

SYMFONY 7

Application E-COMMERCE

C'est un framework PHP, qui offre un cadre de travail PHP haute performance. Il fournit une structure et des composants réutilisables qui vous aideront à créer des applications web robustes et évolutives.

Il vous faudra connaître certains concepts basiques tels que les fonctions, des notions en base de données MySQL car nous utiliserons celle-ci pour la base de données, une bonne connaissance de HTML, du CSS et du JavaScript, ainsi que le terminal de commande. Évidemment il vous faudra exploiter la documentation officielle, et ne pas hésiter à effectuer des recherches par vous-même car c'est une grosse partie de ce métier et dites-vous bien qu'il y a forcément quelqu'un dans le monde qui a déjà été confronté aux mêmes problèmes, donc on cherche.

Tout d'abord vous aurez besoin sur votre ordinateur pour pouvoir utiliser Symfony 7 de :

- PHP 8.2 ou supérieur
- Composer (gestionnaire de dépendances)
- Symfony CLI
- Wamp pour un serveur web local
- Système de gestion de base de données (Wamp contient tout cela)
- Git (et donc un compte GitHub et relier à votre Vs code de très bon tutos existe)

Là vous serez opérationnel pour commencer à travailler en Symfony. Vous allez sur Symfony et on clique sur Download pour sur amd64 qui est le moyen le plus rapide de faire cela (install CLI symfony). Vous allez créer un dossier sur votre disque C, et vous l'appellerez disons Symfony pour faire simple, et vous glisser le .exe dans ce dossier. Vous copier l'emplacement de ce fichier et vous rendez dans les variables d'environnement, Ce PC => click droit Propriétés => Paramètres systèmes avancées => Variable d'environnement => Variable système => Path et modifier

Variables système	
Variable	Valeur
ChocolateyInstall	C:\ProgramData\chocolatey
ComSpec	C:\WINDOWS\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
MAVEN_HOME	C:\apache-maven-3.9.2
NUMBER_OF_PROCESSORS	8
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS COMSPEC=C:\Windows\system32\cmd.exe; PATHTOEXEC=C:\Windows\system32

Cliquez sur nouveau et vous collez le chemin copier précédemment => Ok et encore Ok jusqu'à fermer toutes les fenêtres grâce à l'OK. Puis on test cela dans le terminal en tapant Symfony simplement. Il va vous afficher la version ainsi que les commandes possibles donc c'est que tout va bien. (Tout ceci est pour le système Windows).

Ensuite vous taper Git sur votre moteur de recherche et vous installez cela, ensuite on va passer sur wamp, vous taper WampServer sur votre moteur de recherche et vous installer, vous aurez déjà

installer PHP avec la dernière version, ainsi que PHP myAdmin pour la gestion des base de données ainsi que votre serveur web apache. Et vous revérifier la version, si jamais il vous dit que la commande php n'est pas reconnu il vous faudra chercher le chemin et recréer une variable d'environnement. Il vous faudra installer Composer et nous serons prêt à commencer, à vous de jouer, pensez à vérifier sur le terminal ensuite.

Passons à l'installation de Symfony, rendez-vous sur <https://symfony.com/doc/current/index.html>, et cliquez sur setup/installation, et cherchez **Creating Symfony Applications**. Vu ce que vous venez d'installer vous pouvez utiliser celle qui utilise la commande Symfony.

Vous allez choisir l'endroit où vous désirez créer votre projet, moi je me mettrai dans le dossier Wamp => dossier www => et je lance le terminal puis créer mon projet. Il va vous créer plusieurs dossiers que nous allons voir ensemble.

Ok passons au détail des dossiers de notre projet, car il contient déjà de nombreux dossiers que nous n'avons pas créé nous-même, mais qui sont nécessaires, faisons un petit tour ensemble. Prenons les dans l'ordre :

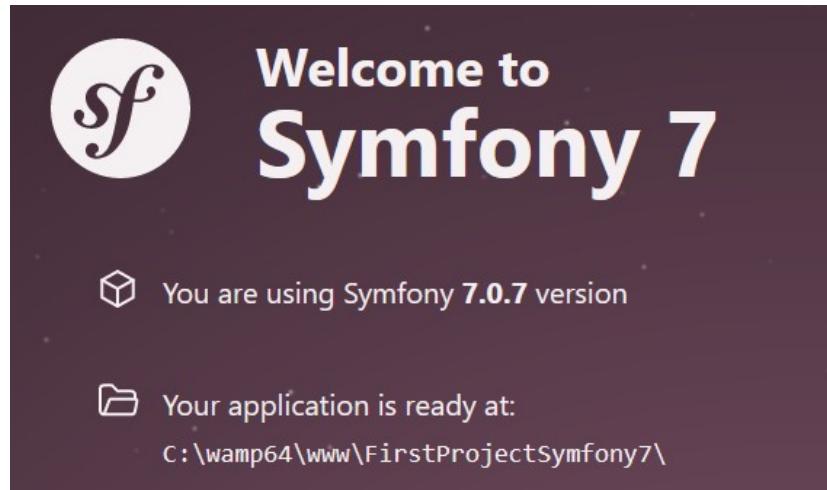
- assets : contient le style, ainsi que quelques fichier js
- bin : on ira jamais dedans mais indirectement oui car on va taper beaucoup de ligne de commande et qu'il contient la console
- config : ben son nom l'indique c'est ici que l'on gérera la config du projet
- migrations : nous avons un gitignore ici , et on pourra gérer les migration doctrine
- public : c'est le seul point d'entrée de l'appli, le file index.php que l'on ne touche surtout pas, mais on pourra ajouter des fichiers comme le css, les images, des docs pdf si besoin etc..
- src : c'est **LE DOSSIER** qui nous intéresse le plus car tout le code que l'on va taper on y passera 70% de notre temps
- templates : et bien c'est ici que l'on créera toutes nos pages de **VUE**, notre rendu visuel de notre projet
- tests : et bien c'est pour les test unitaires
- translation : c'est pour tout ce qui est fichier de traduction en différentes langues
- var : c'est un dossier qui se recharge a chaque fois que l'on va lancer notre projet, il contient les caches(on peut donc effacer le dossier cache a notre convenance vu qu'il revient)
- vendor : celui-ci aussi on y touchera jamais il contient toutes les dépendances dont a besoin symfony
- le fichier .env que l'on va utiliser dès le début car c'est ici que l'on paramètre l'accès à la base de donnée que l'on utilise pour nos projets
- ensuite les composer.json et .lock je vais pas trop rentrer dans les détails mais il ne faut que très très rarement toucher à ses fichiers sauf en cas de gros problèmes parce qu'il liste tout ce que l'on utilise dans notre app.

Installons quelques plugins très intéressants : (vous avez le choix de le faire ou pas)

- Color Highlight
- Format HTML in PHP
- Icon Fonts
- PHP DocBlocker
- PHP Intelephense
- Php namespaces resolver
- Symfony Extension Pack

Les Controllers

Un **Controller** est une **fonction PHP** que l'on crée pour lire les infos à partir de l'objet **request** et créer et renvoyer un objet **response**. Ils sont responsables de la logique métier de votre application. Il gère également le **routing**, c'est un mécanisme qui permet de définir comment les URL sont associés à des actions spécifiques dans votre application. Il est configuré dans un fichier de config appelé **routes.yaml**. Ensuite vous allez pouvoir lancer votre projet, en tapant la commande **symfony server:start**, ou bien **symfony serve -d**, pour ceux qui utilisent WAMP vous pouvez utiliser également **php -S localhost:8000 -t public** vous allez avoir une page qui s'ouvre ou un http sur lequel cliquer et ouvrir la page Symfony,



La page est bien ouverte sur le navigateur ce que vous voyez à l'écran est la page par défaut de Symfony. Voyons comment créer un **Controller**, une page étant une **vue**, une **vue** ne peut être que délivrée par un **Controller**, donc on est obligé de créer un **Controller** avant logiquement. On va faire ça en ligne de commande, vous pouvez si vous taper **symfony console list make** (on rentre dans le dossier bin et on a accès à la console) vous pourrez avoir accès à toutes les commandes possibles, la catégorie make va être la plus intéressante pour nous car c'est celle qui va nous permettre de construire ce dont on a besoin. Si besoin vous pouvez vider votre terminal en tapant **clear** et entrée. Créons donc notre premier **Controller**, donc **symfony console make:controller** puis le nom du **Controller** donc **HomePageController** (on met le mot **Controller** à la fin car c'est d'usage de faire comme cela). Cela aura créé un fichier dans le dossier **Controller** comme indiqué et également la vue qui va avec dans le dossier **templates** il aura créé un dossier qui porte le nom du **Controller** ainsi que son fichier **twig** qui est le rendu de votre page.

```
created: src/Controller/HomePageController.php
created: templates/home_page/index.html.twig
```

On va aller voir cela ensemble et les détailler un peu,

```

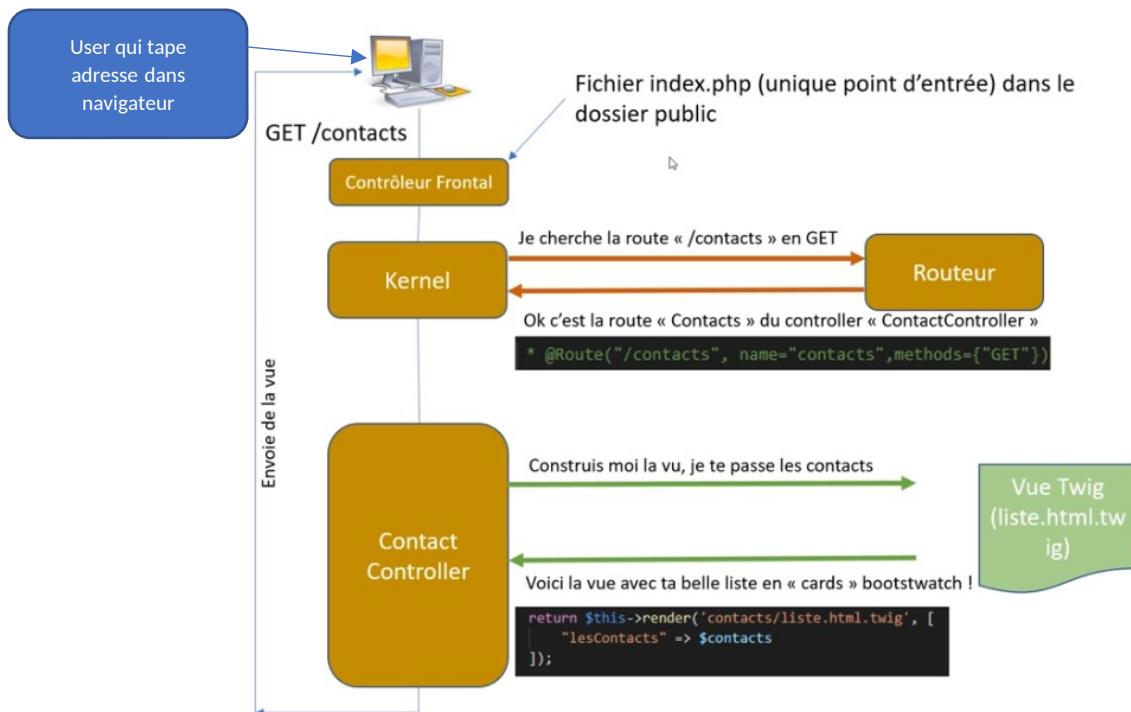
class HomePageController extends AbstractController
{
    #[Route('/home/page', name: 'app_home_page')]
    public function index(): Response
    {
        return $this->render('home_page/index.html.twig', [
            'controller_name' => 'HomePageController',
        ]);
    }
}

```

On a donc une classe qui **extends** une class générée par Symfony, ensuite la route donc /home/page sera la route noté dans le navigateur, le **name** sera lui utilisé pour accéder à cette route partout dans votre application (comme un lien). Ensuite la fonction qui renvoie bien une réponse, puis la méthode **return** qui render la page concernée(homepage). Vous pouvez décider de définir cette page en page d'accueil en laissant '/' dans la route pour quelle soit la page par défaut, recharger votre navigateur et vous verrez votre première page à l'écran.

Les Vues

La vue est l'élément clé de la structure MVC utilisé par Symfony, elle est responsable de la représentation des données à l'utilisateur final. Elle est chargée de générer le contenu HTML qui sera affiché dans le navigateur. Elle est représentée par un fichier twig, qui a le moteur de template utilisé par Symfony. Voici la logique entre **Controller** et **Twig**,



Vous allez ajouter un titre à votre page d'accueil, pour ceux qui veulent vous pouvez installer Bootstrap pour gagner du temps sur le style. Pour Bootstrap vous vous rendez sur la doc, vous

descendez et récupérer les liens CDN et vous les collez dans la base.html qui est la base de votre appli, je vais également coller le lien Js en bas du body. Actualiser votre navigateur et vous verrez déjà une différence rien que sur le titre. Pour voir les routes que vous avez créé ainsi que celle déjà existante vous pouvez taper en terminal **symfony console debug:router**. Voyons les fichiers twig qui sont les vues, pour faire simple twig fonctionne sous forme de bloc. N'hésitez pas à regarder la doc de twig, <https://twig.symfony.com/doc/>.

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

{% block title %}Hello HomePageController!{% endblock %}

{% block body %}

<div class="example-wrapper">
    <h1>Accueil</h1>
</div>
{% endblock %}
```

A savoir que dans un fichier **twig**, vous pouvez écrire des conditions, des variables, etc..

```
{% if %}
    {% elseif %}
        {% else %}
    {% endif %}
```

La syntaxe sera toujours celle-ci, `{% %}`, pas de panique nous allons l'utiliser régulièrement, vous allez tester cela de suite, vous allez vous rendre dans votre **Controller** et créer une variable qui aura pour valeur un array contenant des prénoms et une seconde qui aura pour valeur un chiffre, un âge.

```
#Route('/', name: 'app_home_page')
public function index(): Response
{
    $nomsStudents=['Jérémie', 'Ousmane', 'Alexia', 'Chouaibou'];
    $age =17;

    return $this->render('home_page/index.html.twig', [
        // 'controller_name' => 'HomePageController',
        'lesNoms' => $nomsStudents,
        'age' => $age
    ]);
}
```

Ensuite vous allez les afficher sous forme de liste avec un condition pour l'âge et un message en rapport.

```
<ul>
    {% for leNom in lesNoms %}
        <li> {{leNom}} </li>
    {% endfor %}
</ul>
```

Là vous allez créer une boucle qui va boucler sur l'array afin d'afficher tous les prénoms, ensuite donc vous allez mettre une condition pour afficher un message en rapport avec l'âge.

```
{% for leNom in lesNoms %}  
    <li> {{leNom|upper }} </li>  
{% endfor %}  
{% if age > 18 %}  
    Vous êtes majeur  
{% else %}  
    Vous êtes mineur  
{% endif %}
```

Et voilà vous avez bien vos prénoms avec une condition pour l'âge, vous pouvez voir qu'il y a un filtre (`|upper`), je vous invite à checker la doc twig pour prendre connaissances des différents filtre et leur utilisation. Imaginons que l'on veuille transformer les noms en majuscules par exemple, donc `upper`, donc on vous dit tout simplement que pour appliquer un **filtre** il faut mettre un **pipe**, il y a énormément de filtre donc n'hésitez pas à aller faire un tour sur la doc, testons le **pipe** et **upper**. Imaginons que l'on veuille transformer les noms en majuscules par exemple, donc `upper`, donc on vous dit tout simplement que pour appliquer un **filtre** il faut mettre un **pipe**, il y a énormément de filtre donc n'hésitez pas à aller faire un tour sur la doc, testons le **pipe** et **upper**. Voilà pour la découverte du [twig](#).

Nous allons commencer un tp qui sera un site e-commerce complet, alors préparer vous. Un site e-commerce est un bon tp car il va vous permettre de voir vraiment beaucoup de chose :

- Une création de base de données
- Configuration de la connexion à la bdd

Ensuite nous verrons la gestion des utilisateurs qui comprend :

- Présentation du [bundle security](#)
- Page de connexion
- Page d'inscription
- Page profil
- Page de déconnexion
- Sécurité et rôles d'utilisateurs

Et la gestion de la bdd :

- Modélisation des entités(tables)
- Utilisation de doctrine pour la gestion de la bdd
- Relations entre les entités (tables) produits, catégories, commandes, etc..

Le développement du catalogue de produits :

- Création des entités et des controllers pour les produits
- Gestion des catégories et des sous-catégories
- Intégration des fonctionnalités de recherche et de filtrage

Gestion du panier d'achat :

- Mise en place du panier d'achat
- Ajout, modification et suppression des produits dans le panier

- Calcul des totaux et frais de livraisons

Processus de commande et de paiement :

- Implémentation du processus de commande
- Intégration de passerelle de paiement (style paypal, stripe, etc.)
- Gestion des commandes et des statuts de commande

Et le déploiement :

- Déploiement de l'application

Ce qui en fait donc un projet assez complet qui couvre l'ensemble des points de référence pour un projet Symfony 7, allez c'est parti.

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

Vous allez donc créer un nouveau projet, à vous de jouer, pensez à vous mettre au bon endroit dès le début, et ouvrez-le sur votre navigateur.

Ensuite découvrons le fichier.env, il contient beaucoup d'information mais seulement une partie va vous intéresser,

```
###> doctrine/doctrine-bundle ###
# Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/config.html
# IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
#
# DATABASE_URL="sqlite:///%kernel.project_dir%/var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=16&charset=utf8"
###< doctrine/doctrine-bundle ###
```

Pensez à bien lire et regardez nous allons travailler sur une base de donnée MySql, vous allez commencer par lancer [Wamp](#), pour s'occuper de la configuration de la bdd, vous allez devoir commenter la ligne de Postgresql en utilisant le diease comme la ligne du dessus et donc décommenter la ligne de MySQL. Voyons cette ligne ensemble,

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/my_shop?serverVersion=8.0.32&charset=utf8mb4"
```

Vous avez le type de base de données, ensuite après le // vous avez le nom d'utilisateur puis après les : vous avez le mot de passe, ensuite l'adresse ip de la bdd et donc après le / le nom de la base de données qui s'arrête au ? ensuite ce sont le paramètres. Alors configurons cela, en sachant que je ne met pas de passe pour ce cours.

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/my_shop?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
```

Voilà pour moi, vous n'êtes pas obligé de garder la même chose ce sera juste plus simple mais rien d'obligatoire, et vous sauvegarder cela. Vous allez pouvoir créer votre bdd en ligne de commande avec le terminal avec cette commande-là, et vous aurez un message de confirmation,

```
Jérémie@DESKTOP-2RNBL59 MINGW64 /c/wamp64/www/e-commerce (master)
$ symfony console doctrine:database:create
● Created database `my_shop` for connection named default
```

Là vous venez de créer la bdd grâce à doctrine, vous pouvez aller vérifier dans Php MyAdmin, elle est bien présente, c'est un bon début. Nous allons maintenant voir pour l'authentification et donc la connexion, il y a un bundle pour gérer cela dans Symfony il s'appelle [Security](#) vous avez tout ce qui concerne cela sur la doc de Symfony, il est installé automatiquement à l'initialisation du projet. Vous allez devoir créer un [Controller](#) qui va gérer votre page d'accueil et pour cela vous allez le faire en ligne de commande avec cette commande,

```
Jérémie@DESKTOP-2RNBL59 MINGW64 /c/wamp64/www/e-c
$ symfony console make:controller HomeController
  created: src/Controller/HomeController.php
  created: templates/home/index.html.twig
```

Comme la dernière fois cela génère tous les fichiers, vous pouvez aller vérifier cela, vous allez mettre ce [Controller](#) en page d'accueil par défaut, et testez cela Voilà tout fonctionne bien, on va pouvoir passer à l'étape suivante.

Du coup on va maintenant créer le système d'authentification, Symfony a déjà pensé à tout et il existe une commande pour cela, [symfony console make:user](#), après cette commande il va vous demander le nom de la classe de l'entité, vous allez conserver user car c'est assez clair, ensuite il va vous demander si vous stocker les infos dans la bdd via doctrine(on va y revenir pas de panique). La troisième question sera quel propriété sera unique à la connexion vous allez laisser l'email car lui seul est unique, la dernière sera si l'on souhaite hasher le mot de passe avant de le stocker dans la bdd, et bien oui on souhaite le hasher, c'est un première protection car le mot de passe ne sera qu'une suite de lettre et de chiffre et sera donc protégé. Là il vous montre ce que vous venez de créer, donc une [Entity](#) user ainsi que son [Repository](#). L'[Entity](#) est la table qui est dans la base de données, et le [Repository](#) est lui une classe qui va vous permettre de faire des requêtes sur cette table et qui contient des méthodes qui vont vous permettre d'agir sur cette table. Il vous dit ensuite qu'il va falloir effectuer la migration, la migration en fait c'est ce qui va vous permettre d'envoyer vos tables grâce à l'entity vers votre base de données. Donc il vous donne les commandes, [symfony console make:migration](#), si jamais vous rencontrer des erreurs à la migration on voit cela ensemble ça se passe dans votre fichier.env. Allons voir un peu ce fichier de migration, c'est une classe avec des fonction et qui contient donc les requêtes de création de la table et de ces champs. Le terminal vous donne la ligne à taper pour effectuer la migration en bdd [symfony console doctrine:migration:migrate](#), ensuite allez vérifier dans votre bdd. OK vous avez donc la table user qui est là et prête, vous allez donc créer un formulaire de connexion, symfony donne la possibilité de le générer, [symfony console make:auth](#), vous répondrez le choix 1, ensuite le nom de la classe je vais faire simple SecurityAuthenticator pour ma part, ensuite le nom du controller et le choix par défaut doit être SecurityController et vous validez, et enfin voulez-vous la route logout et bien oui et maintenant est ce que l'on veut qu'il garde la partie 'vous souvenez vous de moi' et je vais dire oui, ensuite comment activer cette fonctionnalité, soit checkbox soi toujours l'activer. Vous choisissez mais je vais mettre le choix deux personnellement pour ce cours, et voilà c'est fini. Il vous donne quelques conseil à vérifier bien sûr, donc la route du formulaire, le logout et vérifier le controller.

Il va falloir désormais créer un formulaire d'inscription avec cette commande [symfony console make:registration-form](#), il va ensuite vous demander si vous souhaiter ajouter la validation unique donc oui, ensuite l'email on mettra non c plus simple, ensuite si l'utilisateur sera connecté direct après l'inscription on va dire oui, il va encore vous générer les fichiers on vérifie et c'est ok. Donc maintenant ce que l'on doit faire c'est faire la redirection, quand on est enregistrer on est connecté

et donc rediriger sur un autre page en général la page d'accueil, faisons cela. Allez sur le file SecurityAuthenticator sur la function onAuthenticationSuccess,

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token,
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName))
        return new RedirectResponse($targetPath);
}

// For example:
// return new RedirectResponse($this->urlGenerator->generate('some_route'));
throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
```

Symfony a déjà tout prévu, vous allez décommenter la ligne et mettre la route vers laquelle vous voulez rediriger vos user. Pensez à tout tester donc déconnexion, login et on test. Parfait tout fonctionne à merveille jusqu'ici. Ok allons voir un peu le register form, Symfony crée les formulaires grâce au builder, vous pouvez aller regarder vous verrez les champ de votre formulaire dans le builder et vous pouvez en ajouter si les champs existent dans votre Bdd MySQL bien sûr.

```
$builder
    ->add('email')
    ->add('name')
    ->add('agreeTerms', CheckboxType::class, [
        'mapped' => false,
        'constraints' => [
            new IsTrue([
                'message' => 'You should agree to our terms.',
            ]),
        ],
    ])
```

Vous allez ajouter les champs firstName et lastName si vous ne les aviez pas déjà, n'oubliez pas que votre formulaire est créé à partir de l'entity User, vous allez donc devoir les ajouter à votre entity, pour cela vous allez taper la commande **symfony console make:entity** à nouveau et là bien sur vous allez préciser le nom de l'entity donc User et il vous dira qu'elle existe déjà et veut savoir si vous souhaitez ajouter de nouveau champ et donc vous les ajouterez.

```
$ symfony console make:entity User
Your entity already exists! So let's add some new fields!

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

On va donc ajouter ce que l'on vient de créer, donc firstName, puis il demande son type, c'est donc du string, et la longueur vous êtes libre, mais n'oubliez pas que c'est un prénom.

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

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

Ensuite donc la longueur du champ , puis si le champ peut être null en base de donnée et bien non on veut qu'il y en ait un obligatoirement,

```
Field length [255]:
```

```
>
```

```
Can this field be null in the database (nullable) (yes/no) [no]:
```

```
>
```

Ce sont les réponses par défaut donc vous validez, ensuite faites de même pour la seconde propriété. Vous pouvez aller vérifier sur votre entity User, vous pourrez voir que ces propriétés ont été ajoutées, il vous reste à faire une étape laquelle ??

La migration exactement, donc les deux lignes de commandes vous sont de toute façon annoncées en terminal, ensuite vous pourrez aller vérifier dans votre Bdd, puis supprimer l'utilisateur en cours et en créer un nouveau grâce à votre register form. Et bien sûr allez vérifier en Bdd ensuite, ok tout va bien, maintenant vous pouvez remarquer que votre bouton register du form est au milieu et vous allez faire en sorte qu'il soit en bas, voici comment faire. Rendez-vous dans le folder [Templates](#) et [Registration](#) puis dans le file [register.html.twig](#), vous allez donc copier cette ligne 2 fois et modifier pour faire apparaître vos deux nouveaux champs.

```
{% form_row(registrationForm.email) %}
```

Ensuite enregistrer et allez vérifier sur votre application, et voilà c'est beaucoup mieux, vos éléments se sont bien calés à leur place.

Rappel des étapes :

Vous avez initialisé votre projet, puis créé un espace [connexion/ déconnexion](#), également une partie [register](#) puis les [redirections](#), ainsi qu'une [création de base de données](#) avec sa table [utilisateur](#) et le [hashage](#) du mot de passe.

Ok on va attaquer la moise en forme de vos formulaires, déjà vous avez remarqué que le background est en bleu, cette propriété CSS se trouve dans le folder assets => styles => app.css. Le style se trouvera donc ici, à moins d'un fichier différent pour chaque page, à vous de voir. Pour le Javascript cela se trouve dans assets => app.js. Donc on va gérer la mise en forme avec [Bootstrap](#), vous pouvez vous rendre sur la doc de [Bootstrap](#), <https://getbootstrap.com/>, Vous allez descendre un peu et vous rendre sur la partie 'include via cdn' et donc récupérer les liens, vous allez copier le premier lien et le coller dans le base.html.twig qui est la page qui contient toute votre appli, la référence si on veut.

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
<link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22>%3Csvg%3E%3C/svg%3>"
```

OK désormais vous allez pouvoir styliser votre [login form](#), vous vous rendez sur votre [twig](#), et vous allez englober tout votre form dans un div class `container`,

```
<div class="container">
    <form method="post">
        {% if error %}
```

Cela va déjà modifier votre visuel, ensuite vous allez ajouter une class au form, je vais mettre `loginForm` pour ma part, ensuite dans le file `app.css` vous allez agir sur cette classe et lui donner une width de 400px par exemple, vous allez également ajouter une classe `btn-block` ou `w-100` qui est une classe Bootstrap à votre bouton, vous allez pouvoir ajouter une nouvelle classe au container et lui attribué les propriétés suivante,

```
.containerForm {
    display: flex;
    align-items: center;
    justify-content: space-around;
    width: 100%;
    height: 100vh;
}
```

Vous les connaissez déjà toutes, et regardez le résultat. Voilà pour le formulaire de connexion, vous allez passer au formulaire de register, si vous désirez aller plus loin dans la personnalisation n'hésitez pas bien évidemment. A vous de tester la personnalisation du register seul. Attention c'est un formulaire fait par un builder de Symfony, faites les bonnes recherches.

