**Installation Steup**

**1)Créer un projet symfony:**

symfony new devwell --version="7.1.\*"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2)Installer les dépandances:**

**Installer Twig pour les templates :**

composer require symfony/twig-bundle

**Installer Doctrine (ORM) pour la gestion de la base de données :**

composer require symfony/orm-pack

**Installer Symfony Security Bundle pour la gestion des utilisateurs et de l’authentification :**

composer require symfony/security-bundle

**Installer Symfony Form et Validator pour la gestion des formulaires et des validations :**

composer require symfony/form symfony/validator

**Installer Symfony Asset Bundle pour la gestion des assets (CSS, JS, images) :**

composer require symfony/asset

**Installer PHPUnit pour les tests :**

composer require --dev phpunit/phpunit

**Installer le Profiler Pack pour le débogage :**

composer require symfony/profiler-pack

**Installer Symfony Maker Bundle pour la génération de code :**

composer require symfony/maker-bundle --dev

composer require symfony/maker-bundle --dev –with-all-dependencies

**2bis)Installer les dépandances:**

**Tailwind :**

composer require symfonycasts/tailwind-bundle

php bin/console tailwind:init

php bin/console tailwind:build --watch

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3)Configuration de la base de données :**

Dans le fichier .env, configure ta connexion SQL (par exemple, pour MySQL) :

DATABASE\_URL="mysql://root:@127.0.0.1:3306/devwell\_db"

**Créer la base de données :**

php bin/console doctrine:database:create

### ****4) Configurer le routing et les contrôleurs de base :****

1)Créer un controller et sa créer la twig en automatique

php bin/console make:controller HomeController

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### ****5) Test de fonctionnement :****

1. **Lancer le serveur Symfony :** Pour vérifier que tout fonctionne, lance le serveur Symfony en utilisant la commande suivante :
2. symfony server:start

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**6) Créer son repo git et le connecter**

Et le connecter à son repo github

git init

git add .

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/ilyaasAS/devwell.git

git push -u origin main

**Mettez à jour régulièrement**

git add .

git commit -m "Home"

git push

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Etape 1 : Créer l’inscription**

**Créez l'entité** User en utilisant la commande Symfony suivante :

php bin/console make:entity User

firstName (string)

lastName (string)

email (string)

password (string)

### 2 : Créer un formulaire pour l'inscription

1. **Utilisez la commande Symfony pour générer un formulaire** :

php bin/console make:form UserType

**3 ) Créer le contrôleur pour l'inscription**

php bin/console make:controller RegistrationController

et sa créer la twig

php bin/console doctrine:migrations:diff

php bin/console doctrine:migrations:migrate

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Etape 2 : Créer l’Authentification**

2. Créer l’authenticator

Générez un authenticator avec Symfony Maker pour gérer la connexion :

php bin/console make:auth

3. Créer le formulaire de connexion

templates/security/login.html.twig

LoginType.php

4. Créer le formulaire de connexion : configuer le security.yaml

https://main--zen-jepsen-04a701.netlify.app/posts/6.symfony6-authentification.article

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Etapes 3 : Faire ma NavBar**

Créer un dossier et le composant :

components/navbar.html.twig

base.html.twig :

<!-- navbar -->

        {% include 'components/navbar.html.twig' %}

#### **JavaScript**

Ajoutez ce script juste avant la fermeture du tag </body> pour activer/désactiver l'affichage du menu mobile lorsque vous appuyez sur le bouton burger.

<script>

    // Récupérer le bouton et le menu

    const menuToggle = document.getElementById('menu-toggle');

    const mobileMenu = document.getElementById('mobile-menu');

    // Ajouter un gestionnaire d'événement pour le bouton

    menuToggle.addEventListener('click', () => {

        // Basculer la classe `hidden` pour afficher/masquer le menu

        mobileMenu.classList.toggle('hidden');

    });

</script>

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Etapes 4: Faire mon Footer :**

Créer un dossier et le composant :

components/footer.html.twig

base.html.twig :

<!-- Footer -->

        {% include 'components/footer.html.twig' %}

faire les pages du footer :

php bin/console make:controller CGVController

php bin/console make:controller CGUController

php bin/console make:controller MentionLegaleController

php bin/console make:controller ContactControlle

et sa créer la twig

attention au nom des routes

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Etapes 5: Faire ma page Admin :**

### 1) Générer un CRUD pour la gestion des utilisateurs

Puisque vous voulez une gestion simplifiée du profil, nous pouvons utiliser la commande make:crud pour générer les bases.

php bin/console make:crud User

Cela va générer :

* Un contrôleur UserController.
* Des vues Twig pour l'ajout, la modification, l'affichage et la suppression des utilisateurs.
* Un formulaire préconfiguré pour gérer l'entité User.

