controller;

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

class AvisController extends AbstractController
{
    /**
     * @Route("/salut-pote", name="salut_mon_pote")
     */
    public function avis()
    {
        return $this->render('salut.html.twig');
    }
}

```

La méthode `render()` permet d'appeler un template pour l'affichage dans le navigateur.

Le nom du template est ici **salut.html.twig**.

Symfony va chercher automatiquement les templates dans le dossier **templates** du dossier racine du projet ici **enemis** : on ne donne donc ici aucune indication de chemin.

#### Remarque :

On n'a plus besoin de l'instruction d'importation de la classe `Response`.

A présent, écrivons donc notre template.

#### b-4) code du template

Voici le code du template :

```

{# templates/salut.html.twig #}

Salut pote !

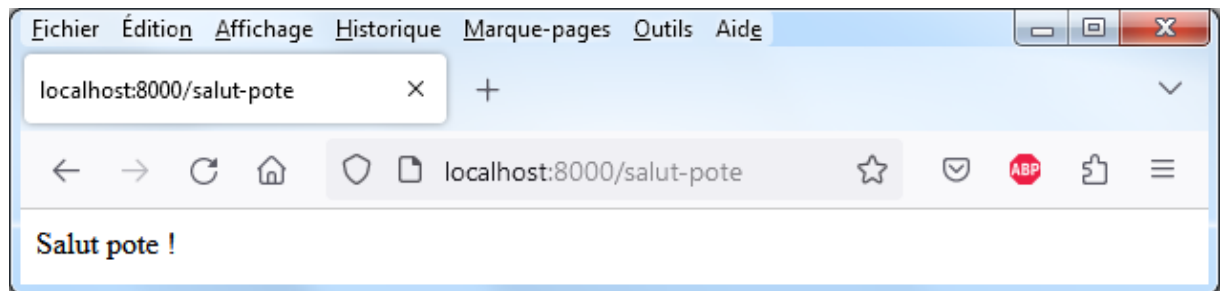
```

### Remarques :

- > Un commentaire Twig se met entre {# et #}.
- > On peut inclure des balises HTML, par exemple pour aller à la ligne.

A présent, affichons cette page avec son URL :  
**localhost:8000/salut-pote**

On obtient toujours :



On résume...

Quand on tape notre URL :  
**localhost:8000/salut-pote**

Symfony recherche dans un contrôleur (dossier **src/Controller**) cette route **salut-pote** et la trouve ainsi :

```
@Route("/salut-pote", name="salut_mon_pote")
```

Il exécute la méthode associée à cette route qui est `avis`.

Dans cette méthode, il y a cette instruction :

```
return $this->render('salut.html.twig');
```

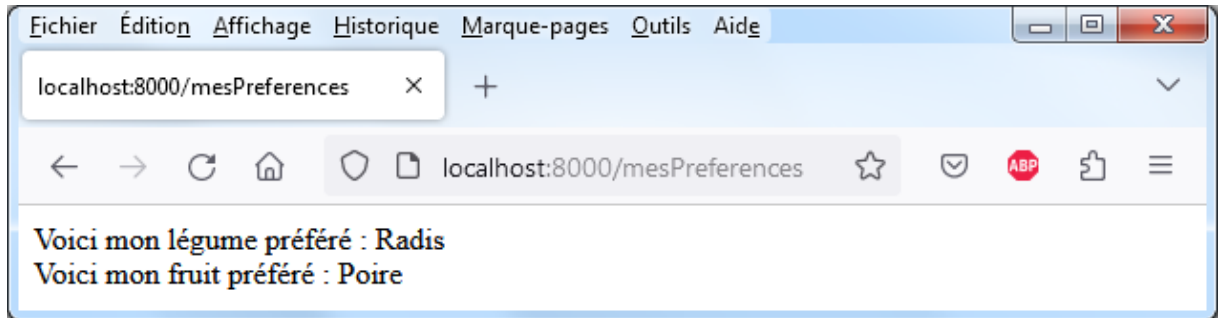
qui appelle le template ici de nom `salut.html.twig` (dossier **templates**).

Ce template réalise l'affichage.

## b-5) exercice 1

\* énoncé

Réaliser la page suivante :



Consignes :

- > La route à mettre en place sera celle apparaissant dans l'URL.
- > La nouvelle fonction de contrôleur sera toujours dans le contrôleur **AvisController**.
- > Le nouveau template s'appellera **preferences.html.twig**.

\* correction

La nouvelle fonction de contrôleur donne :

```
/**
 * @Route("/mesPreferences", name="mesPreferences")
 */
public function mesPreferences()
{
    return $this->render('preferences.html.twig');
}
```

Le nouveau template donne :

```
{# templates/preferences.html.twig #}
Voici mon légume préféré : Radis <br />
Voici mon fruit préféré : Poire
```

b-6) envoi de paramètres au template Twig

Le contrôleur peut envoyer au template un ou plusieurs paramètres.

Cela se fait via l'élément `array` avec son utilisation raccourcie via les crochets.

La syntaxe est :

`array('parametre1' => variable1, 'parametre2' => variable2, ...)`

qui peut être raccourcie en :

`['parametre1' => variable1, 'parametre2' => variable2, ...]`

Dans notre exemple, on souhaite envoyer le prénom du pote au template, qui sera mis en dur dans le code PHP.

Le code devient :

```
<?php

// src/Controller/AvisController.php
namespace App\Controller;

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

class AvisController extends AbstractController
{
    /**
     * @Route("/salut-pote", name="salut_mon_pote")
     */
```



```

public function avis()
{
    $prenomPote = "Marcel";
    return $this->render('salut.html.twig',
                        ['prenom' => $prenomPote]);
}

```

#### Commentaire :

On affecte la variable PHP `$prenomPote` qu'on fournit en paramètre au template, paramètre qui sera appelé simplement `prenom` dans le template.

Voici à présent le nouveau code de notre template :

```

{# templates/salut.html.twig #}
Salut pote {{ prenom }} !

```

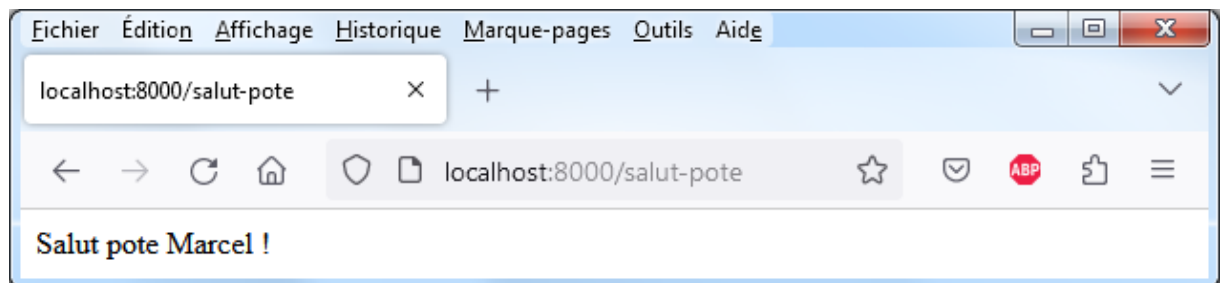
#### Commentaire :

A l'intérieur des doubles accolades, on affiche le paramètre (variable) ici `prenom`.

C'est plus simple qu'avec un script PHP qui donnerait ici par exemple :

```
Salut pote <?php echo $prenom; ?> !
```

On obtient ceci :



## b-7) exercice 2

### \* énoncé

Reprendre l'exercice 1 précédent du paragraphe b-5, et mettre le nom du légume et le nom du fruit dans des variables PHP qui seront affichées dans le template Twig.

\* correction

La fonction de contrôleur devient :

```
/**
 * @Route("/mesPreferences", name="mesPreferences")
 */
public function mesPreferences()
{
    $legumePrefere = "Radis";
    $fruitPrefere = "Poire";
    return $this->render('preferences.html.twig',
        ['legume' => $legumePrefere,
         'fruit' => $fruitPrefere]);
}
```

Le template devient :

```
{# templates/preferences.html.twig #}
Voici mon légume préféré : {{ legume }} <br />
Voici mon fruit préféré : {{ fruit }}
```

b-8) avantages et inconvénient de Twig

Twig présente plusieurs avantages : il est concis, sécurisé, bien adapté aux templates et extensible.

Son inconvénient est qu'il faut que le serveur Web génère du code PHP via son moteur PHP mais comme ce code PHP est mis en cache l'impact sur les performances est très limité.

Après avoir vu les éléments de mise en place d'un contrôleur (avec routage préalable) et d'une vue via le template Twig, penchons-nous sur la programmation du modèle qui va donc concerner les données et donc typiquement une base de données.

Le partenaire par défaut de Symfony pour les bases de données est Doctrine.

## 4) Gestion de base de données avec Doctrine

### a) préliminaires : domaine pratique à implémenter

#### a-1) cahier des charges

Nous souhaitons gérer les données simplifiées d'une organisation responsable de missions à caractère humanitaire à travers le monde.

Une mission est effectuée par plusieurs personnes pouvant être des ingénieurs, des médecins, des infirmiers, etc.

Elle se déroule parfois dans plusieurs pays.

Chaque mission est encadrée sur place notamment au niveau logistique par un employé de l'organisation.

*Notre domaine de gestion se limite à cet encadrement de missions par des employés.*

Une mission comprend :

- un genre prédéfini (mission médicale, mission de prospection, mission technique),
- un nom,
- une durée appartenant à une certaine tranche : 1 semaine, entre 1 et 3 semaines, 1 mois (qui est la durée la plus fréquente) ou plusieurs mois,

Un numéro séquentiel sera attribué à chaque mission.

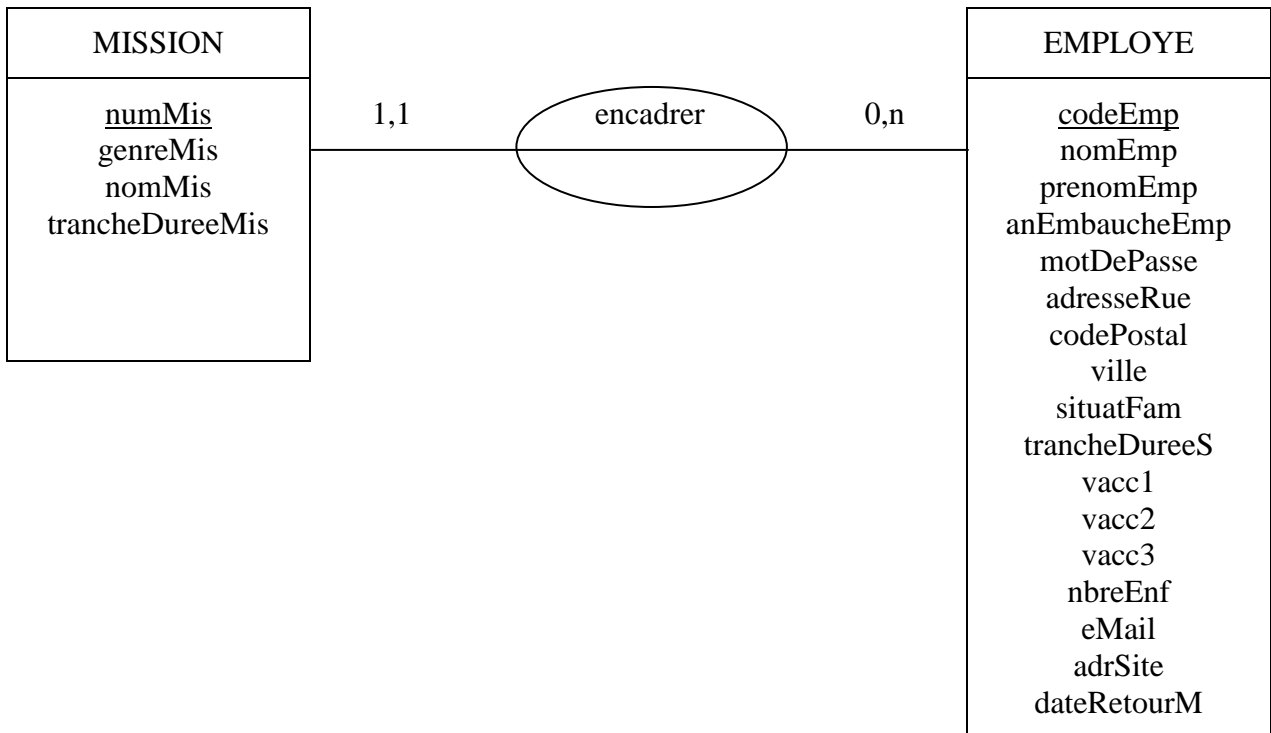
Pour l'employé, plus d'informations seront à mémoriser.

Un employé est caractérisé par :

- un code (qui a une nomenclature propre à l'organisation, il comprend 3 chiffres, puis 1 lettre majuscule, puis 3 chiffres, par exemple 109W827),
- un nom,
- un prénom,
- une année d'embauche,
- un mot de passe,
- une adresse avec code postal et ville,
- une situation familiale,
- une durée souhaitée pour la prochaine mission à encadrer : 1 semaine, entre 1 et 3 semaines, 1 mois (qui est la durée la plus fréquente) ou plusieurs mois,
- les vaccinations effectuées: il y en a de 3 types,
- un nombre d'enfants,
- un email,

- une adresse du site le plus visité, parmi les sites géographiques pour repérer les différents endroits où les missions vont se dérouler,
- une date de retour de la dernière mission encadrée.

### a-2) modèle conceptuel des données



Pas de problème particulier.

Une mission n'est encadrée que par un seul employé.

Un employé peut encadrer plusieurs missions.

La propriété genreMis sera sur un seul caractère : M pour Médicale, P pour Prospection, T pour Technique.

Pareil pour la propriété situatFam : C pour Célibataire, M pour Marié (e), D pour Divorcé (e), V pour Veuf (ve), A pour Autre.

La propriété trancheDureeMis sera définie comme un entier avec 4 valeurs possibles (de 1 à 4) pour chacune des tranches de durée possibles de la mission.

Pareil pour la propriété trancheDureeS.

Les trois propriétés vacc1, vacc2 et vacc3 sont de type booléen.

### a-3) modèle relationnel

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

***Nous nous intéressons ici pour l'instant à la table employes et à ses 4 premiers champs.***

### b) Doctrine en tant que ORM : mapping objet-relationnel

Pour l'instant, la gestion de base de données se fait via un SGBDR comme MySQL, MariaDB ou PostgreSQL.

Dans le code PHP, on inclut des requêtes SQL notamment SELECT, INSERT, UPDATE ou DELETE.

Même si cela se fait dans le cadre de la couche *Modèle*, on mélange du PHP et du SQL.

Symfony en tant que framework possède une librairie ORM (*Object-Relational Mapping*) qui est par défaut Doctrine.

Un ORM est une couche d'abstraction de base de données permettant de s'affranchir d'un langage de requête comme SQL.

Dans notre code PHP au lieu de lancer des requêtes, oninstanciera des classes et on invoquera des méthodes : Doctrine se chargera de lire et d'écrire dans la base de données.

L'insertion d'un enregistrement dans la table *employes* via une instruction PHP classique donne :

```
$requete =
"INSERT INTO employes (codeEmp, nomEmp, prenomEmp,
                        anEmbaucheEmp)
VALUES ('$codeE', '$nomE', '$prenomE', $anneeE)";

$bd->query("SET NAMES utf8");
$bd->query($requete);
```

Voici à titre de comparaison ce que cela va donner via l'ORM Doctrine (et les entités) :

```

ManagerRegistry $doctrine;
$em = $doctrine->getManager();

$monEmploye = new Employe();
$monEmploye->setCode("109W827");
$monEmploye->setNom("HATAN");
$monEmploye->setPrenom("Charles");
$monEmploye->setAnEmbauche(1992);

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

```

On est ici dans ce qu'on appelle le *mapping objet-relationnel* : il y a correspondance entre d'une part une classe, ici **Employe**, et une table, ici **employes**, et d'autre part chaque attribut de cette classe, par exemple ici **code**, et un champ de cette table, par exemple ici **codeEmp**.

Ainsi, plus de mélange PHP/SQL : on ne fera que du PHP.  
C'est bien dans l'esprit d'un framework.

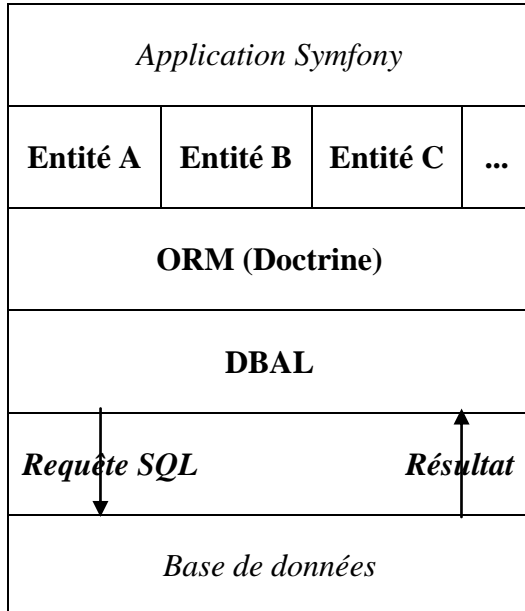
A noter que quand on utilise Twig, on est toujours dans cet esprit : on ne mélange pas du HTML et du PHP. L'affichage par exemple d'une variable ne se fait pas par une instruction PHP mais par du langage propre à Twig, en l'occurrence entre doubles accolades, qui est plus dans l'esprit code de présentation : pas de point-virgule notamment.

Doctrine est souple : il permet de travailler avec divers moteurs de bases de données comme MySQL, MariaDB, PostgreSQL ou SQLite.

Pour pouvoir faire son travail, un ORM comme Doctrine se base sur ce qu'on appelle des entités comme nous allons le voir ci-après.

### c) couches logicielles et ORM Doctrine

Le schéma suivant situe l'ORM Doctrine dans l'architecture complète relative à Symfony :



#### c-1) couche DBAL

Couche la plus basse, DBAL (*DataBase Abstraction Layer*) a pour rôle de lancer des requêtes sur la base de données et d'en récupérer les résultats.

Basée sur aucune logique applicative, elle utilise en interne le composant PDO de PHP.

#### c-2) entités

Une entité est une classe qui correspond à une table d'une base de données, et chacun de ses attributs à un champ (colonne) de la table : mise en œuvre du mapping objet-relationnel.

On parle d'*entité métier* et de *couche métier*.

Elle va être écrite selon certaines normes permettant à l'ORM Doctrine de bien faire le lien entre l'entité et une table de base de données.

Le résultat d'une requête ne sera plus sous la forme d'un tableau (souvent associatif) mais existera en tant que vrai objet.

Les différentes primitives sur une table (sélection d'enregistrements, ajout, mise à jour et suppression d'un enregistrement) se feront à travers ces entités.

### c-3) couche ORM

Cette couche concerne donc Doctrine.

Noyau de l'architecture, elle est l'interface entre les entités utilisées par l'application Symfony et la couche DBAL.

Son rôle est double :

- transformer les données reçues de la DBAL sous forme tabulaire en entités préalablement définies dans le code,
- à l'inverse, passer des différentes entités en requêtes SQL à fournir au DBAL.

On peut dire que l'ORM ici Doctrine fait la correspondance entre la base de données relationnelle, prise en charge par exemple par MySQL, et la programmation objet de PHP.

### d) configuration de Doctrine

Lors de l'installation du framework Symfony, Doctrine a automatiquement été installé comme peut l'être Twig un autre partenaire de Symfony (pour les templates : vues).

Il faut à présent indiquer à Doctrine l'URL de la base de données.

Pour connaître la base de données du projet, Symfony va dans le fichier **config/packages/doctrine.yaml**.

Voici un extrait de ce fichier :

```
# config/packages/doctrine.yaml

doctrine:
    dbal:
        url: '%env(resolve:DATABASE_URL) %'
```

Le paramètre `url` définit l'url de notre base de données.

Les pourcentages et l'élément `env` indiquent que la valeur est celle d'une variable d'environnement dont le nom est ici `DATABASE_URL`.

Où sont définies les variables d'environnement ?

Dans le fichier **.env** qui se trouve dans le dossier racine du projet.

Voici l'extrait correspondant qui nous intéresse et qui est mis au départ en commentaire :

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

Elle spécifie notamment que la base de données à utiliser est sous MySQL.



La ligne d'après est :

```
DATABASE_URL="postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=13&charset=utf8"
```

Elle spécifie notamment que la base de données à utiliser est sous Postgre.

Comme on ne travaille pas sur postgresql mais sur mysql on va décommenter la première ligne et commenter la seconde.

Reconsidérons la ligne :

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

On retrouve ici dans l'ordre :

- le serveur de base de données mysql
- le nom utilisateur pour se connecter à ce serveur de base de données
- le mot de passe pour se connecter à ce serveur de base de données
- l'adresse IP de la machine hébergeant le serveur de base de données : on retrouve l'adresse 127.0.0.1 (**localhost**) car ce serveur se trouve sur la même machine que le serveur Web
- le port d'écoute du serveur de base de données : la valeur par défaut pour MySQL est 3306
- le nom de la base de données avec la version du serveur qui peut être omise

Mettons donc à jour cette ligne pour notre cas.

Cela donne :

```
DATABASE_URL="mysql://root@127.0.0.1:3306/encadrementMissions"
```

Remarque :

Seul le fichier **.env** est modifié.

A présent lançons-nous donc dans la création d'une entité.

On souhaite gérer la table *employees* : notre entité qui sera donc une classe sera logiquement appelée *Employe*.

A noter que *le mot entité désigne tout autant la classe elle-même qu'ensuite chacune des instances de la classe (objet)*.

## e) création d'une entité

Les entités doivent être mises dans le dossier **src/Entity**.

### e-1) code de départ de la classe

Voici donc la classe `Employe` avec ses attributs privés et ses getters/setters :

```
<?php

// src/Entity/Employe.php

namespace App\Entity;

class Employe
{

    // attributs privés
    private $code;
    private $nom;
    private $prenom;
    private $anEmbauche;

    // getters et setters
    public function getCode()
    {
        return $this->code;
    }
    public function setCode($code)
    {
        $this->code = $code;
    }

    public function getNom()
    {
        return $this->nom;
    }
    public function setNom($nom)
    {
        $this->nom = $nom;
    }

    public function getPrenom()
```

```

{
    return $this->prenom;
}
public function setPrenom($prenom)
{
    $this->prenom = $prenom;
}

public function getAnEmbauche()
{
    return $this->anEmbauche;
}
public function setAnEmbauche($anEmbauche)
{
    $this->anEmbauche = $anEmbauche;
}

}

```

#### Remarque :

On peut indiquer pour chaque attribut son type.

Cela donnerait par exemple ceci pour l'attribut \$code.

```
private string $code;
```

De même, on peut indiquer pour chaque méthode (fonction ou procédure) quel type de valeur elle retourne.

Cela donnerait pour le getter sur l'attribut \$code (fonction) :

```
public function getCode() : string
```

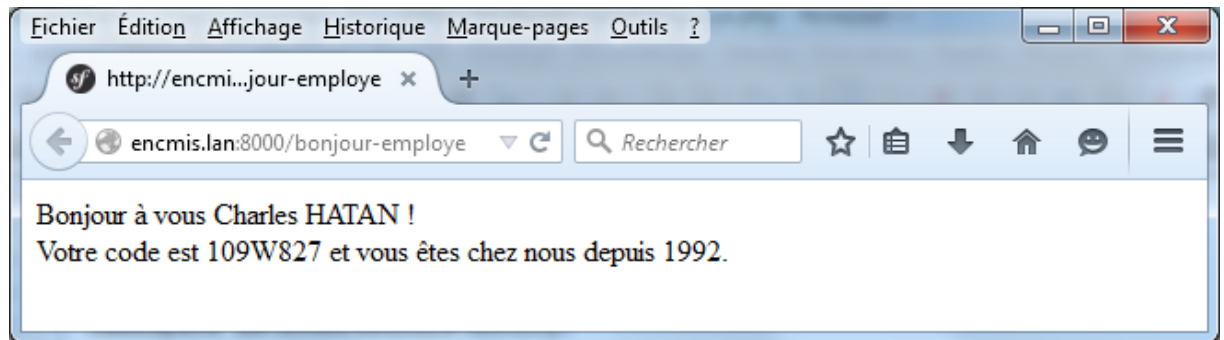
Et cela donnerait pour le setter sur l'attribut \$code (procédure) :

```
public function setCode($code) : void
```

A présent, nous allons faire un exercice permettant de faire un test unitaire de cette classe, tout en révisant la mise en œuvre d'une application Symfony.

e-2) exercice de test\* énoncé

Réaliser l'affichage suivant dans le navigateur (on mettra **localhost** à la place de **encmis.lan** pour tester en local) :

Consignes :

On mettra en place le contrôleur avec une route et une méthode associée.

On utilisera la classe `Employe` (qui sera préalablement importée via la commande `use App\Entity\Employe;`) : elle sera instanciée et les setters seront utilisés pour alimenter les attributs.

On écrira le template Twig : les getters seront utilisés pour fournir la valeur de chaque attribut à afficher par ce template.

\* correction

Le contrôleur nommé **BonjourController.php** donne :

```
<?php

// src/Controller/BonjourController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class BonjourController extends AbstractController
{
    /**
     * @Route("/bonjour-employe", name="bonjour_employe")
     */
    public function index()
    {
        // instantiation
        $monEmploye = new Employe();

        // alimentation des attributs
        $monEmploye->setCode("109W827");
        $monEmploye->setNom("HATAN");
        $monEmploye->setPrenom("Charles");
        $monEmploye->setAnEmbauche(1992);

        // on envoie au template Twig en paramètres les 4 getters
        return
            $this->render('bonjour.html.twig',
                [
                    'code' => $monEmploye->getCode(),
                    'nom' => $monEmploye->getNom(),
                    'prenom' => $monEmploye->getPrenom(),
                    'annee' => $monEmploye->getAnEmbauche()
                ]
            );
    }
}
```

Le template nommé **bonjour.html.twig** donne :

```
{# templates/bonjour.html.twig #}

Bonjour à vous {{ prenom }} {{ nom }} ! <br />
Votre code est {{ code }} et vous êtes chez nous depuis
{{ annee }}.
```

L'URL de la page donne :  
**localhost:8000/bonjour-employe**

### e-3) mise en place du mapping via les annotations

Pour que Doctrine puisse générer une table et ses colonnes à partir de l'entité, et donc faire son travail de mapping, il va falloir indiquer les éléments correspondants dans l'entité via une annotation : rappelons qu'une annotation se met en place avec le caractère @ à l'intérieur d'un commentaire de documentation qui est ainsi formé :

```
/**
 *
 *
 */
```

Voici le code complet avec les différentes annotations (les modifications sont en gras par rapport à la classe de départ) :

```
<?php

// src/Entity/Employe.php

namespace App\Entity;

// définition d'un alias
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="employes")
 */
class Employe
{

    // attributs privés
    /**
     * @ORM\Column(name="codeEmp", type="string", length=7)
     * @ORM\Id
```

```

    */
private $code;

/**
 * @ORM\Column(name="nomEmp", type="string", length=25)
 */
private $nom;

/**
 * @ORM\Column(name="prenomEmp", type="string", length=20,
               nullable=true)
 */
private $prenom;

/**
 * @ORM\Column(name="anEmbaucheEmp", type="integer",
               nullable=true)
 */
private $anEmbauche;

// getters et setters
public function getCode()
{
    return $this->code;
}
public function setCode($code)
{
    $this->code = $code;
}

public function getNom()
{
    return $this->nom;
}
public function setNom($nom)
{
    $this->nom = $nom;
}

public function getPrenom()
{
    return $this->prenom;
}
public function setPrenom($prenom)
{
    $this->prenom = $prenom;
}

public function getAnEmbauche()
{

```

```

        return $this->anEmbauche;
    }
    public function setAnEmbauche($anEmbauche)
    {
        $this->anEmbauche = $anEmbauche;
    }
}

```

On importe au départ la classe `Doctrine\ORM\Mapping` pour effectuer le mapping, en créant ensuite un alias nommé par convention `ORM` pour la suite.

Voyons donc en détail ces annotations et donnons quelques compléments.

#### e-4) l'annotation `\Entity` : définition d'une entité (future table)

Cette annotation permet de stipuler que la classe déclarée juste après est une entité : ***Doctrine saura ainsi qu'il faudra créer une table.***

Cela donne ici :

```

/**
 * @ORM\Entity
 */
class Employe

```

Doctrine créerait une table qui s'appellerait donc *Employe*.

Dans les bons usages de la programmation si le nom d'une classe commence par une majuscule et est au singulier, le nom d'une table commence par une minuscule et est au pluriel.

Pour spécifier un autre nom pour la table, on utilise l'annotation `\Table` et son paramètre `name`.

Cela donne donc finalement :

```

/**
 * @ORM\Entity
 * @ORM\Table(name="employes")
 */
class Employe

```



### e-5) l'annotation \Column : définition d'une colonne de table

Pour stipuler qu'un attribut devra être une future colonne de table, on met l'annotation \Column.

Cette annotation propose notamment 5 paramètres :

- name pour le nom de la colonne de table s'il n'est pas le même que l'attribut
- type pour le type : on retrouve ici notamment string, integer, float, date, datetime
- length pour la longueur maximale
- nullable pour autoriser la valeur NULL pour la colonne : dans ce cas, on met la valeur true. Attention ! Par défaut, la valeur est false, donc le remplissage est obligatoire.
- unique pour ne pas autoriser de doublons pour la colonne : dans ce cas, on met la valeur true (par défaut la valeur est false).

Par exemple pour l'attribut \$prenom, cela donne :

```
/**
 * @ORM\Column(name="prenomEmp", type="string", length=20,
 *             nullable=true)
 */
private $prenom;
```

Pour stipuler qu'un attribut, doit être considéré comme une future clé primaire, on utilise l'annotation \Id.

Pour notre cas, cela donne donc :

```
/**
 * @ORM\Column(name="codeEmp", type="string", length=7)
 * @ORM\Id
 */
private $code;
```

Pour stipuler que cette clé primaire doit être à incrémentation automatique, on utilise l'annotation \GeneratedValue et son paramètre strategy positionné à la valeur AUTO.

Pour le numéro de mission de l'entité Mission (que l'on écrira ultérieurement), cela donnera :

```
/**
 * @ORM\Column(name="numMis", type="integer")
 * @ORM\Id
 * @ORM\GeneratedValue(strategy="AUTO")
 */
private $num;
```

## f) création de la base de données et de ses tables

A présent, on va lancer des commandes pour créer la base de données et ses tables via bien sûr les entités : ici il n'y a qu'une seule entité donc qu'une seule table.

A noter qu'en préliminaire, *il ne faut pas oublier de démarrer le service MySQL, par exemple via WAMP.*

### f-1) création de la base de données

La commande sera à taper depuis *une autre fenêtre console* car la première fenêtre console permet de lancer le service Web du framework Symfony via la commande **symfony serve** (voir paragraphe 2-d).

Elle donne :

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

Rappelons que Doctrine va chercher les informations relatives à la base de données dans le fichier **config/packages/doctrine.yaml**.

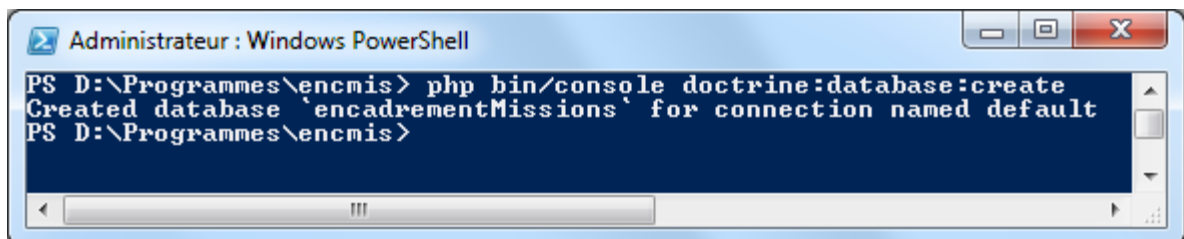
La ligne correspondante est :

```
url: '%env(resolve:DATABASE_URL) %'
```

La variable d'environnement est définie dans le fichier **.env** ainsi :

```
DATABASE_URL="mysql://root@127.0.0.1:3306/encadrementMissions"
```

Cela donne dans la console :



```

Administrateur : Windows PowerShell
PS D:\Programmes\encmis> php bin/console doctrine:database:create
Created database 'encadrementMissions' for connection named default
PS D:\Programmes\encmis>

```

Symfony a bien créé notre base de données *encadrementMissions*.

➔ constater par exemple via phpMyAdmin que la base de données a bien été créée

Pour toute nouvelle modification de la base, on ira vérifier sous cet outil.

## f-2) création des tables et autochargement des classes (*autoload*)

Cela va se faire en deux temps.

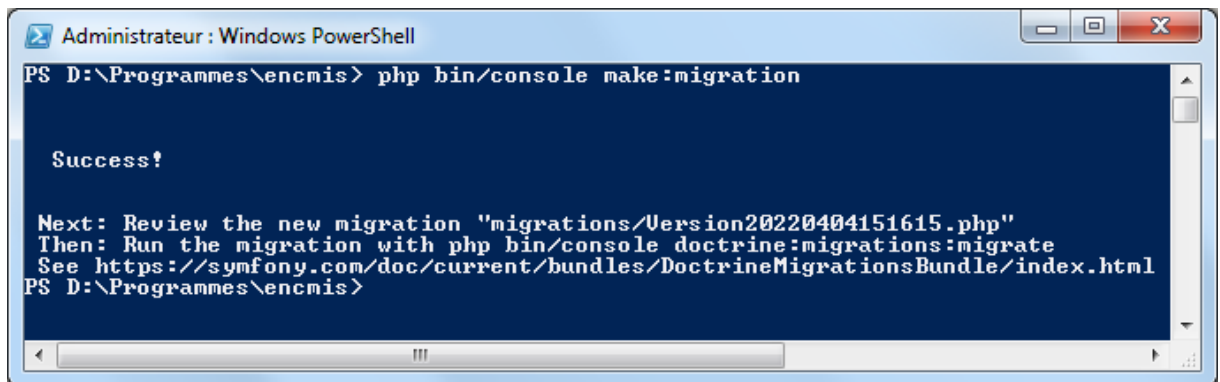
Dans un premier temps, on va générer le script de création de la table via la commande :

### **php bin/console make:migration**

Symfony va aller dans le dossier **src/Entity** et générer autant de scripts de création de table qu'il y aura d'entités dans ce dossier : programmation de la migration de chaque entité vers une table de base de données.

Ici, il va générer à partir de l'entité (classe) `Employe` le script de création de la table correspondante nommée `employes` avec ses 4 champs d'un certain type, et mettre le code employé en clé primaire, en exécutant donc nos différentes annotations.

Cela donne dans la console :



```

Administrateur : Windows PowerShell
PS D:\Programmes\encnis> php bin/console make:migration

Success!

Next: Review the new migration "migrations/Version20220404151615.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
PS D:\Programmes\encnis>
  
```

Une confirmation est ici demandée avant la génération du script.

*1 available migrations* signifie qu'il y a une seule migration à effectuer : une seule table à créer.

Le nom du script (PHP) généré est indiqué : il est dans le dossier **migrations**.

On retrouve dans ce script notamment la ligne suivante :

```

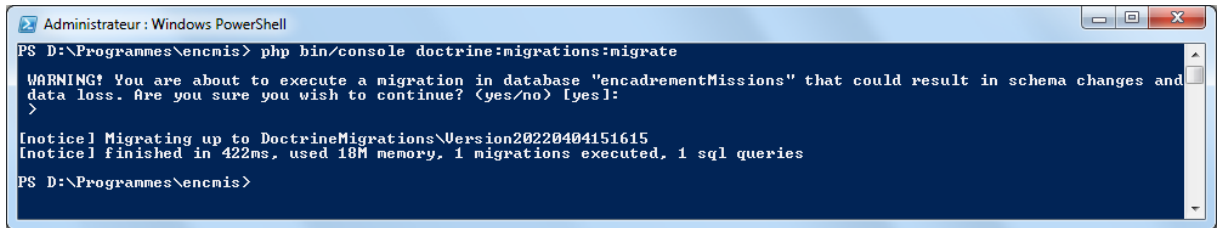
$this->addSql('CREATE TABLE employes
(codeEmp VARCHAR(7) NOT NULL,
 nomEmp VARCHAR(25) NOT NULL,
 prenomEmp VARCHAR(20) DEFAULT NULL,
 anEmbaucheEmp INT DEFAULT NULL,
 PRIMARY KEY(codeEmp))
DEFAULT CHARACTER SET utf8mb4
COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');
  
```

Pour l'instant la migration elle-même n'a pas été opérée : on pourra vérifier par exemple via phpMyAdmin que la table *employees* n'a pas été créée.

La commande de migration proprement dite (qui est aussi indiquée dans la console) est :

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

Cela donne dans la console :



```

Administrateur : Windows PowerShell
PS D:\Programmes\encnis> php bin/console doctrine:migrations:migrate
WARNING! You are about to execute a migration in database "encadrementMissions" that could result in schema changes and
data loss. Are you sure you wish to continue? (yes/no) [yes]:
>
[notice] Migrating up to DoctrineMigrations\Version20220404151615
[notice] finished in 422ms, used 18M memory, 1 migrations executed, 1 sql queries
PS D:\Programmes\encnis>
  
```

Une confirmation est demandée avant l'exécution du script : création de la table proprement dite (migration).

➔ Vérifier que la table *employees*, ses champs et leur type se sont bien créés, avec la présence de la clé primaire.

A noter qu'une autre table nommée *doctrine\_migration\_versions* s'est aussi générée automatiquement : elle va contenir notamment le nom du script PHP de génération de la table.

Symfony est capable de charger automatiquement le fichier définissant une classe dont il a besoin ; par exemple lorsque on met à jour une base de données, Symfony est capable de charger les classes depuis le dossier **Entity**, classes qui généreront des tables : on parle d'autochargement des classes ou *autoload*.

A présent, voyons comment manipuler les enregistrements, bien sûr *via l'ORM Doctrine et non pas directement via des requêtes SQL*.

Cela va se faire avec le service **EntityManager** de Doctrine.

## g) manipulations des enregistrements (opérations CRUD) : le service **EntityManager**

Ces manipulations des enregistrements via l'entité `Employe` vont permettre de mettre en place les opérations CRUD (*Create Read Update Delete*) sur la table de la base de données.

### g-1) préliminaires pour les tests et injection de service (injection de dépendances)

Nous allons créer un contrôleur pour les tests nommé **TestTableController**.

Voici son squelette :

```
<?php

// src/Controller/TestTableController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class TestTableController extends AbstractController
{
    /**
     * @Route("/test-table")
     */
    public function manip(ManagerRegistry $doctrine)
    {
    }
}
```

#### Commentaire :

La manipulation des enregistrements se fera donc dans la méthode `manip()` et son lancement via l'URL :

**localhost:8000/test-table**

Pour pouvoir injecter le service Doctrine dans la méthode du contrôleur ici `manip()` afin de pouvoir manipuler les enregistrements de la table correspondante, on met en paramètre une instance de la classe `ManagerRegistry` qu'on appelle ici `$doctrine` (objet).

Ce type d'injection s'appelle ***injection de dépendances***.

On affichera les messages directement dans le contrôleur (pas de template) car on n'est qu'en phase de test.

### g-2) récupération du service **EntityManager**

Les méthodes de modification au sens large d'un enregistrement de table (ajout, mise à jour, suppression) se lancent via le service **EntityManager**.

Il faut donc dans un premier temps récupérer ce service dans une variable via la méthode `getManager()` qu'on invoque sur l'instance de la classe `ManagerRegistry` permettant d'injecter le service Doctrine.

Cela donne :

```
$em = $doctrine->getManager();
```

### g-3) ajout d'un enregistrement

Dans un premier temps, il faut instancier l'entité correspondant à la table (son importation a eu lieu avant via l'instruction `use`).

Cela donne :

```
$monEmploye = new Employe();
```

Ensuite, il faut injecter (alimenter) les données via les setters.  
On le fera ici en dur puisqu'on est en phase de test.

Cela donne par exemple :

```
$monEmploye->setCode("109W827");  
$monEmploye->setNom("HATAN");  
$monEmploye->setPrenom("Charles");  
$monEmploye->setAnEmbauche(1992);
```

Pour l'instant, rien n'a encore été enregistré dans la base de données.

Doctrine reconnaît les transactions pour tout ce qui est requête `INSERT`, `UPDATE` ou `DELETE` : ou toutes les données sont modifiées, ou aucune en cas de problème (on peut alors consulter les fichiers logs : journaux) ce qui rend la base de données dans un état toujours cohérent.

Pour mettre en place ceci, on agit en deux étapes :

- notification à Doctrine que l'entité a été modifiée et qu'il faut rendre les données actuelles de l'entité persistantes, via la méthode `persist()`.

Cela donne :

```
$em->persist($monEmploye);
```

Pour l'instant, rien n'a encore été écrit dans la base de données sur le disque.

- demande à Doctrine de lancer la requête adéquate pour enregistrer les données dans la base (disque).

Cela donne :

```
$em->flush();
```

C'est là où Doctrine va passer par une transaction mais bien sûr sous Symfony c'est transparent pour le programmeur.

Voici le code complet :

```
<?php

// src/Controller/TestTableController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class TestTableController extends AbstractController
{
    /**
     * @Route("/test-table")
     */
    public function manip(ManagerRegistry $doctrine)
    {
        // récupération de l'Entity Manager
        $em = $doctrine->getManager();

        // instantiation de l'entité et injection des données
        $monEmploye = new Employe();
        $monEmploye->setCode("109W827");
        $monEmploye->setNom("HATAN");
        $monEmploye->setPrenom("Charles");
        $monEmploye->setAnEmbauche(1992);
```

```
// notification de la modification de l'entité
$em->persist($monEmploye);
// demande de modification de la base de données
$em->flush();

return new Response("Insertion réussie !");
}
}
```

Taper l'URL **localhost:8000/test-table**

L'enregistrement doit s'insérer dans la table.

➔ vérifier que cet enregistrement a bien été stocké dans la table

## Le point...

On a donc réussi dans l'ordre à :

- créer la base de données
- créer la table
- insérer un enregistrement dans la table

**sans écrire de commandes SQL qui pourraient entraîner des failles de sécurité.**

C'est bien le principe d'un ORM, qui est Doctrine sous Symfony.

Pour les tests, on remplira sous phpMyAdmin la table *employees* avec 5 enregistrements ainsi :

codeEmp	nomEmp	prenomEmp	anEmbaucheEmp
109W827	HATAN	Charles	1992
110A225	AYMAR	Jean	2002
284B128	COVER	Harry	1994
284C214	AUSSINNE	Emma	2004
284C226	ZETOFRAIS	Mélanie	1995



#### g-4) mise à jour d'un enregistrement et le repository

Contrairement à l'ajout d'un enregistrement, on ne va pas instancier l'entité mais on va la récupérer via Doctrine et plus précisément un *repository* : dépôt.

C'est dans ce repository que va se faire toute demande pour retrouver nos données.

Il y a un repository par entité.

On va donc récupérer dans un premier temps le repository en fournissant la classe correspondant à notre entité `Employe`.

Cela donne :

