NOVAO®
LEARNING

SYMFONY

Sommaire

- Introduction
- Installation
- Architecture
- Création d’un projet
- Controller
- Routing
- Template twig
- Doctrine
- Formulaire
- Fixtures
- Créations d’utilisateur
- Authentification
- Registration
- Mailer

Introduction

INTRODUCTION

- Symfony est un **framework open-source** destiné pour les applications web PHP
- Un ensemble d'outils élégants permettant la création d'applications web plus rapidement
- Sponsorisé par **SensioLabs** et développé par **Fabian POTENCIER** en 2005

INTRODUCTION

- Un framework est une collection de classes qui aident le développeur à développer une application web
- Symfony est un framework **full-stack** il contient un ensemble de composants (classes) PHP réutilisables
- Symfony fonctionne avec une configuration flexible en utilisant **YAML**, **XML** ou les **annotations**
- Les composants de Symfony sont très utilisés par de nombreux projets qui incluent **Composer**, **Drupal**, **PHPBB**, ..

INTRODUCTION

Symfony est conçu pour optimiser le développement d'applications web. De nouvelles caractéristiques sont ajoutées à chaque nouvelle release de Symfony

- Le système Model-View-Controller
- Framework PHP performant
- Un routage flexible des URI
- Un code réutilisable et maintenable
- Gestion de la session
- Logging des erreurs
- Des classes de base de données complètes avec un support sur différentes plateformes
- Une communauté très active
- Un ensemble de composants découplés et réutilisables
- Assure une standardisation et une inter-opérabilité d'applications
- Sécurise l'application contre différentes attaques
- Le moteur de templating **Twig**

Installation

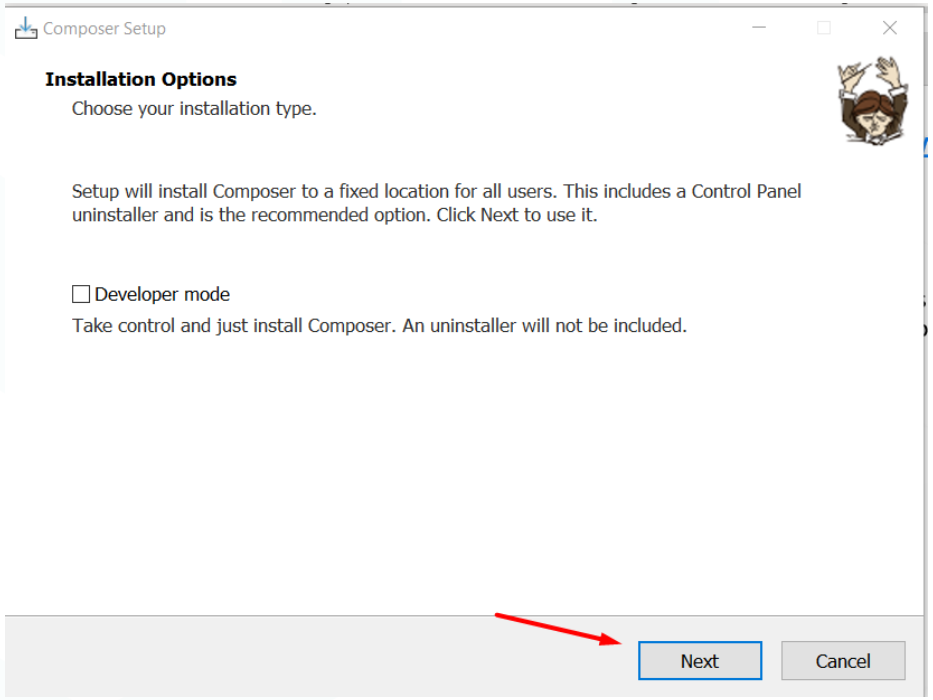
COMPOSER - INSTALLATION

Présentation de Composer :

- **Composer** : simplifie la gestion des dépendances dans les projets PHP en automatisant le processus d'installation, de mise à jour et de chargement des bibliothèques tierces. Composer est donc un gestionnaire de dépendances PHP.

- Installation :

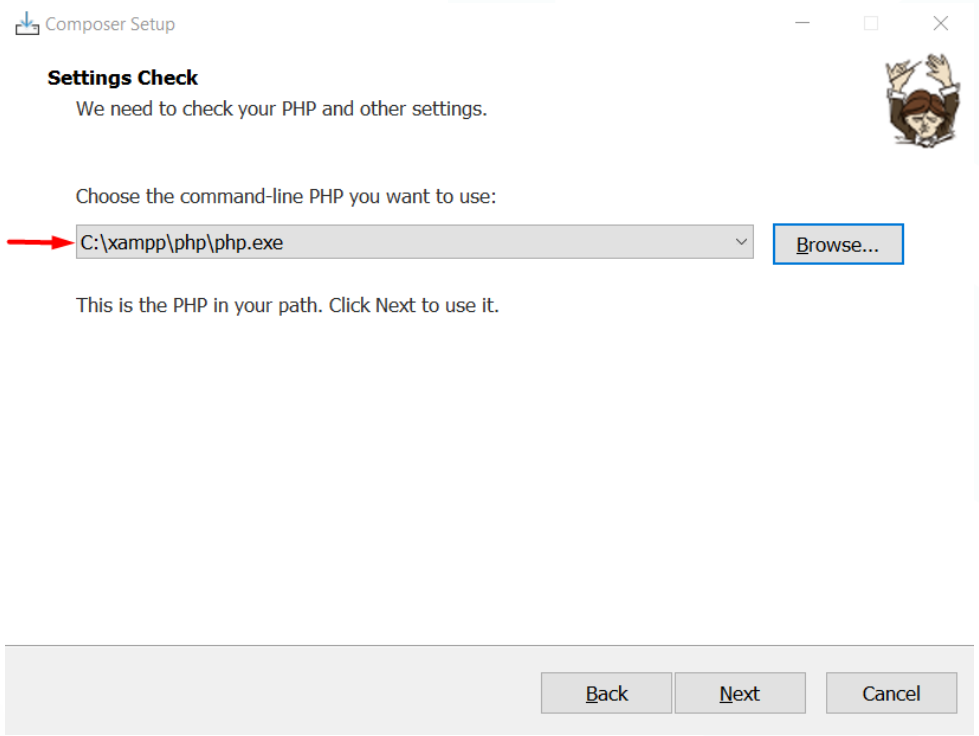
  https://getcomposer.org/download/

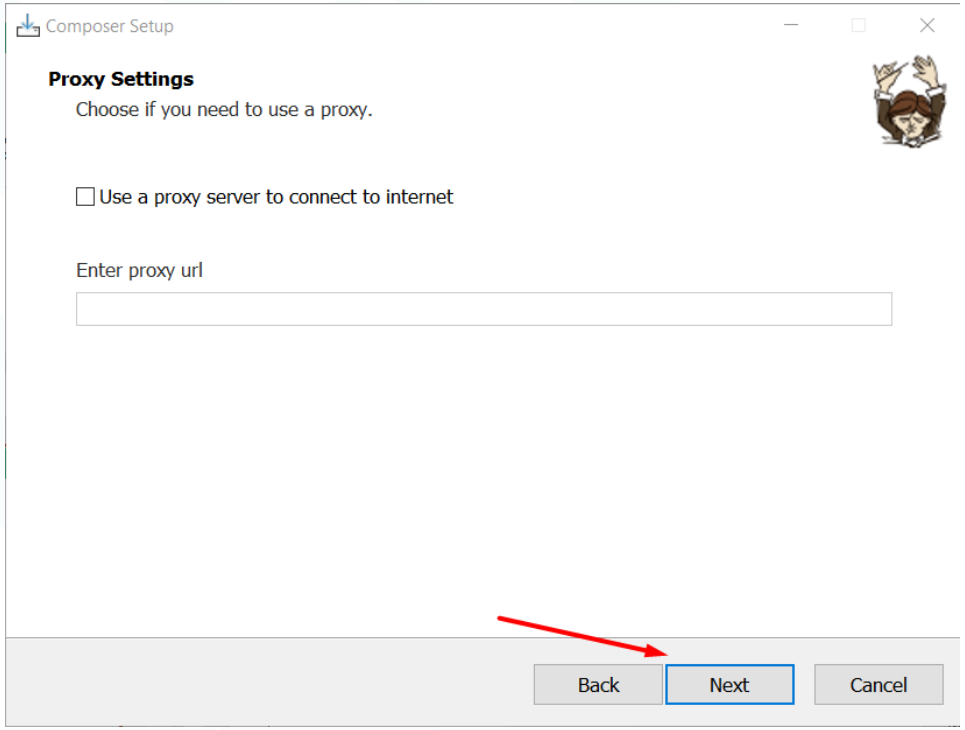


COMPOSER - INSTALLATION

Installation de Composer :

- Vérifier bien que le chemin vers l'installation de votre **PHP** est le bon .





COMPOSER - INSTALLATION

Installation de Composer :

- Pour vérifier que l'installation de composer ce soit bien réaliser utilisez la commande suivante dans un terminal :

  composer –V

```
C:\Users\lenovo\Desktop>composer -V
Composer version 2.7.1 2024-02-09 15:26:28
```

SYMFONY - INSTALLATION

Installation de la cli de Symfony :

via scoop :

installation de Scoop (installeur par ligne de commande windows)

ouvrir une powershell windows et effectuer les commandes suivantes :

- **Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser**
- **Invoke-RestMethod -Uri https://get.scoop.sh | Invoke-Expression**

- Ligne de commande d'installation de symfony cli via scoop :

  scoop install symfony-cli

Architecture

SYMFONY - ARCHITECTURE

Symfony est un ensemble de composants et de paquets de haute qualité

- Les composants sont un ensemble de classes fournissant un corps d'une fonctionnalité unique. Exemple: **CashComponent** est un composant fournissant des fonctionnalités de cash qui peuvent être ajoutées à n'importe quelle application

- Une application Symfony est elle même un paquet contenant une collection de sous paquets. Un paquet peut utiliser des composants Symfony ainsi que des composants tiers pour fournir des fonctionnalités. Exemple: **FrameworkBundle, DoctrineBundle** et **FrameworkExtraBundle**

