Dossier Projet

Concepteur Développeur d’Application

Session : 2024

Candidat : Saïd Chamchi

Tuteur Pédagogique : Mathieu Capon

Tuteur Technique : Jean-François Beachler



Table des matières

[Introduction………………………………………………………….4](#_Toc160421774)

[Création du Projet…………………………………………………...6](#_Toc160421775)

[Conception du Mode……………………………………………….16](#_Toc160421776)

[Création de nos entités……………………………………………..25](#_Toc160421777)

Création de la Base de Données……………………………………29

Création de notre interface d'administration……………………….31

Création des Views de l'application………...……………………...36

Créations des URLS……………………………………………….46

Sérialisation des données………………………………………….48

Tests de l'API………………………………………………………50

Création de l'application Front…………………………………….58

Les Composent de notre application………………………………65

Annexe…………………………………………………………….82

**Web Shop Application**

During my training as a web developer, I had the opportunity to do a work placement with InfoJef. This Mulhouse-based company specialises in IT hardware and servicing.

Already very well established locally and keen to offer its services to a wider audience.

I suggested that they create an a web shop.

The main purpose of the application will be to present the company and the various products and services it offers.

The functionalities were to have a product catalogue visible on the application. But also to be able to create, modify and add new products via an administration panel.

Knowing the objectives of my internship, and with the agreement of my tutor Mr Beachler. We decided to use the Django .

This tools are based on the Python language.

The main advantages of this Framework are its ORM, its user management and its template engine.

The application will consist of two main parts: the front end, which will be based on HTML/CSS and Javascritp, and the JinJa template engine.

The second part will be the back end, which will be developed with Django (controllers, routing, urls, user management and authentication).

The database will be SQLlite.

# Introduction

Depuis le début, des années 2000 avec la démocratisation d’Internet a la naissance du Web2.0 avec les applications complexe et les différents outils (Framework Front-end, Framework Back-end).

Internet est aujourd’hui indispensable et très commun à beaucoup d’usage. Aussi bien pour un usage lambda que pour les Professionnels.

Le Web est devenu un vecteur de communication indispensable pour beaucoup de personnes. Aussi des administrations publiques ou les Professionnels.

Mais aujourd’hui internet est bien plus qu’un simple outil de communication. Il est devenu un outil indispensable pour différentes entités économiques.

En effet internet est bien plus qu’un simple outil de communication, il permet aussi de commercer avec des millions de personnes a travers le monde.

Cet outil formidable a pu engendrer des Acteurs bien connue et surtout indispensable à notre quotidien.

Au travers de mon stage, j’ai pu étudier cet usage principalement.

Effectivement, durant mon stage que j’ai effectué dans l’entreprise « InfoJef ». Je tiens à remercier ici le responsable et le gérant Monsieur Jean-Frédéric Baechler pour son soutien et sa bienveillance.

Durant cette période, mon tuteur de stage m'a confié la mission de développer une application de commerce en ligne « e-commerce ».

L’objectif de cette application était simple mais aussi passionnant.  
Permettre de mettre en ligne le catalogue des différents produits que l’entreprise propose.  
Ainsi que pouvoir créer, modifier, mettre a jour les différents articles.  
(Un CRUD, Create, Read, Update, Delete).  
  
Mais aussi de pouvoir réalisé des transactions électroniques « paiement électronique » via des api tel que PayPal, mais aussi l’API de la banque de l’entreprise.  
  
Suite à nos différents échanges avec Mr Baechler, nous avons pu développer un cahier des charges de ce projet ainsi qu’une charte graphique.  
  
Avec ces différents éléments, j’ai pu réaliser dans un premier temps réalisé une maque de l’application avec Figma.  
  
Après la validation de mon tuteur.  
J’ai commencé à faire les différents diagrammes UML pour structure notre projet. (Use Cases, Diagramme de Classes)  
  
Vous pouvez trouver tous ces éléments en Annexe. (Maquette et les différentes diagrammes)

# Création du Projet

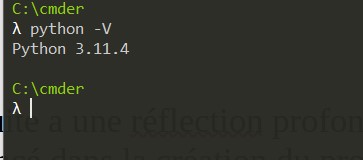
Suite à une réflexion, et la réalisation des digrammes, j’ai pu me lancer dans la création du projet.

Avec les différents diagrammes et suite à une longue réflexion, j’ai choisi d’utiliser le Framework Python Django / Django REST Framework.

Mon choix, c’est porté sur ce Framework pour multiples raisons.

Django et un formidable outil, il dispose d’un excellent ORM (Mapping objet-relationnel), une suite impressionnante de modules disponible grâce à l’outil PIP (pip est le gestionnaire de paquets). Qui me permettra de développé facilement ce projet.

Avant de pouvoir installer Django ou même lancé la création de notre application, je vérifie que Python est bien installer sur mon poste avec la commande « Python –V ».



Une fois que j’ai vérifié que Python est bien installer sur mon poste, je vais pouvoir créer mon environnement virtuel.

C’est quoi un environnement virtuel est quel est son utilité ?

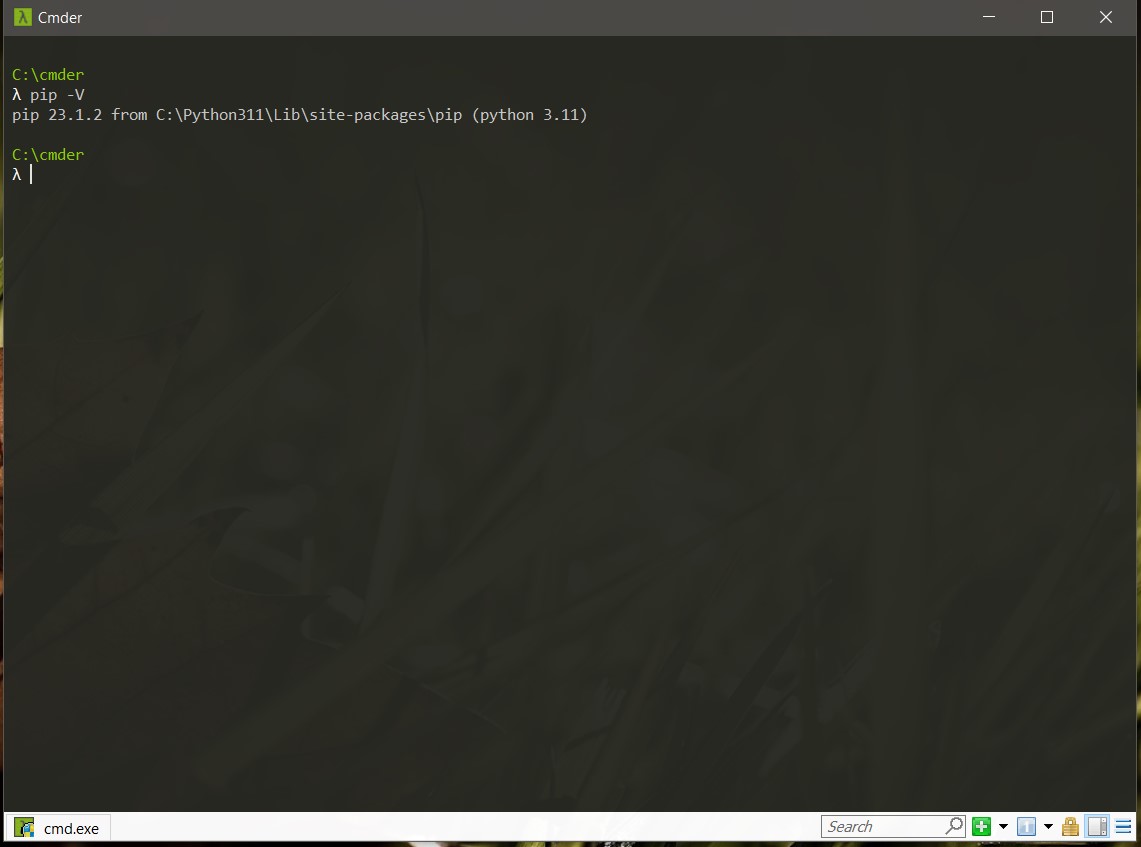
1 - Un environnement virtuel permet de créer un environnement complètement isolé du système

Ça permet aussi d’isoler toutes les dépendances du projet dans l’environnement virtuel propre à l’application

Cela évite de polluer son système d’exploitation. Ainsi qu’éviter les différents soucis que l’on peut avoir avec des versions de modules non-compatibles entre eux.