### 2 : Adapter le CRUD pour la gestion du profil

1. **Restreindre les actions au profil utilisateur connecté** : Modifiez le fichier UserController.php pour que les actions ne concernent que l'utilisateur connecté. Ajoutez les vérifications suivantes dans les méthodes concernées (par exemple edit).

Exemple pour la méthode edit :

UserController.php

templates/user/\_delete\_form.html.twig

templates/user/\_form.html.twig

templates/user/edit.html.twig

templates/user/index.html.twig

templates/user/show.html.twig

configurer le fichier security.yaml :

 access\_control:

        - { path: ^/user, roles: ROLE\_ADMIN }

la route réserver pour l’admin

#[Route('/user')]

final class UserController extends AbstractController

{

    #[Route(name: 'app\_user\_index', methods: ['GET'])]

    public function index(UserRepository $userRepository): Response

    {

        return $this->render('user/index.html.twig', [

            'users' => $userRepository->findAll(),

        ]);

    }

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Etapes 6: Faire ma page profil:**

### 1. ****Créer un contrôleur pour le profil utilisateur (ProfileController)****

Dans le cas où vous souhaitez créer une page de profil spécifique pour chaque utilisateur où il pourra modifier ses informations, vous devez d'abord créer un contrôleur dédié. Vous pouvez le faire avec la commande Symfony suivante :

php bin/console make:controller ProfileController :

<?php

namespace App\Controller;

use App\Form\ProfileType;

use Doctrine\ORM\EntityManagerInterface;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

use Symfony\Bundle\SecurityBundle\Security as SecurityBundleSecurity;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\Routing\Attribute\Route;

use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class ProfileController extends AbstractController

{

    #[Route('/profile', name: 'app\_profile', methods: ['GET', 'POST'])]

    public function edit(

        Request $request,

        EntityManagerInterface $entityManager,

        SecurityBundleSecurity $security,

        UserPasswordHasherInterface $passwordHasher

    ): Response {

        // Récupérer l'utilisateur connecté

        $user = $security->getUser();

        if (!$user instanceof \App\Entity\User) {

            // Rediriger si l'utilisateur n'est pas connecté ou n'est pas une instance valide

            return $this->redirectToRoute('app\_login');

        }

        // Sauvegarder l'ancien mot de passe

        $currentPassword = $user->getPassword();

        // Créer le formulaire

        $form = $this->createForm(ProfileType::class, $user);

        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {

            // Récupérer le mot de passe brut du formulaire

            $plainPassword = $form->get('password')->getData();

            if ($plainPassword) {

                // Si un nouveau mot de passe est défini, le hacher et l'assigner

                $hashedPassword = $passwordHasher->hashPassword($user, $plainPassword);

                $user->setPassword($hashedPassword);

            } else {

                // Si aucun mot de passe n'est fourni, conserver l'ancien

                $user->setPassword($currentPassword);

            }

            // Sauvegarder les modifications

            $entityManager->flush();

            $this->addFlash('success', 'Profile updated successfully.');

            return $this->redirectToRoute('app\_profile');

        }

        return $this->render('profile/edit.html.twig', [

            'form' => $form->createView(),

        ]);

    }

}

### ****ProfileType.php :****

<?php

namespace App\Form;

use App\Entity\User;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\FormBuilderInterface;

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

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

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

use Symfony\Component\OptionsResolver\OptionsResolver;

use Symfony\Component\Validator\Constraints\NotBlank;

class ProfileType extends AbstractType

{

    public function buildForm(FormBuilderInterface $builder, array $options): void

    {

        $builder

            ->add('firstName', TextType::class, [

                'label' => 'First Name',

                'attr' => ['class' => 'form-control']

            ])

            ->add('lastName', TextType::class, [

                'label' => 'Last Name',

                'attr' => ['class' => 'form-control']

            ])

            ->add('email', EmailType::class, [

                'label' => 'Email',

                'attr' => ['class' => 'form-control']