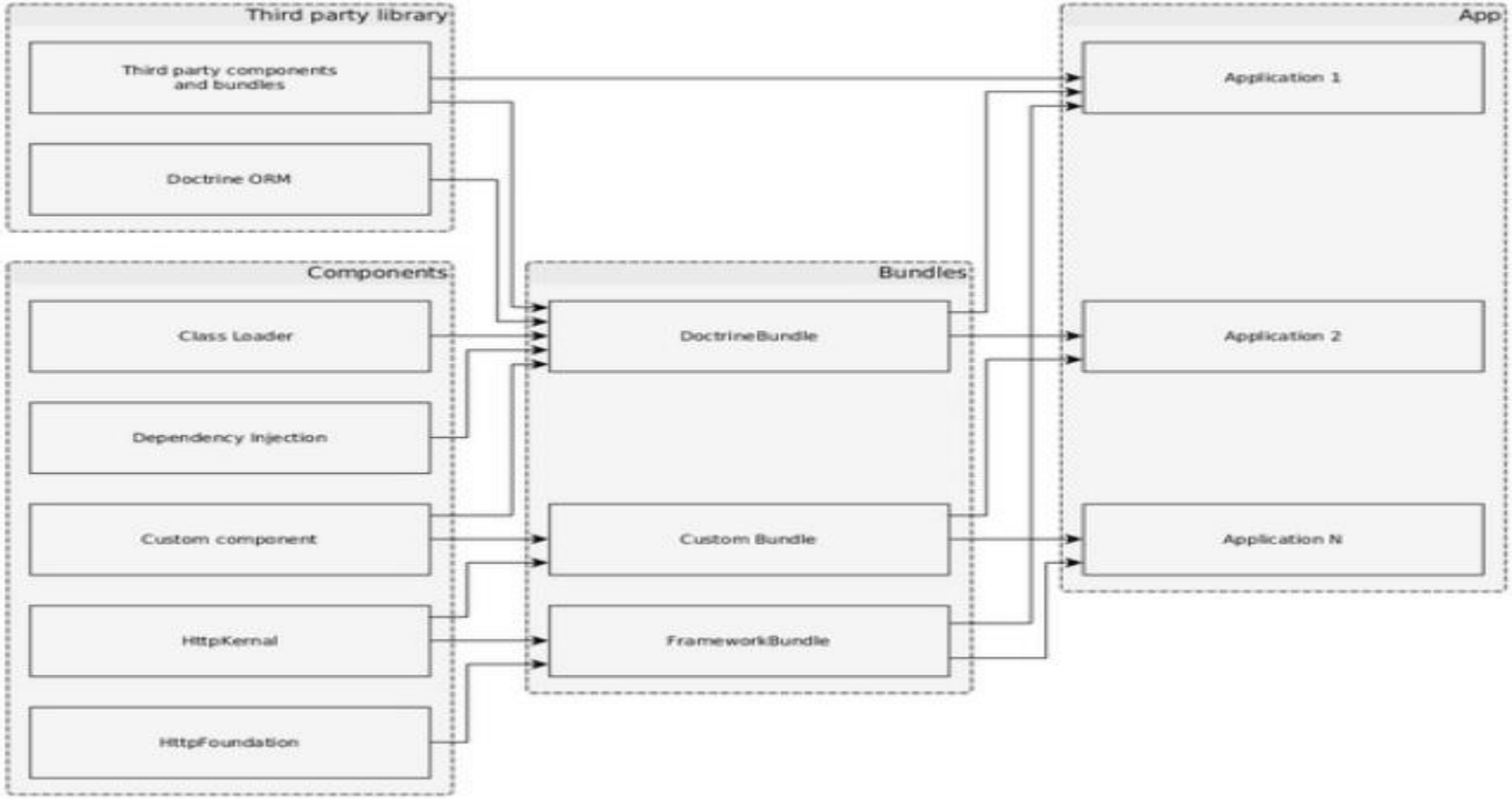
QUELQUES COMPOSANTS DE SYMFONY

- HttpFoundation : composant permettant de bien manipuler les requêtes et réponses HTTP
- Validator : composant permettant de valider les données
- Kernel : c'est le composant cœur de Symfony. Il permet la gestion des environnements , des routes et a la possibilité de manipuler des requêtes HTTP
- FileSystem : composant fournissant un système de commande de base pour la création de fichiers et dossiers, etc
- Console : composant fournissant différentes options de création de commandes qui peuvent être exécutées dans un terminal

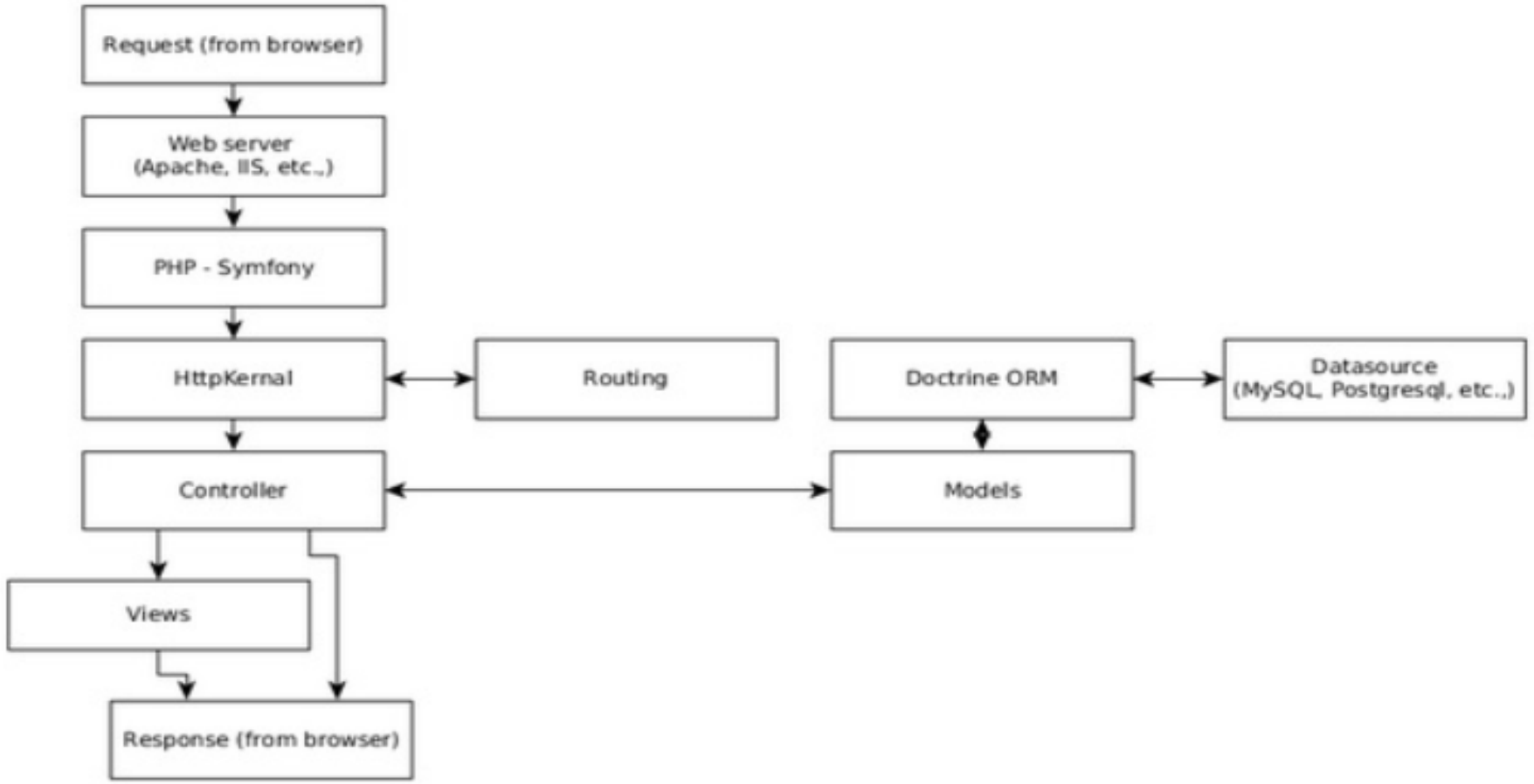
QUELQUES COMPOSANTS DE SYMFONY

- EventDispatcher : fournit un système d'actions basées sur des evets sur des objets PHP (Event et EventListener)
- DependencyInjection : fournit des mécanismes efficaces pour créer des objets avec leurs dépendances
- Serializer : permet la sérialisation d'objet vers différents formats (JSON, XML, …) et la dé sérialisation vers les objets originaux sans perte de données
- Translation : fournit des options pour internationaliser une application
- Debug : fournit différentes options pour activer le debugging en PHP
- PHPUnitBridge : fournit des options pour améliorer PHPUnit (Tests unitaires)
- Routing : composant permettant de faire un mapping entre les requêtes HTTP et une ensemble de variables de configuration prédéfinies
- Security : fournit un système de sécurité complet pour les applications : Authentification, Certification, ACL System

SYMFONY - ARCHITECTURE



SYMFONY - WORKFLOW



SYMFONY - ARCHITECTURE MVC

- MVC (Model View Controller) est une architecture en couches très utilisée pour les applications web
    - Model : couche qui représente la structure des différentes entités métiers (les données)
    - View : couche qui se charge d'afficher les entités (données) à l'utilisateur selon la situation
    - Controller : Manipule les requêtes de l'utilisateur en interagissant avec la couche Model est fourni ensuite une Vue avec les données nécessaires
- Le framework web de Symfony fournit des fonctionnalités de haut niveau nécessaires pour la mise en place d'une application d'entreprise

Création d’un projet

CRÉATION D'UN PROJET

Deux moyens pour créer vos projets Symfony :
Ouvrez votre terminal et rendez-vous à l'endroit où vous souhaitez créer votre projet.
Avec composer :

composer create-project symfony/skeleton:"6.4.*" my_project_directory

Explication :

- symfony/skeleton : Le squelette de base pour un nouveau projet Symfony.
- "6.4.*" : Spécifie la version LTS 6.4 de Symfony.
- my_project_directory : Nom du dossier de votre projet.

Vous pouvez également utiliser l'outil en ligne de commande Symfony :

symfony new my_project_directory --version=lts

Explication :

- symfony new : Crée un nouveau projet Symfony.
- my_project_directory : Nom du dossier de votre projet.
- --version=lts : Indique que le projet doit utiliser la version LTS (dans ce cas, la 6.4.*).

CRÉATION D'UN PROJET

Une fois votre projet Symfony créé, vous pouvez le lancer localement pour commencer à travailler dessus.
Voici les étapes à suivre :

- Étape 1 : Accéder au Répertoire du Projet:
  Dans le terminal, déplacez-vous dans le répertoire de votre projet :
  cd my_project_directory

- Étape 2 : Démarrer le Serveur Web de Symfony
  Utilisez la commande suivante pour lancer le serveur local :
  symfony server:start

- Étape 3 : Accéder à l'Application
  Après avoir démarré le serveur, ouvrez votre navigateur et rendez-vous à l'adresse suivante :
  http://localhost:8000
  Si tout est correct, vous devriez voir la page d'accueil de votre projet Symfony.

- Pour arrêter le Serveur : Pour arrêter le serveur, utilisez **symfony server:stop**

INSTALLATION DU MAKER-BUNDLE

Le **MakerBundle** est un outil indispensable lors du développement d'applications avec Symfony.
Il permet de générer automatiquement du code pour différentes parties de votre application, telles que des contrôleurs, des entités, des formulaires, des tests, et bien plus encore.

Cela permet de gagner du temps et de suivre les meilleures pratiques de développement Symfony.

Utilité du MakerBundle:

- Génération Automatique de Code : Créez des contrôleurs, des entités, des formulaires, et d'autres composants essentiels avec une seule commande.
- Conformité aux Meilleures Pratiques : Le code généré suit les recommandations et conventions de Symfony.
- Gain de Temps : Automatiser la création de code répétitif et complexe, vous permettant de vous concentrer sur la logique métier de votre application.

INSTALLATION DU MAKER-BUNDLE

Pour installer le MakerBundle, exécutez la commande suivante dans le répertoire de votre projet :

composer require symfony/maker-bundle –dev

Explication :

- composer require : Installe un nouveau package dans votre projet.
- symfony/maker-bundle : Nom du bundle à installer.
- --dev : Option indiquant que le bundle est installé uniquement pour l'environnement de développement.