Après ces différentes vérifications, je décide de me lancer.

Dans un premier temps, je vérifie que la commande pip fonctionne bien.



Comme vous en doutez certainement PIP est le gestionnaire de paquets de Python, comme npm pour JavaScript ou composer pour PHP.

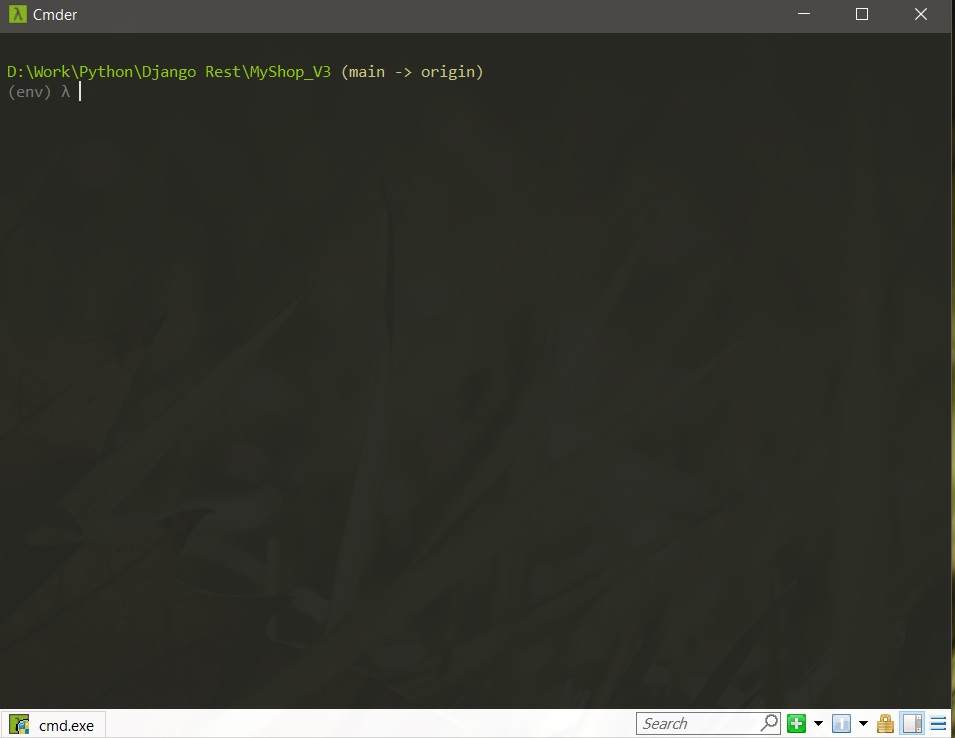
Après ces différentes vérifications, je créais mon environnement virtuel avec la commande « python -m venv env ».

Cette commande va me créer un dossier « env », ou sera isolé toutes les dépendances du projet.

Maintenant, que l’environnement est installé, je vais l’activer et installer les différents paquets indispensables au Projet.

- Pour activer l’environnement virtuel, j’utilise la commande.

« env/script/activate ».



- Comme on peut le constater dans l’image, suite à la commande « env/script/activate », l’environnement est bien actif. « (env) »

-Maintenant il est possible d’installer les différentes dépendances nécessaires au projet.

Dans un premier temps, je commence par installer « Pillow ».

- « Pillow » est une dépendance indispensable pour gère les images dans l’application. Ce module est nécessaire dans la majorité des applications Django.

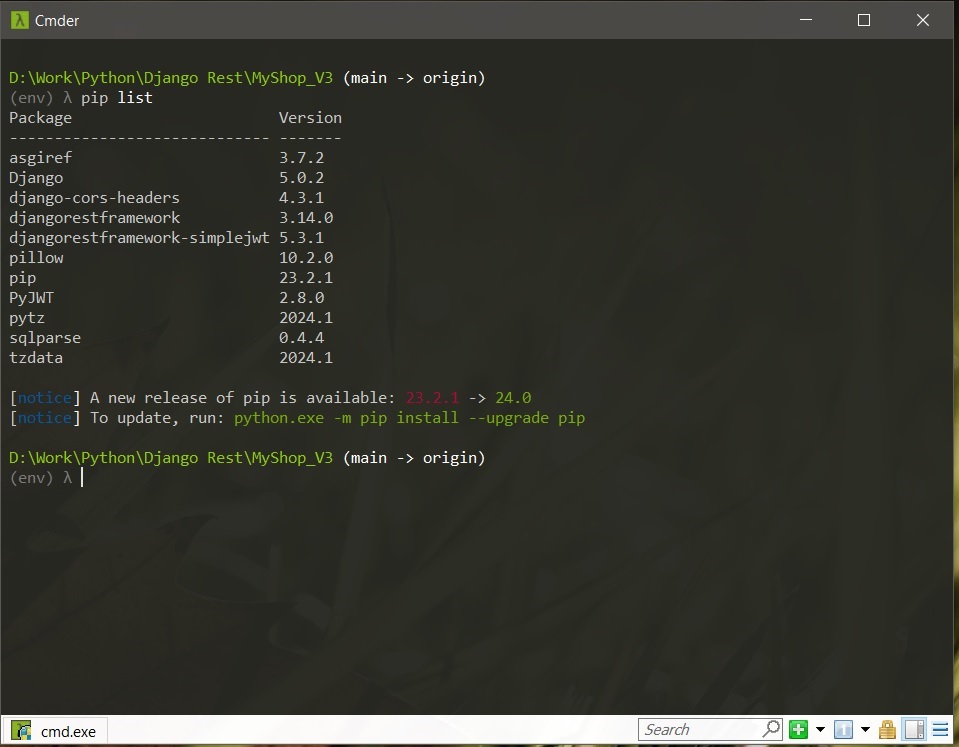
- Apres l’installation de « Pillow » j’installe Django et Django Rest Framework. Il est important de remarque que Django et Django Rest Framework sont de dépendances différentes. Pour ce faire je vais utiliser la commande «pip install Django » et « pip install djangorestframework ». Dans un second temps j’installe aussi simple JWT, avec la commande « pip install djangorestframework-simplejwt ». Une dépendance qui nous permettra de génère facilement un Token JWT (Json Web Token), pour sécuriser notre application.

- Afin de gère les requetés externes a l’url de l’application, je dois installer django-cors-headers.

Pour vérifier que ces dernières dépendances sont bien installées, je vais utiliser la commande « pip list » afin de m’assurer que tous les modules, c'est bien installer.

Comme on peut le voir dans l’image ci-dessous. On voit que tous les modules sont bien installés.

Après avoir vérifié que les dépendances sont installées, je vais pouvoir faire un pip freeze > requiement.txt.



Cette commande me permet d’inscrire toutes les dépendances du projet dans un fichier texte « requiement.txt » automatiquement. Mais aussi me permet d’installer toutes les dépendances avec une seul commande « pip install requiement ». Un peu comme un npm install.

Maintenant, que toutes les vérifications sont faites.

Je peux lancer l’installation du projet en utilisant la commande « django-admin startproject MyShop .».

Une fois la commande lancée cela m’a généré la structure du projet.

Dans le jargon des développeurs Django « un projet » englobe l’application et contient toutes les dépendances au bon fonctionne de notre future application.