            ])

            ->add('password', PasswordType::class, [

                'required' => false,

                'label' => 'Password',

                'attr' => ['class' => 'form-control'],

                'empty\_data' => '', // Si le champ est laissé vide, on assigne une chaîne vide

                'constraints' => [

                    // Supprimez NotBlank ici car le champ peut être vide

                ],

            ]);

    }

    public function configureOptions(OptionsResolver $resolver): void

    {

        $resolver->setDefaults([

            'data\_class' => User::class,

        ]);

    }

}

### 3. ****Créer la vue Twig pour le profil (edit.html.twig)****

Ensuite, vous pouvez créer une vue pour ce profil dans templates/profile/edit.html.twig :

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

{% block title %}Edit Profile{% endblock %}

{% block body %}

    <div class="container mx-auto py-10">

        <h1 class="text-3xl font-bold mb-6">Edit Profile</h1>

        <div class="bg-white p-6 rounded shadow-md">

            {{ form\_start(form, {'attr': {'class': 'space-y-4'}}) }}

                {{ form\_row(form.firstName, {'attr': {'class': 'w-full border border-gray-300 px-4 py-2 rounded'}}) }}

                {{ form\_row(form.lastName, {'attr': {'class': 'w-full border border-gray-300 px-4 py-2 rounded'}}) }}

                {{ form\_row(form.email, {'attr': {'class': 'w-full border border-gray-300 px-4 py-2 rounded'}}) }}

                {{ form\_row(form.password, {'attr': {'class': 'w-full border border-gray-300 px-4 py-2 rounded'}}) }}

                <button class="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600">Update Profile</button>

            {{ form\_end(form) }}

        </div>

        <div class="mt-4">

            <a href="{{ path('app\_home') }}" class="text-blue-500 hover:underline">Back to Home</a>

        </div>

    </div>

{% endblock %}

### 5. ****Mettre à jour le fichier****security.yaml

Vérifiez que votre fichier security.yaml protège correctement l'accès à la page du profil et à d'autres parties de votre application, en vérifiant si l'utilisateur est connecté et a les rôles appropriés.

Exemple d'une restriction d'accès :

security:

# ...

access\_control:

- { path: ^/profile, roles: ROLE\_USER } # L'utilisateur doit être connecté pour accéder à son profil

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Option ajout de produit dans ma page d’admin**

#### 2. ****Création de l'entité****Category

* Créez l'entité Category pour définir les catégories des produits.

php bin/console make:entity Category

Ajoutez un champ name à l'entité Category de type string.

1. **Création de l'entité**Product

* Créez l'entité Product pour définir les produits que vous souhaitez ajouter.

php bin/console make:entity Product

* Ajoutez les champs suivants à l'entité Product :
  + name (type: string)
  + price (type: float)
  + stock (type: integer)
  + image (type: string) – pour stocker le chemin vers l'image.
  + description (type: text)
  + category (type: ManyToOne relation avec l'entité Category).

#### 3. ****Création du CRUD pour****Product

* Utilisez la commande suivante pour générer le CRUD pour l'entité Product.

php bin/console make:crud Product

* Cela créera un contrôleur ProductController, des vues Twig et un formulaire pour ajouter, modifier et lister les produits.

#### 4. ****Création du CRUD pour****Category

* Générer également un CRUD pour l'entité Category :

php bin/console make:crud Category

#### 5. ****Formulaire d'ajout de produit****

* Dans ProductType.php, ajoutez les champs pour créer ou éditer un produit. Exemple :

// src/Form/ProductType.php

namespace App\Form;

use App\Entity\Product;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\FormBuilderInterface;

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

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

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

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

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

use Symfony\Component\OptionsResolver\OptionsResolver;

class ProductType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options): void

{

$builder

->add('name', TextType::class)

->add('price', NumberType::class)

->add('stock', NumberType::class)

->add('image', FileType::class, ['required' => false])

->add('description', TextAreaType::class)

->add('category') // Relation ManyToOne avec Category

->add('save', SubmitType::class);

}

public function configureOptions(OptionsResolver $resolver): void

{

$resolver->setDefaults([

'data\_class' => Product::class,

]);

}

}

Dans votre contrôleur AdminController.php, créez une action pour ajouter un produit.