```
$repository = $doctrine->getRepository(Employe::class);
```

Ensuite, il faut retrouver le bon employé : cela se fait via la méthode `find()` qui admet pour paramètre une valeur du champ qui a été défini comme clé primaire.

Cette méthode `find()` retourne une entité, instance ici de la classe `Employe`.

Cela donne par exemple :

```
$monEmploye = $repository->find("284C214");
```

Ensuite, on invoque les setters pour modifier la valeur des attributs.

Cela donne par exemple :

```
$monEmploye->setNom("HIBULAIRE");
$monEmploye->setPrenom("Pat");
$monEmploye->setAnEmbauche(2003);
```

Et enfin, on procède à la sauvegarde via respectivement les deux méthodes `persist()` et `flush()`.

Cela donne :

```
$em->persist($monEmploye);
$em->flush();
```

```
<?php
```

```
// src/Controller/TestTableController.php
```

```
namespace App\Controller;
```

```
use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;
```

```
// importation de l'entité
```

```
use App\Entity\Employe;
```

```

class TestTableController extends AbstractController
{

    /**
     * @Route("/test-table")
     */
    public function manip(ManagerRegistry $doctrine)
    {

        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);
        // recherche de l'employé
        $monEmploye = $repository->find("284C214");
        // modification des données
        $monEmploye->setNom("HIBULAIRE");
        $monEmploye->setPrenom("Pat");
        $monEmploye->setAnEmbauche(2003);

        // récupération de l'Entity Manager
        $em = $doctrine->getManager();
        // notification de la modification de l'entité
        $em->persist($monEmploye);
        // demande de modification de la base de données
        $em->flush();

        return new Response("Mise à jour réussie !");
    }
}

```

### g-5) suppression d'un enregistrement

Le principe de départ va être le même que pour la mise à jour :

- on va récupérer le repository correspondant à notre entité `Employe`,
- on va rechercher le bon employé.

Ensuite, on notifie la suppression de l'entité via la méthode `remove()` et ensuite on demande la modification de la base de données via la méthode `flush()`.

Le code donne :

```
<?php

// src/Controller/TestTableController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class TestTableController extends AbstractController
{
    /**
     * @Route("/test-table")
     */
    public function manip(ManagerRegistry $doctrine)
    {
        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);
        // recherche de l'employé
        $monEmploye = $repository->find("284C214");

        // récupération de l'Entity Manager
        $em = $doctrine->getManager();
        // notification de la suppression de l'entité
        $em->remove($monEmploye);
        // demande de modification de la base de données
        $em->flush();

        return new Response("Suppression réussie !");
    }
}
```

On remettra l'enregistrement de départ (284C214 - AUSSINNE - Emma - 2004).

### g-6) extraction des enregistrements

Après avoir récupéré le repository correspondant à notre entité `Employe`, il faut cette fois récupérer toutes les entités : tous les employés.

Cela se fait via la méthode `findAll()`.

Ensuite, on fait une boucle de parcours et on affiche les valeurs des attributs via les getters, toujours dans notre contrôleur et non pas par un template Twig car on n'est qu'en phase de test.

Le code donne :

```
<?php

// src/Controller/TestTableController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class TestTableController extends AbstractController
{
    /**
     * @Route("/test-table")
     */
    public function manip(ManagerRegistry $doctrine)
    {
        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);
        // recherche de tous les employés
        $listeEmployes = $repository->findAll();

        // message complet à retourner
        $message = "";
        // parcours de tous les employés
```

```

foreach ($listeEmployes as $monEmploye)
{
    $message .= "Code : " . $monEmploye->getCode() . "<br />";
    $message .= "Nom : " . $monEmploye->getNom() . "<br />";
    $message .= "Prenom : " . $monEmploye->getPrenom() . "<br />";
    $message .= "Année embauche : " . $monEmploye->getAnEmbauche() .
        "<br /><br />";
}

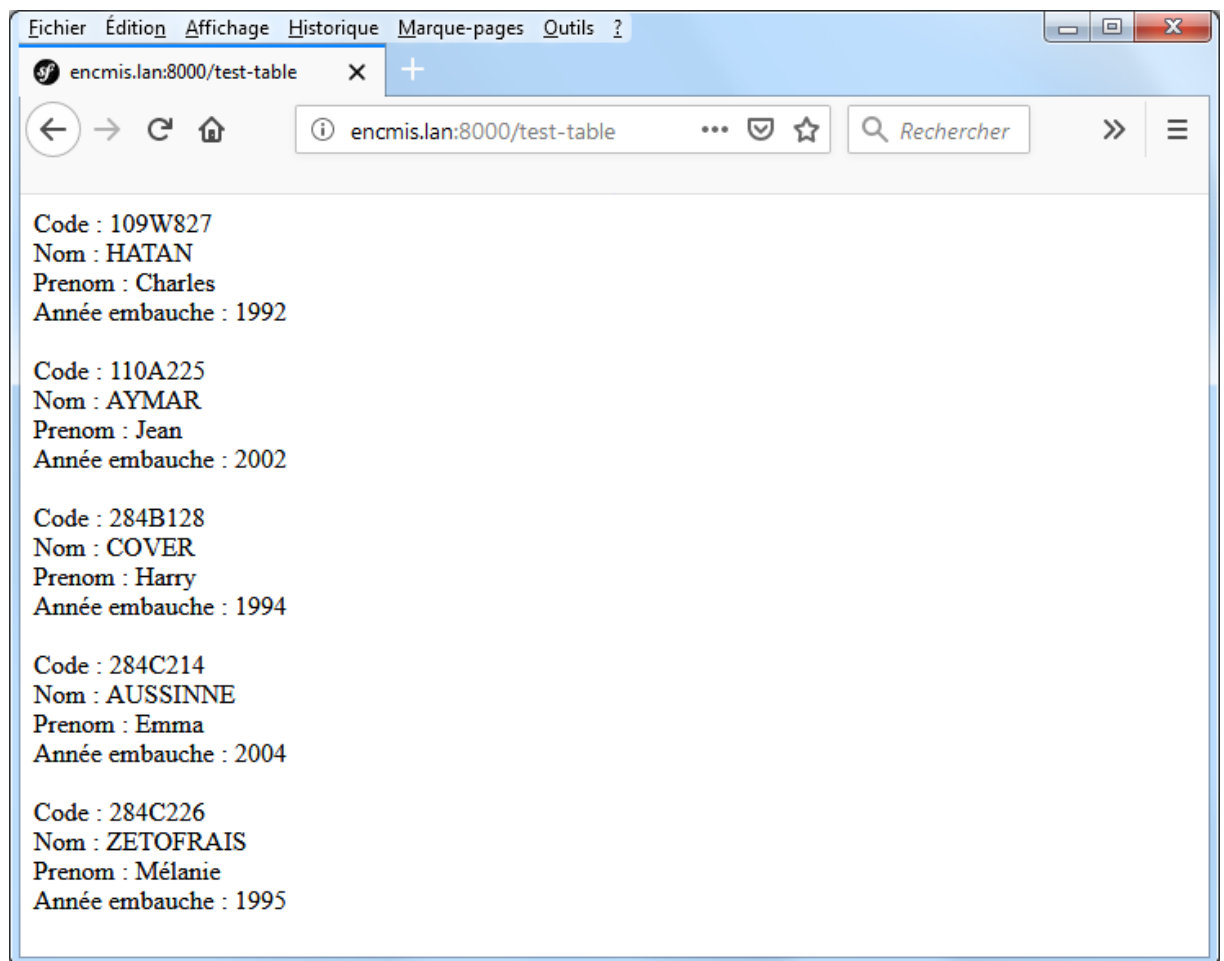
return new Response($message);
}
}

```

### Remarque :

Comme on ne modifie pas (ajout, mise à jour ou suppression) un enregistrement, on n'a pas besoin de récupérer l'**EntityManager**.

Cela donne à l'exécution :



## h) exercice site personnel

Vous allez mettre en place un projet Symfony pour votre site personnel, que vous avez développé dans un premier temps dans le cadre du cours sur le MVC2.

Vous vous aiderez des éléments vus jusqu'à présent.

### h-1) mise en place du projet

1-

Créer un nouveau projet Web Symfony dont la syntaxe sera *projet\_votrePrenom*.

Cela donnera par exemple **projet\_toto**.

2-

Démarrer le service Web relatif à votre projet.

3-

Tester le service Web depuis votre navigateur.

A présent, il faut créer via le framework Symfony la base de données et la table autour desquelles votre site personnel est développé.

### h-2) mise en place de la base de données

#### Précisions :

La base de données de votre site personnel à créer aura pour syntaxe *bd\_votrePrenom*. Cela donnera par exemple **bd\_toto**.

La table et ses champs seront nommés exactement comme pour votre site personnel.

Rappelons que cette table doit avoir une clé primaire de type auto-incrément, et au moins un champ de type texte et un champ de type numérique.

1-

Configurer l'ORM Doctrine afin qu'il prenne en compte la base de données de votre site personnel.

2-

Ecrire l'entité qui correspondra ultérieurement à votre table en mettant en œuvre les annotations adéquates.

3-

Créer la base de données dans la fenêtre console et vérifier qu'elle s'est bien créée par exemple via phpMyAdmin.

4-

Créer la table dans la fenêtre console et vérifier qu'elle s'est bien créée par exemple via phpMyAdmin.

Vous devez avoir une clé primaire et elle doit être de type auto-incrément.

A présent, nous allons faire des tests de manipulation.

5-

Créer 3 objets relatifs à votre entité et les ajouter dans la table.

6-

Afficher dans le navigateur l'ensemble des données de la table.

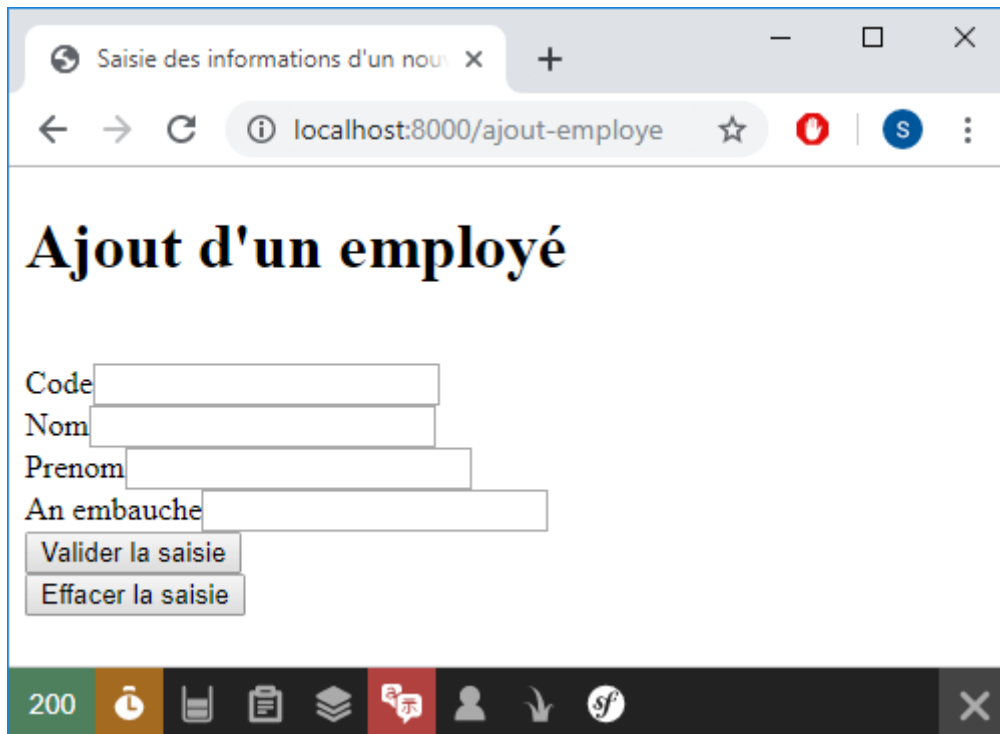
## 5) Mise en place d'un formulaire

### a) principe

La mise en œuvre d'un formulaire sous Symfony est étroitement liée à une entité de la couche *Modèle* : c'est à travers les différents champs de formulaire que seront alimentés (on parle d'injecter ou d'hydrater) ses attributs.

Ce formulaire sera construit via un contrôleur (couche *Contrôleur*) puis passé à un template Twig (couche *Vue*) pour son affichage.

Voici ce que l'on obtiendra :



The screenshot shows a web browser window with the title 'Saisie des informations d'un nou...' and the address bar displaying 'localhost:8000/ajout-employe'. The main content area features a large heading 'Ajout d'un employé'. Below the heading, there are four input fields labeled 'Code', 'Nom', 'Prenom', and 'An embauche'. At the bottom of the form, there are two buttons: 'Valider la saisie' and 'Effacer la saisie'. The browser's status bar at the bottom shows a green tab with the number '200' and various system icons.



## b) éléments de codage PHP et Twig

### b-1) code du contrôleur, constructeur/générateur du formulaire et propriétés d'une entité

Voyons donc les instructions au niveau du contrôleur.

Tout d'abord, on instancie l'entité `Employe`.

Cela donne :

```
$employe = new Employe();
```

Puis, on va créer un **constructeur de formulaire** via la méthode `createFormBuilder()` en fournissant l'entité précédente.

Cela donne :

```
$formBuilder = $this->createFormBuilder($employe);
```

Ensuite, on rajoute successivement via la méthode `add()` les différents attributs de l'entité que l'on veut alimenter : on n'est pas obligé de les mettre tous, par contre on est obligé de fournir à la méthode `add()` le nom de l'attribut avec son type ; on fournit également les boutons de type *Submit* et *Reset*.

Cela donne :

```
$formBuilder->add('code', TextType::class)
->add('nom', TextType::class)
->add('prenom', TextType::class)
->add('anEmbauche', TextType::class)
->add('validation', SubmitType::class,
    ['label' => 'Valider la saisie'])
->add('effacement', ResetType::class,
    ['label' => 'Effacer la saisie']);
```

En deuxième paramètre, on donne la classe du type de widget : `TextType::class`, `SubmitType::class` ou `ResetType::class`.

Pour les boutons de soumission et d'annulation, on fournit via l'option `label` l'intitulé du bouton.

A noter que si on ne met pas d'option `label`, Symfony reprend le nom du widget avec une majuscule au départ.

Ces types devront être importés.

Cela donnera respectivement :

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

```
use Symfony\Component\Form\Extension\Core\Type\ResetType;
```

### Précision :

Ici, il semble qu'on mette les attributs directement pourtant ils sont privés, par exemple l'attribut `code`.

Qu'en est il réellement ?

***Le fait de prévoir un getter sur un attribut, qui commence notamment par `get` ou `is` pour un booléen, entraîne en interne la génération par Symfony d'une propriété qui s'appelle comme l'attribut.***

Comme on a créé le getter `getCode`, Symfony génère automatiquement en interne la propriété `code`.

Dans la méthode `add()`, on invoque cette propriété `code` et non l'attribut `code`.

Si vous enlevez le getter `getCode`, Symfony met une erreur en précisant bien qu'il ne trouve pas de getter et que donc la propriété `code` invoquée n'existe pas.

***En résumé, le nom du widget est celui de la propriété (qui s'appelle comme l'attribut) de l'entité correspondante.***

Symfony déduit l'étiquette du champ de formulaire et met une majuscule pour chaque élément (voir copie d'écran du paragraphe a).

Puis on récupère le formulaire à partir de notre constructeur de formulaire via la méthode `getForm()`.

Cela donne :

```
$form = $formBuilder->getForm();
```

Enfin, on passe ce formulaire au template pour l'affichage. Pour cela, on fournit le paramètre qu'on nommera `form` dont la valeur est le retour de la méthode `createView()` sur le formulaire.

Cela donne :

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

## b-2) codage du template

Au niveau du template, l'affichage du formulaire se fait via la fonction `form` qui reçoit en paramètre l'objet envoyé par le contrôleur soit donc ici `form`.

Cela donne donc simplement :

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

Voyons à présent le code complet.

### c) code complet

Dans un premier temps, on écrit le code du contrôleur avec la route et sa méthode associée.

Cela donne :

```
<?php

// src/Controller/EmployeController.php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ResetType;

// importation de l'entité
use App\Entity\Employe;

class EmployeController extends AbstractController
{
    /**
     * @Route("/ajout-employe", name="form_ajout_employe")
     */
    public function ajout()
    {
        // instantiation de l'entité Employe
        $employe = new Employe();

        // création du constructeur de formulaire en fournissant l'entité
        $formBuilder = $this->createFormBuilder($employe);

        /* ajout successif des propriétés souhaitées de l'entité
           pour les champs de formulaire avec leur type */
    }
}
```

```

$formBuilder->add('code', TextType::class)
    ->add('nom', TextType::class)
    ->add('prenom', TextType::class)
    ->add('anEmbauche', TextType::class)
    ->add('validation', SubmitType::class,
        ['label' => 'Valider la saisie'])
    ->add('effacement', ResetType::class,
        ['label' => 'Effacer la saisie']);

// récupération du formulaire à partir du constructeur de formulaire
$form = $formBuilder->getForm();

// passage du formulaire au template pour affichage
return $this->render('employe/formAjout.html.twig',
    ['form' => $form->createView()]);
}
}

```

Dans un deuxième temps, on écrit le code du template qu'on mettra dans le nouveau dossier **templates/employe**.

Cela donne :

```

{# templates/employe/formAjout.html.twig #}

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>
            Saisie des informations d'un nouvel employé
        </title>
    </head>
    <body>
        <h1>Ajout d'un employé</h1><br />
        {{ form(form) }}
    </body>
</html>

```

Taper l'URL **localhost:8000/ajout-employe**

On obtient :

Saisie des informations d'un nou... x +

localhost:8000/ajout-employe ☆ | S

## Ajout d'un employé

Code

Nom

Prenom

An embauche

Valider la saisie

Effacer la saisie

200

## d) écriture des widgets dans la base de données

### d-1) code

Une fois les widgets remplis par l'utilisateur et après clic sur le bouton *Submit*, les informations doivent être écrites dans la table de base de données : ***persistance des données***.  
 Sous le framework Symfony, cette écriture passe par l'entité et ses attributs.

Voici le nouveau code du contrôleur (les modifications sont en gras) :

```
<?php

// src/Controller/EmployeController.php

namespace App\Controller;

use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ResetType;

// importation de l'entité
use App\Entity\Employe;

class EmployeController extends AbstractController
{
    /**
     * @Route("/ajout-employe", name="form_ajout_employe")
     */
    public function ajout(ManagerRegistry $doctrine,
                        Request $request)
    {
        // instantiation de l'entité Employe
        $employe = new Employe();

        // création du constructeur de formulaire en fournissant l'entité
        $formBuilder = $this->createFormBuilder($employe);

        /* ajout successif des propriétés souhaitées de l'entité
           pour les champs de formulaire avec leur type */
        $formBuilder->add('code', TextType::class)
```

```

->add('nom', TextType::class)
->add('prenom', TextType::class)
->add('anEmbauche', TextType::class)
->add('validation', SubmitType::class,
    ['label' => 'Valider la saisie'])
->add('effacement', ResetType::class,
    ['label' => 'Effacer la saisie']);

// récupération du formulaire à partir du constructeur de formulaire
$form = $formBuilder->getForm();

/* traitement de la requête : Symfony récupère éventuellement les valeurs des
   champs de formulaire et alimente l'objet $employe */
$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid())
    // le formulaire a été soumis et il est valide
    {
        // écriture dans la base de données
        $em = $doctrine->getManager();
        $em->persist($employe);
        $em->flush();
        return new Response("Employé ajouté dans la base...");
    }

// passage du formulaire au template pour affichage
return $this->render('employe/formAjout.html.twig',
    ['form' => $form->createView()]);

}
}

```

### Commentaires :

La méthode `ajout()` a cette fois des paramètres : l'instance de la classe `ManagerRegistry` permettant d'injecter le service `Doctrine` et une instance de la classe `Request` permettant de récupérer les données reçues de la requête.

Une fois le formulaire généré, il faut traiter la requête et récupérer éventuellement les valeurs des champs de formulaire via la méthode `handleRequest()`.

Il y a deux cas de figure :

- soit c'est la première fois qu'on invoque la page  
Le contrôleur est exécuté : pour l'instant le formulaire n'a pas été soumis, la méthode `isSubmitted()` relative au formulaire renvoie la valeur `false` et donc la dernière instruction de passage du formulaire au template est exécutée.
- soit l'utilisateur a cliqué sur le bouton *Submit*

Le contrôleur est exécuté à nouveau : cette fois les champs de formulaire sont remplis et la méthode `handleRequest()` alimente l'entité.

Le formulaire a été soumis et si les valeurs sont valides (nous verrons comment mettre des règles de validation, par exemple une valeur de champ obligatoire), on écrit l'entité dans la base de données.

#### d-2) exercice de test

- ➔ saisir les informations d'un nouveau employé depuis le formulaire et vérifier qu'il est bien dans la table



## e) personnalisation d'un formulaire : customisation

### e-1) présentation

L'affichage d'un formulaire dans un template Twig, invoqué par une méthode du contrôleur, se fait en une seule instruction ainsi :

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

Cette instruction va afficher le formulaire construit par le contrôleur et donc le nom `form` entre parenthèses correspond au paramètre du contrôleur.

Si elle est très simple à utiliser, elle ne permet pas le contrôle du rendu final dans le navigateur.

Voyons donc comment réussir à personnaliser dans ce fichier Twig l'affichage de notre formulaire et de ses composants puisque c'est lui qui a le dernier mot !

### e-2) intégration de Bootstrap

Pour tout ce qui est mise en forme, nous allons utiliser le framework CSS Bootstrap, et plus précisément le thème Bootstrap 4.

Pour cela dans le fichier de configuration **config/packages/twig.yaml**, on va rajouter une ligne.

Le fichier devient :

```
# config/packages/twig.yaml

twig:
    default_path: '%kernel.project_dir%/templates'
    form_themes: ['bootstrap_4_layout.html.twig']
```

Notons la présence des crochets car on peut mettre plusieurs thèmes.

***Ne pas oublier les 4 espaces comme le veut la norme YAML.***

Ensuite, il faudra dans toute page HTML rajouter la balise suivante :

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
bootstrap.min.css">
```

Voici à nouveau le template en utilisant les attributs traditionnels de Bootstrap :

```
{# templates/employe/formAjout.html.twig #}

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>
      Saisie des informations d'un nouvel employé
    </title>
    <link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
        bootstrap.min.css">
  </head>
  <body>

    <div class="container">

      <h1>Ajout d'un employé</h1><br />

      <div class="row">

        {{ form(form) }}

      </div>

    </div>

  </body>
</html>
```

On obtient ceci :

# Ajout d'un employé

Code

Nom

Prenom

An embauche

Valider la saisie

Effacer la saisie

La présentation est beaucoup plus agréable.

### e-3) la customisation du code

Nous allons devoir **personnaliser** les étiquettes notamment pour afficher *Année d'embauche* à la place de *An embauche*.

Le mot **personnaliser** se traduit en anglais technique par **customiser** et le mot personnalisation par **customisation**.

En informatique, on parle souvent de **customiser le code** ou de **customisation du code**.

#### e-4) fonctions Twig primaires sur les formulaires

L'instruction :

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

est équivalente aux instructions successives suivantes :

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

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

Le template devient :

```
{# templates/employe/formAjout.html.twig #}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
  <head>
```

```
    <meta charset="utf-8">
```

```
    <title>
```

```
      Saisie des informations d'un nouvel employé
```

```
    </title>
```

```
    <link rel="stylesheet"
```

```
      href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
        bootstrap.min.css">
```

```
  </head>
```

```
  <body>
```

```
    <div class="container">
```

```
      <h1>Ajout d'un employé</h1><br />
```

```
      <div class="row">
```

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

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

```
      </div>
```

```
    </div>
```

```
  </body>
```

```
</html>
```

Voyons sommairement ces 2 fonctions.

\* form\_start()

Cette fonction permet de générer la balise `<form>` : ouverture du formulaire.

Au niveau du contrôleur, on gère les attributs `method` et `action`.  
On peut donc intercepter ces attributs dans le Twig.

Supposons qu'on souhaite rediriger la page vers la route `autreChemin` et qu'on veuille une requête de type `GET`, la fonction `form_start()` sera ainsi redéfinie :

```
{{ form_start(form, {'action' : path('autre_chemin'),
                      'method' : 'GET'}) }}
```

A noter qu'on peut faire la même chose directement via la fonction `form()` ainsi :

```
{{ form(form, {'action' : path('autre_chemin'),
                'method' : 'GET'}) }}
```

\* form\_end()

Cette fonction permet de générer la balise `</form>` : fermeture du formulaire.

A présent, customisons chaque champ de formulaire.

### e-5) la fonction form\_row()

Cette fonction permet d'agir sur chaque couple champ de formulaire et étiquette.

Supposons qu'on veuille présenter chaque champ de formulaire de manière différente, par exemple, on voudrait décaler un peu la partie du code de l'employé et la partie de l'année d'embauche via la classe Bootstrap `col` pour obtenir ceci :

# Ajout d'un employé

Code

Nom

Prenom

An embauche

Valider la saisie

Effacer la saisie

Le code devient :

```
{# templates/employe/formAjout.html.twig #}

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>
      Saisie des informations d'un nouvel employé
    </title>
    <link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
        bootstrap.min.css">
  </head>
  <body>

    <div class="container">

      <h1>Ajout d'un employé</h1><br />
```

```

<div class="row">

    {{ form_start(form) }}

    <div class="col">
        {{ form_row(form.code) }}
    </div>

    {{ form_row(form.nom) }}

    {{ form_row(form.prenom) }}

    <div class="col">
        {{ form_row(form.anEmbauche) }}
    </div>

    {{ form_end(form) }}

</div>

</div>

</body>
</html>

```

### e-6) fonctions `form_label()` et `form_widget()`

Ces deux fonctions sont montrées en même temps car elles vont de pair.

Elles vont permettre d'agir séparément sur l'étiquette du champ de formulaire ou le champ de formulaire lui-même.

Dans notre cas, on veut notamment customiser les étiquettes pour le prénom (avec l'accent) et l'année d'embauche pour obtenir ceci :

# Ajout d'un employé

Code

Nom

Prénom

Année d'embauche

Valider la saisie

Effacer la saisie

Le code devient :

```
{# templates/employe/formAjout.html.twig #}

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>
      Saisie des informations d'un nouvel employé
    </title>
    <link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
        bootstrap.min.css">
  </head>
  <body>

    <div class="container">
```



```

<h1>Ajout d'un employé</h1><br />

<div class="row">

    {{ form_start(form) }}

        {{ form_row(form.code) }}

        {{ form_row(form.nom) }}

        {{ form_label(form.prenom, "Prénom") }}
        {{ form_widget(form.prenom) }}
        <br />

        {{ form_label(form.prenom, "Année d'embauche") }}
        {{ form_widget(form.anEmbauche) }}
        <br />

    {{ form_end(form) }}

</div>

</div>

</body>
</html>

```

### e-7) éléments complémentaires : pour information

#### \* form\_errors()

Cette fonction permet d'afficher les erreurs de saisie relatives à un champ à partir des règles définies au niveau du modèle ou du contrôleur.

L'extrait de code devient (la nouvelle ligne est en gras) :

```

{{ form_start(form) }}

    {{ form_errors(form) }}

{{ form_end(form) }}

```

\* form\_rest()

Cette fonction permet d'afficher tous les champs non encore rendus, notamment les champs cachés.

Si on voulait afficher d'éventuels champs cachés, l'extrait de code deviendrait (la nouvelle ligne est en gras) :

```
{{ form_start(form) }}

{{ form_row(form.code) }}

{{ form_row(form.nom) }}

{{ form_label(form.prenom, "Prénom") }}
{{ form_widget(form.prenom) }}
<br />

{{ form_label(form.prenom, "Année d'embauche") }}
{{ form_widget(form.anEmbauche) }}
<br />

{{ form_rest(form) }}

{{ form_end(form) }}
```

\* form\_end()

Comme nous le savons, cette fonction permet de générer la balise `</form>` : fermeture du formulaire.

Mais `form_end()` a aussi pour rôle d'afficher les widgets qui n'ont pas été mentionnés via la fonction `form_widget()`.

Pour ne pas afficher au final ces champs non mentionnés, on surcharge la méthode `form_end()` ainsi :

```
{{ form_end(form, {render_rest : false} ) }}
```

Voici donc les éléments pour la personnalisation du formulaire.

## 6) Moteur de template Twig et mise en page des informations

Nous avons écrit quelques templates basiques.

A présent, écrivons des pages Web un peu plus complexes, notamment la page de présentation des différents employés.

Nous allons étudier le langage de template Twig et voir en premier lieu quels sont les bons usages de la programmation Twig.

### a) template Twig de base du framework et héritage

Par défaut, Symfony propose un template (layout) de base pour la mise en place de pages Twig.

Le bon usage est de se baser dessus pour créer ses propres templates : on va *hériter* de ce template de base.

Ce template de base se trouve dans le dossier **templates** comme tous les autres templates Twig.

Son nom est **base.html.twig** et il contient ceci (extrait) :

```
{# templates/base.html.twig #}

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>{% block title %}Welcome!{% endblock %}</title>
    {% block stylesheets %}{% endblock %}
    {% block javascripts %}{% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
  </body>
</html>
```

On retrouve les éléments d'ossature d'HTML5 avec en plus des instructions Twig.

## b) les blocs

Un bloc permet de mettre du contenu à un certain endroit du document HTML.

La syntaxe d'un bloc est :

```
{% block typeBloc %}Contenu{% endblock %}
```

Pour *typeBloc*, on trouve ici les valeurs `title`, `stylesheets`, `javascripts` et `body`.

Dans le template de base, seul le bloc `title` est rempli avec la valeur *Welcome!*

Faisons à présent des tests pour voir comment mettre en place une page en se basant donc sur ce template **base.html.twig**.

Nous reprendrons des éléments déjà existants.

Le squelette du contrôleur avec routage et méthode à exécuter est :

```
<?php

// src/Controller/AvisController.php
namespace App\Controller;

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

class AvisController extends AbstractController
{
    /**
     * @Route("/salut-pote", name="salut_mon_pote")
     */
    public function avis()
    {
    }
}
```

## c) utilisation du template de base et code HTML généré

### c-1) affichage du template de base

Dans un premier temps, écrivons le contrôleur lançant le template de base.

Cela donne (la nouvelle instruction est en gras) :

```
<?php

// src/Controller/AvisController.php
namespace App\Controller;

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

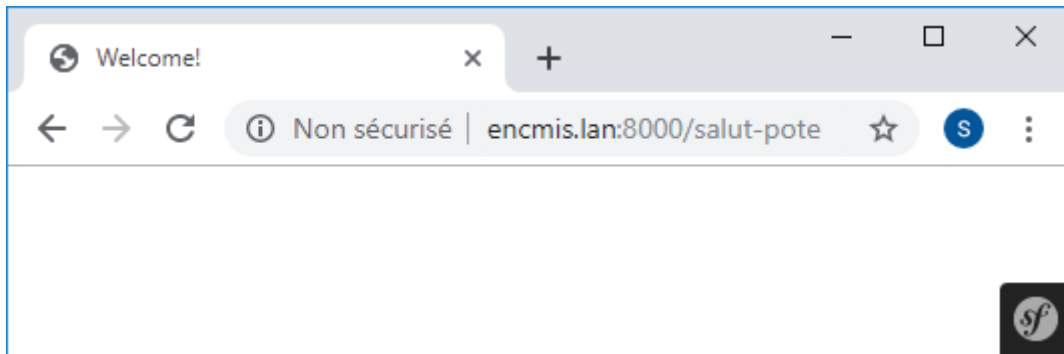
class AvisController extends AbstractController
{
    /**
     * @Route("/salut-pote", name="salut_mon_pote")
     */
    public function avis()
    {
        return $this->render('base.html.twig');
    }
}
```

Il n'y a pas ici de difficulté particulière.

Le code HTML généré donne :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Welcome!</title>
  </head>
  <body>
  </body>
</html>
```

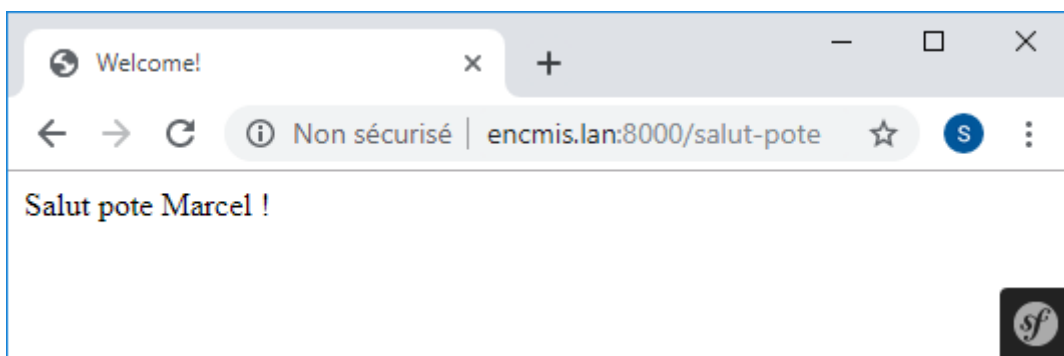
On obtient dans le navigateur :



Il y a bien *Welcome!* dans la barre de titre.

### c-2) écriture de templates Twig hérités

Supposons qu'à présent on veuille écrire la page suivante :



Le titre sera toujours *Welcome!* et on doit changer le corps : on va écrire un template qui **hérite** du template de base et **redéfinit les blocs souhaités**.

Le code du contrôleur va donner (les modifications sont en gras) :

```
<?php

// src/Controller/AvisController.php
namespace App\Controller;

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

class AvisController extends AbstractController
{
    /**
     * @Route("/salut-pote", name="salut_mon_pote")
     */
}
```

```

public function avis()
{
    $prenomPote = "Marcel";
    return $this->render('salut.html.twig',
                        ['prenom' => $prenomPote]);
}
}

```

Le template Twig donne :

```

{# templates/salut.html.twig #}

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

{% block body %}
    Salut pote {{ prenom }} !
{% endblock %}

```

### Commentaires :

L'héritage se fait via l'instruction `extends`.

Pour exécuter une instruction avec Twig, il faut la mettre dans un élément `{% %}`, comme pour la déclaration d'un bloc.

Ensuite, selon le principe de l'héritage, on ne change que ce que l'on souhaite : **redéfinition** des blocs.

Ici cela ne concerne que le bloc `{% block body %}`.

Tous les autres éléments comme le bloc `{% block title %}` ou les balises HTML comme `meta` sont repris tels quels.

Le code HTML généré donne :

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>Welcome!</title>
    </head>
    <body>
        Salut pote Marcel !
    </body>
</html>

```

A présent, supposons qu'on veuille modifier le contenu du titre pour mettre *Bienvenue!* au lieu de *Welcome!*

Le template Twig devient (les nouvelles lignes sont en gras) :

```
{# templates/salut.html.twig #}

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

{% block title %}
    Bienvenue!
{% endblock %}

{% block body %}
    Salut pote {{ prenom }} !
{% endblock %}
```

Le code HTML généré donne :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Bienvenue!</title>
  </head>
  <body>
    Salut pote Marcel !
  </body>
</html>
```

Et pour que ce soit plus international, supposons à présent qu'on ne veuille plus *changer* le titre mais *compléter* le titre pour avoir *Welcome! ou Bienvenue!*

Il va donc falloir afficher dans un premier temps le contenu du bloc du template de base qui est le template parent.

Cela se fait via la fonction `parent()`.

Le template Twig devient (la ligne modifiée est en gras) :

```
{# templates/salut.html.twig #}

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

{% block title %}
    {{ parent() }} ou Bienvenue!
{% endblock %}

{% block body %}
```



```
    Salut pote {{ prenom }} !  
{% endblock %}
```

Le code HTML généré donne :

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="UTF-8" />  
    <title>Welcome! ou Bienvenue!</title>  
  </head>  
  <body>  
    Salut pote Marcel !  
  </body>  
</html>
```

## d) modification du template de base

Si on veut qu'un élément soit valable pour tous les templates, on va logiquement le mettre dans le template de base.

On utilise Bootstrap pour la mise en forme de toutes les pages de notre site : la balise `<link>` relative à Bootstrap sera donc rajoutée dans le fichier **base.html.twig** à l'intérieur du bloc `{% block stylesheets %}{% endblock %}`.

Cela va donner :

```
{# templates/base.html.twig #}

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>{% block title %}Welcome!{% endblock %}</title>
    {% block stylesheets %}
      <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
          bootstrap.min.css">
    {% endblock %}
    {% block javascripts %}{% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
  </body>
</html>
```

Et du coup, on ne mettra plus cette balise `<link>` dans les autres pages du site.

## 7) Mise en place du site d'exemple via Symfony

Nous allons reprendre l'exemple de référence permettant notamment la gestion d'employés (encadrant des missions), en utilisant donc les concepts vus jusque là : contrôleur avec routage, template Twig et entité.

### a) différents écrans à générer avec URL

Chacun des écrans va avoir une certaine URL dans la barre d'adresses : une URL est associée à une méthode dans le contrôleur.

Le premier écran concerne la liste des employés : page d'accueil.

C'est l'écran central, qui sera affiché au départ.

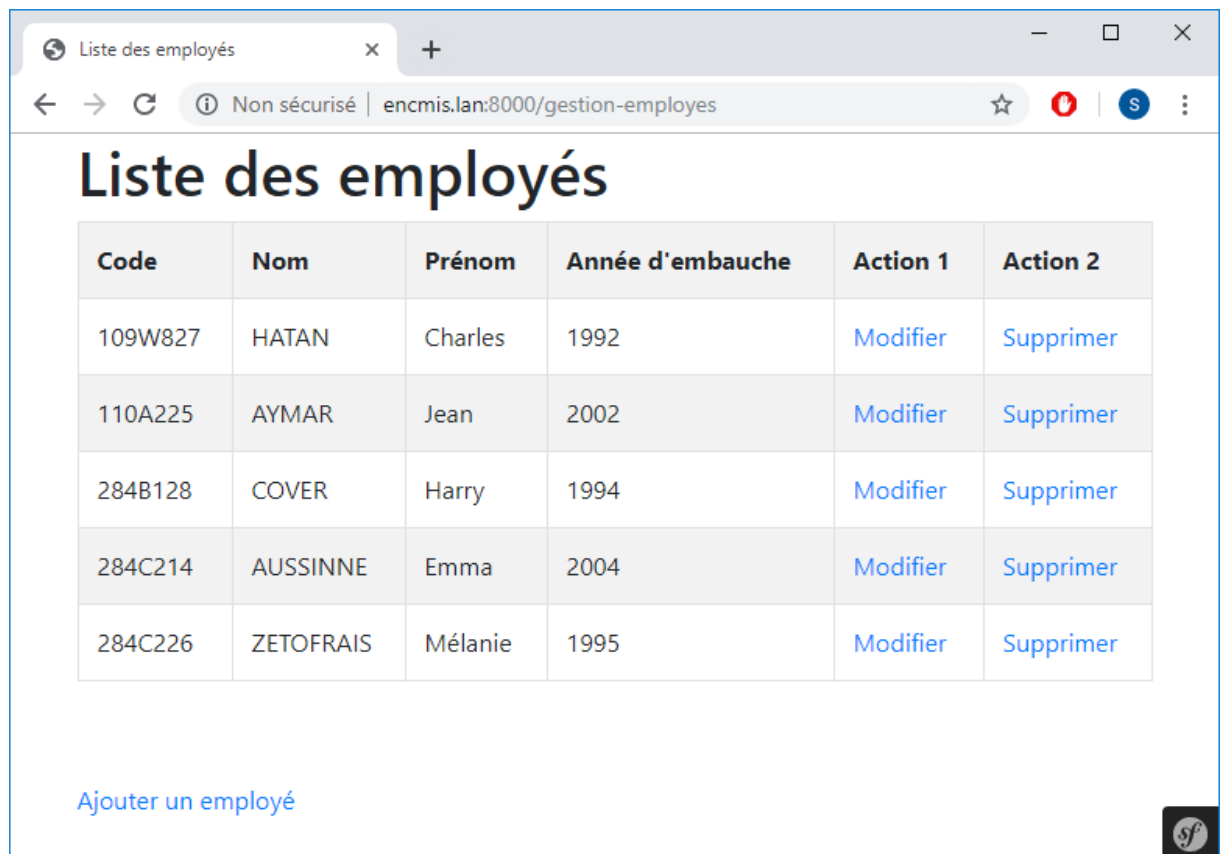
L'URL relative est :

/gestion-employes

De manière pratique, l'utilisateur va taper l'URL (on mettra dans la suite **localhost** à la place de **encmis.lan** si on veut tester en local) :

**encmis.lan:8000/gestion-employes**

Voici son affichage :



Code	Nom	Prénom	Année d'embauche	Action 1	Action 2
109W827	HATAN	Charles	1992	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
110A225	AYMAR	Jean	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284B128	COVER	Harry	1994	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C214	AUSSINNE	Emma	2004	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C226	ZETOFRAIS	Mélanie	1995	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

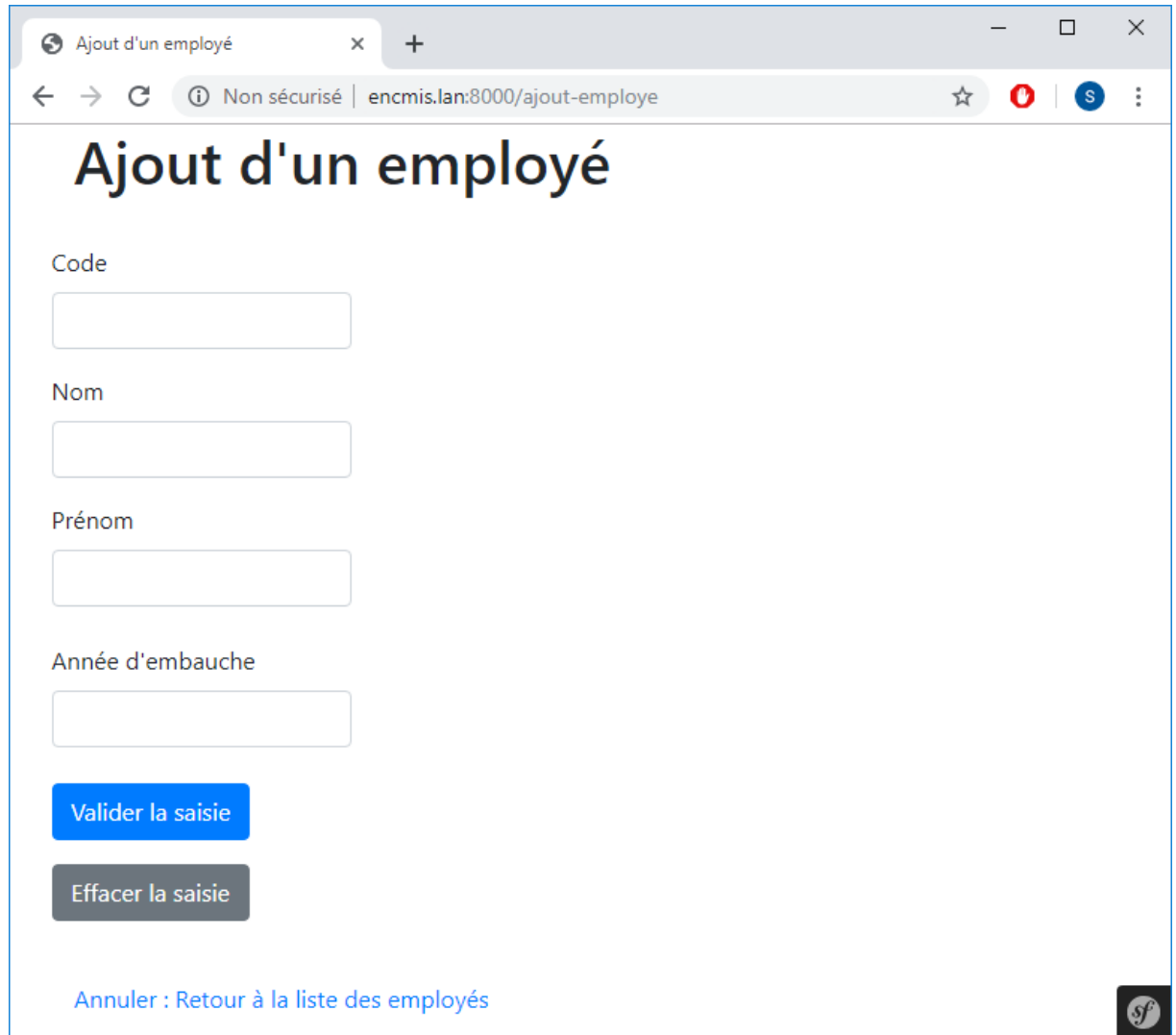
[Ajouter un employé](#)

Le deuxième écran concerne le formulaire d'ajout d'un employé.