Controller

SYMFONY - CONTROLLER

Les contrôleurs sont au cœur de toute application Symfony. Ils traitent les requêtes des utilisateurs et renvoient des réponses, qu'il s'agisse de pages HTML, de JSON, ou de tout autre format. Symfony facilite leur création grâce au MakerBundle.

Utilisez la commande suivante pour créer un nouveau contrôleur :
php bin/console make:controller DemoController

Explication :

- **make:controller** : Commande du MakerBundle pour créer un contrôleur.
- **DemoController** : Nom du contrôleur. Symfony ajoute automatiquement le suffixe "Controller" si ce n'est pas déjà présent.

Note : **php bin/console** peut être remplacer par **symfony console**

SYMFONY - CONTROLLER

La commande génère un fichier de contrôleur : src/Controller/DemoController.php.

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\Routing\Attribute\Route;


class DemoController extends AbstractController
{

    #[Route('/demo', name: 'app_demo')]
    public function index(): JsonResponse
    {
        return $this->json([
            'message' => 'Welcome to your new controller!',
            'path' => 'src/Controller/DemoController.php',
        ]);
    }
}
```

Explication :

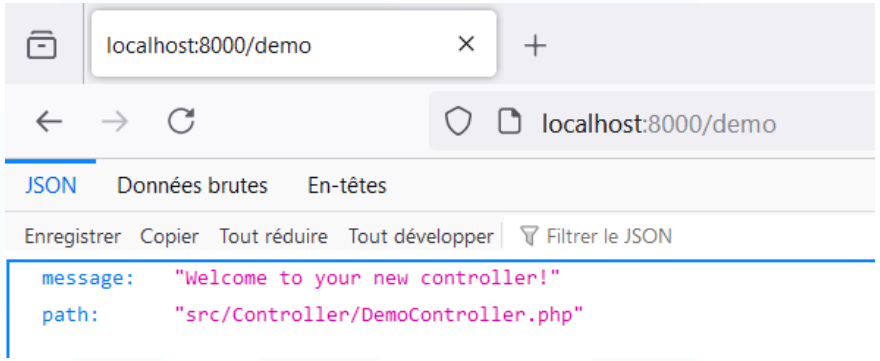
- **#[Route('/demo', name: 'api_demo')]** :
  Cette annotation définit l'URL (/api/data) et le nom de la route (api_data).
- **index()** :
  Méthode appelée lorsque l'URL /demo est visitée.
- **$this->json([...])** :
  Retourne directement un tableau formaté en JSON.

SYMFONY - CONTROLLER

Ouvrez votre navigateur rendez-vous à l'adresse :

http://localhost:8000/demo .

Vous verrez une réponse JSON structurée :



Points à retenir :

- L'utilisation de $this->json([...]) simplifie la génération de réponses JSON.
  Cette méthode garantit un formatage correct et standardisé des données en JSON.

- Les contrôleurs Symfony peuvent facilement gérer des requêtes et fournir des réponses adaptées, que ce soit en JSON, HTML, etc.

Routing

SYMFONY - ROUTING

- Le **Routing** est un mécanisme de correspondance entre des URI et leurs programme (controller) correspondant
- Le composant **HttpKernel** intercepte la requête du navigateur (URL) et détermine la route correspondante à l'URL de la requête
- Les correspondances sont exprimées dans un fichier de configuration YAML, XML ou dans un fichier PHP à travers des **annotations** « **@** » ou « #» depuis symfony 6
- Une URI/URL se compose généralement de trois segments :
  - Machine (host) : www.mon_appli.com , http://localhost:8000 …
  - Chemin (Path) : /articles/detail_article
  - Requête (request segment) : ?id_article=100

SYMFONY – ROUTING

CONFIGURATION AVEC ANNOTATIONS

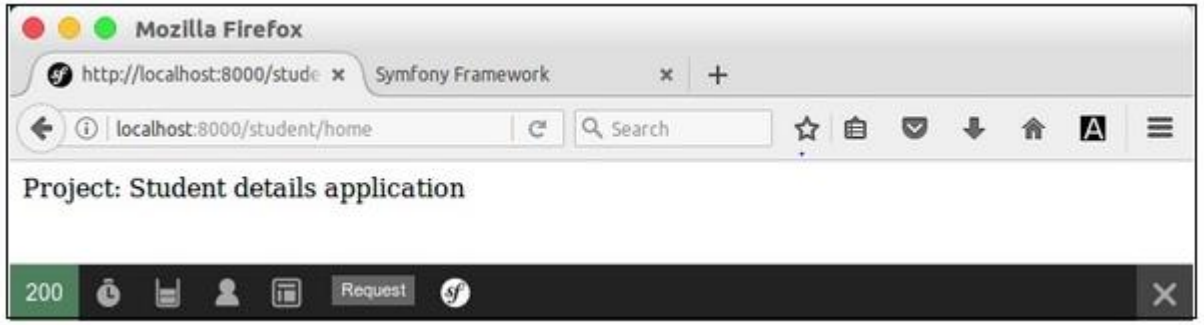
- Dans cette classe, deux Routes sont définies avec l'annotation #Route qui se trouve au-dessus des méthodes

- La fonction sayHello est exécutée si le navigateur envoi une URI se terminant par /hello .
- La fonction sayHelloTo est exécutée si l'URI se termine avec /hello/unNom

```php
namespace App\Controller;

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

4 references | 0 implementations
class HelloController extends AbstractController
{


    #[Route("hello", name:"app_hello")]
    2 references | 0 overrides
    public function sayHello()
    {
        return new Response('Salut ! ');
    }

    #[Route('/hello/{nom}',  name: "app_salut_toi")]
    2 references | 0 overrides
    public function sayHelloTo($nom)
    {
        return $this->render('bonjour.html.twig', [
            'nom'=> $nom,
        ]);
    }
```

EXEMPLE D'UTILISATION D'UNE ROUTE

```php
namespace App\Controller;

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

2 references | 0 implementations
class StudentController extends AbstractController
{
    #[Route('/student/home', name: 'app_student')]
    1 reference | 0 overrides
    public function index(): Response
    {
        $name = 'student details application';

        return new Response('<h1> Project:'.$name . '</h1>');
    }
```



Template twig

INSTALLATION DU BUNDLE TWIG

Pour rendre vos pages web plus dynamiques et flexibles, nous allons installer Twig, le moteur de templates par défaut de Symfony.

Twig permet de créer des vues HTML en offrant des fonctionnalités avancées telles que l'héritage de templates, les filtres, et les boucles, rendant ainsi le développement plus fluide et maintenable.

Commande d'installation :

composer require twig

INSTALLATION DU BUNDLE TWIG

Vérification de l'installation

Vous trouverez un nouveau dossier **templates/** dans la racine de votre projet, où vous pourrez créer vos fichiers de **templates .html.twig**.

Points à retenir

- Twig est le moteur de templates intégré à Symfony, permettant de créer des vues HTML de manière efficace.
- L'installation se fait simplement avec composer require twig.
- Un dossier templates/ sera créé dans votre projet pour contenir tous vos fichiers de templates.

SYMFONY – VIEW ENGINE - TWIG

Maintenant que **Twig** est installé, nous pouvons créer notre première **template** pour afficher du contenu **HTML**.

Symfony a créé automatiquement un dossier **templates/** à la racine du projet lors de l'installation de **Twig**. Nous allons ajouter un nouveau fichier HTML Twig dans ce dossier.

Dans le dossier **templates/**, créez un fichier appelé **home.html.twig**.

Contenu du fichier :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Bienvenue</title>
</head>
<body>
<h1>{{ message }}</h1>
</body>
</html>
```

Ici, nous avons une simple **template** twig contenant une variable **{{ message }}** que nous remplirons depuis le contrôleur.

SYMFONY – VIEW ENGINE - TWIG

Modification du contrôleur existant (DemoController) pour rendre une vue Twig au lieu de renvoyer une réponse JSON.

```php
#[Route('/', name: 'app_home')]
public function home(): Response
{
    return $this->render( view: 'home.html.twig', [
        'message' => 'Bienvenue sur symfony !',
    ]);
}
```

- #[Route('/', name: 'app_home')] définit une route pour l'URL '/'.
- Méthode home() : Cette méthode retourne une réponse en utilisant $this->render().
- $this->render() : C'est une méthode qui rend la vue Twig spécifiée ('home.html.twig').
- Passage de variables : Dans le tableau, la clé 'message' est passée à la vue avec la valeur 'Bienvenue sur symfony !'. Dans le template Twig, cette variable peut être affichée en utilisant {{ message }}.

SYMFONY – VIEW ENGINE - TWIG

Pour ajouter un fichier CSS et le relier à une template Twig. Symfony permet de gérer facilement les ressources (CSS, JavaScript, images) et de les intégrer dans vos templates.

créer un fichier CSS dans le dossier public du projet Symfony. Ce dossier contient tous les fichiers accessibles publiquement depuis le navigateur.

Code du fichier CSS (public/css/style.css) :

```css
body {
    background-color: #f0f0f0;
    font-family: Arial, sans-serif;
}

h1 {
    color: #333;
    text-align: center;
    margin-top: 50px;
}
```

SYMFONY – VIEW ENGINE - TWIG

Avant de faire le lien entre votre css et votre template il faut installer le bundle suivant :

composer require symfony/asset

Cette commande ajoute le package symfony/asset à votre projet, ce qui permet l'utilisation de la fonction **asset()** dans vos templates Twig pour inclure des fichiers statiques.

Pour lier le Fichier CSS à la Template , ajoutez le lien suivant vers le fichier CSS dans la section <head>

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Bienvenue</title>
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
</head>
<body>
<h1>{{ message }}</h1>
</body>
</html>
```

{{ asset('css/style.css') }} :
La fonction asset() génère l'URL du fichier CSS situé dans le dossier public/.

Note : cela est le même procéder pour les images et autres fichiers du dossier public.

SYMFONY – VIEW ENGINE - TWIG

- La couche View de Syfmony se charge de la séparation entre la logique applicative et la logique de présentation
- Si le Controller aura besoin d'afficher du HTML, CSS, il transfert la tâche au moteur de vue (View Engine)
- Par défaut, les fichiers Html, se trouvent dans le dossier « templates »
- Par défaut, Synfony utilise le moteur de temlplating « TWIG »
- Twig est un langage de templating puissant qui permet d'écrire des templates lisible de manière facile
- Syntaxe générale de twig :
  - {{ ….. }} affiche une variable ou le résultat d'une expression dans le template (page)
  - {% …. %} un mot clé (TAG) permettant le contrôle de la logique. Utilisé généralement pour l'appel de fonctions
  - {# …. #} : mettre un commentaire mono ou multi-lignes

SYMFONY – TWIG - NOTION DE LAYOUT

Dans Symfony, il est important de structurer vos vues pour éviter la répétition du code HTML. Cela est possible grâce aux layouts et aux blocs. La création d'un fichier base.html.twig permet de définir un modèle de base pour vos pages.
Création/Modificaiton d'un Layout (base.html.twig)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Bienvenue{% endblock %}</title>
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
</head>
<body>
<header>
    {% block header %} {# Un bloc pour l'en-tête, réutilisable dans toutes les pages #}
        <h1>Mon Site</h1>
    {% endblock %}
</header>
<main>
    {% block body %}{# Un bloc pour le contenu principal de la page #}
    {% endblock %}
</main>
<footer>
    {% block footer %}  {# Un bloc pour le pied de page #}
        <p>&copy; 2024 - Mon Site</p>
    {% endblock %}
</footer>
</body>
</html>
```

SYMFONY – TWIG - NOTION DE LAYOUT

- **{% block title %}** : Utilisé pour personnaliser le titre de la page. Chaque page enfant peut remplacer ce bloc pour changer le contenu du <title>.
- **{% block header %}** : Permet de définir ou de modifier l'en-tête de chaque page.
- **{% block body %}** : Le contenu principal de la page. Les vues enfants viendront injecter leur contenu ici.
- **{% block footer %}** : Permet de personnaliser le pied de page.

Pour utiliser le layout dans une autre vue, on utilise **{% extends 'base.html.twig' %}**

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Bienvenue</title>
</head>
<body>
{% extends 'base.html.twig' %}
{% block title %}Page d'Accueil{% endblock %}
{% block body %}
    <h1>Bienvenue sur la page d'accueil</h1>
    <p>Voici le contenu principal de la page.</p>
{% endblock %}
</body>
</html>
```

Les blocs **({% block ... %})** permettent de personnaliser les différentes parties du layout.

SYMFONY – TWIG - NOTION DE LAYOUT

- **{% block title %}** : Utilisé pour personnaliser le titre de la page. Chaque page enfant peut remplacer ce bloc pour changer le contenu du <title>.
- **{% block header %}** : Permet de définir ou de modifier l'en-tête de chaque page.
- **{% block body %}** : Le contenu principal de la page. Les vues enfants viendront injecter leur contenu ici.
- **{% block footer %}** : Permet de personnaliser le pied de page.

Pour utiliser le layout dans une autre vue, on utilise **{% extends 'base.html.twig' %}**

```
{% extends 'base.html.twig' %}
{% block title %}Page d'Accueil{% endblock %}
{% block body %}
    <h1>Bienvenue sur la page d'accueil</h1>
    <p>Voici le contenu principal de la page.</p>
{% endblock %}
```

Les blocs **({% block ... %})** permettent de personnaliser les différentes parties du layout.

SYMFONY – TWIG – LES PARTIALS

Les partials sont des morceaux de code réutilisables que l'on peut inclure dans différents templates. Ils permettent de ne pas dupliquer le code (par exemple, un menu ou un pied de page). Les partials sont souvent nommés avec un underscore (_) pour indiquer qu'ils sont inclus dans d'autres fichiers.

Créez un fichier dans le répertoire templates/_partials pour le partial :

```
{# templates/_partials/_menu.html.twig #}
<nav>
    <ul>
        <li><a href="{{ path('app_home') }}">Accueil</a></li>
        <li><a href="{{ path('app_about') }}">À Propos</a></li>
        <li><a href="{{ path('app_contact') }}">Contact</a></li>
    </ul>
</nav>
```

La fonction path génère automatiquement l'URL complète en se basant sur la définition de la route et les paramètres fournis

SYMFONY – TWIG – LES PARTIALS

Pour inclure un partial, on utilise {% include %} :

```
{# templates/base.html.twig #}
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Bienvenue{% endblock %}</title>
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
</head>
<body>
<header>
    {% block header %}
        <h1>Mon Site</h1>
        {% include '_partials/_menu.html.twig' %}
    {% endblock %}
</header>
```

SYMFONY – TWIG – LES PARTIALS

Points à retenir :

- **Layouts** : Les layouts (base.html.twig) permettent de créer un modèle de base pour vos pages, avec des blocs à personnaliser dans chaque template enfant.
- **Blocks** : Les blocs ({% block %}) permettent de définir des parties de la vue qui pourront être remplacées par les templates enfants.
- **Partials** : Les partials ({% include %}) sont des fragments réutilisables, utiles pour éviter la duplication de code.

SYMFONY – VIEW ENGINE - TWIG
FILTER TWIG

Dans l'écosystème de Twig, un ensemble complet de filtres offre une palette riche d'outils pour la manipulation et le formatage des données , voici un aperçu de quelques-uns de ces filtres essentiels:

Filtre	Action	Exemple d'utilisation	Résultat
default	Retourne une valeur par défaut si la variable est vide ou non définie	**{{ variable | default('Valeur par défaut') }}**	variable si définie, sinon 'Valeur par défaut'
length	Retourne la longueur d'une chaîne, d'un tableau ou d'un objet	**{{ "abcdef" | length }}**	6
upper	Convertit une chaîne en majuscules	**{{ "hello" | upper }}**	"HELLO"
lower	Convertit une chaîne en minuscules	**{{ "WORLD" | lower }}**	"world"
title	Convertit la première lettre de chaque mot en majuscule	**{{ "this is a title" | title }}**	"This Is A Title"
capitalize	Convertit la première lettre en majuscule et les autres en minuscules	**{{ "foo bar" | capitalize }}**	"Foo bar"
slice	Extrait une partie d'une chaîne ou d'un tableau	**{{ "abcdef" | slice(1, 3) }}**	"bc"
date	Formate une date	**{{ "2024-03-08 15:30:00" | date('d/m/Y H:i:s', 'Europe/Paris') }}**	"08/03/2024 15:30:00"
number_format	Formate un nombre avec des milliers séparés par des virgules	**{{ 10000 | number_format(2, ',', '.') }}**	"10.000,00"
escape	Échappe les caractères spéciaux dans une chaîne	**{{ " Hello " | escape }}**	" Hello "
striptags	Supprime toutes les balises HTML de la chaîne	**{{ " Hello " | striptags }}**	"Hello"

SYMFONY – VIEW ENGINE - TWIG
STRUCTURES DE CONTRÔLE

Twig permet d'ajouter des structures de contrôle dans vos page front-end, ces structures de contrôle ce présente sous forme de block.

Structure if else.. :

```
{% if condition %}

{% elseif condition %}

{% else %}

{% endif %}
```

Structure for in :

```
{% for person in people %}

{% endfor %}
```

Doctrine

SYMFONY - MODEL

- La couche **Model** joue un rôle important dans le framework **Symfony** et dans la plupart des frameworks web. Elle représente les entités du métier.
- Les entités sont créées par les utilisateurs de l'application, récupérées à partir d'une base de données, modifiées par les utilisateurs puis **persistées** (sauvegardées) dans la base de données
- Les classes de la partie **Model** sont aussi affichées pour les utilisateurs à travers les **Vues**
- La principale problématique des classes du Model c'est qu'ils sont en format **objet** tandis que la base données est au format **relationnel** (tables et relations)
- Nous avons besoin de faire **correspondre** (**mapping**) ces classes à nos tables. Ceci est fait grâce au **Bundle Doctrine** de Symfony
- **DoctrineBundle** intègre Symfony avec un **ORM** (**Object Relationnel Mapper**) d'une base données. Le mapping (correspondance classe ⇔ table) est réalisé grâce à l'outil **ORM**

UTILISATION DE DOCTRINE

- Pour installer doctrine (cas d'un projet symfony skeleton) :

  composer require doctrine maker

- Pour créer la base de données (sans tables) :
  - Ouvrir le fichier « .env » qui se trouve à la racine du projet et modifier la propriété DATABASE_URL avec les bons paramètres (user_name, password, hoost, db_name)

```
DATABASE_URL=mysql://root:@127.0.0.1:3306/db_hello_world?serverVersion=5.7
```

  - Lancer la commande suivante (toujours se positionner sur la racine du projet):

    php bin/console doctrine:database:create

```
$ php bin/console doctrine:database:create
Created database `db_hello_world` for connection named default
```

UTILISATION DE DOCTRINE

Pour créer une table dans la base de données, il faut passer par les étapes suivantes :

- Création de la classe « Entity » associée à la table

php bin/console make:entity Person

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

 created: src/Entity/Person.php
 created: src/Repository/PersonRepository.php

 Entity generated! Now let's add some fields!
 You can always add more fields later manually or by re-running this command.

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

- Après avoir créé l'Entity « Person » symfony propose d'ajouter des attributs à l'entity avec leur nom, type, longueur et si ils sont nullables ou pas
- Une fois la saisie terminée, la classe « Person » est créée avec ses attributs avec leurs getters et setters .. Génial non !

UTILISATION DE DOCTRINE

ENTITY

La classe contient les informations suivantes :

- le namespace **App\Entity** : dossier où va se trouver notre classe Person
- Utilisation du bundle Doctrine et de son composant **Mapping** (utilisation de l'alias **ORM)**
- La classe est annotée avec **#[ORM\Entity(..)]**
- L'annotation possède l'attribut **repositoryClass** qui indique la classe se chargeant de la communication avec la base de données pour l'entité
- l'attribut id de l classe Person est annoté avec **#[ORM/Id].** Ceci veut dire que cet attribut représentera l'identifiant de l'entité Person (**PRIMARY KEY)**
- **#[ORM/GeneratedValue]** précise la stratégie utilisée pour générer la clé primaire (id)
- **#[ORM/Column]** proposes des attributs pour donner des informations sur la colonne (nom, type, nulllable, longueur,..)

Person.php

```php
namespace App\Entity;

use App\Repository\PersonRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: PersonRepository::class)]
7 references | 0 implementations
class Person
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    1 reference
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    2 references
    private ?string $firstName = null;

    #[ORM\Column(length: 255)]
    2 references
    private ?string $lastName = null;

    #[ORM\Column]
    2 references
    private ?int $age = null;
```

UTILISATION DE DOCTRINE

REPOSITORY

- La classe **PersonRepository** hérite de la classe **ServiceEntityRepository** de **Doctrine**
- La classe fournit des méthodes toutes prêtes permettant d'effectuer quelques requêtes vers la base de données (find, findOneBy, findAll, findBy
- La classe propose des méthodes exemples utilisant des fonctions customiser à l'aide du QueryBuilder

PersonRepository.php

```php
namespace App\Repository;

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

/**
 * @method Person|null find($id, $lockMode = null, $lockVersion = null)
 * @method Person|null findOneBy(array $criteria, array $orderBy = null)
 * @method Person[]    findAll()
 * @method Person[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class PersonRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Person::class);
    }

    // /**
    //  * @return Person[] Returns an array of Person objects
    //  */
    /*
    public function findByExampleField($value)
    {
        return $this->createQueryBuilder('p')
            ->andWhere('p.exampleField = :val')
            ->setParameter('val', $value)
            ->orderBy('p.id', 'ASC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult()
```

UTILISATION DE DOCTRINE

MIGRATIONS

Afin de transformer (migrer) une classe de type Entity en une table dans une base de données relationnelle, il faut passer par les deux étapes suivantes :

- Exécution de la commande permettant la génération de classes de type « Migration ». Les classes migrations possèderont des méthodes permettant la création /suppression de la table dans/depuis la base de données. Les classes générées peuvent être modifiées au besoin

php bin/console doctrine:migrations:diff

```
$ php bin/console doctrine:migrations:diff
Generated new migration class to "F:\Devloppement\php\symfony\HelloWorld/src/Mi
rations/Version20191120193836.php"

To run just this migration for testing purposes, you can use migrations:execute
--up 20191120193836

To revert the migration you can use migrations:execute --down 20191120193836
```

UTILISATION DE DOCTRINE

MIGRATIONS

- La méthode **up** permet la création de la table dans la base de données
- La méthode **down** permet la suppression de la table depuis la base de données

```php
final class Version20191120193836 extends AbstractMigration
{
    public function getDescription() : string
    {
        return '';
    }

    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql',
                                'Migration can only be executed safely on \'mysql\'.');

        $this->addSql('CREATE TABLE person (
                id INT AUTO_INCREMENT NOT NULL,
                first_name VARCHAR(50) DEFAULT NULL,
                last_name LONGTEXT DEFAULT NULL,
                age INT NOT NULL,
                mail_address VARCHAR(100) NOT NULL,
                PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');
    }

    public function down(Schema $schema) : void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql',
            'Migration can only be executed safely on \'mysql\'.');

        $this->addSql('DROP TABLE person');
    }
}
```

UTILISATION DE DOCTRINE

MIGRER LES MIGRATIONS

- Exécution de la commande effectuant la migration de la classe vers une table de la base de données :

php bin/console doctrine:migrations:migrate

```
WARNING! You are about to execute a database migration that could result in sche
ma changes and data loss. Are you sure you wish to continue? (y/n)y
Migrating up to 20191120193836 from 0

  ++ migrating 20191120193836

     -> CREATE TABLE person (id INT AUTO_INCREMENT NOT NULL, first_name VARCHAR(
50) DEFAULT NULL, last_name LONGTEXT DEFAULT NULL, age INT NOT NULL, mail_addres
s VARCHAR(100) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE
 `utf8mb4_unicode_ci` ENGINE = InnoDB

  ++ migrated (took 1225.4ms, used 14M memory)

  ------------------------

  ++ finished in 1326.9ms
  ++ used 14M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

UTILISATION DE DOCTRINE

LES RELATIONS

Quand vous travaillez sur un projet, il est souvent nécessaire de créer des relations entre les entités pour structurer vos données. Voici un aperçu des types de relations courantes et comment les mettre en place avec Doctrine.

Il existe trois principaux types de relations entre les entités :

- One-to-One (Un-à-un) : Une entité est liée à une autre et uniquement une autre. Par exemple, une personne peut avoir une seule carte d'identité.

- One-to-Many / Many-to-One (Un-à-plusieurs / Plusieurs-à-un) : Une entité est liée à plusieurs autres, tandis que chaque entité liée pointe vers une seule entité principale. Exemple : un auteur peut avoir plusieurs livres, mais chaque livre a un seul auteur.

- Many-to-Many (Plusieurs-à-plusieurs) : Une entité peut être liée à plusieurs autres, et réciproquement. Exemple : un étudiant peut suivre plusieurs cours, et chaque cours peut avoir plusieurs étudiants.

UTILISATION DE DOCTRINE

LES RELATIONS

Exemple : Relation One-to-Many

Prenons l'exemple d'une relation entre une entité Auteur et une entité Livre. Chaque auteur peut écrire plusieurs livres, mais chaque livre est écrit par un seul auteur.

```php
#[ORM\Entity(repositoryClass: AuthorRepository::class)]
class Author
{
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    2 usages
    #[ORM\Column(length: 50)]
    private ?string $name = null;

    5 usages
    #[ORM\OneToMany(targetEntity: Book::class, mappedBy: 'author', orphanRemoval: true)]
    private Collection $books;

    no usages
    public function __construct(){...}


    no usages
    public function addBook(Book $book): static{...}

    no usages
    public function removeBook(Book $book): static{...}
```

UTILISATION DE DOCTRINE

LES RELATIONS

Exemple : Relation One-to-Many

Prenons l'exemple d'une relation entre une entité Auteur et une entité Livre. Chaque auteur peut écrire plusieurs livres, mais chaque livre est écrit par un seul auteur.

```php
#[ORM\Entity(repositoryClass: AuthorRepository::class)]
class Author
{
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    2 usages
    #[ORM\Column(length: 50)]
    private ?string $name = null;

    5 usages
    #[ORM\OneToMany(targetEntity: Book::class, mappedBy: 'author', orphanRemoval: true)]
    private Collection $books;

    no usages
    public function __construct(){...}


    no usages
    public function addBook(Book $book): static{...}

    no usages
    public function removeBook(Book $book): static{...}
```

#[ORM\OneToMany] :
Spécifie qu'un auteur peut avoir plusieurs livres (targetEntity: Book::class).

mappedBy: 'author' : Fait référence à la propriété $author dans l'entité Book.

UTILISATION DE DOCTRINE

LES RELATIONS

Regardons plus en détails les méthodes __construct, addBook et removeBook

```php
public function __construct()
{
    $this->books = new ArrayCollection();
}
```

Le constructeur initialise la propriété $books comme une instance de ArrayCollection.

ArrayCollection est une classe de Doctrine utilisée pour gérer les collections d'objets (ici, les objets Book associés à un Author).

Sans cette initialisation, on ne pourrait pas ajouter ou manipuler les livres liés à l'auteur.

UTILISATION DE DOCTRINE

LES RELATIONS

```php
public function addBook(Book $book): static
{
    if (!$this->books->contains($book)) {
        $this->books->add($book);
        $book->setAuthor($this);
    }

    return $this;
}
```

Paramètre : La méthode prend en entrée un objet de type Book.

Vérification : Elle vérifie d'abord si le livre n'est pas déjà présent dans la collection ($this->books->contains($book)). Ceci évite les doublons.

Ajout : Si le livre n'est pas dans la collection, il est ajouté à $this->books.

Mise à jour de l'Association : La ligne $book->setAuthor($this); définit l'auteur de ce livre. Cela maintient la cohérence entre l'objet Book et l'objet Author.

Retour : La méthode retourne l'objet actuel.

UTILISATION DE DOCTRINE

LES RELATIONS

```php
public function removeBook(Book $book): static
{
    if ($this->books->removeElement($book)) {
        if ($book->getAuthor() === $this) {
            $book->setAuthor( author: null);
        }
    }

    return $this;
}
```

Paramètre :
La méthode prend en entrée un objet Book à retirer de la collection.

Suppression :
La méthode utilise $this->books->removeElement($book) pour retirer le livre de la collection. Cette méthode renvoie true si l'élément a bien été retiré.

Mise à jour de l'Association :
Si le livre était associé à cet auteur ($book->getAuthor() === $this), alors on met l'auteur du livre à null avec $book->setAuthor(null). Cela garantit que la relation est correctement mise à jour des deux côtés.

Retour :
Comme pour addBook(), cette méthode retourne l'objet actuel.

UTILISATION DE DOCTRINE

LES RELATIONS

Classe Book.php

```php
#[ORM\Entity(repositoryClass: BookRepository::class)]
class Book
{
    1 usage
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    2 usages
    #[ORM\Column(length: 100)]
    private ?string $title = null;

    2 usages
    #[ORM\ManyToOne(inversedBy: 'books')]
    #[ORM\JoinColumn(nullable: false)]
    private ?Author $author = null;
```

#[ORM\ManyToOne] :
Indique que chaque livre a un seul auteur (targetEntity: Author::class).

inversedBy: 'books' :
Fait référence à la propriété $books dans l'entité Author.

#[ORM\JoinColumn(nullable: false)] :
Indique que l'association est obligatoire (un livre doit toujours avoir un auteur).

Formulaire

FORMULAIRE

CRÉATION DE FORMULAIRE

- Exécution de la commande du maker bundle pour crée un formulaire :

php bin/console make:form

```
The name of the form class (e.g. AgreeableElephantType):
> PersonType

The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
> Person

created: src/Form/PersonType.php

 Success!
```

FORMULAIRE

CRÉATION DE FORMULAIRE

- **buildForm** : Cette méthode est responsable de la construction du formulaire. En utilisant le paramètre **$builder**, vous pouvez ajouter des champs au formulaire. Ici , le formulaire aura trois champs : firstName, lastName, et age.

- **configureOptions** : Cette méthode configure les options du formulaire. Ici, la **classe Person** est définie comme la classe de données par défaut pour ce formulaire. Cela signifie que lorsque ce formulaire est soumis, les données seront liées à une instance de la classe Person

```php
namespace App\Form;

use App\Entity\Person;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

2 references | 0 implementations
class PersonType extends AbstractType
{
    0 references | 0 overrides
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('firstName')
            ->add('lastName')
            ->add('age')
        ;
    }

    0 references | 0 overrides
    public function configureOptions(OptionsResolver $resolver): void
    {
        $resolver->setDefaults([
            'data_class' => Person::class,
        ]);
    }
```

FORMULAIRE

VALIDATEUR ET CONTRAINTES

- ->add('lastName', **TextType::class**, [...]) Le champ est de type **TextType**, ce qui signifie que l'entrée est un champ de texte simple.
- **constraints** : C'est une option qui permet de définir des contraintes de validation pour ce champ.
- new **NotBlank()** : C'est une contrainte qui indique que ce champ ne peut pas être laissé vide.
- new **Length([...])** : Cette contrainte définit la longueur attendue pour le champ lastName. Avec les paramètres **min** et **max**.

```php
2 references | 0 implementations
class PersonType extends AbstractType
{
    0 references | 0 overrides
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
        ->add('firstName', TextType::class, [
            'constraints' => [
                new NotBlank(),
                new Length([
                    'min' => 4,
                    'max' => 50,
                    'minMessage' => 'Le prénom doit contenir au moins {{ limit }} caractères',
                    'maxMessage' => 'Le prénom ne peut pas contenir plus de {{ limit }} caractères',
                ]),
            ],
        ])
        ->add('lastName', TextType::class, [···
        ])
        ->add('age', IntegerType::class, [···
        ]);

    }
```

FORMULAIRE

CONTROLLEUR

- Une nouvelle instance de Person est créée pour représenter les données saisies dans le formulaire.
- Le formulaire est créé en utilisant **PersonType** pour l'entité Person.
- La méthode **handleRequest** gère les données soumises dans la requête et les associe au formulaire.
- Lorsque le formulaire est soumis et valide, la méthode **persist** prépare l'entité Person pour l'insertion en base de données.
- Ensuite, la méthode **flush** exécute les opérations SQL nécessaires pour insérer l'entité en base de données.

```php
#[Route('/person/new', name: 'person_create', methods: ['GET' , 'POST'])]
3 references | 0 overrides
public function create(Request $request, EntityManagerInterface $entityManager): Response
{
    $person = new Person();
    $form = $this->createForm(PersonType::class, $person);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager->persist($person);
        $entityManager->flush();


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

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

FORMULAIRE

TWIG

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

{% block body %}
    <div class="max-w-md mt-10 mx-auto">
        <h3 class="text-gray-700 text-xl font-bold my-10">Ajouter une personne : </h3>
        {{ form_start(form) }}
            <div class="mb-4">
                {{ form_label (form.firstName, 'Prénom', {'label_attr': {'class': 'block text-gray-700 text-sm font-bold mb-2'}}) }}
                {{ form_widget(form.firstName, {'attr': {'class': 'shadow appearance-none border rounded ...'}}) }}
                {{ form_errors(form.firstName) }}
            </div>
            <div class="mb-4">···
            </div>
            <div class="mb-4">···
            </div>
            <div class="flex items-center justify-between">
                <button type="submit" class="bg-blue-500 hover:bg-blue-700 ..." type="button">Envoyer</button>
            </div>
        {{ form_end(form) }}
    </div>
{% endblock %}
```

FORMULAIRE

TWIG

- **form_start(form)** :

  Démarre le formulaire.