Exemple d'action dans AdminController.php :

#[Route('/admin/product/new', name: 'app\_admin\_product\_new')] public function new(Request $request, EntityManagerInterface $entityManager): Response { $product = new Product(); $form = $this->createForm(ProductType::class, $product); $form->handleRequest($request); if ($form->isSubmitted() && $form->isValid()) { $entityManager->persist($product); $entityManager->flush(); return $this->redirectToRoute('app\_admin\_product\_list'); } return $this->render('admin/product\_new.html.twig', [ 'form' => $form->createView(), ]); }

#### 7. ****Page d'affichage des produits****

Exemple pour afficher les produits dans templates/admin/product\_index.html.twig :

<h1>Liste des Produits</h1>

<table class="table">

<thead>

<tr>

<th>Nom</th>

<th>Prix</th>

<th>Stock</th>

<th>Description</th>

<th>Catégorie</th>

<th>Actions</th>

</tr>

</thead>

<tbody>

{% for product in products %}

<tr>

<td>{{ product.name }}</td>

<td>{{ product.price }}</td>

<td>{{ product.stock }}</td>

<td>{{ product.description }}</td>

<td>{{ product.category.name }}</td>

<td>

<a href="{{ path('app\_product\_edit', {'id': product.id}) }}">Edit</a>

<a href="{{ path('app\_product\_delete', {'id': product.id}) }}" class="text-red-500">Delete</a>

</td>

</tr>

{% endfor %}

</tbody>

</table>

**Option ajout de catégoris dans ma page d’admin**

**Formulaire de contact**

php bin/console make:entity

Contact.php

ContactRepository.php

php bin/console make:form

1.Créer le Formulaire (FormType) :

php bin/console make:controller ContactController

2. Créer un Contrôleur pour le Formulaire :

3. Créer la Vue (Template Twig) :

templates/contact/index.html.twig :

Créer la migration :

php bin/console make:migration

Exécuter la migration :

php bin/console doctrine:migrations:migrate

**Ajout de panier**

Oui, vous avez identifié les fichiers principaux pour documenter la fonctionnalité "Ajout au panier". Voici ce que chaque fichier représente dans ce contexte et ce que vous pouvez documenter :

### 1. Cart.php (Entity)

* Rôle :  
  Représente l'entité Cart, qui stocke les données liées à un article ajouté au panier pour un utilisateur.
* À documenter :
  + Les attributs principaux (par ex. id, user, product, quantity).
  + Les relations (par ex. ManyToOne avec User et Product).
  + Les contraintes de validation sur les propriétés (par ex. @Assert\Positive sur quantity).
  + Les méthodes (par ex. setQuantity(), getTotalPrice()).

### 2. CartRepository.php (Repository)

* Rôle :  
  Gère les interactions avec la base de données pour l'entité Cart.
* À documenter :
  + Les méthodes spécifiques, par exemple :
    - findByUser(User $user) : Récupère les articles du panier pour un utilisateur donné.
    - Méthodes pour des opérations courantes ou personnalisées (comme calculer le total global).

Oui, vous avez identifié les fichiers principaux pour documenter la fonctionnalité "Ajout au panier". Voici ce que chaque fichier représente dans ce contexte et ce que vous pouvez documenter :

### 1. Cart.php (Entity)

* Rôle :  
  Représente l'entité Cart, qui stocke les données liées à un article ajouté au panier pour un utilisateur.
* À documenter :
  + Les attributs principaux (par ex. id, user, product, quantity).
  + Les relations (par ex. ManyToOne avec User et Product).
  + Les contraintes de validation sur les propriétés (par ex. @Assert\Positive sur quantity).
  + Les méthodes (par ex. setQuantity(), getTotalPrice()).

### 2. CartRepository.php (Repository)

* Rôle :  
  Gère les interactions avec la base de données pour l'entité Cart.
* À documenter :
  + Les méthodes spécifiques, par exemple :
    - findByUser(User $user) : Récupère les articles du panier pour un utilisateur donné.
    - Méthodes pour des opérations courantes ou personnalisées (comme calculer le total global).