L'URL relative est :

/ajout-employe

Voici son affichage :



The screenshot shows a web browser window with the title 'Ajout d'un employé'. The address bar indicates the URL is 'encmis.lan:8000/ajout-employe' and the connection is 'Non sécurisé'. The page content includes a title 'Ajout d'un employé' followed by four input fields labeled 'Code', 'Nom', 'Prénom', and 'Année d'embauche'. Below these fields are two buttons: 'Valider la saisie' (blue) and 'Effacer la saisie' (grey). At the bottom, there is a link 'Annuler : Retour à la liste des employés' and a small Symfony logo in the bottom right corner.

Le troisième écran concerne le formulaire de modification d'un employé.

Si par exemple le code de cet employé, sélectionné dans la liste des employés, est 284B128, l'URL relative est :

/modif-employe/284B128

Voici son affichage :

Modification d'un employé

Code

284B128

Nom

COVER

Prénom

Harry

Année d'embauche

1994

Valider les modifications

[Annuler : Retour à la liste des employés](#)

Dans l'URL, il faut fournir une valeur qui sera **variable** : code de l'employé à modifier.

Au niveau de la route, il faudra créer un paramètre.

Le quatrième écran concerne le formulaire de suppression d'un employé.

Si par exemple le code de cet employé, sélectionné dans la liste des employés, est 284B128, l'URL relative est :

/suppr-employe/284B128

Voici son affichage :

Suppression d'un employé

Code

284B128

Nom

COVER

Prénom

Harry

Année d'embauche

1994

Supprimer cet employé

[Annuler : Retour à la liste des employés](#)

On remarque que les 3 formulaires proposés pour l'ajout, la modification et la suppressions sont les mêmes.

Les formulaires d'ajout et de modification permettent de saisir des informations (à part le code de l'employé pour la modification), alors que celui pour la suppression est juste à titre de vérification des informations.

Au niveau implémentation, il y aura un seul template Twig : le contrôleur qui l'appellera donnera en paramètre l'opération réalisée ce qui permettra de customiser l'en-tête.

## b) squelette du contrôleur

Les 4 routes et les méthodes associées seront écrire dans le contrôleur de nom `EmployeController`.

Rappelons son squelette :

```
<?php

// src/Controller/EmployeController.php

namespace App\Controller;

use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ResetType;

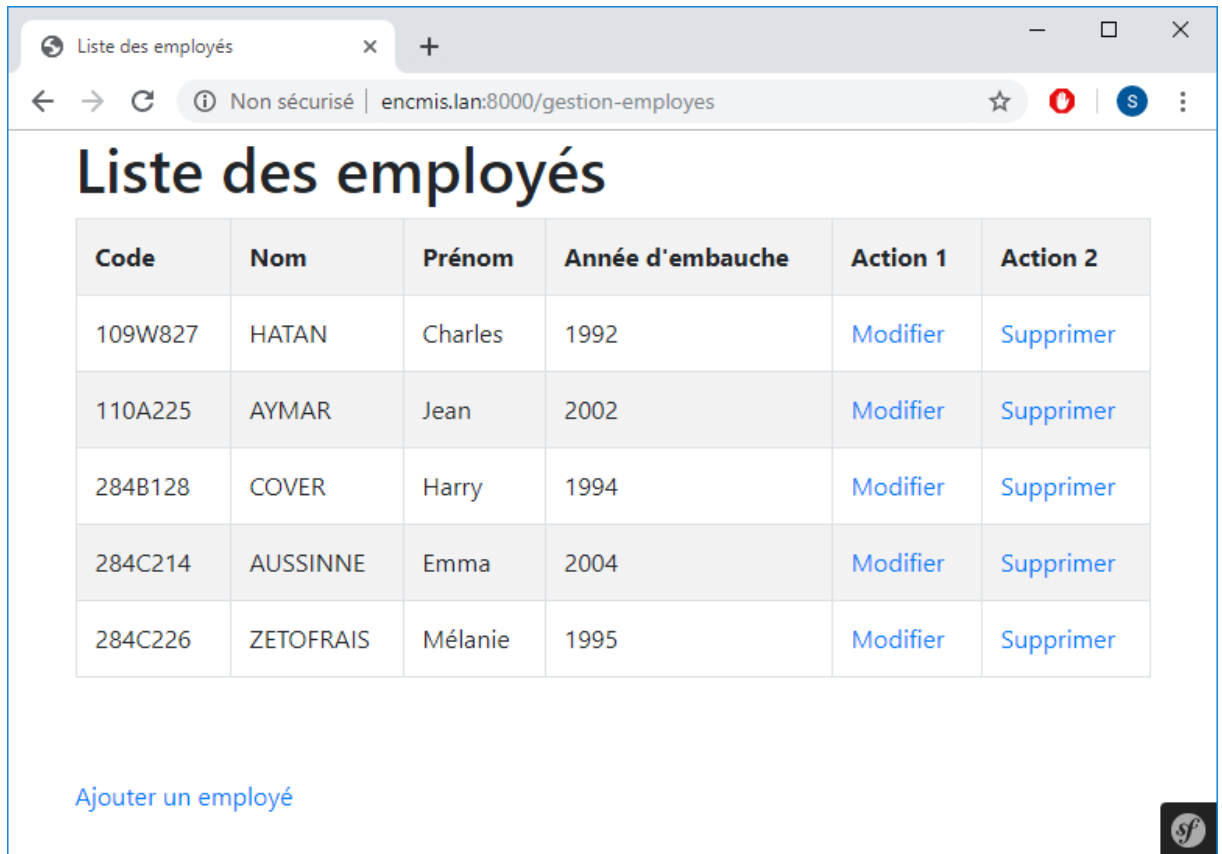
// importation de l'entité
use App\Entity\Employe;

class EmployeController extends AbstractController
{
}
```

A présent, implémentons chacune des options de gestion des employés.

c) mise en place de la liste des employésc-1) méthode du contrôleur et routage

Rappelons l'écran à obtenir :



Code	Nom	Prénom	Année d'embauche	Action 1	Action 2
109W827	HATAN	Charles	1992	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
110A225	AYMAR	Jean	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284B128	COVER	Harry	1994	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C214	AUSSINNE	Emma	2004	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C226	ZETOFRAIS	Mélanie	1995	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajouter un employé](#)

Le code donne :

```
/**
 * @Route("/gestion-employees", name="gestion_employees")
 */
public function index(ManagerRegistry $doctrine)
{
    // récupération du repository relatif à l'entité (classe) Employe
    $repository = $doctrine->getRepository(Employe::class);
    // recherche de tous les employés
    $listeEmployes = $repository->findAll();
    // affichage de la liste des employés
    return $this->render('employe/listeEmployes.html.twig',
        ['employees' => $listeEmployes]);
}
```



Commentaires :

A partir du repository relatif à l'entité `Employe`, on constitue la liste de tous les employés que l'on fournit au template Twig `listeEmployes.html.twig` pour affichage.

Allons-y pour l'écriture de notre template Twig qui sera le fichier le plus conséquent en matière de code !

c-2) template Twig

Voici le code complet du template Twig (le code Twig hors commentaires est en gras) :

```
{# templates/employe/listeEmployes.html.twig #}

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

{% block title %}
    Liste des employés
{% endblock %}

{% block body %}

    <div class="container">

        <h1>Liste des employés</h1>

        {# mise en place du tableau #}
        <table class="table table-bordered table-striped">

            {# mise en place de la ligne de titre #}
            <tr>
                <th>Code</th>
                <th>Nom</th>
                <th>Prénom</th>
                <th>Année d'embauche</th>
                <th>Action 1</th>
                <th>Action 2</th>
            </tr>

            {# affichage de chacune des lignes du tableau #}
```

```

{% for employe in employes %}

    {# affichage de la ligne courante #}
    <tr>
        <td>{{ employe.code }}</td>
        <td>{{ employe.nom }}</td>
        <td>{{ employe.prenom }}</td>
        <td>{{ employe.anEmbauche }}</td>
        <td><a href="
            {{ path('modif_employe',
                    {'codeEmpAction':employe.code}) }}">
            Modifier</a></td>
        <td><a href="
            {{ path('suppr_employe',
                    {'codeEmpAction':employe.code}) }}">
            Supprimer</a></td>
    </tr>

{% endfor %}

    {# fin du tableau #}
</table>

<br /><br />
    {# lien pour ajouter un employé #}
    <a href="{{ path('ajout_employe') }}">
    Ajouter un employé</a>

</div>

{% endblock %}

```

Reprenons les éléments importants.

\* parcours d'un tableau : instruction `{% for ... %}`

La liste des employés s'affiche via un tableau HTML, un employé correspondant à une ligne.

Le contrôleur fournit l'ensemble des employés à travers la variable `employes`. On va donc écrire une boucle de type *for* pour balayer tous les employés.

Voici ce que cela donnait sous PHP (extrait) :

```

<?php foreach ($employes as $employe): ?>
    <tr>

```

```

        <td><?php echo $employe["codeEmp"]; ?></td>
        <td><?php echo $employe["nomEmp"]; ?></td>
        <td><?php echo $employe["prenomEmp"]; ?></td>
        <td><?php echo $employe["anEmbaucheEmp"]; ?></td>
    </tr>
<?php endforeach; ?>

```

Voici le code équivalent avec Twig :

```

{% for employe in employes %}
    <tr>
        <td>{{ employe.code }}</td>
        <td>{{ employe.nom }}</td>
        <td>{{ employe.prenom }}</td>
        <td>{{ employe.anEmbauche }}</td>
    </tr>
{% endfor %}

```

### Commentaires :

On constate d'emblée que le code est plus concis.

La boucle *for* se met en œuvre via les instructions `{% for ... %}` et `{% endfor %}`.

Comme on souhaite un affichage, on utilise l'instruction `{{ ... }}`.

L'accès à un champ du tableau, qui s'appelle `employes` voir code de la méthode du contrôleur précédent paragraphe c-1, se fait simplement avec le point (notation pointée), par exemple `employe.code`.

\* mise en place d'un lien : référencement d'un nom de route avec paramètre éventuel (notation JSON)

Considérons le lien Ajouter un employé.

Son code est :

```

<a href="{ { path('ajout_employe') } }">
Ajouter un employé</a>

```

Il y aura redirection via la fonction `path` vers la route `ajout_employe`.

Considérons le lien Modifier.

Son code est :

```

<a href=
"{{ path('modif_employe',

```

```
{ 'codeEmpAction':employe.code}} } }">Modifier</a>
```

Il y aura redirection vers la route `modif_employe`.

La différence avec l'ajout d'un employé est qu'il faut fournir le code de l'employé à modifier : cela se fait via un paramètre qui est indiqué 'entre accolades.

Pour fournir la valeur de ce paramètre, on utilise la notation JSON (*JavaScript Object Notation*) : nom du paramètre (de la route), deux-points et sa valeur.

Pour le lien Supprimer, même principe.

Son code est :

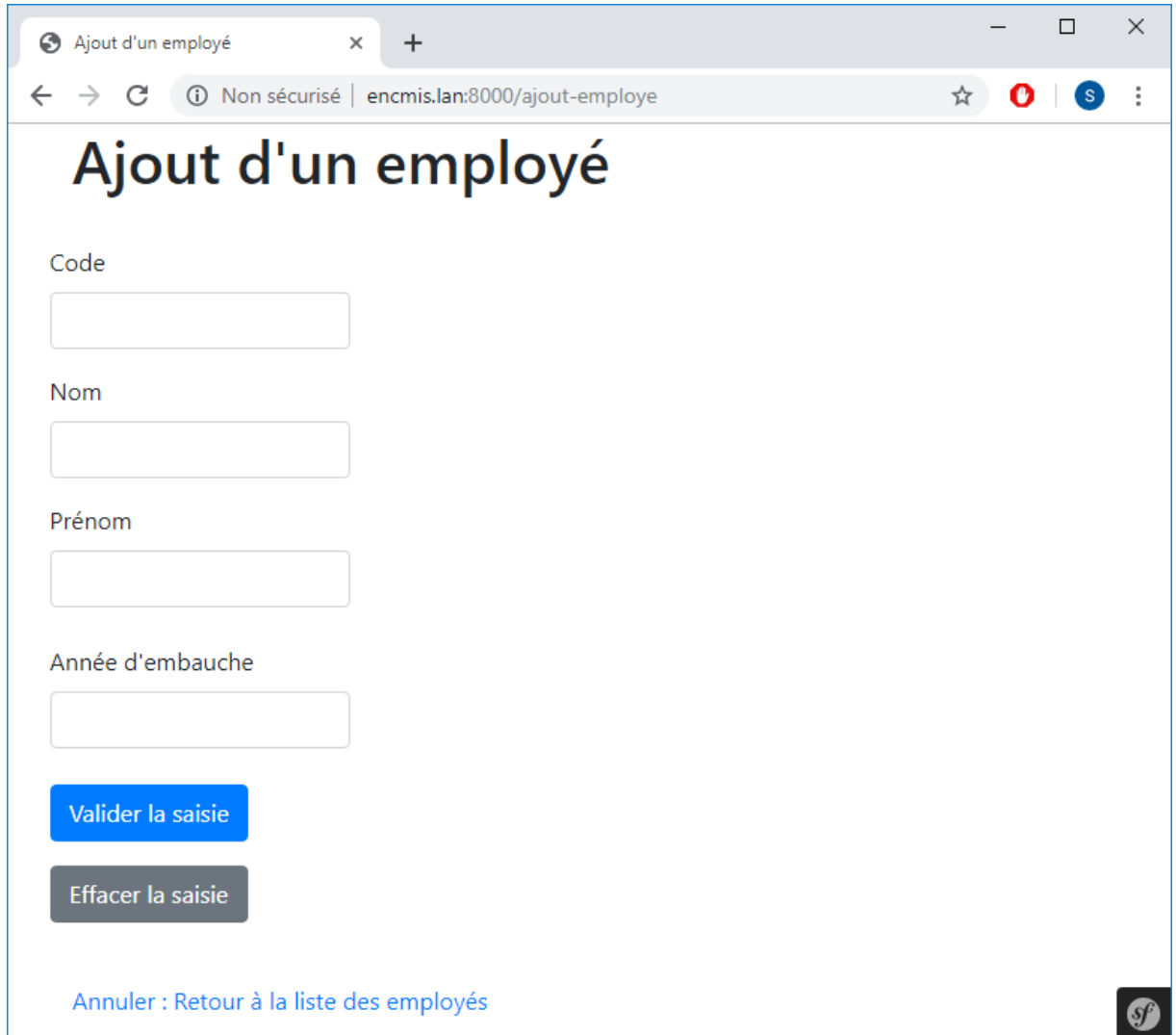
```
<a href=
"{{ path('suppr_employe',
      {'codeEmpAction':employe.code}} } }">Supprimer</a>
```

Il y aura redirection vers la route `suppr_employe` avec le paramètre entre accolades.

## d) ajout d'un employé

### d-1) méthode du contrôleur et routage

Rappelons l'écran à obtenir :



A screenshot of a web browser window. The browser's address bar shows the URL 'encmis.lan:8000/ajout-employe'. The page title is 'Ajout d'un employé'. The form contains four text input fields labeled 'Code', 'Nom', 'Prénom', and 'Année d'embauche'. Below these fields are two buttons: 'Valider la saisie' (blue) and 'Effacer la saisie' (grey). At the bottom of the form, there is a link 'Annuler : Retour à la liste des employés'. A small Symfony logo is visible in the bottom right corner of the page.

Le code donne :

```
/**
 * @Route("/ajout-employe", name="ajout_employe")
 */
public function ajout(ManagerRegistry $doctrine,
                    Request $request)
{
    // instantiation de l'entité Employe
    $employe = new Employe();
```

```

// création du constructeur de formulaire en fournissant l'entité
$formBuilder = $this->createFormBuilder($employe);

/* ajout successif des propriétés souhaitées de l'entité
   pour les champs de formulaire avec leur type */
$formBuilder->add('code', TextType::class)
    ->add('nom', TextType::class)
    ->add('prenom', TextType::class)
    ->add('anEmbauche', TextType::class)
    ->add('validation', SubmitType::class,
        ['label' => 'Valider la saisie'])
    ->add('effacement', ResetType::class,
        ['label' => 'Effacer la saisie']);

// récupération du formulaire à partir du constructeur de formulaire
$form = $formBuilder->getForm();

/* traitement de la requête : Symfony récupère éventuellement les valeurs des
   champs de formulaire et alimente l'objet $employe */
$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid())
    // le formulaire a été soumis et il est valide
    {
        // écriture dans la base de données
        $em = $doctrine->getManager();
        $em->persist($employe);
        $em->flush();

        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);
        // recherche de tous les employés
        $listeEmployes = $repository->findAll();
        // affichage de la liste des employés
        return $this->render('employe/listeEmployes.html.twig',
            ['employes' => $listeEmployes]);
    }

// passage du formulaire au template pour affichage avec l'opération réalisée
return $this->render('employe/formEmploye.html.twig',
    ['form' => $form->createView(),
     'operation' => 'Ajout']);
}

```

### Commentaires :

Nous avons déjà écrit ce contrôleur en très grande partie.

Voyons à nouveau les éléments importants.

Chaque champ de formulaire correspond à une propriété relative à l'entité `Employe` : *mapping*.

Il y a deux cas de figure :

- soit c'est la première fois qu'on invoque la page  
Le contrôleur est exécuté : pour l'instant le formulaire n'a pas été soumis, la méthode `isSubmitted()` relative au formulaire renvoie la valeur `false` et donc la dernière instruction de passage du formulaire au template est exécutée.
- soit l'utilisateur a cliqué sur le bouton *Submit*  
Le contrôleur est exécuté à nouveau : cette fois les champs de formulaire sont remplis et la méthode `handleRequest()` alimente l'entité.  
Le formulaire a été soumis et si les valeurs sont valides (nous verrons comment mettre des règles de validation, par exemple une valeur de champ obligatoire), on écrit l'entité dans la base de données.

On fournit au template Twig le formulaire et l'opération réalisée ici un ajout.

## d-2) template Twig

Voici le code complet du template Twig (le code Twig hors commentaires est en gras) :

```
{# templates/employe/formEmploye.html.twig #}

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

{% block title %}
    {% if operation == "Ajout" %}
        Ajout d'un employé
    {% else %}
        {% if operation == "Modif" %}
            Modification d'un employé
        {% else %}
            Suppression d'un employé
        {% endif %}
    {% endif %}
{% endblock %}

{% block body %}
```

```

<div class="container">

    {% if operation == "Ajout" %}
        <h1>Ajout d'un employé</h1>
    {% else %}
        {% if operation == "Modif" %}
            <h1>Modification d'un employé</h1>
        {% else %}
            <h1>Suppression d'un employé</h1>
        {% endif %}
    {% endif %}
    <br />

    <div class="row">

        {{ form_start(form) }}

        {{ form_row(form.code) }}
        {{ form_row(form.nom) }}
        {{ form_label(form.prenom, "Prénom") }}
        {{ form_widget(form.prenom) }}
        <br />
        {{ form_label(form.prenom, "Année d'embauche") }}
        {{ form_widget(form.anEmbauche) }}
        <br />

        {{ form_end(form) }}

    </div>
    <br />
    {# lien pour revenir à la liste des employés #}
    <a href="{{ path('gestion_employes') }}">
    Annuler : Retour à la liste des employés</a>

</div>

{% endblock %}

```

### Commentaire :

La nouveauté est ici la customisation du titre et de l'en-tête.

On utilise une structure alternative qui se met en œuvre en Twig via les instructions `{% if ... %}`, `{% else %}` et `{% endif %}`.



e) modification d'un employé

Rappelons l'écran à obtenir :

Modification d'un employé

Code

284B128

Nom

COVER

Prénom

Harry

Année d'embauche

1994

Valider les modifications

[Annuler : Retour à la liste des employés](#)

La nouveauté va être dans la présence d'un paramètre dans la route : le code de l'employé à modifier.

Le code donne (la mise en place du paramètre de la route est en gras) :

```
/**
 * @Route("/modif-employe/{codeEmpAction}",
 *       name="modif_employe")
 */
public function modif(ManagerRegistry $doctrine,
                     $codeEmpAction, Request $request)
{
```

```

// récupération du repository relatif à l'entité (classe) Employe
$repository = $doctrine->getRepository(Employe::class);
// recherche de l'employé
$employe = $repository->find($codeEmpAction);

// création du constructeur de formulaire en fournissant l'entité
$formBuilder = $this->createFormBuilder($employe);

/* ajout successif des propriétés souhaitées de l'entité
   pour les champs de formulaire avec leur type */
$formBuilder->add('code', TextType::class,
    ['disabled' => true])
    ->add('nom', TextType::class)
    ->add('prenom', TextType::class)
    ->add('anEmbauche', TextType::class)
    ->add('validation', SubmitType::class,
        ['label' => 'Valider les modifications']);

// récupération du formulaire à partir du constructeur de formulaire
$form = $formBuilder->getForm();

/* traitement de la requête : Symfony récupère éventuellement les valeurs des
   champs de formulaire et alimente l'objet $employe */
$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid())
    // le formulaire a été soumis et il est valide
    {
        // réécriture dans la base de données
        $em = $doctrine->getManager();
        $em->persist($employe);
        $em->flush();

        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);
        // recherche de tous les employés
        $listeEmployes = $repository->findAll();
        // affichage de la liste des employés
        return $this->render('employe/listeEmployes.html.twig',
            ['employes' => $listeEmployes]);
    }

// passage du formulaire au template pour affichage avec l'opération réalisée
return $this->render('employe/formEmploye.html.twig',
    ['form' => $form->createView(),
     'operation' => 'Modif']);
}

```

Commentaires :

Pour la route, il y a cette fois un paramètre : le code de l'employé à modifier.

Cette méthode `modif()` du contrôleur est lancée après clic sur un lien hypertexte provenant du template Twig affichant la liste des employés.

Rappelons le code de ce lien hypertexte (en gras apparaît le nom du paramètre) :

```
<a href=
"{{ path('modif_employe',
        {'codeEmpAction':employe.code}) }}">Modifier</a
```

Le code de l'employé à modifier est transmis via le paramètre noté **codeEmpAction**.

Ce paramètre est récupéré dans le contrôleur via également les accolades ainsi :

```
@Route("/modif-employe/{codeEmpAction}",
        name="modif_employe")
```

Dans la méthode, *le paramètre doit avoir le même nom* avec devant un dollar car c'est une variable PHP. Il s'appelle donc `$codeEmpAction`.

Cela donne :

```
public function modif($codeEmpAction, Request $request)
```

Pour la suite du code, on récupère le formulaire cette fois après avoir pris dans le repository l'entité correspondant au code employé.

Lors de l'ajout du champ de formulaire `code`, on le met en lecture seule en positionnant le paramètre `disabled` à `true`.

Rappelons que chaque champ de formulaire correspond à une propriété relative à l'entité `Employe` : *mapping*.

Il faut traiter la requête et récupérer éventuellement les valeurs des champs de formulaire via la méthode `handleRequest()`.

Dans le cas où on soumet des valeurs valides, on réécrit l'entité dans la base de données.

Rappelons que le template Twig est le même que pour l'ajout d'un employé.

f) suppression d'un employé

Rappelons l'écran à obtenir :

Suppression d'un employé

Code  
284B128

Nom  
COVER

Prénom  
Harry

Année d'embauche  
1994

Supprimer cet employé

[Annuler : Retour à la liste des employés](#)

Le code donne :

```
/**
 * @Route("/suppr-employe/{codeEmpAction}",
 *       name="suppr_employe")
 */
public function suppr(ManagerRegistry $doctrine,
                    $codeEmpAction, Request $request)
{
    // récupération du repository relatif à l'entité (classe) Employe
    $repository = $doctrine->getRepository(Employe::class);
    // recherche de l'employé
    $employe = $repository->find($codeEmpAction);
}
```

```

// création du constructeur de formulaire en fournissant l'entité
$formBuilder = $this->createFormBuilder($employe);

/* ajout successif des propriétés souhaitées de l'entité
   pour les champs de formulaire avec leur type */
$formBuilder->add('code', TextType::class,
    ['disabled' => true])
    ->add('nom', TextType::class,
    ['disabled' => true])
    ->add('prenom', TextType::class,
    ['disabled' => true])
    ->add('anEmbauche', TextType::class,
    ['disabled' => true])
    ->add('suppression', SubmitType::class,
    ['label' => 'Supprimer cet employé']);

// récupération du formulaire à partir du constructeur de formulaire
$form = $formBuilder->getForm();

/* traitement de la requête : Symfony récupère éventuellement les valeurs des
   champs de formulaire et alimente l'objet $employe */
$form->handleRequest($request);

if ($form->isSubmitted())
    // le formulaire a été soumis
    {
        // suppression dans la base de données
        $em = $doctrine->getManager();
        $em->remove($employe);
        $em->flush();

        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);
        // recherche de tous les employés
        $listeEmployes = $repository->findAll();
        // affichage de la liste des employés
        return $this->render('employe/listeEmployes.html.twig',
            ['employes' => $listeEmployes]);
    }

// passage du formulaire au template pour affichage avec l'opération réalisée
return $this->render
    ('employe/formEmploye.html.twig',
    ['form' => $form->createView(),
    'operation' => 'Suppr']);
}

```

### Commentaires :

Pour la route, il y a ici aussi un paramètre : le code de l'employé à supprimer.

Dans la méthode, une fois pris dans le repository l'entité correspondant au code employé, on récupère le formulaire.

Tous les champs du formulaire sont mis en lecture seule.

Dans le cas où on soumet le formulaire, on supprime l'entité dans la base de données.

Le template Twig est le même que pour l'ajout ou la modification d'un employé.

### g) exercice

En vous aidant des éléments de ce paragraphe 7, mettre en place votre site personnel via le framework Symfony.

## 8) Sécurité : gestion des accès utilisateurs

Tout site Web nécessite la mise en place d'une politique de sécurité.

### a) généralités

Pour accéder à une ressource, Symfony se base sur deux éléments :

- l'**authentification** de l'utilisateur au site qui se fait via un pare-feu Symfony : **firewall**
- l'**autorisation** d'accès de cet utilisateur une fois authentifié, qui se fait via le contrôle d'accès : **access control**

Il y a ainsi deux barrages :

- un premier barrage sur l'authentification
- un second barrage sur l'autorisation d'accès

Les différents paramétrages sont consignés dans le fichier **config/packages/security.yaml**.

## b) code du fichier de configuration **security.yaml**

### b-1) code de départ

Voici le contenu principal de ce fichier **security.yaml** consignant donc les paramètres de sécurité (sans certains commentaires) :

```
# config/packages/security.yaml

security:
    enable_authenticator_manager: true
    password_hashers:
        Symfony\Component\Security\Core\User\
            PasswordAuthenticatedUserInterface: 'auto'
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory
    access_control:
        #- { path: ^/admin, roles: ROLE_ADMIN }
        #- { path: ^/profile, roles: ROLE_USER }
```

Le code comprend la section racine `security` stipulant qu'on configure tout ce qui touche à la sécurité et 4 sous-sections que nous allons expliciter.

L'instruction de départ :

```
enable_authenticator_manager: true
```

stipule qu'on active le nouveau système de sécurité.

#### Rappel :

Il faut bien respecter l'indentation qui est de 4 espaces en 4 espaces pour ne pas qu'il y est d'erreur : c'est la spécificité des fichiers d'extension **.yaml** !



## b-2) section password\_hashers

Le code donne :

```
password_hashers:
    Symfony\Component\Security\Core\User\
        PasswordAuthenticatedUserInterface: 'auto'
```

Cette section stipule que pour les mots de passe d'authentification des utilisateurs l'algorithme 'auto' est utilisé, ce qui signifie que Symfony utilisera automatiquement l'algorithme de hachage le plus sécurisé.

## b-3) section providers

Le code donne :

```
providers:
    users_in_memory: { memory: null }
```

Cette section permet de définir les fournisseurs (*providers*) d'utilisateurs.

Il y a un seul fournisseur qui est nommé `users_in_memory`.

Le type du fournisseur d'utilisateurs est `memory` car les informations sur les utilisateurs se trouvent en mémoire c'est à dire directement dans le fichier de configuration, par opposition à une table de base de données, utilisateurs qu'on n'a pas encore définis.

La valeur `null` stipule qu'on n'a pas encore défini d'utilisateurs, en mémoire.

## b-4) section firewalls

Le code donne :

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

Cette section permet de définir les pare-feux.

Un pare-feu va prendre en charge l'authentification de l'utilisateur.

Toute requête, qui se concrétise typiquement par un chemin, par exemple `/ajout-employe` passera obligatoirement au départ par le pare-feu pour savoir si elle est autorisée ou non : ainsi, le pare-feu passe avant la gestion des utilisateurs.

La règle de Symfony est de d'abord lire dans cette section `firewalls` pour toute requête.

Il y a ici 2 pare-feux de noms (arbitraires) respectivement `dev` et `main`.

Lorsqu'il y a plusieurs pare-feux (2 donc ici), Symfony applique les règles du premier pare-feu et s'il n'y a pas de blocage applique les règles du pare-feu suivant.

Considérons le premier pare-feu `dev`.

Le paramètre `pattern` stipule l'expression régulière à appliquer sur le chemin de la requête afin de décider si elle doit passer par le pare-feu ou non : positionnement du paramètre `security` à `true` ou `false`.

Ici, l'expression régulière spécifie que les URL internes à Symfony (comme la barre de débogage) et les fichiers `css`, `images` et `js` ne passent pas par le pare-feu et sont donc accessibles.

Considérons le second pare-feu `main`.

La ligne :

`lazy: true`

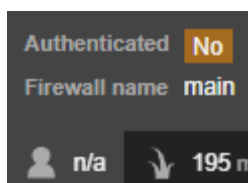
signifie que Symfony empêche le démarrage d'une session s'il n'y a pas besoin d'autorisation.

Et il est stipulé qu'on utilise le fournisseur d'utilisateurs `users_in_memory`.

➔ taper l'URL de notre site **localhost:8000/gestion-employes** (on utilise ici **localhost** comme nom de domaine à la place du nom de domaine **encmis.lan** qui sera celui du site web en production, car on veut tester en local)

➔ aller dans la barre d'état du bas sur **n/a** (**n/a** pour **not authenticated**)

On obtient :



Symfony indique qu'on n'est pas authentifié et que notre pare-feu a pour nom **main**.

### b-5) section access control

On peut définir ici les contrôles d'accès pour les différentes ressources demandées par l'utilisateur.

Mais de manière générale ce contrôle se fait plutôt dans les contrôleurs car il y a une plus grande souplesse.

A présent, occupons nous de la mise en place de l'**authentification** (connexion) des utilisateurs sur le site puis celle des **autorisations** (droits) que chaque utilisateur authentifié pourra avoir.

***Dans l'ordre, le premier barrage est l'authentification et le second barrage sont les autorisations.***

La solution la plus utilisée pour l'authentification passe par un formulaire de connexion.

En plus du paramétrage de l'authentification qui se fait toujours dans ce fichier **security.yaml**, il faudra mettre en place le formulaire avec donc en préalable la création d'un contrôleur.

Symfony en tant que framework va permettre de générer le code correspondant qui sera ensuite à personnaliser (customiser) quelque peu.

### c) éléments pour l'authentification par formulaire de connexion et les commandes de génération automatique de code

L'authentification de l'utilisateur va se faire via un formulaire de connexion.

Sous Symfony, il faut faire deux étapes globales.

La première étape, qui est une étape préliminaire, est de créer les utilisateurs avec leurs autorisations.

Ces informations sur les utilisateurs peuvent être stockées :

- directement dans le fichier **security.yaml** et la section `providers`, plus précisément dans la sous-section `users` de la section `memory` : cette section `memory` stipule que les informations sur les utilisateurs se trouvent en mémoire c'est à dire directement dans le fichier de configuration, par opposition à une table de base de données.
- dans une table de base de données.  
Cette solution, plus souple, est celle qui est retenue fréquemment et nous l'adopterons. Comme pour toute autre table de la base de données, dans notre cas *employees*, il faudra passer par une entité.  
D'autre part, pour que Symfony reconnaisse cette nouvelle entité comme la classe Utilisateur de la couche sécurité, il faut qu'elle implémente notamment l'interface `UserInterface` : `instruction implements`.  
Cette nouvelle entité est souvent nommée par convention `User`.  
Le fichier **security.yaml** devra être mis à jour dans sa section `providers` : il faut stipuler qu'on passe par cette entité `User` pour gérer les utilisateurs.

Via la commande **make:user**, Symfony va générer automatiquement cette classe `User` et mettre à jour le fichier **security.yaml**.

La seconde étape est d'implémenter le code proprement dit pour mettre en œuvre le formulaire de connexion et le processus d'authentification.

Cela supposer de créer :

- le template Twig pour le formulaire,
- le contrôleur correspondant à ce template couplé à une classe qui gère l'authentification.

Et aussi de mettre à jour à nouveau le fichier **security.yaml**.

Via la commande **make:controller**, Symfony va générer automatiquement du code qu'il faudra ensuite modifier.

## d) gestion des utilisateurs

### d-1) génération du code via la commande **make:user**

La génération du code relativement à la gestion des utilisateurs se fait en lançant la commande :

**php bin/console make:user**

Voici ce que cela va donner après réponse à chaque question :

```

Administrateur : Windows PowerShell
PS D:\Programmes\Symfony\encmis> php bin/console make:user

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

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

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

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

created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml

Success!

Next Steps:
- Review your new app\Entity\User class.
- Use make:entity to add more fields to your User entity and then run make:migration.
- Create a way to authenticate! See https://symfony.com/doc/current/security.html
PS D:\Programmes\Symfony\encmis>

```

Dans les questions entre crochets apparaît la valeur par défaut.  
Si on veut la sélectionner on tape directement <Entrée>.

Voici ce que l'on a stipulé ici respectivement :

- la classe pour les utilisateurs s'appelle **User**
- on stocke les données dans la base de données
- le champ de référence pour l'utilisateur est **username**
- on veut un mot de passe

Symfony indique que :

- **src/Entity/User.php** (qui est une entité) a été créé,
- **src/Repository/UserRepository.php** (qui est un repository) a été créé,
- **src/Entity/User.php** (qui est une entité) a été mis à jour,
- **config/packages/security.yaml** (fichier de configuration) a été mis à jour.

**UserRepository.php** permet de récupérer les enregistrements correspondants à l'entité User : c'est un dépôt (repository) de Doctrine.

Nous n'aurons pas à intervenir dans cette classe.

## d-2) code généré du fichier User.php

Voici le code complet généré pour cette classe User :

```
<?php

// src/Entity/User.php

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\
    PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Entity(repositoryClass=UserRepository::class)
 */
class User implements UserInterface,
    PasswordAuthenticatedUserInterface
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $username;

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;
```

```

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

/**
 * @deprecated since Symfony 5.3,
 * use getUserIdentifier instead
 */
public function getUsername(): string
{
    return (string) $this->username;
}

public function setUsername(string $username): self
{
    $this->username = $username;
    return $this;
}

/**
 * A visual identifier that represents this user.
 *
 * @see UserInterface
 */
public function getUserIdentifier(): string
{
    return (string) $this->username;
}

/**
 * @see UserInterface
 */
public function getRoles(): array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    $roles[] = 'ROLE_USER';
    return array_unique($roles);
}

public function setRoles(array $roles): self
{
    $this->roles = $roles;
    return $this;
}

/**
 * @see PasswordAuthenticatedUserInterface
 */

```

```

public function getPassword(): string
{
    return $this->password;
}

public function setPassword(string $password): self
{
    $this->password = $password;
    return $this;
}

/**
 * Returning a salt is only needed, if you are not using a modern
 * hashing algorithm (e.g. bcrypt or sodium) in your security.yaml.
 *
 * @see UserInterface
 */
public function getSalt()
{
    return null;
}

/**
 * @see UserInterface
 */
public function eraseCredentials()
{
    // If you store any temporary, sensitive data on the user, clear it here
    // $this->plainPassword = null;
}
}

```

### Commentaires :

Pour que Symfony reconnaisse cette nouvelle entité comme la classe Utilisateur de la couche sécurité, il faut qu'elle implémente l'interface `UserInterface` et l'interface `PasswordAuthenticatedUserInterface` : instruction `implements`.

L'implémentation d'une interface est globalement la même chose que l'héritage d'une classe sauf qu'ici ***tous les membres de l'interface (attributs et méthodes) doivent être obligatoirement redéclarés ou redéfinis.***



Voici le code de l'interface `ManagerInterface` :

```
interface ManManagerInterface
{
    public function getRoles();
    public function getPassword();
    public function getSalt();
    public function eraseCredentials();
    public function getUsername();
}
```

Ces 5 méthodes doivent donc être redéfinies.

Voici le code de l'interface `PasswordAuthenticatedManagerInterface` :

```
interface PasswordAuthenticatedManagerInterface
{
    public function getPassword();
}
```

Cette unique méthode doit donc être redéfinie.

Dans la classe `User` générée, on retrouve comme attributs :

- `$id` pour l'identifiant de l'utilisateur (qui est un numéro automatique)
- `$username` pour le nom du compte
- `$roles` pour les rôles de l'utilisateur (tableau : présence des crochets)
- `$password` pour le mot de passe

Ensuite, on trouve notamment les méthodes suivantes, à obligatoirement redéfinir.

`getRoles()` renvoie les rôles de l'utilisateur : à noter qu'on retourne obligatoirement au moins un rôle qui est `ROLE_USER`.

`getSalt()` renvoie le sel permettant de crypter éventuellement le mot de passe. Comme il est dit dans le commentaire, cela n'est utile si on n'utilise pas un algorithme de hachage moderne, ce qui n'est pas le cas.

`eraseCredentials()` supprime les identifiants, *credentials* en anglais, de l'utilisateur.

A présent parlons des annotations dans le code.

Nous connaissons déjà l'annotation `\Entity` qui permet de stipuler que la classe est une entité, et donc que Doctrine devra créer une table.

On fournit en plus ici le paramètre `repositoryClass` avec comme valeur la classe `UserRepository`.

L'annotation `@var` permet de dire de quel type est la variable pour la documentation future.

L'annotation `@see` permet de faire référence à un élément structuré, ici une interface.

L'annotation `@deprecated` permet de documenter la version obsolète d'un élément, ici la méthode `getUsername()`.

Dans le commentaire, il est stipulé qu'il faut utiliser à la place la méthode `getUserIdentifier()`, qui contient le même code en l'occurrence il retourne le nom de l'utilisateur.

Si on utilise la méthode obsolète, ici donc `getUsername()`, un message d'avertissement de type dépréciation apparaîtra dans les logs.

### d-3) code modifié du fichier **security.yaml**

Avant de lancer la commande **make:user**, voici quel était le code principal du fichier **security.yaml** :

```
# config/packages/security.yaml

security:
    enable_authenticator_manager: true
    password_hashers:
        Symfony\Component\Security\Core\User\
            PasswordAuthenticatedUserInterface: 'auto'
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory

    access_control:
```

Voici à présent le code de ce fichier du fichier **security.yaml** (les nouveautés sont en gras) :

```
# config/packages/security.yaml

security:
    enable_authenticator_manager: true
    password_hashers:
        Symfony\Component\Security\Core\User\
            PasswordAuthenticatedUserInterface: 'auto'
    providers:
        app_user_provider:
```

```

        entity:
            class: App\Entity\User
            property: username
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
    access_control:

```

### Commentaires :

Dans la section providers, on a un nouveau fournisseur nommé `app_user_provider` qui est de type `entity` : les informations sur les utilisateurs seront consignées désormais dans une table de base de données et non plus directement dans le fichier de configuration d'où la disparition de l'ancien fournisseur de type `memory`.

Cette table est mappée via l'entité `User` et la propriété de référence est `username`.

Dans le pare-feu `main`, on stipule qu'on utilise désormais ce fournisseur `app_user_provider`.

## d-4) customisation de la classe `User` : type d'utilisateur et rôles associés

Nous allons à présent customiser notre classe `User` pour notre contexte.

D'abord, on va nommer la table *utilisateurs* et non pas *user*.

Ensuite, revenons à notre cas de la table *employes*.

Pour notre exemple, on trouvera deux types d'utilisateurs :

- les coordonnateurs : ils ont tous les droits ici sur la table *employes*,
- les secrétaires : ils n'ont que le droit de lecture des informations toujours ici sur la table *employes*.

On devra donc avoir pour notre entité `User` un autre attribut nommé *type* qui sera égal à `S` (pour Secrétaire) ou `C` (pour Coordonnateur).

D'autre part, on prévoira un attribut pour le nom complet et un attribut pour le prénom complet, afin notamment de les afficher lorsque l'utilisateur s'est bien authentifié.

Ensuite dans la table de la base de données qu'on appellera *utilisateurs*, 3 utilisateurs seront créés : 2 secrétaires et 1 coordonnateur.

Voici le contenu attendu :

<i>id</i>	<i>username</i>	<i>roles</i>	<i>password</i>	<i>nom_complet</i>	<i>prenom_complet</i>	<i>type</i>
1	ptronc	[]	Picsou1	TRONC	Paul	S
2	jnastic	[]	Donald2	NASTIC	Jim	C
3	phibulaire	[]	Mickey3	HIBULAIRE	Pat	S

Pour le champ de type json *roles*, on doit mettre sa valeur entre crochets car il peut y en avoir plusieurs (tableau de valeurs) : au départ, on ne met pas de rôle d'où les crochets vides.

Grâce à la méthode `getRoles()`, on a vu que tout utilisateur aura au moins pour rôle `ROLE_USER`.

Nous allons devoir créer deux rôles : celui pour les coordonnateurs et celui pour les secrétaires.

Par convention, un rôle commence toujours par `ROLE_`.

Nous allons ici définir donc deux rôles : `ROLE_COORDO` et `ROLE_SECRET`.

Le rôle `ROLE_COORDO` a plus de droits que le rôle `ROLE_SECRET` : les coordonnateurs auront donc ces deux rôles.

On va devoir réécrire la méthode `getRoles()`.

Voici donc le code complet customisé de la classe `User` (les modifications sont en gras) :

```
<?php

// src/Entity/User.php

namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\
    PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Entity(repositoryClass=UserRepository::class)
 * @ORM\Table(name="utilisateurs")
```

```

*/
class User implements UserInterface,
                        PasswordAuthenticatedUserInterface
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $username;

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;

    /**
     * @ORM\Column(type="string", length=25)
     */
    private $nomComplet;

    /**
     * @ORM\Column(type="string", length=20)
     */
    private $prenomComplet;

    /**
     * @ORM\Column(type="string", length=1)
     */
    private $type; // S pour Secrétaire - C pour Coordonnateur

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

    /**
     * @deprecated since Symfony 5.3,
     * use getUserIdentifier instead

```

```

    */
    public function getUsername(): string
    {
        return (string) $this->username;
    }

    public function setUsername(string $username): self
    {
        $this->username = $username;
        return $this;
    }

    /**
     * A visual identifier that represents this user.
     *
     * @see UserInterface
     */
    public function getUserIdentifier(): string
    {
        return (string) $this->username;
    }

    /**
     * @see UserInterface
     */
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';
        if ($this->type == 'S' || 'C')
            // l'utilisateur a en plus le rôle ROLE_SECRET
            {
                $roles[] = 'ROLE_SECRET';
            }
        if ($this->type == 'C')
            // l'utilisateur a en plus le rôle ROLE_COORD
            {
                $roles[] = 'ROLE_COORDO';
            }
        return array_unique($roles);
    }

    public function setRoles(array $roles): self
    {
        $this->roles = $roles;
        return $this;
    }

    /**
     * @see PasswordAuthenticatedUserInterface

```

```

    */
    public function getPassword(): string
    {
        return $this->password;
    }

    public function setPassword(string $password): self
    {
        $this->password = $password;
        return $this;
    }

    /**
     * Returning a salt is only needed, if you are not using a modern
     * hashing algorithm (e.g. bcrypt or sodium) in your security.yaml.
     *
     * @see UserInterface
     */
    public function getSalt()
    {
        return null;
    }

    public function getNomComplet()
    {
        return $this->nomComplet;
    }
    public function setNomComplet($nomComplet)
    {
        $this->nomComplet = $nomComplet;
    }
    public function getPrenomComplet()
    {
        return $this->prenomComplet;
    }
    public function setPrenomComplet($prenomComplet)
    {
        $this->prenomComplet = $prenomComplet;
    }
    public function getType()
    {
        return $this->type;
    }
    public function setType($type)
    {
        $this->type = $type;
    }

    /**
     * @see UserInterface

```

```
*/  
public function eraseCredentials()  
{  
    // If you store any temporary, sensitive data on the user, clear it here  
    // $this->plainPassword = null;  
}  
  
}
```

Mettons à jour notre base de données, afin donc de créer la nouvelle table *utilisateurs* via les deux commandes correspondantes :

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



### d-5) remplissage de la table avec encodage des mots de passe (hachage et salage)

Pour remplir à présent la table *utilisateurs*, on va utiliser l'ORM Doctrine : écriture de code PHP avec invocation de méthodes de la classe `User` précédente.

Mais pour qu'ensuite l'authentification des utilisateurs sous Symfony puisse se réaliser, les mots de passe en clair, comme par exemple **Picsou1** pour le premier compte, doivent être **encodés**, sachant que *Symfony va à la fois hacher et saler le mot de passe*, à travers la classe `UserPasswordHasherInterface` qui sera injecté via une instance dans la méthode de remplissage de la table.

Dans un premier temps, il faut configurer le hachage pour les mots de passe relativement à la classe `User` dans le fichier **config/packages/security.yaml**.

La nouvelle ligne donne (en gras) :

```
password_hashers:
    Symfony\Component\Security\Core\User\
        PasswordAuthenticatedUserInterface: 'auto'
    App\Entity\User: 'auto'
```

On stipule que pour les mots de passe de la classe `User` l'algorithme 'auto' est utilisé, ce qui signifie que Symfony utilisera automatiquement l'algorithme de hachage le plus sécurisé.

Dans un second temps, créons un contrôleur nommé `UserController` pour effectuer le remplissage de la table *utilisateurs* avec donc les mots hachés (et salés).

Le code complet donne :

```
<?php

// src/Controller/EmployeController.php

namespace App\Controller;

use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

// importation de l'entité
```

```

use App\Entity\User;

class UserController extends AbstractController
{

    /**
     * @Route("/rempl-utilis")
     */
    public function remplissageUtilisateurs
        (ManagerRegistry $doctrine,
         UserPasswordHasherInterface $passwordHasher)
    {

        // récupération de l'Entity Manager
        $em = $doctrine->getManager();

        // création de l'utilisateur 1
        $user = new User();
        $user->setUsername("ptronc");
        $user->setNomComplet("TRONC");
        $user->setPrenomComplet("Paul");
        $user->setType("S");
        // hachage du mot de passe en clair
        $hashedPassword =
            $passwordHasher->hashPassword($user, "Picsoul");
        $user->setPassword($hashedPassword);
        $em->persist($user);

        // création de l'utilisateur 2
        $user = new User();
        $user->setUsername("jnastic");
        $user->setNomComplet("NASTIC");
        $user->setPrenomComplet("Jim");
        $user->setType("C");
        // hachage du mot de passe en clair
        $hashedPassword =
            $passwordHasher->hashPassword($user, "Donald2");
        $user->setPassword($hashedPassword);
        $em->persist($user);

        // création de l'utilisateur 3
        $user = new User();
        $user->setUsername("phibulaire");
        $user->setNomComplet("HIBULAIRE");
        $user->setPrenomComplet("Pat");
        $user->setType("S");
        // hachage du mot de passe en clair
        $hashedPassword =
            $passwordHasher->hashPassword($user, "Mickey3");
    }
}

```

```

$user->setPassword($hashedPassword);
$em->persist($user);

$em->flush();

return new Response("3 utilisateurs insérés");
}
}

```

### Commentaires :

On injecte un objet de la classe `UserPasswordHasherInterface` pour pouvoir hacher (et saler) le mot de passe.

Dans la méthode, on crée 3 objets `User`.

Pour encoder le mot de passe, on utilise la méthode `hashPassword()` de cette classe `UserPasswordHasherInterface` : elle reçoit en paramètres l'objet concerné et le mot de passe en clair, et renvoie le mot de passe haché qui sera écrit dans la table via ici le setter `setPassword()`.

Via l'URL **localhost:8000/rempl-utilis**, la table *utilisateurs* est remplie.

Cela donne :

id	username	roles	password	nom_complet	prenom_complet	type
1	ptronc	[]	\$2y\$13\$5GAMI6uuTx8GHseMXEoyYet8gRDR5NZfwm1hDMdw06d...	TRONC	Paul	S
2	jnastic	[]	\$2y\$13\$f9IQW3uqfvsuPaoCIsabJ.vbEhLV0PAjEv/L0FopiC4...	NASTIC	Jim	C
3	phibulaire	[]	\$2y\$13\$OG0IJxMOFwClar20QnLah.YFnHB4fUFaAP3Da9/0xQ3...	HIBULAIRE	Pat	S

On voit que les 3 mots de passe sont hachés (et salés).

L'utilisateur lui tapera bien sûr le mot de passe en clair de départ, soit par exemple pour le premier compte **Picsou1**.

Notre première étape, préliminaire, de gestion des utilisateurs est finie.

A présent, passons à la seconde étape avec le formulaire de connexion et l'opération d'authentification.

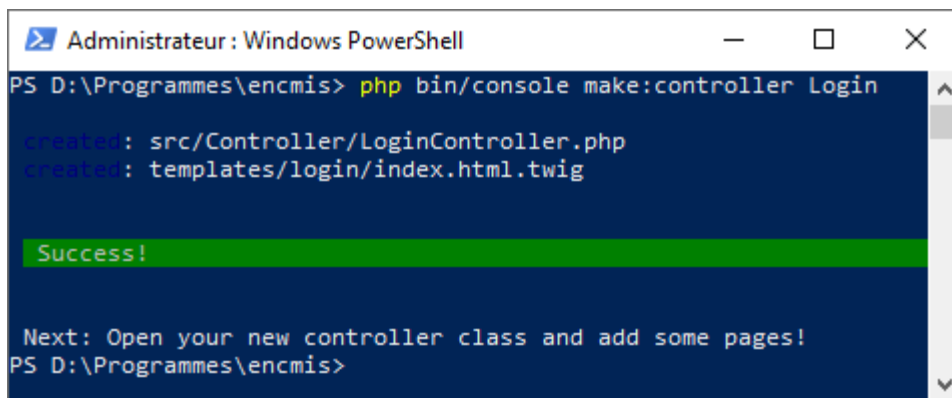
## e) mise en place du formulaire de connexion avec la gestion de l'authentification

### e-1) génération du code via la commande **make:controller**

La génération du code relativement à la mise en œuvre du formulaire de connexion avec contrôleur correspondant qu'on nommera **Login** et du processus d'authentification se font en lançant la commande suivante :

**php bin/console make:controller Login**

Cela va donner :



```

Administrateur : Windows PowerShell
PS D:\Programmes\encmis> php bin/console make:controller Login

created: src/Controller/LoginController.php
created: templates/login/index.html.twig

Success!

Next: Open your new controller class and add some pages!
PS D:\Programmes\encmis>

```

Symfony indique que deux fichiers ont été créés :

- **src/Controller/LoginController.php**, qui est un contrôleur
- **templates/login/index.html.twig**, qui est un template Twig

### e-2) code généré et lancement

Voici le code généré du fichier **LoginController.php** (les éléments les plus importants sont en gras) :

```

<?php

// src/Controller/LoginController.php

namespace App\Controller;

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

```

```

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

```

### Commentaires :

La méthode `index()` lance le template Twig `index.html.twig` avec le paramètre `controller_name` qui a comme valeur le nom de ce contrôleur

L'URL de la route associée est `/login` et son nom est `app_login`.

Voici un extrait du code généré du fichier **index.html.twig** :

```

{# templates/login/index.html.twig #}

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

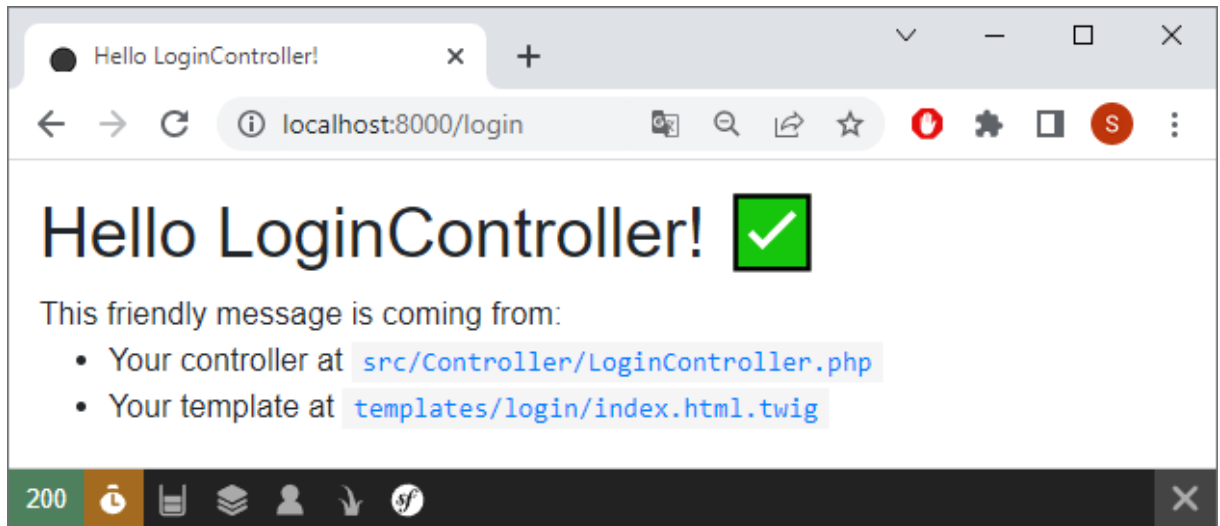
{% block title %}Hello LoginController!{% endblock %}

{% block body %}
    <h1>Hello {{ controller_name }}! □</h1>
    This friendly message is coming from:
    <ul>
        <li>Your controller at <code>
            <a href="...">
                src/Controller/LoginController.php</a></code>
        </li>
        <li>Your template at <code>
            <a href="...">
                templates/login/index.html.twig</a></code>
        </li>
    </ul>
{% endblock %}

```

Le code ne présente pas d'intérêt majeur.

Voici ce qui s'affiche lorsqu'on tape l'URL **localhost:8000/login** :



A la place de l'affichage de ce message "amical", deux choses sont à faire :

- l'affichage d'un formulaire de connexion où on demande de saisir le nom utilisateur et le mot de passe
- la vérification que les identifiants, credentials en anglais, sont valides.  
*Cette vérification va se faire automatiquement par Symfony* comme nous allons le voir.

### e-3) modification du fichier de configuration security.yaml

Dans la section `main`, il faut ajouter un élément spécifiant quel est le nom de la route correspondant au contrôleur lançant le formulaire de connexion du site.

Cet élément a pour **nom prédéfini** `form_login` avec ses deux sous-éléments `login_path` et `check_path`.

Le nom de la route doit être ici `app_login` : voir notre contrôleur `LoginController` généré précédemment.

Cela donne (les nouvelles lignes sont en gras) :

```
main:
    lazy: true
    provider: app_user_provider
    form_login:
        login_path: app_login
        check_path: app_login
```

Symfony va *automatiquement* lire cet élément `form_login` pour lancer la première page du site.

e-4) modification du contrôleur **LoginController.php**

Le fichier devient (les nouveaux éléments sont en gras) :

```
<?php

// src/Controller/LoginController.php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class LoginController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function index
        (AuthenticationUtils $authenticationUtils): Response
    {

        // récupération de l'erreur d'authentification éventuelle
        $error =
            $authenticationUtils->getLastAuthenticationError();

        // récupération du nom d'utilisateur éventuel
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render('login/index.html.twig',
            ['controller_name' => 'LoginController',
             'last_username' => $lastUsername,
             'error' => $error,]);

    }
}
```

Commentaires :

Pour gérer l'authentification via Symfony, on injecte une instance de la classe AuthenticationUtils.

Dans le cas où l'utilisateur a déjà tenté de se connecter, mais a échoué (par exemple le nom d'utilisateur est incorrect), on récupère l'erreur d'authentification ainsi que le nom de l'utilisateur saisi.

On lance le template Twig `index.html.twig` avec ces deux informations récupérées, qui devra être le formulaire de connexion. On va devoir réécrire ce template Twig.

### e-5) modification du template **index.html.twig** (formulaire de connexion)

Le fichier devient (les éléments importants sont en gras) :

```
{# templates/login/index.html.twig #}
{% extends 'base.html.twig' %}

{% block title %}
    Formulaire de connexion
{% endblock %}

{% block body %}

    <div class="container">

        {% if error %}
            <em>Nom d'utilisateur ou mot de passe incorrect</em>
        {% endif %}

        <h1>Formulaire de connexion</h1><br />

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

            <p>
                <label for="password">Mot de passe :</label><br />
                <input type="password" id="password"
                    name="_password" />
            </p>
            <br /><br />

            <input type="hidden"
                name="_target_path" value="/gestion-employes" />
            <p>
```



```

        <input type="submit" value="Se connecter" />
    </p>
</form>

</div>

{% endblock %}

```

### Commentaires :

S'il y a erreur (variable `error` alimentée dans le contrôleur), un message est affiché.

On affiche le nom éventuel de l'utilisateur s'il a déjà tenté de se connecter (variable `last_username` alimentée dans le contrôleur).

Les 2 champs de type texte correspondant au nom utilisateur et au mot de passe **doivent avoir un nom prédéfini** pour que Symfony les reconnaissent : ils s'appellent respectivement `_username` et `_password`.

***Symfony cherchera alors automatiquement lors de la soumission du formulaire si le nom utilisateur et le mot de passe existent bien dans la table utilisateurs.***

Deux cas de figure :

- soit l'utilisateur a réussi à s'authentifier  
 Symfony va alors rediriger vers le chemin indiqué par le champ de texte caché qui lui aussi **a un nom prédéfini** : il s'appelle `_target_path`.  
 L'attribut `value` fournit le chemin souhaité ici donc `/gestion-employees` pour notre site d'exemple (page d'accueil).
- soit l'utilisateur n'a pas réussi à s'authentifier  
 Il y a redirection vers le nom de la route dont le nom est `app_login` qui correspond à la méthode `index()` de notre contrôleur de départ `LoginController`.

## f) tests de connexion

Faisons les différents tests.

Au départ l'URL donne :  
**localhost:8000/login**

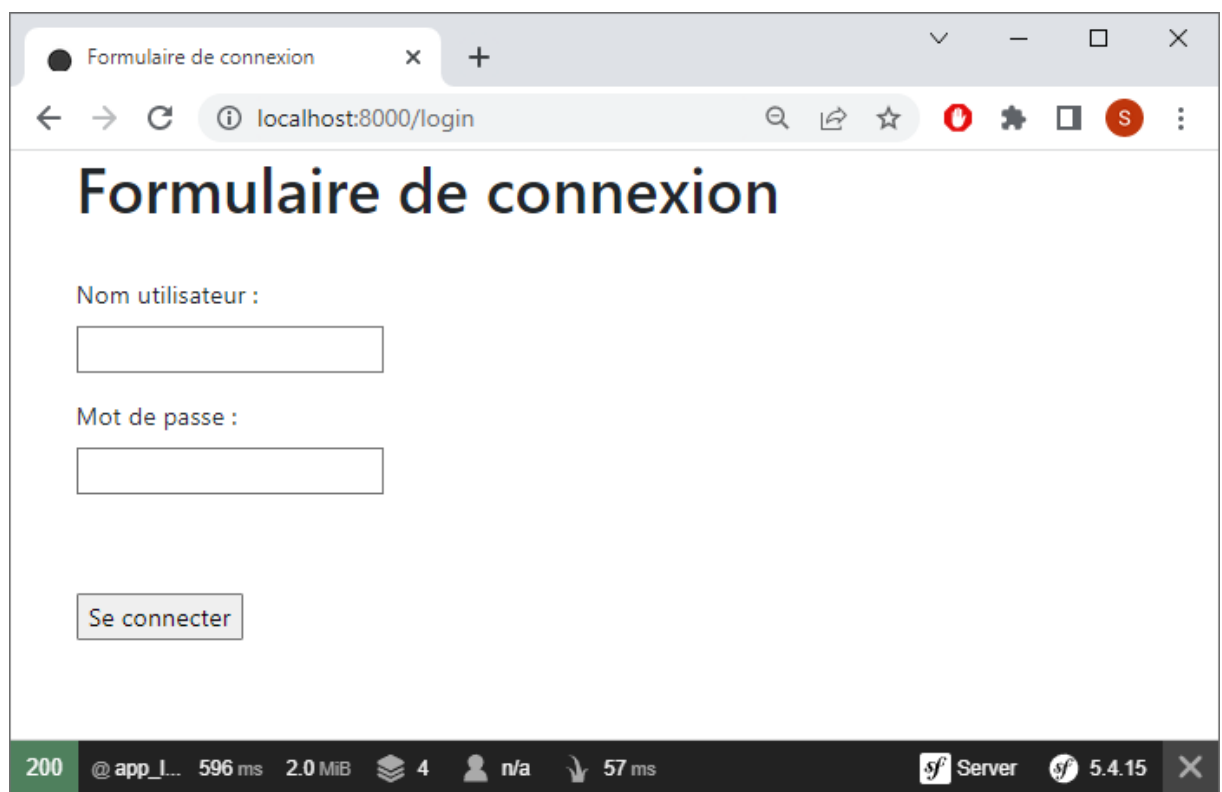
### Rappel :

On utilise ici **localhost** comme nom de domaine à la place du nom de domaine **encmis.lan** qui sera celui du site web en production, car on veut tester en local.

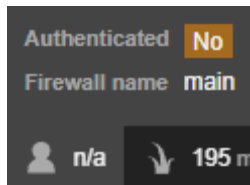
La méthode `index()` du contrôleur `LoginController` associée à l'URL **/login** est exécutée.

Elle affiche le formulaire de connexion.

On obtient l'écran suivant :



Dans la barre d'état en bas, on peut vérifier qu'on n'est non authentifié :

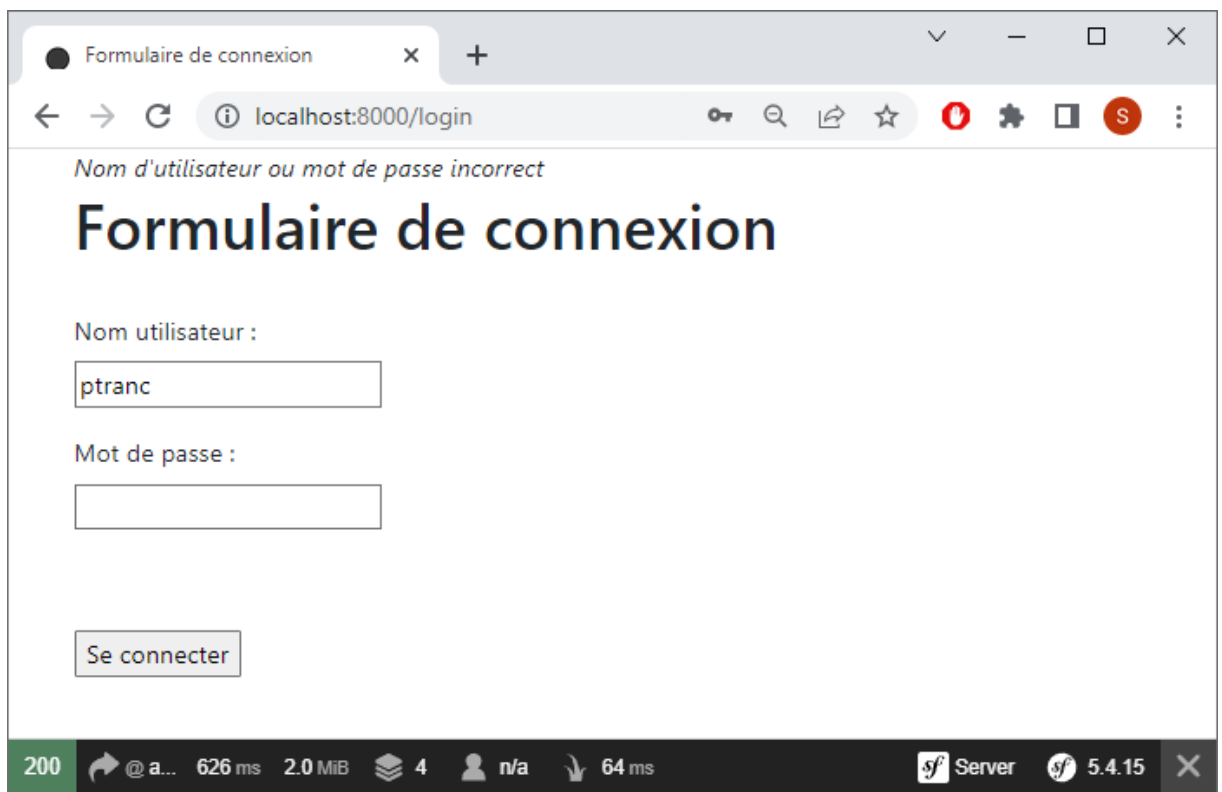


On retrouve aussi le nom de notre pare-feu `main`.

Les bons identifiants, *credentials* en anglais, sont notamment **ptronc** pour le nom utilisateur et **Picsou1** pour le mot de passe.

L'utilisateur se trompe sur le nom utilisateur : il tape **ptranc**.

Cela donne :



Le message d'erreur est affiché et le nom utilisateur saisi précédemment est affiché.

Si l'utilisateur fournit les bons identifiants, il y a redirection vers la route `/gestion-employees` avec l'affichage de la liste des employés : page d'accueil.

Cela donne :

The screenshot shows a web browser window with the address bar displaying 'localhost:8000/gestion-employees'. The page title is 'Liste des employés'. Below the title is a table with the following data:

Code	Nom	Prénom	Année d'embauche	Action 1	Action 2
109W827	HATAN	Charles	1992	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
110A225	AYMAR	Jean	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284B128	COVER	Harry	1994	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C214	AUSSINNE	Emma	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C226	ZETOFRAIS	Mélanie	1995	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

Below the table, there is a link: [Ajouter un employé](#).

The status bar at the bottom shows: 200, @ g..., 965 ms, 4.0 MiB, 14, ptronc, 70 ms, 2, Server, 5.4.15.

Dans la barre d'état en bas, on peut vérifier qu'on est bien connecté cette fois en tant que **ptronc**.

Si on va dans la barre d'état du bas sur le compte **ptronc**, on obtient ceci :

The screenshot shows a user profile dropdown menu for the user 'ptronc'. The menu contains the following information:

- Logged in as: ptronc
- Authenticated: Yes
- Roles: ROLE\_USER + 1 more
- Token class: UsernamePasswordToken
- Firewall name: main

At the bottom of the menu, there is a small summary bar showing: ptronc, 70 ms, 2.

On retrouve le compte **ptronc** et toujours le nom du pare-feu **main**.

Il est indiqué que cet utilisateur a le rôle par défaut **ROLE\_USER** et un autre rôle en l'occurrence **ROLE\_SECRET** car c'est un secrétaire.

A noter la présence d'un jeton (*token*) qui gère en interne l'authentification.

Cette connexion sera valable pendant toute la durée de la session de l'utilisateur, donc tant que le navigateur n'est pas fermé.

A présent mettons en place le processus de déconnexion.

## g) mise en place de la déconnexion

### g-1) problématique

Pour se déconnecter, l'utilisateur doit pour l'instant fermer son navigateur.

Mais cela n'est pas forcément pratique :

- l'utilisateur a peut être d'autres fenêtre ouvertes sous ce navigateur et il veut les conserver telles qu'elle,
- l'utilisateur doit pouvoir se connecter sous un autre compte à partir de la fenêtre de consultation des employés, page centrale.

Pour mettre en place l'opération de déconnexion, il va falloir modifier le code dans :

- le fichier de configuration **security.yaml**
- le contrôleur gérant l'authentification **LoginController.php**
- le template affichant la liste des employés **listeEmployes.html.twig**

### g-2) modification du fichier de configuration security.yaml

Dans la section `main`, il faut ajouter un élément spécifiant quel est le nom de la route correspondant à la déconnexion.

Cet élément a pour ***nom prédéfini*** `logout`.

Le nom de la route sera ici `app_logout`.

Cela donne (les nouvelles lignes sont en gras) :

```
main:
    lazy: true
    provider: app_user_provider
    form_login:
        login_path: app_login
        check_path: app_login
    logout:
        path: app_logout
```

### g-3) modification du contrôleur **LoginController.php**

Il faut rajouter une méthode liée à ce chemin `app_logout`.

Cette méthode nommée par exemple `logout()` peut être vide.

Cela donne :

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

### g-4) modification du template **listeEmployes.html.twig**

Il faut rajouter un lien pour la déconnexion avec donc une redirection vers le chemin `app_logout`.

Cela donne (les nouvelles lignes sont en gras) :

```
<br /><br />
{# lien pour ajouter un employé #}
<a href="{{ path('ajout_employe') }}">
Ajouter un employé</a>

<br /><br />
{# lien pour se déconnecter #}
<a href="{{ path('app_logout') }}">
Se déconnecter</a>
```

### g-5) test

La page d'accueil devient :

Liste des employés

Code	Nom	Prénom	Année d'embauche	Action 1	Action 2
109W827	HATAN	Charles	1992	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
110A225	AYMAR	Jean	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284B128	COVER	Harry	1994	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C214	AUSSINNE	Emma	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C226	ZETOFRAIS	Mélanie	1995	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

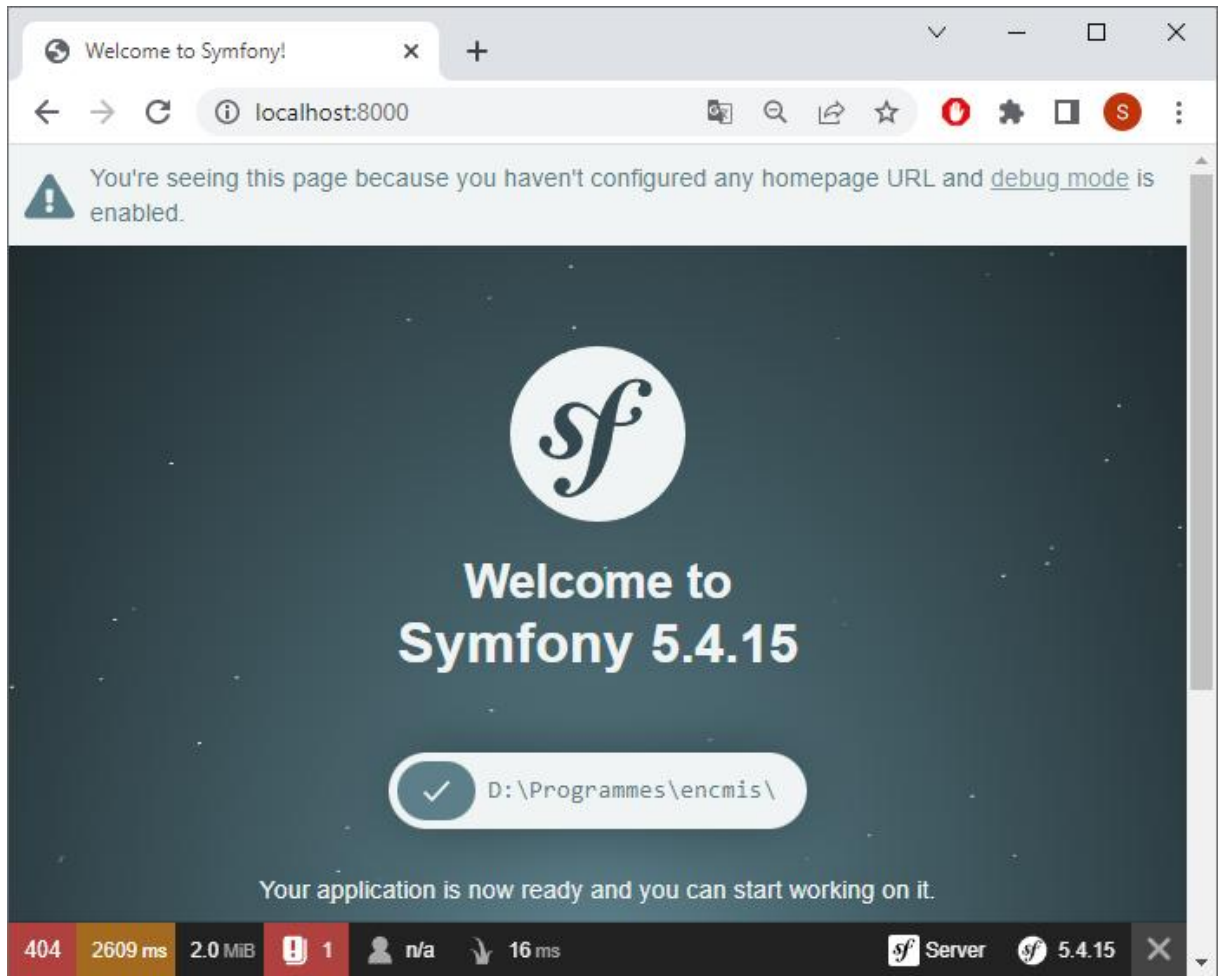
[Ajouter un employé](#)

[Se déconnecter](#)

200 @ g... 834 ms 4.0 MiB 14 ptronc 70 ms 2 Server 5.4.15

Le clic sur le lien Se déconnecter donne :





La déconnexion a marché.

Pour se reconnecter, il faut retaper l'URL **localhost:8000/login**, qui sera mise en pratique en favori.

A présent une fois donc la mise en place complète de l'authentification réalisée, occupons-nous des autorisations.

## h) les autorisations via les rôles

La solution pour mettre en place les autorisations est d'utiliser les rôles.

### h-1) rappel des éléments sur les autorisations pour notre site d'exemple

Pour notre exemple, on trouve deux types d'utilisateurs :

- les coordonnateurs : ils ont tous les droits ici sur la table *employes*,
- les secrétaires : ils n'ont que le droit de lecture des informations toujours ici sur la table *employes*.

Un attribut nommé *type* est égal à S (pour Secrétaire) ou C (pour Coordonnateur).

3 utilisateurs ont été créés : 2 secrétaires et 1 coordonnateur.

Voici le contenu de la table *utilisateurs* (on mettra les mots de passe de départ en clair sans donc leur hachage) :

<i>id</i>	<i>username</i>	<i>roles</i>	<i>password</i>	<i>nom_complet</i>	<i>prenom_complet</i>	<i>type</i>
1	ptronc	[]	Picsou1	TRONC	Paul	S
2	jnastic	[]	Donald2	NASTIC	Jim	C
3	phibulaire	[]	Mickey3	HIBULAIRE	Pat	S

Rappelons le code de la méthode `getRoles()` de la classe `User` qui mappe la table *utilisateurs*.

```
public function getRoles(): array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    $roles[] = 'ROLE_USER';
    if ($this->type == 'S' || 'C')
        // l'utilisateur a en plus le rôle ROLE_SECRET
        {
            $roles[] = 'ROLE_SECRET';
        }
    if ($this->type == 'C')
        // l'utilisateur a en plus le rôle ROLE_COORD

```

```
{  
    $roles[] = 'ROLE_COORDO';  
}  
return array_unique($roles);  
}
```

Tout utilisateur aura au moins pour rôle `ROLE_USER`.

Nous créons ici deux rôles : celui pour les coordonnateurs `ROLE_COORDO` et celui pour les secrétaires `ROLE_SECRET`.

Le rôle `ROLE_COORDO` a plus de droits que le rôle `ROLE_SECRET` : les coordonnateurs auront donc ces deux rôles.

## h-2) écrans à afficher

Comme, on a deux types d'utilisateurs, on aura deux modèles d'écran.

Pour les coordonnateurs, on aura toujours le même écran qui est :

Liste des employés

Code	Nom	Prénom	Année d'embauche	Action 1	Action 2
109W827	HATAN	Charles	1992	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
110A225	AYMAR	Jean	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284B128	COVER	Harry	1994	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C214	AUSSINNE	Emma	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C226	ZETOFRAIS	Mélanie	1995	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

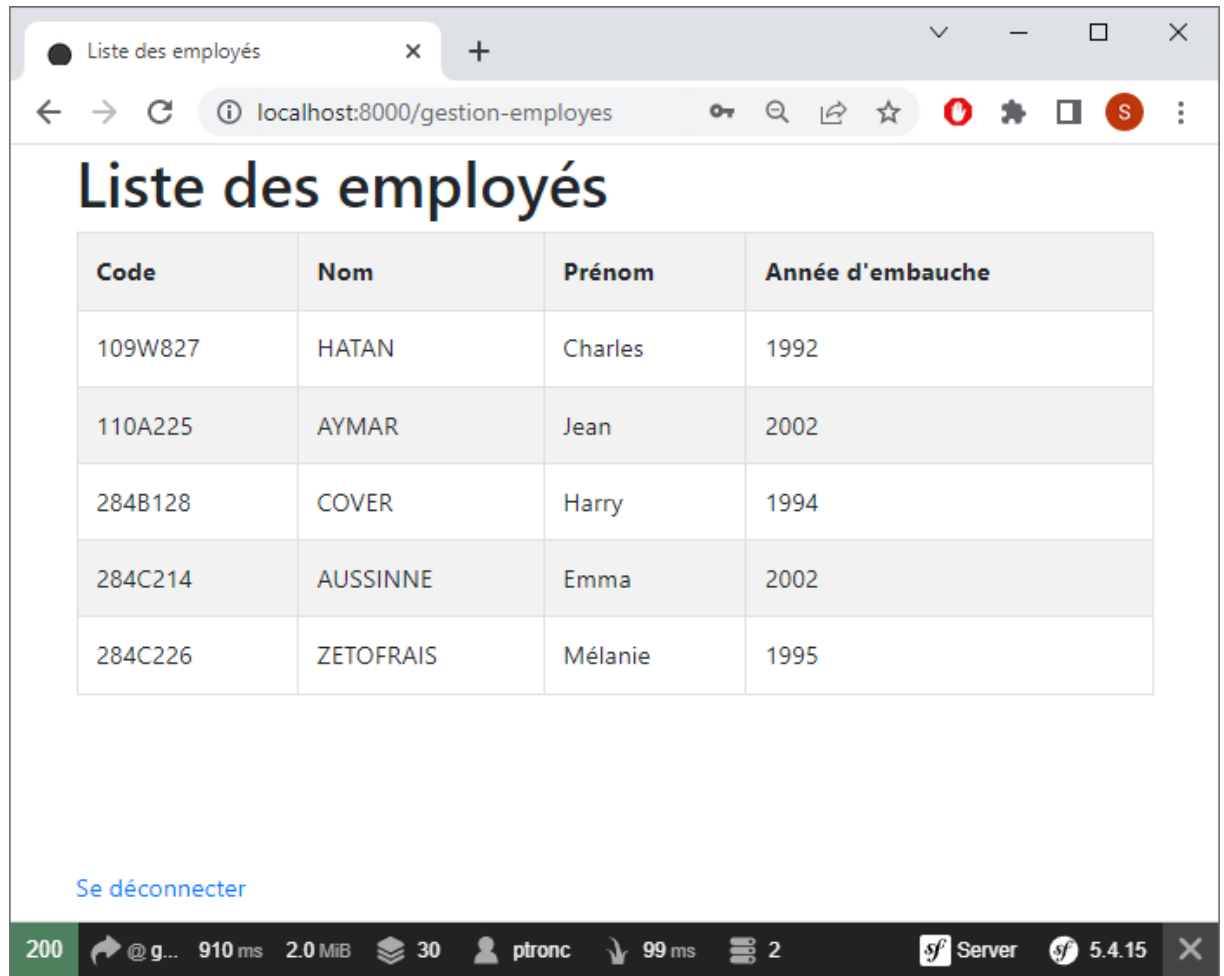
[Ajouter un employé](#)

[Se déconnecter](#)

200 @ g... 828 ms 4.0 MiB 30 jnastic 123 ms 2 Server 5.4.15

Les coordonnateurs doivent avoir tous les droits sur les informations des employés.

Par contre pour les secrétaires, l'écran sera :



Code	Nom	Prénom	Année d'embauche
109W827	HATAN	Charles	1992
110A225	AYMAR	Jean	2002
284B128	COVER	Harry	1994
284C214	AUSSINNE	Emma	2002
284C226	ZETOFRAIS	Mélanie	1995

[Se déconnecter](#)

Les secrétaires ne doivent avoir que le droit de lecture sur les informations des employés.

Les tests des autorisations via les rôles peut se faire soit dans un contrôleur, soit dans un template Twig.

Si ici ces tests ne concerneront en pratique que les templates, voyons à titre d'information comment le faire dans un contrôleur.

### h-3) test des autorisations dans un contrôleur : méthode `isGranted()`

Au lieu de proposer au départ le tableau des employés après l'authentification d'un utilisateur, on va juste afficher son type : méthode `index()` du contrôleur `EmployeController`.

En préliminaire, on fera l'inclusion suivante :

```
use Symfony\Component\HttpFoundation\Response;
```

Et on n'oubliera pas de mettre en commentaires le code de départ de la méthode `index()` afin de pouvoir y revenir ultérieurement.

La méthode correspondante `index()` devient :

```
/**
 * @Route("/gestion-employes", name="gestion_employes")
 */
public function index()
{
    // test d'autorisation
    if ($this->isGranted('ROLE_COORDO'))
    {
        return new Response("Vous êtes un coordonnateur.");
    }
    if ($this->isGranted('ROLE_SECRET'))
    {
        return new Response("Vous êtes un secrétaire.");
    }
    return new Response("Vous n'êtes ni un coordonnateur
                        ni un secrétaire.");
}
```

#### Commentaires :

Le test d'autorisation se fait en invoquant la méthode `isGranted()` avec comme paramètre le rôle.

On reviendra à la méthode `index()` d'origine.

#### h-4) test des autorisations dans un template : fonction `is_granted()`

Réalisons le même affichage que précédemment mais cette fois depuis un template Twig.

Le code donne (à ne pas taper, se contenter de le comprendre) :

```
{% if is_granted('ROLE_COORDO') %}
    Vous êtes un coordonnateur.
{% else %}
    {% if is_granted('ROLE_SECRET') %}
        Vous êtes un secrétaire.
    {% else %}
        Vous n'êtes ni un coordonnateur ni un secrétaire.
    {% endif %}
{% endif %}
```

#### Commentaires :

Le test d'autorisation se fait via la fonction `is_granted()` avec comme paramètre le rôle.

## h-5) prise en compte des autorisations dans notre cas d'exemple

Implémentons maintenant notre cas d'exemple : fichier **listeEmployes.html.twig**.

On ne doit afficher l'en-tête des deux actions de modification et de suppression, les liens Modifier et Supprimer, et le lien Ajouter un employé que si l'utilisateur est de type Coordonnateur.