- **form_label(form.firstName, 'Prénom', ...) :**

  Crée une étiquette pour le champ de formulaire 'firstName' avec le texte 'Prénom'.

  Les attributs de style peuvent être ajoutés à l'étiquette via la clé **'label_attr'**.

- **form_widget(form.firstName , ...) :**

  Génère le champ de saisie réel pour la propriété 'firstName' du formulaire.

  Les attributs de style peuvent être ajoutés au champ de saisie via la clé **'attr'**.

- **form_errors(form.firstName) :**

  Affiche les éventuelles erreurs de validation associées au champ 'firstName' du formulaire.

- **form_errors(form.firstName)** :

  Cette ligne affiche les éventuelles erreurs de validation associées au champ 'firstName' du formulaire

- **{{ form_end(form) }} :**

  Termine le formulaire.

Exemple récupération

RÉCUPÉRATION DE DONNÉES

CONTROLLER

Afin de pouvoir récupérer des données et de les afficher dans la vue, nous devons forcément prévoir une méthode dans le contrôler qui devra faire appel au repository de l'entité correspondante en utilisant l'injection de dépendance de Symfony .

```
no usages
#[Route('/', name: 'app_person_index', methods: ['GET'])]
public function getAllPerson(PersonRepository $personRepository): Response
{
    return $this->render( view: 'person/index.html.twig', [
        'people' => $personRepository->findAll(),
    ]);
}
```

La méthode findAll du repository permet de retourner un tableau d'objet Person correspondant à toutes les ligne de la table Person de la bdd.

RÉCUPÉRATION DE DONNÉES

TWIG

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

{% block title %}Liste Personnes{% endblock %}

{% block body %}

        <h1> Liste des personnes</h1>

        {% for person in people %}

                <p> id: {{ person.id }}</p>
                <p> firstName: {{ person.firstName }}</p>
                <p> lastName: {{ person.lastName }}</p>
                <p> age: {{ person.age }}</p>
                <hr>

        {% endfor %}

       <a href="{{ path('app_person_new') }}">Ajouter une nouvelle personne</a>
{% endblock %}
```

Dans cette Template **Twig** :

L'utilisation du block for permet de boucler sur la variable people

Le bloc **for** de **Twig** est utilisé pour itérer sur une collection, comme une liste ou un tableau, et exécuter du code pour chaque élément de la collection.

La fonction **path** de **Twig** est utilisée pour générer des **URL** à partir du nom d'une **route** dans **Symfony**. Elle prend en argument le **nom** de la route et éventuellement des paramètres à passer à cette route.

RÉCUPÉRATION DE DONNÉES

RÉCUPÉRATION D'UN OBJET

Afin de pouvoir récupérer des données et de les afficher dans la vue le principe reste le même, on peut utiliser la méthode **find** ou **findBy** du repository.

Ici on utilise la méthode **find** du repository qui prend en argument **l'id** de l'entité rechercher

```php
#[Route('/{id}', name: 'app_person_show', methods: ['GET'])]
public function show(int $id, PersonRepository $personRepository): Response
{

    return $this->render( view: 'person/show.html.twig', [
        'person' => $personRepository->find($id),
    ]);
}
```

RÉCUPÉRATION DE DONNÉES

RÉCUPÉRATION D'UN OBJET

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

{% block title %}Person{% endblock %}

{% block body %}

    {% if person %}

    <h1>Détails Personne : </h1>
    <p> id: {{ person.id }}</p>
    <p> firstName: {{ person.firstName }}</p>
    <p> lastName: {{ person.lastName }}</p>
    <p> age: {{ person.age }}</p>

    {% else %}

     <p> Personne non trouvée </p>

    {% endif %}

    <a href="{{ path('app_person_index') }}"> Retour à liste des personnes</a>
{% endblock %}
```

Dans cette Template **Twig** :

L'utilisation de la block **if** permet de vérifier si la variable **person** n'est pas vide afin d'afficher ou non les informations.

{% if condition %} : Cette balise ouvre le bloc conditionnel et spécifie la condition à évaluer.

{% else %} : Cette balise est facultative et permet de définir un bloc de contenu alternatif qui sera affiché si la condition dans le bloc **{% if %}** est fausse.

{% endif %} : Cette balise ferme le bloc conditionnel.

Exemple update

UPDATE DE DONNÉES

CONTROLLER

La méthode édit est une méthode qui va accepter des requêtes **post** et **get**,

la récupération personne grâce au paramètre **id** et à l'**entityManager** avec la méthode **getRepository(Entity::class)** qui permet de récupérer le repository de l'entité.

En utilisant **find()** pour récupérer une entité existante et que vous modifiez ses propriétés, vous n'avez pas besoin d'appeler **persist()** avant d'appeler **flush()**.

Doctrine détecte **automatiquement** les modifications apportées à l'entité gérée et les **synchronise** avec la base de données lors de l'exécution de la transaction de la base de données.

```php
#[Route('/{id}/edit', name: 'app_person_edit', methods: ['GET', 'POST'])]
public function edit(int $id,Request $request, EntityManagerInterface $entityManager): Response
{
    $person = $entityManager->getRepository(Person::class)->find($id);
    $form = $this->createForm( type: PersonType::class, $person);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $entityManager->flush();

        return $this->redirectToRoute( route: 'app_person_index', [], status: Response::HTTP_SEE_OTHER);
    }

    return $this->render( view: 'person/edit.html.twig', [
        'person' => $person,
        'form' => $form,
    ]);
}
```

UPDATE DE DONNÉES

TWIG

{% if person %} : Ce block permet d'afficher le contenu conditionnellement. Si la variable person est définie et non vide.

{{ include('person/_form.html.twig', {'button_label': 'Update'}) }} : Cette ligne inclut person/_form.html.twig.

Cela permet de réutiliser le même formulaire d'édition de personne dans plusieurs pages. Le paramètre **'button_label': 'Update'** est passé au formulaire pour définir le libellé du bouton de soumission du formulaire.

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

{% block title %}Edit Person{% endblock %}

{% block body %}
    <h1>Modifier Personne : </h1>

    {% if person %}

    {{ include('person/_form.html.twig', {'button_label': 'Update'}) }}

    {% endif %}
    <a href="{{ path('app_person_index') }}">Retour à liste des personnes </a>

{% endblock %}
```

Exemple delete

SUPPRESSION DE DONNÉES

CONTROLLER

La méthode delete doit être de type post, cette fois-ci on utilise la méthode remove de l'entityManager

```php
#[Route('/{id}', name: 'app_person_delete', methods: ['POST'])]
public function delete(Request $request, int $id, EntityManagerInterface $entityManager): Response
{
    $person = $entityManager->getRepository(Person::class)->find($id);
    if ($person) {
        $entityManager->remove($person);
        $entityManager->flush();
    }

    return $this->redirectToRoute( route: 'app_person_index', [], status: Response::HTTP_SEE_OTHER);
}
```

SUPPRESSION DE DONNÉES

TWIG

Dans la template twig on peut alors utiliser un formulaire qui pointera vers la route app_person_delete, en passant en paramètre l'id de l'objet personne que l'on manipule.

On peut utiliser la fonction **confirme** pour l'attribut **onSumbit** pour avoir une modal de confirmation rudimentaire.

```
<form
        method="post"
        action="{{ path('app_person_delete', {'id': person.id}) }}"
        onsubmit="return confirm('Voulez-vous vraiment supprimer cette personne ? ');"
>
    <button class="btn">Delete</button>
</form>
```

Fixtures

FIXTURES

Les **fixtures** en Symfony sont des classes utilisées pour charger des **données** de test dans la base de données de votre application.

Cela peut être extrêmement utile lors du développement ou du test de votre application.

Installation du bundle :

- **composer require --dev orm-fixtures**

FIXTURES

Création du fichier du fixtures :

Dans le fichier **AppFixutres**
Implémenter la méthode **load()**

Il suffit de créer un objet ou plusieurs à l'aide d'un tableau

D'utiliser l'ObjectManager pour persiter notre objet et de le flush

Pour lancer vos fixutres il vous suffira d'utlilser la commande :

php bin/console doctrine:fixtures:load

```php
namespace App\DataFixtures;

use ...
4 usages
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {


        $authors= [];
        for($i= 0 ;  $i< 5; $i++){

            $authors[$i] = new Author();
            $authors[$i]->setName( name: "John".$i);
            $authors[$i]->setFirstName( firstName: "Smtih".$i);
            $authors[$i]->setAge( age: $i+10);


            $manager->persist($authors[$i]);

        }

        $manager->flush();
```

Création d'utilisateur

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

L'intégration d'une gestion utilisateur efficace constitue une étape cruciale dans le développement de toute application.

Grâce au **Maker Bundle** de Symfony, nous sommes en mesure de mettre en place une base solide et sécurisée pour la gestion de nos utilisateurs. Pour ce faire, nous devons commencer par ajouter le **bundle de sécurité** via **Composer** :

composer require security

Une fois le bundle de sécurité installé, Symfony génère automatiquement le fichier de configuration **security.yaml**.
Ce fichier joue un rôle crucial dans la définition des stratégies de sécurité pour votre application.

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

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

when@test:
    security:
        password_hashers:

            Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt
                time_cost: 3 # Lowest possible value for argon
                memory_cost: 10 # Lowest possible value for argon
```

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

password_hashers :
Cette section définit la méthode de **hachage** des mots de passe des utilisateurs. La valeur **'auto'** indique que Symfony choisira automatiquement l'algorithme.

providers :
La section **'providers'** définit la source de données pour l'authentification des utilisateurs. Dans cet exemple, la configuration '**users_in_memory**' utilise un stockage en mémoire pour les utilisateurs.

firewalls :
Les **'firewalls'** définissent les règles de sécurité pour différentes zones de votre application. Dans cet exemple, le pare-feu **'dev'** est configuré pour exclure certains chemins de l'authentification.

access_control :
Cette section vous permet de **contrôler l'accès** à des parties spécifiques de votre application en fonction des **rôles des utilisateurs**.

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

Nous pouvons maintenant créer l'entité utilisateur à l'aide de la commande :

php bin/console make:user

Vous serez invité à fournir les détails suivants :

```
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]:
>

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
```

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

Deux nouveaux fichiers on était créé :
- **User.php** qui correspond à notre entité **User**
- **UserRepository.php** le repository de notre entité **User**

La commande a aussi mis à jour le fichier **security.yml**

Providers :

La commande a ajouté une nouvelle entrée dans la section providers.
le fournisseur **app_user_provider** est configuré pour charger les utilisateurs à partir d'une entité **User** de l'application.

Firewalls :

La configuration du pare-feu principal **'main'** a été modifiée pour inclure le fournisseur **app_user_provider**. Cela permet de l'utiliser pour gérer l'authentification des utilisateurs.

```yaml
providers:
    # used to reload user from session & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider

access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
```

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

User.php qui correspond à notre entité **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: ``user``)]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    no usages
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;
```

MAKER BUNDLE

CRÉATION DE L'UTILISATEUR

la classe **User** implémente deux interfaces clés :

- **UserInterface** est utilisée pour représenter un utilisateur dans le système. Les méthodes de cette interface comprennent :

  - **getUserIdentifier** : Cette méthode retourne un identifiant unique pour l'utilisateur, généralement son adresse e-mail ou son nom d'utilisateur.

  - **getRoles :** Cette méthode renvoie un tableau de rôles attribués à l'utilisateur.

  - **eraseCredentials** : Cette méthode est utilisée pour effacer les données sensibles de l'utilisateur.

PasswordAuthenticatedUserInterface est utilisée pour représenter un utilisateur authentifié par mot de passe. Les méthodes de cette interface comprennent :

  - **getPassword** : Cette méthode retourne le mot de passe haché de l'utilisateur.

AUTHENTIFICATION

MAKER BUNDLE

AUTHENTIFICATION

La commande **make:auth** de **Symfony** simplifie grandement la mise en place d'un système d'authentification complet dans une application.
Elle génère automatiquement les contrôleurs, les vues, les routes, les entités et d'autres composants nécessaires à l'authentification des utilisateurs que nous pouvons ensuite customiser.

php bin/console make:auth

```
What style of authentication do you want? [Empty authenticator]:
 [0] Empty authenticator
 [1] Login form authenticator
> 1

The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> UserAuthenticator

Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
>

Do you want to generate a '/logout' URL? (yes/no) [yes]:
>

Do you want to support remember me? (yes/no) [yes]:
> no
```

vous serez invité à fournir les détails suivants :

L'utilisation de formulaire de login,
Le nom de la classe pour l'authentificateur,
Le nom du contrôleur de sécurité,
La génération d'une url de déconnexion,
L'activation du remember me,

MAKER BUNDLE

AUTHENTIFICATION

Regardons les trois fichiers générés :

- **UserAutenticator :**
  Cette classe est responsable de la gestion du processus d'authentification de l'utilisateur. Elle s'occupe de valider les informations d'identification de l'utilisateur et de gérer les redirections après une authentification réussie.

- **Le SecurityController :**
  Ce contrôleur gère les fonctionnalités d'authentification, y compris l'affichage du formulaire de connexion et la déconnexion des utilisateurs

- **Login.html.twig :**
  Ce fichier de modèle intègre le formulaire de connexion dans votre application.

```
updated: config/packages/security.yaml
created: src/Controller/SecurityController.php
created: templates/security/login.html.twig
```

MAKER BUNDLE

USERAUTHENTICATOR

- Attribut privé **$urlGenerator** de type **UrlGeneratorInterface** qui est injecté via le constructeur de la classe. Cet attribut est utilisé pour générer des URL dans le processus d'authentification.

- Méthode **authenticate** : Cette méthode est responsable de l'authentification de l'utilisateur.

  Elle récupère les données de la requête, telles que l'email et le mot de passe, et les utilise pour créer un objet **Passport**.

- L'objet **Passport** contient des informations sur les informations d'identification de l'utilisateur et d'autres **badges** de sécurité tels que le jeton **CSRF**.

```php
class UserAuthenticator extends AbstractLoginFormAuthenticator
{
    use TargetPathTrait;

    1 usage
    public const LOGIN_ROUTE = 'app_login';

    no usages
    public function __construct(private UrlGeneratorInterface $urlGenerator)
    {
    }
    no usages
    public function authenticate(Request $request): Passport
    {
        $email = $request->request->get( key: 'email', default: '');

        $request->getSession()->set(SecurityRequestAttributes::LAST_USERNAME, $email);

        return new Passport(
            new UserBadge($email),
            new PasswordCredentials($request->request->get( key: 'password', default: '')),
            [
                new CsrfTokenBadge( csrfTokenId: 'authenticate', $request->request->get( key: '_csrf_token')),
        );
    }
```

MAKER BUNDLE

USERAUTHENTICATOR

- Méthode **onAuthenticationSuccess** :
  est appelée lorsque l'authentification réussit. Elle vérifie d'abord s'il existe un chemin cible (target path) enregistré dans la session. Si un chemin cible existe, l'utilisateur est redirigé vers ce chemin. Sinon, vous devez
  personnaliser cette méthode pour rediriger l'utilisateur vers une page spécifique après son authentification.

- Méthode **getLoginUrl** :
  renvoie l'URL de la page de connexion en cas d'échec de l'authentification.
  Elle utilise **l'UrlGeneratorInterface** pour générer l'URL de la route de connexion spécifiée par la constante **LOGIN_ROUTE**

```
no usages
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
        return new RedirectResponse($targetPath);
    }
    // For example:
     return new RedirectResponse($this->urlGenerator->generate( name: 'app_home'));
}

no usages
protected function getLoginUrl(Request $request): string
{
    return $this->urlGenerator->generate( name: self::LOGIN_ROUTE);
}
```

MAKER BUNDLE

SECURITYCONTROLLER

- Méthode **login** :
  Responsable de l'affichage du formulaire de connexion. Elle prend en paramètre l'objet **AuthenticationUtils**, qui est utilisé pour récupérer les erreurs d'authentification précédentes et le dernier nom d'utilisateur saisi.
  La méthode renvoie ensuite le rendu de la vue **security/login.html.twig**, en passant les variables **last_username** et **error** en cas d'erreur de connexion.

- Méthode **logout** :
  Gère la déconnexion de l'utilisateur. Dans cet exemple, une exception logique est levée pour indiquer que la méthode peut rester vide, car elle est interceptée par la clé de déconnexion sur votre pare-feu de sécurité.

```php
no usages
class SecurityController extends AbstractController
{
    #[Route(path: '/login', name: 'app_login')]
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        if ($this->getUser()) {
            return $this->redirectToRoute( route: 'app_home');
        }
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();
        return $this->render( view: 'security/login.html.twig', ['last_username' => $lastUsername, 'error' => $error]);
    }

    #[Route(path: '/logout', name: 'app_logout')]
    public function logout(): void
    {
        throw new \LogicException( message: 'This method can be blank - it will be intercepted by the logout key on your firewall.');
    }
}
```

MAKER BUNDLE

AUTHENTIFICATION

La mise à jour du fichier **security.yaml** suite à l'exécution de la commande **make:auth** permet de configurer les paramètres de sécurité spécifiques à votre système d'authentification nouvellement généré. Voici comment cette mise à jour s'articule par rapport à la commande.

La section **firewalls** est mise à jour pour inclure des détails sur le pare-feu principale (**main**) de l'application.

Cette mise à jour spécifie **l'authentificateur** personnalisé (**custom_authenticator**) **UserAuthenticator** pour gérer le processus d'authentification de l'utilisateur.

Ainsi que le chemin pour le **logout**

```yaml
providers:
    # used to reload user from session & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        custom_authenticator: App\Security\UserAuthenticator
        logout:
            path: app_logout
            # where to redirect after logout
            # target: app_any_route
```

Registration

MAKER BUNDLE

REGISTRATION

Le maker bundle permet de générer un formulaire de d'inscription pour une application Symfony.
Ce formulaire est déjà pré-configuré avec les validations nécessaires pour garantir la qualité des données saisies par les utilisateurs .

Il constitue une base solide pour l'inscription, offrant la possibilité de personnaliser et d'ajuster le formulaire selon les besoins spécifiques de votre application.

php bin/console make:registration-form

```
Creating a registration form for App\Entity\User

Do you want to add a #[UniqueEntity] validation attribute to your User class to make sure duplicate accounts aren't created? (yes/no) [yes]:
>

Do you want to send an email to verify the user's email address after registration? (yes/no) [yes]:
> no

Do you want to automatically authenticate the user after registration? (yes/no) [yes]:
>

updated: src/Entity/User.php
created: src/Form/RegistrationFormType.php
created: src/Controller/RegistrationController.php
created: templates/registration/register.html.twig
```

MAKER BUNDLE

REGISTRATION

Regardons les trois fichiers générés :

- **RegistrationFormType:**

  Ce fichier représente le formulaire d'inscription de l'utilisateur, définissant les champs et les validations nécessaires pour la saisie des informations d'inscription.

- **Le RegistrationController:**

  Ce contrôleur est responsable de la gestion du processus d'inscription, traitant les demandes d'inscription, validant les données saisies et enregistrant les nouveaux utilisateurs dans la base de données.

- **register.html.twig :**

  Cette template Twig intègre le formulaire de type RegistrationForm, offrant une interface conviviale pour que les utilisateurs puissent saisir leurs informations d'inscription.

MAKER BUNDLE

REGISTRATIONFORMTYPE

Ce fichier contient la classe du formulaire de d'inscription. Cette classe est dérivée de la classe **AbstractType** de Symfony. La classe **AbstractType** fournit une base pour la création de formulaires Symfony.

La classe **RegistrationFormType** contient les propriétés suivantes :

email : Cette propriété définit les propriétés du champ email du formulaire.

Le champ agreeTerms : Cette propriété définit à cocher pour l'acceptation des termes et conditions.

plainPassword : Cette propriété définit les propriétés du champ password du formulaire.

```php
class RegistrationFormType extends AbstractType
{
    no usages
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add( child: 'email')
            ->add( child: 'agreeTerms', type: CheckboxType::class, [
                'mapped' => false,
                'constraints' => [
                    new IsTrue([
                        'message' => 'You should agree to our terms.',
                    ]),
                ],
            ])
            ->add( child: 'plainPassword', type: PasswordType::class, [
                'mapped' => false,
                'attr' => ['autocomplete' => 'new-password'],
                'constraints' => [
                    new NotBlank([
                        'message' => 'Please enter a password',
                    ]),
                    new Length([
                        'min' => 6,
                        'minMessage' => 'Your password should be at least {{ limit }} characters',
                        'max' => 4096,
                    ]),
                ],
            ])
```

MAKER BUNDLE

REGISTER.HTML.TWIG

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

{% block title %}Register{% endblock %}

{% block body %}
    <h1>Register</h1>

    {{ form_errors(registrationForm) }}

    {{ form_start(registrationForm) }}
        {{ form_row(registrationForm.email) }}
        {{ form_row(registrationForm.plainPassword, {
            label: 'Password'
        }) }}
        {{ form_row(registrationForm.agreeTerms) }}

        <button type="submit" class="btn">Register</button>
    {{ form_end(registrationForm) }}
{% endblock %}
```

MAKER BUNDLE

REGISTERATIONCONTROLLER

```php
class RegistrationController extends AbstractController
{
    no usages
    #[Route('/register', name: 'app_register')]
    public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher, UserAuthenticatorInterface $userAuthenticator,
    {
        $user = new User();
        $form = $this->createForm( type: RegistrationFormType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            // encode the plain password
            $user->setPassword(
                $userPasswordHasher->hashPassword(
                    $user,
                    $form->get('plainPassword')->getData()
                )
            );

            $entityManager->persist($user);
            $entityManager->flush();

            return $userAuthenticator->authenticateUser(
                $user,
                $authenticator,
                $request
            );
        }
```

MAKER BUNDLE

REGISTERATION CONTROLLER

Ce code représente le contrôleur **RegistrationController** dans une application Symfony.
La méthode **register** est l'action invoquée lorsqu'un utilisateur souhaite s'inscrire sur l'application.
Elle gère la logique d'inscription de l'utilisateur, y compris la création du formulaire d'inscription.

Un nouvel objet **User** est créé pour représenter l'utilisateur en cours d'inscription, puis le formulaire est créé à partir de **RegistrationFormType** et de cet objet **User**.

La méthode **handleRequest** gère la demande **HTTP** associée à l'inscription de l'utilisateur et traite les données soumises par le formulaire.