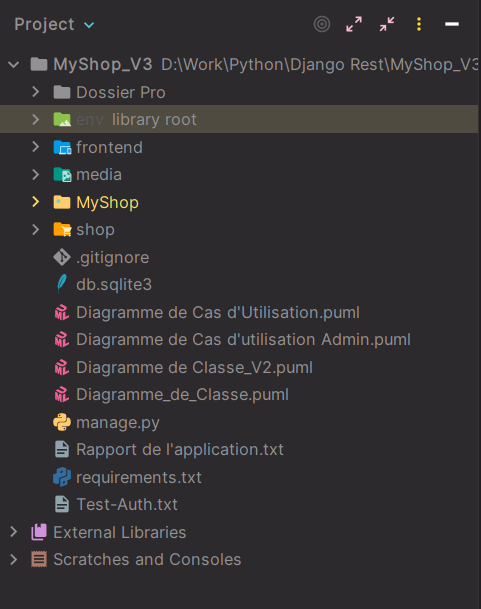
A noter que Django peut gérer une multitude d’applications dans « un Seul Projet »

Une fois que le projet et créer nous pouvons nous atteler à la création de l’application, en utilisant la commande « python manage.py startapp "Nom de l'application" ».

Dans notre cas d'application se nommera « shop ».

Structure du Projet

Comme on peut le constater dans la capture d’écran ci-dessous, on peut voir la structure du projet.



Ici, le nom du projet est MyShop « L’ensemble de l’application » et l’application s’appelle « shop ».

Le projet MyShop est composé de plusieurs fichiers. Les deux fichiers les plus importants ici sont « setting », « views » et « url ».

« Setting » et le fichier de configuration du projet. Il contient toutes les informations relatives à l’application, comme le listing de l’ensemble des applications du « projet » mais aussi les « Middlewars », la configuration de la base de données et aussi le « Password Hasher », le « langage code » ainsi que la gestion des origines des requêtes autorisées et etc.

Le fichier « views.py » est un des fichiers les plus important, car il contient tous les contrôleurs de l’application.

Le fichier « url.py » contient comme son nom l’indique l’ensemble des urls du projet.

Maintenant, regardons de plus près notre application « shop ». Comme dit précédemment ce répertoire est le cœur de notre projet.

C’est bien ici que nous allons coder notre application.

Le répertoire est lui-même composer de plusieurs répertoires et de fichiers.

Il compte deux répertoires, qui sont « migrations » qui contient toutes les migrations des entités créer dans la base de données. La migration est un peu comme le plan de notre structure de notre base de données.

Nous trouvons un répertoire « tests » qui contient les tests de l’application. (Nous développerons plus tard cette partie).

Au sein de notre répertoire shop, nous trouvons aussi des fichiers de type Python :

- Le premier est « \_init\_.py », ce fichier sers principalement a indiqué la présence de package ou de fichier python dans le répertoire.

- « admin.py » :

Ce fichier est un peu particulier, il est utilisé pour importer les différentes entités du projet dans la page d’administration du Projet.

En fait Django contient en son sein un panneau d’administration.

Que vous pouvez configurer comme vous le souhaitez via ce fichier.

- « Model » :

Ce fichier est vraiment très important, il contient toutes les entités de notre application. (Produits, Catégories et etc).

- « urls.py » :

Ce document, contient toutes les urls de l’application. Ce fichier est très utile aussi et on va souvent l’utiliser.

- « view.py » :

Le fichier view.py contient tous les contrôleurs et la logique de l’application.

- « manage.py » :

Le fichier manage.py est le point d’entrée de l’application, il contient la méthode main(). Il est indispensable pour lancer l’application !!

Avec la commande « python manage.py runserver ».

- « app.py » :

Le fichier app.py est un fichier de configuration pour une application Django. En résumé, ce fichier de configuration définit les paramètres de l'application shop pour Django, notamment le type de champ utilisé pour les ID automatiques et le nom de l'application.

- « serializers.py » :

Le fichier serializers.py est un fichier de code Python qui est utilisé pour définir les serialisateurs dans une application Django.

Un serialisateur est un objet qui permet de convertir des objets Python en données JSON ou XML, et vice versa. Dans Django, les serialisateurs sont utilisés pour transmettre des données entre l'application et les clients (Dans notre cas, notre application React JS et l’application Mobile React Native).

- « permission.py » :

Dans une application Django REST, le fichier permissions.py est utilisé pour définir les permissions qui sont nécessaires pour accéder aux ressources protégées par des autorisations.

Après avoir vu les Principaux Fichiers de notre application, nous allons commencer à construire notre Futur Application.

# Conception du Model

Après une certaine réflexion sur note future application de Commerce Électronique.

J’ai réalisé un diagramme de Cas d’utilisation « Uses Cases ». Afin de définir les principales fonctionnalités du projet.

Comme le diagramme le montre les principales fonctionnalités de l’application sont :

Pour l’utilisateur :

* L’authentification
* Listing des produits
* La vue détaillée du Produit
* La fonction ajoutée au panier
* La vue détaillée du Panier

Pour L’administrateur :

- L’authentification

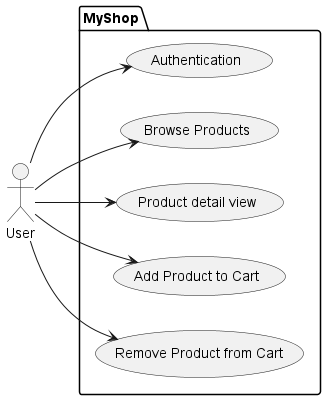
- Connexion

- Interface de gestion

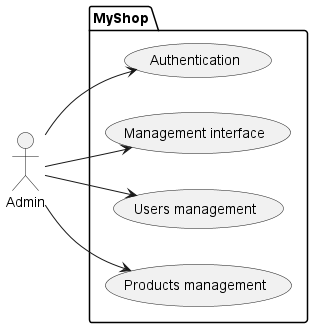
- Gestion des Utilisateurs

- Gestion des Produits

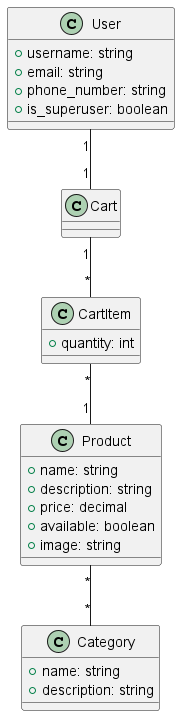
Diagramme « Use Case Utilisateur » :



**Diagramme Use Cases « Administrateur » :**



Une fois que nous avons identifiées les principales fonctionnalités de l’application. J’ai pu créer le diagramme de Classe.



Comme on peut le constater dans le diagramme. Nos Entités principales sont la Classe Produits, Catégorie, Panier, Produit du panier.

* **User** :

Cette classe représente un utilisateur dans le système. Elle a les attributs suivants :

- username : une chaîne de caractères représentant le nom d'utilisateur.

-email : une chaîne de caractères représentant l'adresse e-mail de l'utilisateur.

- phone\_number : une chaîne de caractères représentant le numéro de téléphone de l'utilisateur.

- is\_superuser : un booléen indiquant si l'utilisateur est un superutilisateur ou non.

* **Product** :

Cette classe représente un produit dans l’application. Elle a les attributs suivants :

- name : une chaîne de caractères représentant le nom du produit.

- description : une chaîne de caractères décrivant le produit.

- price : un nombre décimal représentant le prix du produit.

- available : un booléen indiquant si le produit est disponible ou non.

- image : une chaîne de caractères représentant l'URL de l'image du produit.

**Category** :

Cette classe représente une catégorie de produits dans le système. Elle a les attributs suivants :

-name : une chaîne de caractères représentant le nom de la catégorie.

-description : une chaîne de caractères décrivant la catégorie.

**Cart** :

Cette classe représente un panier d'achat dans l’application. Elle n'a pas d'attributs spécifiés.

**CartItem** :

Cette classe représente un élément dans le panier d'achat. Elle a l'attribut suivant :

quantity : un entier représentant la quantité de l'élément dans le panier.

**Voici les relations entre ces classes sont représentées par des lignes et les cardinalités :**

-User : "1" -- "1" Cart : Cela signifie qu'un utilisateur est associé à un seul panier, et un panier est associé à un seul utilisateur. C'est une relation oneToOne.

-Cart "1" -- "\*" CartItem : Cela signifie qu'un panier peut contenir plusieurs articles, mais chaque article appartient à un seul panier. C'est une relation oneToMany.

-CartItem "\*" -- "1" Product : Cela signifie qu'un produit peut être associé à plusieurs éléments du panier (si un utilisateur ajoute le même produit plus d'une fois), mais chaque élément du panier est associé à un seul produit. C'est aussi une relation oneToMany.

Product "\*" -- "\*" Category : Cela signifie qu'un produit peut appartenir à plusieurs catégories, et une catégorie peut contenir plusieurs produits. C'est une relation manyToMany.

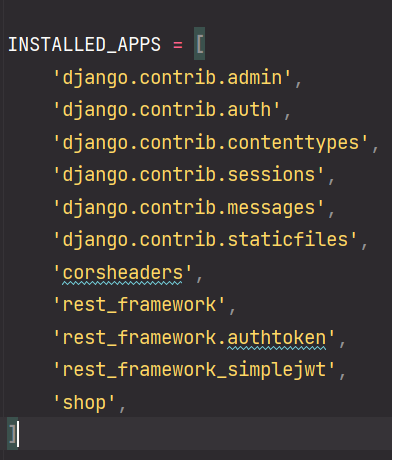
**Configuration du Fichier Setting**

Afin d’assurer le bon fonctionnement de l’application je dois configurer mon fichier setting.py qui englobe l’ensemble de la configuration du projet.

Le fichier setting.py contient différents paramètres de l’application.

1. Import os et from pathlib import Path : Ces lignes importent les modules os et Path de pathlib, qui sont utilisés pour manipuler les chemins de fichiers.
2. INSTALLED\_APPS : Cette liste contient toutes les applications installées dans ton projet Django.

Dans notre cas nous avons ajouté l’application « shop », « rest\_framework », « simple JWT » et « corsheaders ».



Les autres dépendances présentées sont les éléments de base a tout application Django :

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

3. Middleware : Ce sont les middlewares utilisés dans notre projet. Les middlewares dans une application Django sont des couches de logiciel qui s'exécutent entre la requête entrante et la vue, ainsi qu'entre la vue et la réponse sortante. Ils sont utilisés pour effectuer des tâches de pré-traitement et de post-traitement pour chaque requête et réponse. (Authentification, Autorisation, Gestion des sessions, Gestion des CSRF, Journalisation)

4. ROOT\_URLCONF = 'MyShop.urls' : Ce paramètre indique à Django d'utiliser le module urls.py dans le répertoire MyShop pour les URLs du projet.

5. DATABASES : Ce paramètre configure la base de données utilisée par Django. Dans ce cas, une base de données SQLite3 est utilisée.

6. REST\_FRAMEWORK : Ce paramètre configure Django Rest Framework pour utiliser l'authentification JWT.

7. AUTH\_USER\_MODEL = 'shop.User' : Ce paramètre spécifie le modèle User que j’ai utilisé.

8. AUTH\_PASSWORD\_VALIDATORS : Ce paramètre configure les validateurs de mot de passe utilisés par Django.

9. STATIC\_URL : Ce paramètre configure l'URL des fichiers statiques.

10. MEDIA\_ROOT et MEDIA\_URL : Ces paramètres configurent le stockage des images et des fichiers multimédias.

11. DEFAULT\_AUTO\_FIELD : Ce paramètre configure le type de champ de clé primaire par défaut.

12. CORS\_ALLOWED\_ORIGINS et CORS\_ALLOW\_ALL\_ORIGINS : Ces paramètres configurent les Cross-Origin Resource Sharing (CORS), qui permettent aux requêtes d'être envoyées depuis des URLs externes.

Maintenant que nous avons configurer le fichier setting de notre projet.

Nous pouvons passer à la construction de notre application.

# Création de nos entités

Apres avoir créer nos différents diagrammes de notre application. Nous avons une idée très précise des entités et qui seront nécessaire à son bon fonctionnement.

Pour ce faire, je vais utiliser le fichier « Models.py », qui comme dit précédemment qui va contenir les classes qui représentent nos différentes entités.

Dans un premier temp je commence par la classe « User ».

Class User:

Cette classe définit un modèle d'utilisateur personnalisé qui hérite d'AbstractUser. Je vais ajouter les champs suivants :

name : Un champ de type CharField pour le nom de l'utilisateur. Il peut contenir jusqu'à 200 caractères.

email : Un champ de type CharField pour l'adresse e-mail de l'utilisateur. Il peut contenir jusqu'à 100 caractères.

address : Un champ de type CharField pour l'adresse de l'utilisateur. Il peut contenir jusqu'à 200 caractères.

city : Un champ de type CharField pour la ville de l'utilisateur. Il peut contenir jusqu'à 100 caractères.

phone\_number : Un champ de type CharField pour le numéro de téléphone de l'utilisateur. Il peut contenir jusqu'à 15 caractères. Il est facultatif (blank=True, null=True).

Class Category:

Cette classe définit un modèle de catégorie. Elle a les champs suivants :

name : Un champ de type CharField pour le nom de la catégorie. Il peut contenir jusqu'à 255 caractères.

description : Un champ de type TextField pour la description de la catégorie. Il est facultatif (blank=True, null=True).

Class Product :

Cette classe définit un modèle de produit. Elle a les champs suivants :

name : Un champ de type CharField pour le nom du produit. Il peut contenir jusqu'à 255 caractères.

description : Un champ de type TextField pour la description du produit.

price : Un champ de type DecimalField pour le prix du produit. Il peut contenir jusqu'à 10 chiffres, avec 2 décimales.

available : Un champ de type BooleanField pour indiquer si le produit est disponible ou non. La valeur par défaut est True.

image : Un champ de type ImageField pour l'image du produit. Les images sont téléchargées dans le répertoire 'product\_images/'. Ce champ est facultatif (blank=True, null=True).

categories : Un champ ManyToManyField pour les catégories du produit. Il utilise le paramètre related\_name='products' pour spécifier le nom de la relation inverse.

**Class Cart**:

Cette classe définit un modèle de panier. Elle a les champs suivants :

user : Un champ OneToOneField pour l'utilisateur associé au panier. Lorsqu'un utilisateur est supprimé, son panier est également supprimé (on\_delete=models.CASCADE).

products : Un champ ManyToManyField pour les produits dans le panier. Il utilise le modèle CartItem comme modèle intermédiaire (through='CartItem').

**Class CartItem**:

Cette classe définit un modèle d'élément de panier. Elle a les champs suivants :

cart : Un champ ForeignKey pour le panier associé à l'élément. Lorsqu'un panier est supprimé, tous ses éléments sont également supprimés (on\_delete=models.CASCADE).

product : Un champ ForeignKey pour le produit associé à l'élément. Lorsqu'un produit est supprimé, tous ses éléments de panier sont également supprimés (on\_delete=models.CASCADE).

quantity : Un champ IntegerField pour la quantité du produit dans l'élément. La valeur par défaut est 1.

Chaque entité du modèle a également une méthode \_\_str\_\_ qui retourne une chaîne de caractères représentant l'instance du modèle. Cette méthode est utilisée pour afficher les instances du modèle dans l'interface d'administration de Django.

**Création de la Base de Données**

Après avoir créé mes entités dans le model, je vais pouvoir créer ma base de données en effectuent une migration.

Django nous facilite encore une fois la vie, il dispose de tous les outils nécessaires pour effectuer cette opération délicate.

Pour effectuer cette délicate opération, je vais utiliser la commande.

**« python manage.py makemigrations »**

Cette commande va créer les fichiers de migrations avec les différentes tables et les relations entre elles.

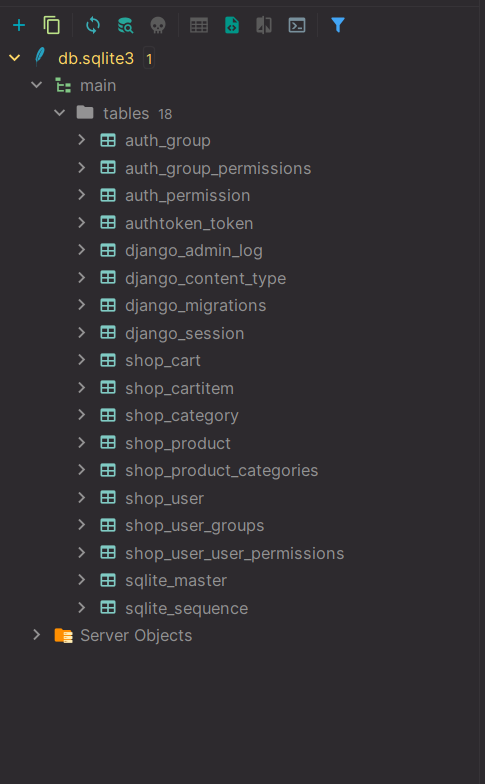
Après cela je vais pouvoir créer les tables de dans ma base de données, avec la commande :

**« python manage.py migrate »**

Maintenant que ma base de données et créer et prêt, il faut noter que « Django », nous crées aussi d’autre tables, pas uniquement les Tables « Category & Product », (shop\_category, shop product et etc).

Le Framework nous a créer toutes les tables nécessaires au bon Fonctionnement de notre application. (authe\_group, auth\_group\_permission, auth\_user, auth\_user\_groups, auth\_user\_user\_permission, django\_admin\_log, django\_content\_type, django\_migrations, django\_session).

Ces Tables concerne les futurs utilisateurs et les différentes permissions.



Vous pouvez remarquerai que j’utilise une base de données de type SQLite. Mais je peux à tout moment changer si nécessaire. Django supporte un grand nombre de Base de Données.

**Création du Super User :**

Pour pouvoir administre notre application, nous avons besoin de créer un « super User » qui aura les droits de créer les différents produits et catégories de notre site de commerce en ligne.

Donc je vais utiliser la commande « python manage.py createsuperuser », j'entre le nom ainsi que le mot de passe de mon super User.

**Configuration de la Page d’administration :**

Comme stipulé précédemment ce fichier a pour bus principal de configure l’interface d’administration de Django.

Pour ce faire il a besoin de plusieurs éléments.

1. Dans un premier temps je dois importer le module admin de Django : from django.contrib import admin
2. Je dois aussi importer la classe UserAdmin de Django qui est utilisée comme base pour créer une classe d'administration personnalisée pour le modèle d'utilisateur.
3. J’importe aussi la fonction get\_user\_model de Django, qui est utilisée pour obtenir le modèle d'utilisateur actuel.
4. User = get\_user\_model() : Cette ligne obtient le modèle d'utilisateur actuel et le stocke dans la variable User.
5. Class UserAdmin(BaseUserAdmin) : Cette classe définit une classe d'administration personnalisée pour le modèle d'utilisateur. Elle hérite de BaseUserAdmin et définit les champs à afficher, les champs de recherche, les champs en lecture seule, etc.
6. Class ProductAdmin(admin.ModelAdmin) : Cette classe définit une classe d'administration pour le modèle Product. Elle définit les champs à afficher, les champs de recherche, etc.
7. Class CategoryAdmin(admin.ModelAdmin) : Cette classe définit une classe d'administration pour le modèle Category. Elle définit les champs à afficher, les champs de recherche, etc.
8. Class CartAdmin(admin.ModelAdmin) : Cette classe définit une classe d'administration pour le modèle Cart. Elle définit les champs à afficher, les champs de recherche, etc. Elle définit également deux méthodes personnalisées : get\_items\_count et get\_total\_price, qui sont utilisées pour afficher le nombre d'articles et le prix total du panier, respectivement.
9. Class CartItemAdmin(admin.ModelAdmin) : Cette classe définit une classe d'administration pour le modèle CartItem. Elle définit les champs à afficher, les champs de recherche, etc.

admin.site.register(User, UserAdmin) :

Cette ligne enregistre le modèle User avec la classe d'administration UserAdmin dans l'interface d'administration.

admin.site.register(Product, ProductAdmin) :

Cette ligne enregistre le modèle Product avec la classe d'administration ProductAdmin dans l'interface d'administration.

admin.site.register(Category, CategoryAdmin) :

Cette ligne enregistre le modèle Category avec la classe d'administration CategoryAdmin dans l'interface d'administration.

admin.site.register(Cart, CartAdmin) :

Cette ligne enregistre le modèle Cart avec la classe d'administration CartAdmin dans l'interface d'administration.

admin.site.register(CartItem, CartItemAdmin):

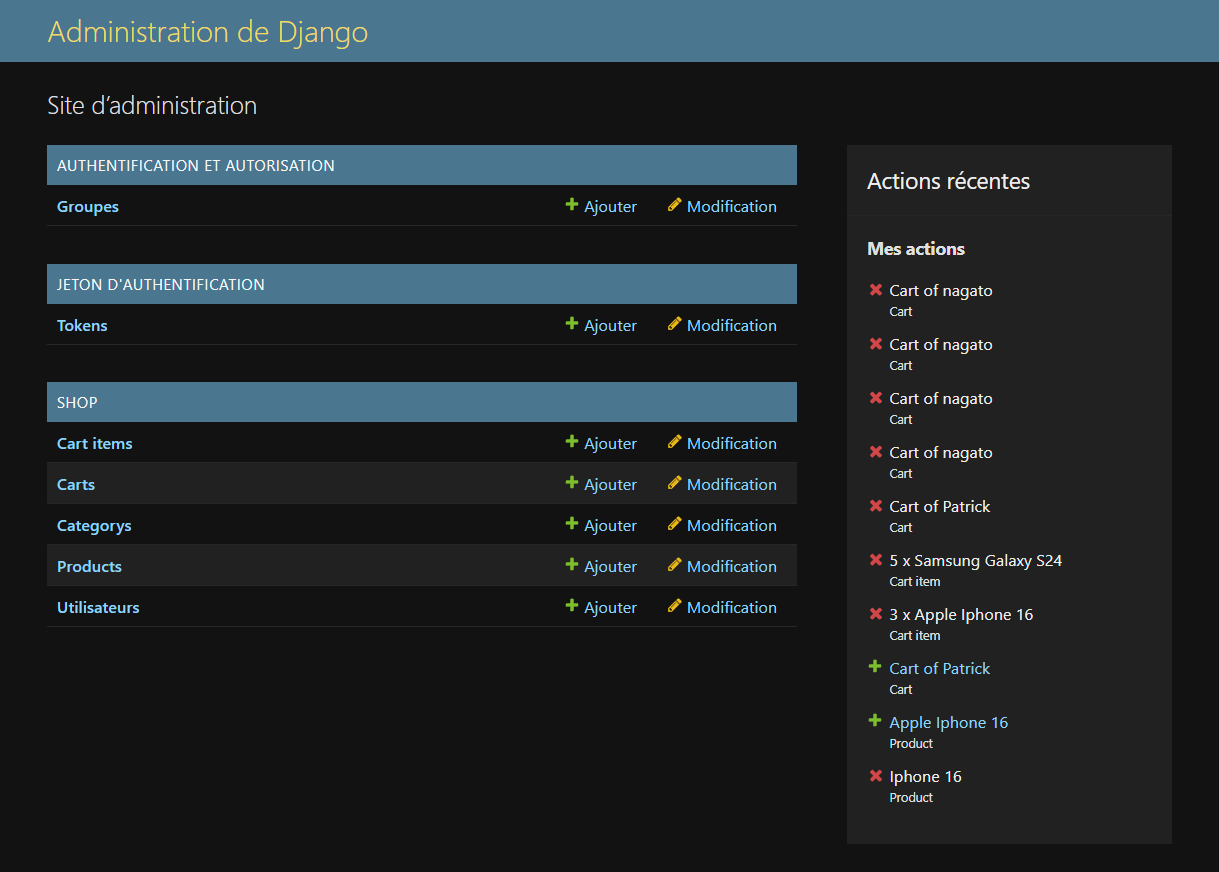
Cette ligne enregistre le modèle CartItem avec la classe d'administration CartItemAdmin dans l'interface d'administration.

Chaque classe d'administration a des attributs tels que list\_display, list\_filter, search\_fields, etc., qui sont utilisés pour personnaliser l'affichage et le comportement des modèles dans l'interface d'administration. Par exemple, list\_display spécifie les champs à afficher dans la liste des objets du modèle, list\_filter spécifie les champs qui peuvent être utilisés pour filtrer les objets, et search\_fields spécifie les champs qui peuvent être utilisés pour rechercher des objets.

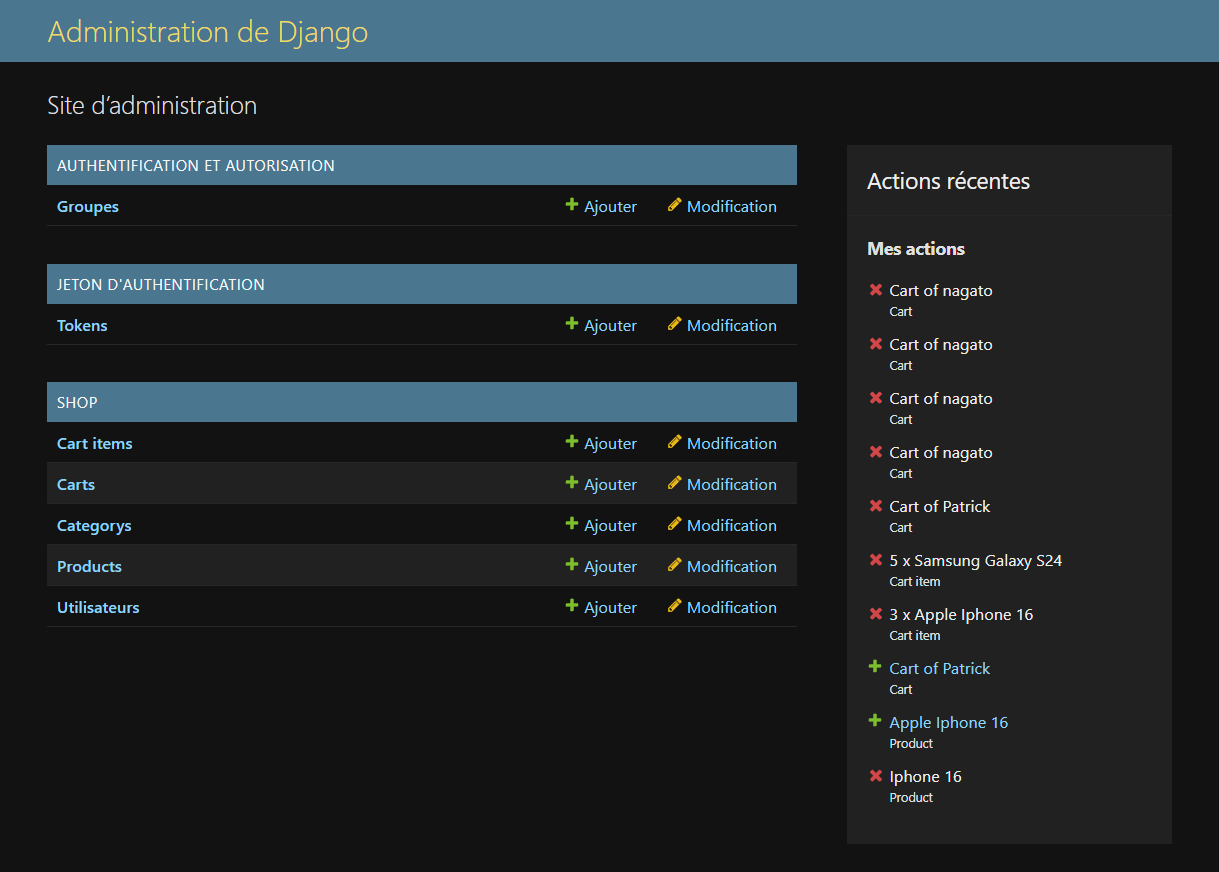


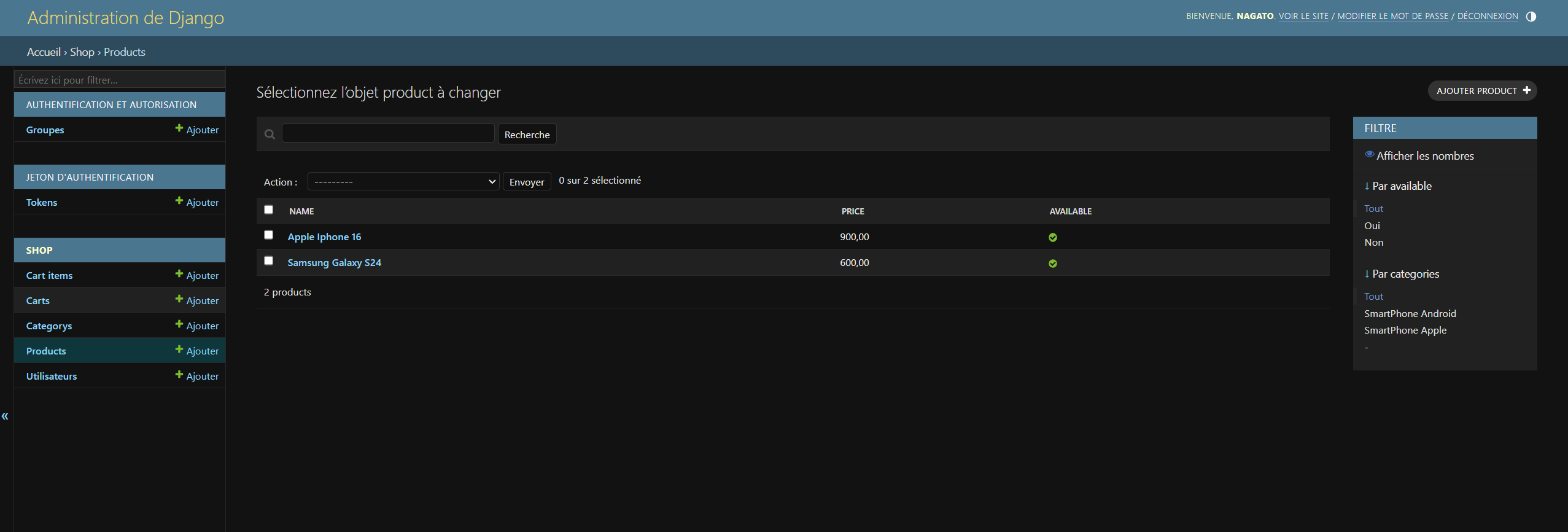
Après cela, j'aurai accès à mes différentes entités directement depuis le panel administrateur.

Panneau D’administration :



Comme on peut le constater cela a pour effet d’avoir une présentation bien plus claire et lisible de nos « Entités ».





**Création des Views de l’application**

Apres la conception de la base de données et la configuration de l’interface d’administration de l’application je vais pour m’attelé à la création des views.

Comme vous le savez surement Django repose sur un pattern MVT.

(Models, Views, Templates) similaire au Design Pattern (MVC)

Comme nous avons traite le model nous devons développer les views (les contrôleurs) du projet.

Pour ce faire je vais pouvoir utiliser le fichier views.py.

Pour créer des contrôleurs dans Django vous avez la possibilité d’utilisé des classes ou des fonctions avec le décorateur « @apiview ».

Dans notre cas je vais utiliser des fonctions. C’est un simple choix personnel.

Comme dans les précédents fichiers, nous avons un ensemble d’imports.

**- from rest\_framework.decorators import api\_view, permission\_classes:**

Ces lignes importent les décorateurs api\_view et permission\_classes de Django Rest Framework. api\_view est utilisé pour définir les méthodes HTTP autorisées pour une vue, et permission\_classes est utilisé pour spécifier les permissions requises pour accéder à une vue.

**- from django.shortcuts import get\_object\_or\_404 :**

Cette ligne importe la fonction get\_object\_or\_404 de Django, qui est utilisée pour récupérer un objet du modèle ou renvoyer une erreur HTTP 404 si l'objet n'existe pas.

**- from rest\_framework.permissions import IsAuthenticated :**

Cette ligne importe la classe IsAuthenticated de Django Rest Framework, qui est utilisée pour vérifier si un utilisateur est authentifié avant d'accéder à une vue.

**- from rest\_framework.response import Response :**

Cette ligne importe la classe Response de Django Rest Framework, qui est utilisée pour renvoyer une réponse HTTP.

**- from .models import Product, Category, Cart, CartItem, User :**

Cette ligne importe les modèles de mon application.

**- from .serializers import ProductSerializer, CategorySerializer, UserSerializer, CartSerializer, CartItemSerializer :**

Cette ligne importe les sérialiseurs de ton application, qui sont utilisés pour convertir les objets du modèle en JSON et vice versa.

**- from rest\_framework import status :**

Cette ligne importe les codes de statut HTTP de Django Rest Framework.

**- from .permissions import IsSuperUser :**

Cette ligne importe la permission personnalisée IsSuperUser de ton application, qui est utilisée pour vérifier si un utilisateur est un superutilisateur.

**- import logging** :

Cette ligne importe le module logging de Python, qui est utilisé pour la journalisation.

Maintenant que nous avons fait le tour de l’ensemble de nos imports . Nous pouvons coder nos contrôleurs

1. **Controlleur user\_details :**

Cette view gère les requêtes GET, PUT et DELETE pour un utilisateur spécifique. Elle prend un paramètre id qui est l'identifiant de l'utilisateur dans la base de données.

Si la requête est de type GET, elle récupère l'utilisateur correspondant à l'identifiant donné à l'aide de la fonction get\_object\_or\_404 qui renvoie une réponse HTTP 404 si l'utilisateur n'existe pas. Ensuite, elle sérialise l'utilisateur en JSON à l'aide du sérialiseur UserSerializer et renvoie une réponse HTTP 200 avec les données de l'utilisateur.

Si la requête est de type PUT, elle récupère également l'utilisateur correspondant à l'identifiant donné. Ensuite, elle essaie de mettre à jour les données de l'utilisateur avec les données envoyées dans la requête en utilisant le sérialiseur UserSerializer. Si les données sont valides, elle enregistre l'utilisateur mis à jour dans la base de données et renvoie une réponse HTTP 200 avec les données de l'utilisateur. Sinon, elle renvoie une réponse HTTP 400 avec les erreurs de validation.

Si la requête est de type DELETE, elle récupère l'utilisateur correspondant à l'identifiant donné et le supprime de la base de données. Ensuite, elle renvoie une réponse HTTP 204 sans contenu.

1. **Contrôleur « user\_list » :**

Cette fonction gère les requêtes GET et POST pour la liste des utilisateurs.

Si la requête est de type GET, elle récupère tous les utilisateurs de la base de données à l'aide du modèle User et les sérialise en JSON à l'aide du sérialiseur UserSerializer. Ensuite, elle renvoie une réponse HTTP 200 avec la liste des utilisateurs en JSON.

Si la requête est de type POST, elle essaie de créer un nouvel utilisateur avec les données envoyées dans la requête en utilisant le sérialiseur UserSerializer. Si les données sont valides, elle enregistre le nouvel utilisateur dans la base de données et renvoie une réponse HTTP 201 avec les données de l'utilisateur. Sinon, elle renvoie une réponse HTTP 400 avec les erreurs de validation.

1. **Contrôleur « product\_detail » :**

Cette vue gère les requêtes GET, PUT et DELETE pour un produit spécifique. Elle prend un paramètre id qui est l'identifiant du produit dans la base de données.

Si la requête est de type GET, elle récupère le produit correspondant à l'identifiant donné à l'aide de la fonction get\_object\_or\_404 qui renvoie une réponse HTTP 404 si le produit n'existe pas. Ensuite, elle sérialise le produit en JSON à l'aide du sérialiseur ProductSerializer et renvoie une réponse HTTP 200 avec les données du produit.

Si la requête est de type PUT, elle récupère également le produit correspondant à l'identifiant donné. Ensuite, elle essaie de mettre à jour les données du produit avec les données envoyées dans la requête en utilisant le sérialiseur ProductSerializer. Si les données sont valides, elle enregistre le produit mis à jour dans la base de données et renvoie une réponse HTTP 200 avec les données du produit. Sinon, elle renvoie une réponse HTTP 400 avec les erreurs de validation.

Si la requête est de type DELETE, elle récupère le produit correspondant à l'identifiant donné et le supprime de la base de données. Ensuite, elle renvoie une réponse HTTP 204 sans contenu.

1. **Contrôleur « product\_list » :**

Ce contrôleur traite les requêtes GET et POST pour la liste des produits.

Lorsqu'une requête GET est reçue, le contrôleur récupère tous les produits stockés dans la base de données en utilisant Product.objects.all(). Les produits sont ensuite sérialisés en JSON à l'aide du sérialiseur ProductSerializer avec l'option many=True, car plusieurs objets sont renvoyés. Enfin, une réponse HTTP contenant les données sérialisées est renvoyée.

Lorsqu'une requête POST est reçue, le contrôleur crée une nouvelle instance du sérialiseur ProductSerializer avec les données envoyées dans la requête. Si les données sont valides, le sérialiseur enregistre un nouveau produit dans la base de données en utilisant serializer.save(). Une réponse HTTP avec un code de statut 201 Created et les données sérialisées du nouveau produit sont renvoyées. Si les données ne sont pas valides, une réponse HTTP avec un code de statut 400 Bad Request et les erreurs de validation sont renvoyées.

En résumé, ce contrôleur fournit une API RESTful pour la liste des produits, permettant de récupérer tous les produits et d'en créer de nouveaux en utilisant des requêtes HTTP GET et POST

1. **Contrôleur « category\_list » :**

Ce contrôleur est une vue basée sur une fonction dans Django REST Framework qui gère les requêtes GET et POST pour une liste de catégories.

Lorsqu'une requête GET est reçue, le contrôleur récupère toutes les catégories stockées dans la base de données en utilisant `Category.objects.all()`. Les catégories sont ensuite sérialisées en JSON à l'aide du sérialiseur `CategorySerializer` avec l'option `many=True`, car plusieurs objets sont renvoyés. Enfin, une réponse HTTP contenant les données sérialisées est renvoyée.

Lorsqu'une requête POST est reçue, le contrôleur crée une nouvelle instance du sérialiseur `CategorySerializer` avec les données envoyées dans la requête. Si les données sont valides, le sérialiseur enregistre une nouvelle catégorie dans la base de données en utilisant `serializer.save()`.

Une réponse HTTP avec un code de statut 201 Created et les données sérialisées de la nouvelle catégorie sont renvoyées. Si les données ne sont pas valides, une réponse HTTP avec un code de statut 400 Bad Request et les erreurs de validation sont renvoyées.

En résumé, ce contrôleur fournit une API RESTful pour la liste des catégories, permettant de récupérer toutes les catégories et d'en créer de nouvelles en utilisant des requêtes HTTP GET et POST.

1. **Contrôleur « category\_detail »**:

Ce contrôleur basé sur une fonction avec le décorateur @apiview, permet de gérer les détails d'une catégorie en particulier. Il accepte les requêtes GET, PUT et DELETE avec un ID de catégorie spécifique.

Lorsqu'une requête GET est reçue, le contrôleur récupère la catégorie correspondante à l'ID et la sérialise en JSON à l'aide de CategorySerializer. Ensuite, il renvoie les données sérialisées dans une réponse HTTP.

Lorsqu'une requête PUT est reçue, le contrôleur récupère également la catégorie correspondante à l'ID, mais cette fois, il met à jour les données de la catégorie avec les données envoyées dans la requête. Pour ce faire, il crée une nouvelle instance de CategorySerializer avec les données de la catégorie existante et les données envoyées dans la requête. Si les données sont valides, il enregistre la catégorie mise à jour et renvoie les données sérialisées dans une réponse HTTP. Si les données ne sont pas valides, il renvoie les erreurs de validation dans une réponse HTTP avec un code d'état 400.

Enfin, lorsqu'une requête DELETE est reçue, le contrôleur supprime simplement la catégorie correspondante à l'ID et renvoie une réponse HTTP avec un code d'état 204, indiquant que la suppression a réussi.