Le code devient (les nouveautés sont en gras) :

```
{# templates/employe/listeEmployes.html.twig #}

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

{% block title %}
    Liste des employés
{% endblock %}

{% block body %}

    <div class="container">

        <h1>Liste des employés</h1>

        {# mise en place du tableau #}
        <table class="table table-bordered table-striped">

            {# mise en place de la ligne de titre #}
            <tr>
                <th>Code</th>
                <th>Nom</th>
                <th>Prénom</th>
                <th>Année d'embauche</th>
                {% if is_granted('ROLE_COORDO') %}
                <th>Action 1</th>
                <th>Action 2</th>
                {% endif %}
            </tr>

            {# affichage de chacune des lignes du tableau #}

            {% for employe in employes %}

                {# affichage de la ligne courante #}
                <tr>
                    <td>{{ employe.code }}</td>
```



```

        <td>{{ employe.nom }}</td>
        <td>{{ employe.prenom }}</td>
        <td>{{ employe.anEmbauche }}</td>
        {% if is_granted('ROLE_COORDO') %}
            <td><a href="
                {{ path('modif_employe',
                    {'codeEmpAction':employe.code}) }}">
                Modifier</a></td>
            <td><a href="
                {{ path('suppr_employe',
                    {'codeEmpAction':employe.code}) }}">
                Supprimer</a></td>
        {% endif %}
    </tr>

    {% endfor %}

    {# fin du tableau #}
</table>

<br /><br />
{% if is_granted('ROLE_COORDO') %}
    {# lien pour ajouter un employé #}
    <a href="{{ path('ajout_employe') }}">
    Ajouter un employé</a>
{% endif %}

<br /><br />
{# lien pour se déconnecter #}
<a href="{{ path('app_logout') }}">
Se déconnecter</a>

</div>

{% endblock %}

```

Tester à présent avec un compte ayant le rôle `ROLE_COORDO`, puis avec un compte ayant le rôle `ROLE_SECRET`.

### i) accès aux données personnalisées de l'utilisateur

Dans l'entité `User` et ensuite dans la table correspondante *utilisateurs*, nous avons mis des attributs personnalisés, notamment le nom complet et le prénom complet de l'utilisateur.

Ces données sont ensuite affichables.

L'écran va par exemple ressembler à ceci :

Bienvenue à Jim NASTIC

## Liste des employés

Code	Nom	Prénom	Année d'embauche	Action 1	Action 2
109W827	HATAN	Charles	1992	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
110A225	AYMAR	Jean	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284B128	COVER	Harry	1994	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C214	AUSSINNE	Emma	2002	<a href="#">Modifier</a>	<a href="#">Supprimer</a>
284C226	ZETOFRAIS	Mélanie	1995	<a href="#">Modifier</a>	<a href="#">Supprimer</a>

[Ajouter un employé](#)

[Se déconnecter](#)

L'accès aux données personnalisées peut se faire soit dans un contrôleur, soit dans un template Twig.

Si cet accès pour ici l'affichage ne concerne que les templates, voyons à titre d'information comment l'implémenter dans un contrôleur.

### i-1) accès aux données personnalisées via un contrôleur : la méthode `getUser()`

La méthode `getUser()` retourne l'utilisateur courant.

Il suffit alors d'invoquer les getters de l'entité.

Au lieu de proposer au départ le tableau des employés après l'authentification d'un utilisateur, on va juste afficher son compte, son nom et son prénom complet.

La méthode correspondante `index()` du contrôleur `EmployeController` devient :

```
/**
 * @Route("/gestion-employes", name="gestion_employes")
 */
public function index()
{
    // récupération de l'utilisateur courant
    $utilisateur = $this->getUser();

    // récupération des informations via les getters
    $message = "Mon compte est : " .
        $utilisateur->getUserIdentifier();
    $message .= "<br />Mon nom est : " .
        $utilisateur->getNomcomplet();
    $message .= "<br />Mon prénom est : " .
        $utilisateur->getPrenomcomplet();

    return new Response($message);
}
```

On reviendra à la méthode `index()` d'origine.

### i-2) accès aux données personnalisées via un template : l'objet `app.user`

Réalisons le même affichage que précédemment mais cette fois depuis un template Twig.

On va utiliser l'objet `app.user` et ses différentes propriétés.

La propriété a le même nom que l'attribut, sachant qu'on passe par le getter correspondant.

Le code donne (à ne pas taper, se contenter de le comprendre) :

```
Mon compte est : {{ app.user.username }}
<br />Mon nom est : {{ app.user.nomComplet }}
<br />Mon prénom est : {{ app.user.prenomComplet }}
```

### i-3) utilisation pour notre cas d'exemple

Revenons à notre cas.

On souhaite afficher un message de bienvenue à l'utilisateur avant de lui présenter la liste des employés.

L'extrait de code du fichier **listeEmployes.html.twig** devient (la nouvelle ligne est en gras) :

```
<div class="container">

    <strong>Bienvenue à {{ app.user.prenomComplet }}
                        {{ app.user.nomComplet }}</strong>
    <br /> <br /> <br />

    <h1>Liste des employés</h1>
```

## j) exercice

En vous aidant des éléments de ce paragraphe 8, gérer l'authentification et les autorisations des utilisateurs de votre site personnel.

Il y aura deux types d'utilisateur :

- l'administrateur qui aura tous les droits sur les informations de votre table : dans la base de données le type sera alors égal à A et le rôle au sein de Symfony sera `ROLE_ADMIN`
- l'utilisateur standard qui n'aura que le droit de lecture sur les informations de votre table : dans la base de données le type sera alors égal à U et le rôle au sein de Symfony sera `ROLE_UTIL_STAND`

## k) traçabilité des accès

### k-1) rappel sur la gestion des utilisateurs et processus à mettre en œuvre

Au sein d'une application (solution applicative), la gestion des utilisateurs se base sur le sigle AAA : *Authentication Authorization Accounting*.

Pour l'instant dans le site d'exemple et dans votre site personnel via Symfony, on a mis en œuvre les deux premiers A.

Voyons à présent comment mettre en œuvre le 3ème A : *Accounting*.

L'objectif est de tracer notamment les différentes connexions d'un utilisateur.

Cette journalisation de chaque connexion, avec mémorisation de sa date et de son heure, se fera dans une table qu'on appellera par convention *logs*.

Voici un exemple d'enregistrement de la table *logs* :

id	nomUtilisateur	dateConnexion	heureConnexion
1	ptronc	2022-01-20	15:52:44

L'utilisateur **ptronc** a réussi à se connecter le **20/01/2022 à 15h52**.

### k-2) rappel sur le code actuel

Le framework Symfony gère l'authentification de l'utilisateur.

Lorsque l'utilisateur a réussi à se connecter, Symfony exécute automatiquement la méthode `index()` du contrôleur `LoginController` correspondant à la route `/gestion-employes`.

Voici le code correspondant :

```
/**
 * @Route("/gestion-employes", name="gestion_employes")
 */
public function index(ManagerRegistry $doctrine)
{
    // récupération du repository relatif à l'entité (classe) Employe
    $repository = $doctrine->getRepository(Employe::class);
    // recherche de tous les employés
```

```

$listeEmployes = $repository->findAll();
// affichage de la liste des employés
return $this->render('employe/listeEmployes.html.twig',
    ['employes' => $listeEmployes]);
}

```

Cette route `/gestion_employes` correspond à notre page d'accueil qui va notamment afficher la liste des employés.

Avant cet affichage de la liste des employés en début de méthode, on va journaliser cette connexion dans la table *logs*.

Le mieux est de créer une méthode interne privée qui va assurer cette journalisation.

Cela va donc donner :

```

public function index(ManagerRegistry $doctrine)
{
    // journalisation de la connexion
    $this->journalisationConnexion($doctrine);
    // METHODE A ECRIRE

    // récupération du repository relatif à l'entité (classe) Employe
    $repository = $doctrine->getRepository(Employe::class);
    // recherche de tous les employés
    $listeEmployes = $repository->findAll();
    // affichage de la liste des employés
    return $this->render('employe/listeEmployes.html.twig',
        ['employes' => $listeEmployes]);
}

```

### k-3) mise en place de la traçabilité : exercice

\* énoncé

***Le code sera le même pour le site d'exemple et le site personnel : on choisira pour l'implémentation du code un des deux sites.***

1-

Ecrire l'entité `Log` qui correspondra ultérieurement à votre table `logs` en mettant en œuvre les annotations adéquates, puis créer la table dans la fenêtre console.

Remarques :

La date sera de type texte et elle sera au format français par exemple 03/11/2022.

L'heure sera aussi de type texte et elle sera au format standard par exemple 19:32.

2-

Ecrire la méthode privée `journalisationConnexion()` dans votre contrôleur correspondant qui permet de tracer chaque accès utilisateur dans la table `logs` en important l'entité correspondante en préalable.

3-

Appeler cette nouvelle méthode depuis votre méthode `index()`.

4-

Tester votre code.



\* correction

1-

Le code de l'entité Log donne :

```

<?php

// src/Entity/Log.php

namespace App\Entity;

// définition d'un alias
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="logs")
 */
class Log
{

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

    /**
     * @ORM\Column(name="nomUtilisateur", type="string",
     *              length=180)
     */
    private $nomUtilisateur;

    /**
     * @ORM\Column(name="dateConnexion", type="string",
     *              length=10)
     */
    private $dateConnexion;

    /**
     * @ORM\Column(name="heureConnexion", type="string",
     *              length=8)
     */
    private $heureConnexion;

```

```

// getters et setters
public function getId()
{
    return $this->id;
}

public function getNomUtilisateur()
{
    return $this->nomUtilisateur;
}
public function setNomUtilisateur($nomUtilisateur)
{
    $this->nomUtilisateur = $nomUtilisateur;
}

public function getDateConnexion()
{
    return $this->dateConnexion;
}
public function setDateConnexion($dateConnexion)
{
    $this->dateConnexion = $dateConnexion;
}

public function getHeureConnexion()
{
    return $this->heureConnexion;
}
public function setHeureConnexion($heureConnexion)
{
    $this->heureConnexion = $heureConnexion;
}

}

```

On met ensuite à jour notre base de données, afin donc de créer la nouvelle table *logs* via les deux commandes correspondantes :

```

php bin/console make:migration
php bin/console doctrine:migrations:migrate

```

2-

Il faut d'abord importer l'entité Log ainsi :

```
use App\Entity\Log;
```

Le code de la méthode donne :

```
private function journalisationConnexion
                                (ManagerRegistry $doctrine)
{

    // définition du fuseau horaire
    date_default_timezone_set('Europe/Paris');
    // récupération de la date courante
    $dateCourante = date('d/m/Y');
    // récupération de l'heure courante
    $heureCourante = date('H:i:s');

    // instantiation de l'entité Log et injection des données
    $unLog = new Log();
    $unLog->setNomUtilisateur($this->getUser()->getUserIdentifier());
    $unLog->setDateConnexion($dateCourante);
    $unLog->setHeureConnexion($heureCourante);

    // écriture dans la base de données
    $em = $doctrine->getManager();
    $em->persist($unLog);
    $em->flush();

}
```

Remarque :

Pour trouver la date courante et l'heure courante, il faut d'abord définir le fuseau horaire, puis utiliser la fonction PHP `date()` en spécifiant à chaque fois le bon format.

## 9) Formulaire avec différents types de champs (widgets)

Voyons de nouveau les données que nous devons gérer à travers notre cas d'exemple.

### a) rappel du domaine pratique à implémenter et modification de la couche Modèle

#### a-1) cahier des charges

Nous souhaitons gérer les données simplifiées d'une organisation responsable de missions à caractère humanitaire à travers le monde.

Une mission est effectuée par plusieurs personnes pouvant être des ingénieurs, des médecins, des infirmiers, etc.

Elle se déroule parfois dans plusieurs pays.

Chaque mission est encadrée sur place notamment au niveau logistique par un employé de l'organisation.

*Notre domaine de gestion se limite à cet encadrement de missions par des employés.*

Une mission comprend :

- un genre prédéfini (mission médicale, mission de prospection, mission technique),
- un nom,
- une durée appartenant à une certaine tranche : 1 semaine, entre 1 et 3 semaines, 1 mois (qui est la durée la plus fréquente) ou plusieurs mois,

Un numéro séquentiel sera attribué à chaque mission.

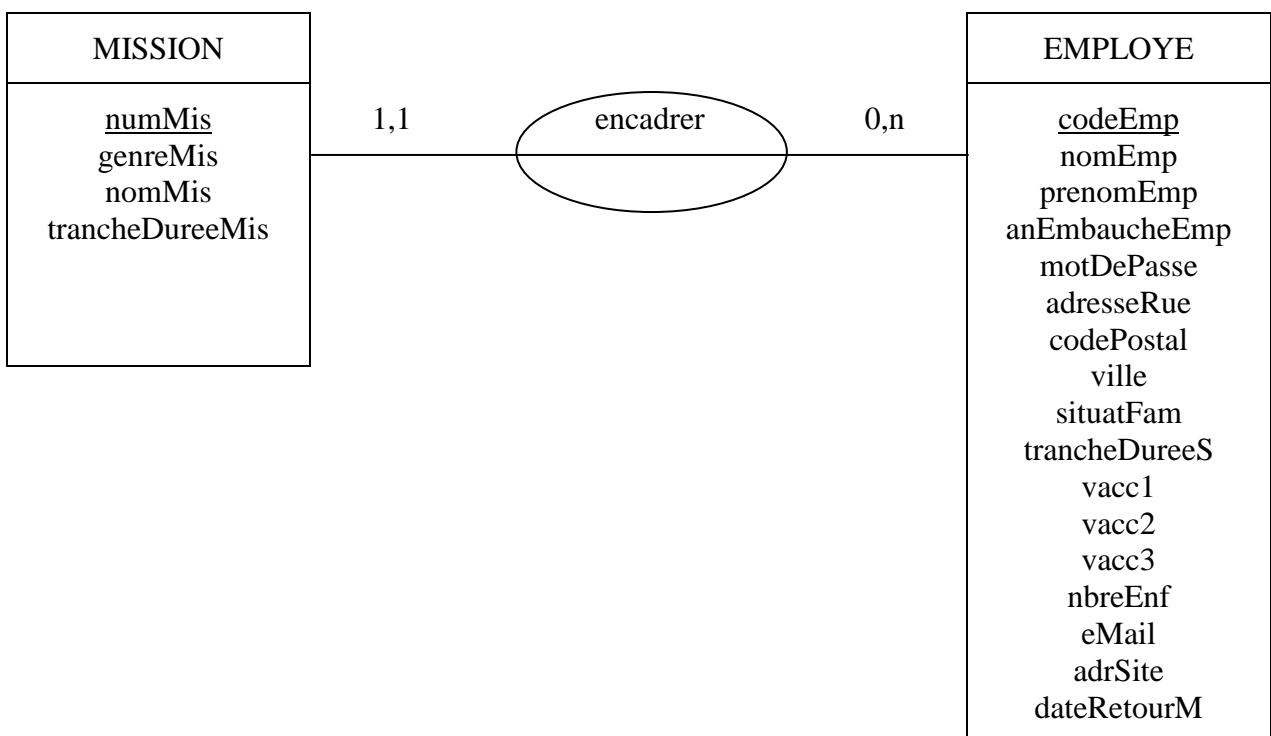
Pour l'employé, plus d'informations seront à mémoriser.

Un employé est caractérisé par :

- un code (qui a une nomenclature propre à l'organisation, il comprend 3 chiffres, puis 1 lettre majuscule, puis 3 chiffres, par exemple 109W827),
- un nom,
- un prénom,
- une année d'embauche,
- un mot de passe,
- une adresse avec code postal et ville,
- une situation familiale,

- une durée souhaitée pour la prochaine mission à encadrer : 1 semaine, entre 1 et 3 semaines, 1 mois (qui est la durée la plus fréquente) ou plusieurs mois,
- les vaccinations effectuées: il y en a de 3 types,
- un nombre d'enfants,
- un email,
- une adresse du site le plus visité, parmi les sites géographiques pour repérer les différents endroits où les missions vont se dérouler,
- une date de retour de la dernière mission encadrée.

## a-2) modèle conceptuel des données



Pas de problème particulier.

Une mission n'est encadrée que par un seul employé.

Un employé peut encadrer plusieurs missions.

La propriété genreMis sera sur un seul caractère : M pour Médicale, P pour Prospection, T pour Technique.

Pareil pour la propriété situatFam : C pour Célibataire, M pour Marié (e), D pour Divorcé (e), V pour Veuf (ve), A pour Autre.

La propriété `trancheDureeMis` sera définie comme un entier avec 4 valeurs possibles (de 1 à 4) pour chacune des tranches de durée possibles de la mission.

Pareil pour la propriété `trancheDureeS`.

Les trois propriétés `vacc1`, `vacc2` et `vacc3` sont de type booléen.

### a-3) modèle relationnel

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

Nous nous intéressons toujours ici pour l'instant à la table *employees* en tenant compte cette fois de tous ses champs qui vont donner lieu pour certains à des composants d'interface graphique (formulaire) autres que des zones de texte simples.

### a-4) les différents types Doctrine et MySQL

Au sein d'une entité, nous avons vu l'annotation `\Column` où on définit notamment le type de la colonne via le paramètre `type`.

Le tableau suivant fournit pour chaque type principal Doctrine le type MySQL qui sera généré.

Type Doctrine	Type MySQL
string	VARCHAR
smallint	SMALLINT
integer	INT
decimal	DECIMAL
boolean	BOOLEAN
date	DATE
time	TIME
datetime	DATETIME

A présent, on va modifier la couche *Modèle* pour dans un premier temps mettre à jour la nouvelle classe entité `Employe`, qui donnera ensuite lieu à une modification en base de données.

#### a-5) entité `Employe` (couche *Modèle*)

Voici l'entité `Employe` constituée à partir du cahier des charges :

```
<?php

// src/Entity/Employe.php

namespace App\Entity;

// définition d'un alias
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="employes")
 */
class Employe
{

    // attributs privés
    /**
     * @ORM\Column(name="codeEmp", type="string", length=7)
     * @ORM\Id
     */
    private $code;

    /**
     * @ORM\Column(name="nomEmp", type="string", length=25)
     */
    private $nom;

    /**
     * @ORM\Column(name="prenomEmp", type="string", length=20,
     *              nullable=true)
     */
    private $prenom;

    /**
```

```

    * @ORM\Column(name="anEmbaucheEmp", type="integer",
                  nullable=true)
    */
private $anEmbauche;

/**
    * @ORM\Column(name="motDePasse", type="string", length=20,
                  nullable=true)
    */
private $motDePasse;

/**
    * @ORM\Column(name="adresseRue", type="string", length=80,
                  nullable=true)
    */
private $adresseRue;

/**
    * @ORM\Column(name="codePostal", type="string", length=5,
                  nullable=true)
    */
private $codePostal;

/**
    * @ORM\Column(name="ville", type="string", length=20,
                  nullable=true)
    */
private $ville;

/**
    * @ORM\Column(name="situatFam", type="string", length=1,
                  nullable=true)
    */
private $situatFam;

/**
    * @ORM\Column(name="trancheDurees", type="smallint",
                  nullable=true)
    */
private $trancheDurees;

/**
    * @ORM\Column(name="vaccl", type="boolean",
                  nullable=true)
    */
private $vaccl;

/**
    * @ORM\Column(name="vaccl2", type="boolean",
                  nullable=true)
    */

```



```

private $vacc2;

/**
 * @ORM\Column(name="vacc3", type="boolean",
               nullable=true)
 */
private $vacc3;

/**
 * @ORM\Column(name="nbreEnf", type="smallint",
               nullable=true)
 */
private $nbreEnf;

/**
 * @ORM\Column(name="eMail", type="string", length=30,
               nullable=true)
 */
private $eMail;

/**
 * @ORM\Column(name="adrSite", type="string", length=30,
               nullable=true)
 */
private $adrSite;

/**
 * @ORM\Column(name="dateRetourM", type="date",
               nullable=true)
 */
private $dateRetourM;

// getters et setters
public function getCode()
{
    return $this->code;
}
public function setCode($code)
{
    $this->code = $code;
}

public function getNom()
{
    return $this->nom;
}
public function setNom($nom)
{
    $this->nom = $nom;
}

```

```

public function getPrenom()
{
    return $this->prenom;
}
public function setPrenom($prenom)
{
    $this->prenom = $prenom;
}

public function getAnEmbauche()
{
    return $this->anEmbauche;
}
public function setAnEmbauche($anEmbauche)
{
    $this->anEmbauche = $anEmbauche;
}

public function getMotDePasse()
{
    return $this->motDePasse;
}
public function setMotDePasse($motDePasse)
{
    $this->motDePasse = $motDePasse;
}

public function getAdresseRue()
{
    return $this->adresseRue;
}
public function setAdresseRue($adresseRue)
{
    $this->adresseRue = $adresseRue;
}

public function getCodePostal()
{
    return $this->codePostal;
}
public function setCodePostal($codePostal)
{
    $this->codePostal = $codePostal;
}

public function getVille()
{
    return $this->ville;
}
public function setVille($ville)

```

```

{
    $this->ville = $ville;
}

public function getSituatFam()
{
    return $this->situatFam;
}

public function setSituatFam($situatFam)
{
    $this->situatFam = $situatFam;
}

public function getTrancheDurees()
{
    return $this->trancheDurees;
}
public function setTrancheDurees($trancheDurees)
{
    $this->trancheDurees = $trancheDurees;
}

public function getVacc1()
{
    return $this->vacc1;
}
public function setVacc1($vacc1)
{
    $this->vacc1 = $vacc1;
}

public function getVacc2()
{
    return $this->vacc2;
}
public function setVacc2($vacc2)
{
    $this->vacc2 = $vacc2;
}

public function getVacc3()
{
    return $this->vacc3;
}
public function setVacc3($vacc3)
{
    $this->vacc3 = $vacc3;
}

public function getNbreEnf()

```

```

{
    return $this->nbreEnf;
}
public function setNbreEnf($nbreEnf)
{
    $this->nbreEnf = $nbreEnf;
}

public function getEmail()
{
    return $this->eMail;
}
public function setEmail($eMail)
{
    $this->eMail = $eMail;
}

public function getAdrSite()
{
    return $this->adrSite;
}
public function setAdrSite($adrSite)
{
    $this->adrSite = $adrSite;
}

public function getDateRetourM()
{
    return $this->dateRetourM;
}
public function setDateRetourM($dateRetourM)
{
    $this->dateRetourM = $dateRetourM;
}

}

```

Ensuite, on met comme d'habitude à jour la base de données.

Les deux commandes correspondantes sont :

**php bin/console make:migration**

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

b) formulaire complet d'exemple

Voici le formulaire pour la saisie de toutes les informations d'un employé :

## Ajout d'un employé

Code

Nom

Prénom

Année d'embauche

Mot de passe

Confirmation

Adresse

Code postal

Ville

Situation familiale

Durée souhaitée prochaine mission  
☐ 1 semaine  
☐ entre 1 et 3 semaines  
☒ 1 mois  
☐ plusieurs mois

☐ vaccin type 1  
☐ vaccin type 2  
☐ vaccin type 3

Nombre d'enfants

Email

Adresse du site le plus visité

Date de retour de mission  
Jour  Mois  Année

On retrouve différents types de widget en plus de la zone de texte simple, comme des cases à cocher, une liste déroulante ou une zone de sélection de date.

### c) rappel de l'organisation

La mise en œuvre d'un formulaire sous Symfony est étroitement liée à une entité de la couche *Modèle* : c'est à travers les différents champs de formulaire que seront alimentés (on parle d'injecter ou d'hydrater) ses attributs.

Ici, le formulaire est celui d'ajout d'un employé : il va donc être basé sur l'entité `Employe` écrite au paragraphe a-5.

Ce formulaire sera construit via un contrôleur (couche *Contrôleur*) puis passé à un template Twig (couche *Vue*) pour son affichage.

## d) le contrôleur générateur du formulaire (couche *Contrôleur*) : widgets Symfony

### d-1) le contrôleur générateur du formulaire

Voyons à nouveau le contrôleur qui génère le formulaire d'ajout d'un employé et qui traite la requête de soumission (avec donc ajout de l'employé en base de données via Doctrine).

Le code donne (fichier **src/Controller/EmployeController.php**) :

```
/**
 * @Route("/ajout-employe", name="ajout_employe")
 */
public function ajout(ManagerRegistry $doctrine,
                    Request $request)
{

    // instantiation de l'entité Employe
    $employe = new Employe();

    // création du constructeur de formulaire en fournissant l'entité
    $formBuilder = $this->createFormBuilder($employe);

    /* ajout successif des propriétés souhaitées de l'entité
       pour les champs de formulaire avec leur type */
    $formBuilder->add('code', TextType::class)
        ->add('nom', TextType::class)
        ->add('prenom', TextType::class)
        ->add('anEmbauche', TextType::class)
        ->add('validation', SubmitType::class,
            ['label' => 'Valider la saisie'])
        ->add('effacement', ResetType::class,
            ['label' => 'Effacer la saisie']);

    // récupération du formulaire à partir du constructeur de formulaire
    $form = $formBuilder->getForm();

    /* traitement de la requête : Symfony récupère éventuellement les valeurs des
       champs de formulaire et alimente l'objet $employe */
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid())
        // le formulaire a été soumis et il est valide
        {
            // écriture dans la base de données
            $em = $doctrine->getManager();
            $em->persist($employe);
            $em->flush();
        }
    }
```



```

    // récupération du repository relatif à l'entité (classe) Employe
    $repository = $doctrine->getRepository(Employe::class);
    // recherche de tous les employés
    $listeEmployes = $repository->findAll();
    // affichage de la liste des employés
    return $this->render('employe/listeEmployes.html.twig',
                        ['employees' => $listeEmployes]);
}

// passage du formulaire au template pour affichage avec l'opération réalisée
return $this->render('employe/formEmploye.html.twig',
                    ['form' => $form->createView(),
                     'operation' => 'Ajout']);
}

```

A présent, modifions le code pour prendre en compte tous ces nouveaux attributs relatifs à un employé.

***Une seule instruction est à changer.***

Il s'agit de l'instruction d'ajout successif de tous les widgets qui est :

```

$formBuilder
->add('code', TextType::class)
->add('nom', TextType::class)
->add('prenom', TextType::class)
->add('anEmbauche', TextType::class)
->add('validation', SubmitType::class,
     ['label' => 'Valider la saisie'])
->add('effacement', ResetType::class,
     ['label' => 'Effacer la saisie']);

```

Donnons la nouvelle instruction.

Ne vous affolez pas par le nombre d'éléments présents, nous reprendrons tout en détail ensuite.

Voici la (grosse) instruction de remplacement :

```

$formBuilder
->add('code', TextType::class,
    ['attr' => ['size' => 7,
                'maxlength' => 7,
                'pattern' => '[0-9][0-9][0-9][A-Z]
                            [0-9][0-9][0-9]']]
->add('nom', TextType::class)
->add('prenom', TextType::class,
    ['label' => 'Prénom',
     'required' => false])
->add('anEmbauche', TextType::class,
    ['label' => 'Année d\'embauche',
     'required' => false,
     'attr' => ['maxlength' => 4,
                'pattern' => '[0-9][0-9][0-9][0-9]']]
->add('motDePasse', RepeatedType::class,
    ['type' => PasswordType::class,
     'required' => false,
     'first_options' => ['label' => 'Mot de passe'],
     'second_options' => ['label' => 'Confirmation'],
     'invalid_message' => 'Non concordance du mot de
                           passe de confirmation'])
->add('adresseRue', TextareaType::class,
    ['label' => 'Adresse',
     'required' => false])
->add('codePostal', TextType::class,
    ['label' => 'Code postal',
     'required' => false,
     'attr' => ['size' => 5]])
->add('ville', TextareaType::class,
    ['label' => 'Ville',
     'required' => false])
->add('situatFam', ChoiceType::class,
    ['label' => 'Situation familiale',
     'required' => false,
     'placeholder' => 'Choisir une valeur',
     'choices' => ['Célibataire' => 'C',
                   'Marié (e)' => 'M',
                   'Divorcé (e)' => 'D',
                   'Veuf (ve)' => 'V',
                   'Autre' => 'A']])
->add('trancheDurees', ChoiceType::class,
    ['expanded' => true,
     'label' => 'Durée souhaitée prochaine mission',
     'choices' => ['1 semaine' => 1,
                   'entre 1 et 3 semaines' => 2,
                   '1 mois' => 3,
                   'plusieurs mois' => 4],
     'data' => 3])
->add('vaccl', CheckboxType::class,
    ['label' => 'Vaccin type 1',

```

```

        'required' => false])
->add('vacc2', CheckboxType::class,
    ['label' => 'Vaccin type 2',
     'required' => false])
->add('vacc3', CheckboxType::class,
    ['label' => 'Vaccin type 3',
     'required' => false])
->add('nbreEnf', IntegerType::class,
    ['label' => 'Nombre d\'enfants',
     'required' => false,
     'data' => 1])
->add('eMail', EmailType::class,
    ['label' => 'Email',
     'required' => false])
->add('adrSite', UrlType::class,
    ['label' => 'Adresse du site le plus visité',
     'required' => false])
->add('dateRetourM', DateType::class,
    ['label' => 'Date de retour de mission',
     'required' => false,
     'years' => range(2019, 2026),
     'format' => 'dd-MMM-yyyy',
     'placeholder' => ['year' => 'Année',
                       'month' => 'Mois',
                       'day' => 'Jour']])
->add('validation', SubmitType::class,
    ['label' => 'Valider la saisie'])
->add('effacement', ResetType::class,
    ['label' => 'Effacer la saisie']);

```

Les types comme par exemple `TextType` doivent être importés.

Cela donne :

```

use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ResetType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\Extension\Core\Type>PasswordType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\UrlType;
use Symfony\Component\Form\Extension\Core\Type\DateType;

```

Voyons de plus près la syntaxe de cette méthode `add()`.

## d-2) syntaxe de la méthode add()

Tout widget (ou contrôle) est donc rajouté en invoquant la méthode `add()` sur le constructeur de formulaire.

La syntaxe générale est :

```
add(nomWidget, classeType,
    ['option1' => valeur1,
     'option2' => valeur2,
     'option3' => valeur3,
     etc.])
```

***Le nom du widget est celui de la propriété (qui s'appelle comme l'attribut) de l'entité correspondante.***

Chacune des options est donc fournie via des crochets avec la syntaxe particulière de PHP.

Par exemple, l'implémentation du nombre d'enfants donne :

```
->add('nbreEnf', IntegerType::class,
    ['label' => 'Nombre d\'enfants',
     'required' => false,
     'data' => 1])
```

Ici, le widget correspond à l'attribut `nbreEnf` de l'entité correspondante (ici `Employe`).

Il est de type zone de sélection numérique d'où la valeur `IntegerType::class`.

On définit pour cette zone de texte les options `label`, `required` et `data` (correspondant respectivement à son étiquette, l'indication que la valeur n'est pas requise et sa valeur par défaut).

### Rappel :

Les crochets sont en fait un raccourci de l'élément `array` (des crochets font bien penser à un tableau...).

Ainsi l'invocation précédente peut s'écrire :

```
->add('nbreEnf', IntegerType::class,
      array('label' => 'Nombre d\'enfants',
            'required' => false,
            'data' => 1))
```

Parfois, on retrouve dans les crochets (élément `array`) d'autres crochets (autre élément `array`) quand l'option correspondante a besoin de plusieurs valeurs.

Considérons l'implémentation de la situation familiale.

Cela donne :

```
->add('situatFam', ChoiceType::class,
      ['label' => 'Situation familiale',
       'required' => false,
       'placeholder' => 'Choisir une valeur',
       'choices' => ['Célibataire' => 'C',
                     'Marié (e)' => 'M',
                     'Divorcé (e)' => 'D',
                     'Veuf (ve)' => 'V',
                     'Autre' => 'A']])
```

Pour l'option `choices` définissant les choix, plusieurs couples de valeurs sont à donner (il y a la valeur affichée par exemple `Célibataire` et la valeur qui sera stockée dans l'attribut par exemple `C`) : on les fournit à l'intérieur de crochets (élément `array`).

Voyons à présent les principaux types de widget proposés, puis les options essentielles.

Nous nous baserons sur ce formulaire d'exemple.

### d-3) types de widget

Le tableau suivant fournit les principaux types invocables avec la méthode `add()`, et leur description.

<i>Type de widget</i>	<i>Description</i>
<code>TextType</code>	Zone de texte sur une ligne
<code>TextareaType</code>	Zone de texte sur plusieurs lignes
<code>PasswordType</code>	Zone de texte de type mot de passe
<code>RepeatedType</code>	Duplication de deux zones de texte qui doivent concorder (pour mots de passe)
<code>EmailType</code>	Zone de texte d'adresse email
<code>UrlType</code>	Zone de texte d'adresse URL
<code>SearchType</code>	Zone de texte de recherche
<code>IntegerType</code>	Zone de sélection numérique entier
<code>NumberType</code>	Zone de texte avec gestion de nombres
<code>MoneyType</code>	Zone de texte avec gestion de données monétaires
<code>PercentType</code>	Zone de texte avec gestion de pourcentages
<code>ChoiceType</code>	Liste déroulante d'éléments, ensemble de boutons radio ou de cases à cocher
<code>RadioType</code>	Bouton radio : on utilise plutôt le type <code>choice</code>
<code>CheckboxType</code>	Case à cocher
<code>DateType</code>	Zone de sélection de l'année, du mois et du jour d'une date
<code>TimeType</code>	Zone de sélection de l'heure, des minutes et des secondes d'une heure
<code>DateTimeType</code>	Zone de sélection à la fois des informations d'une date et d'une heure
<code>BirthdayType</code>	Zone de sélection pour la gestion des dates de naissance
<code>CountryType</code>	Liste déroulante des pays du monde
<code>LanguageType</code>	Liste déroulante de langues du monde
<code>LocaleType</code>	Liste déroulante de couples langue/pays du monde
<code>CurrencyType</code>	Liste déroulante de devises

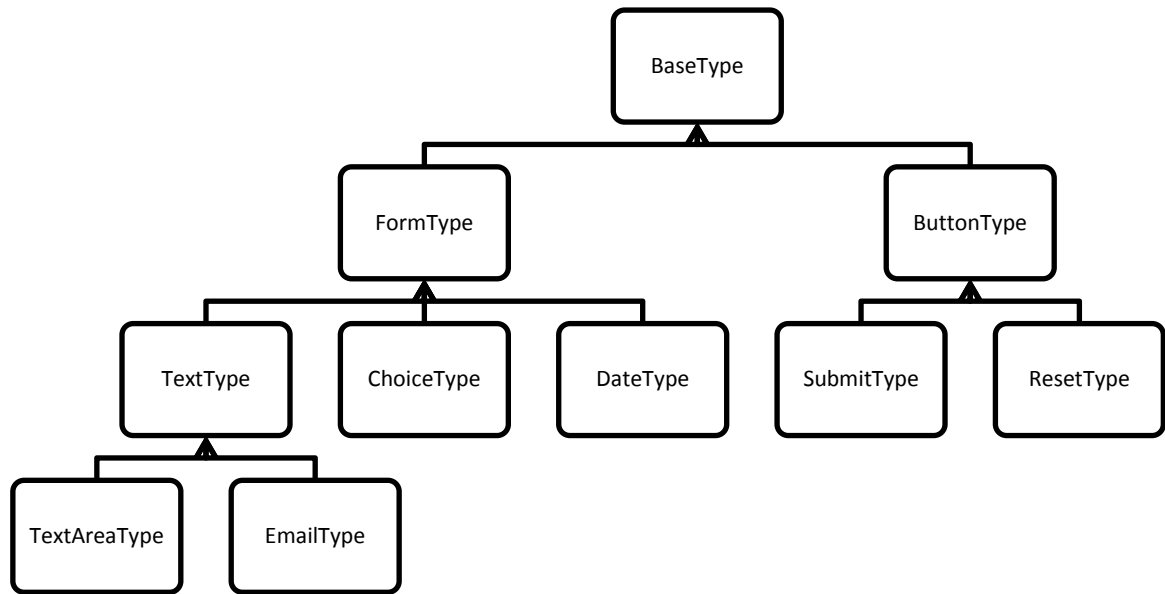
<i>Type de widget</i>	<i>Description</i>
TimezoneType	Liste déroulante de fuseaux horaires
FileType	Zone de sélection d'un fichier
EntityType	Liste déroulante de données relative à une entité
HiddenType	Champ caché
SubmitType	Bouton de soumission
ResetType	Bouton d'annulation
ButtonType	Bouton de commande

#### d-4) type de base BaseType et héritage des types

Chaque widget (contrôle) comprend un certain nombre d'options qui sont mises en place via les crochets (élément `array`).

Certaines options comme l'option `required` se retrouvent pour plusieurs widgets et leur classe correspondante, par exemple `TextType`, `ChoiceType` ou `DateType` mais aussi pour un formulaire dont la classe correspondante est `FormType`.

Du coup, Symfony met en place un héritage relativement aux différents types dont nous donnons ici un extrait simplifié.



Pour notre exemple, l'option `required` se définit au niveau de la classe `FormType` dont héritent `TextType`, `ChoiceType` et `DateType`.

En haut de l'arbre d'héritage, on trouve la classe `BaseType` qui va implémenter toutes les options dont tous les widgets ont besoin.

A présent, reprenons donc les options de chaque type dont nous avons ou aurons besoin pour nos formulaires relatifs aux employés en commençant logiquement par celles du type de base `BaseType`.

On peut retrouver à l'URL suivante les différents types :

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



## d-5) options de widget

\* type BaseType

Le tableau suivant fournit les principales options avec leur description et un exemple éventuel de code.

<i>Option</i>	<i>Description</i>	<i>Exemple éventuel</i>
label	Etiquette à gauche du widget	'label ' => 'Année d\'embauche '
attr	Ajout d'un attribut HTML	Voir ci-après
disabled	Valeur en lecture seule (false par défaut)	'disabled' => true

### Remarque :

Si on ne met pas d'option `label`, Symfony reprend le nom du widget avec une majuscule au départ.

Parlons à présent de l'option `attr` qui est particulière.

Cette option `attr` permet d'ajouter un attribut HTML qui n'est pas repris par une option Symfony, ce qui est souvent bien pratique !

Par exemple, prenons le cas de l'attribut HTML `size` qui spécifie la taille en largeur du widget à l'écran en nombre de caractères : il n'y a pas d'option Symfony équivalente.

De même pour l'attribut HTML `maxlength` qui spécifie le nombre de caractères maximal saisis de la zone.

Et pareil pour l'attribut HTML `pattern` qui spécifie une expression régulière pour vérifier la validité de la saisie.

Donc si par exemple on veut mettre pour le code de l'employé le contrôle sur 7 caractères avec blocage de la saisie au delà, qui comprennent 3 chiffres, 1 lettre et 3 chiffres, l'extrait de code donne :

```
->add('code', TextType::class,
    ['attr' => ['size' => 7,
                'maxlength' => 7,
                'pattern' => '[0-9][0-9][0-9][A-Z]
                             [0-9][0-9][0-9]']])
```

\* type FormType

Le tableau suivant fournit les principales options avec leur description et un exemple éventuel de code déjà implémenté.

<i>Option</i>	<i>Description</i>	<i>Exemple éventuel</i>
required	Valeur requise (true par défaut)	'required' => false
invalid_message	Message d'erreur de validation	'invalid_message' => 'Non concordance du mot de passe de confirmation'
data	Valeur par défaut	'data' => 1
trim	Suppression des espaces éventuels avant et après (true par défaut)	
mapped	Association avec un attribut d'entité (true par défaut)	

Voyons à présent certaines spécificités de widgets.

\* type RepeatedType

Le tableau suivant fournit les principales options avec leur description et un exemple de code déjà implémenté.

<i>Option</i>	<i>Description</i>	<i>Exemple</i>
type	Type des deux zones de texte (TextType par défaut)	'type' => 'password'
first_options	Options pour la première zone	'first_options' => ['label' => 'Mot de passe']
second_options	Options pour la seconde zone	'second_options' => ['label' => 'Confirmation']

### \* type ChoiceType

Voyons tout d'abord à part les options `expanded` et `multiple` qui fonctionnent ensemble et qui permettent de définir le type du champ.

Le tableau suivant fournit pour chaque type de champ souhaité le positionnement à effectuer pour ces deux options.

<i>Type de champ</i>	<i>expanded</i>	<i>multiple</i>
Liste déroulante	false	false
Liste déroulante à sélection multiple	false	true
Boutons radios	true	false
Cases à cocher	true	true

Par défaut, `expanded` et `multiple` sont à `false` : liste déroulante simple.

De manière pratique, pour les boutons radio on utilise ce type `choice` et pour les cases à cocher on utilise plutôt le type `checkbox`.

Le tableau suivant fournit les principales options avec leur description et un exemple éventuel de code déjà implémenté.

<i>Option</i>	<i>Description</i>	<i>Exemple éventuel</i>
<code>expanded</code>	Voir ci-dessus	
<code>multiple</code>	Voir ci-dessus	
<code>choices</code>	Éléments de la liste : valeur + clé. La valeur est affichée dans le navigateur et la clé est envoyée par la requête (attribut de l'entité) .	<code>'choices' =&gt; ['Célibataire' =&gt; 'C', 'Marié (e)' =&gt; 'M', 'Divorcé (e)' =&gt; 'D', 'Veuf (ve)' =&gt; 'V', 'Autre' =&gt; 'A']])</code>
<code>placeholder</code>	Étiquette s'affichant en début de liste	<code>'placeholder' =&gt; 'Choisir une valeur'</code>

### \* type DateType

Le tableau suivant fournit les principales options avec leur description et un exemple de code déjà implémenté.

<i>Option</i>	<i>Description</i>	<i>Exemple</i>
years	Plage d'années à afficher (par défaut les 5 précédant et les 5 suivant l'année en cours). On utilise la fonction range.	'years' => range(2019, 2026)
format	Format de la date (par défaut yyyy-MM-dd)	'format' => 'dd-MMM-YYYY'
placeholder	Etiquette s'affichant en début de chacune des 3 listes (année, mois, jour)	'placeholder' => ['year' => 'Année', 'month' => 'Mois', 'day' => 'Jour']

Remarque :

L'option `placeholder` est utilisée pour une liste déroulante et pour une zone de sélection de date.

Pour la liste déroulante, il y a une seule valeur à fournir.

Cela donne :

```
'placeholder' => 'Choisir une valeur'
```

Par contre, pour la zone de sélection d'une date, il faut fournir 3 valeurs pour respectivement la sélection d'une année (option `year`), d'un mois (option `month`) et d'un jour (option `day`) : on utilise donc un élément `array` via les crochets (raccourci).

Cela donne :

```
'placeholder' => ['year' => 'Année',
                  'month' => 'Mois',
                  'day' => 'Jour']
```

### e) le template Twig (couche Vue)

Le template Twig invoqué par le contrôleur n'est pas modifié sauf pour l'affichage du formulaire et de ses différents éléments où on remettra l'instruction Twig de base sur les formulaires `{{ form }}`.