Si le formulaire est valide, le mot de passe de l'utilisateur est haché à l'aide de **UserPasswordHasherInterface** avant d'être stocké dans la base de données via **EntityManagerInterface.**

La méthode **authenticateUser** authentifie l'utilisateur récemment inscrit à l'aide de **UserAuthenticatorInterface**.

Si le formulaire n'est pas soumis ou n'est pas valide, la méthode **render** génère la vue '**registration/register.html.twig**' pour afficher le formulaire d'inscription à l'utilisateur.

Mailer

SYMFONY

MAILER

Symfony Mailer est une bibliothèque puissante pour créer et envoyer des emails. Vous pouvez l'installer avec la commande suivante :

- **composer require symfony/mailer**

Pour envoyer des emails avec Symfony Mailer , vous devez utiliser un transporteur, un service de messagerie qui permet l'envoi et la réception d'emails. Le transporteur suit le protocole de communication Simple Mail Transfer Protocol (**SMTP**), un protocole qui définit les règles et les conventions pour l'acheminement des emails entre les serveurs de messagerie.

Quelque exemple de transporteur connu :

- Mailgun
- MailJet
- SendGrip
- Mandrill
- Amazon SES
- Gmail

SYMFONY

MAILER

Nous verrons ici un exemple de configuration avec Gmail. Il est important de noter que le transporteur Gmail ne doit être utilisé que pour la phase de développement. La configuration reste la même pour les autres transporteurs.

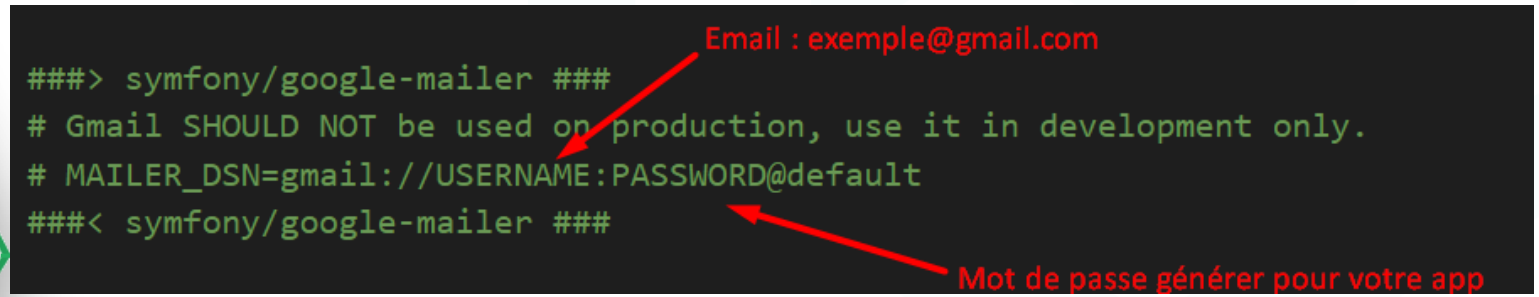
Installer google-mailer :

- **Composer require symfony/google-mailer**

Pour utiliser une adresse Gmail pour une application, l'adresse Gmail doit avoir la double authentification d'activé.
Il faut générer un mot de passe pour relier votre application à l'adresse Gmail grâce au lien suivant :

https://myaccount.google.com/apppasswords

Puis décommenter et modifier dans votre fichier .env la ligne suivante :



SYMFONY

ENVOI DE MAIL VIA UN CONTROLLER

Le **MailerInterface** est une interface définie dans **SymfonyMailer** qui fournit des méthodes pour envoyer des e-mails. En l'utilisant comme type d'argument dans la méthode index, Symfony va **automatiquement injecter une instance** de classe qui implémente cette interface, ce qui permet d'utiliser les fonctionnalités d'envoi d'e-mail.

```
// Définition de la route pour cette méthode du contrôleur
// La méthode index prend un objet MailerInterface en argument (injection de dépendance)
#[Route('/send', name: 'app_email_send')]
public function index(MailerInterface $mailer): Response
{
 $email = (new Email()) // Création d'un objet Email
     ->from('coursinsy2s@gmail.com') // Adresse e-mail de l'expéditeur
     ->to("coursinsy2s@gmail.com") // Adresse e-mail du destinataire
     ->subject("Mon premierMail avec symfony") // Objet du message
     ->text("ceci est un email envoyer via symfony"); // Corps du message contenant que du text
   //->html("<h2> Merci de pas répondre à cet email </h2>"); Corps du message peut contenir du html

 $mailer->send($email); // Envoi de l'e-mail à travers le service de messagerie
 return $this->render('email/index.html.twig', [ // Rendu d'un template Twig pour cette action
      'controller_name' => 'EmailController',
  ]);
}
```

SYMFONY

TEMPLATED EMAIL

Pour définir le contenu de votre e-mail avec Twig, utilisez la classe TemplatedEmail. Cette classe étend la classe Email normale mais ajoute de nouvelles méthodes pour les templates Twig :

```
#[Route('/send', name: 'app_email_send')]
public function index(MailerInterface $mailer): Response
{  // Création d'une instance de l'e-mail avec TemplatedEmail pour utiliser les templates Twig
   $email = (new TemplatedEmail())
     ->from("coursinsy2s@gmail.com") // Adresse e-mail de l'expéditeur
     ->subject("Mail avec un templatedEmail ! ") // Sujet de l'e-mail
     ->to("coursinsy2s@gmail.com") // Adresse e-mail du destinataire
     ->htmlTemplate("email/contact.html.twig") // Utilisation d'un template Twig pour le contenu HTML
      // Définition du contexte pour le template Twig (les variables données à la Template)
     ->context(
          [
          'userName' => "John Doe", // Exemple de variable à passer au template
          'message' => "Lorem ipsum dolor sit amet consectetur adipisicing elit !"
          ]);
      $mailer->send($email); // Envoi de l'e-mail à travers le service de messagerie
      return $this->render('email/index.html.twig', [
          'controller_name' => 'EmailController',
      ]);
}
```

SYMFONY

TEMPLATED EMAIL

Dans le fichier de modèle, vous pouvez personnaliser l'apparence de l'e-mail en utilisant Twig pour ajouter du HTML et du CSS. Vous pouvez également afficher dynamiquement le contenu en utilisant des variables Twig

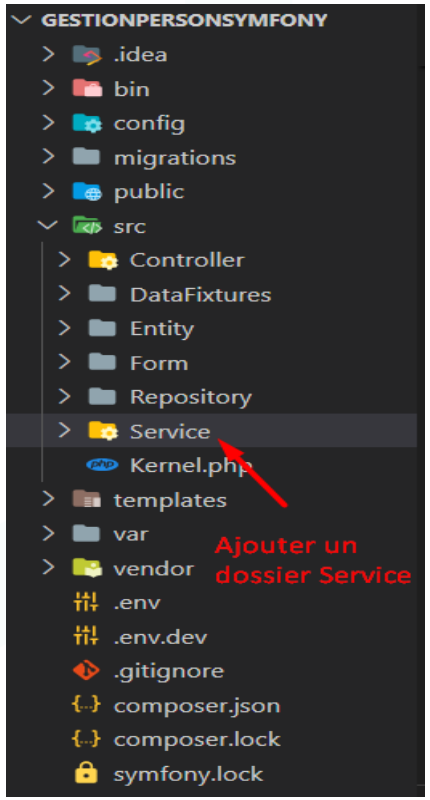
```
{% extends "base.html.twig" %}

{% block body %}

<h1>Demande de contact de {{ userName }}</h1>

<p>
    Message : {{ message }}
</p>

{% endblock %}
```

SYMFONY

CRÉATION D'UN SERVICE EMAIL

Dans Symfony, un **service** représente une unité modulaire de fonctionnalité **réutilisable** au sein de l'application. En créant un service d'envoi d'e-mails, nous **encapsulons** la logique d'envoi dans une **classe dédiée**, favorisant ainsi la modularité et la réutilisabilité du code. Cette approche permet une gestion centralisée des fonctionnalités de messagerie, améliorant ainsi la cohérence et la maintenance de l'application.

Pour garantir une architecture bien organisée, il est essentiel de créer un répertoire "Service" au sein du dossier "src" de votre application Symfony. Ce répertoire sera dédié à regrouper tous les services créés pour l'application, ce qui facilitera la gestion et la structuration du code.



SYMFONY

MAILSERVICE

Abordons la gestion des e-mails de manière structurée en utilisant un service dédié. Ce service, nommé MailService, encapsule la logique d'envoi d'e-mails dans une classe réutilisable

```php
class MailService
{   // Déclaration d'une propriété privée pour stocker l'interface du service de messagerie
    private MailerInterface $mailer;
    // Méthode de construction prenant en paramètre une instance de MailerInterface
    public function __construct(MailerInterface $mailer)
    {
        // Affectation de l'instance du service de messagerie à la propriété privée
        $this->mailer = $mailer;
    }
    // Méthode pour envoyer un e-mail avec les détails spécifiés
    public function sendMail(string $from, string $subject, string $template, array $context, string $to)
    {
        // Création d'un objet Email à partir du modèle TemplatedEmail
        $email = (new TemplatedEmail())
            ->from($from)
            ->subject($subject)
            ->to($to)
            ->htmlTemplate($template)
            ->context($context);
        // Envoi de l'e-mail en utilisant le service de messagerie
        $this->mailer->send($email);
    }
}
```

SYMFONY

MAILSERVICE

Cet exemple définit une classe MailService dans l'espace de noms App\Service. Cette classe est responsable de l'envoi d'e-mails à l'aide du composant Mailer de Symfony.

La classe prend un objet **MailerInterface** comme dépendance via son constructeur. Le constructeur initialise l'attribut $mailer avec cet objet.

La méthode **sendMail** est définie pour envoyer un e-mail avec les informations spécifiées. Les paramètres de la méthode incluent l'adresse e-mail de l'expéditeur, le sujet de l'e-mail, le modèle Twig à utiliser, le contexte des variables pour le modèle et l'adresse e-mail du destinataire.

À l'intérieur de la méthode **sendMail**, un nouvel e-mail est créé en utilisant TemplatedEmail de Twig. Les informations fournies sont utilisées pour configurer l'e-mail, puis il est envoyé en utilisant le service Mailer injecté.

Cette classe permet de centraliser la logique d'envoi d'e-mails dans un service réutilisable, ce qui favorise la maintenabilité et la modularité du code , ce qui permet réutiliser de la même méthode d'envois de mail pour différents mails.

SYMFONY

MAILSERVICE UTILISATION

Pour intégrer un service dans un contrôleur, vous pouvez procéder par injection de dépendance :

```
#[Route('/contact', name: 'app_contact')]
public function contact(MailService $mailService): Response
{

// Appel du service MailService pour envoyer un email
 $mailService->sendMail(

        "coursinsy2s@gmail.com", // Adresse email de l'expéditeur
        "mail via un service", // Sujet de l'email
        "email/contact.html.twig", // Template Twig à utiliser pour le contenu de l'email
         [
           'userName' => "John Doe", // Données à transmettre au template Twig
           'message' => "Lorem ipsum dolor sit amet consectetur adipisicing elit.lib Atque!" // Données à
transmettre au template Twig
         ],
         "coursinsy2s@gmail.com" // Adresse email du destinataire

       );

 // Rendu d'un template Twig pour cette action
 return $this->render('email/index.html.twig', [
    'controller_name' => 'EmailController',
 ]);
}
```