### 3. CartController.php (Controller)

* Rôle :  
  Contient les actions pour gérer le panier : ajouter, mettre à jour, afficher, ou supprimer des articles.
* À documenter :
  + Routes principales, par exemple :
    - /cart/add/{id} : Ajouter un produit au panier.
    - /cart/update/{id} : Mettre à jour la quantité d’un produit dans le panier.
    - /cart/remove/{id} : Supprimer un produit du panier.
  + Les paramètres des méthodes et leur logique (par ex. validation, gestion des erreurs).
  + Les interactions avec le modèle et la vue (par ex. récupération des données via le repository).

### 4. cart/index.html.twig (Template)

* Rôle :  
  Gère l’affichage du panier dans l’interface utilisateur.
* À documenter :
  + La structure HTML/CSS pour afficher les articles (ex. boucle cartItems).
  + Les formulaires pour mettre à jour les quantités ou supprimer un article.
  + L'affichage des totaux (totalItems, totalPrice).
  + Les interactions possibles avec le contrôleur (via les chemins générés par path()).

### 1. src/Service/CartService

* Rôle :  
  Centralise la logique du panier pour éviter de répéter le code dans les contrôleurs ou les templates.
* À documenter :
  + Méthodes principales :
    - getTotalItems(User $user): int :  
      Retourne le nombre total d’articles dans le panier d’un utilisateur. Vous pouvez expliquer la logique de calcul, par exemple :

public function getTotalItems(User $user): int

{

$cartItems = $this->cartRepository->findBy(['user' => $user]);

$totalItems = 0;

foreach ($cartItems as $item) {

$totalItems += $item->getQuantity();

}

return $totalItems;

}

Autres méthodes associées (par exemple getCartItems, getTotalPrice) si elles sont utilisées ailleurs.

* Injection de dépendances : Expliquez comment le service utilise le CartRepository ou d’autres services via injection.
* Utilisation dans les contrôleurs ou dans un event listener :
* Si ce service est appelé dans un listener (par ex. pour rendre les variables globales accessibles aux templates), mentionnez le KernelEvent.

### 2. navbar.html.twig

* Rôle :  
  Affiche un badge dynamique avec le nombre total d’articles dans le panier.
* À documenter :
  + Code Twig : Expliquez le rôle du code utilisé, par exemple

<a href="{{ path('cart\_view') }}" class="relative">

<span class="badge">{{ totalItems }}</span> <!-- Nombre total d'articles -->

<i class="fas fa-shopping-cart"></i>

</a>

**Gestion des message du formulaire de contact:**

ContactController.php :

#[Route('/admin/messages', name: 'admin\_messages')]

    public function adminMessages(EntityManagerInterface $em): Response

    {

        // Récupérer tous les messages de contact

        $messages = $em->getRepository(Contact::class)->findBy([], ['createdAt' => 'DESC']);

        return $this->render('contact/messages.html.twig', [

            'messages' => $messages,

        ]);

    }

    #[Route('/admin/messages/delete/{id}', name: 'admin\_message\_delete')]

public function deleteMessage(int $id, EntityManagerInterface $em): Response

{

    // Récupérer le message à supprimer

    $message = $em->getRepository(Contact::class)->find($id);

    if ($message) {

        $em->remove($message);

        $em->flush();

        $this->addFlash('success', 'Le message a été supprimé.');

    } else {

        $this->addFlash('error', 'Le message n\'a pas été trouvé.');

    }

    return $this->redirectToRoute('admin\_messages');

}

#[Route('/admin/messages/view/{id}', name: 'admin\_message\_view')]

public function viewMessage(int $id, EntityManagerInterface $em): Response

{

    // Récupérer le message spécifique par son ID

    $message = $em->getRepository(Contact::class)->find($id);

    if (!$message) {

        // Si le message n'existe pas, afficher une erreur

        $this->addFlash('error', 'Le message n\'existe pas.');

        return $this->redirectToRoute('admin\_messages');

    }

    // Rendre la vue avec le message

    return $this->render('contact/message\_detail.html.twig', [

        'message' => $message,

    ]);

}

faire les twig :

contact/message\_detail.html.twig

contact/messages.html.twig

Mettre le lien sur navBar :

  <!-- Afficher Dashboard seulement si l'utilisateur a le rôle 'ROLE\_ADMIN' -->

                    {% if is\_granted('ROLE\_ADMIN') %}

                        <a href="/user" class="text-lg font-bold hover:text-secondary\_dw">Dashboard</a>

                        <a href="/product" class="text-lg font-bold hover:text-secondary\_dw">Gestion de Produit</a>