Son code donne :

```
{# templates/employe/formEmploye.html.twig #}

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

{% block title %}
    {% if operation == "Ajout" %}
        Ajout d'un employé
    {% else %}
        {% if operation == "Modif" %}
            Modification d'un employé
        {% else %}
            Suppression d'un employé
        {% endif %}
    {% endif %}
{% endblock %}

{% block body %}

    <div class="container">

        {% if operation == "Ajout" %}
            <h1>Ajout d'un employé</h1>
        {% else %}
            {% if operation == "Modif" %}
                <h1>Modification d'un employé</h1>
            {% else %}
                <h1>Suppression d'un employé</h1>
            {% endif %}
        {% endif %}
        <br />

        <div class="row">

            {{ form(form) }}

        </div>
        <br />
        {# lien pour revenir à la liste des employés #}
        <a href="{{ path('gestion_employes') }}">
            Annuler : Retour à la liste des employés</a>
```

```

</div>

{% endblock %}

```

## f) formulaire de modification et de suppression : même principe

Pour générer le formulaire de modification, on va utiliser la même grosse instruction.

Quelques changements très minimes : il faut mettre en lecture seule le code employé, changer l'intitulé du bouton `submit`, supprimer le bouton `reset` et enlever les valeurs par défaut (plus d'option `data`).

L'extrait de code donne (les modifications sont en gras) :

```

$formBuilder
->add('code', TextType::class,
    ['disabled' => true,    // mise en lecture seule
     'attr' => ['size' => 7]])
.....
->add('validation', SubmitType::class,
    ['label' => 'Valider les modifications']);

```

Pour générer le formulaire de suppression, on reprend le formulaire de modification en mettant toutes les zones en lecture seule et en changeant l'intitulé du bouton `submit`.

## g) la validation des données

Une fois le formulaire généré, voyons comment opérer la validation des données.

### g-1) la validation côté client

Lors de la récupération du formulaire, un processus de validation côté client peut être mise en place : il se fait au niveau du navigateur.

Cela peut se faire via le type du champ ou par une option du champ.

#### Exemple 1 :

On souhaite valider la saisie d'un email.

On utilise le type de widget `EmailType::class`.

Cela donne :

```
->add('email', EmailType::class,
      ['label' => 'Email',
       'required' => false])
```

#### Exemple 2 :

On souhaite valider la saisie du code employé qui doit comprendre 3 chiffres, 1 lettre majuscule et 3 chiffres.

On utilise l'attribut HTML `pattern`, donc via l'option `attr`.

Cela donne :

```
->add('code', TextType::class,
      ['attr' => ['size' => 7,
                  'maxlength' => 7,
                  'pattern' => '[0-9][0-9][0-9][A-Z]
                               [0-9][0-9][0-9]']])
```

Si cette validation côté client est utile car elle permet de réduire le trafic réseau et de soulager le serveur, elle ne peut pas suffire pour une validation sûre car on est tributaire du navigateur de l'utilisateur ! Son navigateur peut ne pas reconnaître certains paramètres.

Il va donc falloir agir aussi côté serveur.

## g-2) la validation côté serveur via les contraintes sur un exemple

Côté serveur, les règles de validation s'appellent des *contraintes*.

Il y a notamment deux manières de mettre en place ces contraintes :

- au niveau de la récupération du formulaire (couche *Contrôleur*),
- au niveau de l'entité (couche *Modèle*) associé au formulaire.

La deuxième manière est plus utilisée car :

- elle propose plus de choix,
- elle pourra être partageable par plusieurs contrôleurs notamment lors de la génération de formulaires.

Pour mettre en place les contraintes au niveau de l'entité donc, on utilisera les annotations : rappelons qu'on peut aussi réaliser l'implémentation en YAML, XML ou PHP.

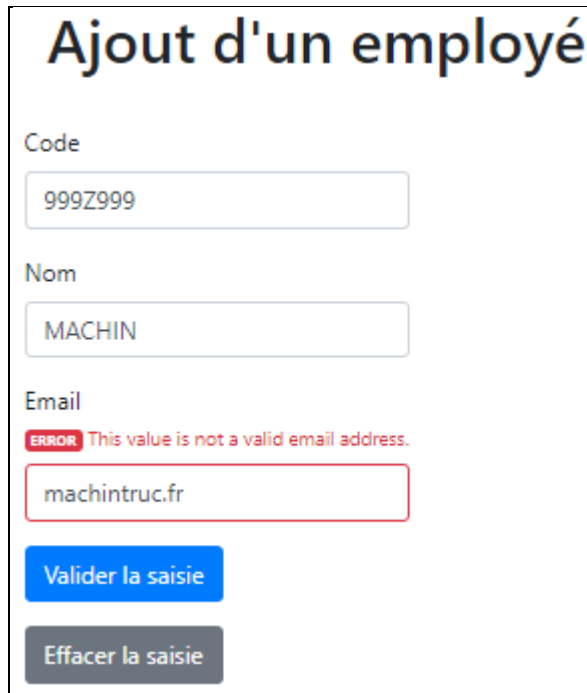
Reprenons l'exemple 1 du paragraphe g-1 qui vérifie la saisie d'un email.

On va devoir agir en 3 temps :

- mise en place de l'annotation dans l'entité (couche *Modèle*),
- lancement de la validation dans le contrôleur (couche *Contrôleur*),
- prise en compte du message d'erreur dans le formulaire (couche *Vue*).

Voici ce que donnera la validation côté serveur avec l'affichage du message d'erreur (on ne mettra dans le formulaire que les zones pour le code employé, son nom et son email) :





**Ajout d'un employé**

Code

Nom

Email  
ERROR This value is not a valid email address.

\* mise en place de l'annotation dans l'entité (couche *Modèle*)

Dans l'entité `Employe`, on va d'abord importer l'espace de noms `Validator\Constraints` et définir un alias qu'on appellera `Assert` ainsi :

```
use Symfony\Component\Validator\Constraints as Assert;
```

Ensuite, voici ce que va donner la nouvelle déclaration de l'email avec ses annotations correspondantes :

```
/**
 * @Assert\Email()
 * @ORM\Column(name="eMail", type="string", length=30,
 *             nullable=true)
 */
private $eMail;
```

Commentaire :

On utilise l'annotation de vérification `Email`.

Remarque :

On n'a pas besoin de mettre à jour la base de données car ces contraintes ne concernent que la saisie et non pas les données persistantes.

\* lancement de la validation dans le contrôleur (couche *Controller*)

Les modifications se feront lorsque le formulaire a été soumis, donc dans le bloc introduit par la condition :

```
if ($form->isSubmitted())
```

A noter qu'on ne met pas la condition en plus :

```
$form->isValid()
```

car la validation est cette fois faite par nous-mêmes.

Au niveau du contrôleur **src/Controller/EmployeController.php**, on va d'abord importer la classe-interface `ValidatorInterface` ainsi :

```
use Symfony\Component\Validator\Validator\ValidatorInterface;
```

Dans l'en tête de la méthode `ajout()`, on va mettre un paramètre, objet de cette classe `ValidatorInterface`.

Cela va donner :

```
public function ajout(Request $request,
                    ValidatorInterface $validator)
```

On va ensuite traiter la validation lorsque le formulaire est soumis.

Le code devient :

```
if ($form->isSubmitted())
{

    // déclenchement de la validation sur notre formulaire
    $listErrors = $validator->validate($form);

    // test s'il y a au moins une erreur
    if (count($listErrors) > 0)
    {
        // passage du formulaire à la vue pour affichage
        // sans donc écriture dans la base de données
        return $this->render('employe/formEmploye.html.twig',
                            ['form' => $form->createView(),
                             'operation' => 'Ajout']);
    }

    // écriture dans la base de données
    $em = $doctrine->getManager();
    ....
}
```

Commentaires :

On lance la méthode `validate()` sur notre formulaire.

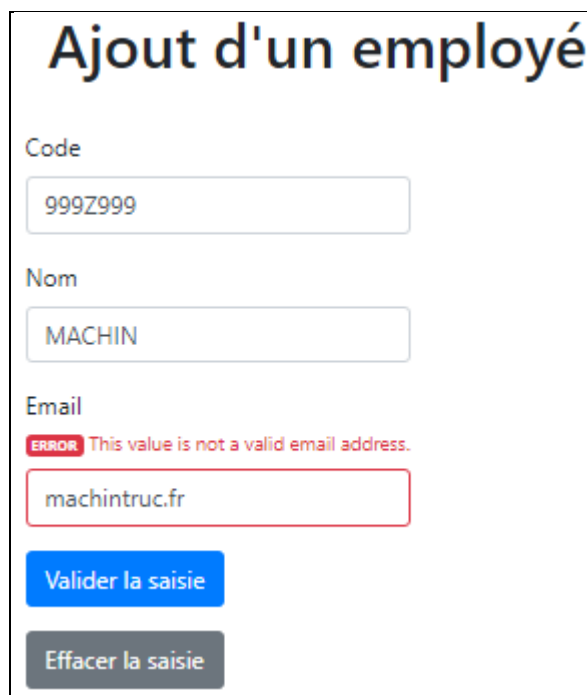
Si le nombre d'erreurs (fonction `count()`) est supérieur à 0, on passe le formulaire à la vue **formEmploye.html.twig** sans donc écrire dans la base de données.

Remarque :

Rien n'est à changer dans le formulaire Twig.

L'annotation `@Assert\Email()` permet d'afficher les erreurs relatives au champ `eMail`.

Pour tester cette validation côté serveur, on positionnera momentanément le type du widget à `TextType::class` au lieu de `EmailType::class` et on ne mettra dans le formulaire que les zones pour le code employé, son nom et son email.



**Ajout d'un employé**

Code  
999Z999

Nom  
MACHIN

Email  
ERROR This value is not a valid email address.  
machintruc.fr

Valider la saisie

Effacer la saisie

Si on veut mettre un message personnalisé ainsi :

## Ajout d'un employé

Code

Nom

Email

ERROR

 Adresse email invalide
 

Valider la saisie

Effacer la saisie

Il faut positionner dans l'annotation `Email` le paramètre (option) `message`.

Cela donne :

```
/**
 * @Assert\Email(message="Adresse email invalide")
 * @ORM\Column(name="eMail", type="string", length=30,
 *              nullable=true)
 */
```

Examinons à présent les principales contraintes.

### g-3) principales contraintes

Les contraintes vont être implémentées au sein de l'entité concernée (couche *Modèle*) via des annotations.

Examinons tout d'abord les différentes contraintes principales par type et la vérification effectuée sur la valeur soumise par la requête issue du formulaire.

Notons en préambule que toutes les contraintes sauf `Length` et `Range` possèdent l'option `message` permettant d'afficher un message d'erreur personnalisé.

#### \* contraintes de base

Le tableau suivant explicite chacune des principales contraintes de base avec les options éventuelles.

<i>Contrainte</i>	<i>Vérification</i>	<i>Options éventuelles</i>
<code>NotBlank</code>	Ni une chaîne de caractères vide, ni <code>NULL</code>	
<code>NotNull</code>	Pas <code>NULL</code>	
<code>Type</code>	Valeur du type défini	<code>type</code> (option par défaut) : égal à <code>bool</code> , <code>int</code> , <code>string</code> , etc.

#### \* contraintes de chaînes de caractères

Le tableau suivant explicite chacune des contraintes sur les chaînes de caractères avec les options éventuelles.

<i>Contrainte</i>	<i>Vérification</i>	<i>Options éventuelles</i>
Length	Longueur entre deux valeurs	exactMessage : message si la valeur n'est pas égale à min et max min : nombre minimum max : nombre maximum minMessage : message si la valeur est inférieure à min maxMessage : message si la valeur est supérieure à max
Regex	Par rapport à une expression régulière	pattern : expression régulière (option par défaut) match : valeur doit correspondre (true par défaut) ou non (false)
Email	Adresse email valide	
Url	Adresse URL valide	
Ip	Adresse IP valide	version : 4 pour IPv4 (par défaut) ou 6 pour Ipv6
Country	Code pays sur 2 lettres valide	
Language	Langue du monde valide	
Locale	Couple langue/pays valide	

\* contrainte sur les nombres

Le tableau suivant explicite la contrainte caractères avec les options éventuelles.

<i>Contrainte</i>	<i>Vérification</i>	<i>Options éventuelles</i>
Range	Valeur entre deux nombres	min : nombre minimum max : nombre maximum minMessage : message si la valeur est inférieure à min maxMessage : message si la valeur est supérieure à max invalidMessage : message si la valeur n'est pas un nombre

\* contraintes sur les informations temporelles

Le tableau suivant explicite chacune des principales contraintes de base.

<i>Contrainte</i>	<i>Vérification</i>
Date	Date valide
Time	Heure valide
Datetime	Date et heure valides

\* contraintes sur les fichiers

Le tableau suivant explicite chacune des contraintes sur les fichiers avec les options éventuelles.

<i>Contrainte</i>	<i>Vérification</i>	<i>Options éventuelles</i>
File	Fichier valide	maxSize : taille maximum du fichier
Image	Fichier image valide	maxSize : taille maximum du fichier minHeight : hauteur minimale maxHeight : hauteur maximale minWidth : largeur minimale maxWidth : largeur maximale

## g-4) exemple complet d'implémentation (couche *Modèle*)

### \* cahier des charges

Par rapport à la gestion de nos employés, voyons les règles à respecter :

- > Le code employé et le nom employé doivent être remplis.
- > Le code employé a une longueur de 7 caractères qui comprend 3 chiffres, 1 lettre majuscule et 3 chiffres.
- > L'année est sur 4 chiffres.
- > Le code postal est sur 5 caractères.
- > Le nombre d'enfants, qui est donc un entier, est au maximum de 7 (cette information est à connaître pour la prime de Noël qui se calcule en fonction de ce nombre avec un plancher à 7).
- > La saisie de l'email, de l'url du site le plus visité et de la date de retour de mission doivent être conformes.

### \* entité *Employe* avec les contraintes

Voici le nouveau code de l'entité *Employe* avec donc les contraintes sur les attributs (en gras) :

```
<?php

// src/Entity/Employe.php

namespace App\Entity;

// définition des alias
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 * @ORM\Table(name="employes")
 */
class Employe
{
    // attributs privés
    /**
```



```

* @Assert\NotBlank(message="Le code employé doit être
                        renseigné.")
* @Assert\Length(min=7, max=7,
                  exactMessage="Le code employé doit être
                                sur exactement
                                7 caractères.")
* @Assert\Regex(pattern="#[0-9][0-9][0-9][A-Z]
                  [0-9][0-9][0-9]#",
                 message="Le code employé doit comprendre 3
                           chiffres, puis 1 lettre
                           majuscule, puis 3 chiffres.")
* @ORM\Column(name="codeEmp", type="string", length=7)
* @ORM\Id
*/
private $code;
// ne pas oublier les dièses dans le pattern

/**
* @Assert\NotBlank(message="Le nom employé doit être
                        renseigné.")
* @ORM\Column(name="nomEmp", type="string", length=25)
*/
private $nom;

/**
* @ORM\Column(name="prenomEmp", type="string", length=20,
               nullable=true)
*/
private $prenom;

/**
* @Assert\Regex(pattern="#[0-9][0-9][0-9][0-9]#",
                 message="L'année d'embauche doit
                           comprendre 4 chiffres.")
* @ORM\Column(name="anEmbaucheEmp", type="integer",
               nullable=true)
*/
private $anEmbauche;

/**
* @ORM\Column(name="motDePasse", type="string", length=20,
               nullable=true)
*/
private $motDePasse;

/**
* @ORM\Column(name="adresseRue", type="string", length=80,
               nullable=true)
*/
private $adresseRue;

```

```

/**
 * @Assert\Length(min=5, max=5,
                  exactMessage="Le code postal doit être
                                sur exactement
                                5 caractères.")
 * @ORM\Column(name="codePostal", type="string", length=5,
               nullable=true)
 */
private $codePostal;

/**
 * @ORM\Column(name="ville", type="string", length=20,
               nullable=true)
 */
private $ville;

/**
 * @ORM\Column(name="situatFam", type="string", length=1,
               nullable=true)
 */
private $situatFam;

/**
 * @ORM\Column(name="trancheDureeS", type="smallint",
               nullable=true)
 */
private $trancheDureeS;

/**
 * @ORM\Column(name="vaccl", type="boolean",
               nullable=true)
 */
private $vaccl;

/**
 * @ORM\Column(name="vacc2", type="boolean",
               nullable=true)
 */
private $vacc2;

/**
 * @ORM\Column(name="vacc3", type="boolean",
               nullable=true)
 */
private $vacc3;

/**
 * @Assert\Type(type="int")
 * @Assert\Range(min=0, max=7,
                 minMessage="Le nombre d'enfants est d'au
                               moins 0.",

```

```

        maxMessage="Le nombre d'enfants est d'au
        plus 7.")
    * @ORM\Column(name="nbreEnf", type="smallint",
        nullable=true)
    */
private $nbreEnf;

/**
    * @Assert\Email(message="Adresse email invalide")
    * @ORM\Column(name="eMail", type="string", length=30,
        nullable=true)
    */
private $eMail;

/**
    * @Assert\Url(message="Adresse url invalide")
    * @ORM\Column(name="adrSite", type="string", length=30,
        nullable=true)
    */
private $adrSite;

/**
    * @Assert>Date(message="Date invalide")
    * @ORM\Column(name="dateRetourM", type="date",
        nullable=true)
    */
private $dateRetourM;

// getters et setters
.....
}

```

### Remarque :

Pour les tests, on enlèvera dans la méthode du contrôleur les options concernées au niveau de la génération du formulaire.

Par exemple pour le code employé, cela donnera :

```

->add('code', TextType::class,
    ['required' => false])

```

### g-5) test (extrait)

Testons la saisie du code employé avec donc les 3 contrôles correspondants aux 3 contraintes.

Voici ce que cela va donner par exemple :

Code

**ERREUR** Le code employé doit être renseigné.

Code

**ERREUR** Le code employé doit être sur exactement 7 caractères.

**ERREUR** Le code employé doit comprendre 3 chiffres, puis 1 lettre majuscule, puis 3 chiffres.

### g-6) messages et langue française

Le framework Symfony possède un certain nombre de messages notamment d'erreurs prédéfinis.

Ils sont en anglais : on l'a vu par exemple pour le champ de l'email avec le message par défaut sur la contrainte `Email` qui est :

***ERROR** This value is not a valid email address.*

Pour le tester à nouveau enlever l'option message correspondante.

\* `@Assert\Email()`

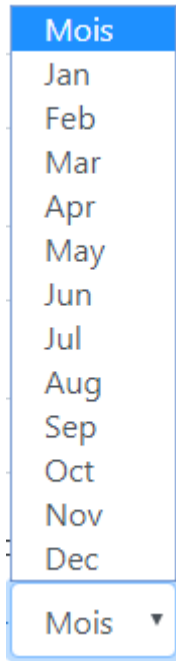
Cela donne :

Email

**ERROR** This value is not a valid email address.

D'autre part, par exemple pour les dates la liste déroulante des mois se met aussi en anglais.

Cela donne :



La langue est définie dans le fichier **config/packages/translation.yaml** ainsi :

```
framework:
    default_locale: en
```

Par défaut, la langue est donc en anglais.

Pour la mettre en français, la modification donne (en gras) :

```
framework:
    default_locale: fr
```

Le message par défaut sur la contrainte Email devient :

*ERREUR Cette valeur n'est pas une adresse email valide.*

Cela donne :

Email

**ERREUR** Cette valeur n'est pas une adresse email valide.

machintruc.fr

Et par exemple pour les dates la liste déroulante des mois se met en français.

Cela donne :

Mois
janv.
févr.
mars
avr.
mai
juin
juil.
août
sept.
oct.
nov.
déc.

Mois ▼

## 10) Relations entre entités

### a) présentation

Une base de données est un ensemble de tables en relation.

Cette relation se matérialise par une clé étrangère (ou externe ou secondaire) qui va référencer la clé primaire d'une autre table, ce qui permettra de mettre en œuvre l'intégrité référentielle.

Sous le framework Symfony, on n'accède pas directement à la base de données.

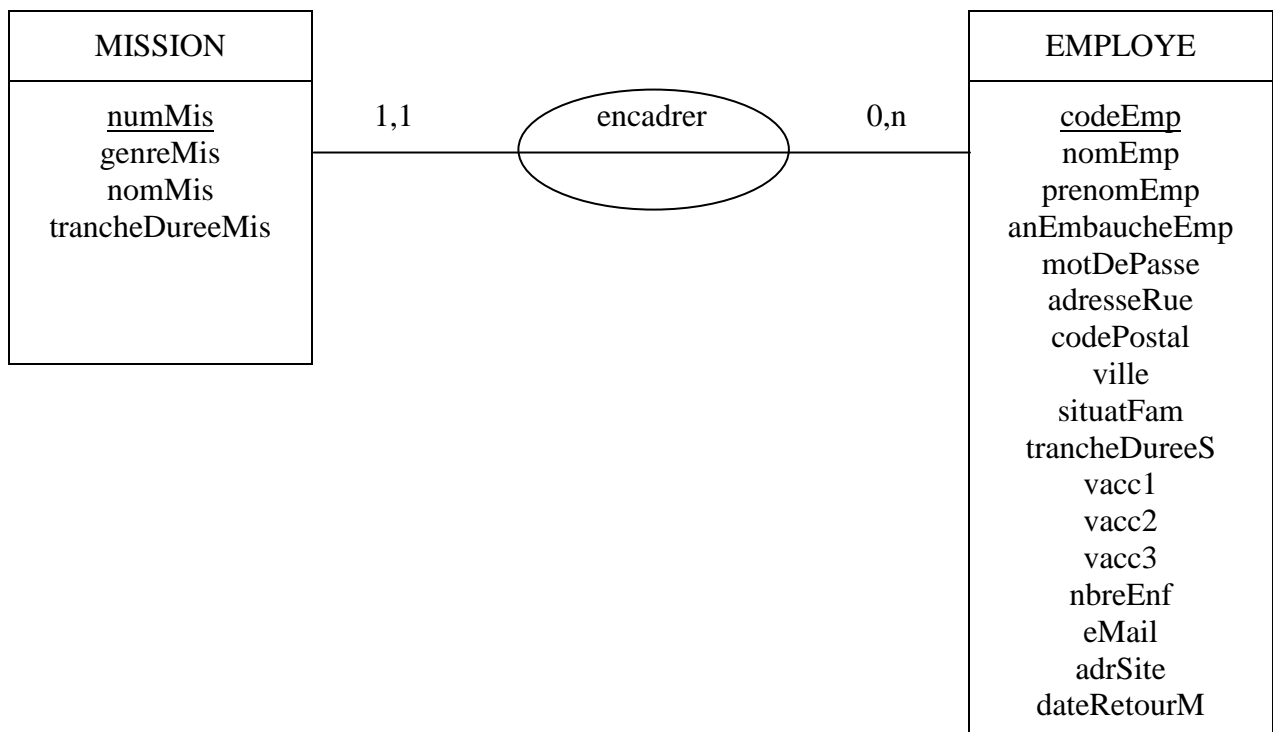
Tout se fait en amont via les entités et c'est ensuite Doctrine qui réalise le travail pour mettre en place les éléments de la base de données, et notamment les relations inter-tables.

On va donc agir sur les entités afin de créer ces liens.

### b) préliminaires : création de l'entité Mission

Rappelons le MCD de notre cas et le modèle relationnel déduit.

Cela donne :



et :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

On a une autre table qui est *missions*.

L'id est le numéro de mission qui est à incrémentation automatique.

Voici la nouvelle entité correspondante Mission, sans se préoccuper pour l'instant du lien avec l'autre entité Employe :

```
<?php

// src/Entity/Mission.php

namespace App\Entity;

// définition des alias
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 * @ORM\Table(name="missions")
 */
class Mission
{
    // attributs privés
    /**
     * @ORM\Column(name="numMis", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $num;

    /**
     * @Assert\NotBlank(message="Le genre de mission doit être renseigné.")
     * @ORM\Column(name="genreMis", type="string", length=15)
     */
    private $genre;

    /**
     * @Assert\NotBlank(message="Le nom mission doit être
```



```

                                renseigné.")
    * @ORM\Column(name="nomMis", type="string", length=25)
    */
    private $nom;

    /**
     * @ORM\Column(name="trancheDureeMis", type="smallint",
                    nullable=true)
     */
    private $trancheDureeMis;

    // getters et setters
    public function getNum()
    {
        return $this->num;
    }

    public function getGenre()
    {
        return $this->genre;
    }
    public function setGenre($genre)
    {
        $this->genre = $genre;
    }

    public function getNom()
    {
        return $this->nom;
    }
    public function setNom($nom)
    {
        $this->nom = $nom;
    }

    public function getTrancheDureeMis()
    {
        return $this->trancheDureeMis;
    }
    public function setTrancheDureeMis($trancheDureeMis)
    {
        $this->trancheDureeMis = $trancheDureeMis;
    }

}

```

Remarque :

Il n'y a pas de setter pour le numéro de mission car c'est un id à numérotation automatique.

A présent, occupons-nous du lien entre nos deux entités `Employe` et `Mission`.

c) les 3 relations possibles et les cardinalités

Doctrine reconnaît 3 relations :

- la relation *One-To-One* : une entité X est associée à une (et une seule) entité Y (1-1),
- la relation *Many-To-One* : plusieurs entités X sont associées à une entité Y (n-1) ,
- la relation *Many-To-Many* : plusieurs entités X sont associées à plusieurs entités Y (n-n).

La mise en place de chacune de ces relations se fait via l'annotation correspondante.

Par exemple, la relation *One-To-One* se fait via l'annotation `@ORM\One-To-One`.

Revenons-en à notre MCD et voyons la correspondance avec les types de relation Doctrine.

***Les nombres exprimés dans la relation correspondent aux cardinalités maximum dans le MCD.***

Le tableau suivant fournit la correspondance.

<i>Relation Doctrine</i>	<i>Cardinalités du MCD</i>
<b><i>One-To-One</i></b>	<b><i>x,1 - x,1</i></b>
<b><i>Many-To-One</i></b>	<b><i>x,n - x,1</i></b>
<b><i>Many-To-Many</i></b>	<b><i>x,n - x,n</i></b>

x est égal à 0 ou 1.

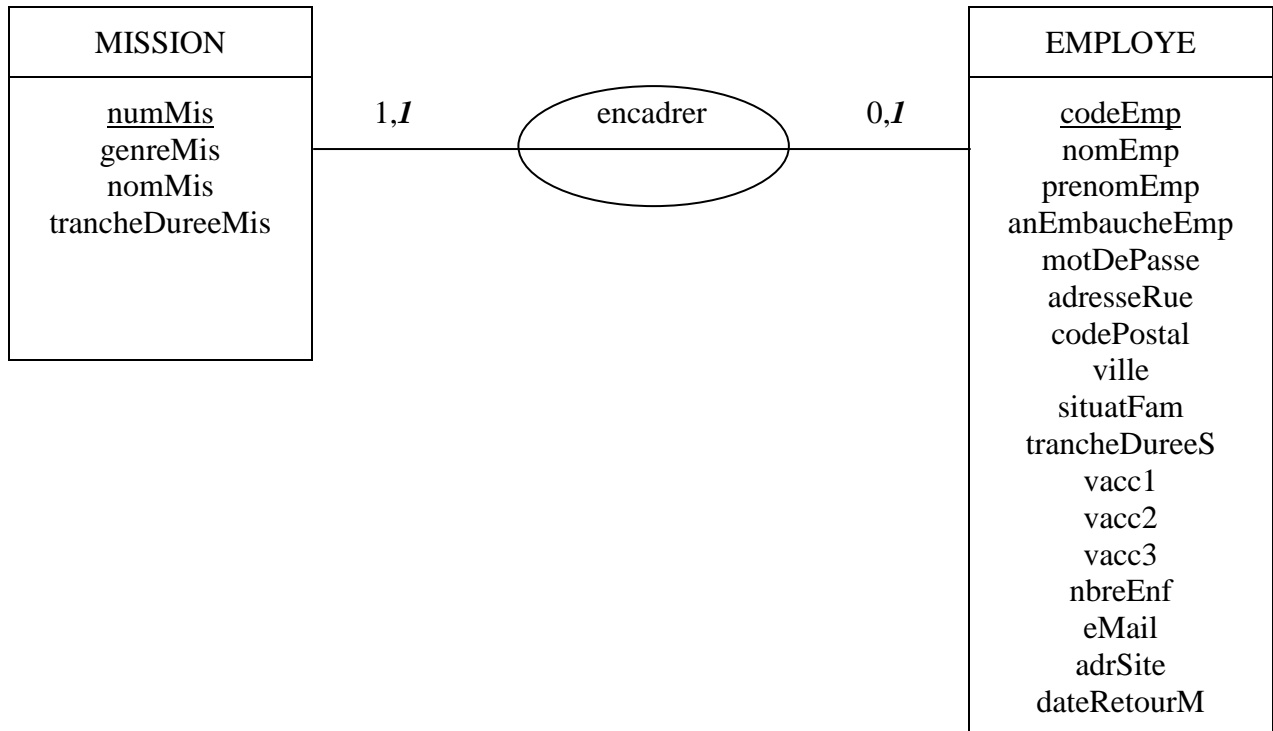
Pour voir tous les cas de figure, on changera au fur et à mesure les cardinalités dans le MCD d'exemple.

## d) mise en place d'une relation *One-To-One*

### d-1) cas d'exemple et code

Considérons à nouveau notre domaine d'encadrement des missions.

Une relation *One-To-One* Doctrine correspond à la mise en place du modèle conceptuel suivant (cela se joue sur les cardinalités maximum) :



On y stipule qu'une mission n'est encadrée que par ***un seul*** employé et qu'un employé ne peut encadrer qu'***une seule*** mission.

Le modèle relationnel déduit est le suivant :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

La relation *One-To-One* s'implémentera en amont, entre l'entité Mission et l'entité Employe.

Si on raisonne au niveau relationnel, on met dans la table *missions* un ***champ*** *codeEmp* qui est la clé étrangère.

Mais ici, nous devons travailler sur les entités donc sur un *objet*, Doctrine assurant ensuite le mapping objet-relationnel pour gérer au final une clé étrangère.

Voyons comment mettre en place une relation *One-To-One*.

Cette relation part de l'entité *Mission* car on veut signifier qu'une mission est encadrée par un employé.

Le code de l'entité *Mission* devient (extrait) :

```
.....
private $trancheDureeMis;

/**
 * @ORM\OneToOne
 *         (targetEntity="App\Entity\Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                 referencedColumnName="codeEmp",
 *                 nullable=false)
 */
private $employe;

// getters et setters
.....
public function getEmploye()
{
    return $this->employe;
}
public function setEmploye(Employe $employe)
{
    $this->employe = $employe;
}
.....
```

### Commentaires :

Il faut faire plusieurs choses :

- rajouter un attribut privé `$employe` qui est un objet,
- mettre une annotation `@ORM\OneToOne` en indiquant que l'entité cible est `Employe` via l'option `targetEntity`,
- mettre une annotation `@ORM\JoinColumn` en indiquant :
  - > le nom à utiliser pour la clé étrangère à générer (ici de la table *missions*) via l'option `name`,
  - > le nom de la clé primaire de la table référencée (ici de la table *employes*) via l'option `referencedColumnName`,
  - > une valeur requise ou non pour la clé étrangère via l'option `nullable`.

Si `nullable` est à `true` (valeur par défaut) la clé étrangère peut être vide, sinon elle doit être obligatoire.

On crée ensuite corrélativement en plus le getter et le setter sur l'objet `$employe`.

A présent lançons les commandes pour la mise à jour de la base de données :

**php bin/console make:migration**

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

## d-2) tests

### \* tests au sein de la base de données

Vérifier que la table *missions* a bien été créée et surtout que la clé étrangère a bien été mise en œuvre.

De plus, il faut tester une chose importante par rapport aux cardinalités de notre modèle : un employé ne peut encadrer qu'une seule mission, donc on ne pourra pas mettre le même code employé en clé étrangère sur deux missions différentes.

### \* tests au sein d'un contrôleur

Cela donne :

```
<?php
```

```
// src/Controller/TestBDController.php
```

```
namespace App\Controller;
```

```
use Doctrine\Persistence\ManagerRegistry;
```

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

```
use Symfony\Component\HttpFoundation\Response;
```

```
use Symfony\Component\Routing\Annotation\Route;
```

```
// importation des entités
```

```
use App\Entity\Employe;
```

```
use App\Entity\Mission;
```

```
class TestBDController extends AbstractController
{
```

```

/**
 * @Route("/testBD")
 */
public function manip(ManagerRegistry $doctrine)
{

    // récupération de l'Entity Manager
    $em = $doctrine->getManager();

    // instantiation 1 de l'entité Employe et injection des données obligatoires
    $employe1 = new Employe();
    $employe1->setCode("111A111");
    $employe1->setNom("TRUC");
    // notification de la modification de l'entité
    $em->persist($employe1);

    // instantiation 2 de l'entité Employe et injection des données obligatoires
    $employe2 = new Employe();
    $employe2->setCode("222B222");
    $employe2->setNom("MACHIN");
    // notification de la modification de l'entité
    $em->persist($employe2);

    /* instantiation 1 de l'entité Mission, injection des données obligatoires
    avec affectation de l'employé 1 */
    $mission1 = new Mission();
    $mission1->setGenre("P");
    $mission1->setNom("Mission 1");
    $mission1->setEmploye($employe1);
    // notification de la modification de l'entité
    $em->persist($mission1);

    /* instantiation 2 de l'entité Mission, injection des données obligatoires
    avec affectation de l'employé 2 */
    $mission2 = new Mission();
    $mission2->setGenre("T");
    $mission2->setNom("Mission 2");
    $mission2->setEmploye($employe2);
    // notification de la modification de l'entité
    $em->persist($mission2);

    // demande de modification de la base de données
    $em->flush();

    return new Response("Insertions réussies !");

}
}

```

### d-3) notations simplifiées

L'option `targetEntity` a ici pour valeur le chemin complet de l'entité cible du lien : nom FQCN (*Fully Qualified Class Name*).

Comme les deux entités `Mission` et `Employe` appartiennent au même espace de noms, on peut omettre cette option.

Cela donne donc plus simplement (la modification est en gras) :

```
/**
 * @ORM\OneToOne(targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                  referencedColumnName="codeEmp",
 *                  nullable=false)
 */
private $employe;
```

A noter à titre informatif que l'annotation `@ORM\JoinColumn` n'est pas obligatoire si la clé primaire de la table référencée, ici *employes*, s'appelle *id*.

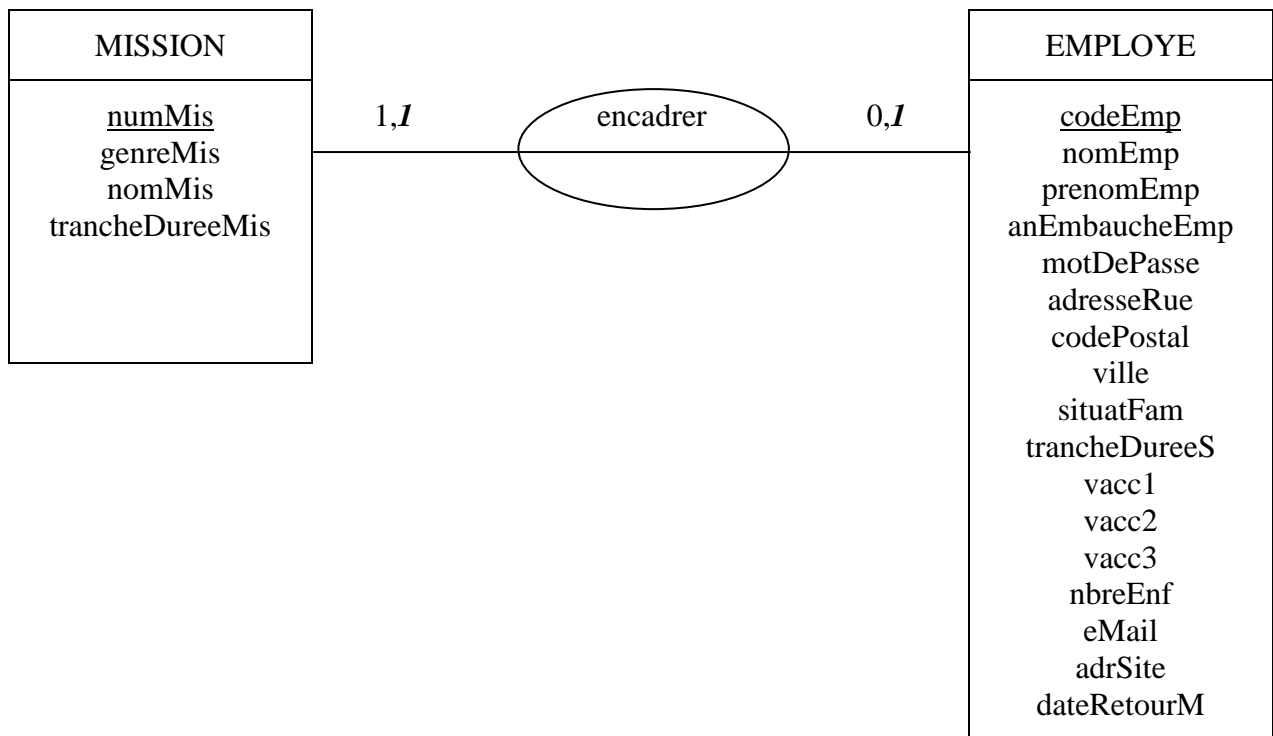
Cela donnerait alors plus simplement, s'il n'y avait pas la ligne `nullable=false` :

```
/**
 * @ORM\OneToOne(targetEntity="Employe")
 */
private $employe;
```

Par défaut, Doctrine nomme alors la clé étrangère avec le nom de l'entité en minuscule concaténé avec *\_id* : cela donne donc ici *employe\_id*.

d-4) variante de cardinalités et mise à jour d'annotation

Considérons à nouveau notre MCD.



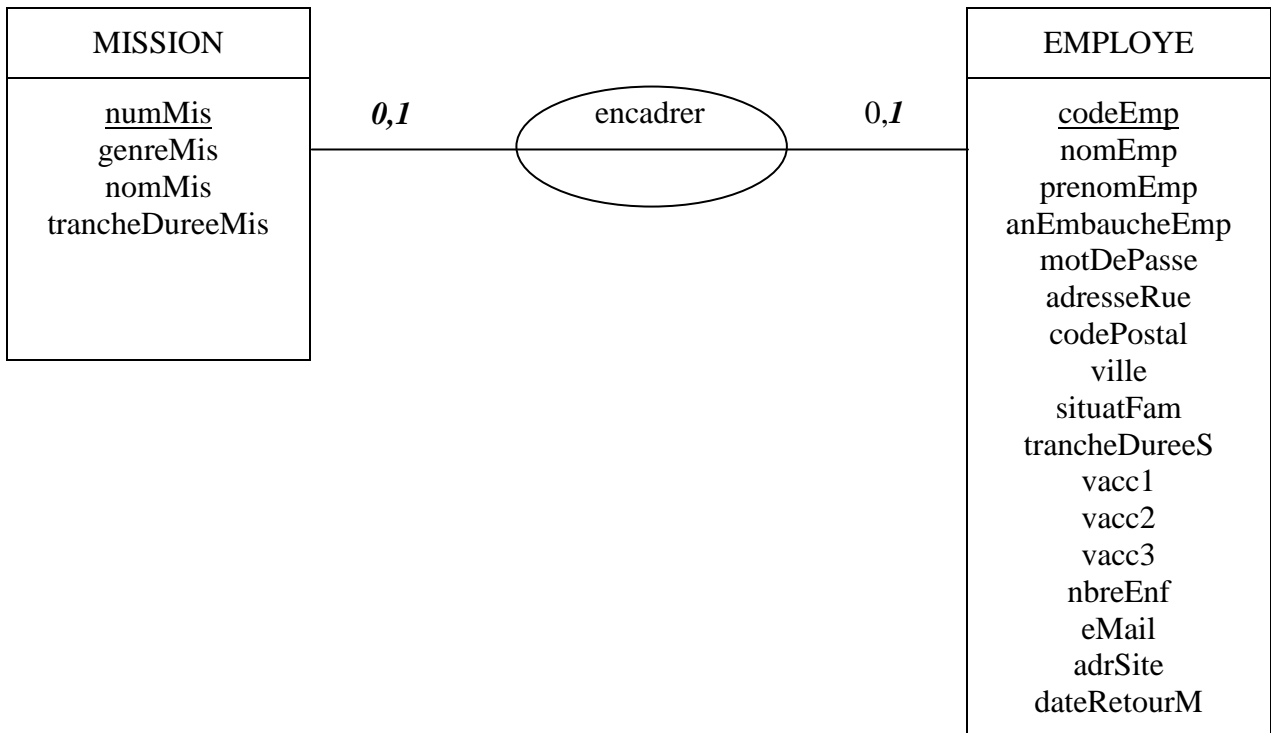
Pour l'instant, **une** mission doit être encadrée par **un** employé.

Supposons que certaines missions soient encadrées par un organisme extérieur, plus par un employé.

Les cardinalités côté MISSION passent de 1,1 à 0,1.

Cela donne :





Dans l'annotation `@ORM\JoinColumn`, il faut positionner du coup l'option `nullable` à `true`.

Cela donne (la modification est en gras) :

```

/**
 * @ORM\OneToOne(targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                  referencedColumnName="codeEmp",
 *                  nullable=true)
 */
private $employe;
  
```

Comme `nullable` a par défaut la valeur `true` (Doctrine propose le moins restrictif), on peut écrire plus simplement :

```

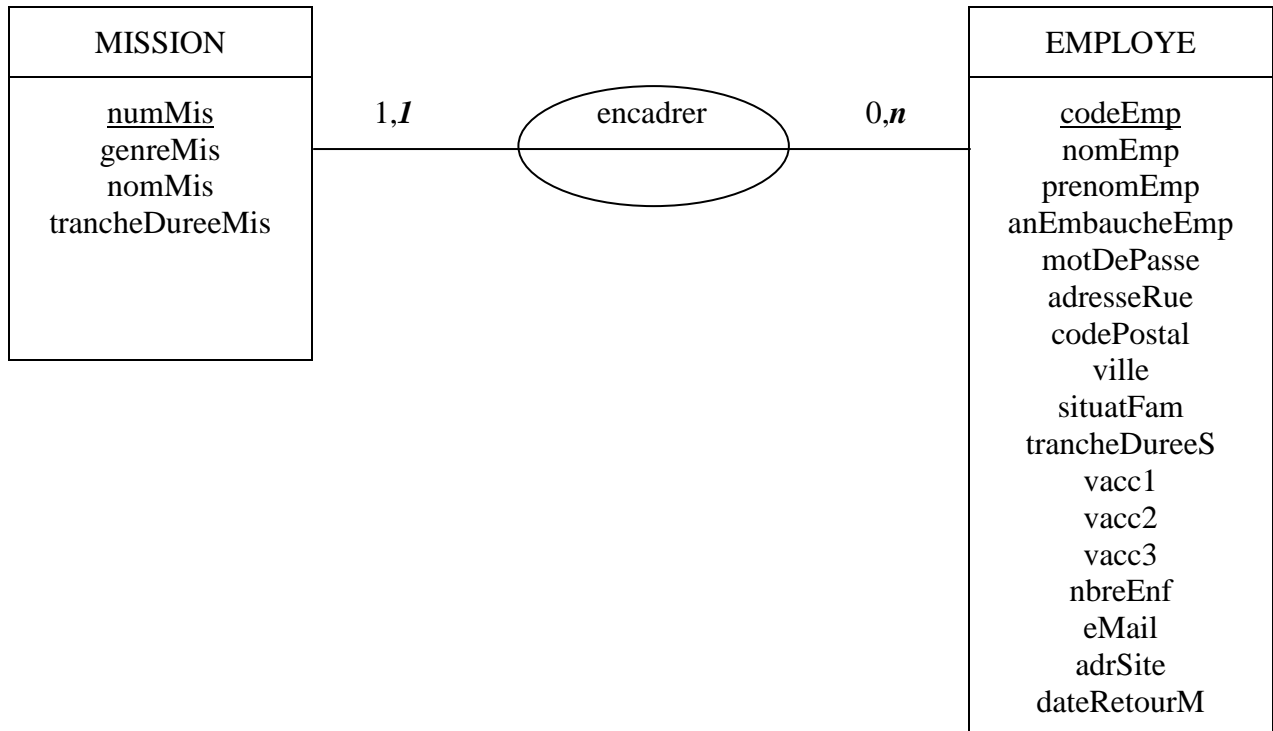
/**
 * @ORM\OneToOne(targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                  referencedColumnName="codeEmp")
 */
private $employe;
  
```

Pour la suite, on supprimera dans l'entité `Mission` cette relation *One-To-One* (avec son attribut `$employe` et son `getter/setter`), et au sein de la base de données la table *missions* ainsi que les deux employés rajoutés via le contrôleur.

e) mise en place d'une relation *Many-To-One*e-1) cas d'exemple et code

Considérons à nouveau notre domaine d'encadrement des missions.

Une relation *Many-To-One* Doctrine correspond à la mise en place du modèle conceptuel suivant (cela se joue sur les cardinalités maximum) :



On y stipule qu'une mission n'est encadrée que par un seul employé et qu'un employé peut encadrer plusieurs missions.

Dit autrement, *plusieurs* missions sont encadrées par *un* employé.

Le modèle relationnel déduit est le suivant :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

Au niveau des champs, on va obtenir la même chose pour la relation *Many-To-One* par rapport à une relation *One-To-One*.

Par contre, on verra plus tard dans la base de données qu'on pourra affecter le même code employé à 2 missions différentes.

Voyons comment mettre en place une relation *Many-To-One*.

Tout est pareil que pour une relation *One-To-One* sauf que bien sûr l'annotation sera `@ORM\ManyToOne` et non plus `@ORM\OneToOne`.

Cette relation part de l'entité `Mission` car on veut signifier que plusieurs missions sont encadrées par un employé.

Le code de l'entité `Mission` devient (extrait) :

```
.....
private $trancheDureeMis;

/**
 * @ORM\ManyToOne(targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                  referencedColumnName="codeEmp",
 *                  nullable=false)
 */
private $employe;

// getters et setters
.....
public function getEmploye()
{
    return $this->employe;
}
public function setEmploye(Employe $employe)
{
    $this->employe = $employe;
}
.....
```

### Commentaires :

Il faut faire plusieurs choses :

- ajouter un attribut privé `$employe` qui est un objet,
- mettre une annotation `@ORM\ManyToOne` en indiquant que l'entité cible est `Employe` via l'option `targetEntity`,
- mettre une annotation `@ORM\JoinColumn` en indiquant :
  - > le nom à utiliser pour la clé étrangère à générer (ici de la table *missions*) via l'option `name`,
  - > le nom de la clé primaire de la table référencée (ici de la table *employes*) via l'option `referencedColumnName`,
  - > une valeur requise ou non pour la clé étrangère via l'option `nullable`.

Si `nullable` est à `true` (valeur par défaut) la clé étrangère peut être vide, sinon elle doit être obligatoire.

On crée ensuite corrélativement en plus le getter et le setter sur l'objet `$employe`.

#### Rappel :

L'annotation `@ORM\JoinColumn` n'est pas obligatoire si la clé primaire de la table référencée, ici *employes*, s'appelle *id*.

Cela donnerait alors plus simplement, s'il n'y avait pas la ligne `nullable=false` :

```
/**
 * @ORM\ManyToOne(targetEntity="Employe")
 */
private $employe;
```

Par défaut, Doctrine nomme alors la clé étrangère avec le nom de l'entité en minuscule concaténé avec `_id` : cela donne donc ici *employe\_id*.

A présent lançons les commandes pour la mise à jour de la base de données :

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

## e-2) tests

### \* tests au sein de la base de données

Vérifier que la table *missions* a bien été créée et surtout que la clé étrangère a bien été mise en œuvre.

De plus, il faut tester une chose importante par rapport aux cardinalités de notre modèle : un employé peut cette fois encadrer plusieurs missions, donc on mettra le même code employé en clé étrangère sur deux missions différentes.

### \* tests au sein d'un contrôleur

Cela donne :

```
<?php

// src/Controller/TestBDController.php
```

```

namespace App\Controller;

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

// importation des entités
use App\Entity\Employe;
use App\Entity\Mission;

class TestBDController extends AbstractController
{
    /**
     * @Route("/testBD")
     */
    public function manip(ManagerRegistry $doctrine)
    {
        // récupération de l'Entity Manager
        $em = $doctrine->getManager();

        // instantiation de l'entité Employe et injection des données obligatoires
        $employe1 = new Employe();
        $employe1->setCode("111A111");
        $employe1->setNom("TRUC");
        // notification de la modification de l'entité
        $em->persist($employe1);

        /* instantiation 1 de l'entité Mission, injection des données obligatoires
           avec affectation de l'employé */
        $mission1 = new Mission();
        $mission1->setGenre("P");
        $mission1->setNom("Mission 1");
        $mission1->setEmploye($employe1);
        // notification de la modification de l'entité
        $em->persist($mission1);

        /* instantiation 2 de l'entité Mission, injection des données obligatoires
           avec affectation du même employé */
        $mission2 = new Mission();
        $mission2->setGenre("T");
        $mission2->setNom("Mission 2");
        $mission2->setEmploye($employe1);
        // notification de la modification de l'entité
        $em->persist($mission2);

        // demande de modification de la base de données
    }
}

```

```

$em->flush();

return new Response("Insertions réussies !");

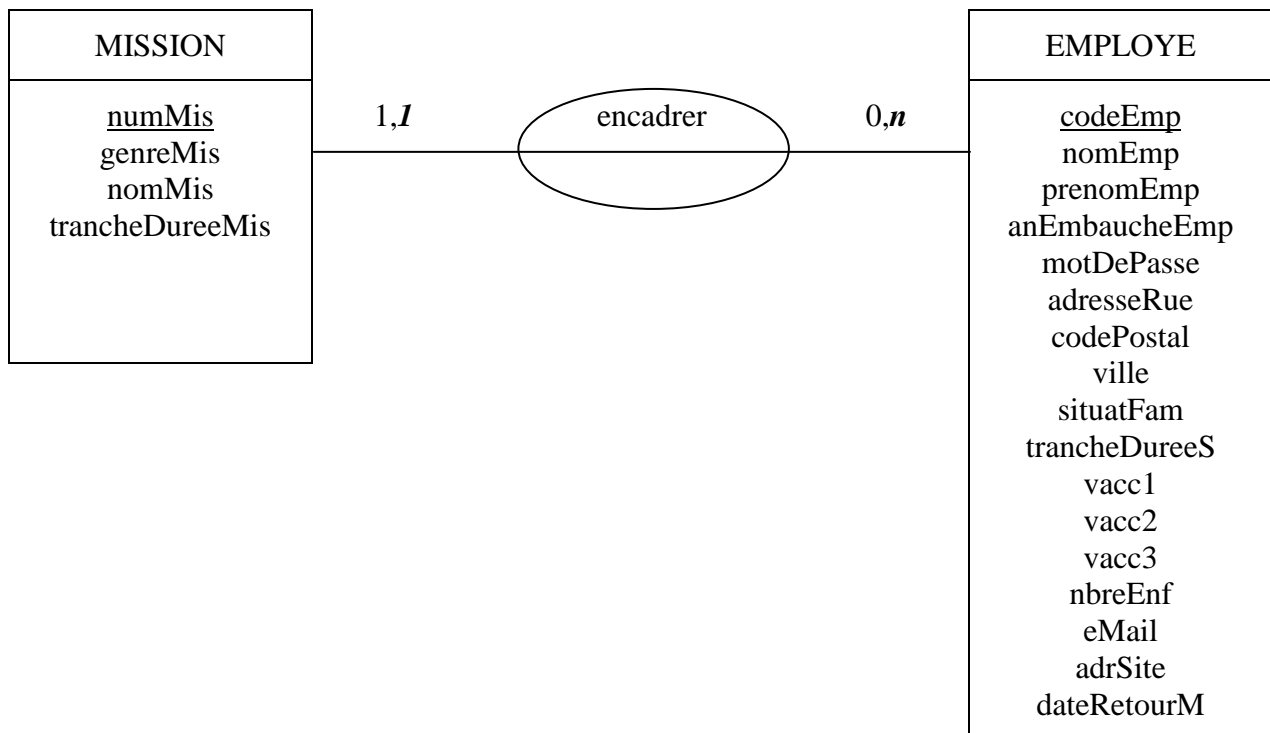
}
}

```

Nous allons voir à présent la même variante de cardinalités que pour la relation *One-To-One*.

### e-3) variante de cardinalités et mise à jour d'annotation

Considérons à nouveau notre MCD.

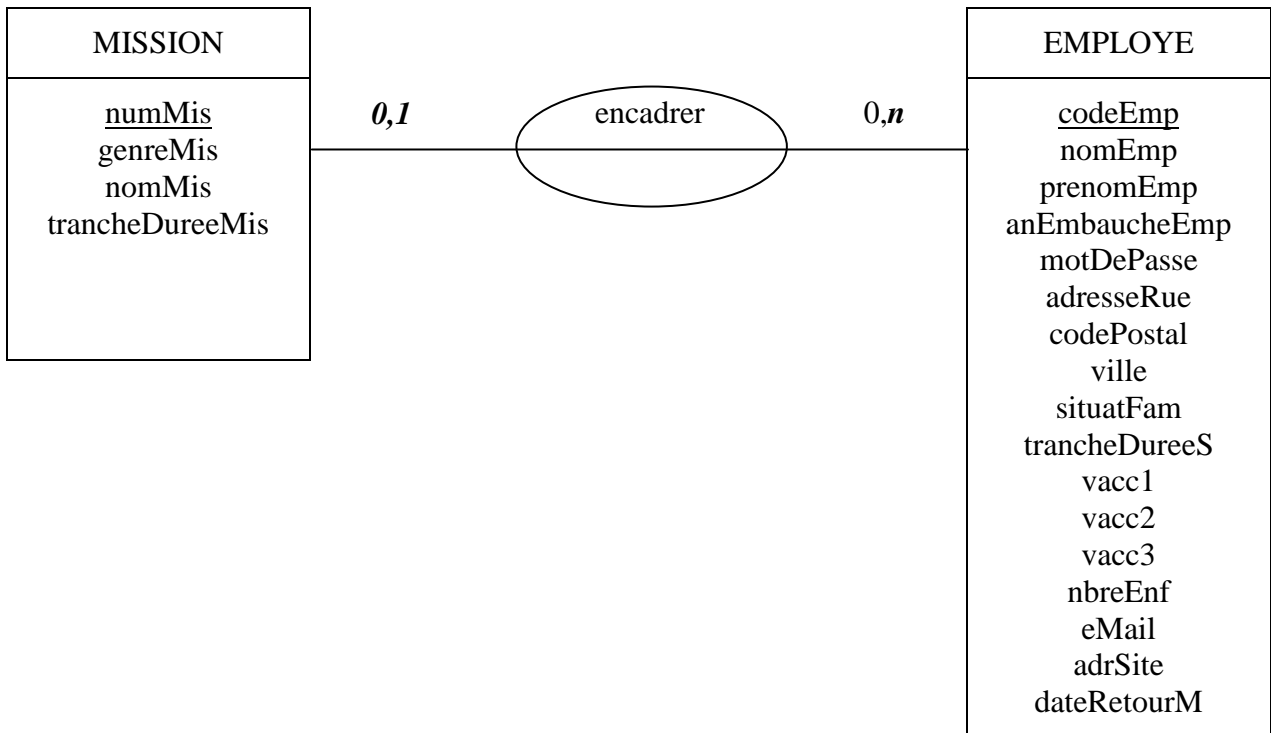


Pour l'instant, une mission doit être encadrée par un employé.

Supposons que certaines missions soient encadrées par un organisme extérieur, plus par un employé.

Les cardinalités côté MISSION passent de 1,1 à 0,1.

Cela donne :



Dans l'annotation `@ORM\JoinColumn`, il faut positionner du coup l'option `nullable` à `true`.

Cela donne (la modification est en gras) :

```

/**
 * @ORM\ManyToOne
 *         (targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                 referencedColumnName="codeEmp",
 *                 nullable=true)
 */
private $employe;
  
```

Comme `nullable` a par défaut la valeur `true` (Doctrine propose le moins restrictif), on peut écrire plus simplement :

```

/**
 * @ORM\ManyToOne
 *         (targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                 referencedColumnName="codeEmp")
 */
private $employe;
  
```

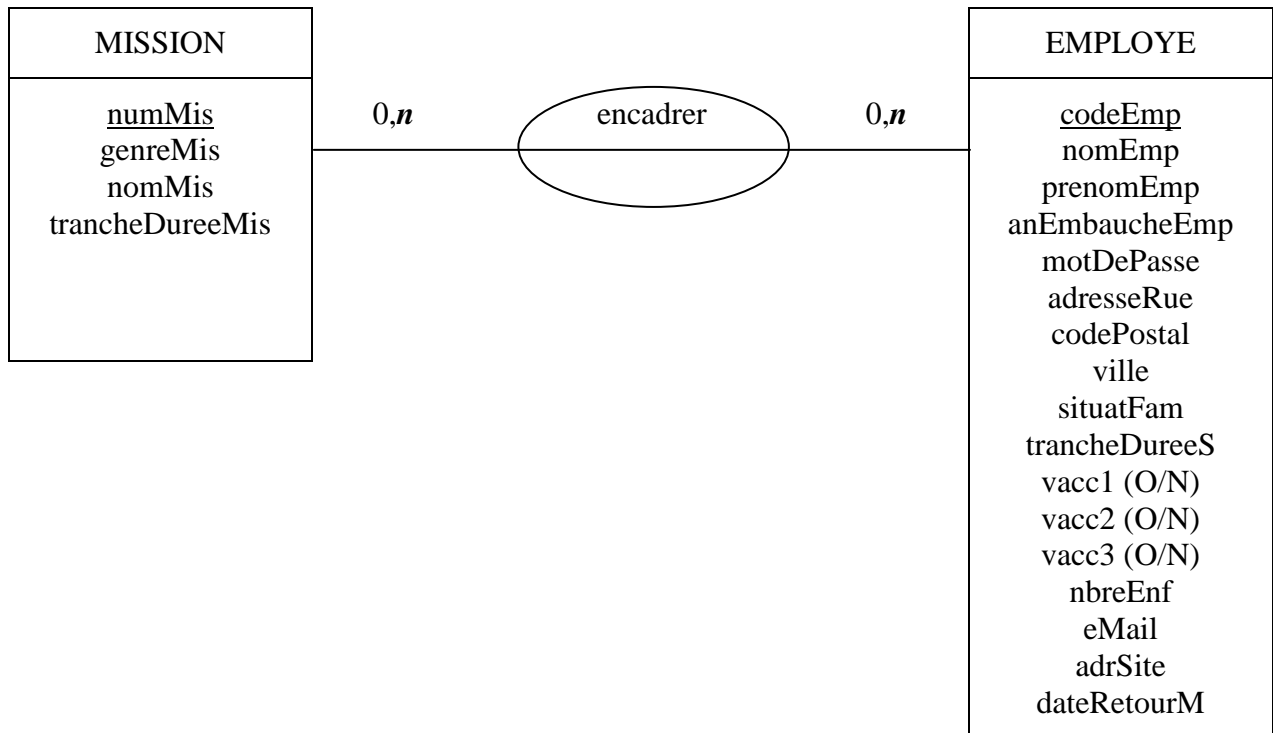


Pour la suite, on supprimera dans l'entité `Mission` cette relation *Many-To-One* (avec son attribut `$employe` et son `getter/setter`), et au sein de la base de données la table *missions* ainsi que l'employé rajouté via le contrôleur.

f) mise en place d'une relation *Many-To-Many*f-1) cas d'exemple et code

Considérons à nouveau notre domaine d'encadrement des missions.

Une relation *Many-To-Many* Doctrine correspond à la mise en place du modèle conceptuel suivant (cela se joue sur les cardinalités maximum) :



On y stipule qu'une mission peut être encadrée par plusieurs employés et qu'un employé peut encadrer plusieurs missions.

Dit autrement *plusieurs* missions sont encadrées par *plusieurs* employés.

Le modèle relationnel déduit est le suivant :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis)

ENCADRER (#numMis, #codeEmp )

Au final, il y a aura une nouvelle table à implanter et il n'y aura plus de clé secondaire `#codeEmp` dans la table *missions*.

Si les relations *One-To-One* et *Many-To-One* s'implémentent de manière très similaire, ce n'est pas le cas de la relation *Many-To-Many*.

Cette relation *Many-To-Many* part toujours de l'entité *Mission* car on veut signifier que plusieurs missions sont encadrées par plusieurs employés.

Le code de l'entité *Mission* devient (extrait) :

```
.....
/**
 * @ORM\ManyToMany(targetEntity="Employe")
 * @ORM\JoinTable(name="encadrer",
 *               joinColumns=
 *                 {
 *                   @ORM\JoinColumn(name="numMis",
 *                                   referencedColumnName="numMis")
 *                 },
 *               inverseJoinColumns=
 *                 {
 *                   @ORM\JoinColumn(name="codeEmp",
 *                                   referencedColumnName="codeEmp")
 *                 })
 */
private $employes;

// getters et setters
.....
public function getEmployes()
{
    return $this->employes;
}
public function addEmploye(Employe $employe)
{
    $this->employes[] = $employe;
}
.....
```

### Commentaires :

Il faut faire plusieurs choses :

- rajouter un attribut privé `$employes` (au pluriel) qui est un objet de type tableau,
- mettre une annotation `@ORM\ManyToMany` en indiquant que l'entité cible est *Employe* via l'option `targetEntity`,

- mettre une annotation `@ORM\JoinTable` en indiquant :
  - > le nom à utiliser pour la table à générer (ici la table *encadrer*) via l'option `name`,
  - > la première clé externe via l'option `joinColumns` avec dedans une annotation `@ORM\JoinColumn` stipulant la clé étrangère à générer (ici de la table *encadrer*) via l'option `name` et la clé primaire de la table référencée (ici de la table *missions*) via l'option `referencedColumnName`,
  - > la seconde clé externe via l'option `inverseJoinColumns` avec dedans de la même manière une annotation `@ORM\JoinColumn` stipulant la clé étrangère à générer (ici de la table *encadrer*) via l'option `name` et la clé primaire de la table référencée (ici de la table *employes*) via l'option `referencedColumnName`.

On crée ensuite corrélativement en plus le getter sur l'objet-tableau `$employes`, et non plus un setter qui n'aurait pas de sens mais une méthode nommée `addEmploye()` permettant de rajouter un employé à notre tableau `$employes`.

A présent lançons les commandes pour la mise à jour de la base de données :

**php bin/console make:migration**

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

Cela va avoir pour effet de créer deux tables : *missions* et *encadrer*.

## f-2) tests

### \* tests au sein de la base de données

Vérifier que les tables *missions* et *encadrer* ont bien été créées et que la clé primaire de *encadrer* est bien composée des deux clés étrangères.

Ensuite, on met deux employés sur une première mission et ces deux mêmes employés sur une seconde mission.

### \* tests au sein d'un contrôleur

Cela donne :

```

<?php

// src/Controller/TestBDController.php

namespace App\Controller;

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

// importation des entités
use App\Entity\Employe;
use App\Entity\Mission;

class TestBDController extends AbstractController
{
    /**
     * @Route("/testBD")
     */
    public function manip(ManagerRegistry $doctrine)
    {
        // récupération de l'Entity Manager
        $em = $doctrine->getManager();

        // instantiation 1 de l'entité Employe et injection des données obligatoires
        $employe1 = new Employe();
        $employe1->setCode("111A111");
        $employe1->setNom("TRUC");
        // notification de la modification de l'entité
        $em->persist($employe1);

        // instantiation 2 de l'entité Employe et injection des données obligatoires
        $employe2 = new Employe();
        $employe2->setCode("222B222");
        $employe2->setNom("MACHIN");
        // notification de la modification de l'entité
        $em->persist($employe2);

        // instantiation 1 de l'entité Mission et injection des données obligatoires
        $mission1 = new Mission();
        $mission1->setGenre("P");
        $mission1->setNom("Mission 1");
        // notification de la modification de l'entité
        $em->persist($mission1);

        // instantiation 2 de l'entité Mission et injection des données obligatoires
        $mission2 = new Mission();

```

```

$mission2->setGenre("T");
$mission2->setNom("Mission 2");
// notification de la modification de l'entité
$em->persist($mission2);

// affectation des deux employés à la mission 1
$mission1->addEmploye($employe1);
$mission1->addEmploye($employe2);
// notification de la modification de l'entité
$em->persist($mission1);

// affectation des deux employés à la mission 2
$mission2->addEmploye($employe1);
$mission2->addEmploye($employe2);
// notification de la modification de l'entité
$em->persist($mission2);

// demande de modification de la base de données
$em->flush();

return new Response("Insertions réussies !");
}
}

```

### f-3) notation simplifiée

Notons à titre informatif que l'annotation `@ORM\JoinTable` n'est pas obligatoire si chacune des deux clés primaires des tables référencées, ici *employes* et *missions*, s'appelle *id*.

Cela donne alors (beaucoup) plus simplement :

```

/**
 * @ORM\ManyToMany(targetEntity="Employe")
 */
private $employes;

```

Par défaut, Doctrine crée alors ici une table avec le nom des deux entités jointes en minuscule séparé par un underscore, soit ici *mission\_employe*, avec deux colonnes comme clés étrangères ayant chacune comme nom celui de l'entité concerné en minuscule concaténé avec *\_id*, soit ici respectivement *mission\_id* et *employe\_id*.

Pour la suite, on supprimera dans l'entité *Mission* cette relation *Many-To-Many* (avec son attribut *\$employes* et son getter/méthode d'ajout), et au sein de la base de données les tables *encadrer* et *missions*, ainsi que les deux employés rajoutés via le contrôleur.

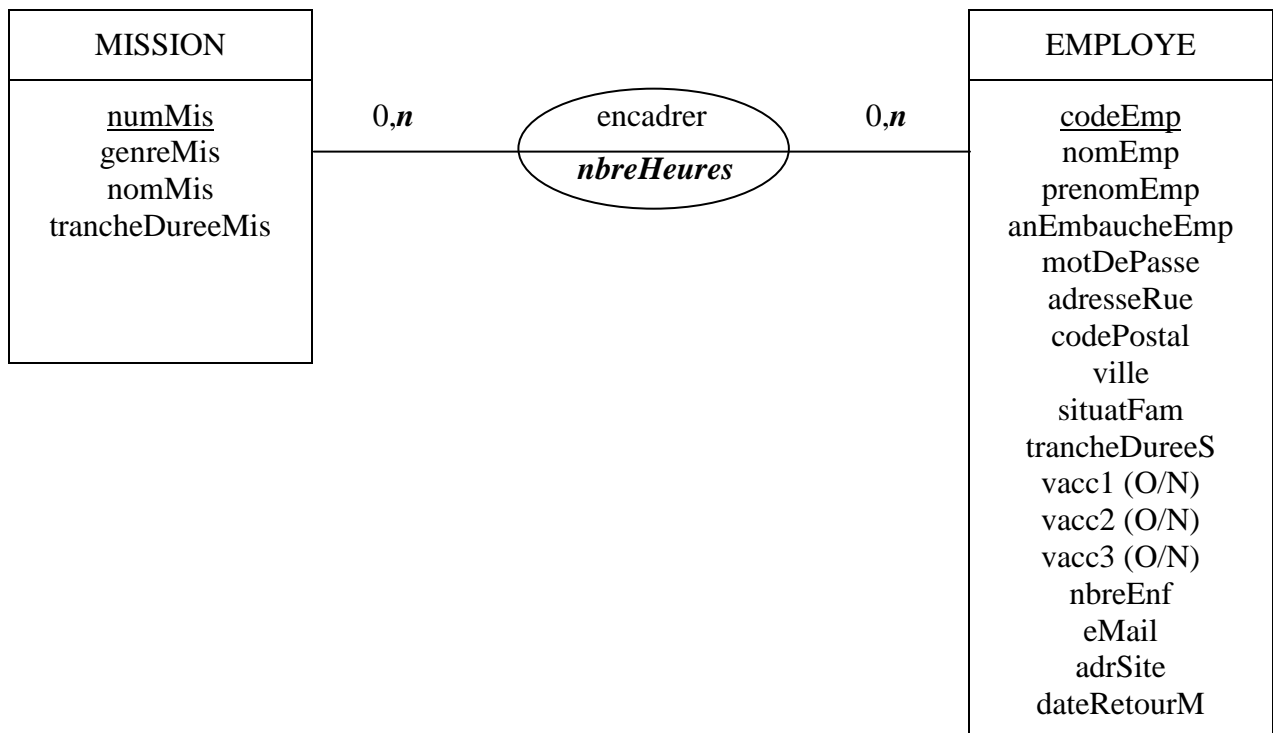
g) mise en place d'une relation *Many-To-Many* avec attributs : création d'une entité ayant deux relations *Many-To-One*

g-1) cas d'exemple et code

Considérons à nouveau notre domaine d'encadrement des missions.

On souhaite à présent mémoriser le nombre d'heures d'encadrement de chaque mission par chaque employé.

Le modèle conceptuel devient :



Le modèle relationnel déduit est le suivant :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis)

ENCADRER (#numMis, #codeEmp, **nbreHeures**)

La question est où mettre cet attribut nbreHeures ?

On ne peut pas le mettre dans la définition d'une relation *Many-To-Many* car l'annotation correspondante ne prévoit pas ce cas d'ajout d'attribut.

Il va falloir créer une nouvelle entité qu'on appellera *Encadrement* avec dedans deux attributs objets : *\$mission* et *\$employe*.

Dans cette nouvelle entité, on va mettre en place deux relations *Many-To-One* :

- une relative à *\$mission* car plusieurs encadrements concernent une mission (une mission est encadrée plusieurs fois),
- une relative à *\$employe* car plusieurs encadrements concernent un employé (un employé encadre plusieurs fois).

La future clé primaire étant la concaténation des deux clés étrangères, il faudra en amont mettre une annotation *@ORM\Id* pour chacun des deux attributs objets *\$mission* et *\$employe*.

Voici le code complet de la nouvelle entité *Encadrement* :

```
<?php

// src/Entity/Encadrement.php

namespace App\Entity;

// définition des alias
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 * @ORM\Table(name="encadrer")
 */
class Encadrement
{
    // attributs privés

    /**
     * @ORM\ManyToOne(targetEntity="Mission")
     * @ORM\JoinColumn(name="numMis",
                     referencedColumnName="numMis",
                     nullable=false)
     * @ORM\Id
     */
    private $mission;

    /**
     * @ORM\ManyToOne(targetEntity="Employe")
     * @ORM\JoinColumn(name="codeEmp",
                     referencedColumnName="codeEmp",
                     nullable=false)
     * @ORM\Id
```



```

    */
    private $employe;

    /**
     * @ORM\Column(name="nbreHeures", type="smallint",
     *             nullable=true)
     */
    private $nbreHeures;

    // getters et setters
    public function getMission()
    {
        return $this->mission;
    }
    public function setMission(Mission $mission)
    {
        $this->mission = $mission;
    }

    public function getEmploye()
    {
        return $this->employe;
    }
    public function setEmploye(Employe $employe)
    {
        $this->employe = $employe;
    }

    public function getNbreHeures()
    {
        return $this->nbreHeures;
    }
    public function setNbreHeures($nbreHeures)
    {
        $this->nbreHeures = $nbreHeures;
    }

}

```

### Commentaires :

Dans un premier temps, on stipule que la classe déclarée doit être vue comme une entité et que la table correspondante devra s'appeler *encadrer*.

Ensuite, on met un attribut privé \$mission (objet) avec notamment une annotation @ORM\ManyToOne en indiquant que l'entité cible est Mission.

On met un autre attribut privé `$employe` (objet) avec notamment une annotation `@ORM\ManyToOne` en indiquant que l'entité cible est `Employe`.

Puis, on rajoute un attribut `$nbreHeures` qui est la fameuse donnée engendrée par l'association d'une mission et d'un employé.

Enfin, on définit les différents getters et setters.

A présent lançons les commandes pour la mise à jour de la base de données :

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

## g-2) tests

### \* tests au sein de la base de données

Vérifier que les tables *missions* et *encadrer* ont bien été créées, que la clé primaire de *encadrer* est bien composée des deux clés étrangères et que le nombre d'heures a bien été ajouté.

Ensuite, on met deux employés sur une première mission et ces deux mêmes employés sur une seconde mission avec à chaque fois un certain nombre d'heures.

### \* tests au sein d'un contrôleur

Cela donne :

```
<?php

// src/Controller/TestBDController.php

namespace App\Controller;

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

// importation des entités
use App\Entity\Employe;
```

```

use App\Entity\Mission;
use App\Entity\Encadrement;

class TestBDController extends AbstractController
{

    /**
     * @Route("/testBD")
     */
    public function manip(ManagerRegistry $doctrine)
    {

        // récupération de l'Entity Manager
        $em = $doctrine->getManager();

        // instantiation 1 de l'entité Employe et injection des données obligatoires
        $employe1 = new Employe();
        $employe1->setCode("111A111");
        $employe1->setNom("TRUC");
        // notification de la modification de l'entité
        $em->persist($employe1);

        // instantiation 2 de l'entité Employe et injection des données obligatoires
        $employe2 = new Employe();
        $employe2->setCode("222B222");
        $employe2->setNom("MACHIN");
        // notification de la modification de l'entité
        $em->persist($employe2);

        // instantiation 1 de l'entité Mission et injection des données obligatoires
        $mission1 = new Mission();
        $mission1->setGenre("P");
        $mission1->setNom("Mission 1");
        // notification de la modification de l'entité
        $em->persist($mission1);

        // instantiation 2 de l'entité Mission et injection des données obligatoires
        $mission2 = new Mission();
        $mission2->setGenre("T");
        $mission2->setNom("Mission 2");
        // notification de la modification de l'entité
        $em->persist($mission2);

        /* demande de modification de la base de données
           pour que les missions et employés soient mémorisées */
        $em->flush();

        /* instantiation 1 de l'entité Encadrer avec affectation de la mission 1,
           de l'employé 1 et d'un nombre d'heures */
        $encadrement1 = new Encadrement();

```

```

$encadrement1->setMission($mission1);
$encadrement1->setEmploye($employe1);
$encadrement1->setNbreHeures(140);
// notification de la modification de l'entité
$em->persist($encadrement1);

/* instantiation 2 de l'entité Encadrer avec affectation de la mission 1,
   de l'employé 2 et d'un nombre d'heures */
$encadrement2 = new Encadrement();
$encadrement2->setMission($mission1);
$encadrement2->setEmploye($employe2);
$encadrement2->setNbreHeures(170);
// notification de la modification de l'entité
$em->persist($encadrement2);

/* instantiation 3 de l'entité Encadrer avec affectation de la mission 2,
   de l'employé 1 et d'un nombre d'heures */
$encadrement3 = new Encadrement();
$encadrement3->setMission($mission2);
$encadrement3->setEmploye($employe1);
$encadrement3->setNbreHeures(155);
// notification de la modification de l'entité
$em->persist($encadrement3);

/* instantiation 4 de l'entité Encadrer avec affectation de la mission 2,
   de l'employé 2 et d'un nombre d'heures */
$encadrement4 = new Encadrement();
$encadrement4->setMission($mission2);
$encadrement4->setEmploye($employe2);
$encadrement4->setNbreHeures(160);
// notification de la modification de l'entité
$em->persist($encadrement4);

// demande de modification de la base de données
$em->flush();

return new Response("Insertions réussies !");
}
}

```

**Remarque :**

Une fois instanciées les deux employés et les deux missions, il faut les écrire dans la base de données afin qu'elles soient reconnues lors des instanciations relatives aux encadrements.

A présent que nous avons vu les différents cas de figure, nous allons nous intéresser au sens des relations et leur type : unidirectionnel et bidirectionnel.

En préliminaire, supprimer les 3 tables *encadrer*, *missions* et *employes* (après avoir fait une copie de cette dernière car elles contiennent des informations saisies dans les paragraphes précédents).

## h) relations unidirectionnelles et bidirectionnelles

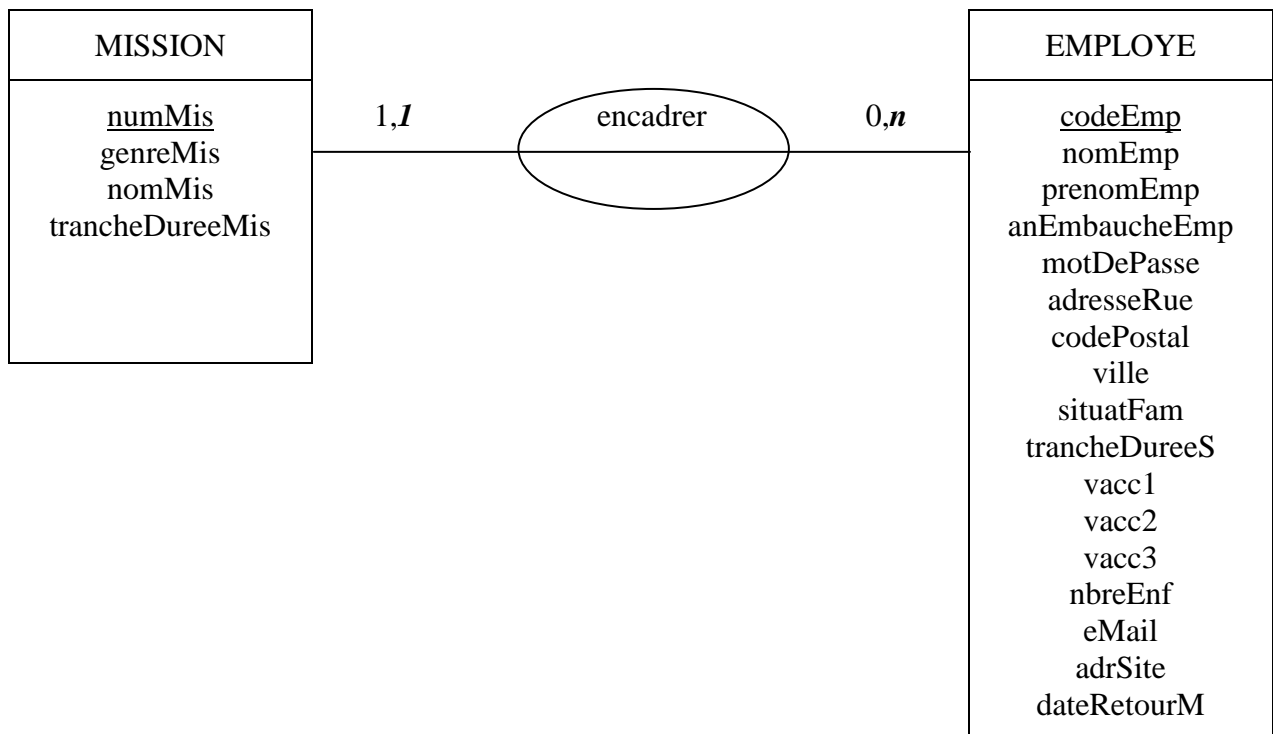
Pour l'instant, nous n'avons vu que des relations unidirectionnelles.

Voyons qu'est-ce qu'on entend par là.

### h-1) relation unidirectionnelle, et côté propriétaire / côté inverse

Considérons le cas du paragraphe e qui met en place la relation *Many-To-One*.

Le MCD est le suivant (cela se joue sur les cardinalités maximum) :



On y stipule qu'une mission n'est encadrée que par un seul employé et qu'un employé peut encadrer plusieurs missions.

Dit autrement, *plusieurs* missions sont encadrées par *un* employé.

Le modèle relationnel déduit est :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

On souhaite partir des missions, c'est pour cela qu'ensuite on a défini une relation *Many-To-One* de l'entité Mission vers l'entité Employe.

L'entité Mission donne (extrait) :

```
.....
private $trancheDureeMis;

/**
 * @ORM\ManyToOne(targetEntity="Employe")
 * @ORM\JoinColumn(name="codeEmp",
 *                  referencedColumnName="codeEmp",
 *                  nullable=false)
 */
private $employe;

// getter
public function getEmploye()
{
    return $this->employe;
}
.....
```

Mission est l'entité *propriétaire* et Employe est l'entité *inverse*.

Avec l'objet \$employe qui est de type (classe) Employe, on peut retrouver l'employé qui encadre une certaine mission via la méthode getEmploye().

Mais supposons qu'à présent, on veuille retrouver l'inverse : les missions qu'un certain employé encadre.

Il va falloir que notre relation passe du type unidirectionnel au type bidirectionnel.

## h-2) mise en place d'une relation bidirectionnelle

On va agir dans les deux entités `Mission` et `Employe`.

Le code de l'entité `Mission` devient (extrait) :

```
.....
private $trancheDureeMis;

/**
 * @ORM\ManyToOne(targetEntity="Employe",
 *                 inversedBy="missions")
 * @ORM\JoinColumn(name="codeEmp",
 *                 referencedColumnName="codeEmp",
 *                 nullable=false)
 */
private $employe;

// getter
public function getEmploye()
{
    return $this->employe;
}
.....
```

### Commentaires :

Dans l'entité propriétaire, ici `Mission`, on stipule qu'une relation inverse doit exister via l'option `inversedBy` de l'annotation `@ORM\ManyToOne`, en indiquant le nom de la propriété qui sera créée dans l'entité `Employe` par exemple `missions` (au pluriel car c'est un tableau).

Le code de l'entité `Employe` devient (extrait) :

```
.....
private $dateRetourM;

/**
 * @ORM\OneToMany(targetEntity="Mission",
 *                 mappedBy="employe")
 */
private $missions;
```

```
// getter
public function getMissions()
{
    return $this->missions;
}
```

### Commentaires :

Dans l'entité inverse, ici *Employe*, on stipule une relation inverse de *Many-To-One* soit donc *One-To-Many*.

Dans l'annotation `@ORM\OneToMany`, on indique l'entité propriétaire cible, ici *Mission*, (option `targetEntity`) et on stipule via l'option `mappedBy` le nom de la propriété dans l'entité propriétaire qui assure la liaison avec cette entité inverse, soit ici *employe*.

Ensuite, on rajoute un getter qui renvoie les missions de l'employé sous forme d'un tableau.

Enfin, on lance les commandes pour la mise à jour de la base de données :

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

Nous avons donc vu comment mettre en place les divers types de relations avec donc la gestion des clés étrangères.

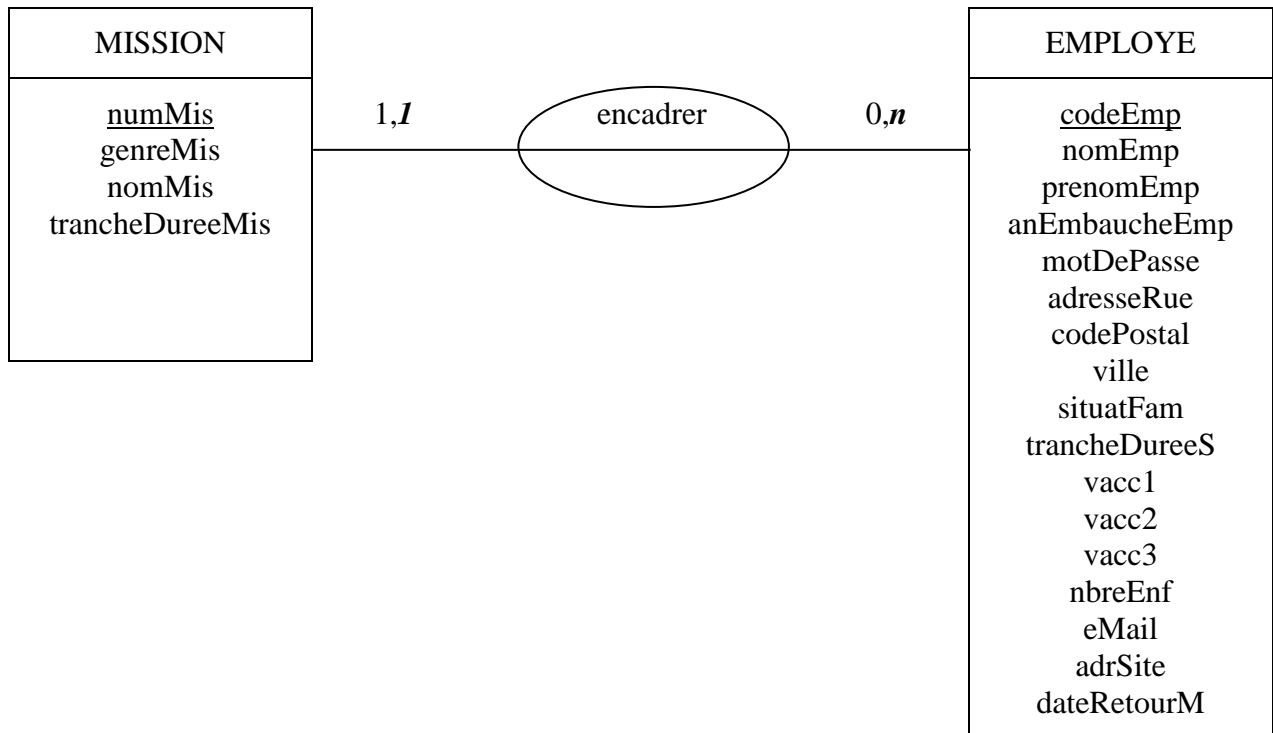
A présent, voyons comment implémenter l'utilisation de ces clés étrangères à travers l'interface utilisateur via donc un formulaire.



i) clés étrangères et prise en compte dans l'interface utilisateuri-1) cas traité et entité correspondante

Nous allons prendre le cas le plus courant, les autres pouvant se déduire.

Le modèle conceptuel des données sera :



On y stipule qu'une mission n'est encadrée que par un seul employé et qu'un employé peut encadrer plusieurs missions.

Dit autrement, *plusieurs* missions sont encadrées par *un* employé.

Revoir éventuellement le paragraphe e pour la mise en place du cas de ce MCD.

Le modèle relationnel déduit est le suivant :

EMPLOYE (codeEmp, nomEmp, prenomEmp, anEmbaucheEmp, motDePasse, adresseRue, codePostal, ville, situatFam, trancheDureeS, vacc1, vacc2, vacc3, nbreEnf, eMail, adrSite, dateRetourM)

MISSION (numMis, genreMis, nomMis, trancheDureeMis, #codeEmp)

Le code de l'entité Mission donne (la mise en œuvre de la relation avec l'entité Employe est en gras) :

```

<?php

// src/Entity/Mission.php

namespace App\Entity;

// définition des alias
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 * @ORM\Table(name="missions")
 */
class Mission
{

    // attributs privés
    /**
     * @ORM\Column(name="numMis", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $num;

    /**
     * @Assert\NotBlank(message="Le genre de mission doit être
                                renseigné.")
     * @ORM\Column(name="genreMis", type="string", length=15)
     */
    private $genre;

    /**
     * @Assert\NotBlank(message="Le nom mission doit être
                                renseigné")
     * @ORM\Column(name="nomMis", type="string", length=25)
     */
    private $nom;

    /**
     * @ORM\Column(name="trancheDureeMis", type="smallint",
                                nullable=true)
     */
    private $trancheDureeMis;

    /**
     * @ORM\ManyToOne(targetEntity="Employe")
     * @ORM\JoinColumn(name="codeEmp",
                                referencedColumnName="codeEmp",

```

```

        nullable=false)

    */
    private $employe;

    // getters et setters
    public function getNum()
    {
        return $this->num;
    }

    public function getGenre()
    {
        return $this->genre;
    }
    public function setGenre($genre)
    {
        $this->genre = $genre;
    }

    public function getNom()
    {
        return $this->nom;
    }
    public function setNom($nom)
    {
        $this->nom = $nom;
    }

    public function getTrancheDureeMis()
    {
        return $this->trancheDureeMis;
    }
    public function setTrancheDureeMis($trancheDureeMis)
    {
        $this->trancheDureeMis = $trancheDureeMis;
    }

    public function getEmploye()
    {
        return $this->employe;
    }
    public function setEmploye(Employe $employe)
    {
        $this->employe = $employe;
    }

}

```

A présent, il faut lancer les commandes pour la mise à jour de la base de données :

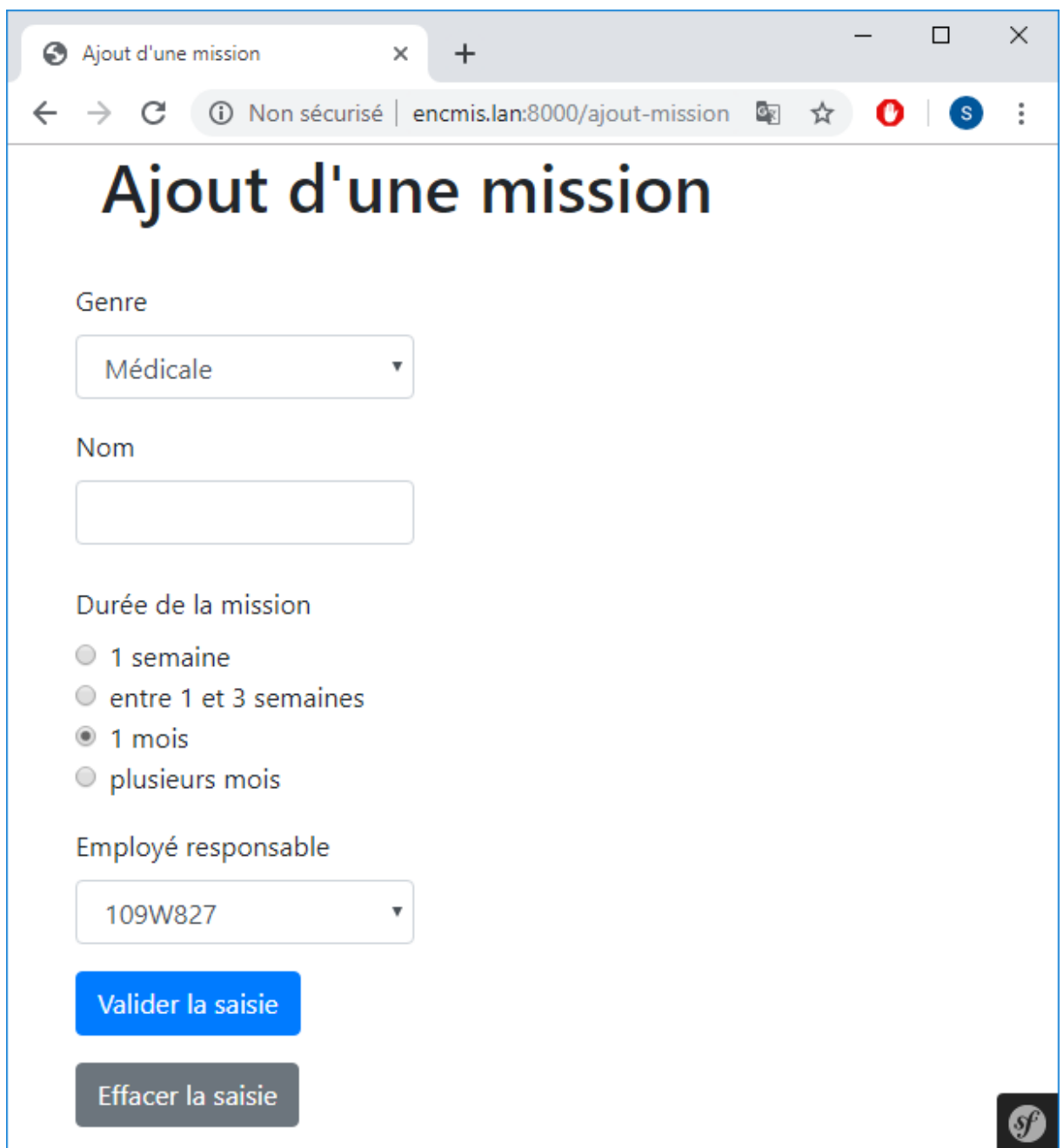
**php bin/console make:migration**

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

## i-2) formulaire à mettre en place

Voyons le formulaire à mettre en place pour l'ajout d'une mission (pour la modification et la suppression, la programmation sera similaire comme on a pu le voir lors de la gestion des employés).

Cela donne :



The screenshot shows a web browser window with the title 'Ajout d'une mission'. The address bar shows 'Non sécurisé | encmis.lan:8000/ajout-mission'. The form contains the following fields and controls:

- Genre**: A dropdown menu with 'Médicale' selected.
- Nom**: A text input field.
- Durée de la mission**: A group of radio buttons with options: '1 semaine', 'entre 1 et 3 semaines', '1 mois' (selected), and 'plusieurs mois'.
- Employé responsable**: A dropdown menu with '109W827' selected.
- Buttons**: Two buttons at the bottom, 'Valider la saisie' (blue) and 'Effacer la saisie' (grey).
- Logo**: A small logo in the bottom right corner.

La nouveauté est ici la présence d'une liste déroulante de valeurs provenant d'une autre table : ici le code employé de la table *employes*.

Nous verrons au paragraphe i-5 comment mettre en plus le nom de l'employé car juste le code de l'employé ce n'est pas très parlant...

### i-3) choix de valeur de clé étrangère : widget de type `EntityType`

Pour mettre en place une liste déroulante de valeurs provenant d'une autre table donc au départ d'une autre entité, il faut ajouter un widget de type `EntityType::class` en indiquant :

- la classe correspondant à l'entité de cette autre table via l'option `class`,
- la propriété (attribut) de cette autre entité via l'option `choice_label`.

Le code va ici donner :

```
->add('employe', EntityType::class,
    ['label' => 'Employé responsable',
     'class' => Employe::class,
     'choice_label' => 'code'])
```

Le type `EntityType` doit être importé.

Cela donne :

```
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
```

Attention !

On remarquera que ce type `EntityType` n'est pas au même endroit que les autres types comme `TextType` : on retrouve notamment l'élément `Doctrine`.

Lors du clic sur le bouton *Submit*, Symfony saura constituer l'objet `$employe` de l'entité *Mission* concerné et corrélativement Doctrine saura quelle valeur de *codeEmp*, clé étrangère, mettre dans la table *missions*.

#### Rappel :

Ici, il semble qu'on mette l'attribut `code` directement pourtant il est privé !

Qu'en est il réellement ?

Comme on a créé le getter `getCode`, Symfony génère automatiquement en interne la propriété `code`.

La valeur de l'option `choice_label` est donc ici cette propriété `code`.

Si vous enlevez le getter `getCode`, Symfony met une erreur en précisant bien qu'il ne trouve pas de getter et que donc la propriété `code` invoquée n'existe pas.

A présent, voyons comment mettre en place le code complet notamment de génération de ce formulaire d'ajout d'une mission.

i-4) code complet\* contrôleur

Le nouveau contrôleur donne :

```
<?php

// src/Controller/MissionController.php

namespace App\Controller;

use Doctrine\Persistence\ManagerRegistry;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ResetType;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;

// importation des entités
use App\Entity\Employe;
use App\Entity\Mission;

class MissionController extends AbstractController
{
    /**
     * @Route("/ajout-mission", name="ajout_mission")
     */
    public function ajout(ManagerRegistry $doctrine,
                        Request $request)
    {

        // instantiation de l'entité Mission
        $mission = new Mission();

        // création du constructeur de formulaire en fournissant l'entité
        $formBuilder = $this->createFormBuilder($mission);
```

```

/* ajout successif des propriétés souhaitées de l'entité
   pour les champs de formulaire avec leur type */
$formBuilder
->add('genre', ChoiceType::class,
    ['choices' => ['Médicale' => 'Médicale',
                    'Prospection' => 'Prospection',
                    'Technique' => 'Technique']])
->add('nom', TextType::class)
->add('trancheDureeMis', ChoiceType::class,
    ['expanded' => true,
      'label' => 'Durée de la mission',
      'choices' => ['1 semaine' => 1,
                    'entre 1 et 3 semaines' => 2,
                    '1 mois' => 3,
                    'plusieurs mois' => 4],
      'data' => 3])
->add('employe', EntityType::class,
    ['label' => 'Employé responsable',
      'class' => Employe::class,
      'choice_label' => 'code'])
->add('validation', SubmitType::class,
    ['label' => 'Valider la saisie'])
->add('effacement', ResetType::class,
    ['label' => 'Effacer la saisie']);

// récupération du formulaire à partir du constructeur de formulaire
$form = $formBuilder->getForm();

/* traitement de la requête : Symfony récupère éventuellement les valeurs des
   champs de formulaire et alimente l'objet $mission */
$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid())
    // le formulaire a été soumis et il est valide
    {
        // écriture dans la base de données
        $em = $doctrine->getManager();
        $em->persist($mission);
        $em->flush();

        return new Response("Mission ajoutée dans la base...");
    }

// passage du formulaire à la vue pour affichage
return $this->render('mission/formMission.html.twig',
    ['form' => $form->createView()]);
}
}

```



### \* template Twig

Le nouveau template Twig donne :

```
{# templates/mission/formMission.html.twig #}

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

{% block title %}
    Ajout d'une mission
{% endblock %}

{% block body %}

    <div class="container">

        <h1>Ajout d'une mission</h1>
        <br />

        <div class="row">

            {{ form(form) }}

        </div>

    </div>

{% endblock %}
```

i-5) optimisation : proposition d'informations supplémentaires relatives à la clé étrangère

Plutôt de ne proposer que le code de l'employé qui n'est pas très parlant, on pourrait mettre le code et le nom de l'employé dans la liste déroulante ainsi :

The screenshot shows a web browser window with the title 'Ajout d'une mission'. The address bar shows 'Non sécurisé | encmis.lan:8000/ajout-mission'. The main heading is 'Ajout d'une mission'. Below the heading, there are several form fields:

- Genre:** A dropdown menu with 'Médicale' selected.
- Nom:** A text input field.
- Durée de la mission:** A group of radio buttons with options: '1 semaine', 'entre 1 et 3 semaines', '1 mois' (selected), and 'plusieurs mois'.
- Employé responsable:** A dropdown menu with '109W827 - HATAN' selected.

At the bottom of the form, there are two buttons: 'Valider la saisie' (blue) and 'Effacer la saisie' (grey). A small 'sf' logo is visible in the bottom right corner of the browser window.

Dans l'entité `Employe`, on rajoute dans un premier temps un getter qui va renvoyer un message avec le code et le nom de l'employé.

Son code donne :

```
public function getCodeEtNom()
{
    return $this->code . " - " . $this->nom;
}
```

A présent, on va invoquer la propriété qui est générée par Symfony à partir de ce getter. Le nom du getter est `getCodeEtNom` : la propriété s'appelle donc `codeEtNom`.

L'instruction correspondante devient dans le contrôleur (la modification est en gras) :

```
->add('employe', EntityType::class,
    ['label' => 'Employé responsable',
     'class' => Employe::class,
     'choice_label' => 'codeEtNom'])
```

## 11) Repository Doctrine, et requêtes d'interrogation via DQL et QueryBuilder

### a) rappel et problématique

Le repository centralise l'endroit géré par Doctrine où sont récupérées les données.

Nous avons déjà utilisé le repository pour récupérer les informations via deux méthodes :

- `findAll()` pour récupérer toutes les entités pour une table donnée mappée par une Entity.

Voici un extrait de code utilisant cette méthode `findAll()`.

```
// récupération du repository relatif à l'entité (classe) Employe
$repository = $doctrine->getRepository(Employe::class);
// recherche de tous les employés
$listeEmployes = $repository->findAll();
```

- `find()` pour récupérer une seule entité via un id.

Voici un extrait de code utilisant cette méthode `find()`.

```
// récupération du repository relatif à l'entité (classe) Employe
$repository = $doctrine->getRepository(Employe::class);
// recherche de l'employé
$monEmploye = $repository->find("284C214");
```

Doctrine est ainsi une ***couche d'abstraction*** logicielle au dessus du SQL : on travaille sur des objets Doctrine et derrière Doctrine lance des requêtes SQL sur la base.

Les méthodes de recherche `findAll()`, `find()` ou autres variantes sont assez figées.

Si on veut faire des requêtes personnalisées, par exemple extraire les employés par ordre alphabétique embauchés depuis 1995 et donc le nom commence par A, deux outils existent :

- le DQL (*Doctrine Query Language*) qui comme on s'en doute va être proche du SQL
- le QueryBuilder qui va ressembler par exemple au constructeur de formulaire (FormBuilder)

QueryBuilder est plus récent que DQL.

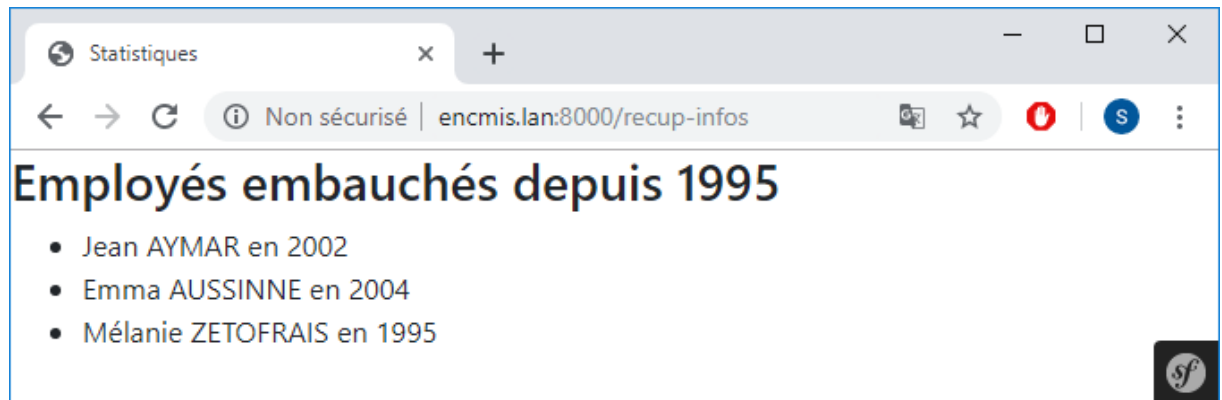
Dans la suite, on se basera sur le contenu suivant des deux tables *employes* (extrait de champs) et *missions*.

codeEmp	nomEmp	prenomEmp	anEmbaucheEmp
109W827	HATAN	Charles	1992
110A225	AYMAR	Jean	2002
284B128	COVER	Harry	1994
284C214	AUSSINNE	Emma	2004
284C226	ZETOFRAIS	Mélanie	1995

numMis	genreMis	nomMis	trancheDureMis	codeEmp
1	Prospection	Orchidée des sables	4	284C214
2	Médicale	Santé 2015	3	284C226
3	Prospection	Nénuphar des bois	2	109W827

b) DQLb-1) un premier exemple complet

On souhaite afficher une liste à puces des employés embauchés depuis 1995 ainsi :

\* contrôleur avec méthode et routage

Le nouveau contrôleur donne :

```
<?php

// src/Controller/RecupInfosController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class RecupInfosController extends AbstractController
{
    /**
     * @Route("/recup-infos")
     */
}
```

```

public function index(ManagerRegistry $doctrine)
{
    // récupération de l'Entity Manager
    $em = $doctrine->getManager();

    // création de la requête
    $query = $em->createQuery
        ("SELECT e
         FROM App\Entity\Employe e
         WHERE e.anEmbauche >= 1995");

    // récupération du résultat
    $employees = $query->getResult();

    // passage du résultat à la vue pour affichage
    return $this->render('stats/recupInfos.html.twig',
        ['employees' => $employees]);
}
}

```

### Commentaires :

La création d'une requête se fait via la méthode `createQuery()`.

Elle admet en paramètre une requête DQL : la syntaxe est très proche du SQL, sauf que dans le `SELECT` on met non pas des champs mais un alias de l'entité `Employe` nommé ici arbitrairement `e`.

D'autre part, on invoque des noms d'attribut de l'entité concernée, et non pas des noms de champs de la table.

La récupération du résultat de la requête se fait via la méthode `getResult()`.

### \* template Twig

Le nouveau template Twig donne :

```

{# templates/stats/recupInfos.html.twig #}

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

{% block title %}
    Statistiques
{% endblock %}

```

```
{% block body %}
    <h3>Employés embauchés depuis 1995</h3>
    <ul>
        {% for employe in employes %}
            <li>
                {{ employe.prenom }} {{ employe.nom }} en
                {{ employe.anEmbauche }}
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

Rappelons que l'accès à un champ de tableau se fait avec le point (notation pointée), par exemple `employe.nom`.

A présent, voyons les éléments permettant de réaliser les requêtes DQL sur une entité.

## b-2) mots-clés SQL : les mêmes en DQL

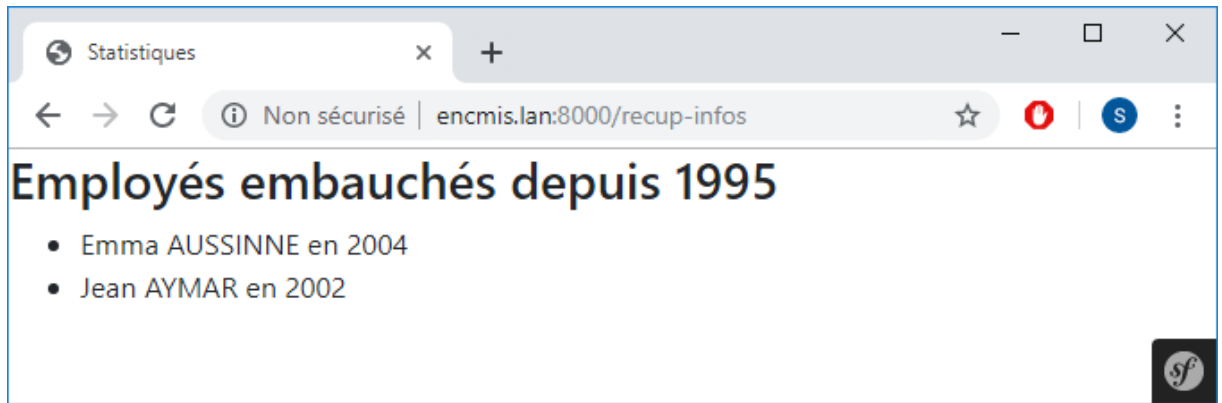
Les mots-clés du SQL sont reconnus en DQL comme SELECT, FROM, WHERE, AND, OR, BETWEEN, IN, LIKE, EXISTS, ORDER BY.

Par exemple, si on veut afficher les employés embauchés depuis 1995 donc le nom commence par A et ce par ordre alphabétique, la création de la requête donne dans le contrôleur :

```
$query = $em->createQuery
    ("SELECT e
     FROM App\Entity\Employe e
     WHERE e.anEmbauche >= 1995
     AND e.nom LIKE 'A%'
     ORDER BY e.nom");
```

Voici la page affichée (on laissera le même intitulé) :





### b-3) paramètre de requête

Considérons la page précédente.

L'année à partir de laquelle on filtre est 1995.

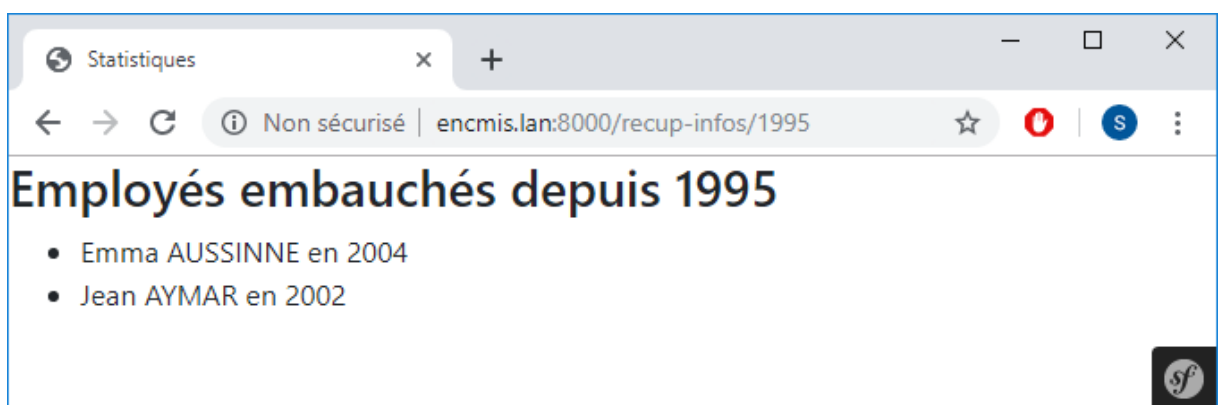
Si à présent, on veut afficher ces employés depuis une certaine année à définir, il va y avoir un paramètre pour la méthode du contrôleur.

On pourrait saisir la valeur depuis un formulaire : on va le faire en dur dans l'URL pour simplifier.

Cela va donner pour l'exemple de l'année 1995 :

**localhost:8000/recup-infos/1995**

On va obtenir :



Le contrôleur devient (les modifications sont en gras) :

```

<?php

// src/Controller/RecupInfosController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class RecupInfosController extends AbstractController
{
    /**
     * @Route("/recup-infos/{annee}")
     */

    public function index(ManagerRegistry $doctrine, $annee)
    {
        // récupération de l'Entity Manager
        $em = $doctrine->getManager();

        // création de la requête
        $query = $em->createQuery
            ("SELECT e
             FROM App\Entity\Employe e
             WHERE e.anEmbauche >= :annee
             AND e.nom LIKE 'A%'
             ORDER BY e.nom");

        // affectation d'une valeur au paramètre
        $query->setParameter('annee', $annee);

        // récupération du résultat
        $employees = $query->getResult();

        // passage du résultat à la vue pour affichage
        return $this->render('stats/recupInfos.html.twig',
            ['employees' => $employees,
             'annee' => $annee]);
    }
}

```

Commentaires :

On définit un paramètre dans la requête via le préfixe deux-points (:).

La méthode `setParameter()` permet d'affecter une valeur effective au paramètre.

On passe ensuite à la vue (template) en plus la valeur du paramètre pour affichage.

Le code du template Twig devient (la modification est en gras) :

```
{# templates/stats/recupInfos.html.twig #}

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

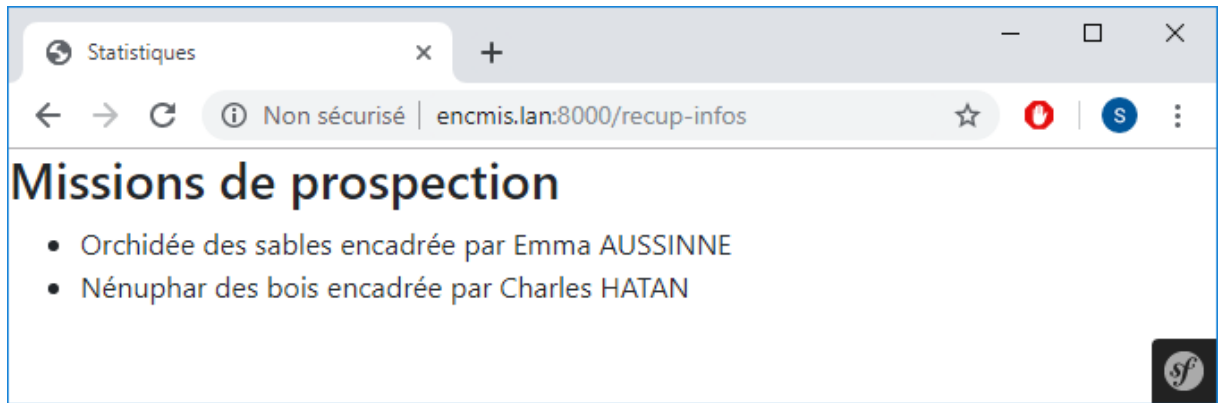
{% block title %}
    Statistiques
{% endblock %}

{% block body %}
    <h3>Employés embauchés depuis {{ annee }}</h3>
    <ul>
        {% for employe in employes %}
            <li>
                {{ employe.prenom }} {{ employe.nom }} en
                {{ employe.anEmbauche }}
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

**b-4) les jointures**

Une jointure DQL se fait via le mot-clé JOIN.

On voudrait afficher les missions de prospection (genreMis égal à Prospection) avec le nom et le prénom de l'employé encadrant ainsi :



Ici les deux entités Mission et Employé sont impliquées.

\* contrôleur

Le nouveau code donne (les éléments sur la jointure sont en gras) :

```
<?php
```

```
// src/Controller/RecupInfosController.php
```

```
namespace App\Controller;
```

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

```
// importation de l'entité Mission
```

```
use App\Entity\Mission;
```

```
class RecupInfosController extends AbstractController
{
```

```
    /**
     * @Route("/recup-infos")
     */
```

```
    public function index(ManagerRegistry $doctrine)
    {
```

```
        // récupération de l'Entity Manager
        $em = $doctrine->getManager();
```

```

// création de la requête
$query = $em->createQuery
    ("SELECT m, e
     FROM App\Entity\Mission m
     JOIN m.employe e
     WHERE m.genre = 'Prospection'");

// récupération du résultat
$missions = $query->getResult();

// passage du résultat à la vue pour affichage
return $this->render('stats/recupInfos.html.twig',
    ['missions' => $missions]);
}

}

```

### Commentaires :

Comme on part cette fois de l'entité `Mission`, on récupère son repository correspondant.

Dans la requête, on n'indique que la première entité `Mission`.

Le fait ensuite de mettre `m.employe` indique à Symfony de chercher dans l'entité (classe) `Mission` la variable `$employe` qui est stipulée comme étant un objet de l'entité (classe) `Employe`.

### \* template Twig

Le nouveau code donne :

```

{# templates/stats/recupInfos.html.twig #}

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

{% block title %}
    Statistiques
{% endblock %}

{% block body %}
    <h3>Missions de prospection</h3>
    <ul>
        {% for mission in missions %}
            <li>

```

```
        {{ mission.nom }}    encadrée par
        {{ mission.employe.prenom }} {{ mission.employe.nom }}
    </li>
    {% endfor %}
</ul>
{% endblock %}
```

### Commentaires :

On a une double notation pointée pour retrouver le prénom de l'employé de la mission.  
Pareil pour le nom de l'employé de la mission.

### b-5) intérêts du DQL

Il y a 2 intérêts majeurs de l'utilisation du DQL par rapport au SQL :

- niveau d'abstraction supérieur : on utilise des noms d'entités et d'attributs, et non pas des noms de tables et de champs,
- concision du code dans le cadre des jointures car on n'a pas besoin d'indiquer les colonnes impliquées dans une jointure : Doctrine les connaît via les entités et leurs relations définies dans chacune des `Entity`.

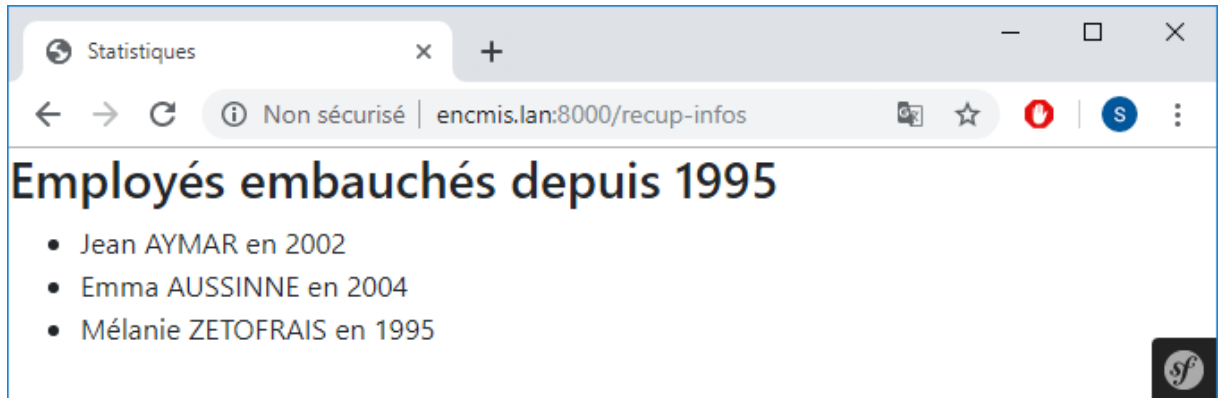
Voyons à présent un outil alternatif à DQL plus récent : Query Builder.

### c) Query Builder

Nous reprendrons les mêmes exemples que pour le DQL.

#### c-1) un premier exemple complet

On souhaite afficher une liste à puces des employés embauchés depuis 1995 ainsi :



#### \* contrôleur avec méthode et routage

Le nouveau contrôleur donne (les lignes spécifiques au QueryBuilder sont en gras) :

```
<?php

// src/Controller/RecupInfosController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class RecupInfosController extends AbstractController
{
    /**
     * @Route("/recup-infos")
```

```

* /

public function index(ManagerRegistry $doctrine)
{

    // récupération du repository relatif à l'entité (classe) Employe
    $repository = $doctrine->getRepository(Employe::class);

    // mise en place du constructeur de la requête
    $queryBuilder = $repository->createQueryBuilder("e")
        ->where("e.anEmbauche >= 1995");

    // récupération de la requête
    $query = $queryBuilder->getQuery();

    // récupération du résultat
    $employees = $query->getResult();

    // passage du résultat à la vue pour affichage
    return $this->render('stats/recupInfos.html.twig',
        ['employees' => $employees]);

}

}

```

### Commentaires :

On récupère au préalable le repository de l'entité Employe.

Pour mettre en place le constructeur de requête, on invoque la méthode `createQueryBuilder()` en fournissant comme paramètre un alias de l'entité Employe nommé ici arbitrairement `e`.

A la suite, on invoque la méthode `where()` pour effectuer une sélection.

Ensuite, on invoque la méthode `getQuery()` pour générer la requête.



### \* template Twig

Le template Twig donne toujours :

```
{# templates/stats/recupInfos.html.twig #}

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

{% block title %}
    Statistiques
{% endblock %}

{% block body %}
    <h3>Employés embauchés depuis 1995</h3>
    <ul>
        {% for employe in employes %}
            <li>
                {{ employe.prenom }} {{ employe.nom }} en
                {{ employe.anEmbauche }}
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

A présent, voyons les éléments permettant de réaliser les requêtes sur une entité via QueryBuilder.

### c-2) mots-clés SQL : des méthodes QueryBuilder

Les mots-clés du SQL comme WHERE, AND, OR ou ORDER BY donnent lieu à des méthodes QueryBuilder.

Par exemple, on retrouve la méthode `where()` correspondant à la clause WHERE.

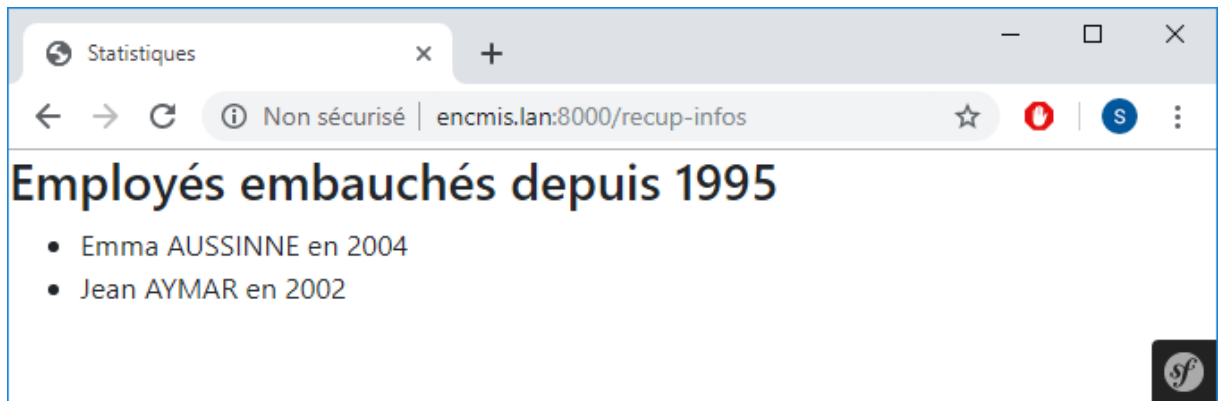
A noter qu'on ne traduit pas le SELECT FROM car on fournit simplement l'alias à la méthode `createQueryBuilder()`.

Des exceptions cependant : pour les mots clé SQL AND et OR, les méthodes correspondantes sont respectivement `andWhere()` et `orWhere()`.

Par exemple, si on veut afficher les employés embauchés depuis 1995 donc le nom commence par A et ce par ordre alphabétique, la création du constructeur de requête donne dans le contrôleur :

```
$queryBuilder = $repository->createQueryBuilder("e")
    ->where("e.anEmbauche >= 1995")
    ->andWhere("e.nom LIKE 'A%'")
    ->orderBy("e.nom");
```

Voici la page affichée :



### c-3) paramètre de requête

Considérons la page précédente.

L'année à partir de laquelle on filtre est 1995.

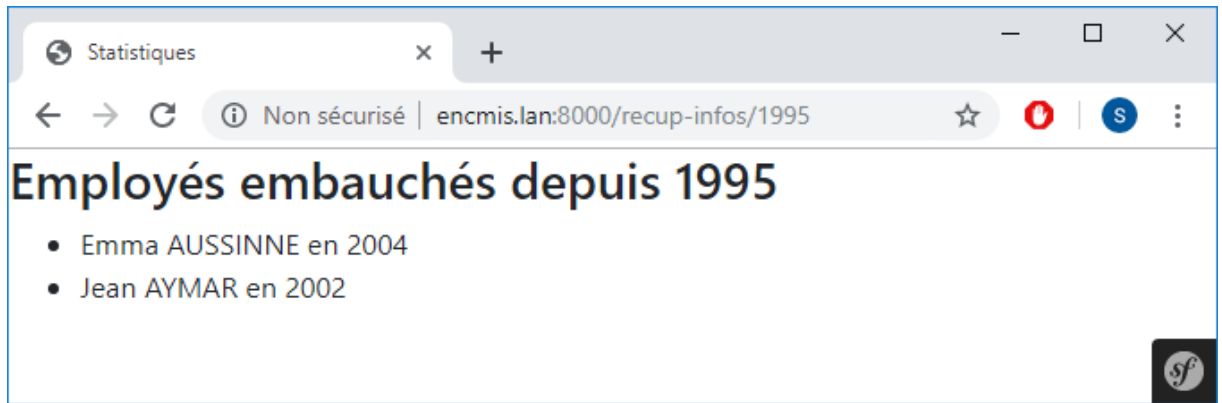
Si à présent, on veut afficher ces employés depuis une certaine année à définir, il va y avoir un paramètre pour la méthode du contrôleur.

On pourrait saisir la valeur depuis un formulaire : on va le faire en dur dans l'URL pour simplifier.

Cela va donner pour l'exemple de l'année 1995 :

**localhost:8000/recup-infos/1995**

On va obtenir :



Le contrôleur devient (les modifications sont en gras) :

```
<?php

// src/Controller/RecupInfosController.php

namespace App\Controller;

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

// importation de l'entité
use App\Entity\Employe;

class RecupInfosController extends AbstractController
{
    /**
     * @Route("/recup-infos/{annee}")
     */

    public function index(ManagerRegistry $doctrine, $annee)
    {

        // récupération du repository relatif à l'entité (classe) Employe
        $repository = $doctrine->getRepository(Employe::class);

        // mise en place du constructeur de la requête
        $queryBuilder = $repository->createQueryBuilder("e")
            ->where("e.anEmbauche >= :annee")
            ->setParameter('annee', $annee)
            ->andWhere("e.nom LIKE 'A%'");
```

```

->orderBy("e.nom");

// récupération de la requête
$query = $queryBuilder->getQuery();

// récupération du résultat
$employees = $query->getResult();

// passage du résultat à la vue pour affichage
return $this->render('stats/recupInfos.html.twig',
    ['employees' => $employees,
     'annee' => $annee]);
}
}

```

### Commentaires :

On définit un paramètre dans la requête en préfixant par un deux-points (:).

La méthode `setParameter()` permet d'affecter une valeur effective au paramètre : elle est invoquée juste après avoir mentionné ce paramètre.

On passe ensuite à la vue (template) en plus la valeur du paramètre pour affichage.

Le code du template Twig devient (la modification est en gras) :

```

{# templates/stats/recupInfos.html.twig #}

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

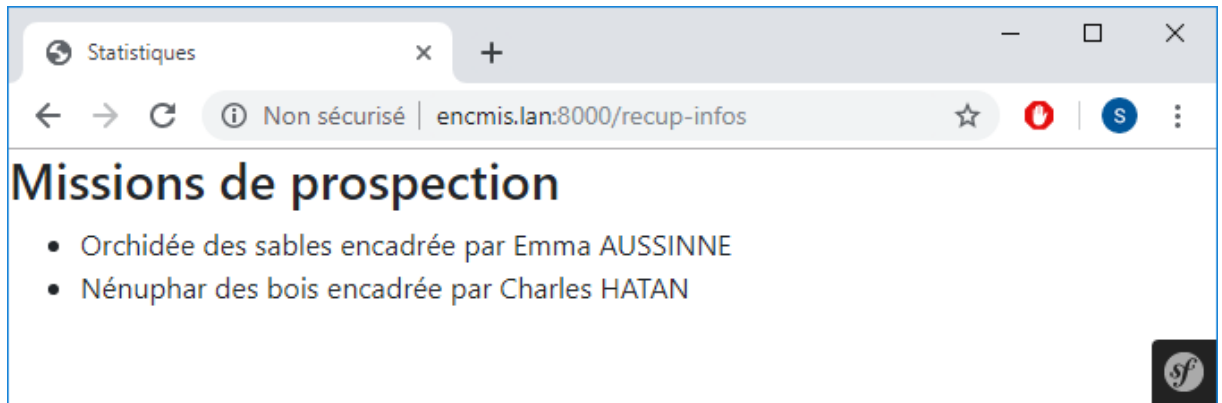
{% block title %}
    Statistiques
{% endblock %}

{% block body %}
    <h3>Employés embauchés depuis {{ annee }}</h3>
    <ul>
        {% for employe in employees %}
            <li>
                {{ employe.prenom }} {{ employe.nom }} en
                {{ employe.anEmbauche }}
            </li>
        {% endfor %}
    </ul>
{% endblock %}

```

#### c-4) les jointures

On voudrait afficher les missions de prospection (genreMis égal à Prospection) avec le nom et le prénom de l'employé encadrant ainsi :



Ici les deux entités Mission et Employe sont impliquées.

#### \* contrôleur

Le nouveau code donne (les éléments sur la jointure sont en gras) :

```
<?php
```

```
// src/Controller/RecupInfosController.php
```

```
namespace App\Controller;
```

```
use Doctrine\Persistence\ManagerRegistry;
```

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

```
use Symfony\Component\Routing\Annotation\Route;
```

```
// importation de l'entité Mission
```

```
use App\Entity\Mission;
```

```
class RecupInfosController extends AbstractController
{
```

```
    /**
```

```
     * @Route("/recup-infos")
```

```

*/

public function index(ManagerRegistry $doctrine)
{
    // récupération du repository relatif à l'entité (classe) Mission
    $repository = $doctrine->getRepository(Mission::class);

    // création du constructeur de la requête
    $queryBuilder = $repository->createQueryBuilder("m")
        ->join("m.employe", "e")
        ->where("m.genre = 'Prospection'");

    // récupération de la requête
    $query = $queryBuilder->getQuery();

    // récupération du résultat
    $missions = $query->getResult();

    // passage du résultat à la vue pour affichage
    return $this->render('stats/recupInfos.html.twig',
        ['missions' => $missions]);
}
}

```

### Commentaires :

Comme on part cette fois de l'entité `Mission`, on récupère son repository correspondant.

La jointure se fait via la méthode `join()`.

Le fait ensuite de mettre `m.employe` indique à Symfony de chercher dans l'entité (classe) `Mission` la variable `$employe` qui est stipulée comme étant un objet de l'entité (classe) `Employe`.

### \* template Twig

Le nouveau code donne :

```

{# templates/stats/recupInfos.html.twig #}

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

```

```
{% block title %}
    Statistiques
{% endblock %}

{% block body %}
    <h3>Missions de prospection</h3>
    <ul>
        {% for mission in missions %}
            <li>
                {{ mission.nom }} encadrée par
                {{ mission.employe.prenom }} {{ mission.employe.nom }}
            </li>
        {% endfor %}
    </ul>
{% endblock %}
```

#### Commentaires :

On a une double notation pointée pour retrouver le prénom de l'employé de la mission.  
Pareil pour le nom de l'employé de la mission.

### c-5) intérêts de QueryBuilder

Il y a 3 intérêts majeurs de l'utilisation de QueryBuilder par rapport au DQL :

- le code est encore plus concis car on s'affranchit de l'équivalent du `SELECT FROM`,
- on est complètement dans le paradigme objet : on ne fait qu'appeler des méthodes correspondant aux clauses SQL,
- on est plus souple dans la rédaction des requêtes car on peut à tout moment invoquer des méthodes sur le constructeur de requête pour personnaliser le résultat.