On peut donc le faire de cette manière,

```
{{ form_start(registrationForm) }}
{{ form_row(registrationForm.email, {'attr':{'class':'form form-control',
                                             'placeholder': "email"}) }}}
```

Afin de donner un attribut, donc une classe ainsi qu'un place Holder si vous le désirez, vous pouvez faire tous vos champs et voilà ce sera bien mieux, ensuite vous avez les label qui sont en anglais comment pourriez-vous remédier à cela ?

Et bien deux solutions se présente à vous soit vous transformer le `form_row` en `form_widget` ce qui fera disparaître les labels, ou bien vous ajoutez des label en français et oui, aussi bête que cela. Si vous mettez des `form_widget` vous aurez quelques réglage à faire, à vous de jouer. Voilà pour le login ainsi que le register, bravo à vous.

OK cette partie fonctionne bien, on va passer aux choix des catégories, on va faire en sorte qu'on puisse ajouter, modifier ou supprimer une catégorie. Il va falloir tout d'abord créer une [entity Category](#),

Exercice 1

Donc votre [Entity](#) est créé, on va ensuite créer un controller qui gérera les catégories avec la commande `symfony console make:controller CategoryController`, j'essaie de rester cohérent dans mes fichiers et de [Controller](#) comme pour tout d'ailleurs cela doit être très clair. Vous pouvez

vous rendre sur le [Controller](#) et vérifier que tout fonctionne bien, il va falloir créer la fonctionnalité d'ajout de catégorie. Vous avez donc la première route qui a été générée, on va faire en sorte que cette route affiche toutes les catégories, il va falloir d'abord s'occuper donc de l'ajout. Vous allez créer une seconde route qui sera `category/new` qui sera une public function qui renverra une réponse,

```
#[Route('/category/new', name: 'app_category_new')]
public function addCategory(): Response
{
    return $this->render('category/index.html.twig', [
        'controller_name' => 'CategoryController',
    ]);
}
```

Cette fonction aura besoin d'un paramètre qui sera `entityManagerInterface`, et de modifier le return car ce n'est pas la même vue, et donc de créer cette vue.

```
#[Route('/category/new', name: 'app_category_new')]
public function addCategory(EntityManagerInterface $entityManager): Response
{
    return $this->render('category/newCategory.html.twig');
}
```

Super, allez on s'occupe de la vue, donc dans le folder `category` vous créez le file `newCategory`, et ensuite vous allez créer le formulaire avec la commande `symfony console make :form`, il vous demandera le nom de la classe et se sera `CategoryFormType`, ensuite à quelle `Entity` il est rattaché et ce sera `Category`, et vous validez. Ensuite allez vérifier dans le folder `form`, ok maintenant il reste à préciser le formulaire dans le `Controller`. Pour cela on va utiliser le `createFormBuilder` et lui rattacher le formulaire,

```
$form = $this->createFormBuilder(CategoryFormType::class);
```

Ici on initialise une variable appelée `$form` qui aura pour valeur `$this` (qui est la super variable), et qui prendra le `createFormBuilder` en lui rattachant le formulaire qui est une class bien sûr, mais on va devoir l'associer également à l' `Entity` et oui sinon comment irai t'elle dans la bdd. On va mettre en place la `request`, et bien sur la création d'un variable utilisable en front qui contiendra le formulaire.

```
$category = new Category();

$form = $this->createForm(CategoryFormType::class, $category);
$form->handleRequest($request);

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

OK passons sur la view, il vous faudra importer `base.html.twig`, et créer la base du formulaire,

```

{% extends 'base.html.twig' %}
{% block body %}
    <div class="container">
        <div class="form">
            <h2>Ajouter une catégorie</h2>
            {{ form_start(form) }}
                {{ form_row(form.name, {'attr':{'class':'form form-control'}}) }}

                <input type="submit" value='sauvegarder' class='btn btn-primary mt-4'>
            {{ form_end(form) }}
        </div>
    </div>
{% endblock %}

```

Voilà qui est un bon début, jai mis une petite marge sur le bouton afin qu'il ne soit pas collé à l'input. Vous pouvez tester votre formulaire, il ne se passera rien. Il manque quelque chose c'est quoi ?

Enfaiit il faut désormais dans votre [Controller](#) vérifier que le formulaire et bien soumis et surtout qu'il est bien valide, allez on écrit cela sous forme de condition.

```

$category = new Category();

$form = $this->createForm(CategoryFormType::class, $category);
$form->handleRequest($request);

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

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

```

Ici nous avons donc dit si le formulaire et soumis et qu'il est valide alors [lentityManager](#) va persister c'est-à-dire mettre de cote en attendant d'envoyer en bdd et ensuite on flush donc qui veut dire que cela va envoyer en bdd, là, ont est bien, à vous de vérifier votre formulaire désormais. Et donc créer votre première catégorie, libre à vous de choisir quelle boutique vous voulez faire. Vous avez été vérifier en bdd que votre catégorie c'est bien ajouté, ensuite vous allez créer un autre action qui sera de mettre à jour la catégorie, essayer seul. Indice regardez ce qu'est une [wildcard](#) ? Vous en aurez besoin.

Exercice 2

Alors on crée la route dans le [Controller](#) ainsi que la logique, et bien c'est très simple car on a juste a copier-coller ce que l'on a fait précédemment pour la création de catégorie en enlevant le persist ou pas d'ailleurs,

```

#[Route('/category/{id}/update', name: 'app_category_update')]
public function updateCategory(Category $category, EntityManagerInterface $entityManager, Request $request): Response
{
    $form = $this->createForm(CategoryFormType::class, $category);
    $form->handleRequest($request);

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

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

```

Ensuite on crée la view, qui là encore sera similaire avec celle de la création,

```

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

{% block body %}
    <div class="container">
        <div class="form">
            <h2>Modifier une catégorie</h2>
            {{ form_start(form) }}
                {{ form_row(form.name, {'attr':{'class':'form form-control'}}) }}

                <input type="submit" value='Modifier' class='btn btn-primary mt-4'>
            {{ form_end(form) }}
        </div>
    </div>
{% endblock %}

```

L'action est exactement la même si ce n'est le titre qui change et voilà pas plus difficile que cela, bravo à ceux qui ont réussi.

Maintenant on va faire en sorte d'afficher les catégories qui existent en bdd, vous avez déjà la route qui fait cela, /category vous avez juste quelques petites modifications à effectuer. Pour cela vous aurez besoin du repository, qui lui-même est créé lors de la création de l'[Entity](#) ils fonctionnent ensemble, regardons les fichiers ensemble.

```

#[Route('/category', name: 'app_category')]
public function index(CategoryRepository $repo, ): Response
{
    $categories = $repo->findAll(); //permet de récupérer tous les éléments de la bdd

    return $this->render('category/index.html.twig', [
        'categories' => $categories //contient un array avec toutes les catégories
    ]);
}

```

Ok, très bien il reste à afficher les catégories dans la view, comment affiche-t-on tous les éléments d'un array ?

Ensuite vous pouvez modifier votre view afin d'afficher les catégories.

```

{% block body %}
    <table class='table-borderless'>
        <tr>
            <th>Nom de la catégorie</th>
            <th>Action</th>
        </tr>
        {% for category in categories %}
            <tr>
                <td>
                    {{ category.name }}
                </td>
                <td>
                    Action
                </td>
            </tr>
        </table>
    {% endblock %}

```

Voilà qui est un bon début, puis une petite condition au cas où il n'y est pas de catégorie,

```

        </tr>
        {% else %}
            <p>Aucune catégorie à afficher</p>
        {% endfor %}

```

Allez déjà tester cela, mettez un titre ça sera un peu mieux ainsi que la div container, et enlever la classe borderless ce sera beaucoup mieux en visuel. Il faut ensuite ajouter les boutons d'action,

```

<tr>
    <td>
        {{ category.name }}
    </td>
    <td>
        <a href="{{ path('app_category_update', {'id':category.id}) }">>Modifier</a>
    </td>
</tr>

```

Voilà pour le modifier, on a créer un lien pour le moment avec le path qui est la route avec sa logique et on précise le paramètre id car c'est l'id qui permet de savoir quelle catégorie on modifie. Maintenant vous aurez également besoin d'un bouton supprimer même si sa logique et sa route n'existe pas encore. Vous allez constater que c'est l'opération la plus simple de toutes, retournez dans votre [Controller](#) et faisons cela.

```

#[Route('/category/{id}/delete', name: 'app_category_delete')]
public function deleteCategory(Category $category, EntityManagerInterface $entityManager): Response
{
    $entityManager->remove($category);
    $entityManager->flush();

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

```

Et voilà, et après suppression et bien on redirige l'utilisateur vers la page d'accueil, personnellement je vais mettre la redirection après chaque action pour être tranquille, vous actualisez et vous testez cela. On pourra par exemple ajouter un bouton qui nous permet d'ajouter une catégorie sur la page de l'affichage se sera plus simple, allez-y seul. Voilà pour le système des catégories, bravo à tous.

Correction Ajout de catégorie

Il suffit juste de créer un bouton, avec le path qui contient le name du [Controller](#) afin de vous amener sur le formulaire d'ajout de catégorie.

```

</table>
<a href="{{ path('app_category_new') }}" class="btn btn-success">Ajouter une catégorie</a>

```

Parfait tout fonctionne bien, vous allez mettre en place l'affichage de message flash pour informer l'utilisateur que l'action qu'il a réalisé s'est bien passé, c'est toujours mieux d'informer les utilisateurs et nous mettrons ensuite en place un système de droit d'accès user/admin.

Vous allez créer un folder appelé [layouts](#) (dans le folder templates) et dans ce folder un file qui se nommera [_flash_message.html.twig](#). Dans ce fichier vous allez créer une boucle for in qui prendra le type du message et donc préciser le message, il y aura à l'intérieur de cette boucle une nouvelle boucle for in qui bouclera sur les messages et qui renverra cela dans une div.

```

{% for type, messages in app.flashes(['success', 'danger', 'info']) %}
    {% for message in messages %}
        <div class="col-md-12 mx-auto mt-2">
            <div class="alert alert-{{ type }} alert-dismissible fade show">
                {{ message }}
            </div>
        </div>
    {% endfor %}
{% endfor %}

```

A l'intérieur de cette div vous allez donc afficher le message,

```

<div class="alert alert-{{ type }} alert-dismissible fade show">
    {{ message }}
</div>

```

Une fois enregistré vous allez retourner au niveau de votre [Controller](#) des catégories, et vous allez ajouter cela à chaque action, créer, modifier et supprimer. Donc commencez par la création, après le flush et avant la redirection vous placez ceci

```

$entityManager->flush();

$this->addFlash('success', 'Votre catégorie à bien été créée');

return $this->redirectToRoute('app_category');

```

Vous précisez le \$this puis la méthode addFlash , laquelle prend deux paramètres qui sont le type du message définit précédemment, et le message en question. Faites de même pour l'update, et le delete attention pour le delete se sera le type danger.

Maintenant il faudra préciser l'endroit où sera affiché le message, en vous rappelant qu'il y a une redirection après l'action donc selon vous ou mettrons nous ce message ??

Je le placerai après le titre de la page personnellement, donc pour la modification nous sommes redirigé vers la page qui est dans le dossier category et le fichier index.html.twig donc on le placera ici.

```

<h1 class='mt-5 mb-4'>Liste des catégories</h1>

{% include 'layouts/_flash_message.html.twig' %}

<table class='table'>

```

Evidemment comme toujours vous testez cela sur toutes les actions, chez moi tout fonctionne parfaitement. Parfait ceci est terminé, nous allons passer à la gestion des droits d'accès.

Vous allez vous rendre dans le folder config->packages et le file security.yaml, et trouver la partie acces_control :

```

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

```

Ici cela gère les rôles par rapport a l'url, vous allez déjà décommenter la première ligne, cela signifie que les urls commençant par /admin ne seront accessible que par les utilisateurs qui auront le rôle admin. Retourner dans votre Controller category et modifions ces routes en ajoutant le admin a l'url.

```

#[Route('/admin/category',

```

Vous pouvez retester votre application et vous verrez que l'accès sera refusé car vous n'êtes pas en rôles admin. Vous aurez une erreur **Access Denied**. Vous pouvez retourner sur le file qui contient les rôles et créer une nouvelle ligne avec un rôle editor qui lui pourra jouter ou modifier l'ajout de produit en boutique par exemple,

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

Il faudra ensuite attribuer le rôle admin à un de vos utilisateurs personnellement je vais me mettre moi en admin. Donc vous vous rendez sur votre BDD, et on va faire cela ensemble, vous allez sur votre table user et vous avez une colonne rôles, vous cliquer sur le profil , et sur éditer ensuite vous vous rendez sur la partie rôles et vous ajoutez ceci

```
["ROLE_ADMIN", "ROLE_EDITOR", "ROLE_USER"]
```

Pourquoi tout cela, et bien tout simplement car ce profil-là aura tous les rôles, de l'ADMIN jusqu'au USER, il pourra donc effectuer toutes les actions possibles sur cette appli, ainsi qu'accéder à toutes les pages. Vous enregistrez et relancez votre page et l'accès devrait être autorisé désormais. Cela va automatiquement déconnecter l'utilisateur, vous devrez vous reconnecter car cela détecte la modification de la BDD, et tout fonctionnera parfaitement. Créez par la suite un second utilisateur, et essayez d'aller sur la page catégorie et là ça ne fonctionnera plus, les rôles fonctionnent bien, je vais me remettre sur le user admin pour la suite. Voilà on a bien avancé, bravo à tous et toutes.

Désormais on s'occupera de pouvoir afficher les users ainsi que leur rôles, il y aura donc 3 rôles, l'admin, un éditeur et les users classiques. Mais on va d'abord faire une petite modification dans notre code si jamais un user veut supprimer une catégorie on va afficher une petite boîte de dialogue lui demandant de confirmer son action afin d'être sûr. Vous allez ajouter une classe sur votre bouton modifier qui sera la classe `btn btn-outline-primary` donc dans le fichier twig `index.html.twig` du dossier `category`.

```
<a class="btn btn-outline-primary" href="{{ path('app_category_update'),
```

Vous devriez déjà avoir la classe `btn btn-danger` sur le bouton supprimer, lorsque vous cliquez sur ce bouton il supprime automatiquement la catégorie mais cela sera mieux d'avoir une petite boîte de dialogue de confirmation. Mettons donc cela en place, vous allez ajouter une méthode javascript '`onclick`' qui renverra un retour de confirmation avec un message personnalisé.

```
<a onclick="return confirm('Voulez-vous vraiment supprimer cette catégorie?')"
```

Retournez tester votre application en ayant bien sur actualiser votre page et vous aurez le message qui s'affichera, super. Vous allez ajouter un menu Bootstrap à votre application également, vous allez le faire seul on corrigera ensuite si besoin.

Exercice 3

Dans votre folder `templates`, vous avez un folder `layouts` et bien vous allez créer un nouveau file qui se nommera `nav` ou `navbar.html.twig` et ici vous allez coller votre code Bootstrap de votre navbar, et ensuite vous allez importez cela dans `base.html.twig` dans la balise `body` mais avant le block `body`, vous allez créer un block `nav` et l'inclure.

```

<body>
    {% block nav %}
        {% include 'layouts/navbar.html.twig' %}
    {% endblock %}
    {% block body %}{% endblock %}
</body>

```

Voilà maintenant je peux allez actualiser et vérifier que cela fonctionne, ok chez moi et si vous désirez lui attribuer une couleur regardez la classe `data-bs-theme`.

```

<nav class="navbar navbar-expand-lg bg-body-tertiary" data-bs-theme="dark">
    <div class="container">

```

Voilà pour moi, il faudra juste importer le javascript afin que tout fonctionne ainsi que le menu burger en responsive. Pour cela il faudra vous rendre sur la home page de Bootstrap et récupérer la partie script en lien CDN, puis le coller dans la partie link de `base.html.twig`.

```

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet"
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
<link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 vie

```

Pensez à actualiser et tester cela bien sûr, on actualisera les liens et le menu au fur et à mesure pas de panique, vous pouvez déjà changer le nom et lui attribuez le nom de votre shop vu que l'on fait une e-boutique.

```

<div class="container">
    <a class="navbar-brand" href="#">SneakHub</a>

```

OK super, maintenant on pourra ajouter la partie catégorie dans notre menu déroulant, rendez-vous sur ce bout de code, c'est la partie `dropdown-menu`, on va faire en sorte que seul ceux qui ont le rôle adéquat y ait accès, en symfony il y a une fonction qui sert à cela c'est '`if is_granted`'. Rendez-vous sur la liste et mettons cela en place si c'est bien l'admin connecté alors on pourra effectuer certaines actions et afficher certaines choses dans votre navbar.

```

{% if is_granted("ROLE_ADMIN") %}
    <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
            Link
        </a>
        <ul class="dropdown-menu">
            <li><a class="dropdown-item" href="#">Action</a></li>
            <li><a class="dropdown-item" href="#">Another action</a></li>
            <li><hr class="dropdown-divider"></li>
            <li><a class="dropdown-item" href="#">Something else here</a></li>
        </ul>
    </li>
{% endif %}

```

Et donc vous allez personnaliser votre menu un petit peu, commencez par ajouter le lien vers la page catégorie par exemple

```
<ul class="dropdown-menu">
    <li><a class="dropdown-item" href="{{ path('app_category') }}">Catégories</a></li>
```

Voilà cela fonctionne bien, ajoutez le lien pour l'ajout de catégorie dans le menu, à vous de jouer seul.

```
<li><a class="dropdown-item" href="{{ path('app_category') }}">Catégories</a></li>
<li><a class="dropdown-item" href="{{ path('app_category_new') }}">Ajouter une catégorie</a></li>
```

Super passons a la partie utilisateurs comme prévu au départ,

```
<ul class="dropdown-menu">
    <li><a class="dropdown-item" href="{{ path('app_category') }}>Catégories</a></li>
    <li><a class="dropdown-item" href="{{ path('app_category_new') }}>Ajouter une catégorie</a></li>
    <li><hr class="dropdown-divider"></li>
    <li><a class="dropdown-item" href="#">Utilisateurs</a></li>
</ul>
```

Il faudra donc ensuite créer quelque chose qui permettra de récupérer tout les utilisateurs ainsi que leurs rôles car l'admin lui a le droit de gérer tout ce qui concerne les users que doit on créer ??

Un [Controller](#) exactement, alors allez y seul encore une fois. Si besoin je suis évidemment là pour vous aiguiller.

```
Jérémie@DESKTOP-2RNBL59 MINGW64 /c/wamp64/www/e-co
$ symfony console make:controller UserController
```

Vous patienter et tout vos fichiers seront générés automatiquement, ensuite allez sur le [Controller](#) et protéger la route, et ensuite vous allez ajouter le repository nécessaire qui sera ??

UserRepository exactement, voilà comment écrire cela

```
class UserController extends AbstractController
{
    #[Route('/admin/user', name: 'app_user')]
    public function index(UserRepository $userRepository): Response
    {
        return $this->render('user/index.html.twig', [
            'users' =>$userRepository->findAll(),
        ]);
    }
}
```

La méthode `findAll()` permet de récupérer tout les utilisateur cela créera un array qui les contiendra tous.

>Déjà vous allez enlever tout ce qui n'est pas nécessaire sur votre fichier twig ensuite comment récupère t'on les données d'un array ?

Avec un boucle `for in` bravo !! Allez créons ce fichier twig, on va créer un tableau,

```

{% block body %}
    <div class="container">
        <table class='table'>
            <tr>
                <th>id</th>
                <th>Nom</th>
                <th>prénom</th>
                <th>Email</th>
                <th>Rôle</th>
                <th>Action</th>
            </tr>
        </table>
    </div>
{% endblock %}

```

Et donc en dessous du tr vous allez créer la récupération des users allez y .

```

<th>Action</th>
</tr>
{% for user in users%}
<tr>
    <td>{{ user.id }}</td>
    <td>{{ user.firstName }}</td>
    <td>{{ user.lastName }}</td>
    <td>{{ user.email }}</td>
    <td></td>
    <td>
        <a href="" class="btn btn-danger">Supprimer</a>
    </td>
</tr>

```

Bravo a ceux qui ont réussi, ok parfait cela affiche tout les users avec leur bouton supprimer. Ajoutez un titre a votre page,

```

<div class="container">
    <h1>Liste des utilisateurs</h1>
    <table class='table'>

```

Maintenant il faut réfléchir à comment afficher les rôles, comment ferez-vous cela ? Sachez que même si en bdd le tableau rôles est vide il y a bien le rôle user par défaut.

Encore une boucle for in, et oui, mettons cela en place de suite,

```
<td>
    {% for role in user.roles %}
        {{ role }}
    {% endfor %}
</td>
```

Et comme toujours vérifier cela. OK tout fonctionne parfaitement, maintenant on va créer la route qui permet de changer le rôle, afin que l'admin puisse changer le rôle d'un user et le passer éditeur ou autre.

Exercice 4

```
#[Route('/admin/user/{id}/to/editor', name: 'app_user_to_editor')]
public function changeRole(EntityManagerInterface $entityManager, User $user): Response
{
    $user->setRoles(['ROLE_EDITOR', 'ROLE_USER']);
    $entityManager->flush();

    $this->addFlash('success', "Le rôle éditeur a bien été ajouté à l'utilisateur");
    return $this->redirectToRoute('app_user');
}
```

Voici la correction de la route qui permet de changer le rôle d'un utilisateur, Bien joué à ceux qui ont réussi. Maintenant vous allez créer un lien sur la page User qui permettra donc de changer le rôle d'un user en éditeur, à vous de jouer seul. Essayer de mettre un message pour confirmer l'action si possible.

```
<a onclick= "return confirm('Voulez vous vraiment affecter ce rôle à ce user ?')"
```

Voici pour le message de confirmation,

```
class="btn btn-outline-primary" href="{{ path('app_user_to_editor', {'id':user.id}) }}" >Ajouter le rôle éditeur
```

Et voici la suite du a. Bravo à ceux qui ont réussi ce n'était pas forcément simple, il y avait beaucoup de chose à prendre en compte.

Ce que l'on pourra faire ensuite, et bien c'est de vérifier si un utilisateur à le rôle admin et bien on lui retire cette option qui n'aurait pas grand intérêt, si vous voulez essayer seul, allez y sinon on le voit ensemble, c'est une simple condition que vous connaissez.

```
{% if ('ROLE_ADMIN' in user.roles) == false %}
    <a onclick= "return confirm('Voulez vous vraiment affecter ce rôle à ce user ?')"
        <a href="" class="btn btn-danger">Supprimer</a>
    {% endif %}
```

Tout simplement si dans le tableau des rôles le rôle admin est sur false donc pas admin, et bien on affiche les deux boutons sinon on ne les affiche pas. Il ne vous reste plus qu'à insérer le message flash sur cette page vous l'avez fait précédemment donc je vous laisse le faire seul.

```
<h1 class="mt-4 mb-5">Liste des utilisateurs</h1>

{% include 'layouts/_flash_message.html.twig' %}

<table class='table'>
```

Tout simplement en rappelant [l'include](#). Bien joué. La prochaine étape va être de modifier le bouton **Ajouter le rôle éditeur** si le user est déjà éditeur mais plutôt mettre à la place **Retirer le rôle éditeur** ce qui sera plus logique. Quelle seront donc les étapes ?

Et bien c'est simple on va devoir refaire une route qui permettra d'enlever ce rôle, alors allons y. Vous pouvez copier la route précédente, et vous allez la modifier pour s'adapter à ce que vous désirez effectuer comme action. J'aimerai que vous essayiez par vous-même, je reste disponible si besoin.

Correction :

```
#[Route('/admin/user/{id}/remove/editor/role', name: 'app_user_remove_editor_role')]
public function removeRoleEditor(EntityManagerInterface $entityManager, User $user): Response
{
    $user->setRoles([]);
    $entityManager->flush();

    $this->addFlash('success', "Le rôle éditeur a bien été retiré à l'utilisateur");

    return $this->redirectToRoute('app_user');
}
```

Voici donc la nouvelle route qui comme vous pouvez le constater n'a subit que très peu de changement par rapport à la précédente, on lui attribue un tableau vide dans le setter car le rôle par défaut étant le rôle user donc pas besoin de le préciser. Et donc il faut également modifier le message flash et c'est tout. Maintenant l'affichage, donc le fichier twig,

```
{% if ('ROLE_ADMIN' in user.roles) == false %}

    {% if ('ROLE_EDITOR' in user.roles) == false %}
        <a onclick= "return confirm('Voulez vous vraiment affecter ce rôle à ce user ?')"
    {% else %}
        <a onclick= "return confirm('Voulez vous vraiment retirer ce rôle à ce user ?')
    {% endif %}
    <a href="" class="btn btn-danger">Supprimer</a>
    {% endif %}
```

Evidemment pensez à modifier le path à la fin et la class du bouton et allez vérifier, super cela fonctionne à merveille. On va modifier légèrement la code pour qu'il n'affiche que admin, ou éditeur mais pas besoin de préciser le ROLE_EDITOR à chaque fois. Cela va se faire dans le twig,

Voilà comment faire ceci, pensez comme toujours à aller tester ensuite, je vous laisse mettre le style que vous désirez. Il reste encore à penser à pouvoir supprimer un utilisateur. On va voir une nouvelle façon de faire.

```
#Route('/admin/user/{id}/remove', name: 'app_user_remove')
public function ruserRemove(EntityManagerInterface $entityManager, UserRepository $userRepository): Response
{
    $user = $userRepository->find($id);
    $entityManager->remove($user);
    $entityManager->flush();

    $this->addFlash('danger', "L'utilisateur a bien été supprimé.");

    return $this->redirectToRoute('app_user');
}
```

Il n'y a rien de bien sorcier on utilise le repository, et on récupère la bon user grâce à la méthode find() en lui passant en paramètre l'id du user en question et ensuite on le remove puis on flush, et on mais un petit message de confirmation et voilà. Maintenant il reste la partie affichage et c'est réglé.

```
{% endif %}
<a href="{{ path('app_user_remove', {'id':user.id}) }}" class="btn btn-danger">Supprimer</a>
{% endif %}
```

Le bouton était déjà présent mais désormais il fonctionne et l'admin peut donc bannir un utilisateur de sa boutique. Il reste juste à mettre le petit message et c'est bon. Bravo à vous tous, on va passer à une nouvelle étape pour la suite du projet.

On va passer sur la gestion des sous-catégories, vous allez donc créer une table sous-categorie, je mets tout en anglais personnellement je vous conseille encore une fois de prendre cette habitude dès le début, à vous de jouer.

```
$ symfony console make:entity subCategory
  created: src/Entity/SubCategory.php
  created: src/Repository/SubCategoryRepository.php

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

Voici, ensuite vous allez créer les champs :

- Le nom de la sous categorie donc `name`, de type string, longueur maximale et si cela peut être null et bien non cela ne peut pas être null,
- Il va falloir penser à une clef étrangère qui permettra de relier une sous-catégorie à une catégorie, et vous appellerez ce champ `category`, ensuite le type et bien là, il faudra mettre la relation (vous pouvez appuyer sur ? pour voir les types différent) dans notre cas se sera `ManyToOne`, ensuite il va demander à quel entity cela va être relier et ce sera à l'entity category évidemment, ensuite il va vous demander si cela peut-être être null, donc si une sous-catégorie peut être relié à aucune catégorie et bien non, ensuite vous répondrez oui, puis il demandera le champ qui sera utilisé dans l'entity category pour pouvoir récupérer la sous-catégorie vous pouvez laisser le choix par défaut, ensuite si l'on souhaite supprimer automatiquement les entité orpheline, donc si vous supprimez une catégorie cela

supprimera automatiquement les sous-catégorie lier a celle-ci, donc vous mettrez oui. Ensuite vous aurez la question de savoir si vous voulez une autre propriété et là vous ferez entrée et ce sera fini. Cela aura donc généré vos fichiers que vous pouvez vérifier. Ensuite vous partez sur la migration, la ligne de commande vous a été donne sur le terminal, ensuite vous lancez l'exécution de votre migration avec la ligne suivante, puis répondez yes ensuite et ce sera bon en bdd. Allez voir cette table en bdd.

Pour les catégories vous avez créé toutes les méthodes à la main, celle ui permet de créer de modifier et de supprimer une catégorie afin que vous compreniez la manière de faire ainsi que la logique. Désormais vous allez taper une ligne de commande qui va nous générer toute ces méthodes d'un seul coup et oui je sais je suis pas sympa 😊. Donc dans le terminal vous allez taper `symfony console make:crud`. Il va vous demander à quelle entity vous désirez lier votre crud et ce sera donc subCategory, il vous demande le nom du controller normalement vous pouvez prendre le choix par défaut (SubCategoryController). Ensuite il demande si vous désirez générer les fichiers test vous pouvez dire non, puis entrée et il vous donnera la liste des fichier générer. Rappelez-vous, tout ces fichiers vous les aviez créer vus même à la main, et là et bien tout a été créer pour vous, merci Symfony, allez voir ces fichiers et vous verrez le gain de temps.

Allez sur le [Controller](#) et testez votre route principal, vous aurez un tableau vide et c'est normal pour le moment, allez sur le fichier index.html.twig dans le dossier sub_category, et on va modifier légèrement le code, vous allez entourer le code par une div avec la classe container déjà,

```
{% block body %}

<div class="container">
    <h1>SubCategory index</h1>
```

Comme ceci et vous la refermer tout en bas, regardez la différence, vous pouvez également mettre quelques marge afin d'aérer tout cela. Je vous laisse mettre un peu de style selon vos goûts, ensuite vous cliquerez sur crée une nouvelle catégorie, et on va encore modifier légèrement à notre goûts, ensuite allez sur le form, et on va le mettre en forme car la c'est assez simpliste.

```
{{ form_start(form) }}
    <label for="">Nom de la sous-catégorie</label>
    {{ form_widget(form.name, {attr:{'class': 'form form-control'}}) }}
    <button class="btn">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Voilà pour commencer, on a mis un label et une classe pour modifier le champ name du form,

```
{% form_start(form) %}
    <label for="">Nom de la sous-catégorie</label>
    {{ form_widget(form.name, {attr:{'class': 'form form-control'}}) }}
    <label for="">Catégorie</label>
    {{ form_widget(form.category, {attr:{'class': 'form form-control'}}) }}
    <button class="btn btn-outline-primary">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Voilà qui est beaucoup mieux maintenant il y a quelque chose qui me gêne et vous ??

Ce que l'on veut c'est voir le nom de la catégorie et pas son identifiant car à moins de se rappeler par cœur c'est pas du tout pratique. Donc on va modifier cela, vous allez vous rendre dans le [form](#) `subCategory` et vous allez voir que le choice label propose l'id depuis la version 7 avant il mettait le name et bien il faut remplacer et demander le name et vous aurez le nom des catégories qui sera affiché.

```
$builder
    ->add('name')
    ->add('category', EntityType::class, [
        'class' => Category::class,
        'choice_label' => 'name',
    ])
]
```

Faites les petites modification sur vos boutons que vous désirez pour le style, allez-y seul. Je vous laisse faire sur toutes les pages catégories et sous-catégories. Pensez ensuite à entourez les fichiers du dossier sous-catégories par une div avec la classe container de Bootstrap afin d'améliorer le rendu visuel.

Ensuite sur la partie index des sous catégories on pourra également afficher la catégorie à laquelle est rattaché une sous-catégorie afin que cela soit plus clair non ? Comment feriez-vous cela ?

Et bien vous allez ajouter un « th » catégorie et dans les « td » et bien vous allez donc rappelez sous-catégorie et demande le nom de la catégorie rattachée à celle-ci aussi simple que cela, pourquoi car dans l'entité elle sont bien rattachée.

```
<tr>
    <th>Nom</th>
    <th>Catégorie</th>
    <th>Actions</th>
</tr>
```

```
{% for sub_category in sub_categories %}
    <tr>
        <td>{{ sub_category.name }}</td>
        <td>{{ sub_category.category.name }}</td>
        <td>
```

Parfait jusqu'à présent, passons à la nouvelle étape qui va concerner les produits et leur [crud](#).

Exercice 5

Donc à présent vous avez la nouvelle table qui apparait dans votre bdd, ainsi que la table de relation [ManyToMany](#), bravo à vous tous.

Ensuite vous allez créer un [crud complet](#) qui sera donc rattachée à l'entité Product, et vous ne générerez pas le fichier de test qui n'est pas nécessaire, à vous de jouer. Pensez également à ajouter les messages flash au endroits nécessaires, là encore je vous laisse faire cela seul désormais, rien ne vaut la pratique 😊. Et ensuite et bien on vérifie la route [product](#) pour commencer, celle qui affiche tout les produits. A la suite de cela pensez à mettre votre class container sur tous vos fichiers, ensuite

vous vous rendez sur le `form` qui est indiqué dans le file `new.html.twig` qui se trouve dans le folder `Product`, et on va mettre en forme le formulaire.

```
{% form_start(form) %}
{{ form_widget(form.name, {'attr':{'class':'form form-control'}}) }}
{{ form_widget(form.description, {'attr':{'class':'form form-control'}}) }}
{{ form_widget(form.price, {'attr':{'class':'form form-control'}}) }}
{{ form_widget(form.subCategories, {'attr':{'class':'form form-control'}}) }}
<button class="btn btn-outline-primary">{{ button_label|default('Sauvegarder') }}</button>
{{ form_end(form) }}
```

Voilà pour commencer, vous pouvez déjà regarder la différence, il faudra ajouter les labels, et préciser le nom des sous-catégories, allons y. Petite astuce pour les labels, attention cependant ils ne seront pas personnalisé ou en français de cette manière car nous les avons écrit en anglais au début, sinon vous mettez des labels comme on a vu précédemment.

```
{% form_start(form) %}
{{ form_row(form.name, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.description, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.price, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.subCategories, {'attr':{'class':'form form-control'}}) }}
<button class="btn btn-outline-primary">{{ button_label|default('Sauvegarder') }}</button>
{{ form_end(form) }}
```

Ensuite allez sur le file index des products pour afficher le message flash lors de la création d'un nouveau produit car vous êtes redirigé sur cette page après la création.

```
<h1 class="mt-5">Liste des produits</h1>
{% include '_flash_message.html.twig' %}
```

Occupons-nous donc d'afficher le nom plutôt que l'id, comme nous avions fait précédemment. Rendez vous donc sur le form builder et changer cela,

```
$builder
    ->add('name')
    ->add('description')
    ->add('price')
    ->add('subCategories', EntityType::class, [
        'class' => SubCategory::class,
        'choice_label' => 'name',
        'multiple' => true,
    ])
]
```

Et voilà, ensuite petite modification sur le menu et vous allez ajouter le lien qui vous permet d'aller sur la page de la liste des sous-catégories, à vous de jouer. Nous allons passer à la prochaine étape, bravo pour le travail accompli, jusqu'à présent.

Maintenant on va s'attaquer à la partie image, vous allez vous rendre sur le `ProductController` et vous allez modifier la fonction qui permet d'ajouter un produit, il faudra également ajouter le champ image à la table. Mettons à jour la table dans le terminal, pour cela il suffit de recréer la même table et cela détectera que la table existe et vous pourrez la modifier. `Symfony console make:entity`

[Product](#) et il va vous demander d'ajouter un champ vous noterez image, vous laisserais de type string et pourquoi ??

Et car nous n'allons pas stocker les images car cela serai très lourd pour la bdd et votre appli, vous allez stocker les noms des images et les images elles seront dans un dossier spécifique à part. Vous mettrez la longueur maximale, et oui cela pourra être null, et vous effectuez la migration complète. Vous pouvez aller sur le formulaire [ProductType](#) et il va falloir ajouter le champ du formulaire pour l'image.

```
$builder
    ->add('name')
    ->add('description')
    ->add('price')
    ->add('image')
    ->add('subCategories', EntityType::class, [
        'class' => SubCategory::class,
        'choice_label' => 'name',
        'multiple' => true,
    ])
;
```

Allez actualiser votre formulaire et le nouveau champ aura apparu, maintenant rendez vous dans le template du form,

```
{{ form_row(form.subCategories, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.image, {'attr':{'class':'form'}}) }}
```

Parfait, comme ce sera un champ d'image vous allez rajouter un petit quelque chose au form builder,

```
->add('image', FileType::class)
```

Voilà et l'on va ajouter des paramètres sous forme d'array,

```
$builder
    ->add('name')
    ->add('description')
    ->add('price')
    ->add('image', FileType::class, [
        'label' => 'Image du produit',
        'mapped' => false,
        'required' => false, /*ce n'est pas obligatoire comme champ*/
        'constraints'=>[
            new File([ /* c'est cette classe ci "Symfony\Component\Validator\Constraints\File;" */
                'maxSize' => '1024k',
                'mimeType' => [
                    'image/jpeg',
                    'image/png',
                    'image/jpg',
                ],
                'mimeTypeMessage' => 'Veuillez choisir un fichier de type image valide(jpeg, png, jpg)!!',
            ])
        ]
    ])
```

Voici pour le `form` avec ces contraintes, et il reste le `Controller`,

```
#Route('/new', name: 'app_product_new', methods: ['GET', 'POST'])
public function new(Request $request, EntityManagerInterface $entityManager, SluggerInterface $slugger):
{
    $product = new Product();
    $form = $this->createForm(ProductType::class, $product);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $image = $form->get('image')->getData();/* on recup l'image et son contenu*/

        if ($image) {/*si l'image existe*/
            $originalName = pathinfo($image->getClientOriginalName(), PATHINFO_FILENAME);
            $safeImageName = $slugger->slug($originalName);/* permet de recup des image avec espace dans
            $newFileName = $safeImageName.'-'.$uniqid().'.'.$image->guessExtension();/*cree un id un

            try {
                $image->move
                    ($this->getParameter('image_directory'),
                     $newFileName);/* on recup l'image et on la renomme et on la stocke dans le repo
            }catch (FileNotFoundException $exception) {}/*en cas d'erreur*/
                $product->setImage($newFileName);

            }
        }
    }
}
```

Pour le `Controller` il y a quelques étapes à respecter mais c'est plutôt simple dans l'idée, voici le code avec ces commentaires on va détailler cela ensemble. Il vous faudra créer le paramètre '`image_directory`' et se sera sur le fichier `Services.yaml`

```
parameters:
    image_directory: '%kernel.project_dir%/public/uploads/images'
```

Ensuite effectuer un test avec une image et vérifier en bdd le résultat vous devez retrouver le nom de votre image avec un id unique.

4	Nike Tn Enfant	Qui a dit que les plus jeunes n'ont pas besoin d'a...	145	tnEnfant- 668d530eade3e.jpg
---	-------------------	--	-----	--------------------------------

Donc tout fonctionne très bien, bravo à tous. C'était une bonne étape, avec pas mal d'info importante. Maintenant il faudrait protéger ces pages car tout les utilisateur ne doivent pas avoir accès à ces pages là ce sera plutôt réservé aux éditeurs et donc admin, changeons cela et ce sera fini.

```
#Route('/editor/product')
class ProductController extends AbstractController
```

Et voilà encore une étape de fini, l'étape suivante sera l'affichage de vos produits sur la page d'accueil car pour le moment c'est vide, et la gestion du stock.

Rendez vous sur le folder `Product` et le file `index`, et gérer les classes des boutons si cela n'est pas encore fait, ensuite on va modifier la taille de la description, imaginez qu'elle soit plutôt longue, on va utiliser la méthode `slice`

```

<tr>
    <td>{{ product.name }}</td>
    <td>{{ product.description|slice(0,100) }} ...</td>
    <td>{{ product.price }}</td>
    <td>

```

Voilà qui est mieux vous pouvez aller vérifier, ensuite il va falloir ajouter un champ stock en base de données, Allez-y c'est à vous ajouter le champ stock, de type integer, et ne peut pas être null, et vous valider, et vous effectuer la migrations et vous vérifier.

Super, il faut désormais dans le [builder form](#), ajouter le stock,

```

$builder
    ->add('name')
    ->add('description')
    ->add('price')
    ->add('stock')
    ->add('image', FileType::class, [

```

Ensuite, vous pouvez remarquer que le champ stock est dessous le bouton de save, pour régler cela il faudra aller dans le file du [form](#),

```

{{ form_start(form) }}
{{ form_row(form.name, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.description, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.price, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.stock, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.subCategories, {'attr':{'class':'form form-control'}}) }}
{{ form_row(form.image, {'attr':{'class':'form'}}) }}
<button class="btn btn-outline-primary">{{ button_label|default('Sauvegarder') }}</button>
{{ form_end(form) }}

```

Voilà avec ca il sera à sa place, maintenant on va pouvoir donner la possibilité d'ajouter du stock, avant cela ajouter un produits avec du stock, et allez vérifier, c'est bon on va donc afficher le stock maintenant,

Exercice 6

Ok il suffisait donc de faire ceci,

```

<tr>
    <th>Nom</th>
    <th>Description</th>
    <th>Prix</th>
    <th>stock</th>
    <th>Actions</th>
</tr>

```

Et donc d'afficher les datas du stock,

```

{% for product in products %}
    <tr>
        <td>{{ product.name }}</td>
        <td>{{ product.description|slice(0,100) }} ...</td>
        <td>{{ product.price }}</td>
        <td>{{ product.stock }}</td>
        <td>

```

Bravo cela affiche le stock, maintenant il y a des stock à zéro mais on peut faire mieux que cela en affichage. Essayer de faire une condition qui affiche le stock seulement s'il est **supérieur à 0** et sinon vous affiché **stock épuisé**.

```

    {% if product.stock > 0  %}
        {{ product.stock }}
    {% else %}
        <span class="text-danger">Stock épuisé</span>
    {% endif %}

```

Maintenant on va faire en sorte que l'on puisse ajouter en stock par un bouton ce sera plutôt sympa, sans avoir à modifier le produit bien sûr. Vous allez donc créer un 3^{ème} bouton,

```

<a class="btn btn-primary" href="{{ path('app_product_show', {'id': product.id}) }"></a>
<a class="btn btn-outline-success" onclick="return confirm('Voulez-vous ajouter du stock ?')"></a>
<a class="btn btn-outline-info" href="#">Ajouter du stock</a>

```

Maintenant il faut créer l'action qui elle n'existe pas encore, on va faire cela mais en pensant à créer l'historique des réapprovisionnement du stock, on va rajouter un paramètre qui dit qu'un produit sera unique on ne pourra pas créer le même,

```

class Product
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255, unique: true)]
    private ?string $name = null;
}

```

On va désormais créer une entité qui gèrera les données du stock, qui aura pour champ, `product` de type `ManyToOne`, il sera lié à l'entité `Product`, ensuite ça ne peut pas être null, ensuite la `quantity` de type `integer`, ne peut pas être nullable, ensuite `createdAt` et le type sera donnée par défaut et ne peut pas être null, et c'est bon vous faites les migrations complète. Parfait la table est prête, vous

allez vous rendre sur le controller `ProductController` et avoir `persist` et `flush` vous allez ajouter le stock history.

```
$entityManager->persist($product);
$entityManager->flush();

$stockHistory = new AddProductHistory(); /*nouvelle instanciation de la classe*/
$stockHistory->setQuantity($product->getStock()); /*on recup l'id du produit*/
$stockHistory->setProduct($product); /*on insere le produit*/
$stockHistory->setCreatedAt(new DateTimeImmutable());
$entityManager->persist($stockHistory);
$entityManager->flush(); /*effectue la mise a jour en bdd*/

$this->addFlash('success', 'Votre produit a été ajouté');
```

Voilà on va tester en ajoutant un nouveau produit et voir la réaction, une fois le message flash de réussite allez voir sur votre nouvelle table et vérifier que tout c'est bien passé.

id	product_id	quantity	created_at (DC2Type:datetime_immutable)
1	6	10	2024-07-10 14:39:10

Vous devez avoir quelque chose de ce style, maintenant on va faire en sorte de pouvoir ajouter du stock simplement, par action. Pour faire cela, vous allez vous servir de la table `AddProductHistory` et réaliser un `crud`, et tapez la commande `symfony console make:form`, il va demandez la classe du form à quelle entity on relie ce form donc on précise, ensuite validez et allez vérifier votre form nouvellement créer, sur ce form vous n'allez laisser que la quantité.

```
public function buildForm(FormBuilderInterface $builder)
{
    $builder
        ->add('quantity')
    ;
}
```

Vous allez retourner sur le `Controller` a la fin des méthodes et créer une nouvelle route pour cette action,

```
#[Route('/add/product/{id}/', name: 'app_product_stock_add', methods: ['POST'])]
public function stockAdd($id, EntityManagerInterface $entityManager, Request $request): Response
{
    $stockAdd = new AddProductHistory();
    $form = $this->createForm(AddProductHistoryType::class, $stockAdd);
    $form->handleRequest($request);

    return ...
}
```

Et vous allez retourner un fichier qu'il vous faudra créer, donc allez-y et dans le dossier [Product](#) des [Templates](#), je l'ai appelé `addStock.html.twig` personnellement, et donc on va le retourner.

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

Voilà et bien il va falloir tester, vous prenez donc le [name](#) de la route, et vous ajouter le path sur le file index de [Product](#).

```
href="{{ path('app_product_stock_add', {'id': product.id})}}
```

Et comme toujours on test, il faut se rendre sur la page [Product](#) et essayer d'ajouter du stock à un des produit qui à le stock épuisé. Vous aurez très certainement une erreur (quasi sûr) et bien celle-ci c'est à vous de la résoudre. Et vous devez faire en sorte d'atterrir sur la page du [form](#) qui pour le moment est vide. Ensuite vous allez sur la page template du form et on mettre cela en place, vous allez essayer seul.

Exercice 7

Voici le formulaire,

```
{% extends 'base.html.twig' %}
{% block title %}Ajouter du stock{% endblock %}

{% block body %}
    <div class="container">
        <h1 class="mt-5 mb-5">Ajouter du stock</h1>
        {{ form_start(form) }}
        {{ form_row(form.quantity, {'attr':{'class':'form form-control mb-3'}}) }}
        <input type="submit" class='btn btn-success btn-block'>
        {{ form_end(form) }}
    </div>
{% endblock %}
```

Pour l'affichage du produit en question il fallait aller sur le [Controller](#) et ajouter un paramètre qui est le repo des produits, et appliquer une méthode qui est la méthode `find()` et lui passer en paramètres l'id du produit.

```
Request $request, ProductRepository $productRepository): Response
```

Et voici pour la transmission du back au front du produit,

```

$product = $productRepository->find($id);

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

```

Il reste plus qu'à récupérer cela pour l'affichage sur la vue,

```

<div class="container">
    <h1 class="mt-5 mb-5">Ajouter du stock pour : {{ product.name }} </h1>

```

Super, on va rajouter un petit quelque chose sur le [Controller](#), pour faire en sorte que la mise à jour s'effectue parfaitement.

```

$product = $productRepository->find($id);

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

    if($stockAdd->getQuantity()>0){
        $newQuantity = $product->getStock() + $stockAdd->getQuantity();
        $product->setStock($newQuantity);

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

        $this->addFlash('success', "Le stock du produit a été modifié");
        return $this->redirectToRoute('app_product_index');
    }
}

```

Premier cas de figure si le stock est supérieur à zéro, et voici le second cas de figure,

```

    return $this->redirectToRoute('app_product_index');
} else {
    $this->addFlash('danger', "Le stock du produit ne doit pas être inférieur à zéro");
    return $this->redirectToRoute('app_product_stock_add', ['id'=>$product->getId()]);
}

```

Il faut ensuite ajouter le message flash afin qu'il s'affiche au niveau du form en vue,

```

{% block body %}
    <div class="container">
        <h1 class="mt-5 mb-5">Ajouter du stock pour : {{ product.name }} </h1>
        {{ include '_flash_message.html.twig' }}
        {{ form_start(form) }}

```

Et voilà à vous de tester tout cela et les différents cas de figure possible, vous allez avoir deux erreurs successive je vais vous faire gagner du temps pour celle-ci il vous faut tout simplement ajouter ceci,

```

if($stockAdd->getQuantity()>0){
    $newQuantity = $product->getStock() + $stockAdd->getQuantity();
    $product->setStock($newQuantity);

    $stockAdd->setCreatedAt(new DateTimeImmutable());
    $stockAdd->setProduct($product);
    $entityManager->persist($stockAdd);
    $entityManager->flush();
}

```

Ajouter les setter pour le `createdAt` et pour le `Product` et la tout fonctionne à merveille. La prochaine étape sera d'afficher l'historique d'approvisionnement sur la page produit, vous allez donc vous rendre sur le file `show.html.twig` et commencer par englober par un div classe `container` comme toujours. Je vous laisse mettre un peu de style et on va continuer. On va donc afficher un historique de réapprovisionnement du stock des produits, comment faire cela ?

Et bien comme toujours vous allez créer une nouvelle route qui va effectuer cela, donc sur le `ProductController`, et vous n'aurez besoin que de la méthode `GET`.

```
#Route('/add/product/{id}/stock/history', name: 'app_product_stock_add_history', methods: ['GET'])
```

Vous devrez créer une public function, avec en paramètre l'id, le `ProductRepository`, et `AddProductHistory` bien évidemment,

```

#[Route('/add/product/{id}/stock/history', name: 'app_product_stock_add_history',]
public function showHistoryProductStock($id, ProductRepository $productRepository,
methods: ['GET'])
, AddProductHistory $addProductHistory): Response

```

Voici pour le début de la fonction, ensuite on va donc crée notre logique et récupérer les infos dont on a besoin.

```

$product = $productRepository->find($id); /*on récupere le produit passé en paramètre*/
$productAddHistory = $addProductHistoryRepository->findBy(['product'=>$product], ['id'=>'DESC']);

```

On applique donc la méthode `find()` pour récupérer le produit par son id, ensuite oncrée une varaiable qui contiendra le repo et on applique la méthode `findBy()` à laquelle on applique deux tableaux en paramètre les produits et on les tri par ordre Descending. On va désormais ajouter le lien de cette route dans `Product show.html.twig`,

```


Voici pour le lien qui est fonctionnel, maintenant il reste a afficher cela. Vous allez créer un nouveau fichier que moi j'appellerai addedHistoryStockShow.html.twig, mais vous êtes libre tant que cela reste logique et cohérent. Avant de pouvoir afficher cela il faudra créer le return sur votre Controller, essayer de finaliser cela seul.


```

```

return $this->render('product/addedHistoryStockShow.html.twig', [
    "productsAdded"=>$productAddHistory
]);

```

Voici pour le return, ensuite vous allez vous rendre sur votre vue donc le fichier twig. Il va falloir créer ce fichier entièrement,

Exercice 8

Correction :

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

{% block body %}

    <div class="container">
        <table class="table">
            <tr>
                <th>Id</th>
                <th>Quantité</th>
                <th>Date d'ajout</th>
            </tr>
            {% for productAdded in productsAdded %}
            <tr>
                <td>{{ productAdded.id }}</td>
                <td>{{ productAdded.quantity }}</td>
                <td>{{ productAdded.createdAt|date }}</td>
            </tr>
            {% endfor %}
        </table>
    </div>
{% endblock %}
```

Voici pour la correction de l'exercice, là vous avez bien le résultat attendu à l'affichage, certes il n'y a pas de style mais c'est entièrement fonctionnel, nous allons modifier cela par la suite afin que cela ressemble à votre application. Modifions légèrement le filtre,

```
<td>{{ productAdded.createdAt|date('d-m-Y H:i:s') }}</td>
```

Voilà qui est déjà beaucoup mieux, pensez à tester en ajoutant du stock et retourner vérifier, ensuite nous devrons modifier certaines chose quand nous irons sur la modification du produit. Il y a la possibilité de modifier le stock et ça ce n'est pas très bon, on va donc devoir supprimer ce champ et pour faire cela il y deux possibilités. La méthode la plus simple est de créer un nouveau formulaire, donc dans votre terminal vous créer un nouveau formulaire avec la commande `symfony console make:form ProductUpdateType`, il va falloir le lier à l'entity `Product` bien évidemment, et dans form, `ProductUpdateType` et vous mettez le champ stock en commentaire,

```
$builder
    ->add('name')
    ->add('description')
    ->add('price')
    ->add('image')
    //->add('stock')
```

IL vous faudra bien sur modifier cela dans votre Controller car ce n'est pas ce form qui est appelé, donc sur l'edit, vous modifier cela.

```
#[Route('/{id}/edit', name: 'app_product_edit', methods: ['GET', 'P
public function edit(Request $request, Product $product, EntityManager
{
    $form = $this->createForm(ProductUpdateType::class, $product);
    $form->handleRequest($request);
```

Très bien allez donc tester cela en cliquant sur la modification, vous allez être confronté a un erreur, comment régler cette erreur et surtout pourquoi apparait-elle ??

Et bien car le champ stock qui existe alors que dans le form il n'existe pas, vous allez devoir faire une modification afin de régler cette erreur, je vous laisse chercher comment résoudre cela et si vous trouvez seul c'est parfait, lisez bien l'erreur vous avez les informations à l'écran.

Dans votre file `_form.html.twig` qui se trouve dans le folder Product, il faut simplement ajouter une condition sur le button `Update` afin de pouvoir faire disparaître ou apparaître ce champ.

```
{{ form_row(form.price, {'attr':{'class':'form form-control'}}) }}
{% if button_label != 'Update' %}
    {{ form_row(form.stock, {'attr':{'class':'form form-control'}}) }}
{% endif %}
{{ form_row(form.subCategories, {'attr':{'class':'form form-control'}}) }}
```

Et là il a disparu là où vous le désiriez, depuis la lise a jour du produit on ne peut plus modifier le stock, car on a récupéré le label du bouton dans la condition. IL reste quelques modification à effectuer et ce sera bon.

Maintenant pour le coté des mises à jour on a le champ image qui n'est pas encore optimisé on va donc régler cela aussi, vous allez vous rendre sur `ProductType` e récupérer tout ce qui concerne l'image.

```

->add('image', FileType::class, [
    'label' => 'Image du produit',
    'mapped' => false,
    'required' => false, /*ce n'est pas obligatoire comme
    'constraints'=>[
        new File([ /* c'est cette classe ci "Symfony\Component\HttpFoundation\File\UploadedFile"
            'maxSize' => '1024k',
            'mimeTypes' => [
                'image/jpeg',
                'image/png',
                'image/jpg',
            ],
            'maxSizeMessage'=>'Votre image ne doit pas dépasser 1024k',
            'mimeTypesMessage' => 'Veuillez choisir un fichier valide'
        ])
    ]
])

```

Et donc vous allez copier cela dans le [ProductUpdateType](#) a la place de l'ajout d'image, pensez à résoudre les erreurs en important les classes nécessaires. Ensuite il vous faudra vous rendre sur le file [edit.html.twig](#) et entourer le tout d'une dev classe container comme toujours. Il reste également à gérer la partie de l'image dans l'update du [Controller](#) vous allez donc copier cela et le coller

```

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

    $image = $form->get('image')->getData();/* on recup l'image

    if ($image) {/*si l'image existe*/
        $originalName = pathinfo($image->getClientOriginalName())
        $safeImageName = $slugger->slug($originalName);/* permet de faire une url amical */
        $newFileName = $safeImageName.'-'.$_POST['id'].'.'.$image->guessExtension();

        try {
            $image->move(
                ($this->getParameter('image_directory'),
                $newFileName);/* on recup l'image et on la sauvegarde */
        }catch (FileNotFoundException $exception) {}/*en cas d'erreur*/
        $product->setImage($newFileName);

    }
}

```

Et vous allez donc coller cela dans la partie [edit](#) comme moi, pensez à résoudre l'erreur que vous voyez sur la cap écran, a vous de chercher c'est quand même très simple. Pensez comme toujours à enregistrer, et tester cela.

Attaquons la suite, bravo à tous pour le travail accompli, nous allons passer à l'affichage des produits sur la page d'accueil, assurez vous d'avoir au moins une dizaine d'article dans votre boutique pour commencer. Vous devriez avoir une erreur lors de la création d'un nouveau produit, essayer de la résoudre seul sinon je vais vous aider évidemment.

Variable "button_label" does not exist.

Comment régler cela ? Et la comprendre et bien sur la partie [edit](#) nous avions un paramètre supplémentaire qui était le [button_label](#) mais sur le new non, il faudra donc ajouter cela.

```
{'button_label': 'Update'}
```

Sauf qu'il faudra remplacer l'Update par Ajouter par exemple car il y a une condition sur l'Update rappeler vous, et ce qui à générer cette erreur. Et voilà problème réglé.

Rendez vous sur la page d'accueil après avoir ajouté vos articles dans les différentes catégories bien sûr, cette page est vide, dans un premier temps nous allons afficher tous les produits sur la page d'accueil, et par la suite on ajoutera un menu afin de pouvoir sélectionner par catégories. Donc on se rend sur le [Controller](#) de la homepage, on va déjà ajouter la méthode GET pour commencer. Ensuite il va falloir accéder au produits et les afficher.

Exercice 9

Correction :

Donc voici le [Controller](#) modifié pour permettre la récupération et l'affichage de tous vos produits.

```
#Route('/', name: 'app_home', methods: ['GET'])
public function index(Repository $productRepository): Response
{
    return $this->render('home/index.html.twig', [
        'products' => $productRepository->findAll()
    ]);
}
```

Ensuite il faut bien sur se rendre sur la partie vue, la méthode [findAll\(\)](#) permet la récupération de tout les objets du [repository](#) et donc de l'[entity](#) Product.

Donc dans le dossier Home, vous avez un fichier [index.html.twig](#) qui gère la vue de la page d'accueil, et bien c'est donc ici que vous avez du créer vos card et votre affichage, vous avez du effacer ce qui était proposé, et le remplacer par le code de la card Bootstrap sauf ce qui ont voulu le créer eux-mêmes. On commence donc par importer notre [base.html.twig](#) puis les block nécessaire,

```

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

{% block title %}Hello HomeController!{% endblock %}

{% block body %}

```

Puis on met la card et son contenu en affichage,

```

{% block body %}
<div class="container">
    <h1 class="mb-5 mt-5">Produits</h1>
    <div class="row">
        {% for product in products %}
            <div class="col-md-3">
                <div class="card" style="width: 18rem;">
                    
                    <div class="card-body">
                        <h5 class="card-title titleProduct">{{ product.name }}</h5>
                        <p class="card-text">{{ product.description|slice(0,60) }}...</p>
                        <a href="#" class="btn btn-outline-success">Ajouter au panier</a>
                    </div>
                </div>
            </div>
        {% endfor %}
    </div>
{% endblock %}

```

Et voici pour l'affichage, c'était un bon petit défi pour vous, la chose la plus dur a dû être l'image du produit en elle-même, mais il y a plusieurs façon d'y arriver celle-ci étant la plus simple et la plus optimisé, je vous félicite donc pour votre travail, soyez fier de vous.

La suite va être de créer une page produit, lorsque vous cliquerez sur un produit vous aurez cette produit qui s'affichera avec tous les détails. Pour cela on va donc créer une nouvelle route dans votre [HomeController](#),

```

#[Route('/product/{id}/show', name: 'app_product_show', methods: ['GET'])]
public function showProduct(Product $product): Response //ici on récupère di
{
    return $this->render('home/show.html.twig', [ //il faut bien sur créer c
        'product'=>$product
    ]);
}

```

Après avoir créé le fichier twig on va faire simple pour commencer

```

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

{% block title %}Hello HomeController!{% endblock %}

{% block body %}
    <div class="container">
        <h1>{{ product.name }} </h1>
    </div>
{% endblock %}

```

Ensuite il va falloir faire une légère modification sur la card afin qu'elle soit cliquable, vous allez englober le titre,

```

<a href="{{ path('app_product_show',{'id':product.id}) }}"><h5 class="card-title titleProductLink">

```

Vous pouvez aller vérifier sur votre appli, vous allez voir le changement au niveau du titre des produits, et là on va voir que j'ai fait une erreur quelle est-elle ?

Oui le nom que j'ai donné à la route était déjà pris, une erreur bête mais les test servent à ça aussi donc on modifie sur le [Controller](#) et dans le path bien sûr. Je l'ai donc appelé [app_home_product_show](#). On test et voilà qui est mieux, on peut avancer. On a désormais accès aux produits et donc leur nom, on va bien sûr faire mieux que cela, déjà on va faire un peu de CSS sur le titre du produit car le text-decoration est bien c'est moche donc on fait un peu de CSS sur un fichier séparé.

```

.titleProductLink{
    text-decoration: none;
    color: #rgb(8, 104, 8);
}

```

Moi j'ai fait simple mais vous faites selon vos goûts, ok nous allons customiser la page produits, vous allez vous rendre sur la doc de Bootstrap et choisir un card qui sera en longueur car c'est plus agréable quand il n'y a qu'un seul produit je trouve mais vous êtes libre bien sûr, c'est votre appli et votre projet.

Exercice 10

```

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

{% block title %}Page produit{% endblock %}

{% block body %}
    <div class="container">
        <div class="card mb-12">
            <div class="row g-0">
                <div class="col-md-4">
                    
                <div class="col-md-8">
                    <div class="card-body">
                        <h1 class="card-title">{{ product.name }}</h1>
                        <p class="card-text">{{ product.description }}</p>
                        <a href="#" class="btn btn-outline-success">Ajouter au panier</a>
                        <p class="card-text"><small class="text-body-secondary">Last updated 3 mins ago</small></p>
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>

```

Voilà pour mon affichage produit, il faudra ajouter un peu de style car la card est collé en haut mais cela affiche bien ce que je vous avais demandé, donc bravo à vous tous. Ce qui sera pas mal c'est de pouvoir afficher les derniers produits ajoutés par exemple. Je vais ajouter un h2 avec ce que je vais ajouté

```

        </div>
    </div>
    <h2 class="mt-5">Dernier produits ajoutés</h2>
</div>
&dblock %}

```

Et bien sûr vous devez ensuite allez sur le [Controller](#) que l'on vient de créer pour pouvoir créer la logique d'affichage let's go.

```

$lastProductsAdd = $productRepository->findBy([], ['id'=>'DESC'], 5);/

return $this->render('home/show.html.twig', [ //il faut bien sur créer
    'product'=>$product,
    'products'=>$lastProductsAdd
]);

```

Et voilà pour la logique, il va falloir s'occuper de l'affichage ensuite, vous allez récupérer sur la page accueil à partir de la div class row jusqu'à la fin de cette div et ensuite vous rendre sur la page show donc, et vous coller cela en dessous de votre h2,

```

<h2 class="mt-5">Dernier produits ajoutés</h2>
<div class="row">
    {% for product in products %}
        <div class="col-md-3 mt-4">
            <div class="card" style="width: 18rem;">
                
                <div class="card-body">
                    <a class="titleProductLink" href="{{ path('app_home_product_show',{'id':product.id}) }...>{{ product.title }}</a>
                    <p class="card-text">{{ product.description|slice(0,60) }}...</p>
                    <a href="#" class="btn btn-outline-success">Ajouter au panier</a>
                </div>
            </div>
        </div>
    {% endfor %}
</div>

```

Et vous aurez donc les 5 derniers produits ajoutés, exactement comme vous le désiriez, bravo pour le travail accompli, soyez fier de ce que vous avez réalisé jusqu'à présent même si ce n'est pas encore fini. On va ajouter le lien vers la page d'accueil sur notre menu car on ne l'a pas encore fait, et c'est assez simple pour vous désormais donc allez-y.

Ensuite on va créer un bouton dans votre menu pour trier les produits par catégories, donc vous allez vous rendre sur votre [HomeController](#), et sur la route de la page d'accueil, donc pour faire cela on va devoir récupérer quelque chose c'est quoi ?

Le repo des catégories et donc lui attribué une variable, et donc en plus de retourner les produits il faudra retourner toutes les catégories tout simplement et vous savez désormais quelle méthode permet cela.

```

return $this->render('home/index.html.twig', [
    'products'=>$productRepository->findAll(),
    'categories'=>$categoryRepository->findAll()
]);

```

Ensuite il faudra se rendre sur la navbar donc dans le folder Layout, et vous allez récupérer ce qui se trouve en dessous du if is granted rôle admin

```

<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
        Admin
    </a>
    <ul class="dropdown-menu">
        <li><a class="dropdown-item" href="{{ path('app_category') }}>Catégories
        <li><a class="dropdown-item" href="{{ path('app_category_new') }}>Ajouter
        <li><hr class="dropdown-divider"></li>
        <li><a class="dropdown-item" href="{{ path('app_sub_category_index') }}"
        <li><a class="dropdown-item" href="{{ path('app_sub_category_new') }}>Ajouter
        <li><hr class="dropdown-divider"></li>
        <li><a class="dropdown-item" href="{{ path('app_product_index') }}>Produits
        <li><a class="dropdown-item" href="{{ path('app_product_new') }}>Ajouter
        <li><hr class="dropdown-divider"></li>
        <li><a class="dropdown-item" href="{{ path('app_user') }}>Utilisateurs
    </ul>
</li>

```

Et vous allez le coller en dessous du a qui contient le mot link

```

<a class="nav-link" href="#">Link</a>
</li>
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
        Catégories
    </a>
    <ul class="dropdown-menu">
        <li><a class="dropdown-item" href="{{ path('app_category') }}>Catégories
        <li><a class="dropdown-item" href="{{ path('app_category_new') }}>Ajouter
        <li><hr class="dropdown-divider"></li>
        <li><a class="dropdown-item" href="{{ path('app_sub_category_index') }}"
        <li><a class="dropdown-item" href="{{ path('app_sub_category_new') }}>Ajouter
        <li><hr class="dropdown-divider"></li>
        <li><a class="dropdown-item" href="{{ path('app_product_index') }}>Produits
        <li><a class="dropdown-item" href="{{ path('app_product_new') }}>Ajouter
        <li><hr class="dropdown-divider"></li>
        <li><a class="dropdown-item" href="{{ path('app_user') }}>Utilisateurs
    </ul>

```

Et donc remplacer le mot admin par Catégories puis on va modifier les liens car vous n'avez pas besoin de tout cela,

```

<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
        Catégories
    </a>
    <ul class="dropdown-menu">
        {% for category in categories %}
            <li><a class="dropdown-item" href="">{{ category.name }}</a></li>
        {% endfor %}
        <li><hr class="dropdown-divider"></li>
    
```

Voilà pour le menu, une petite boucle for in classique et on gère l'affichage donc vous sauvegarder et tester cela de suite. Parfait pour moi. Donc là on a bien les catégories j'ai personnellement déplacer le divider dans la boucle afin que cela soit plus jolie, maintenant il faut penser que vous avez des sous-catégories et bien on va s'occuper de celle-ci. Cela va être assez simple car elle sont liées aux catégories donc encore une boucle for in à l'intérieur de la première boucle for in.

```

    <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
        Catégories
    </a>
    <ul class="dropdown-menu">
        {% for category in categories %}
            <li><a class="dropdown-item" href="">{{ category.name }}</a></li>
            {% for subCategory in category.subCategories %}
                <li><a class="dropdown-item" href="">{{ subCategory.name }}</a></li>
            {% endfor %}
            <li><hr class="dropdown-divider"></li>
        {% endfor %}
    
```

Et voilà là, vous avez l'affichage des sous-catégories, on va juste ajouter un peu de style car ce n'est pas super jojo,

```

        {% for category in categories %}
            <li><a class="dropdown-item text-primary" href="">
    
```

En ajoutant cette classe, je différencie mes catégories des sous-catégories par une couleur, c'est simple mais on voit la différence, vous êtes libre de mettre le couleur de votre choix bien sûr. J'ai même ajouté un bold sur les catégories pour être sur, on va aussi faire une condition pour vérifier que cela existe avant de l'afficher afin d'être sur.

Désormais il reste à faire en sorte que lorsque l'on clique sur une catégorie cela nous amène sur la catégorie en question car à l'heure actuelle ce n'est pas fonctionnel, donc comme toujours on se rend sur le [Controller HomeController](#) et on va devoir créer une méthode pour cela,

```

#[Route('/product/subcategory/{id}/filter', name: 'app_home_product_filter', methods: ['GET'])]
public function filter($id, SubCategoryRepository $subCategoryRepository): Response //ici on récupère l'objet subCategory
{
    return $this->render('home/filter.html.twig', [ //il faut bien sûr créer ce fichier
        'products' => $subCategoryRepository->find($id)->getProducts(),
    ]);
}

```

Voilà bon ça vous avez l'habitude maintenant, rien de compliqué, ensuite il faut créer le fichier [twig](#), on le remplira par la suite, vous allez récupérer le lien de ce [Controller](#) (la route) et le mettre ou en a besoin donc dans la nav au niveau du lien des subCategory.

```

{% for subCategory in category.subCategories %}
<li><a class="dropdown-item" href="{{ path('app_home_product_filter',{'id':subCategory.id}) }}" >

```

Voilà le lien est désormais fonctionnel, alors comme toujours vous allez tester cela après avoir actualiser votre appli bien sûr. Super cela fonctionne parfaitement, maintenant vous retournez sur le Controller et vous allez commencer à gérer l'affichage,

```

$product = $subCategoryRepository->find($id)->getProducts();
return $this->render('home/filter.html.twig', [ //il faut bien sûr créer ce fichier
    'products' => $product,
]);

```

Ensuite vous allez donc pouvoir gérer l'affichage, vous allez donc simplement récupérer le code de la page index.html.twig donc votre page d'accueil et le coller sur votre filter.html.twig, et bien sûr vous allez actualiser et tester cela de suite. Et on va modifier légèrement pour afficher le nom de la sous-catégories afin que cela soit plus clair pour l'utilisateur. Donc on retourne sur le [Controller](#) et on fait cela.

```

$product = $subCategoryRepository->find($id)->getProducts();
$subCategory = $subCategoryRepository->find($id);

return $this->render('home/filter.html.twig', [ //il faut bien sûr créer ce fichier
    'products' => $product,
    'subCategory' => $subCategory,
]);

```

Et sur la vue on va donc afficher le nom de la sous-catégorie,

```

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

{% block title %}Sous-catégories{% endblock %}

{% block body %}
<div class="container">
    <h1 class="mb-5 mt-5">{{ subCategory.name }}</h1>

```

Et comme toujours on teste cela, parfait tout fonctionne bien. Il faudra penser au menu qui ne s'affiche pas sur toutes les pages donc on va gérer cela, ça va se gérer au niveau du [Controller](#) bien évidemment mais on va faire le plus simple donc sur le [Controller](#) de la route product/show.

```
'product'=>$product,
'products'=>$lastProductsAdd,
'categories'=>$categoryRepository->findAll()
```

Evidemment pensez à l'ajouter en paramètre de la fonction, et on va l'ajouter sur la route filter, de la même manière. Et comme toujours on test cela, super tout est bon, il faudra ajouter également au menu un bouton s'inscrire ou se connecter si l'utilisateur n'est pas connecté, et aussi le prix de chaque produit, donc gardons cela en tête. Donc le prix est déjà stocké quelque part, j'aimera que vous affichiez les prix de vos produits seul, à vous de jouer.

Sur le file [index.html.twig](#) qui se trouve dans le folder home, et bien vous ajoutez le prix de cette manière,

```
<p class="card-text">{{ product.description|slice(
    <h3>{{ product.price }} €</h3> <!-- altgr + e -->
```

Et voilà pour les prix, aussi simple que cela, vous allez pouvoir l'ajouter à la page qui filtre bien sûr, et également sur la page show. Testez cela et on voit qu'il n'apparaît pas sur la page du produit donc on va l'ajouter là encore, et voilà. On va s'occuper du bouton dans le cas où l'utilisateur n'est pas connecté, cela va se passer au niveau du menu donc dans [nav.html.twig](#). Vous allez vous placer au niveau du form et créer cela, allez y seul.

```
<li class="nav-item">
  <a href="{{ path('app_register') }}" class='nav-link'>S'inscrire</a>
</li>
<li class="nav-item">
  <a href="{{ path('app_login') }}" class='nav-link'>Se connecter</a>
</li>
</ul>
<form class="d-flex" role="search">
```

Voilà c'est simple mais cela fonctionne très bien, maintenant ce que l'on doit faire c'est affiché cela uniquement si l'utilisateur n'est pas inscrit ou connecté, la encore j'aimera que vous cherchiez comment faire et le faire seul, assurez-vous si vous faites la bonne recherche vous allez vite trouver.

Et voilà il suffisait de faire une condition qui demande si l'utilisateur est connecté il affiche rien sinon il affiche et oui.

```
{% if app.user == false %}
<li class="nav-item">
  <a href="{{ path('app_register') }}" class='nav-link'>S'inscrire</a>
</li>
<li class="nav-item">
  <a href="{{ path('app_login') }}" class='nav-link'>Se connecter</a>
</li>
[% endif %]
```

Bravo à vous là votre appli commence à vraiment être pas mal du tout, on va aborder la pagination, imaginons que vous ayez beaucoup de produit cela permet d'économiser la bande passante et le délai de chargement afin de ne pas alourdir votre site et d'améliorer l'expérience utilisateur.

Vous allez regarder KNP paginator et regarder un peu le fonctionnement, c'est un bundle [Symfony](#) qui est là pour la pagination, vous regarder un peu et ensuite on va installer cela ensemble.

```
composer require knplabs/knp-paginator-bundle
```

Voici la ligne d'installation vu dans la doc, et on laisse s'installer, ensuite vous devrez aller dans votre [Controller](#) d'accueil car c'est ici que vous afficher la totalité de vos produits, et on va mettre cela en place. Vous allez devoir importer la [Request](#), pour les requêtes et ensuite importer [PaginatorInterface](#), puis créer une variable qui contiendra le repo et la méthode `findby()`.

```
$categoryRepository, Request $request, PaginatorInterface $paginator)
```

Ensuite on crée la logique de la pagination et on enregistre puis on teste pour voir la différence, la customisation viendra par la suite, pas de panique.

```
$data = $productRepository->findby([], ['id'=>"DESC"]);
$products = $paginator->paginate(
    $data,
    $request->query->getInt('page', 1), //met en place la pagination
    8 //je choisi la limite de 8 articles par page
);

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

Parfait je n'ai plus que 8 produits sur ma première page donc cela fonctionne très bien, maintenant il nous manque le bouton de pagination pour changer de page, il va donc falloir l'ajouter sur notre page d'accueil, c'est très simple vous pouvez chercher cela seul encore une fois

- [Indice](#) c'est une petite ligne de code rien de plus.

```
    {% endfor %}

```

Pas plus dur que cela, et bien sûr allez vérifier comme toujours, parfait pour moi, on va donc customiser cela car c'est pas jojo et plutôt petit, et Bootstrap contient cela encore une fois, vous trouvez celui qui vous intéresse et vous le mettez en place. Vous allez devoir créer un template qui va gérer cela et l'importer à l'endroit désiré.

```
{% knp_pagination_render(products, 'layouts/pagination_template.html.twig') %}
```

Et donc créer ce fichier, vous collez le code récupérer dans Bootstrap et vous tester mais garder en tête que ce n'est pas fonctionnel il va falloir penser à gérer les pages. Il va falloir aussi penser à l'afficher uniquement si il y a plusieurs pages, et bien cela va être vraiment très simple vous allez voir.

```

{%
  if pageCount > 1 %
}
<nav aria-label="...">
  <ul class="pagination">
    {% if previous is defined %}
      <li class="page-item">
        <a class="page-link">Précédent</a>
      </li>
    {% endif %}
    {# <li class="page-item"><a class="page-link" href="#">1</a></li> #}
    <li class="page-item active" aria-current="page">
      <a class="page-link" href="#">2</a>
    </li>
    <li class="page-item"><a class="page-link" href="#">3</a></li> #
    {% if next is defined %}
      <li class="page-item">
        <a class="page-link" href="#">Suivant</a>
      </li>
    {% endif %}
  </ul>

```

Cette façon de faire est dans le cas où vous voudriez afficher seulement suivant et précédent ce qui pour ma part est beaucoup plus simple et clair. Ensuite il va falloir mettre les liens pour que cela fonctionne,

```
<a href="{{ path(route, query|merge({(pageParameterName):previous})) }}" class="page-link">Précédent
```

Là on a fait quoi ? On a bien sur ouvrir la route a partir de la requête a laquelle on va mettre le paramètre previous donc précédent, et vous tester, pour aller sur votre page deux dans l'url vous ajouter cela

```
/?page=2
```

Et vous pourrez tester votre bouton précédent, et cela fonctionne très bien. Faites le bouton suivant seul du coup, parfait tout fonctionne bien. On va essayer d'afficher le numéro des pages se sera pas mal aussi, ça va beaucoup ressembler à ce que l'on a fait jusqu'à présent.

```

{%
  for page in pagesInRange %
}
  <li class="page-item active" aria-current="page">
    <a class="page-link" href="{{ path(route, query|merge({(pageParameterName):page})) }}>{{ page }} <
  </li>
{%
  endfor %
}
```

Je n'invente rien ce sont des chose que vous pouvez trouver dans la doc quand vous faites des recherches. OK c'est parfait chez moi, pensez a tester tout les boutons bien sûr. La prochaine chose sera de montrer sur quelle page notre utilisateur se trouve car la il ne le sait pas vraiment a moins de regarder l'url et on ne fait pas comme ca par habitude on est d'accord.

```

{% for page in pagesInRange %}
  {% if page != current %}
    <li class="page-item active" aria-current="page">
      <a class="page-link" href="{{ path(route, query|merge({(pageParameterName):page}))}">
    </li>
  {% else %}
    <li class="page-item active" aria-current="page">
      <a style="background: green" class="page-link" href="{{ path(route, query|merge({(pageParameterName):page}))}">
    </li>
  {% endif %}
{% endfor %}

```

Et voilà il suffit de mettre une condition plutôt simple désormais pour vous, et mettre le numéro de page en vert par exemple, ça fonctionne très bien. Si jamais vous aviez 50pages cela afficherai tous les numéros de pages, dans ce cas la je laisserai seulement suivant précédent ou bien alors trois numéros de pages et les petits points, vous faites selon vos gouts personnel.

Parfait, jusqu'à présent tout fonctionne bien, passons au panier et à sa gestion. Donc vous allez créer un [Controller](#) pour cela,

Exercice 11

Correction :

Voilà pour la session en paramètre,

```
public function index(SessionInterface $session): Response
```

Et voici pour la fonction avec le constructeur,

```
public function __construct(private readonly ProductRepository $productRepository)
{}
```

Pas plus compliqué que cela, en revanche nous allons créer la logique pour le panier, je vous ai mis tous les commentaires.

```

public function index(SessionInterface $session): Response
{
    // Récupère les données du panier en session, ou un tableau vide si il n'y a rien
    $cart = $session->get('cart', []);
    // Initialisation d'un tableau pour stocker les données du panier avec les informations de produit
    $cartWithData = [];
    // Boucle sur les éléments du panier pour récupérer les informations de produit
    foreach ($cart as $id => $quantity) {
        // Récupère le produit correspondant à l'id et la quantité
        $cartWithData[] = [
            'product' => $this->productRepository->find($id), // Récupère le produit via son id
            'quantity' => $quantity // Quantité du produit dans le panier
        ];
    }
}

```

```

// Calcul du total du panier
$total = array_sum(array_map(function ($item) {
    // Pour chaque élément du panier, multiplie le prix du produit par la quantité
    return $item['product']->getPrice() * $item['quantity'];
}, $cartWithData));

// Rendu de la vue pour afficher le panier
return $this->render('cart/index.html.twig', [
    'items'=>$cartWithData, //on retourne ses deux variable afin de les récupérer dans la vue
    'total'=>$total
]);

```

Voici la fonction complète de la route /cart, ensuite nous allons passer à la suite donc une nouvelle route, qui va permettre d'ajouter les produits aux paniers,

```

#[Route("/cart/add/{id}/", name: "app_cart_new", methods: ['GET'])]
// Définit une route pour ajouter un produit au panier

public function addProductToCart(int $id, SessionInterface $session): Response
// Méthode pour ajouter un produit au panier, prend l'ID du produit et la session en paramètres

{
    $cart = $session->get('cart', []);
    // Récupère le panier actuel de la session, ou un tableau vide si il n'existe pas
    if (!empty($cart[$id])){
        $cart[$id]++;
    }else{
        $cart[$id]=1;
    }
    // Si le produit est déjà dans le panier, incrémenté sa quantité, sinon l'ajoute avec une quantité de
    $session->set('cart',$cart);
    // Met à jour le panier dans la session
    return $this->redirectToRoute('app_cart');
    // Redirige vers la page du panier
}

```

Et voici, la route pour l'ajout de produit, ensuite vous allez pouvoir tester la route sur la page index dans le folder home, vous allez ajouter le lien sur votre ajout au panier.

```

<h3>{{ product.price }} €</h3> <!-- altgr + e pour le sigle euro-->
<a href="{{ path('app_cart_new',{'id':product.id}) }}" class="btn b"

```

Et bien évidemment vous allez devoir l'ajouter sur toutes les pages où vous avez l'ajout au panier donc productShow, et ensuite vous tester cela. Super on est bien redirigé sur la page du panier même si c'est une page blanche, vous pouvez le voir dans l'url,

Si vraiment vous désirez le voir vous pouvez faire un dd, je vous montre où le placer.

```

// Calcul du total du panier
$total = array_sum(array_map(function (
    // Pour chaque élément du panier,
    return $item['product']->getPrice()
), $cartWithData));

dd($cartWithData);

```

Comme cela vous aurez un petit rendu qui vous confirmera cela, vous pouvez faire la même chose pour le total de la commande.

```
CartController.php on line 43:  
array:1 [▼  
  0 => array:2 [▼  
    "product" => App\Enti...\\Product {#747 ▼  
      -id: 12  
      -name: "Jordan Sport x Fédération Française de Basketball"  
      -description: "T-shirt pour homme"  
      -price: 45  
      -subCategories: Doctrine...\\PersistentCollection {#761 ►}  
      -image: "FFB-66992b4624a59.jpg"  
      -stock: 12  
      -addProductHistories: Doctrine...\\PersistentCollection {#777 ►}  
    }  
    "quantity" => 2  
  ]  
]
```

Tout fonctionne à merveille jusqu'à présent, il va ensuite falloir penser aux autres actions du panier, et l'affichage du panier bien sûr. On va s'occuper de la vue du panier et ensuite nous verrons le reste des actions possible dans le panier. Allez on attaque, donc vous vous rendez sur la partie vue, folder [cart](#) et file [index.html.twig](#), il faudra donc créer un tableau, et renseigner les éléments essentiels comme le nom du produit, le prix la quantité le total du prix et les actions possible (suppression etc..).

```
{% block body %}  
  <div class="container">  
    <h1>Panier</h1>  
    <table class="table table-bordered">  
      <tr>  
        <th>Produit</th>  
        <th>Pri...</th>  
        <th>Quantité</th>  
        <th>Pri... total</th>  
        <th>Action</th>  
      </tr>  
      </table>  
    </div>
```

Rien d'extraordinaire pour le moment, vous pouvez tester la route afin d'afficher cela, il va donc falloir boucler sur les [items](#), c'est ce que vous avez noté dans le [Controller](#), a vous d'essayer cela seul.

Moi j'ai fait cela de cette manière,

```

</tr>
{% for item in items %}
    <tr>
        <td>{{ item.product.name }}</td>
        <td>{{ item.product.price }}</td>
        <td>{{ item.quantity }}</td>
        <td>{{ item.product.price * item.quantity }}</td>
        <td>
            <a href="" class="text-danger text-decoration-none">Supprimer du panier</a>
        </td>
    </tr>
{% endfor %}

```

Vous pouvez reprendre celle-ci qui est simple à réaliser et qui fonctionne parfaitement, évidemment on teste cela, et cela fonctionne très bien. Il faut penser au cas où le panier est vide et afficher un message, donc on met une condition, à vous de jouer encore.

```

</tr>
{% else %}
    <p>Votre panier est vide</p>
{% endfor %}

```

Maintenant on va afficher le total à payer, on va le mettre dans un h2 allez-y seul encore une fois, après tout vous êtes là pour travailler eh eh, vous allez également ajouter 2 boutons qui seront valider la commande et la suppression du panier, c'est à vous.

```

<div class="row">
    <a href="" class="btn btn-primary">Valider la commande</a>
    <a href="" class="btn btn-outline-danger">Supprimer le panier</a>
</div>

```

Voilà qui est mieux, bravo à vous. Maintenant on va gérer la suppression d'un élément du panier, donc on retourne sur le [Controller](#) et on crée cette action. On va créer l'action `delete`, vous allez donc reprendre une route déjà existante et nous allons la modifier, et créer la fonction que vous appellerez `removeToCart` par exemple, il faudra l'id ainsi que la session en paramètre.

```

// Récupération du contenu du panier en session, ou initialisation à un
$cart = $sessionInterface->get('cart', []);
// Vérification si le produit à supprimer existe dans le panier
if (!empty($cart[$id])) {
    // Suppression du produit du panier
    unset($cart[$id]);
}
// Mise à jour du contenu du panier en session
$sessionInterface->set('cart', $cart);
// Redirection vers la page du panier
return $this->redirectToRoute('app_cart');

```

Vous pouvez déjà tester cette route, en mettant le path au niveau du fichier `index.html.twig` dans le dossier `cart`,

```

<td>
    <a href="{{ path('app_cart_product-remove',{'id':item.product.id }) }>
</td>

```

Pensez à aller tester sur votre panier si cela supprime bien un article choisi, super ça fonctionne parfaitement, il va rester un petit travail encore à accomplir, il faut donc créer l'action qui permet de supprimer le panier au complet, et là vous allez essayer de faire cela seul.

Exercice 12

```

#[Route("/cart/remove", name: "app_cart_remove", methods: ['GET'])]
public function remove(SessionInterface $sessionInterface): Response
{
    // Mise à jour du contenu du panier en session
    $sessionInterface->set('cart', []);
    // Redirection vers la page du panier
    return $this->redirectToRoute('app_cart');
}

```

Bien sûr il faudra mettre votre lien sur le bouton supprimer le panier et vérifier que cela fonctionne, parfait cela fonctionne bien, on pourra se dire que l'on affiche pas le prix si le panier est vide par exemple en faisant une condition même si cela n'est pas obligatoire mais allez y tentez cela seul.

```

{% if total > 0 %}
    <h2>Total: {{ total }} €</h2>
    <div class="row">
        <a href="" class="btn btn-primary col-2 m-lg-1">Valider la commande</a>
        <a href="{{ path('app_cart_remove') }}" class="btn btn-outline-danger col-2 m-lg-1">Supprimer le panier</a>
    </div>
{% endif %}

```

Et voilà pas plus difficile que cela, bravo à vous. Pour la suite nous allons voir l'idée de mettre en place des frais de livraisons, vous allez devoir créer une **entity** qui contiendra les villes ainsi que les frais associés.

Exercice 13

Allez vérifier sur votre bdd si la nouvelle table est apparue, parfait elle est là. On va donc vérifier le système d'ajout des villes, en créant un crud.

Exercice 14

Vous aurez donc tous les fichiers générés, sur votre **Controller** vous aurez déjà les premières routes de créées, vous allez reconnaître cela désormais, vous allez pouvoir tester la route `city/` et nous allons gérer la mise en forme car là cela reste assez sommaire. Donc dans le folder `city` et le file `index.html.twig`, vous allez entourer le tout d'une **div classe container** et ajouter la **classe table-bordered** au tableau, moi j'ai mis en français comme le reste de mon appli ensuite vous cliquerez sur `create new`, et donc vous allez également mettre en forme cette page, ensuite il va falloir s'occuper du formulaire un peu comme vous aviez fait sur le formulaire des produits. On va le faire directement dans le builder, donc dans `CityType.php`. Vous allez voir personnellement je trouve cela plus propre mais vous êtes libre de choisir,

```

$builder
    ->add('name', null, [
        'required'=>'true', //si besoin d'obliger le champ ou pas true/false
        'label'=>'Nom de la ville',
        'attr'=>['class'=>'form form-control', 'placeholder'=>'Nom de la ville']
    ])
    ->add('shippingCost', null, [
        'required'=>'true',
        'label'=>'Frais de livraison',
        'attr'=>['class'=>'form form-control']
    ])

```

Là déjà votre formulaire est beaucoup plus propre et professionnel, maintenant on va gérer les boutons qui se trouve dans le file `_form.html.twig` car c'est un include.

```

{{ form_start(form) }}
    {{ form_widget(form) }}
    <button class="btn btn-outline-primary mt-4 mb-2">{{ button_label|default('Enregistrer') }}</button>
{{ form_end(form) }}

```

Moi j'ai mis cela, ensuite vous créez une ville et son frais de port afin de vérifier que tout fonctionne, c'est parfait pour moi. Ensuite vous allez passer à l'edit, pensez à modifier vos fichier twig avec la div classe container et mettre en français si vous avez envie, vous avez quelques fichier à modifier là. Pensez que c'est encore juste l'éditeur qui aura accès à cela il faut donc modifier cela dans le [Controller](#) également, je vous laisse faire cela, vous savez faire maintenant, parfait tout fonctionne très bien, la suite va être un peu plus sympa, il va falloir penser à récupérer toutes ces infos pour les appliquer et afficher à la commande afin de pour livrer vos clients. Et surtout s'occuper de stocker ces infos de commande en bdd car pour le moment ce n'est pas le cas. Essayer de rajouter quelques produits dans votre panier déjà, ensuite vous allez créer une nouvelle entity, [order](#) qui gérera les commandes, vous connaissez la commande maintenant. Il faudra comme propriété :

- Le `firstName` vous faites entrée sur tout le reste
- Le `lastName` vous faites entrée sur tout le reste
- Le `phone` vous faites entrée sur tout le reste
- Et l'`adresse` vous faites entrée sur tout le reste
- La `city` on prendra la ville qui sera en fonction des frais de livraisons, et là vous choisirez dans le type `ManyToOne` et vous lierez ce champ à la table `City`, ensuite il demande si cela peut être null vous pouvez dire oui, (si la personne veut venir chercher son colis au magasin par exemple), ensuite entrée une fois, puis après ce sera le champ à l'intérieur de la table `city` vous noterez le choix proposé donc entrée (ça devrait être `orders`)
- Le `createdAt`, il vous propose le type `datetime_immutable` vous faites entrée, et encore entrée

Et voilà votre table est finie, vous pouvez vérifier vos fichiers créer, maintenant il va falloir créer le [Controller](#) qui va pouvoir gérer tout cela. Vous l'appellerez [OrderController](#). Ensuite vous allez tester la page [order](#), vous verrez bien qu'elle fonctionne. Allons donc supprimez ce qui ne va pas dans ce template, et vous allez faire ce que vous avez désormais l'habitude votre classe container, et ensuite préparer l'affichage.

```

{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-8">
                <h1 class='mt-4 mb-4'>Commande</h1>
            </div>
            <div class="col-4">

            </div>
        </div>
    </div>
    {% endblock %}

```

Voilà pour le début, et maintenant il va falloir créer un formulaire, que vous appellerez OrderType, vous l'associerez évidemment à l'entité Order. Ensuite allez vérifier ce formulaire, il faudra ensuite bien sur l'associer avec votre [Controller](#), J'aimerais que vous essayez cela seul vu que vous l'avez fait plusieurs fois maintenant, et bien sur affichez ce formulaire dans la vue, en créant le formulaire.

```

#[Route('/order', name: 'app_order')]
public function index(Request $request): Response
{
    $order = new Order();
    $form = $this->createForm(OrderType::class, $order);
    $form->handleRequest($request);

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

```

Voici pour le Controller, passons à la vue, dans le formulaire je vais mettre un input de type submit.

```

{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-8">
                <h1 class='mt-4 mb-4'>Commande</h1>
                {{ form_start(form) }}
                    <input type="submit" value="Continuer">
                {{ form_end(form) }}
            </div>
            <div class="col-4">

            </div>
        </div>
    </div>

```

Voilà, là on affiche bien notre formulaire, il faudra gérer la mise en forme évidemment, vous pouvez mettre une classe `btn` sur votre bouton submit. Ok on va s'attaquer à la mise en forme, vous allez sur `OrderType` dans le builder et vous mettez cela en forme comme la dernière fois. Le `createdAt` n'est pas nécessaire vous pouvez le commenter (`ctrl+k+c`). On va le passer directement à partir du [Controller](#).

```

$builder
    ->add('firstName', null, [
        'attr'=>[
            'class'=>'form form-control'
        ]
    ])

```

Il suffisait d'ajouter cela à chaque champ restant, et il reste à afficher le nom des villes plutôt que leur Id, à vous de jouer. Et voilà, super. On pourra vouloir afficher le récapitulatif de la commande sur un des côtés, vous allez revenir sur le [OrderController](#), et ajouter en paramètre de la fonction le [SessionInterface](#) et sa [variable](#). Et ton va devoir injecter les informations du panier à ce niveau-là, en fait vous allez juste copier/coller la partie du [Controller CartController](#) de la route `/cart` qui contient les infos nécessaires et voilà. Et modifier très légèrement, au lieu du `$this` vous aurez donc le `$productRepository` ce qui revient au même en soi mais on précise pour être tranquille. Donc comme dit précédemment nous allons gérer les frais de livraisons en fonction de la ville e manière dynamique, et donc faire cela en Javascript. Rappelez-vous dans le [OrderController](#) vous avez dans le fucntion index fait en sorte de gérer les produits du panier, la quantité et le total dans la session, les fris de livraisons sont stockées dans votre table city.

```

#[Route('/order', name: 'app_order')]
public function index(Request $request, SessionInterface $session, ProductRepository $productRepository)
{
    // Récupère les données du panier en session, ou un tableau vide si il n'y a rien
    $cart = $session->get('cart', []);
    // Initialisation d'un tableau pour stocker les données du panier avec les informations de produit
    $cartWithData = [];
    // Boucle sur les éléments du panier pour récupérer les informations de produit
    foreach ($cart as $id => $quantity) {
        // Récupère le produit correspondant à l'id et la quantité
        $cartWithData[] = [
            'product' => $productRepository->find($id), // Récupère le produit via son id
            'quantity' => $quantity
        ];
    }
}

```

Et donc ajouter déjà le total de la somme à payer afin de l'afficher à la vue dans un h3 ou h4 par exemple,

```

</div>
<div class="col-4 mt-5">
    <h3>Montant à payer: {{ total }} €</h3>
</div>

```

A vous de gérer votre mise en forme à votre convenance, en pensant qu'il y aura les frais de livraisons également je vais donc modifier cela moi aussi,

```

<div class="col-4 mt-5">
    <span class='mb-2'>Montant à payer: </span>
    <h3>{{ total }} €</h3>
    <span>Frais de livraison</span>
</div>

```

Il va donc falloir penser à l'affichage des frais de livraisons également du coup, ajouter plusieurs villes afin de tester cela car on va devoir gérer les frais en fonctions des villes différentes. Comment feriez-vous cela ??

Si vous utiliser l'inspecteur vous verrez dans le select on a la ville et la valeur, si le user change la ville on va récupérer la value donc l'id, on va devoir faire une requête qui va récupérer les frais de livraisons qui sont en rapport avec la ville et on va faire cela en Javascript, et même en Jquery exceptionnellement, il va donc falloir ajouter Jquery à votre projet, pour cela rien de plus simple, on se rend sur la doc <https://releases.jquery.com/>, vous allez sélectionner minified sur la version la plus récente cela permet d'importer jquery simplement sur votre projet, et vous copier le lien juste avant votre balise script sauf si vous désirez l'utiliser sur toutes les pages dans ce cas là se sera sur base.html.twig, et vous allez créer une alert hello world afin de vérifier si cela fonctionne bien.

```

<script src="https://code.jquery.com/
<script>
    $(document).ready(function(){
        alert('hello world')
    });
</script>

```

Vous actualiser votre page afin d'afficher l'alert, ok parfait cela fonctionne bien, donc le document est bien chargé il va falloir récupérer la valeur de ce champ, donc la ville est sa valeur, pensez que l'inspecteur vous montre tout, donc sur le select vous avez les infos que vous recherchez,

```
$(document).ready(function(){
    const citySelector = $('#order_city');
    const cityValue = citySelector.val();

})
```

Ce n'est pas fini qu'on soit bien d'accord, on va également devoir créer une route qui affiche le prix par rapport à la ville donc on se rend sur le [Controller](#) et on crée cela,

```
#[Route('/city/{id}/shipping/cost', name: 'app_city_shipping_cost')]
public function cityShippingCost(City $city): Response
{
    dd($city);
}
```

J'ai juste fait la base, et un dd, on peut alors retourner sur notre Js et continuer notre travail,

```
$(document).ready(function(){
    const citySelector = $('#order_city');
    const cityValue = citySelector.val();

    const url = `http://localhost:8000/city/${cityValue}/shipping/cost`;

    //requete ajax
    $.ajax({
        url:url,
        type:'GET',
        success:function(response){

        },
        error:function(xhr,status,error){

        }
    })
})
```

On a donc créé notre début de logique ainsi que notre requête ajax, on va tout détailler pas de panique, ensuite on retourne sur le [Controller](#) et on modifie le dd car cela ne sera plus bon.

```

#[Route('/city/{id}/shipping/cost', name: 'app_city_shipping_cost')
public function cityShippingCost(City $city): Response
{
    $cityShippingPrice = $city->getShippingCost();

    return new Response($cityShippingPrice);
}

```

La dans le corps de la requête il vous donnera le prix que vous avez défini dans votre entity, on retourne sur notre jquery,

```

success: function(response) {
    const newResponse = JSON.parse(response)
    if (parseInt(newResponse.status) === 200){
        console.log(newResponse.status)
    }
},

```

On va détailler encore une fois, et on va modifier l'affichage évidemment,

```

<span>Frais de livraison</span>
<h3 id="shippingCost"></h3>

```

Pour le moment il est vide bien sûr, on va utiliser cette id, et donc s'en servir afin de renvoyer le prix du frais de livraison grâce à la réponse de la requête,

```

success: function(response) {
    const newResponse = JSON.parse(response)
    if (parseInt(newResponse.status) === 200){
        console.log(newResponse.status)
        $("#shippingCost").text(newResponse.content)
    }
},

```

Là on va donc injecter à l'intérieur de notre balise h3 avec l'id donc en fait entre les crochets, le prix qui vient du serveur, de la requête, de la réponse. Si vous retournez sur votre appli et que vous actualisez et bien le montant des frais de livraisons va apparaître. Donc cela n'est pas chargé depuis le Controller mais depuis le serveur, après que la page soit chargée une requête est envoyée au serveur à travers la route que vous avez définie (la const url) et cette requête va renvoyer une réponse en json et vous l'avez changé en text puis vous l'afficher. On va donc légèrement modifier afin de préciser que c'est en euro si vous avez choisi l'euro bien sûr,

```

<h3>
    <span id="shippingCost"></span>
    €
</h3>

```

Et voilà pour moi, maintenant il reste à régler le souci que si l'on change la ville le changement n'est pas automatique on doit actualiser la page ce n'est pas tip top donc on va régler cela. Retournons sur le script et faisons cela,

```
        },
        // Fonction appelée en cas d'erreur de la req
        error: function(xhr, status, error) {
            // À remplacer par le code à exécuter en cas
            //
        }
    })
    citySelector.on('change',function() {
        alert('hello world')
    })
}
```

On a fait quelques modifications, essayer donc d'actualiser votre appli et changer la ville vous allez vite comprendre, en fait si la valeur change cela va afficher votre alert. Comme vous voulez que la valeur change donc la ville et bien cela doit effectuer une nouvelle requête afin de vous retourner la nouvelle réponse donc les nouveaux tarifs des frais de livraisons.

```
        }
    })
    citySelector.on('change',function() {
        alert($(this).val())
    })
}
```

La vous verrez que vous récupérez la valeur qu'il y a en bdd, donc comme dit précédemment à chaque changement on va avoir une requête pour afficher le changement de valeurs des frais de livraisons ce sera dommage de répéter nos requêtes à chaque fois, donc on a une solution, laquelle ??

Et bien mettre notre requête dans une fonction ce qui évitera les répétitions, je crée donc une fonction et je mets la requête dedans en lui passant en paramètre l'url bien sûr,

```
function ajaxRequest(url) {
    // Envoie une requête AJAX de type GET à l'URL
    $.ajax({
        // L'URL de la requête
        url: url,
        // Le type de requête (ici, GET)
        type: 'GET',
        // Fonction appelée en cas de succès de la requête
        success: function(response) {
            const newResponse = JSON.parse(response)
        }
    })
}
```

Ensuite on doit bien sûr appeler cette fonction,

```

        })
    }

    ajaxRequest(url)

}

citySelector.on('change',function() {

```

Et on actualise pour vérifier cela, parfait tout fonctionne, ensuite vous allez créer une constante, à laquelle on donne le champ du select,

```

citySelector.on('change',function() {
    const urlUpdate = `http://localhost:8000/city/${$(this).val()}/shipping/cost`;
    alert(urlUpdate)
})

```

Et bien sur le alert pour vérifier cela vous verrez que l'id change bien donc c'est ok, on va pouvoir mettre notre appel de fonction à la place du alert bien sûr,

```

citySelector.on('change',function() {
    const urlUpdate = `http://localhost
//alert(urlUpdate)
    ajaxRequest(urlUpdate)
})

```

Et comme toujours vous allez tester cela, tout fonctionne comme vous le désiriez, on aurai pu écrire notre Js, dans un fichier bien spécifique mais là cette requête ne sera consommé qu'à cet endroit donc ce n'est pas grave du tout. Maintenant il reste à afficher le prix total à payer, donc le prix de la commande plus le montant des frais de livraisons. On va donc ajouter une nouvelle ligne pour cela,

```

<h3>
    <span id="shippingCost"></span>
    €
</h3>
<span>Montant total à payer</span>
<h3>
    <span class="total_Price"></span>
</h3>

```

Par exemple de cette manière, et donc le total nous allons l'afficher dans un span avec un id de manière à pouvoir le récupérer facilement,

```

<h3>
    <span id='card_Price'>{{ total }}</span>
    €</h3>
    <span>Frais de livraison</span>

```

En ayant fait de cette façon vous allez pouvoir récupérer la valeur en js, on va de ce pas modifier notre fonction,

```
console.log(newResponse.status)
$("#shippingCost").text(newResponse.content)

const cardPrice = parseInt($('#card-price').text());
console.log(cardPrice)
```

Avec le petit console.log pour vérifier que cela retourne la bonne information, ok pour moi. Maintenant on va récupérer cela et le convertir pour l'affichage,

```
const cardPrice = parseInt($('#card-price').text());
const shippingCost = parseInt($('#shippingCost').text());

console.log(cardPrice)
console.log(shippingCost)
```

On récupère bien toutes les infos on a plus qu'à les insérer dans le span,

```
const cardPrice = parseInt($('#card-price').text());
const shippingCost = parseInt($('#shippingCost').text());
$('.total-price').text(cardPrice+shippingCost);
```

Voilà pour l'opération mathématique et l'affichage tout fonctionne bien, j'ai ajouté le sigle euro bien sûr, vous pouvez tester en changeant la ville cela se calcule directement, c'est parfait.

Ok maintenant on va devoir s'occuper de la partie commande est donc récupérer ses informations de commande et les envoyer en base de données, et on s'occuper de voir si le client souhaite payer a la livraison ou en ligne également. Il va falloir donc modifier le formulaire de validation de commande afin de demander si le paiement se fera donc a la livraison ou en ligne, déjà on va faire en sorte que sur la page /cart on puisse valider la commande le **path** n'est pas encore mis en place donc on va commencer par cela, vous allez donc récupérer la route dans votre **Controller Order** et récupérer la route sur /order, et vous l'ajouter dans la validation du panier, à vous de jouer.

```
<div class="row">
  <a href="{{ path('app_order')}}" class="btn btn-primary col-2 m-lg-1">Valider la commande</a>
  <a href="{{ path('app_cart_remove')}}" class="btn btn-outline-danger col-2 m-lg-1">Supprimer
</div>
```

Et vous tester bien sûr, tout fonctionne, mais pour certains il est possible qu'au chargement le frais de livraisons ne s'affiche pas directement vous allez rajouter un attribut a votre **path**, qui est **data-turbo='false'**, c'est un attribut qui réactive les ancêtres lorsqu'ils sont retirés donc ça palie a cette erreur possible. Donc la au click au lieu de charger la page et d'avoir un loader ça utilisera un turbo donc cela doit résoudre l'erreur, cela recharge le composant mais pas toute la page, parfait. La prochaine étape va donc être de rajouter l'option de savoir si le visiteur souhaite payer en ligne ou directement à la livraison, vu que ce formulaire (order) est lié à votre **entity Order** on va rajouter une propriété a l'entity ce sera un booléen pour le bouton, pour éviter les erreurs vous placer sur l'entity

afin de ne pas faire d'erreur dans le nom, et donc création de l'entity Order et comme elle existe déjà cela va vous proposer d'ajouter un champ, vous vous rappeler de la commande ??

Symfony console make:entity Order

Et le nouveau champ sera nommée **payOnDelivery**, type **boolean**, null on dit non et entrée plus les migrations complète. Et bien sur vous vérifier en base de donnée, ensuite si vous vous rendez sur votre formulaire ce champ n'est pas présent il faut donc l'ajouter, à vous de jouer.

Donc Form, OrderType et on ajoute le champ,

```
    ])
->add('payOnDelivery')
```

Vu que c'est un type booléen Symfony va automatiquement générer une checkbox, vous pouvez ajouter des attribut si vous le désirez,

```
->add('payOnDelivery', null, [
    'label'=>'Payez à la livraison'
```

Si vous désirez une appli en français et bien c'est comme cela que vous changer les labels, voilà c'est déjà bien, il va falloir se rendre sur le [Controller](#) et gérer la soumission du formulaire car vous ne l'aviez pas fait jusqu'à présent. A vous de jouer vous savez faire désormais.

Ensuite il faut penser que lors de l'enregistrement de la commande, il va falloir penser que les produits qui sont dans la commande eux ne seront pas stocké comment gérer cela ?

Et bien simplement en créant une nouvelle table qui sera liée à la table commande et qui contiendra les produits de la commandes ce qui nous permettra par la suite de pouvoir filtrer tout cela, alors c'est parti pour la création de la nouvelle [entity](#).

Exercice 15

Parfait vous avez créer votre table, on va désormais ajouter a la table [Order](#) le prix total de la commande que l'on avait pas mis, donc comme si vous créer la table [Order](#) et on ajoute le champ, totalPrice qui sera de type float, est ce que le champ peut-être null et bien non ensuite on valide et donc les migrations. Comme toujours allez vérifier en Bdd, on va pouvoir faire les manipulations sur le Controller, dans notre validation de formulaire que l'on a laissé en suspens.

Il faudra ajouter un paramètre a la fonction qui sera

```
EntityManagerInterface $entityManager)
```

Afin de pouvoir récupérer les infos nécessaires dans les entity pour le formulaire, on aura besoin également de récupérer du code qui est dans cart mais bon en symfony comme d'autre langage on va éviter les copie de code, cela fonctionnera parfaitement mais on a plutôt tendance a créer un [Service](#) qui permet d'importer un bout de code plusieurs fois c'est une très bonne pratique donc on va voir cela ensemble de suite.

Donc vous vous rendez sur le src et vous créer un folder Service, dans ce folder vous allez créer une classe, que vous appellerez Cart,

```
<?php  
  
namespace App\Services;  
  
class Cart{  
  
}
```

Voilà qui est prêt à être rempli, on va pouvoir le customiser, vous allez devoir ajouter le constructeur au-dessus,

```
public function __construct(private readonly ProductRepository $productRepository){  
}
```

C'est celui qui est dans cart donc copier-coller et ensuite dans la [function getcart](#) vous aller copier-coller le code de cart , je vous montre lequel vous intéresse, vous allez commencer ici

```
// Recupere les donnees du panier en session  
$cart = $session->get('cart', []);
```

Et finir à ce niveau ci ou vous aurez certainement besoin d'ajouter la dernière ligne que vous verrez sur la cap écran, le return

```
// Calcul du total du panier  
$total = array_sum(array_map(function ($item) {  
    // Pour chaque élément du panier, multiplie le prix du produit par la quantité  
    return $item['product']->getPrice() * $item['quantity'];  
}, $cartWithData));  
  
return [  
    'cart' => $cartWithData,  
    'total' => $total  
];
```

Car la fonction prend renvoie un array et a donc besoin d'un return, voilà qui est mieux, maintenant sur le [CartController](#), vous pouvez supprimer cette partie de code que vous venez de dupliquer dans le Service, et donc injecter ce service,

```
public function index(SessionInterface $session, Cart $cart): Response  
{  
  
    $data = $cart->getcart($session);
```

Voilà pour le début, vu que dans ce service on retourne un tableau avec les deux clés `cart` et `total`, on va devoir modifier légèrement notre code sur le `CartController` afin de tout faire fonctionner parfaitement.

```
$data = $cart->getCart($session);

// Rendu de la vue pour afficher le panier
return $this->render('cart/index.html.twig', [
    'items'=>$data['cart'], //on retourne ses données
    'total'=>$data['total']

]);
```

Et voilà il nous reste à tester que tout fonctionne, pour cela il suffit d'aller sur votre page cart et d'actualiser et vérifier que rien n'a bougé. Il est possible que vous ayez des erreurs car votre service n'est pas existant dans le dossier config/services.yaml à vous de débuguer cela eh eh. Ensuite tout votre code fonctionnera parfaitement, il va falloir également injecter votre service sur le `OrderController`,

```
#[Route('/order', name: 'app_order')]
public function index(Request $request, SessionInterface $session,
    ProductRepository $productRepository,
    EntityManagerInterface $entityManager,
    Cart $cart): Response
```

OK on va ensuite finaliser la soumission du formulaire et gérer l'envoie de toutes les données en Bdd, mais première chose vous allez pouvoir importer le service afin de gagner de la place et faire du clean code,

```
$data = $cart->getCart($session);

$order = new Order();
$form = $this->createForm(OrderType::class, $order);
$form->handleRequest($request);
```

Voilà qui est importé, on va pouvoir continuer le form,

```

if ($form->isSubmitted() && $form->isValid()) {
    if($order->isPayOnDelivery()) {
        //dd($order);
        $order->setTotalPrice($data['total']);
        $order->setCreatedAt(new \DateTimeImmutable());
        $entityManager->persist($order);
        $entityManager->flush();
    }
}

```

On met à jour les infos, on persist et on flush, parfait. Après on doit passer au return,

```

return $this->render('order/index.html.twig', [
    'form'=>$form->createView(),
    'total'=>$data['total'],
]);

```

Et la-vous allez actualiser votre panier, remplir le form et tester si cela envoie bien les données en Bdd, pour moi tout fonctionne à merveille. Vous noterez que le prix qui se met en bdd est le montant du panier hors frais, mais ce n'est pas suffisant car cela fonctionne uniquement si la checkbox est coche car seule la condition a été prise en compte. OK vous allez pouvoir commenter le flush, et faire un petit dd du panier afin de vérifier les informations.

```

$entityManager->persist($order);
//$entityManager->flush();

dd($data['cart']);

```

On va devoir stocker tout cela dans une variable, et le setter,

```

$orderProduct = new OrderProducts();
$orderProduct->setOrder($order);

```

Et bien sûr vous allez devoir boucler sur cette ensemble de produits,

```

foreach($data['cart'] as $value) {
    $orderProduct = new OrderProducts();
    $orderProduct->setOrder($order);
}

```

On aura aussi besoin du produit, et son identifiant, et la quantité, ensuite le persist et le flush, voici le bloc de code au complet,

```

if($order->isPayOnDelivery()) {

    $order->setTotalPrice($data['total']);
    $order->setCreatedAt(new \DateTimeImmutable());
    //dd($order);
    $entityManager->persist($order);
    $entityManager->flush();

    foreach($data['cart'] as $value) {
        $orderProduct = new OrderProducts();
        $orderProduct->setOrder($order);
        $orderProduct->setProduct($value['product']);
        $orderProduct->setQuantity($value['quantity']);
        $entityManager->persist($orderProduct);
        $entityManager->flush();
    }
}

```

Là cela fonctionne très bien, bravo à vous. Là on va devoir modifier quelques petites choses,

```

if($order->isPayOnDelivery()) {

    if(!empty($data['total'])) {
        $order->setTotalPrice($data['total']);
        $order->setCreatedAt(new \DateTimeImmutable());
        //dd($order);
        $entityManager->persist($order);
        $entityManager->flush();

        foreach($data['cart'] as $value) {
            $orderProduct = new OrderProducts();
            $orderProduct->setOrder($order);
            $orderProduct->setProduct($value['product']);
            $orderProduct->setQuantity($value['quantity']);
            $entityManager->persist($orderProduct);
            $entityManager->flush();
        }
    }
}

```

Donc on a mis une condition si le panier n'est pas vide on peut enregistrer cela, sinon on retourne au panier vidé.

```

        $entityManager->flush();
    }

}

// Mise à jour du contenu du panier en session
$session->set('cart', []);
// Redirection vers la page du panier
return $this->redirectToRoute('app_cart');

```

Ensuite vous effacez les données de la table `order_product` puis celle de `order` et vous recommencez le processus de validation afin de vérifier ce que vous venez de faire. On est bien redirigé vers le paniervidé le visiteur ne pourra pas commander deux fois le même panier aucune erreur possible. Ce que l'on peut faire éventuellement créer une page en disant que la commande à bien été envoyé, afin d'informé le client. Donc `template ->order` et on crée un fichier, allez-y seul. Vous allez créer le fichier `twig`, plus la route dans le `orderController`,

```

#[Route('/order_message', name: 'order_message')]
public function orderMessage():Response
{
    return $this->render('order/order_message.twig');
}

```

Voici la route que j'ai créée, et je la place a la place de la redirection vers le panier vide qui n'as pas beaucoup d'intérêt.

```

// Mise à jour du contenu du panier en session
$session->set('cart', []);
// Redirection vers la page du panier
return $this->redirectToRoute('order_message');

```

Maintenant il reste à customiser votre fichier `twig`, je vous laisse également gérer cela seul eheh !!

On va maintenant laisser la possibilité a l'admin ou l'éditeur de pouvoir consulter les commandes, il faudra donc créer une nouvelle route pour cela, je vais vous laisser essayer de faire cela vous-mêmes, on restera bien sur dans le `OrderController`, et ensuite créer donc le template.

```

#[Route('/editor/order', name: 'app_orders_show')]
public function getAllOrder():Response
{
    return $this->render('order/orders.html.twig');
}

```

Et donc son `Template` associé, ensuite il faut penser que pour récupérer toutes les commander (`orders`) il faut appeler quelque chose dans votre route du Controller, c'est quoi ??

Exercice 16

Correction exercice 16

```
#Route('/editor/order', name: 'app_orders_show')
public function getAllOrder(OrderRepository $orderRepository):Response
{
    $orders = $orderRepository->findAll();
    //dd($orders);
    return $this->render('order/order.html.twig', [
        "orders"=>$orders
    ]);
}
```

Désormais je passe au fichier twig,

```
{% block body %}
    <h1>Commandes</h1>
    <table class="table table-bordered">
        <tr>
            <th>Nom</th>
            <th>Prénom</th>
            <th>Téléphone</th>
            <th>Adresse</th>
            <th>Ville</th>
        </tr>
    </table>
{% endblock %}
```

Voilà qui pas trop mal pour un début, ensuite il faudra penser que nous n'avons pas forcément qu'une seule commande en base de données donc il faudra trouver un moyen de gérer cela, et quoi de mieux qu'une boucle et oui.

```
<h1>Commandes</h1>
{% for order in orders %}
    <table class="table table-bordered">
        <tr>
            <th>Nom</th>
            <th>Prénom</th>
            <th>Téléphone</th>
            <th>Adresse</th>
            <th>Ville</th>
        </tr>
    </table>
{% endfor %}
```

Voilà qui est mieux, ensuite on va afficher les infos,

```

{% for order in orders %}
    <table class="table table-bordered">
        <tr>
            <th>Nom</th>
            <th>Prénom</th>
            <th>Téléphone</th>
            <th>Adresse</th>
            <th>Ville</th>
        </tr>
        <tr>
            <td>{{ order.firstName }}</td>
            <td>{{ order.lastName }}</td>
            <td>{{ order.phone }}</td>
            <td>{{ order.adress }}</td>
            <td>{{ order.city.name }}</td>
        </tr>
    </table>
    {% endfor %}

```

Voilà le fichier `twig` fini chez moi, avec la personnalisation, à la suite des cap écran du dessus bien sûr,

```

</table>
<h5 class='text-primary'>Corps de la commande</h5>
<table class="table table-bordered">
    <tr>
        <th>Nom du produit</th>
        <th>Prix du produit</th>
        <th>Quantité</th>
        <th>Prixt total</th>
    </tr>
    {% for product in order.orderProducts %}
        <tr>
            <td>{{ product.product.name }}</td>
            <td>{{ product.product.price }} €</td>
            <td>{{ product.quantity }}</td>
            <td>{{ product.product.price*product.quantity }} €</td>
        </tr>
    {% endfor %}
</table>
<h5>Frais de livraisons : {{ order.city.shippingCost }} €</h5>
<h5 class='mb-5'>Total à payer : {{ order.totalPrice }} €</h5>

```

Et bien évidemment la première boucle for se termine ensuite, voilà pour moi. Par souci du détail je vais également afficher les images des produits, ok super, on pourra par la suite ajouter commande livré ou non, ou bien supprimer la commande par exemple.

```
<span>Frais de livraisons : {{ order.city.shippingCost }} €</span>
<h5 class='mb-5'>Total à payer : {{ order.totalPrice }} €</h5>
<a href="" class='btn btn-outline-primary col-2 m-lg-1'>Marquer comme livrée</a>
<a href="" class='btn btn-danger col-2 m-lg-1'>Supprimer la commande</a>
<hr>
```

ON pourra également mettre en place une pagination car si vous avez plus de 10 commandes cela peut être très lourd à afficher. Rappelez vous nous l'avons mis en place sur la [Home page](#), alors servez vous de l'existant.

Exercice 17

Correction

Il suffisait de récupérer le bout de code que vous aviez mis en place sur la [Home page](#) et de le modifier légèrement pour ce [Controller](#).

```
$products = $paginator->paginate(
    $data,
    $request->query->getInt('page', 1), //met en place la pagination
    8 //je choisi la limite de 8 articles par page
);
```

Et on adapte cela, vous aurez donc besoin d'importer Request et PaginatorInterface évidemment, ensuite vous aurez besoin d'injecter \$order et non plus \$data, ensuite vous avez la méthode `findAll()`, et là encore cela ne convient pas il faudra la méthode `findBy()` car vous n'aurez pas besoin de toutes les order mais seulement de récupérer les commandes par leur date de création donc je dirai de la plus récente à la plus ancienne. Ensuite par question de logique et simplicité nous allons modifier les noms de variables mais rien d'obligatoire. Je vais renommer \$orders par \$data et \$products par \$orders ce qui sera donc renvoyé au twig. Ce qui donnera ceci,

```
$data = $orderRepository->findBy([], ['id'=>'DESC']);
//dd($orders);

$orders = $paginator->paginate(
    $data,
    $request->query->getInt('page', 1), //met en place la pagination
    6 //je choisi la limite de 6 commandes par page
);

return $this->render('order/order.html.twig', [
    "orders"=>$orders
]);
```

Voilà qui est prêt, vous pouvez voir que cela ressemble fortement a ce que vous aviez mis en place sur la [HomePage](#), désormais vous allez actualisez votre appli et vérifier que tout cela fonctionne

même si je sais que c'est bon, Bravo à vous tous. Il est normal que vous ne voyez pas le bouton de pagination, pourquoi ??

ET bien c'est simple déjà vous demandez (enfin-moi vous je ne connais pas vos préférences de paramétrages) 6 commandes par pages donc c'est normale j'en ai que 3 donc je vais modifier à 2 pour voir un résultat déjà, ok cela ne m'en affiche que deux, parfait. Ensuite rappelez vous qu'il y a l'affichage et que cela aussi cela existe déjà donc vous récupérez le code et vous le mettez au bon endroit, en modifiant ce qui à besoin de l'être bien sûr.

```
{% knp_pagination_render(products, 'layouts/pagination_template.html.twig') %}
```

Maintenant donc on modifie, et cela donne,

```
{% knp_pagination_render(orders, 'layouts/pagination_template.html.twig') %}
```

Pourquoi `orders`, et bien car on boucle sur `orders` donc c'est bien là-dessus que l'on va paginer, et voilà tout est mis en place et tout fonctionne parfaitement, pensez à remettre vos paramètres de référence car nous avons mis 1 ou 2 afin de tester et vu que cela fonctionne je vais remettre 6 moi personnellement.

Nous allons maintenant nous occuper des deux boutons qui sont sous les commandes, le marquage en livré et la suppression. Cela reste à la charge de l'administrateur ou de l'éditeur, on va devoir ajouter un nouveau champ dans la table `order`, qui va permettre de dire si la commande a été livré ou pas.

Donc `symfony console make:entity Order`, il va vous dire que cela existe déjà, on mettra le champ `isCompleted`, comme on met `is` devant il détecte que cela devrait être un booléen, on valide, il demande si cela peut-être être null en bdd on dira oui, et vous pourrez lancer la migration complète.

Ensuite il faudra allez dans le vue pour afficher cela, donc dans le file `order.html.twig` évidemment,

```
{% for order in orders %}
    <h2>Commande n° {{ order.id }}</h2>
    {% if order.isCompleted ==true %}
        <span class='text-success'>Commande livrée</span>
    {% else %}
        <span class='text-danger'>Commande non livrée</span>
    {% endif %}
    <br>
    <span class='text-primary'>Informations du client</span>
```

A cela personnellement je vais ajouter un filtre de date, vous faites comme vous préférez je vous le met et vous décidez.

On va maintenant gérer le bouton de commande livrée, pour cela vous allez créer une nouvelle route, je vais vous donner la route afin que l'on ai tous la même par souci de praticité, mais je vais vous laisser essayer de faire le suite seul

```
#[Route('/editor/order/{id}/is-completed/update', name: 'app_orders_is-completed-update')]
```

Essayer d'inclure un message flash qui précisera que la modification a bien été faites en cas de réussite, et une redirection vers la page `order-show`.

```

#[Route('/editor/order/{id}/is-completed/update', name: 'app_orders_is-completed-update')]
public function isCompletedUpdate($id, OrderRepository $orderRepository, EntityManagerInterface
{
    $order = $orderRepository->find($id);
    $order->setIsCompleted(true);
    $entityManager->flush();
    $this->addFlash('success', 'Modification effectuée');
    return $this->redirectToRoute('app_orders_show');
}

```

Pensez ensuite à ajouter le lien au bon endroit sur le bouton bien sûr,

```

<h5 class='mb-5'>Total à payer : {{ order.totalPrice }} €</h5>
<a href="{{ path('app_orders_is-completed-update',{'id':order.id}) }}" class

```

Ainsi que l'affichage du message flash qui est déjà existant dans product, on réutilise toujours.

```

<h1 class="mt-5 mb-5 text-center">Commandes</h1>
|   {{ include ('layouts/_flash_message.html.twig') }}
|   {% for order in orders %}

```

Et ensuite on actualise et on teste, super tout fonctionne parfaitement, il est possible que certain ai une erreur (ou pas), à vous de la résoudre eheh. J'ai bien le message flash qui s'affiche et le statut qui change que ce soit en vue ou en bdd. Ce que l'on va mettre en place désormais c'est que le bouton marqué comme livrée ne soit présent que si la commande n'a pas été livrée dans le cas où elle est livré le bouton n'a plus d'utilité donc il ne sera pas là.

Essayez de le faire seul encore une fois, c'est assez simple à mettre en place,

```

{% if order.isCompleted != true %}
|   <a href="{{ path('app_orders_is-completed-update',
|   {% endif %}

```

Voilà pas plus difficile que cela, bravo si vous avez réussi. On va avoir aussi le cas de la suppression de la commande on va donc créer une nouvelle route pour cela, à vous de jouer encore une fois, pensez à mettre un petit message flash 😊.

```

#[Route('/editor/order/{id}/remove', name: 'app_orders_remove')]
public function removeOrder(Order $order, EntityManagerInterface $entityManager):Response
{
    $entityManager->remove($order);
    $entityManager->flush();
    $this->addFlash('danger', 'Commande supprimée');
    return $this->redirectToRoute('app_orders_show');
}

```

Ensuite la partie vue, où je vais mettre une message de confirmation avant d'effacer la commande, sur un `onclick`.

```

<a onclick="return confirm ('Voulez-vous vraiment supprimer cette commande ?')"

```

Evidemment le `path` vient en suivant. N'oubliez pas de tout tester comme à chaque fois, tout fonctionne parfaitement bien, bravo pour votre travail.

On va penser à la génération de la facture ce qui me semble essentiel sur une application de ce type, donc on va devoir créer un Controller qui va gérer cela, je vais l'appeler `BillController` car je fais en anglais comme toujours. Vérifiez que vous avez bien tout les fichiers générés et vous vous rendez sur le `Controller` que l'on va donc modifié, on va partir de l'idée que seul l'éditeur ou l'admin peut gérer cela, à vous de jouer,

```
#Route('editor/order/{id}/bill', name: 'app_bill')
public function index($id, OrderRepository $orderRepository): Response
{
    $order = $orderRepository->find($id);

    return $this->render('bill/index.html.twig', [
        'order'=>$order,
    ]);
}
```

Rien de bien compliqué pour le moment, ok on va passer à la vue, vous connaissez les étapes de base, on efface tout ce qui ne sert à rien et on affiche ce que l'on a mis en place dans le `Controller`, afin encore une fois de ne pas récrire le code vous allez pouvoir récupérer tout ce qui se trouve dans le `for` du file `order` (`twig`) car c'est encore un tableau alors autant récupérer et ne pas avoir à réécrire tout cela, on aura juste à modifier légèrement. Vous allez en fait supprimer certaines choses, je vous montre la fin du fichier car il est trop long,

```
<span>Frais de livraisons : {{ order.city.shippingCost }} €</span>
<h5 class='mb-5'>Total à payer : {{ order.totalPrice }} €</h5>
</div>
{% endblock %}
```

Voici donc la fin du fichier, vous allez ensuite copier le lien de la route donc le `app_bill`, et vous allez simplement ajouter un lien supplémentaire dans le file `order.html.twig` afin d'avoir accès à cela.

```
<span>Frais de livraisons : {{ order.city.shippingCost }} €</span>
<h5 class='mb-5'>Total à payer : {{ order.totalPrice }} €</h5>
<a href="{{ path('app_bill',{'id':order.id} )}}>Imprimer la facture</a>
```

Vous allez actualiser et vérifier que cela s'affiche, j'ai ajouté une classe pour avoir un bouton vert comme les autres, ensuite vous tester ce bouton. Vous allez constater que tout cela fonctionne, on va devoir faire en sorte de pouvoir l'imprimer par la suite. Faites les petites modifications comme le titre car c'est une facture désormais et on va modifier certaines infos dont on a pas besoin, on a pas besoin de savoir si la commande a été livrée donc on peut effacer, et je vais mettre la date de la commande,

```
<h2>Facture n° {{ order.id }}</h2>
<span>Date de la commande : {{ order.createdAt|date }}</span>
<br>
```

Je vais laisser le descriptif du produit mais il n'est pas nécessaire sur une facture donc vous pouvez l'enlever si vous le désirez, en revanche je vais enlever l'image ce n'est pas du tout nécessaire à ce niveau. On va installer un plugin qui permettra la génération de pdf mais attention il ne prendra pas en compte le style Bootstrap, vous allez devoir coder le `Css` entièrement à la main en dur pour qu'il y

ai du style, je vais donc supprimer l'import de la base et je n'aurai plus de style sur l'impression. On va donc faire du style dans ce fichier (je précise que je n'aime pas cela du tout je préfère importer un fichier à part, mais c'est exceptionnel dans ce cas-là).

```
<style>
    table{
        width: 100%;
        border-collapse: collapse;
    }
    th, td{
        border: 1px solid #ddd;
        padding: 8px;
        text-align: left;
    }
    thead{
        background: #f2f2f2;
    }
    tbody tr:nth-child(odd){
        background: #f9f9f9
    }

</style>
```

Rien d'extraordinaire je vais vous montrer où se trouve le thead car j'ai un peu modifié, ok super, on va maintenant installer un bundle qui va gérer la génération du pdf, vous allez taper dans votre terminal : **composer req dompdf/dompdf**

Et vous patientez, quand c'est fait on va retourner sur le [Controller](#) et écrire le code qui va gérer le [pdf](#), je vous le donne et on va l'expliquer ensemble.

```

$order = $orderRepository->find($id);

$pdfOptions = new Options(); //definit la nouvelle i
$pdfOptions->set('defaultFont','Arial'); //Définit l
$domPdf = new Dompdf($pdfOptions); //On ajoute les op
$html = $this->renderView('bill/index.html.twig', [
    'order'=>$order,
]);
// On insere ce que l'on veut imprimer
$domPdf->loadHtml($html); // On charge le html dans
$domPdf->render(); //On crée le rendu
$domPdf->stream('bill-'.$order->getId().'.pdf',[//On
    'Attachment'=>false //ca permet de dire on va te
]);
}

return new Response('',200,[ //
    'Content-Type' => 'application/pdf'
]);

```

Voici le code et si vous cliquez désormais sur imprimer la facture après avoir actualiser bien sur vous aurez désormais un pdf qui s'affiche à l'écran et vous avez le choix de le télécharger ou bien de l'imprimer, donc on a bien résolu ce souci, bravo à nous. Evidemment vous pouvez télécharger ou imprimer autant de fois que vous le désirez.

Ok on va passer à l'envoi d'un mail de confirmation de la commande après avoir passé une commande pour les clients sa améliore et rassure l'utilisateur. Rendez-vous dans le [OrderController](#) afin de faire cela, avant la redirection vous allez écrire le code nécessaire,

```

// Mise à jour du contenu du panier en session
$session->set('cart', []);
// Redirection vers la page du panier
return $this->redirectToRoute('app_order_message');

```

Voilà pour le début du code,

```

$session->set('cart', []);

$html = $this->renderView('mail/orderConfirm.html.twig',[ //crée une vue mail
    'order'=>$order //on recupere le $order apres le flush donc on a toutes les infos
]);
// Redirection vers la page du panier
return $this->redirectToRoute('app_order_message');

```

Ensuite il faudra créer le fichier [twig](#) que l'on vient de noter qui lui n'existe pas et à l'intérieur de ce fichier vous pouvez déjà copy/paste le fichier [index.html.twig](#) qui se trouve dans le dossier [bill](#) ensuite je vais vous guider pas de panique. Vous n'aurez pas besoin de transmettre certaines

information telle que le numéro de facture par exemple ainsi que les information de client car c'est à lui que l'on va transmettre le mail donc aucun intérêt, je vais juste mettre un titre disant Confirmation de commande personnellement, et un petit message.

```
<div class="container">
    <!--Possible de mettre le logo de l'entreprise ici en html-->
    <h1>Confirmation de commande</h1>
    <p>Merci pour votre commande elle sera traité dans les plus brefs délais</p>
        <span class='text-primary'>Détail de la commande</span>
        <table class="table table-bordered">
```

Voilà ce que j'ai mis histoire personnaliser un peu cela, vous allez pouvoir retourner sur le [Controller](#),

```
$html = $this->renderView('mail/orderConfirm.html.twig',[ //crée une vue mail
    'order'=>$order //on récupère le $order après le flush donc on a toutes les infos
]);
$email = (new Email()) //On importe la classe depuis Symfony\Component\Mime\Email;
->from('sneakhub@gmail.com') //Adresse de l'expéditeur donc notre boutique ou vous même
->to('to@gmail.com') //Adresse du receveur
->subject('Confirmation de réception de commande') //Intitulé du mail
->html($html);
```

Voilà pour le début, ensuite vous allez devoir créer un fonction tout en haut du Controller pour gérer cela, je vous montre,

```
class OrderController extends AbstractController
{
    public function __construct(private MailerInterface $mailer){

    }

#[Route('/order', name: 'app_order')]
```

On importe la classe comme toujours et on avance sur le code, le mailer est crée on peut retourner sur le code plus bas,

```
->subject('Confirmation de re
->html($html);
$this->mailer->send($email);
```

Là on est prêt à envoyer les mails, le mieux est de tester, on va donc faire une commande et la valider et vous devriez avoir une erreur MAILER DSN environment variable, pourquoi ??

Car ce canal n'est pas configuré donc c'est assez logique, donc cela est déjà prévu dans votre fichier. env car c'est déjà prévu mais il est commenté il suffit de le décommenter,

```
###> symfony/mail ##
MAILER_DSN=null://null
###< symfony/mail ##
```

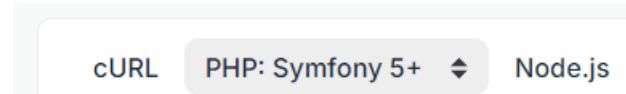
Vous avez actualisé et l'erreur a disparu mais attention ce n'est pas fini pour autant, il va falloir ajouter une valeur au [MailerDsn](#), pour le cours nous allons utiliser mail trap, il va stocker tous les mails sortant de l'application. Pour l'utiliser vous allez taper [mailtrap](#) sur votre navigateur, vous allez cliquer sur ceci,

The screenshot shows the Mailtrap homepage. At the top left is the Mailtrap logo (a stylized 'M'). Next to it is the text "Mailtrap" and the URL "https://mailtrap.io · Traduire cette page". To the right is a three-dot menu icon. Below this, the title "Mailtrap: Email Delivery Platform" is displayed in a large, bold, blue font. Underneath the title, a subtitle reads "Email Delivery Platform to test, send and control email infrastructure in one place. Great for dev teams. Start today - Sign Up free!".

Vous allez cliquer sur login et créer un compte (c'est gratuit pas de panique), une fois tout prêt vous allez choisir ceci [Testing email](#), cela va vous ouvrir votre box **MY INBOX**,

The screenshot shows the "Inboxes" section of the Mailtrap interface. On the left is a sidebar with navigation links: Home, Email API/SMTP, Email Testing (selected), Inboxes (highlighted with a blue arrow), Email Marketing (new), Sending Domains, Templates, and Billing. The main area has a search bar and several icons. A central image shows a computer monitor with an envelope and a mail truck. Below it is a "How it works" section with three steps: 1. Choose your technology, 2. Copy configuration, 3. Paste configuration to your project. To the right is the "My Inbox" dashboard with tabs for Integration, Email Address, Auto Forward, Manual Forward, and Access Rights. The "Integration" tab is active. It contains instructions to integrate with Mailtrap to send emails from an email client or mail transfer agent, with examples for SMTP, API, and POP3. The SMTP section shows credentials with Host set to "sandbox.smtp.mailtrap.io" and Port set to "25, 465, 587 or 2525".

Ça donne cela, ensuite vous allez cliquer sur inbox, et ensuite sur la roue crantée, cela va vous permettre de paramétriser votre box [mailtrap](#), ensuite vous allez descendre et cliquer sur PHP et sélectionner symfony 5+,



Cela va vous donner une ligne de code MAILER_DSN etc et vous la copier car c'est cela que l'on va utiliser, et vous allez remplacer la ligne MAILER du fichier. env par celle-ci, ici vous avez le mot de passe mais on ne le voit pas donc cela ne fonctionnera pas

e89a:*****b6c3@:

Il commence au : et fini sur @, il faudra retourner sur votre box sur la ligner mot de passe

Password *****b6c3

Et cliquer dessus puis le copier, ensuite vous viendrez le coller à la place de celui qui contient les étoiles dans le. env et vous enregistrez, une fois cela fait vous pourrez recevoir les mails de test dans votre box parfait c'est ce que l'on voulait.

Vous allez donc tenter cela en finalisant votre commande, attendez la redirection afin de vérifier que tout fonctionne, depuis la version 6 de Symfony on est obligé de démarrer en terminal le délivreur de message (un messenger), on va donc faire cela, dans un autre terminal vous allez taper **php bin/console messenger:consume async -vv**

Car pour le moment votre mail est sauvegarde en arrière-plan, il attend patiemment le Messenger, là vous pouvez aller vérifier sur mailtrap vous aurez le mail.

Confirmation de commande

Merci pour votre commande elle sera traité dans les plus brefs délais

Détail de la commande

Nom du produit	Description du produit	Prix du produit	Quantité	Prix total
----------------	------------------------	-----------------	----------	------------

Frais de livraisons : 3.6 €

Total à payer : 45 €

J'ai bien le mail mais il y a une petite erreur je n'ai pas toutes les infos, comme tout n'est pas nécessaire je vais simplement les enlever seul le montant de la commande est nécessaire vu qu'il y a déjà la facture cela ne sert à rien de répéter les choses, on pense à l'écologie. Vous allez après avoir effacé les choses inutiles recréer une commande et recommencer l'opération.

Confirmation de commande

Merci pour votre commande elle sera traité dans les plus brefs délais

Détail de la commande

Frais de livraisons : 3.6 €

Total à payer : 45 €

Parfait c'est beaucoup mieux, maintenant il faut penser que nous avons paramétré le receveur à to@gmail.com sauf que nous on veut que ce soit celui qui passe commande, sauf que sur la validation de commande on a oublié de mettre un champ email pour la récupérer et on va devoir modifier cela et oui. On va donc devoir ajouter ce champ-là, vous allez donc taper la ligne de commande qui sert à modifier l'`entity Order`, `symfony console make:entity Order`, et ajouter le champ email, qui sera de type string, la longueur vous pouvez laisser la proposition maximale, et il ne sera pas null, ensuite vous effectuez la migration. A la suite de cela il va bien évidemment falloir modifier le formulaire pour y ajouter le nouveau champ, donc on va sur `OrderType` et on ajoute cela, je l'ai placé sous le `lastname` car cela me semble être bien ici.

Ok super maintenant au niveau du `Controller` dans le mailer la ou il y a le `->to` on va modifier un peu,

```
->from('sneakhub@gmailcom')
//->to('to@gmailcom') //Adresse de l'expéditeur
->to($order->getEmail())
```

Voilà, on enregistre et on recrée une commande et on envoie la commande, vous aurez le mail et vous pourrez constater que cela aura bien récupérer le mail de l'utilisateur

Confirmation de réception de commande

From: <sneakhub@gmailcom>
To: <chabanaisjeremie@gmail.com>

Une fois en production ce sera bien envoyé au client et d'autre service seront utilisé vous n'utiliserez pas forcément [mailtrap](#), a vois avec les entreprise ou vos choix personnel, super tout fonctionne bien. On va passer à la suite. On va passer à la partie paiement car on a géré que la partie paiement à la livraison mais si la personne paie en ligne ce n'est actuellement pas géré donc on va faire cela, vous allez sur le [OrderController](#), et on va continuer cela, on va donc gérer le fait que le visiteur n'est pas coché la case paiement à la livraison, mais pour cela on va créer un nouveau service et oui, cela permettra de le réutiliser n'importe où sur votre appli, vous allez donc créer un new file que vous appellerez [StripePayment](#). Et ensuite vous créez une variable public que moi j'ai appelé [\\$redirectUrl](#) mais vous êtes libre tant que cela reste logique, ensuite vous ouvrez votre navigateur et vous vous rendez sur stripe,



Vous faites l'inscription classique, une fois cela fait vous allez atterrir sur cette page

The screenshot shows the 'Démarrer avec Stripe' (Start with Stripe) page. On the left, there's a sidebar with links: 'Accueil', 'Soldes', 'Transactions', 'Clients', and 'Catalogue de produits'. Below that is a 'Produits' section with 'Payments', 'Billing', 'Reporting', and 'Plus'. The main area has a large button 'Démarrer la configuration →'. To the right, there's text: 'Vous n'aurez besoin que de quelques minutes pour signer les documents. Nous enregistrons votre programme et vous pouvez reprendre la procédure à tout moment.' Below this is another button 'Démarrer avec Stripe'. At the bottom, there are two buttons: 'Peu de code' and 'Code', with the latter being highlighted.

Vérifiez que vous êtes bien en mode test en haut sur la droite, normalement vous y serez directement car vous n'aurez pas renseignez les données d'un entreprise.

On va ensuite regarder sur le boc Démarrez avec [Stripe](#) et vous verrez une clé publique et une clé secrète et c'est celle-ci dont on aura besoin, pour l'obtenir vous pouvez tout simplement cliquez en haut sur **Développeurs**,



Ensute clé API et vous cliquez sur révélez la clé, vous allez copier cette clé secrète, puis une fois cela fait vous allez l'enregistrez sur le `.env` généralement c'est ici que l'on met ce genre de chose.

```
MAILER_DSN=smtp://d  
###< symfony/mailerservice  
STRIPE_SECRET_KEY=s
```

Ensute, vous retournez sur votre service [Stripe](#), en vous allez créer un constructeur puis on va coder le nécessaire mais nous aurons besoin d'un bundle qui va nous faciliter tout ça,

```
public function __construct()  
{  
}  
}
```

Vous pouvez vous rendre dans la documentation de [Stripe](#) et il y a un onglet documentation de l'API, c'est ce qui va nous intéresser, vous cliquez sur le langage utilisé donc PHP et il vous donne la ligne à installer pour commencer,



Et vous lancez l'installation de la dépendance, une fois fini vous retournez dans le code et on va utiliser cette dépendance, pensez à importer le use de [Stripe](#), sinon vous aurez des erreurs dans votre code.

```
namespace App\ServiceService;  
  
use Stripe\Stripe;
```

Ensute on va donc récupérer cette clé secrète, mais je vais vous apprendre encore un petit tips, en PHP on peut récupérer une variable du `.env` en tapant `$_SERVER` puis en paramètre le nom de la variable en question.

Puis on ajoute la version de l'api que l'on retrouve dans la partie Développeurs de [Stripe](#), puis apercu ou overview et on l'a

API versions

2024-06-20 Default Latest

```
public function __construct()
{
    Stripe::setApiKey($_SERVER['STRIPE_SECRET_KEY']); //on gère la clé
    Stripe::setApiVersion('2024-06-20'); //on gère la version
}
```

OK on est bien pour le constructeur on va pouvoir passer à la fonction du paiement,

```
public function startPayment($cart){
}
```

C'est un bon début on va passer au [OrderController](#) et on va injecter le service que l'on vient de créer,

```
return $this->redirectToRoute('app_order_message');
}
// $order->setTotalPrice($data['total']);
// $order->setCreatedAt(new \DateTimeImmutable());
// dd($order);
// $entityManager->persist($order);
// $entityManager->flush();

$paymentStripe = new StripePayment(); //on importe notre service avec sa classe
$paymentStripe->startPayment($data); //on importe le panier donc $data
,
```

N'écrivez pas les lignes en commentaires je précise, et on retourne sur le service et on va vérifier cela,

```
public function startPayment($cart){
    dd($cart);
}
```

Et vous validez une commande en ne cochant pas la case paiement à la livraison, vous devriez avoir une erreur pourquoi ?? Lisez bien l'erreur !!

Et oui c'est un array et nous avons mis des parenthèses donc erreur eheh, bien vu, on modifie cela du coup

```

public function __construct()
{
    Stripe::setApiKey($_SERVER['STRIPE_SECRET_KEY']);
    Stripe::setApiVersion('2024-06-20'); //on gère la
}

```

Tout fonctionne on récupère bien le panier on va pouvoir créer la session dans stripe donc votre service,

```

public function startPayment($cart){
    //dd($cart);
    $session = Session::create([]);
}

```

Pensez à l'import de la classe de session qui sera celle-ci e toute logique,

```

use Stripe\Stripe;
use Stripe\Checkout\Session;

```

```

public function startPayment($cart){
    //dd($cart);
    $session = Session::create([ //création de la session Stripe
        'line_items'=>[ //produit qui vont etre payer
            ],
        'mode' => 'payment', //mode de paiement
        'cancel_url' => 'http://127.0.0.1:8000/pay/cancel', //url de cancellation
        'success_url' => 'http://127.0.0.1:8000/pay/success', /
        'billing_address_collection' => 'required', //si on autorise les adresses
        'shipping_address_collection' => [ //pays ou on souhaite envoyer
            'allowed_countries' => ['FR','EN'],
        ],
        'metadata' => [
            // 'order_id' => $cart->id, //id de la commande
        ]
    ]);
}

```

On a bien tout créé normalement 😊, ensuite on va créer après la session, la récupération de l'url, il est possible que les routes doivent être en localhost et non pas avec une adresse ip directement.

```

]);
$this->redirectUrl = $session->url;

```

On va juste modifier la **variable redirectUrl** en privé au lieu de public c'est mieux, et crée une fonction qui retourne **redirectUrl** qui elle sera en public,

```
    $this->redirectUrl = $session->url;  
  
}  
public function getStripeRedirectUrl(){  
    return $this->redirectUrl;  
}
```

Il reste à récupérer les produits à payer et les transmettre à **Stripe**, ils sont contenus dans **\$data** qui se trouve dans le **CartController**. On va rajouter au **OrderController** quelque détail pour Stripe,

```
$paymentStripe = new StripePayment(); //on importe notre service avec sa classe  
$paymentStripe->startPayment($data); //on importe le panier donc $data  
$stripeRedirectUrl = $paymentStripe->getStripeRedirectUrl();  
  
return $this->redirect($stripeRedirectUrl);
```

Là on est bon de ce côté pour les urls et les redirections, alors finissons le file **StripePayment** afin de pouvoir être redirigé dessus pour le paiement de nos commandes.

```
public function startPayment($cart){  
    //dd($cart);  
    // Récupération des produits du panier  
    $cartProducts = $cart['cart'];  
    // Initialisation d'un tableau vide pour stocker les produits formatés  
    $products = [];  
  
    // Boucle pour parcourir chaque produit du panier  
    foreach ($cartProducts as $value) {  
        // Initialisation d'un tableau vide pour stocker les informations d'un produit  
        $productItem = [];  
        // Récupération du nom du produit  
        $productItem['name'] = $value['product']->getName();  
        // Récupération du prix du produit  
        $productItem['price'] = $value['product']->getPrice();  
        // Récupération de la quantité du produit  
        $productItem['qte'] = $value['quantity'];  
        // Ajout du produit formaté au tableau des produits  
        $products[] = $productItem;  
    }  
  
    $session = Session::create(); //création de la session Stripe
```

Ensuite on va finaliser la **Session::create()**,

```

$session = Session::create([
    //création de la session Stripe
    'line_items' => [
        //produits qui vont être payés
        array_map(fn(array $product) => [
            'quantity' => $product['qte'],
            'price_data' => [
                'currency' => 'EUR',
                'product_data' => [
                    'name' => $product['name']
                ],
                'unit_amount' => $product['price']*100, //prix donnée en centimes donc on multiplie
            ],
        ], $products)
    ],
    'mode' => 'payment', //mode de paiement
]

```

Voilà là on est bien, vous pouvez actualiser le panier et valider la commande, vous serez redirigé vers [Stripe](#) afin d'effectuer le paiement de la commande.

Maintenant vous devriez avoir remarqué quelque chose qui devrait vous chiffrer sur [Stripe](#), il n'y a pas les frais de port qui s'ajoute et oui ce n'est jamais fini 😊. La méthode la plus simple pour ajouter les frais de livraisons sera d'ajouter les frais de livraisons comme frais supplémentaires car [Stripe](#) ne gère pas cela. Et comme on a déjà les frais de livraisons à partir du panier on va les ajouter en paramètres de la fonction,

```

public function startPayment($cart, $shippingCost){

```

Vous allez vous rendre dans le [OrderController](#) et on va récupérer cela grâce à la city,

```

$paymentStripe = new StripePayment(); //on importe notre service
$shippingCost = $order->getCity()->getShippingCost();
$paymentStripe->startPayment($data, $shippingCost); //on importe notre service
$stripeRedirectUrl = $paymentStripe->getStripeRedirectUrl();

```

Maintenant dans le [StripePayment](#) il va falloir injecter cela comme étant un produit, on a un tableau vide qui prend les produits et bien on va lui injecter les frais de livraisons comme produit et les autres produits viendront s'ajouter à cela,

```

$products = [
    [
        'qte' => 1,
        'price' => $shippingCost,
        'name' => "Frais de livraison"
    ]
];

```

On actualise le panier et on teste, cela fonctionne parfaitement cela ajoute bien les frais de livraisons j'ai essayé avec plusieurs villes et tout est bon.

On va passer au informations supplémentaires, les [metadata](#) que nous avions laissé libre, mais on va tester un paiement pour voir un peu. Vous allez rentrer ce que vous voulez, et pour les données de paiement vous allez noter le numéro qui est fourni par [Stripe](#) et c'est le 4242 4242 4242 4242, (j'invente rien c'est sur Stripe) puis la date de votre choix et le CVC de votre choix également.

Evidemment vous aurez une erreur car les page pay/success et pay/cancel n'existe pas du moins on ne les a pas créées encore, mais tout fonctionne le paiement passe bien. Vous pouvez allez vérifier sur votre box **Stripe** il apparaitra,

Volume brut	Hier
48,60 €	0,00 €
15:46	

Voilà pour moi tout a bien fonctionné, si vous descendez vous avez tout les détails de la commande, des infos clients aux infos de la commande, donc superbe tout fonctionne à merveille. Maintenant on va créer les routes de redirection après paiement.

Exercice 18

Correction

Donc on lance la commande `symfony console make:controller StripeController`, puis on copie cole la route et on met les infos que l'on a déjà créer plus tôt dans le fil `StripePayment`.

```
#[Route('/pay/success', name: 'app_stripe_success')]
public function success(): Response
{
    return $this->render('stripe/index.html.twig', [
        'controller_name' => 'StripeController',
    ]);
}

#[Route('/pay/cancel', name: 'app_stripe_cancel')]
public function cancel(): Response
{
    return $this->render('stripe/index.html.twig', [
        'controller_name' => 'StripeController',
    ]);
}
```

Ensuite bien sur il va falloir créer ces deux templates là. Bravo a vous c'était pas bien compliqué pour vous je sais mais bravo a tous.

Et là après paiement vos utilisateurs seront redirigés et il n'y aura plus d'erreur. Ce qui serai pas mal ce serai aussi de pouvoir envoyer une notification a nos utilisateurs, dans la partie développeur de **Stripe** il y a un onglet webhooks, on va cliquer sur créer un endpoint (point de terminaison), ensuite cliquez sur sélectionner tous les évènements, et la vous arriverai sur cette page,



Déclenchez automatiquement des workflows dans vos applications en réponse aux événements envoyés par Stripe

Envoyez des événements Stripe vers différentes destinations, comme des endpoints de webhook ou des écouteurs locaux.

[En savoir plus sur les webhooks](#)

[+ Créer un endpoint](#)

ou [créer un écouteur local](#)

Ce seraient bien de recevoir cela sur l'application et de pouvoir mettre à jour les infos en bdd quand même, car on pouvait tout gérer sur la partie admin, mais bon. Car pour le moment ce n'est pas relié avec la bdd, il va donc falloir créer un endpoint,

Pour finir, configurez votre endpoint

Indiquez à Stripe où envoyer les événements et rédigez une description pertinente pour votre endpoint.

URL d'endpoint

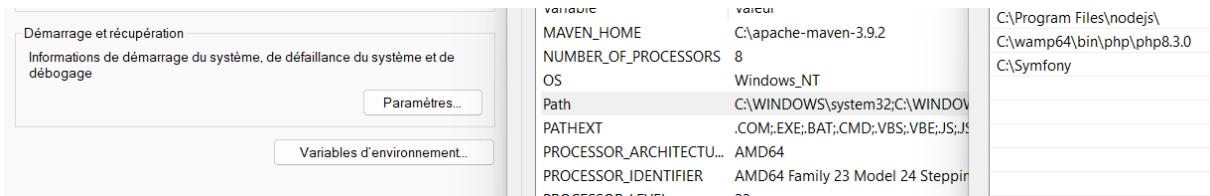
Une URL d'endpoint en mode production est nécessaire pour recevoir les événements envoyés par les webhooks.

Description Facultatif

Une description facultative de cet endpoint...

Pour créer donc ce endpoint il va falloir ajouter l'url, ceci se fait en production là nous sommes en mode dev donc cela n'est pas possible mais je voulais vous montrer cela malgré tout. On va quand même gérer cela mais d'une autre manière du coup, donc vous allez cliquer sur créer un écouteur local, et cela va vous afficher une fenêtre avec la marche à suivre pas de panique on le fait ensemble. Vous allez cliquer sur **Téléchargez l'interface de ligne de commande de Stripe**, vous allez choisir votre système d'exploitation pour ma part ce sera **window**, puis on va suivre les étapes donc on clique sur le lien **Github**, et ont choisi sa version moi c'est **window 64 bits**, une fois téléchargé on ouvre et si besoin on décomprime. Pour avoir accès à cela dans tout notre application on va devoir le déplacer dans le répertoire C. Une fois cela fait vous allez simplement cliquez dessus une fois afin de faire apparaître l'adresse du fichier, et vous copier cette adresse donc j'imagine C:/ et le nom du

fichier, et on va devoir l'ajouter au path du système, (variable d'environnement). Vous allez donc vous rendre dans les variables d'environnement, vous allez cliquer sur variable d'environnement en bas, et cliquer sur path,



Voici un aperçu des 3 fenêtres ouverte, vous allez cliquer sur nouveau dans la dernière fenêtre ouverte,



Vous collez votre lien vers le dossier en c, puis ok sur chaque fenêtre jusqu'à fermeture de toutes les fenêtres. Voilà vous venez d'ajouter une variable d'environnement à votre système, pour vérifier que cela fonctionne bien vous allez ouvrir un invite de commande et taper tout simplement **stripe** et vous verrez stripe s'ouvrir dans l'invite de commande donc les manips sont bonne et vous avez bien bosser, bravo. On va pouvoir continuer avec notre doc, on a donc téléchargé notre fichier et tout fonctionne, on va donc d'abord créer une route qui va gérer cette notification-là, on va faire cela donc le [StripeController](#), je vais l'appeler simplement [StripeNotify](#).

```
#[Route('/stripe/notify', name: 'app_stripe_notify')]  
public function stripeNotify(Request $request): Response  
{  
}  
}
```

Pour le moment c'est tout simple, on ajoutera le reste ensuite, on va donc pouvoir passer à l'étape suivante.

1 Téléchargez l'interface

```
$ stripe login
```

Ce qui va nous permettre de créer une session local, donc sur l'invite de commande toujours, on tape stripe login, cela va vous donner un lien que vous allez copier dans un navigateur et vous faites entrée, cela va vous demander d'autoriser l'accès, vous dites oui et vous aurez cela ensuite.

stripe



Vous aurez un message avec un Done,

```
C:\Users\Jérémie>stripe login
Your pairing code is: brisk-luxury-proud-quiet
This pairing code verifies your authentication with Stripe.
Press Enter to open the browser or visit https://dashboard.stripe.com/stripecli/confirm_auth?8SYUCu1 (^C to quit)
> Done! The Stripe CLI is configured for your account with account id acct_1Q0grS04Ha88nC6n
```

Ensute on passe à l'étape suivante, vous allez copier cela dans l'invite de commande,

② Transférer les événements vers vi

```
$ stripe listen --forward-to
```

```
C:\Users\Jérémie>stripe listen --forward-to 127.0.0.1:8000/stripe/notify
> Ready! You are using Stripe API Version [2024-06-20]. Your webhook signing
ec325225147e39308429459f540637fc2205b (^C to quit)
```

Avec l'adresse de notre port puis l'adresse que l'on a donner dans le [Controller](#), et on tape sur entrée, cela à générer une clé et il faudra la copier ça commence par whsec, et vous la stocker quelque part, et on va coder dans le Contrôler la suite du code, je vais vous l'expliquer bien évidemment,

```
/*
 * Route pour la notification de Stripe (webhook)
 */
@Route("/stripe/notify", name="app_stripe_notify")
*/
public function stripeNotify(Request $request): Response
{
    // Définir la clé secrète de Stripe à partir de la variable d'environnement
    Stripe::setApiKey($_SERVER['STRIPE_SECRET_KEY']);

    // Définir la clé de webhook de Stripe
    $endpoint_secret = 'whsec_762838b6a470d9e24ef2fd08b52ec325225147e39308429459f540637fc2205b';
    // Récupérer le contenu de la requête
    $payload = $request->getContent();
    // Récupérer l'en-tête de signature de la requête
    $sigHeader = $request->headers->get('Stripe-Signature');
    // Initialiser l'événement à null
    $event = null;
```

Voici la suite,

```

$event = null;

try {
    // Construire l'événement à partir de la requête et de la signature
    $event = \Stripe\Webhook::constructEvent(
        $payload, $sigHeader, $endpoint_secret
    );
} catch (\UnexpectedValueException $e) {
    // Retourner une erreur 400 si le payload est invalide
    return new Response('Invalid payload', 400);
} catch (\Stripe\Exception\SignatureVerificationException $e) {
    // Retourner une erreur 400 si la signature est invalide
    return new Response('Invalid signature', 400);
}

```

Encore un peu,

```

    return new Response('Invalid signature', 400);
}

// Gérer les différents types d'événements
switch ($event->type) {
    case 'payment_intent.succeeded': // Événement de paiement réussi
        // Récupérer l'objet payment_intent
        $paymentIntent = $event->data->object;

        // Enregistrer les détails du paiement dans un fichier
        $fileName = 'stripe-detail-' . uniqid() . '.txt';
        file_put_contents($fileName, $paymentIntent);
        break;
    case 'payment_method.attached': // Événement de méthode de paiement attachée
        // Récupérer l'objet payment_method
        $paymentMethod = $event->data->object;
        break;
    default :
        // Ne rien faire pour les autres types d'événements
        break;
}

```

Et le return,

```

        break;
}

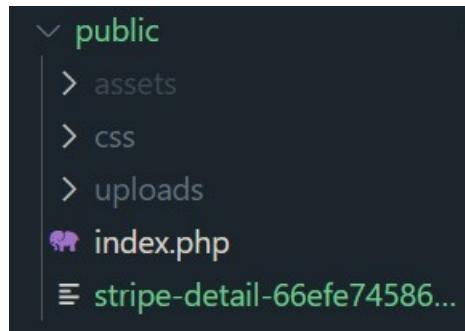
// Retourner une réponse 200 pour indiquer que l'événement a été reçu avec succès
return new Response('Événement reçu avec succès', 200);
}

```

Voilà pour la fonction `stripeNotify`, vous allez pouvoir enregistrer et on va tester cela, vous retourner sur l'appli et vous allez initier un nouveau paiement. Dans votre invite de commande vous devez voir la requête POST avec un statut 200 ce qui veut dire que tout s'est très bien déroulé,

```
--> charge.updated [evt_3Q1mMJ04Ha88nC6n09a44DWx]
<-- [200] POST http://localhost:8000/stripe/notify [evt_3Q1mMJ04Ha88nC6n09a44DWx]
--> checkout.session.completed [evt_1Q1mMP04Ha88nC6nNjfyQDAi]
<-- [200] POST http://localhost:8000/stripe/notify [evt_1Q1mMP04Ha88nC6nNjfyQDAi]
```

Et dans votre dossier public, vous pourrez donc constater que vous aurez un fichier qui contient donc tout les renseignement sur la requête de paiement donc tout fonctionne à merveille.



Là on va pouvoir faire une mise a jour de la base de données, là cela va dans un fichier test mais ce que l'on veut c'est que cela aille en base de données pour mettre à jour les informations.

Donc jusqu'à présent on récupérer le contenu dans un fichier qui se trouve dans le dossier public mais nous ce que l'on veut désormais c'est que cela aille dans la bdd, donc on modifier un peu notre code, dans la table order nous avons un champ `payOnDelivery` qui dit si c'est un paiement à la livraison ou pas, on va devoir faire en sorte que cela note false pour un paiement en ligne, et également le champ `isCompleted` qui permet de savoir si la commande a été réglé ou pas donc on va faire en sorte de mettre à jour ces champs-là. Sur l'appli vous avez la page `editor/order` qui vous donner accès aux commandes ou vous pouvez gérer les mise à jour des commandes, livraisons, factures etc.

On va donc devoir ajouter un champ à `l'entity Order` et ajouter un champ, donc on se rend sur la table en terminal, `symfony console make:entity Order`, et vous allez ajouter le champ `isPaymentCompleted` (rappelez-vous quand le champ commence par `is` il est automatiquement en boolean) donc ensuite vous aurez la question si cela peut-être null, on dit non et c'est bon. Ensuite on effectue la migration complète,

is_payment_completed

0

0

Evidemment on vérifie comme toujours, ensuite dans le `Controller`, (`Order`) ont stocké la commande dans le if, et bien avant de procéder au paiement on va stocké la commande en local, on va donc faire quelques petites mise à jour dans le code,

```

{
    // Récupère les données du panier à partir de la session using le se
    $data = $cart->getCart($session);
    // Crée un nouvel objet Order
    $order = new Order();
    // Crée un formulaire pour gérer la création de la commande using le
    $form = $this->createForm(OrderType::class, $order);
    // Gère la soumission du formulaire
    $form->handleRequest($request);

    //quand c'est true
    if ($form->isSubmitted() && $form->isValid()) {
        // Vérifie si le total du panier n'est pas vide
        if(!empty($data['total'])) {
            // Définit le prix total de la commande
            $order->setTotalPrice($data['total']);
            // Définit la date de création de la commande
            $order->setCreatedAt(new \DateTimeImmutable());
            $order->setPaymentCompleted(0); //on initialise a false
        }
    }
}

```

Voici la suite,

```

    $order->setPaymentCompleted(0); //on initialise a false
    //dd($order);
    $entityManager->persist($order);
    $entityManager->flush();
    // Boucle sur chaque élément du panier
    foreach($data['cart'] as $value) {
        // Crée un nouvel objet OrderProducts
        $orderProduct = new OrderProducts();
        // Définit la commande pour le produit de la commande
        $orderProduct->setOrder($order);
        // Définit le produit pour le produit de la commande
        $orderProduct->setProduct($value['product']);
        // Définit la quantité pour le produit de la commande
        $orderProduct->setQuantity($value['quantity']);
        // Enregistre le produit de la commande dans la base de données
        $entityManager->persist($orderProduct);
        $entityManager->flush();
    }

    if($order->isPayOnDelivery()) {
}

```

La suite encore,

```

if($order->isPayOnDelivery()) {
    // Mise à jour du contenu du panier en session
    $session->set('cart', []);

    $html = $this->renderView('mail/orderConfirm.html.twig', [ //crée une vue
        'order'=>$order //on récupère le $order après le flush donc on a tout
    ]);

    $email = (new Email()) //On importe la classe depuis Symfony\Component\Mailer\Email
        ->from('sneakhub@gmail.com') //Adresse de l'expéditeur donc notre boutique
        ->to('to@gmail.com') //Adresse du receveur
        ->to($order->getEmail())
        ->subject('Confirmation de réception de commande') //Intitulé du mail
        ->html($html);
    $this->mailer->send($email);

    // Redirection vers la page du panier
    return $this->redirectToRoute('app_order_message');
}

// quand c'est false
$paymentStripe = new StripePayment(); //on importe notre service avec sa configuration
$shippingCost = $order->getCity()->getShippingCost();

```

Et la fin,

```

$shippingCost = $order->getCity()->getShippingCost();
$paymentStripe->startPayment($data, $shippingCost); //on importe notre service avec sa configuration
$stripeRedirectUrl = $paymentStripe->getStripeRedirectUrl();
//dd( $stripeRedirectUrl);

return $this->redirect($stripeRedirectUrl);
}

return $this->render('order/index.html.twig', [
    'form'=>$form->createView(),
    'total'=>$data['total'],
]);

```

Voilà pour les gros changements, il va falloir tester cela donc on ajoute des produits au panier, et on effectue un paiement, on va juste commenter la redirection **Stripe**,

```

$stripeRedirectUrl = $paymentStripe->getStripeRedirectUrl();
//dd( $stripeRedirectUrl);

//return $this->redirect($stripeRedirectUrl);

```

On n'en a pas besoin pour le test, évidemment après le form cela va rester sur le form de validation de commande il faudra aller voir en bdd si cela a bien fonctionné.



OK le payOnDelivery est bon, il reste à gérer le paiement, on va décommenter la redirection maintenant, précédemment on avait mis en place le service StripePayment, pendant le paiement à Stripe on va faire en sorte d'envoyer l'identifiant de la commande et après le paiement avec succès au niveau du webhooks on simplement récupérer cet identifiant-là, puis on pourra se servir de cela pour effectuer la mise à jour de la bdd. Et passer la commande en payé.

Donc dans le `StripePayment`, on va s'occuper des metadata et ajouter ce dont on a besoin, on va ajouter un paramètre à la fonction `startPayment`,

```
public function startPayment($cart, $shippingCost, $orderId){  
    //dd($cart);
```

Et donc les metadata,

```
],  
'payment_intent_data' => [  
    'metadata' => [  
        'order_id' //id de la commande  
    ]  
]
```

Et donc sur le `OrderController` au niveau du `startPayment` il faudra ajouter ceci,

```
$paymentStripe->startPayment($data, $shippingCost, $order->getId());
```

De manière à récupérer l'identifiant, finissons les metadata, de façon à setter l'order

```
'payment_intent_data' => [
    'metadata' => [
        'order_id' => $orderId // id de la commande
    ]
]
```

Allons donc sur le [StripeController](#), là où s'enregistrer le fichier et on va modifier ça,

```
$fileName = 'stripe-detail-' . uniqid() . '.txt';
$orderId = $paymentIntent->metadata->order_id;
file_put_contents($fileName, $orderId);
```

Au lieu de stocker paymentIntent, on va pouvoir stocker orderId, on va faire un petit test, il va falloir se reconnecter à [Stripe](#) depuis l'invite de commande, avec stripe login etc..

Et on relance un paiement, on vérifie la requête dans l'invite de commande et la redirection sur l'appli, tout fonctionne parfaitement, vérifier en public si le fichier est là, ok je récupère bien l'id de la commande dans le fichier [Stripe](#) dans le public,

```
public > ≡ stripe-detail-66f174abd64a6txt
      1      32
```

Et c'est bien l'id de la commande que j'ai en bdd, j'ai bien la commande avec l'id 32, donc désormais ce qu'il nous reste à faire c'est de mettre le statut de la commande en **valider**. On va donc retourner sur le StripeController et ajouter en paramètre le [OrderRepository](#) car c'est lui qui contient les infos des commandes,

```
#[Route('/stripe/notify', name: 'app_stripe_notify')]
public function stripeNotify(Request $request, OrderRepository $orderRepository): Response
```

OK ca c'est fait, et on va faire un [find\(\)](#), et faire un find() sur orderId qui contient ce qui nous intéresse, et on va setter,

```
    ...
    $fileName = 'stripe-detail-' . uniqid() . '.txt';
    $orderId = $paymentIntent->metadata->order_id;
    $order = $orderRepository->find($orderId);
    $order->setIsPaymentCompleted(1);
    // file_put_contents($fileName, $orderId);
```

On va également avoir besoin de EntityManagerInterface évidemment donc on l'ajoute en paramètre aussi,

```
#[Route('/stripe/notify', name: 'app_stripe_notify')]
public function stripeNotify(Request $request,
                            OrderRepository $orderRepository,
                            EntityManagerInterface $entityManager): Response
```

Et on effectue la mise à jour avec l'entityManager et le flush,

```
$orderId = $paymentIntent->metadata->orderId;
$order = $orderRepository->find($orderId);
$order->setIsPaymentCompleted(1);
$entityManager->flush();
```

Et ensuite et bien on reteste avec une commande, on valide le paiement on vérifie bien dans l'invite de commande que tout se déroule bien et on vérifie en bdd que le statut passe en valider, Ok parfait le statut IsPaymentCompleted passe bien en 1. Maintenant on pourra faire des vérifications, sur le prix par exemple, que stripe renvoie bien le bon prix,

```
$orderId = $paymentIntent->metadata->orderId;
$order = $orderRepository->find($orderId);

$cartPrice = $order->getTotalPrice();
$stripeTotalAmount = $paymentIntent->amount/100;
if($cartPrice==$stripeTotalAmount){
    $order->setIsPaymentCompleted(1);
    $entityManager->flush();
}
```

Et là on va retester un paiement afin de vérifier encore et toujours, et bien là on a une erreur la commande n'est pas mise à jour sur le statut valider il y a donc un petit souci, lequel ?

Après une petite recherche je pense avoir trouvé, voici ce que je viens de constater,

<input type="checkbox"/>	Montant
<input type="checkbox"/>	657,60 € EUR Réussi ✓
0	654 NULL chabanaisjeremie@gmail.com

Voilà le petit écart donc forcément le prix n'est pas le même, sur stripe et sur la bdd, et bien en bdd je n'ai pas les frais de port qui sont pris en compte, donc erreur, on va devoir modifier un peu du coup.

Rappelez-vous sur le OrderController on avait dit

```
if(!empty($data['total'])) {
    // Définit le prix total de la commande
    $order->setTotalPrice($data['total']);
```

Mais là on le total sans les frais de port donc ce sera plutôt \$data['total']+ frais de port je pense,

```
// vérifie si le total du panier n'est pas vide
if(!empty($data['total'])) {
    $totalPrice = $data['total'] + $order->getCity()->getShippingCost();
    // Définit le prix total de la commande
    $order->setTotalPrice($totalPrice);
}
```

Voilà qui devrait être beaucoup mieux déjà, encore un fois il va falloir tester avec un nouveau paiement. Ce qui prouve que lorsqu'on gère le paiement il faut penser à bien tout tester, ok tout fonctionne à merveille désormais le montant est le bon.

Ce qui sera ensuite c'est de vider le panier aussi car là il reste présent alors qu'au paiement à la livraison il se vide grâce à la session, mais au paiement en ligne non. Comment résoudre cela ? Et je vous le dit on va jouer sur la logique.

Supposons que votre client a commandé des produits au moment de la validation il a cliqué sur payer en ligne, ce qui se passe sur l'appli c'est que **Stripe** va générer les identifiants qui lui seront utiles pour le paiement, donc avant la redirection

```
$stripeRedirectUrl = $paymentStripe->getStripeRedirectUrl();
```

On peut supprimer le panier et là cela devrait fonctionner, comme le `startPayment` contenait déjà les informations de paiement de la commande, on peut le supprimer sauf que si l'utilisateur a oublié un produit et qu'il revient sur le site et bien son panier n'existerait plus. Donc ce que l'on devrait faire ici c'est plutôt supprimer le panier dans la page de succès comme cela on parle à toutes les options. Il sera donc supprimer une fois payé et pas avant 😊.

Donc on se rend sur le [Controller](#) qui gère cela, (`StripeController`), et on met cela en place,

```
#[Route('/pay/success', name: 'app_stripe_success')]
public function success(SessionInterface $session): Response
{
    $session->set('cart', []);
    // Rendre la vue "index.html.twig" avec le nom du contrôleur
    return $this->render('stripe/index.html.twig', [
        'controller_name' => 'StripeController',
    ]);
}
```

Ensuite on refait un paiement et on teste eheh, et après la redirection vous allez vous rendre sur la page du cart et normalement votre panier est bien vide. Je confirme il est vide, cela a bien fonctionné, bravo à nous tous.

On va mettre en place désormais un filtre sur les commandes pour les différencier, commande payé, livré, paiement en ligne ou pas, etc.. Vous avez déjà fait un super travail jusqu'à présent je tiens à vous féliciter, alors bravo à tous et toutes.

Donc on va donc chercher à filtrer les commandes, elles sont toutes dans la table `order` et la table `orderProduct` contient tous les produits qui se trouvent dans les commandes, tout les champs qui nous intéressent pour faire cela se trouvent dans la table `order`, `payOnDelivery`, `isCompleted` et `isPaymentCompleted`. La première chose à faire il va falloir ajouter des liens dans le menu, on va

commencer par gérer les commandes livrées, donc vous allez vous rendre sur le [Controller OrderController](#), sur la route `/editor/order` et on va ajouter un filtre dynamique pour gérer cela,

```
#Route('/editor/order/{type}', name: 'app_orders_show')
public function getAllOrder($type, OrderRepository $orderRe
```

Ça va nous éviter de copier-coller le code et avoir un code trop long, en faisant cela on va se servir de cette [wildcard](#) pour filtrer les commandes par type.

```
#Route('/editor/order/{type}', name: 'app_orders_show')
public function getAllOrder($type, OrderRepository $orderRepository, Request $request)
{
    if($type == 'is-completed'){
        $data = $orderRepository->findBy(['isCompleted'=>1], ['id'=>'DESC']);
    }

    //dd($orders);

    $orders = $paginator->paginate(
```

Si vous vous rendez sur votre appli avec la route `/editor/order` vous aurez une erreur 404 pourquoi ??

Et bien car on a modifier la route en ajoutant une [wildcard](#), donc ajoutez `/is-completed` à la fin et actualiser vous verrez les commandes livrées apparaitre, super parfait cela commence très bien, on va donc ajouter cela au menu comme cela fonctionne.

Dans la navbar vous allez copier le li et le premier a

```
{% if is_granted("ROLE_ADMIN") %}
    <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle"
            Admin
        </a>
```

Vous allez le copier en dessous du endif, avec se connecter ou s'inscrire,

```
    ...
    {% endif %}
    <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle"
            Commandes
        </a>
    </li>
</ul>
<form class="d-flex" role="search">
```

Et on va ajouter un ul déjà,

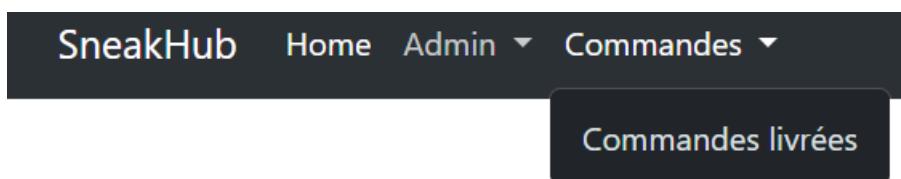
```

{%- endif %}

<!--Menu des commandes-->
- Commandes
- Commandes livrées

```

Et là vous essayez de créer vous-mêmes la route et le path, ensuite vous retournez sur votre appli et vous actualisez, cela devrait apparaître en menu.



Comme ce n'est que pour l'éditeur ou l'admin on doit mettre la condition if is granted, allez-y,

```

<!--Menu des commandes-->
{% if is_granted("ROLE_EDITOR") %}
- Commandes
- Commandes livrées

{% endif %}

```

Comme toujours on va actualiser pour vérifier, maintenant on va récupérer un deuxième cas pour les commandes payé mais non livrées, toujours dans le même [Controller](#) et même route,

```

} else if($type == 'pay-on-stripe-not-delivered'){
    $data = $orderRepository->findBy(['isCompleted'=>null, 'pay_on_delivery'=>0, 'is_payment_completed'=>1],
}

```

Evidemment il y aura le second tableau ensuite, ensuite on crée le nouveau lien et on teste, à vous de créer le lien encore une fois. Vous devriez avoir des erreurs et ce sera à vous de la résoudre 😊. Et voilà tout fonctionne bien, là vous devez vous rendre compte que sur une même vue on aura du contenu différent, on va gérer un autre cas les commandes payé en ligne et déjà livrées. A vous d'essayer seul,

```

} else if($type == 'pay-on-stripe-is-delivered'){
    $data = $orderRepository->findBy(['isCompleted'=>1, 'payOnDelivery'=>0, 'isPaymentCompleted'=>1], ['id'=>1]);
}

```

Et donc le lien du menu,

```
<li><a class="dropdown-item" href="{{ path('app_orders_show',{'type':'pay-on-stripe-is-delivered'})}}</a></li>
```

Et voilà tout fonctionne bien, on pourra ajouter autant de lien que l'on veut pour tout les cas de figure possible vous avez compris l'idée si vous désirez ajouter des liens allez y seul. On va avoir un erreur qui est que après la redirection si vous passez une commande comme livrée on est redirigé vers `app_order_show` et là on a une erreur car il ne reconnaît pas le typage, la `wildcard`, on va donc résoudre cela de suite, il suffit de passer le type à la redirection et en paramètres également sinon il n'y a pas accès. Mais on va faire cela de manière dynamique, on va passer la `request` en paramètre, et modifier la redirection,

```

#[Route('/editor/order/{id}/is-completed/update', name: 'app_orders_is-completed_update')]
public function isCompletedUpdate(Request $request, $id, OrderRepository

```

Et la partie redirect modifiée,

```

    $entityManager->flush(),
    $this->addFlash('success', 'Modification effectuée');
    return $this->redirect($request->headers->get('referer'));
}

```

Cela vous amène sur la page précédent cette route ci, donc c'est parfait. Il vous faudra un lien pour les commandes non livrées, car vous n'y avez plus accès à l'heure actuelle. Moi je l'ai fait personnellement, c'est quand même important. Maintenant qu'on arrive à la fin on va s'occuper du moteur de recherche de notre appli, rappelez vous que lorsque l'on cherche une info sur **Symfony** on a besoin du repository et évidemment de l'entity, c'est le système du MVC ne l'oubliez jamais c'est la base de **Symfony**.

Prenez le `ProductRepository`, et voyons certaines choses ensemble, vous avez utilisé jusqu'à présent plusieurs méthodes, elles sont intégrées par défaut à Symfony, on peut dire que c'est l'équivalent de requête que l'on fait à la bdd

- `find()`, permet de rechercher par un certains nombres de critères (`SELECT * FROM id`)
- `findBy()`, permet de récupérer par élément dans deux tableaux et possible un paramètre supplémentaire (`SELECT * FROM blabla WHERE blabla`)
- `findAll()`, permet de récupérer tous les éléments dans un tableau (`SELECT * FROM`)
- `findOneBy()`, permet de récupérer dans un tableau par un paramètre, et possible un autre paramètre

Imaginons que vous vouliez faire une requête du style, sélectionne-les-moi id qui sont supérieur à 10 et inférieur à 20. Et bien là vous n'aurez pas la possibilité d'utiliser les méthodes qui sont intégré à Symfony, des requêtes qui utiliserai le sélecteur `LIKE` par exemple, donc le moteur de recherche par exemple car c'est en utilisant des requêtes avec le sélecteur `LIKE`, (`SELECT * FROM user LIKE blabla`). Si vous avez donc une sélection qui diffère de ce que l'on a vu et bien il vous faudra créer vous-mêmes, et **Symfony** permet évidemment cela, vous voyez quelques exemples sous le constructeur du `ProductRepository`, ce qui en commentaire.

On va décommenter la première méthode et on va voir cela ensemble, vous allez l'appeler `findByIdUp()`, ou autre mais rappelez vous en du coup.

```
    public function findByIdUp($value): array
{
    return $this->createQueryBuilder('p') //retourner la requete
        ->andWhere('p.id > :val') // ajoute des critères val = $value
        ->setParameter('val', $value) //on set les parametres
        ->orderBy('p.id', 'ASC') //on definit les criteres
        ->setMaxResults(10) //definit le nbr de resultat
        ->getQuery() //
        ->getResult() //
}
```

Voilà on a crée une méthode, si on se rend désormais sur le [HomeController](#), et prenez sur la page d'accueil, on va donc injecter notre méthode, et la tester.

```
#[Route('/', name: 'app_home', methods: ['GET'])]
public function index(ProductRepository $productRepository, CategoryRepo
{
    $p = $productRepository->findByIdUp(10); //rappel de la méthode crée
    dd($p);
```

Et vous actualiser votre page, vous verrez le résultat.

```
HomeController.php on line 22:
array:2 [▼
  0 => App\Enti...\Product {#837 ▶}
  1 => App\Enti...\Product {#902 ▶}
]
```

On aura bien les produits dont l'id est supérieur à 10, donc cela fonctionne très bien. Vous pouvez donc créer des méthodes qui vont pouvoir utiliser beaucoup de chose qui ne sont pas intégré à [Symfony](#) comme des jointures etc.. Vous pouvez re commenter la méthode que l'on avait créé, c'était à titre d'exemple afin de parfaire votre culture dev.

On va devoir créer notre repo pour gérer le moteur de recherche, vous pouvez le faire dans le repo [ProductRepository](#),

```

    public function searchEngine(string $query) {
        // Crée un objet de requête qui permet de construire la requête de recherche.
        return $this->createQueryBuilder('p')
            // Recherche les éléments dont le nom contient la requête de recherche.
            ->where('p.name LIKE :query')
            // OU recherche les éléments dont la description contient la requête de recherche.
            ->orWhere('p.description LIKE :query')
            // Définit la valeur de la variable "query" pour la requête.
            ->setParameter('query', '%' . $query . '%')
            // Exécute la requête et récupère les résultats.
            ->getQuery()
            ->getResult();
    }
}

```

Parfait la tout fonctionne super bien. Voilà maintenant il reste à

tester cela évidemment, on garde les bonnes habitudes, donc on va sur la page d'accueil et on teste de la même manière que précédemment,

```

$search = $productRepository->searchEngine('nike');
dd($search);

```

On rappelle donc la méthode et on passe un nom de produit qui apparaît plusieurs fois dans notre table **product**, et on voit le résultat, super cela fonctionne bien donc la méthode est bonne, on va pouvoir passer à la suite.

```

HomeController.php on line 24:
array:6 [▼
    0 => App\Enti...\Product {#1043 ▶}
    1 => App\Enti...\Product {#1108 ▶}
    2 => App\Enti...\Product {#1125 ▶}
    3 => App\Enti...\Product {#1132 ▶}
    4 => App\Enti...\Product {#1139 ▶}
    5 => App\Enti...\Product {#1146 ▶}
]

```

Donc notre moteur fonctionne parfaitement, il va donc falloir créer un Controller qui va gérer cela, vous vous doutez de la suite, eh bien oui EXERCICE eheh.

Exercice 19

Correction exercice 19

```

class SearchEngineController extends AbstractController
{
    #[Route('/search/engine', name: 'app_search_engine', methods: ['POST'])]
    public function index(Request $request): Response
    {
}

```

Voilà pour le début du **Controller**, et donc la condition,

```

#[Route('/search/engine', name: 'app_search_engine')]
public function index(Request $request): Response
{
    if ($request->isMethod('POST')) {
        $data = $request->request->all();
    }
}

```

Voici la condition, donc on passe au formulaire de recherche qui se trouve dans le **folder layouts, file navbar.html.twig**, il se situe en bas du fichier, et donc on ajoute ce qui est nécessaire,

```

</ui>
<form method="post" action="{{ path('app_search_engine') }}" class="d-flex align-items-center">
    <input name="word" class="form-control me-2" type="search" placeholder="Rechercher..." />
    <button class="btn btn-outline-success" type="submit">Recherche</button>
</form>

```

Voilà pour le **form**, et donc vous tester cela, et donc on revient au **Controller** pour avancer et faire fonctionner cela,

```

// Vérifie si la requête est de type POST
if ($request->isMethod('POST')){
    // Récupère les données de la requête
    $data = $request->request->all();
    // Récupère le mot-clé de recherche
    $word = $data['word'];

    // Appelle la méthode searchEngine du repository pour obtenir les résultats
    $results = $productRepository->searchEngine($word);
}

// Rendu de la vue search_engine/index.html.twig avec les résultats
return $this->render('search_engine/index.html.twig', [
    'products' => $results,
]);
}

```

Voilà qui est beaucoup mieux désormais, pour ceux qui ont réussi bravo et si vous n'avez pas réussi et bien bravo quand même pour l'effort de recherche c'est la base du boulot de dev. 😊

Il va falloir récupérer le code de la page d'accueil donc dans le **folder home, file index.html.twig**, et vous allez le copier-coller, dans le file du moteur de recherche selon le nom que vous avez donné à votre **Controller** cela a généré le fichier twig en rapport. Pensez à changer ce qui doit l'être, tester sur votre appli, vous devriez avoir une erreur pourquoi ??

Et bien on a la pagination mais nous n'en avons pas besoin donc on peut supprimer la ligne de pagination. Et là cela fonctionne très bien, on peut ajouter le **data-turbo a false** dans le form histoire de booster cela mais on est bon.

On a un léger souci, quand on recharge la page on se retrouve avec une erreur, on nous dit que seule la méthode **POST** est allouée à cette adresse, qui est recherché en **GET**. On va donc modifier cela en **GET** pour régler ce souci, il faut également modifier un petit truc comme on passe en **GET**,

```
// Vérifie si la requête est de type GET
if ($request->isMethod('GET')){
    // Récupère les données de la requête
    $data = $request->query->all();
```

Pour la récupération on utilise désormais le query comme c'est en **GET**, on actualise la page d'accueil et on effectue une nouvelle recherche, et l'erreur a disparu si on recharge car on est désormais en **GET** donc ça effectue le même requête. Dans le Controller on peut envoyer le mot rechercher pour l'afficher,

```
return $this->render('search_engine/index.html.twig', [
    'products' => $results,
    'word' => $word,
]);
```

Et comme on a cette variable qui a été envoyé on peut l'afficher, donc on va sur le front dans le file **index.html.twig** dans le folder **searchEngine** si vous l'avais appelé comme cela,

```
{% block body %}
<div class="container">
    <h1 class="mb-5 mt-5">Résultats de la recherche du mot " {{ word }} " </h1>
    <div class="row">
```

Et voilà c'est super comme cela, bravo à tous. Pensez à aller regarder sur la doc pour ajouter le CSRF token pour protéger votre formulaire encore plus.

Vous êtes donc arrivé au bout de ce projet, et là après avoir géré le CSRF token et bien vous pouvez vous nommer Développeur Symfony, évidemment la compréhension est importante, mais selon moi rien ne vaut la pratique et là vous avec un projet très complet et tout à fait opérationnel, il reste évidemment du front à faire selon vos goûts, j'ai fait assez simple mais cela est tout à fait fonctionnel. Je vous laisse du temps pour marquer votre projet de votre empreinte et modifier la front à moins que vous l'ayez déjà fait au fur et à mesure, surprenez-moi et surtout faites-vous plaisir, c'est votre premier projet complet, alors soyez fier de vous, car moi je le suis.