                        <a href="/categories" class="text-lg font-bold hover:text-secondary\_dw">Gestion de Catégorie</a>

                    <a href="/admin/messages" class="block py-2 hover:text-secondary\_dw">Gestion de message</a>

                    {% endif %}

<!-- Afficher Dashboard seulement si l'utilisateur a le rôle 'ROLE\_ADMIN' -->

            {% if is\_granted('ROLE\_ADMIN') %}

                <li>

                    <a href="/user" class="block py-2 hover:text-secondary\_dw">Dashboard</a>

                </li>

                <li>

                    <a href="/product" class="block py-2 hover:text-secondary\_dw">Gestion de Produit</a>

                </li>

                <li>

                    <a href="/categories" class="block py-2 hover:text-secondary\_dw">Gestion de Catégorie</a>

                </li>

                <li>

                    <a href="/admin/messages" class="block py-2 hover:text-secondary\_dw">Gestion de message</a>

                </li>

            {% endif %}

### Étapes principales pour ajouter la fonctionnalité de réponse aux messages

#### 1. Ajouter un champ de réponse dans l'entité Contact

Nous devons d'abord ajouter un champ response dans l'entité Contact afin de pouvoir stocker la réponse de l'administrateur.

* Ce que tu dois faire :
  + Ajouter la propriété response dans l'entité Contact.
  + Ajouter le getter et setter pour la propriété response.

#[ORM\Column(type: "text", nullable: true)]

private ?string $response = null;

public function getResponse(): ?string

{

return $this->response;

}

public function setResponse(?string $response): self

{

$this->response = $response;

return $this;

}

#### 2. Mettre à jour le formulaire ContactType

Nous allons maintenant mettre à jour le formulaire pour inclure un champ de réponse, mais uniquement visible pour l'administrateur (pas pour l'utilisateur qui soumet un message).

* Ce que tu dois faire :
  + Ajouter un champ response dans le formulaire de contact, mais en le rendant nullable et en le configurant pour qu'il ne soit pas lié à l'entité Contact (mapped => false).

->add('response', TextareaType::class, [

'label' => 'Réponse',

'required' => false,

'attr' => ['class' => 'form-control', 'rows' => 5],

'mapped' => false, // Ce champ n'est pas lié à l'entité Contact

])

#### 3. Créer une route et un contrôleur pour répondre au message

Ensuite, nous devons créer une route et une méthode dans le contrôleur pour afficher un message spécifique et permettre à l'administrateur d'y répondre.

* Ce que tu dois faire :
  + Créer une méthode dans ton contrôleur qui affichera un message spécifique et permettra à l'administrateur de le répondre.
  + Utiliser form->handleRequest($request) pour traiter la soumission de la réponse.

#[Route('/admin/messages/view/{id}', name: 'admin\_message\_view')]

public function viewMessage(int $id, Request $request, EntityManagerInterface $em): Response

{

// Récupérer le message spécifique par son ID

$message = $em->getRepository(Contact::class)->find($id);

if (!$message) {

$this->addFlash('error', 'Message introuvable');

return $this->redirectToRoute('admin\_messages');

}

// Créer le formulaire pour la réponse

$form = $this->createForm(ContactType::class, $message);

$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid()) {

// Enregistrer la réponse

$response = $form->get('response')->getData();

$message->setResponse($response);

$em->flush();

$this->addFlash('success', 'Réponse envoyée avec succès');

return $this->redirectToRoute('admin\_message\_view', ['id' => $message->getId()]);

}

return $this->render('admin/message\_detail.html.twig', [

'message' => $message,

'form' => $form->createView(),

]);

}

#### 4. Afficher le formulaire de réponse dans le template

Ensuite, il faut afficher un formulaire pour l'administrateur dans une vue dédiée où il pourra répondre à un message.

* Ce que tu dois faire :
  + Créer une page message\_detail.html.twig où le formulaire de réponse sera affiché.
  + Ajouter les informations du message (nom, email, sujet, etc.) et un champ pour la réponse.

{{ form\_start(form) }}

<div class="form-group">

{{ form\_label(form.response) }}

{{ form\_widget(form.response) }}

{{ form\_errors(form.response) }}

</div>

<button type="submit">Envoyer la réponse</button>

{{ form\_end(form) }}

**Pagination :**

Je veux que tu me met une Pagination sur la page produit: