



# DOSSIER DE PROJET DE FIN DE FORMATION

## Création d'un site E-commerce sur Symfony 5

14.02.2022

Edouard Abgrall

Formation Développeur Web et Web Mobile 2021-2022

Pour le Titre Professionnel Développeur Web et Web mobile de Niveau III

Fontenay-le-Comte

## SOMMAIRE

- Remerciements
- Introduction
- Cahier des Charges
- Liste de compétences du référentiel
  - Compétences professionnelles
  - Compétences transversales
- Présentation du site internet
- Logiciels utilisés pour le Projet et Framework utilisés
- Organisation du travail et Installation de l'environnement de développement
- Développement du site internet
- Création d'un back office pour les administrateurs
- Conclusion

## REMERCIEMENTS

Je tiens, dans un premier temps, à remercier mon formateur, Jérôme Lesaint, pour nous avoir fourni un enseignement de qualité tout au long de la formation ainsi que pour sa pédagogie et sa patience.

Mes camarades de la formation Développeur Web et Web Mobile, qui m'ont permis de surmonter les difficultés techniques et rendu l'atmosphère de travail agréable pendant ces 9 mois.

La communauté de développeurs présente sur internet, pour leur travail en amont et pour la résolution de problèmes qui m'a permis d'apporter des réponses à mes problématiques.

Le personnel de L'AFPA et du CMFP qui permet à nous, militaires, d'envisager un avenir après notre carrière militaire et nous donne les moyens d'y parvenir.

Ma famille, ma compagne Elise, sans qui je n'aurais probablement pas pris l'initiative de me reconverter et qui ont su m'apporter un soutien sans faille.

Merci à toutes les personnes qui ont pu m'apporter un soutien, moral ou technique, lors de la réalisation de ce projet.

Et enfin, merci à vous, membres du jury, de prendre le temps de me lire et de m'écouter.

## INTRODUCTION

**995 868 <sup>1</sup>**, c'est le nombre de créations de micro entreprises pendant l'année 2021, une hausse record par rapport à l'année précédente, marquée par l'arrivée du premier confinement.

Ce nombre représente à lui seul la volonté d'un nombre important de personnes souhaitant se lancer dans l'aventure de l'auto-entreprenariat et de s'émanciper des modèles classiques de travail dans la vie active.

Comme nous avons pu le constater lors des différents confinements survenus en 2020 et 2021, la population a dû s'adapter et apprendre à utiliser de plus en plus internet pour subvenir à ses besoins, aussi bien essentiels que secondaires.

Le pourcentage de ventes en ligne est passé de 9,8% à 13,4% <sup>2</sup> en France en 2021, soit l'équivalent de 3 à 4 ans de croissance en une année. Cela se traduit par une forte demande de ces nouveaux commerces pour s'établir en ligne afin de toucher un maximum de potentiels clients.

C'est là le but de ce projet : proposer à ces nouveaux entrepreneurs se lançant dans le commerce, un site de boutique e-commerce simple d'utilisation pour des personnes non-initiés au langage informatique, leur permettant de toucher un public plus large, en addition à une boutique physique ou non.

Ils pourront accéder à une base de données regroupant les informations de leurs clients ainsi que gérer le contenu du site, dans l'ajout de nouveaux produits, la modification de ceux-ci, le paiement, et l'envoi de mails commerciaux.

Le site présenté est un template qui nécessitera une adaptation visuelle personnalisée en fonction de chaque entreprise, mais le fonctionnement en Back-End restera sensiblement le même.

Pour la création de ce site, nous utiliserons différentes technologies, frameworks et logiciels qui seront détaillés ci-dessous.

## CAHIERS DES CHARGES

Le site étant un projet personnel, le cahier des charges reste relativement simple, le but étant d'identifier les besoins et les utilisations des futures entreprises.

### Objectifs du site :

Plateforme de vente en ligne, proposant une expérience de qualité aux visiteurs, leur permettant de réaliser des achats rapidement.

Le Back-office du site doit permettre une gestion quotidienne des activités e-commerce ( suivi des commandes, mise à jour du catalogue produit) sans nécessiter d'intervention technique.

### Cible du site :

Le site étant un template, n'importe quel consommateur peut être visé en fonction de la gamme de produits.

Etant une boutique en ligne, le public le plus touché sera une clientèle relativement jeune (18-35 ans) utilisant les supports digitaux de manière intensive.

### Charte Graphique :

À la charge du Client, l'adaptation sera mise en place lors de la réception.

Mise en place d'un prototype ou preuve de concept afin d'aider à la projection du client.

### Arborescence du site :

(cf annexe 1 )

### Description fonctionnelle :

Le site fonctionnera avec le framework Symfony, avec une base de données sous PhpMyAdmin, il utilisera la solution de paiement "Stripe", la gestion des emails par "Mailjet" et on utilisera Bootstrap pour l'aspect responsive.

## LISTE DES COMPÉTENCES DU RÉFÉRENTIEL

- Maquetter une application
- Réaliser une interface utilisateur web statique et adaptable
- Développer une interface utilisateur web dynamique
- Créer une base de données
- Développer les composants d'accès aux données
- Développer la partie backend d'une application web ou web mobile

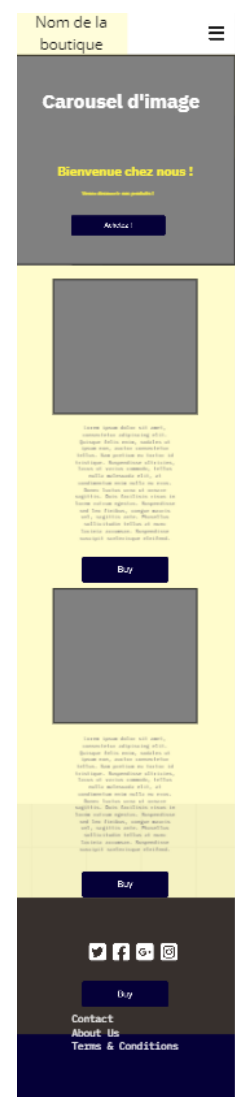
## PRÉSENTATION DU SITE INTERNET

Le projet étant destiné à n'importe quel prospect, il était important de garder une identité visuelle neutre dans laquelle chacun pouvait se projeter. Et, lors d'une présentation, facilement implémenter des idées que le client potentiel aurait. Je suis donc parti de ce postulat pour maquetter le site internet.

Je suis ressorti avec 2 maquettes, une pour le format desktop et une pour le format mobile.

La page d'accueil se présenterait comme suit :

- Un header comprenant les liens indispensables (produits, présentation, contact, connexion/inscription qui se transforme en accès à l'espace membre une fois l'utilisateur connecté, accès au panier).
- Un carrousel d'images pour présenter les nouveautés que la boutique veut mettre en avant.
- Des articles de contenu intégrés par la boutique composé d'une photo ainsi que d'un texte de présentation, afin de promouvoir certains produits ou événements avec des boutons de redirections.
- Un footer comprenant les liens utiles tels que la politique de confidentialité, le contact, une foire à questions, des liens vers les différents réseaux sociaux.



Pour ce qui est de la version mobile, je ne voulais pas trop changer la version desktop pour que le site reste reconnaissable par l'utilisateur qui voudrait s'y rendre une seconde fois, cette fois-ci sur le mobile.

Il est identique dans les tons, à la différence près que :

- Les photos ne sont plus alignées avec le texte mais l'une en dessous de l'autre pour fluidifier la lecture de la page.
- Le header se transforme en menu dit "burger" qui s'étend lors de l'appui et propose les mêmes liens et redirections vers d'autres pages du site.

## OUTILS, LANGAGES ET FRAMEWORK UTILISÉ

Afin de créer ce site, j'ai eu recours en permanence à des logiciels et des framework, afin de nous permettre d'abattre le travail de la façon la plus efficace possible.

### Outils utilisé :

(cf annexe 1-2)

### Traduction paragraphe :

**Anglais :**

#### **Visual Studio Code :**

Visual Studio Code is an extensible code editor developed by Microsoft for Windows, Linux and MacOS.

The range of functionality includes debugging, highlighting syntax, smart completion of code, refactoring code and Git integrated in it. Users can customize the editor with all sorts of themes, keyboard shortcuts, set preferences and download extensions that add extended functionality.

#### **PHPMysqlAdmin:**

PHPMysqlAdmin is an online management web application for relational databases like MySQL and MariaDB. It has been conceived mainly using PHP.

It is one of the most famous interfaces for managing MySQL databases on a PHP Server. This interface allows, without extended knowledge in DB, all kinds of requests like table creation, insertion of data, update, suppression and modification of the Database structure.

## Francais

- **Visual studio code :**



Visual Studio Code est un éditeur de code extensible développé par Microsoft pour Windows, Linux et MacOS.

Les fonctionnalités incluent la prise en charge du débogage, la mise en évidence de la syntaxe, la complétion intelligente de code, la refactorisation du code et Git intégré. Les utilisateurs peuvent modifier le thème, les raccourcis clavier, les préférences et installer des extensions qui ajoutent des fonctionnalités supplémentaires.

- **PHPMyAdmin :**



PhpMyAdmin est une application Web de gestion pour les SGBDR MySQL et MariaDB. Réalisée principalement en PHP, il s'agit de l'une des plus célèbres interfaces pour gérer une base de données MySQL sur un serveur PHP. Cette interface permet d'exécuter sans grande connaissance en BDD des requêtes comme les créations de table, insertions, mises à jour, suppressions et modifications de structures de la base de données.

## Langage et framework utilisé :

- **Symfony :**



Symfony est un ensemble de composants PHP ainsi qu'un framework MVC libre écrit en PHP, il fournit des fonctionnalités modulables et adaptables qui permettent de faciliter et d'accélérer le développement d'un site web.

- **PHP:**





PHP: Hypertext Preprocessor, plus connu sous son sigle PHP, est un langage de programmation libre principalement utilisé pour produire des pages Web dynamiques via un serveur HTTP, mais pouvant également fonctionner comme n'importe quel langage interprété de façon locale. PHP est un langage Orienté Objet.

Il a permis de créer un grand nombre de sites web célèbres, comme Facebook et Wikipedia. Il est considéré comme une base de sites web dits dynamiques mais également des applications web.

## INSTALLATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

### Installation de Visual Studio Code

(cf annexe 3)

Avoir un éditeur de code ergonomique et efficace permet d'optimiser son temps au maximum. L'autocomplétion, la colorisation, indentation, Terminal intégré, ne sont que quelques exemples de ce que peut apporter un IDE (Integrated Development Environment) bien configuré.

Comme pour la plupart des installations, je me suis rendu sur le site et ai suivi la procédure indiquée. Une fois installé, l'installation d'extensions est une étape cruciale afin d'optimiser le travail.

Quelques exemple d'extensions qui, à mon sens, facilitent le travail :

- Bracket Pair colorizer:  
Permet de colorer parenthèses, crochets et accolade afin de rapidement identifier les paires.
- Live-share:  
Permet lors de projet commun de pouvoir travailler à plusieurs sur le même fichier afin d'accélérer le développement.
- PHP Intelephense :  
Permet d'utiliser l'auto-complétion pour PHP, d'ajouter automatiquement des Usa dans le code et offre des propositions de résolution d'erreur.
- Prettier :

Permet de formater son code aux règles d'indentation en vigueur, rend plus lisible le code et permet donc une recherche plus rapide des éléments à l'intérieur de celui-ci.

Une fois cela installé, je suis passé à l'installation de Symfony.

## Installation de symfony

### (cf annexe 3)

L'installation du projet par Symfony représente le cœur même de celui-ci. Une installation ratée signifiera des bugs par la suite lors du développement. Il faut donc prêter une attention particulière à cette étape.

J'ai utilisé la documentation de Symfony afin de l'installer

(<https://symfony.com/doc/current/setup.html>).

Il faut s'assurer que la version de PHP est au minimum 8.0.2

L'installation au préalable de **Composer** est requise, pour cela, il suffit de se rendre sur la page d'installation du site et suivre les instructions.

( <https://getcomposer.org/doc/00-intro.md>).

Il faut télécharger le fichier exécutable, récupérer Composer.phar et ensuite lancer en invite de commande les lignes suivante :

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
'906a84df04cea2aa72f40b5f787e49f22d4c2f19492ac310e8cba5b96ac8b64115ac402c8cd292
b8a03482574915d1a8') { echo 'Installer verified'; } else { echo 'Installer
corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Une fois fait, on passe maintenant à l'installation de la CLI de Symfony :

```
scoop install symfony-cli
```

Une fois faite, on peut maintenant créer notre projet en utilisant la commande suivante :

```
symfony new my_project_directory --full
```

(la commande project/skeleton de la docs est utilisée de façon sous-jacente).

Cette installation nous permettra d'avoir un dossier complet, avec toutes les dépendances nécessaires pour développer un site internet sur Symfony.

En prévision, On crée un fichier .htaccess afin de gérer les redirections.

**Ligne de commande : composer require symfony/apache-pack**

Les fichiers .htaccess sont des fichiers de configuration (Apache) permettant de définir des règles bien spécifiques dans un répertoire. Ce type de fichier peut être utilisé pour réaliser des redirections ou protéger un répertoire par un mot de passe.

## Organisation et architecture de Symfony 5

Les 2 dossiers les plus importants, et qui seront le plus utilisés pendant le développement sont le dossier **"src"** et le dossier **"templates"**,

Que contiennent-ils ?

Dans le dossier **"src"** se trouve : **"Entity, Controller et Repository"**, généré automatiquement par l'installation de Symfony.

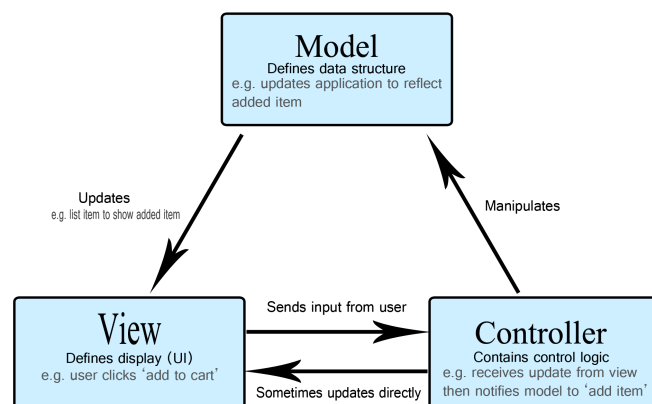
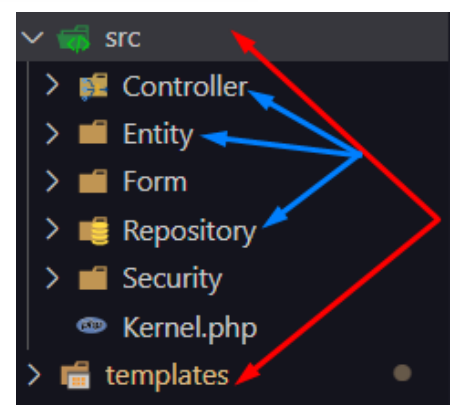
Et **"templates"** est le dossier qui regroupe ce qui s'appellera les **VUES (views)**, qui sont générées par le moteur de templates **TWIG**. Nous y reviendrons plus tard.

À quoi servent les 3 sous-dossiers de **"src"** et le dossier **"templates"**?

Dans une logique de création d'un **"MVC"** (Model View Controller) qui apparaît comme la logique à appliquer dans le développement du site:

( Doc :

<https://developer.mozilla.org/en-US/docs/Glossary/MVC> )



- **Controller** : Le controller contient la logique de mise à jour du Model (Entity) ou des vues, en réponse aux actions des utilisateurs. Il récupère les requêtes triées par des routes et définit la réponse appropriée.
- **Entity** : Il définit quelles données seront contenues, si l'état des données change (ajout, suppression) il informera la **Vue** et l'affichage sera modifié.

Ce sont les Classe qui représentent la structure de nos tables (BDD) et dont les attributs représentent les champs contenus dans ces tables.

- **Repository** : Il centralise tout ce qui touche à la récupération de nos données. L'avantage est qu'il nous évite de taper des requêtes SQL dans notre code.
- **Templates** : Ce dossier contient les **Vues**, il permet d'indiquer comment l'affichage se fera lorsque l'utilisateur emprunte la route dans le controlleur qui correspond à cette Vue.

### Pour resumer :

Les actions possibles de l'utilisateur passent par le **controller** qui définit ces actions par des routes, ces routes entraînent des requêtes dans la base de données définies par le **model**, qui lui renvoie des données, qui sont ensuite affichées sur la **Vue** liée à la route empruntée.

Le dossier contient aussi d'autres sous-dossiers tels que :

- **Public** : il contiendra le CSS, le JavaScript et les différentes photos utilisées.
- **Migration** : il permet de stocker les fichiers utiles à la mise à jour de la base de données.
- **.env** : il permet de configurer l'environnement de développement (dev > prod, DEBUG,) mais aussi de faire la liaison avec la base de données.

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

## DÉVELOPPEMENT DU SITE

### Création de la première page

(cf annexe 3)

Grâce à l'arborescence faite du site, je sais que la première page à développer est la page d'accueil ( nommé "home" ici, juste "/" dans le projet.).

Dans Symfony, pour créer une page et ce qui va avec ( routes,vue... ), on peut utiliser la ligne de commande suivante dans le terminal :

**Symfony console make:controller**

Cette commande nous demandera un nom pour le controller et agira ensuite seule, elle créera dans le projet un controller ainsi que la vue liée à celui-ci.

Une route sera alors attribuée pour voir le contenu de la page, ici `"/home"` , Ce processus sera ainsi répété pour chaque nouvelle route créée. Si on crée une route, on doit créer la vue qui lui correspond.

Afin de rendre le site responsive, je me suis rendu sur Bootstrap pour prendre un thème prédéfini neutre, qui permet d'être responsive et d'avoir un header qui sera utilisé à chaque page.

Le lien avec la bibliothèque Bootstrap se fait grâce a un lien situé dans la balise `<head>` du fichier de base et une balise `<script>` en bas de page.

```
<body>
<header>
  <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
    <a class="navbar-brand" href="{{path('home')}}">Projet Boutique E-Commerce.</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbar">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarCollapse">
      <ul class="navbar-nav mr-auto">
        <li class="nav-item active">
          <a class="nav-link" href="{{path('products')}}">Nos produits</a>
        </li>
        <li class="nav-item active">
          <a class="nav-link" href="">Qui sommes-nous ? </a>
        </li>
        <li class="nav-item active">
          <a class="nav-link" href="{{path('contact')}}">Contact</a>
        </li>
      </ul>
      <div class="navbar-item-custom">
        {% if app.user %} {# Création d'une condition pour l'affichage en fonction de la co
          <a href="{{path('account')}}">Mon Compte <small>{{app.user.firstname}}</small>
        {% else %}
          <a href="{{path('app_login')}}">Connexion</a> | <a href="{{path('register')}}">
        {% endif %}
      </div>
    </div>
  </nav>
</header>
```

```
<link href= "{{ asset('assets/css/bootstrap.min.css') }}"
rel="stylesheet">
```

La balise `{{asset(#)}}` indique a la vue d'aller se servir dans le dossier **"public"**

Au lieu de devoir réécrire le code correspondant à ce thème dans chaque nouvelle vue, Symfony et TWIG, nous permettent d'utiliser une fonction **d'héritage** qu'on appelle avec **"extends"** et qui reprend les éléments contenus dans le fichier base ( link, script, etc).

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

{% block title %}{{product.name}}{% endblock %}

{% block content %}
  <div class="row">
    <div class="col-md-5">
      
    </div>
  </div>
{% endblock %}
```

RAPPEL DE LA VUE DE BASE

Les Vues générées par Symfony permettent de segmenter son code en utilisant un système de “**block**”. Dans le fichier de base, ceux-ci peuvent être laissés vides mais doivent apparaître à l’emplacement où l’on veut rajouter du contenu sur une autre vue il est possible ensuite de rajouter du contenu. Il s’écrit de la manière suivante :

```
{% block body %}

{% endblock %}
```

Et s’utilise ainsi : **(cf annexe 4 )**

En exemple, si le block **title** n’est pas défini dans une nouvelle vue, le contenu de celui-ci sera récupéré du fichier dont il est étendu.

On entoure le bloc de code qui affiche le “carousel” afin de pouvoir l’appeler dans la page home, mais qu’il ne soit pas inhérent au fichier base.html.twig.

## Création de la première entité.

Maintenant que la page d’accueil est définie, nous allons créer notre première entité, qui correspond à la première table de notre base de données: - **Les Users**. (Utilisateur). Ils auront à terme la possibilité de s’inscrire, se connecter, d’accéder à un espace membre ou admin (en fonction du **rôle donné**), de changer de mot de passe, d’adresse de livraison ou encore d’afficher l’historique de leur commande.

Pour ce faire, je vais créer cette entité, un formulaire d’inscription/connexion, et un espace membre.

Pour créer l’entité, Symfony nous permet encore d’être aidé en le faisant en ligne de commande sur le terminal qui se résume à :

```
php bin/console make:user
```

```
PS C:\xampp\htdocs\test-symfony> symfony console make:user

The name of the security user class (e.g. User) [User]:
>

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
>

Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
>

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed by some other system (e.g. a single sign-on server).
Does this app need to hash/check user passwords? (yes/no) [yes]:
>
```

Dans le fichier base.html.twig

Cette commande nous a créée 2 fichiers (User.php et UserRepository.php) et en a mis 2 à jour (user.php et security.yaml)

```
/**
 * @ORM\Column(type="string", length=180, unique=true)
 */
private $email;
```

EXEMPLE DE PROPRIETE GENERE PAR LA COMMANDE

Symfony génère automatiquement les **getter et setter** liés à ces propriétés afin de pouvoir les récupérer ou les modifier en base de données.

Le fichier **repository** créé par la commande définit les méthodes de récupération en BDD, ce qui nous évite de coder manuellement des requêtes SQL. Cela est rendu possible par DOCTRINE, qui est un ORM (*Object Relational Mapping*). (cf annexe 5)

## Création de la base de données et première migration. ( Cf annexe 9 )

Pour créer une base de données, une fois connecté à **MySQL** via **XAMPP**, en entrant dans le terminal : **Doctrine:database:create**. Une base de données portant le nom du projet s'ajoute.

Il est temps d'ajouter notre première table, User, car même si l'entité est créée, l'ajout en BDD n'est pas automatique. La première chose est de faire en sorte que la requête SQL se crée. Pour cela, dans le terminal, on entre : **symfony console make:migration**

Mais créer le fichier de migration ne suffit pas, il faut par la suite exécuter cette requête, en entrant : **Symfony console doctrine:migration:migrate**

```
public function up(Schema $schema): void
{
    // this up() migration is auto-generated, please modify it to your needs
    $this->addSql('CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, email VARCHAR(180) NOT NULL, PRIMARY KEY (id))');
}
```

EXEMPLE DE REQUETE DANS LE FICHIER DE MIGRATION

Une fois fait, on peut constater que sa table en BDD comporte bien les champs inclus dans la requête de la migration.

Cette procédure sera répétée à chaque création ou modification d'entité pour agrémenter la BDD.

## Création du formulaire d'inscription.

Maintenant que l'entité est créée est qu'elle est aussi une table dans notre base de données, Il faut permettre aux utilisateurs de s'inscrire.

Pour cela nous allons créer un formulaire d'inscription. On répète la commande de création de controller en lui donnant le nom **RegisterController**, et lui donnons la route `"/inscription"`.

La vue twig ne contient pour l'instant rien mais en faisant appel au block **base** on obtient rapidement notre squelette de page. Je crée un filtre pour éviter que le carrousel s'ajoute.

Pour afficher le bloc, il faut dorénavant l'appeler sur les vues.

```
56 <main role="main">
57 {# Création d'une condition pour afficher le carrousel que sur les pages adéquates #}
58 {% if block('carousel') is defined %}
59
60 <div id="myCarousel" class="carousel slide" data-ride="carousel">
```

Pour la création de formulaires, encore une fois Symfony nous propose une solution pour lier le formulaire directement à la table concernée avec la commande **symfony console make:form**. Nous liions donc ce formulaire à la table user, ce qui permet lors de sa complétion de fournir des entrées dans la table.

La commande nous indique qu'elle a créé un nouveau fichier, appelé **RegisterType.php** dans le dossier **"form"**. Dans ce fichier nous allons indiquer quels inputs introduire dans le formulaire. Par défaut, Symfony entre autant d'inputs qu'il y a de propriétés dans l'entité. Nous enlevons donc la possibilité d'entrer/modifier son rôle car cela sera réservé à l'admin.

```
PS C:\xampp\htdocs\projet_boutique_e_commerce> symfony console make:form

The name of the form class (e.g. OrangePuppyType):
> Register

The name of Entity or fully qualified model class name that the new form will be bound to (empty for
> User

created: src/Form/RegisterType.php
```

Pour entrer un champ, **->add()** est utilisé, on renseigne la propriété correspondante, et ensuite on ouvre un tableau pour attribuer différentes propriétés à l'input, tel que des classes, un label, une contrainte de saisie par exemple.

```
->add('email', EmailType::class,[
    'label'=>'Email',
    'required'=> true,
    'constraints' => new Length(min:2,max:100),
    'attr'=>[
        'placeholder'=>"Merci d'entrer votre Email"
    ]
])
```

Le but est maintenant d'afficher ce formulaire sur la vue, comment faire ?

On retourne dans le controller adéquat (RegisterController ici) afin de quérir le formulaire lorsque l'utilisateur emprunte la route.

Dans un premier temps, on indique au controller qu'on va utiliser, l'entité User et le formulaire créé à l'aide de **use**, cela indique le chemin à prendre lors de leur utilisation.

```
use App\Entity\User;
use App\Form\RegisterType;
```

On instancie ensuite la classe User dans la fonction index() et on utilise la méthode createForm() en indiquant que l'on veut le modèle créé dans RegisterType et la classe user.

```
$user = new User();
$form = $this->createForm(RegisterType::class, $user);
```



Dans cette première partie, l'affichage du formulaire est paramétré, il suffit maintenant de faire passer ces variables à la vue, comment ?

En utilisant **return** avec l'indication du fichier où on veut le rendu (RegisterType/index.html.twig) et en tableau associé la(es) variable(s): ici on utilise en plus la méthode `CreateView()` pour qu'il génère un formulaire.

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

{{form(form)}}
```

Quand à son appel sur la vue twig, il suffit d'écrire :

Une fois que ces étapes sont respectées et que vous vous rendez sur la route correspondante, un formulaire s'affiche. (cf annexe 5)

Afin que la stylisation des formulaires soit automatique, le package bootstrap pour twig a été installé en entrant dans le fichier `packages/twig.yaml` :

```
twig:
    default_path: '%kernel.project_dir%/templates'
    form_themes: ['bootstrap_5_layout.html.twig']
```

## Sauvegarde des informations dans la base de données.

Le but ici est de faire comprendre au contrôleur que :

- dès que le formulaire est envoyé et validé, je veux que tu traites l'information, pour ce qui est du rôle, Symfony paramètre par défaut à `User`, ce qui pour le moment suffit.

On utilisera la Doctrine pour envoyer les infos dans la base de données, on demandera au contrôleur de traiter l'information, de figer cette information pour l'envoyer en Base de données avec les méthodes `persist()`, et enfin de l'envoyer avec `flush()`.

```
if($form->isSubmitted() && $form->isValid()) {
    //La variable user prends la valeur du formulaire.
    $user= $form->getData();

    //
    $this->entityManager->persist($user); //Figeage des data de L
    $this->entityManager->flush(); // execute la persistance, pre
```

## Encodage du mot de passe.

Même si Symfony nous demande si on veut encoder nos mots de passe et indique avec quoi les encrypter dans le **security.yaml**:

```
password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    App\Entity\User:
        algorithm: auto
```

*"auto" est actuellement paramétré sur "Bcrypt" (cf [symfony/doc current/security/passwords.html](https://symfony.com/doc/current/security/passwords.html))*

Il faut quand même lui indiquer la marche à suivre. Pour ce faire, on fait une injection de dépendance dans la fonction `index()`, celle de `UserPasswordHasherInterface`. Celui-ci, nouveau dans Symfony 5.3, extrait tout le code existant relatif à l'encodage et les renomme "hash".

```
#[Route('/inscription', name: 'register')]
public function index(Request $request, UserPasswordHasherInterface $passwordHasher)
{
```

Une fois cela mis, on récupère le mot de passe transmis par le formulaire et avant d'envoyer les données, ce mot de passe entré par l'utilisateur passe par la fonction `hashPassword()` et est `set()` encrypté. À ce moment-là seulement on injecte les données dans la bdd.

```
//Hashage du mot de passe
$password = $passwordHasher->hashPassword($user, $user->getPassword());
// on set le MDP pour l'injecter correctement dans la base de données.
$user->setPassword($password);
```

Un chaîne de caractère simple devient donc :

password

\$2y\$13\$1Niuc.blupHCkduCFXYyz.sPlbx21eGO3Ry59YfU01a...

## Création du login pour les utilisateur

Maintenant que la création de compte utilisateur est fonctionnelle, il faut leur permettre de se connecter. Symfony propose une solution en ligne de commande qui se présente sous la forme de : **symfony console make:auth**, la console nous propose ensuite de choisir la méthode d'authentification, elle nous donne la possibilité d'un formulaire de connexion prédéfinie. C'est le choix effectué ici, il nous demande le nom du contrôleur et si on veut une route de déconnexion.

On entre la route convenue dans le `securityController` et le bloc "content" dans le template.

```
public function authenticate(Request $request): PassportInterface
{
    $email = $request->request->get('email', '');
    $request->getSession()->set(Security::LAST_USERNAME, $email);

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

Logged in as **edouard.abgrall@gmail.com**  
 Authenticated **Yes**  
 Token class **PostAuthenticationToken**  
 Firewall name **main**  
 Actions **Logout**  
 edouard.abgrall@gmail.com 9 ms

On remplit le formulaire, le soumet, et dans le debugger on peut voir que l'utilisateur est connecté.

## Création de vue privée pour l'utilisateur.

En créant les controller et templates adéquats, on a la possibilité de créer des vues accessibles seulement par un utilisateur connecté, comment faire ?

Il faut se rendre dans le fichier **security.yaml** qui nous donne la possibilité de restreindre l'accès à un certain type d'utilisateur, étant donné que lorsque qu'un compte est créé, il a par défaut le ROLE\_USER, on filtre l'accès à ces pages seulement pour les utilisateurs ayant ce rôle.

```
- { path: ^/compte, roles: ROLE_USER }//Compte est la route vers la vue
```

On modifie de fait le header afin de proposer un accès à ces vues privées et la possibilité de se déconnecter plutôt que de s'inscrire et se connecter.

```
{% if app.user %} {# Création d'une condition pour l'affichage en fonction de la connexion #}
  <a href="{{path('account')}}">Mon Compte <small>{{app.user.firstname}}</small></a> | <a href="{{path('app_logout')}}">Déconnexion</a>
{% else %}
  <a href="{{path('app_login')}}">Connexion</a> | <a href="{{path('register')}}">Inscription</a>
{% endif %}
```

## Modification du mot de passe.

Pour qu'un utilisateur puisse changer son mot de passe, on lui propose de remplir un formulaire (en suivant le même procédé que pour la création de compte) et lorsqu'il se rends sur la route, on récupère ses informations, auto-remplissant le formulaire, en rendant tous les champs à part le mot de passe *Disabled -> true (impossible a modifier)*.

On rajoute le code correspondant au mot de passe dans le formulaire, et lors de l'envoi de celui-ci, à la récupération dans le controller :

-On vérifie si l'ancien mot de passe est le bon, ce qui permet de vérifier la légitimité du changement.

```
->add('password', RepeatedType::class,[
    'type'=>PasswordType::class,
    'invalid_message'=>'Le mot de passe et la confirmation ne sont pas identique.',
    'required'=> true,
    'label'=>'Mot de Passe',
    'first_options' => [ 'label' => 'Mot de passe'],
    'second_options' => [ 'label' => 'Confirmer votre Mot de passe'],
])
```

-On crypte le nouveau mot de passe, puis on utilise le “setter” de password dans l'entité User afin de modifier celui-ci.

```
if ($form->isSubmitted() && $form->isValid()) { // si le formulaire est juste et valide
    $old_pwd = $form->get('old_password')->getData(); // recuperation du mot de passe actuel dans le formulaire

    if($encoder->isPasswordValid($user,$old_pwd)){ // comparaison entre le mot de passe utilisateur et celui dans le formulaire.
        $new_pwd= $form->get('new_password')->getData(); // recuperation de la donnée "new_password" du formulaire.
        $password = $encoder->hashPassword($user, $new_pwd); // encodage de cette donnée

        $user->setPassword($password); // on set le mot de passe avec cette nouvelle data encodée
        $this->entityManager->flush(); // execute la persistance, prends la data et enregistre le formulaire
        $notification= 'Votre mot de passe a bien été mis a jour';
    }
}
```

## Création des entités catégorie et produit.

### Catégorie

La création de l'entité Catégorie a pour seul but de pouvoir filtrer les produits à terme. On utilise donc la commande Symfony console make:entity avec pour seule propriété son nom. On la liera ensuite avec les produits.

Une migration est effectuée à chaque création ou modification d'entités.

### Produit

Pour la création des produits, des propriétés supplémentaires seront nécessaires, elles seront composées de :

- Un nom
- Un slug ( une chaine de caractère pour rendre les URLs plus lisible)
- Une illustration
- Une description
- Un sous-titre (subtitles)

-Un prix (de type float au lieu de string)

-Une **catégorie** (de type “**relation**” la console demandera alors avec quelle entité la lier et le type de relation. Ici, les produits ne peuvent avoir qu'une catégorie mais celle-ci peut avoir plusieurs produits, on choisit donc la relation ManyToOne, et l'impossibilité d'être nulle.

Maintenant que les entités sont créées, on ajoute des produits dans la table depuis le back office (cf Création du Back-office). Le but est maintenant d'afficher les produits aux utilisateurs dans une vue et que celle-ci reprenne les données relatives aux produits.

On crée donc un contrôleur dans lequel on va effectuer des requêtes pour récupérer les données de la table produits.

Pour cela on va utiliser l'entityManager Interface qui va interroger la base de données avec les fonctions :

- *FindBy(ex:price)* : trouve-moi les produits ayant ce prix
- *FindAll* : récupère tous les produits
- *FindOneBy(ex:id)* : Trouve-moi le produit correspondant à tel ID

On génère donc un construct de l'EntityManager et dans la fonction index du contrôleur, on effectue la requête *FindAll* pour récupérer tous les produits. En affichant le résultat on obtient un tableau reprenant les propriétés associées ainsi que leur valeur (nom,prix... )

La requête est de la forme:

```
$products = $this->entityManager->getRepository(Product::class)->findAll(); // on recupere le repository de la class product
//et on fait un findAll() pour tous les récupérer
```

Puis on transmet la variable \$products à la vue TWIG.

Pour l'affichage des données sur le template, on les insère en utilisant :

`{{product.name(une des entrées du tableau inséré dans la variable)}}`

Ou **product** correspond au nom de la variable passée et **name** correspond à une entrée dans le tableau inséré dans la variable).

Ainsi pour afficher tous les produits on utilise la boucle **"for"** qui permet d'afficher les informations pour chaque produit trouvé dans la base de données.

```
{% block title %}Nos Produit - Projet de boutique E-commerce {% endblock %}

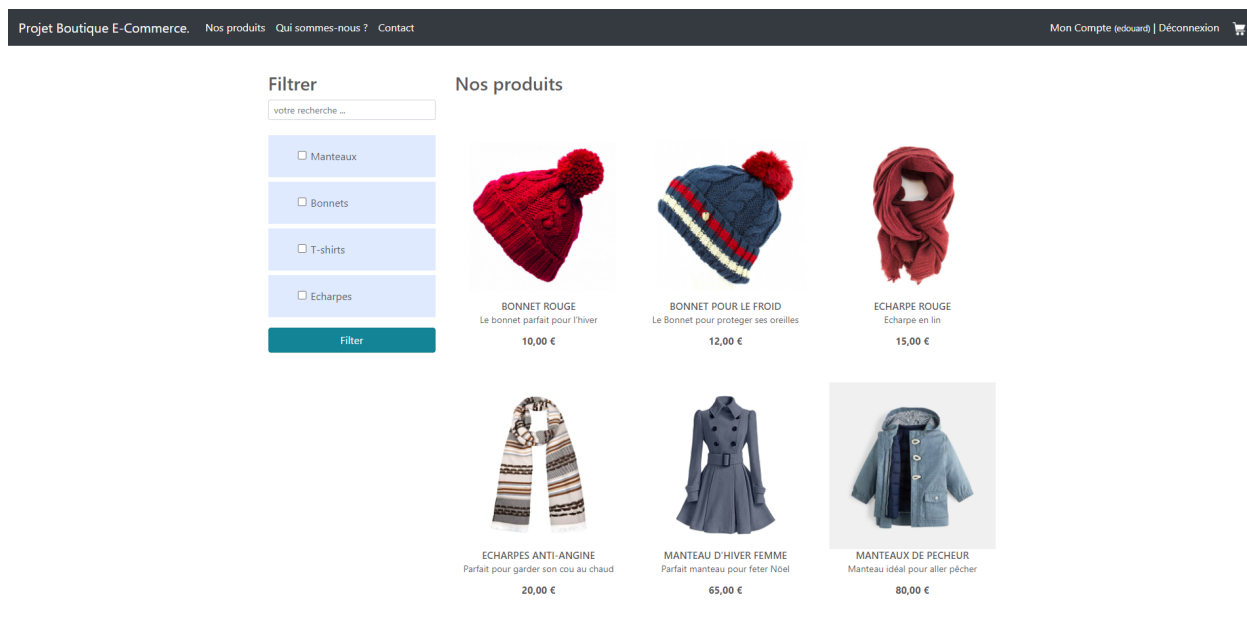
{% block content %}
<h1>
Nos produits
</h1>

<div class='row product-container'>
  {% for product in products %}

    <div class="col-md-4">
      <div class="product-item text-center">
        
        <h5> {{product.name}}</h5>
        <span class="product-subtitles">{{product.subtitle}}</span>
        <span class="product-price">{{(product.price / 100) | number_format(2,',','.')}} €</span>
      </div>
    </div>

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

Avec l'ajout de quelques propriétés de style sur le fichier CSS, l'utilisateur voit apparaître :



Chaque produit ayant une valeur propre pour chaque propriété peut être affiché avec les informations qui lui correspondent.

Une fois que l'utilisateur clique sur le produit, grâce à une nouvelle fonction dans le controller (donc nouvelle route), on récupère les informations de ce produit afin d'afficher une fiche produit.

Pour la création de la route, on ajoute en paramètre de la route "{slug}", ce qui permet avec une injection de dépendance de la variable **\$slug** de créer une route spécifique pour chaque article sans avoir à la spécifier à chaque création d'article.

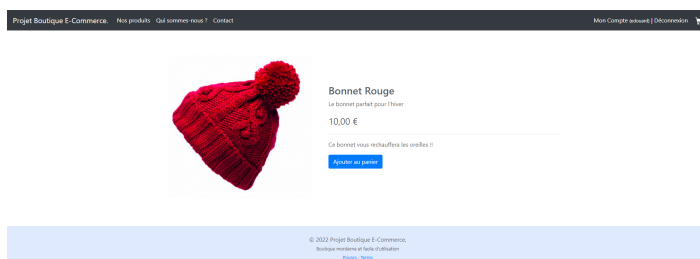
Pour récupérer les informations de l'article, la méthode : **findOneBySlug(\$slug)** sera utilisée puisque cette variable est transmise dans la fonction. Puis de la même manière qu'avec la fonction d'affichage de tous les produits, on utilise le **Repository** pour interroger la base de données.

```
$product=$this->entityManager->getRepository(Product::class)->findOneBySlug($slug)
```

Dans le cas où aucun produit n'est trouvé, une redirection est mise en place pour retourner sur la page des produits.

```
if(!$product){
    return $this->redirectToRoute('products');
}
```

La vue, elle, comprend des informations plus détaillées sur le produit ainsi qu'un bouton pour ajouter le produit au panier.



## Ajout d'une barre de recherche.

La possibilité de filtrer les résultats de recherche est ajoutée afin d'améliorer l'expérience utilisateur,

Pour créer cette fonctionnalité, j'ai dû créer un sous-dossier **class** dans le dossier **src**, un fichier **search.php** est créé à l'intérieur.

On indique avec un *use App\Entity\Category* qu'elle doit accéder aux informations de l'entité Category.

```
class Search
{
    /**
     * @var string
     */
    public $string = '';
    /**
     * @var Category[]
     */
    public $categories= [];
}
```

**String** correspond à la recherche entré par l'utilisateur, et le tableau **category[]** correspond à la récupération des catégories.

On crée par la suite un formulaire (SearchType) qui ne sera finalement qu'un input correspondant à la barre de recherche, ainsi que des checkbox pour pouvoir sélectionner des catégories de produits.

Le même cheminement de validation et de requête dans la base de données est utilisé, à la différence près que la fonction de recherche : **FindWithSearch()** dans le Repository associé est une fonction créée.

```
->add('string', TextType::class, [
    'label' => false,
    'required' => false,
    'attr' => [
        'placeholder' => 'votre recherche ...',
        'class' => 'form-control-sm'
    ]
])
->add('categories', EntityType::class, [
    'label' => false, // pas de label
    'required' => false, // n'est pas requis
    'class' => Category::class, // permet de faire le lien avec Category
    'multiple' => true, // pour en afficher plusieurs puisque c'est un tableau
    'expanded' => true, // permet d'avoir les checkboxes
])
```

```

public function FindWithSearch( Search $search)
{
    $query = $this
    ->createQueryBuilder('p') // on commencer a créer une requete et on la mappe avec une table ( p pour product)
    ->select('c','p') // c pour categorie et p pour product
    ->join( 'p.category','c'); // permet de faire une jointure entre les categorie et les produit

    //CONDITION POUR FAIRE FILTRE AVEC LES CHECKBOXES (CATEGORIE)
    if (!empty($search->categories)) { // lorsque et seulement lorsque des categories sont cochées on crée un filtre
        $query = $query // la query qui a été créée
        ->andWhere('c.id IN(:categories)') // Recherche SQL avec un parametre pour la recherche
        ->setParameter('categories', $search->categories); // on set le parametre par le nom de la variable qu'on a créée
    }

    //CONDITION POUR FAIRE FILTRES AVEC L'INPUT TEXTE
    if (!empty($search->string)) {
        $query = $query // la query qui a été créée
        ->andWhere('p.name LIKE :string') // Recherche SQL avec un parametre pour la recherche (ici ce que l'utilisateur a entré dans sa barre de recherche)
        ->setParameter('string', "%{$search->string}%"); // on set le parametre par le nom de la variable qu'on a créée . "{}%" Permet de faire des recherches partielles
    }
}

```

Cette fonction, qui permet d'interroger la base de données, est une requête SQL écrite en php avec:

->**andWhere** qui correspond à WHERE en SQL

**\$query** qui correspond à SELECT ET JOIN (cf annexe 5)

## Création du panier.

Qu'est ce que la SessionInterface ?

C'est un tableau qui va suivre l'utilisateur pendant sa navigation, le but pour le panier est de transmettre des variables à ce tableau, plus spécialement pour l'ajout de produits, un ID et une quantité. La classe "Cart" sera utilisée pour agréger le panier.

Si on utilise une fonction pour ajouter des éléments au panier, comme ici, et qu'on enlève ce code et rafraîchit, l'ajout figure toujours. Pour supprimer on utilisera la méthode **remove()**. À partir de là, on peut ajouter des produits et par la suite récupérer les données de ces produits dans d'autres pages.

```

$this->session->set('cart', [
    [
        'id'=>52,
        'quantity'=>12
    ]
]);
$cart = $this->session->get('cart');
dd($cart);

```

## Création de la classe "cart"

On effectue le même schéma que pour la classe "Search", sauf qu'à l'intérieur de celle-ci, on créera des fonctions d'ajout { add() }, de suppression, et d'autres pour récupérer toute ou en partie les informations de ce panier, en passant en paramètre l'id du produit concerné.



Ajout, suppression et affichage du panier. (cf annexe 6)

On crée un contrôleur avec des routes qu'on associera aux boutons ou liens "voir mon panier", "ajouter au panier" par exemple, et les fonctions à l'intérieur de ces routes permettent d'interagir avec le panier, directement lié avec la session Interface.

```
// Ajouter au panier
#[Route('/cart/add/{id}', name: 'add_to_cart')]
public function add(Cart $cart,$id) // permet d'emmener la classe Cart dans une variable ainsi que l'id
{
    $cart ->add($id);
}
```

Exemple de lien qui permet d'ajouter un article au panier:

```
<a href="{{path('add_to_cart',{'id' : product.id})}}">
```

Enfin, dans le fichier "**cart.php**", pour l'ajout, on crée une condition qui permet, si le panier comprend déjà le produit, de rajouter +1 à sa quantité.

```
$cart = $this->session->get('cart',[]); // recuperation du panier
if (!empty($cart[$id])) { // si le panier n'est pas vide de l'id spécifique d'un produit alors on rajoute +1
    $cart[$id]++;
}else{ // sinon on set sur 1
    $cart[$id] = 1 ;
}
$this->session->set('cart', $cart); // création ou modification du "panier"
```

Pour afficher le panier à l'utilisateur, la route "/mon-panier" a été choisie. On récupère la variable "\$cart" et la passons au template, et dans celui-ci une boucle est créée afin de récupérer pour chaque produit ajouté ses informations.

On initialise une variable "total" au début de la boucle pour faire un prix total du panier.

Afin de faire la suppression d'article dans le panier, on va créer une fonction dans la classe Cart que l'on appellera dans le contrôleur pour supprimer un article du panier.

Une condition est mise en place pour afficher le tableau et le lien vers l'achat pour les utilisateurs dans le cas où le panier est vide.

*Condition d'accès*

```
{% if cart|length > 0 %}
```

```
{# declaration de variable nulle pour le total du panier #}
{% set total = null %}
{# on parcourt le panier afin de récupérer toute les infos à l'intérieur #}
{% for product in cart %}
<tr>
<td scope="col">

</td>
<th>
{{product.product.name}}<br/>
<small>{{product.product.subtitle}}</small>
</th>
<td>
<a href="{{path('decrease_to_cart',{'id' : product.product.id})}}">

</a>
x {{product.quantity}}
<a href="{{path('add_to_cart',{'id' : product.product.id})}}">

</a>
</td>
<td>{{(product.product.price / 100) |number_format(2,',','.')}} €</td>
<td>{{((product.product.price * product.quantity) / 100) |number_format(2,',','.')}} €</td>
<td>
<a href="{{path('delete_to_cart',{'id' : product.product.id})}}">

</a>
</td>
</tr>
{% set total = total + product.product.price * product.quantity %}
{# On rajoute le prix X la quantité à chaque itération afin d'avoir le prix total du panier #}
{% endfor %}
```

```
//suppression d'un produit
public function delete($id)
{
    $cart = $this->session->get('cart',[]);
    unset($cart[$id]);
    return $this->session->set('cart', $cart);
}
```

## Création de l'entité Adress() et Carrier()

Pour pouvoir recevoir une commande achetée, l'utilisateur doit pouvoir entrer des adresses. Pour cela, on crée une entité adresse avec plusieurs propriétés (nom de l'adresse, user, nom, prénom, compagnie, code postal) on crée une relation ManyToOne avec l'entité User (1 User pour plusieurs adresses).

Avec cette nouvelle entité et en suivant toujours la procédure Symfony, on crée un controller afin de gérer les ajouts d'adresse de l'utilisateur en base de données, l'affichage de celles-ci...

On crée aussi un formulaire lié à cette entité (avec la commande **symfony console make:form**), (**cf annexe 7**). Une fois le formulaire créé, on utilise la méthode `->createView()` pour faire passer le formulaire sur la vue correspondante.

Une fois les adresses en base de données, on utilise l'interface **repository** pour récupérer toutes les adresses liées à l'utilisateur connecté,

Pour afficher les différentes adresses à l'utilisateur dans le template, encore une fois, quand la variable est passée, les données de celle-ci sont accessibles comme dans un tableau. Par exemple, accéder à la ville de l'adresse s'écrira : `{{ address.city }}`.

Enfin si l'utilisateur n'a pas d'adresse, on paramètre une condition d'affichage l'invitant à en ajouter une.

```
{% if app.user.addresses|length == 0 %}
  {# Permet de dissocier les affichages en fonction de si l'utilisateur a des adresses sur son compte #}
  <p class="text-center">
    Vous n'avez pas encore ajouté d'adresses. Pour en ajouter une, veuillez
    <a href="{{path('account_address_add')}}">cliquer ici</a>
  </p>
```

On crée ensuite une entité Carrier() correspondant aux différents transporteurs utilisables, avec les propriétés nom, description et prix.

Celle-ci sera utile pour ajouter des frais de port lors de l'achat.

## Création des entités Order() et OrderDetails().

Ces 2 entités seront développées l'une après l'autre. L'une sera dépendante de l'autre puisque pour avoir le détail d'une commande, il faut une commande.

**Order():** Elle contiendra toutes les informations de base de la commande, la table comprendra :

- L'utilisateur qui la passe
- La date de création de la commande
- Le transporteur
- Le prix du transporteur
- L'adresse de livraison
- Le détail de la commande (OrderDetails)

**OrderDetails:** Elle contiendra :

- La référence de la commande
- Les produits
- La quantité par produit
- Le prix par produit
- Le prix total (quantité x prix)

Pour l'entité **Order**, la liaison se fera avec ManyToOne, car encore une fois, un utilisateur peut faire plusieurs commandes, mais une commande ne peut être reliée qu'à un seul utilisateur.

Pour ce qui est du **transporteur** (carrier), une séparation entre le nom et son prix est faite pour 2viter, si modification il y a, que des incohérences au niveau des commandes soient présentes, c'est à dire qu'une commande archivé et disponible chez l'utilisateur ait un prix de livraison différent de celui qu'il était au moment de la commande, et si le transporteur est supprimé, alors la relation ne sera plus faite, ce qui entraînerait un bug dans l'affichage. On stocke la donnée "en dur" pour qu'elle ne soit pas modifiable.

## Tunnel d'achat : Choix de l'adresse de livraison et du transporteur.

La partie " commande" implique de créer un nouveau controller.

Nous allons nous pencher sur la procédure qui suit le clic sur "valider mon panier".

Première problématique, il faut empêcher les utilisateurs non connectés de pouvoir accéder à cette partie du site sans se connecter et avec un panier vide.

On ajoute dans le **security.yaml** une ligne avec le chemin "commande" et tous les chemins qui découleraient de /commande, qui ne seront accessibles qu'à des utilisateurs connectés, de la même manière que pour l'espace membre.

La page du choix de l'adresse sera un formulaire, récupérant le choix de l'utilisateur lors de sa validation. Mais pour qu'il choisisse, il faut lui proposer les adresses qu'il a renseignées.

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

On crée donc un formulaire (**OrderType**) sans le lier à une entité.

On utilise le repository Interface et on inclut la classe address pour qu'il puisse afficher les adresses qu'il trouve.

Lors de l'affichage sur la vue, une erreur apparaît car Symfony ne sait pas quoi afficher de la table Address, en conséquence une fonction est créée pour guider l'affichage.

```
public function __toString()
{
    return $this->getName();
}
```

Cela nous donne accès à toutes les adresses de la base de données.

Deuxième problématique : afficher les adresses directement liées à l'utilisateur.

On récupère alors la variable \$user dans le controller qu'on transmet au formulaire (orderType).

```
$form = $this->createForm(OrderType::class, null, [
    'user' => $this->getUser() // permet de faire passer en
]);
```

```
$user= $options['user'];
$builder
->add('addresses', EntityType::class, [
    'label' => 'Choisissez votre adresse',
    'required' => true,
    'class' => Address::class,
    'choices' => $user->getAddresses(), //
    'multiple' => false, // empeche le c
    'expanded' => true, // mais affiche
]);
```

Si l'utilisateur n'a aucune adresse ?

On met en place une redirection sur la page de création d'adresses. Une fois l'adresse entrée, il est redirigé vers la page de sélection d'adresse.

```
if (!$this->getUser()->getAddresses()->getValue()) { // co
    return $this->redirectToRoute('account_address_add');
}
```

*(!\$this->getUser->...) signifie : si cela n'existe pas, alors :*

Pour ce qui est du transport, on réutilise le même code dans le formulaire en changeant la classe pour **Carrier**.

La fonction **\_\_toString()** de la classe Carrier regroupe le nom, la description et son prix.

Une fois le choix de ces 2 caractéristiques effectué, il s'agit maintenant de récupérer les données du formulaire et de les stocker dans la base de données.

On fait traiter le formulaire dans une autre route grâce à :

Puis de diriger l'utilisateur vers un récapitulatif de sa commande.

```
{% set formHtml %}
    {#Permet de changer de route pour traiter la requete. #}
    {{form_start(form, {action:path('order_recap')}})}}
```

## Récapitulatif de la commande

Dans le controller, afin de pouvoir fournir le contenu du panier avant validation de toutes les informations, on récupère la classe Cart avec une injection de dépendance et la méthode `get Full()`, qui permet de récupérer les produits et leur quantité, qu'on transmet à la vue pour l'affichage.

C'est un formulaire pré-rempli qui permet à l'utilisateur de valider l'entièreté de la commande.

Dans le controller, on ajoute la fonction `add()` qui correspond au récapitulatif de commande, ( cf Annexe 8)

Cette fonction permet d'ajouter le contenu de la commande dans la base de données, la décision de l'enregistrer avant même l'achat a été prise dans l'optique où, si le client change d'avis, pour une raison quelconque, le propriétaire du site peut voir quels sont les produits les plus sélectionnés, les transporteur favoris etc.

Cette fonction, après avoir soumis le formulaire récapitulatif, récupère la requête avec la méthode **`handleRequest($request->le formulaire)`** de Symfony. On vérifie avec **`isSubmitted()`** et **`isValid()`** que le formulaire est conforme et on commence à décomposer les éléments de la requête afin de les injecter en Base de Données.

- On ajoute une date à la commande
- On récupère le nom du transporteur, l'adresse
- Le nom et prénom du client
- Les détails de l'adresse
- S'il a entré le nom d'une entreprise, on le récupère
- On paramètre l'état de la commande à 0 comme "en attente"

Une fois les informations récupérées, on paramètre la commande avec les **setter** de l'objet **Order**.

- On définit une référence à la commande en joignant la date avec la fonction **`uniqid()`**
- On paramètre chaque propriété de Order avec les données correspondant
- On fige les données avec la fonction **`persist()`**

On effectue ensuite la même manipulation avec les détails de la commande ( `OrderDetails` ).

On envoie tout en Base de Données avec la fonction **`flush()`**, et on envoie des variables au template pour le faire passer au paiement en ligne.

## Le paiement.

Pour le paiement en ligne, on utilise Stripe, cela nous permet de déléguer. Cela étant, des manipulations sont tout de même requises.

Un compte doit être créé, ce qui donne accès aux clés API ainsi qu'à un tableau de bord avec un récapitulatif des versements reçus et en attente.

Pour l'installation, on utilise la documentation de Stripe pour se guider (<https://stripe.com/docs/checkout/quickstart>)

On rentre la commande : `_composer require stripe/stripe-php`

## INTEGRATION

Un code tout prêt est fourni par Stripe, on l'ajoute dans un contrôleur prévu pour rester propre.

On change **Your domain** par le domaine du site. Les clés sont préremplies.

On ajoute dans les différentes colonnes du tableau les renseignements nécessaires à Stripe pour qu'il récupère les informations.

Une fois le paiement effectué, l'application redirige vers la page souhaitée, la variable `$yourdomain` est configurable.

Un script doit être ajouté dans `base.html.twig`.

Une fois que le paiement est effectué, qu'il soit validé ou non, on emmènera l'utilisateur sur une vue personnalisée avec les informations de sa commande.

Si le paiement est validé, le statut `IsPaid` de la commande est passé à 1, ce qui indique qu'elle est payée, on en profite pour vider le panier de l'utilisateur. (cf annexe 9)

```
        $product_for_stripe
    ],
    'payment_method_types' => [
        'card',
    ],
    'mode' => 'payment',
    'success_url' => $YOUR_DOMAIN . '/success.html',
    'cancel_url' => $YOUR_DOMAIN . '/cancel.html',
]);

$response = new JsonResponse(['id' => $checkout_session->id]);

return $response;

// return $this->redirect($checkout_session->url);
```

```
// Modification du statut isPaid de
if (!$order->getIsPaid()) { // si e
    $cart->remove(); // on v
    $order->setIsPaid(1); // on la f
    $this->entityManager->flush(); /
}
```

## Affichage des commandes dans l'espace membre.

```
#[Route('/compte/mes-commande', name: 'account_order')]
```

Avec la création d'un controller spécial, on ajoute une nouvelle route afin d'arriver sur une vue qui récapitule les commandes passées par l'utilisateur.

Pour récupérer les commandes payées de l'utilisateur, on crée une requête SQL personnalisée dans le OrderRepository :

```
public function findByIsPaid($user)
{
    return $this->createQueryBuilder('o') // o est l'alias de order
        ->andWhere('o.isPaid = 1') // on recupere les commande qui on isPaid a true
        ->andWhere('o.user = :user') // le :user est un flag pour lui dire d'utiliser un parametre pour comparer
        ->setParameter('user', $user) // Le parametre de comparaison pour trouver l'utilisateur.
        ->orderBy('o.id', 'DESC') // on les trie par ID de la plus grande a la plus petite.
        ->getQuery()
        ->getResult();
}
```

On code ensuite un tableau pour afficher les résultats de cette requête dans la vue correspondante.

La possibilité de voir les détails de la commande est ajoutée, avec l'ajout de la route.

```
#[Route('/compte/mes-commande/{reference}', name: 'account_order_show')]
```

Ou bien on recherche la commande passée grâce à la référence de la commande envoyée par l'URL puis transmise en paramètre de la fonction.

## L'envoi des Emails.

L'utilisation de Mailjet est due à la possibilité de modéliser des templates de façon ludique sans être initié au code, avoir une liste/groupe de contacts et potentiellement plus tard à envoyer des newsletters et autres offres.

On crée un compte sur le site, on écrit la ligne de commande qui est dans la documentation du site pour installer la dépendance sur notre site.

```
1 composer require mailjet/mailjet-apiv3-php
```

Un code préparé par la documentation de Mailjet nous permet de paramétrer l'envoi des emails.

Afin de rester dans l'optique de segmenter son code, une class Mail() va être créée à l'intérieur de laquelle nous ajouterons le code fourni par Mailjet.

Dans ce code on peut voir que la fonction send() est créée. Donc, si l'on veut envoyer un mail pendant la navigation sur le site, il suffit de rajouter cette fonction avec les paramètres souhaités et d'ajouter la class mail dans les dépendances et le mail sera envoyé

```
$this->entityManager->persist($user); //Figeage des data de la var user.
$this->entityManager->flush(); // execute la persistence, prends la data et enregistre le formulaire

$mail = new Mail;
$content = " c'est un plaisir de vous compter parmi nos membres ".$user->getFirstName()."<br>Votre compte à bien été créé.";
$mail->send($user->getEmail(),$user->getFirstName(),"Bienvenue dans notre projet de boutique",$content);
}
```

*Exemple de code ajouter sur le registerController avec la personnalisation de celui lors de l'inscription d'un nouveau membre.*

## Mot de passe oublié.

Si un utilisateur oublie son mot de passe, nous allons créer une mécanique pour lui permettre de le changer.

On crée une nouvelle entité appelée ResetPassword() avec une relation à la table User (ManyToOne), un token et une date de création.

Dans le controller associé, le but sera de traiter la demande :

On vérifie dans un premier temps que l'email renseigné dans le formulaire de demande existe bien dans la base de données. Si c'est le cas, on crée un nouvel objet resetPassword dans lequel on crée un token unique (avec uniqId()), on lui donne une date et on envoie cela dans la table pour sauvegarder.

Afin d'amener l'utilisateur sur la page de changement de mot de passe, on crée l'URL grâce à la fonction **generateUrl** et en paramètre le token créé lors de la soumission du formulaire.

Dans le même temps, un mail contenant un lien fait de cet URL est envoyé sur la boîte mail. Lorsque l'utilisateur cliquera dessus, une vérification mise en place permettra de s'assurer si le temps imparti pour changer de mot de passe est validé.

```
//verifier si le createdAt est supérieur à now - 30min
$now = new DateTime();
if ($now > $reset_password->getCreatedAt()->modify('+ 3 hour')) {
    $this->addFlash('notice', 'Votre demande de mot de passe a expiré. Merci de la renouveler.');
```

Par la suite, le même procédé mis en place pour changement de mot de passe est mis en place que lors de l'inscription.



## CRÉATION D'UN BACK OFFICE POUR LES ADMINISTRATEURS

L'accès au back office pour les administrateurs doit être paramétré afin de leur laisser la possibilité de modifier certaines parties du site ainsi que d'accéder à la liste des utilisateurs, ajouter des produits, catégories, ou transporteurs entre autres.

La première problématique est de sécuriser l'accès à la partie admin. Pour cela, dans la base de données, on attribue le rôle admin à l'utilisateur, ensuite dans le security.yaml on ajoute dans access\_control:

En plus de cela, dans la vue des espaces membres, une condition est mise en place afin d'afficher un lien pour ceux qui les amènent sur le back office.

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

### ~Mapping des entités avec EasyAdmin.

Le bundle easiAdmin permet de générer un Back-office en fonction de nos entités, Après l'installation avec la ligne de commande:

#### **composer require easycorp/easyadmin-bundle:**

Un dashboardController est ajouté dans le dossier controller/admin, la route /admin est maintenant sur un tableau de bord du BO. Il faut maintenant mapper les entités dessus afin qu'EasyAdmin mette en place un CRUD pour chacun.

*Un CRUD ou Create, Read, Update, Delete est un ensemble de fonctions lié à une table de la Base de Données qui permet de créer, lire, mettre à jour ou effacer un élément de cette table.*

Pour mapper une entité, il suffit d'entrer la ligne de commande :

#### **symfony make:admin:crud**

La console nous interrogera sur l'entité à lier, une fois fait, le crud est mis en place.

Un CRUDController sera créé pour chaque entité,

Il faut ensuite paramétrer l'ajout d'un produit de façon personnalisée, il faut donc développer une fonction dans ce controller,

Par exemple pour l'ajout d'un produit il faut prendre certaines spécificités en compte

Ce formulaire d'ajout permet de configurer l'ajout dans la base de données ainsi que de paramétrer les champs que l'admin pourra remplir lors de l'ajout.

Cette procédure sera à appliquer à chaque ajout d'entité dans EasyAdmin.

Une fois que cela est paramétré, il suffit de faire un lien sur le tableau de bord pour accéder à cette vue à l'aide du code suivant.

## Créer une mécanique de gestion de commande pour informer l'utilisateur.

Pour pouvoir informer les utilisateurs de l'état de leur commande, on ajoute la possibilité dans le BO de modifier l'état : Non-payé, Payé, En cours de préparation, En cours de livraison. On ajoute une propriété à l'entité Order, **State**, modifiable.

On ajoute ensuite de nouveaux onglets cliquables sur la vue détaillée des commandes,

On fait passer le changement d'état en Base de données avec la fonction flush(), et l'utilisateur verra l'état de sa commande lors de son retour sur le site.

On paramètre aussi l'envoi d'un mail lors du changement d'état pour améliorer le service Client.

## POINTS D'AMÉLIORATIONS FUTURES

Afin de pouvoir proposer un site plus compétitif à de potentiels clients, voici les améliorations envisageables pour le site :

- Une fonction de gestion de stock, paramétrable dans le BO, avec une soustraction à chaque commande qui permet d'informer l'utilisateur de la disponibilité de l'article.
- Ajouts de codes promotionnels entraînant une réduction lors de l'achat afin d'améliorer la fidélisation.
- Une version Multilingue du site pour pouvoir exporter sa marchandise et ouvrir son commerce à de nouveaux marchés.
- Ajout d'articles de Blog afin de proposer des ventes par thème en fonction de l'actualité.

## CONCLUSION

Nous arrivons ici à la fin de ce dossier de présentation. Il est le résultat d'un travail personnel long, fastidieux, minutieux mais ô combien satisfaisant.

Ce projet m'a permis d'apprendre à utiliser des API tierces et leur intégration dans un site internet, d'approfondir mes connaissances en PHP et plus particulièrement sur Symfony, qui est devenu est outil agréable et très efficace une fois apprivoisé.

La logique adoptée pour le développement m'a fait rencontrer plusieurs problématiques au fur et à mesure de l'avancée du chantier, mais la plupart ont pu trouver une réponse grâce à la documentation en ligne.

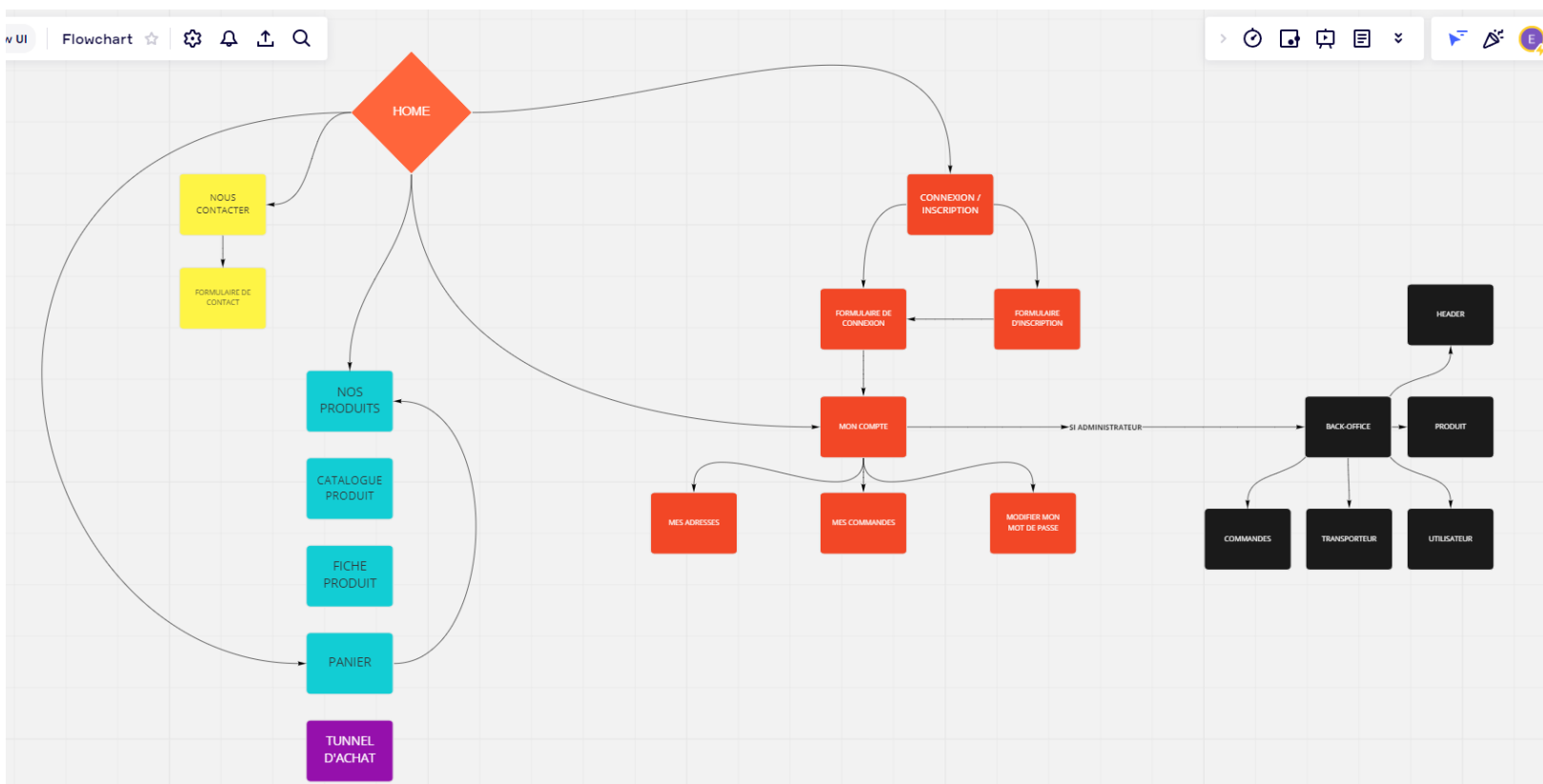
Nous avons donc pu voir pendant ce projet :

- Le maquettage d'un site web
- La création d'une base de données et son développement
- La création Back-End d'un site et le développement de plusieurs fonctionnalités
- Le développement Front-end qui a permis de créer une interface intuitive pour l'utilisateur
- La création d'un Back-office qui permettra aux administrateurs du site une gestion autonome lors de la livraison

## ANNEXE

### ANNEXE 1

Arborescence du site:



Outils utilisés :

- **Miro**:

Créé en 2011, Miro est un logiciel propriétaire en ligne d'outils de travail collaboratif.



## Annexe 2 :

- **HTML5 :**

Le HyperTextMarkup Language, est un langage de balisage qui gère l'affichage sur les page web.

- **CSS :**

De l'anglais Cascading Style Sheets forme un langage informatique qui décrit la présentation des documents HTML.

- **XAMPP :**

XAMPP est un ensemble de logiciels permettant de mettre en place un serveur Web local.



## API Tierce :

- **Mailjet :**

Mailjet est une API qui permet d'envoyer.



- **Stripe :**

Stripe est une solution de paiement qui permet de déléguer la gestion du paiement à un tiers.



## Framework :

- **MySQL :**

MySQL est un système de gestion de base de données relationnelles (SGBDR).



- **Composer :**

Composer est un logiciel gestionnaire de dépendances libres écrit en PHP. Il permet à ses utilisateurs de déclarer et d'installer les bibliothèques dont le projet principal a besoin.



### - **Bootstrap :**

Bootstrap est une collection d'outils utiles à la création du design de site et d'applications web.



## Annexe 3:

### - Exemple d'affichage VSCode :

```

OrderController.php - projet_boutique_e_commerce - Visual Studio Code

EXPLORATEUR
PROJET_BOUTIQUE_E_COMMERCE
├── Controller
│   ├── Admin
│   │   ├── CarrierCrudController.php
│   │   ├── CategoryCrudController.php
│   │   ├── DashboardController.php
│   │   ├── HeaderCrudController.php
│   │   ├── OrderCrudController.php
│   │   ├── ProductCrudController.php
│   │   ├── UserCrudController.php
│   │   ├── gllgnore
│   │   ├── AccountAddressController.php
│   │   ├── AccountController.php
│   │   ├── AccountOrderController.php
│   │   ├── AccountPasswordController.php
│   │   ├── CartController.php
│   │   ├── ContactController.php
│   │   ├── HomeController.php
│   │   ├── OrderCancelController.php
│   │   └── OrderController.php
│   ├── OrderSuccessController.php
│   ├── ProductController.php
│   ├── RegisterController.php
│   └── ResetPasswordController.php
├── src
│   ├── Controller
│   │   └── OrderController.php
├── ...
└── ...

OrderController.php
...
});

$form->handleRequest($request); // on recupere la requete (formulaire)

if ($form->isSubmitted() && $form->isValid()) { // si il est envoyé et valide

    $date = new DateTimeImmutable();
    $carriers = $form->get('carriers')->getData(); // on recupere le nom du transporteur
    $delivery = $form->get('addresses')->getData(); // on recupere l'adresse choisi par l'utilisateur.
    $delivery_content = $delivery->getFirstname(). " " . $delivery->getLastName(); // on explode et recupere en morceau les données de Adresses.
    $delivery_content .= "<br/>" . $delivery->getAddress();

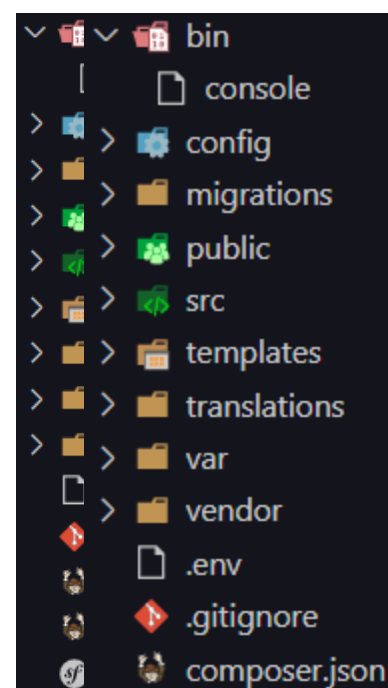
    if ($delivery->getCompany()) {
        $delivery_content .= "<br/>" . $delivery->getCompany();
    }

    $delivery_content .= "<br/>" . $delivery->getPostal() . " " . $delivery->getCity();
    $delivery_content .= "<br/>" . $delivery->getCountry();

    //Enregistrer ma commande : order()
    $order = new Order(); //création d'un objet Order
    $reference = $date->format('d/m/Y') . " - " . uniqid(); // on crée un numero de reference unique avec la date et un chiffre random
    $order->setReference($reference); // La reference de la commande devient ce numero unique.
    $order->setUser($this->getUser()); // on set le user de la commande par l'utilisateur actuel
    $order->setCreatedAt($date); // on reprends la date du jour grace a DateTime.
    $order->setCarrierName($carriers->getname()); // on parametre le nom du transporteur grace a la donnée recuperé
    $order->setCarrierPrice($carriers->getPrice()); // ainsi que son prix.
    $order->setDelivery($delivery_content); // Recuperation des informations de livraisons.
    $order->setState(0);
}

```

Exemple de structure créée par l'installation d'un nouveau projet Symfony :



## Annexe 4:

Controller crée par la commande **symfony console make:form**

```
<?php

namespace App\Controller;

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

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

LA ROUTE CORRESPONDANTE A LA PAGE

LA VUE SUR LAQUELLE EST ENVOYE LES INFORMATIONS

Utilisation du block Content :

Dans le fichier base.html.twig

```
<div class="container marketing {% if block %}
    {% block content %}
    {% endblock %}
</div><!-- /.container -->

<!-- FOOTER -->
<footer class="footer-custom">
```

APPEL DU BLOC CONTENT

Dans le fichier home.html.twig

```
{% block content %}
<div class="row">
    <div class="col-md-5">
        
    </div>
    <div class="col-md-7 my-auto">
        <h3>{{product.name}}</h3>
        <p>{{product.subtitle}}</p>
        <span class="product-page-price">
            {{product.price}}
        </span>
        <p>{{product.description}}</p>
        <a href="{{path('add_to_cart', {'id': product.id})}}">
            Ajouter au panier
        </a>
    </div>
</div>
{% endblock %}
```

APPEL DU BLOC CONTENT

CONTENU DANS LE BLOC

## Annexe 5

Exemple de Repository :

```
namespace App\Repository;

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

/**
 * @method Order|null find($id, $lockMode = null, $lockVersion = null)
 * @method Order|null findOneBy(array $criteria, array $orderBy = null)
 * @method Order[]    findAll()
 * @method Order[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
```

EXEMPLE DE REQUETE SQL  
GENERE PAR SYMFONY

Formulaire d'inscription :

Projet Boutique E-Commerce.
 [Nos produits](#)
[Qui sommes-nous ?](#)
[Contact](#)

[Connexion](#)
[Inscription](#)

### Inscription

Prénom

Nom

Email

Mot de passe

Confirmer votre Mot de passe

[S'inscrire](#)

© 2022 Projet Boutique E-Commerce,  
Boutique moderne et facile d'utilisation  
[Privacy](#) - [Terms](#)

Exemple de recherche avec les filtres :

### Filtrer

☒ Manteaux

☐ Bonnets

☐ T-shirts

☐ Echarpes

[Filtrer](#)

### Nos produits



MANTEAU D'HIVER FEMME  
Parfait manteau pour fêter Noël

65,00 €



MANTEAUX DE PECHEUR  
Manteau idéal pour aller pêcher

80,00 €



## Annexe 6

### Affichage du panier :

Projet Boutique E-Commerce.

Nos produits

Qui sommes-nous ?

Contact

Mon Compte (edouard) | Déconnexion

### Mon Panier

Retrouvez ici l'ensemble des produits que vous avez ajouté à votre Panier.

Produit	Quantité	Prix	Total
 <div><b>Bonnet Rouge</b> Le bonnet parfait pour l'hiver</div>	→ x 1 +	10,00 €	10,00 €

Nombre de produit : 1  
Total de mon panier : 10,00

Valider mon panier

© 2022 Projet Boutique E-Commerce,  
Boutique moderne et facile d'utilisation  
[Privacy](#) - [Terms](#)

### Formulaire d'ajout d'adresse :

Projet Boutique E-Commerce.

Nos produits

Qui sommes-nous ?

Contact

Mon Compte (edouard) | Déconnexion

### Gérer mes adresses

Retour

A quoi correspond cette adresse ?

CMFP

Votre prénom

Developpeur

Votre nom

Web

Votre société

AFFA

Votre adresse

1 rue de l'informatique

Code postal

01000

ville

NightCity

Votre pays

Îles Cook

Votre numéro de téléphone

0612365478

Valider

© 2022 Projet Boutique E-Commerce,  
Boutique moderne et facile d'utilisation  
[Privacy](#) - [Terms](#)

## Annexe 7

### Affichage des adresses existantes :

Projet Boutique E-Commerce. Nos produits Qui sommes-nous ? Contact

Mon Compte (edouard) | Déconnexion

Mes adresses

Ajouter une adresse

C'est dans cet espace que vous allez pouvoir gerer vos adresses.

Retour

Maison Toulouse

17 CHEMIN DE FAGES,  
31400 - TOULOUSE - FR

Modifier | Supprimer

CMFP

1 rue de l'informatique,  
01000 - NightCity - CK

Modifier | Supprimer

© 2022 Projet Boutique E-Commerce,  
Boutique moderne et facile d'utilisation  
[Privacy](#) · [Terms](#)

### Récapitulatif de la commande :

Projet Boutique E-Commerce. Nos produits Qui sommes-nous ? Contact

Mon Compte (edouard) | Déconnexion

Mon Récapitulatif | Boutique E-Commerce

Une Dernière Vérification !


Mon Adresse de livraison

edouard Abgrall  
17 CHEMIN DE FAGES  
31400 TOULOUSE  
FR

Mon Transporteur

ChronoPost  
Vous êtes livré hier tellement ils sont efficaces.  
**19,90 €**

Ma commande



Bonnet Rouge  
Le bonnet parfait pour l'hiver  
x 1

10,00 €

Total de mon panier : 10,00 €

Livraison : 19,90

Total :29,90 €

Payer 29,90 €

© 2022 Projet Boutique E-Commerce,  
Boutique moderne et facile d'utilisation  
[Privacy](#) · [Terms](#)

## Annexe 8

```

#[Route('/commande/recapitulatif', name: 'order_recap', methods:["POST","GET"])]
public function add( Cart $cart, Request $request) // injection de dependance de cart pour acceder au panier
                                                    // et de Requete pour recuperer Les info du formulaire.
{
    $form = $this->createForm(OrderType::class, null, [
        'user'=>$this->getUser() // permet de faire passer en option l'utilisateur
    ]);

    $form->handleRequest($request); // on recupere la requete (formulaire)

    if ($form->isSubmitted() && $form->isValid()) { // si il est envoyé et valide

        $date= new DateTimeImmutable();
        $carriers= $form->get('carriers')->getData(); // on recupere le nom du transporteur
        $delivery= $form->get('addresses')->getData(); // on recupere l'adresse choisi par l'utilisateur.
        $delivery_content= $delivery->getFirstname().' '.$delivery->getLastName();// on exploite et recupere en morceau Les données de Adresses.
        $delivery_content.='<br/>'.$delivery->getAddress();

        if ($delivery->getCompany()) {
            $delivery_content.='<br/>'.$delivery->getCompany();
        }

        $delivery_content.='<br/>'.$delivery->getPostal().' '.$delivery->getCity();
        $delivery_content.='<br/>'.$delivery->getCountry();

        //Enregistrer ma commande : order()
        $order = new Order(); //création d'un objet Order
        $reference = $date->format('dmY') . '-' . uniqid(); // on crée un numero de reference unique avec la date et un chiffre random
        $order->setReference($reference); // La reference de la commande devient ce numero unique.
        $order->setUser($this->getUser()); // on set le user de la commande par l'utilisateur actuel
        $order->setCreatedAt($date); // on reprends la date du jour grace a dateTime.
        $order->setCarrierName($carriers->getName()); // on parametre le nom du transporteur grace a la donnée recuperé
        $order->setCarrierPrice($carriers->getPrice()); // ainsi que son prix.
        $order->setDelivery($delivery_content); // Recuperation des informations de livraisons.
        $order->setState(0);

        $this->entityManager->persist($order); // on fige la données de la commande

    }

    //enregister mes produits dans OrderDetails
    foreach ($cart->getFull() as $product) { // Pour chaque produit rajouté au panier fait une entrée dans orderDetails
        $orderDetails = new OrderDetails();
        $orderDetails->setMyOrder($order); // on parametre l'entité order dans orderDetails
        $orderDetails->setQuantity($product['quantity']); // le nom du produit
        $orderDetails->setProduct($product['product']->getName()); // la quantité
        $orderDetails->setPrice($product['product']->getPrice()); // le prix
        $orderDetails->setTotal($product['product']->getPrice()*$product['quantity']); // Prix total des produits de la même categories.
        $this->entityManager->persist($orderDetails); // on fige la donnée du details de la commande.

    }

    $this->entityManager->flush();

    return $this->render('order/add.html.twig',[
        'cart'=> $cart->getFull(),
        'carrier' => $carriers,
        'address' => $delivery_content,
        'reference' => $order->getReference(),
    ]);
}

```

Fonction d'ajout de la commande en base de données :

## Annexe 9

### Page de paiement : confirmation

Projet Boutique E-Commerce

Payer Projet BoutiqueEcommerce

29,90 €

Bonnet Rouge 10,00 €

ChemisePort 19,90 €

Payer par carte

E-mail: edouard.abgrail@gmail.com

Informations de la carte

1234 1234 1234 1234

MM / AA CVC

Nom du titulaire de la carte

Pays/région

France

☐ Enregistrer mes informations pour le paiement ultérieur en 1 clic. Regardez plus rapidement sur Projet BoutiqueEcommerce et des milliers d'autres sites.

Payer

### Affichage de la page de

Projet Boutique E-Commerce. Nos produits. Qui sommes-nous ? Contact

Mon Compte edouard | Déconnexion

### Votre commande se prépare a Voyager !

Merci pour votre achat edouard !!

La Commande n°14022023 - 630a8a3776065 sera bientôt entre vos mains !  
Vous venez de recevoir une confirmation par email à l'adresse suivante: edouard.abgrail@gmail.com

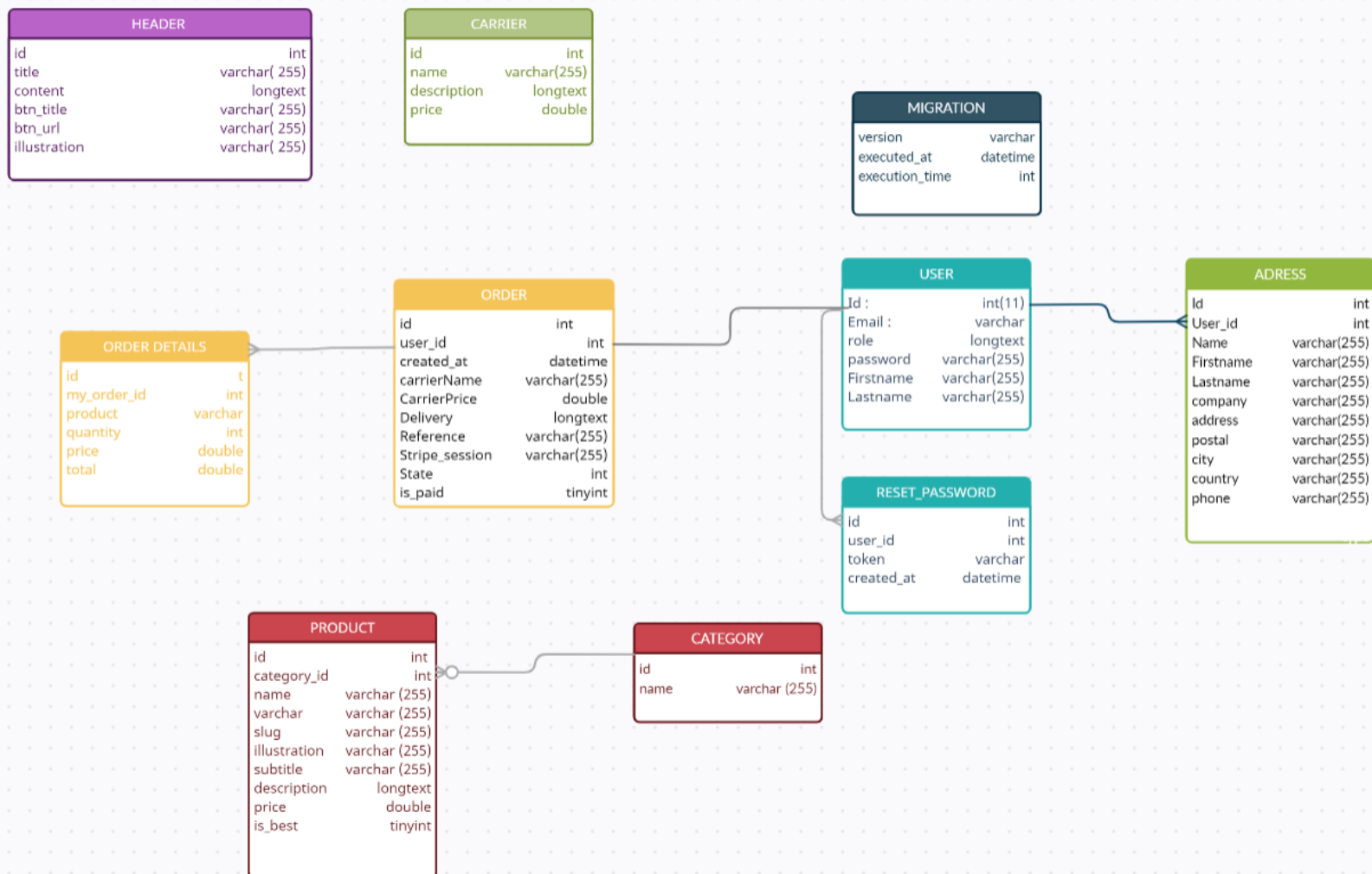
Votre Commande est prise en charge par Colissimo pour arriver a l'adresse suivante :

edouard Abgrail  
17 CHEMIN DE FAGES  
31400 TOULOUSE  
FR

Pour retrouver votre commande rendez vous dans votre [espace membre](#)

© 2022 Projet Boutique E-Commerce.  
Boutique ouverte et toute d'utilisation  
[Privacy](#) [Terms](#)

### Schéma de la base de données :



## Annexe 10

EasyAdmin :

Projet Boutique E Commerce

Dashboard

Utilisateur

Commandes

Catégorie

Products

Carriers

Header

Rechercher

Order

Créer Order

<input type="checkbox"/>	ID ↓	Passée le	Par	Total	Transporteur ↕	Frais de port ↕	Payée ↕	State ↕	
<input type="checkbox"/>	3	14 févr. 2022, 18:15:35	edouard Abgrall	10.00 €	Colissimo	9.90 €	<input checked="" type="checkbox"/>	Payé	...
<input type="checkbox"/>	2	14 févr. 2022, 18:11:37	edouard Abgrall	10.00 €	ChronoPost	19.90 €	<input type="checkbox"/>	Non-Payé	...
<input type="checkbox"/>	1	13 févr. 2022, 11:04:25	edouard Abgrall	20.00 €	Colissimo	9.90 €	<input checked="" type="checkbox"/>	En cours de préparation	...

Projet Boutique E Commerce

Dashboard

Utilisateur

Commandes

Catégorie

Products

Carriers

Header

Rechercher

User

Créer User

<input type="checkbox"/>	ID ↕	Email ↕	Firstname ↕	Lastname ↕	
<input type="checkbox"/>	2	edouard.abgrall@gmail.com	edouard	Abgrall	...
<input type="checkbox"/>	3	john@doe.com	john	doe	...

2 résultats

Précédent

1

Suivant

## Order #3

☒ Livraison en cours☒ Préparation en cours☒ Supprimer

Retour à la liste

Éditer

ID	3
Passée le	14 févr. 2022, 18:15:35
Par	edouard Abgrall
Adresse de livraison	edouard Abgrall 17 CHEMIN DE FAGES 31400 TOULOUSE FR
Total	10,00 €
Transporteur	Colissimo
Frais de port	9,90 €
Payée	<input checked="" type="checkbox"/> OUI
State	Payé
produit acheté	<ul style="list-style-type: none"><li>Bonnet Rouge x 1</li></ul>