En résumé, ce contrôleur permet de gérer les détails d'une catégorie spécifique en répondant aux requêtes HTTP GET, PUT et DELETE avec un ID de catégorie donné. Il sérialise les données de la catégorie en JSON et les renvoie dans les réponses HTTP, tout en gérant les mises à jour et les suppressions de la catégorie.

1. **Contrôleur « cart\_detail » :**

Ce contrôleur permet de récupérer les détails du panier d'un utilisateur spécifique. Il accepte uniquement les requêtes GET avec un ID d'utilisateur.

Le contrôleur utilise la classe de permission IsAuthenticated, ce qui signifie que seuls les utilisateurs authentifiés peuvent accéder à cette vue.

Lorsqu'une requête GET est reçue, le contrôleur récupère le panier correspondant à l'ID d'utilisateur donné en utilisant get\_object\_or\_404. Ensuite, il sérialise les données du panier en JSON à l'aide de CartSerializer. Enfin, il renvoie les données sérialisées dans une réponse HTTP.

Il convient également de noter que le contrôleur utilise l'objet logger pour enregistrer les événements de l'application.

En résumé, ce contrôleur permet de récupérer les détails du panier d'un utilisateur spécifique en répondant aux requêtes HTTP GET avec un ID d'utilisateur donné. Il nécessite une authentification et sérialise les données du panier en JSON avant de les renvoyer dans une réponse HTTP. Il utilise également l'objet logger pour enregistrer les événements de l'application.

1. **Contrôleur « add\_product\_to\_cart » :**

Ce contrôleur Django REST Framework permet d'ajouter un produit au panier d'un utilisateur authentifié. Il accepte uniquement les requêtes POST.

Le contrôleur utilise la classe de permission IsAuthenticated, ce qui signifie que seuls les utilisateurs authentifiés peuvent accéder à cette vue.

Lorsqu'une requête POST est reçue, le contrôleur récupère l'ID du produit et la quantité à partir des données de la requête. Ensuite, il récupère le produit correspondant à l'ID en utilisant get\_object\_or\_404. Il récupère également le panier de l'utilisateur en utilisant get\_or\_create.

Si le produit existe déjà dans le panier, le contrôleur met à jour la quantité en conséquence. Sinon, il crée un nouvel élément de panier avec le produit et la quantité spécifiés.

Enfin, le contrôleur sérialise les données de l'élément de panier en JSON à l'aide de CartItemSerializer et renvoie les données sérialisées dans une réponse HTTP avec un code d'état 201 Created.

En résumé, ce contrôleur permet d'ajouter un produit au panier d'un utilisateur authentifié en répondant aux requêtes HTTP POST. Il récupère l'ID du produit et la quantité à partir des données de la requête, récupère le produit et le panier correspondants, met à jour la quantité ou crée un nouvel élément de panier, puis renvoie les données sérialisées de l'élément de panier dans une réponse HTTP avec un code d'état 201 Created.

1. **Contrôleur « update\_cart\_item » :**

Ce contrôleur traite les requêtes POST pour mettre à jour la quantité d'un élément dans le panier d'achat. Il prend en entrée deux paramètres à partir des données de la requête : cart\_item\_id et quantity. Il récupère l'objet CartItem correspondant à l'aide de la fonction get\_object\_or\_404, met à jour sa quantité avec la nouvelle valeur et enregistre l'objet mis à jour en utilisant la méthode save(). Enfin, il renvoie une réponse HTTP avec un message de succès indiquant que l'élément du panier a été mis à jour avec succès.

**10.Controleur « remove\_product\_from\_cart » :**

Cette fonction traite les requêtes DELETE pour supprimer un élément du panier d’achat. Il prend en entrée un paramètre cart\_item\_id à partir de l’URL et récupère l’objet CartItem correspondant à l’aide de la fonction get\_object\_or\_404. Il supprime ensuite l’objet à l’aide de la méthode delete() et renvoie une réponse http avec un message de succès indiquant que le produit a été supprimé du panier avec succès. Le code de statut http utilisé est 204, ce qui indique qu’il n’y a pas de contenu à renvoyer dans la réponse.

**11.Controleur « cart\_list » :**

Ce contrôleur traite les requêtes GET pour récupérer la liste de tous les paniers d’achat. Il récupère tous les objets Cart de la base de données à l’aide du gestionnaire d’objets Django objects, sérialise les données à l’aide du sérialiseur CartSerializer avec l’option many=True et renvoie une réponse http avec les données sérialisées.

Maintenant que nous avons traité les contrôleurs nous pouvons créer nos urls.

**Créations des URLS.**

Dans tout application Django nous avons besoin de définir nos urls.

Les URLs sont définies en utilisant la fonction path() de django.urls et sont associées à des vues dans le fichier views.py de l'application.

Dans ce fichier, les URLs sont regroupées en quatre catégories :

1. URLs pour les utilisateurs :

* users/ : une vue pour afficher la liste de tous les utilisateurs (accessible uniquement à l'admin).
* users/<int:id>/ : une vue pour afficher les détails d'un utilisateur spécifique en fonction de son ID.

1. URLs pour les catégories :

* categories/ : une vue pour afficher la liste de toutes les catégories.
* categories/<int:id>/ : une vue pour afficher les détails d'une catégorie spécifique en fonction de son ID.

1. URLs pour les produits :

* products/ : une vue pour afficher la liste de tous les produits.
* products/<int:id>/ : une vue pour afficher les détails d'un produit spécifique en fonction de son ID.

1. URLs pour le panier :

* cart/<int:user\_id>/ : une vue pour afficher les détails du panier d'un utilisateur spécifique en fonction de son ID.
* carts/ : une vue pour afficher la liste de tous les paniers.
* cart/add/ : une vue pour ajouter un produit au panier.
* cart/update/ : une vue pour mettre à jour la quantité d'un produit dans le panier.
* cart/remove/<int:cart\_item\_id>/ : une vue pour supprimer un produit du panier en fonction de son ID de panier.

Chaque URL est associée à une vue spécifique et a un nom unique pour faciliter la référence à cette URL dans d'autres parties de l'application. Les URLs pour les utilisateurs et les catégories utilisent des expressions régulières \* pour capturer l'ID de l'objet spécifique à afficher. Les URLs pour le panier utilisent également des expressions régulières pour capturer l'ID de l'utilisateur et l'ID du produit dans le panier.

Comme on peut le constater nous avons l’ensembles des urls utiles à notre application.



N.B : Les expressions régulières sont comme un outil de recherche avancé pour les chaînes de caractères. Elles te permettent de définir un motif de recherche et de trouver toutes les correspondances dans un texte donné.

**Sérialisation des données**

Afin de permettre le bon fonctionnement de notre application, nous avons besoin d’un serialiser.

Les sérialiseurs sont utilisés pour convertir des objets Python complexes en formats plus simples, tels que JSON ou XML, qui peuvent être facilement transmis sur le web. Dans notre fichier, il y a quatre sérialiseurs définis pour les modèles User, Category, Product et Cart de l'application.

Le premier sérialiseur est UserSerializer, qui est utilisé pour sérialiser les objets utilisateur. Il hérite de serializers.ModelSerializer et spécifie les champs à inclure dans la sérialisation. Dans ce cas, les champs 'id', 'username', 'email' et 'phone\_number' sont inclus.

Le deuxième sérialiseur est CategorySerializer, qui est utilisé pour sérialiser les objets de l’entité catégorie. Il spécifie que tous les champs du modèle Category doivent être inclus dans la sérialisation.

Le troisième sérialiseur est ProductSerializer, qui est utilisé pour sérialiser les objets de produit. Comme pour CategorySerializer, il spécifie que tous les champs du modèle Product doivent être inclus dans la sérialisation.

Le quatrième sérialiseur est CartItemSerializer, qui est utilisé pour sérialiser les objets de l’entité panier. Il comprend deux champs en lecture seule, 'product\_name' et 'product\_price', qui sont obtenus à partir de l'objet de produit associé.

Enfin, le cinquième sérialiseur est CartSerializer, qui est utilisé pour sérialiser les objets de panier complets. Il comprend un champ 'cart\_items' qui utilise CartItemSerializer pour sérialiser les objets de panier associés. Il comprend également un champ 'user' en lecture seule qui obtient le nom d'utilisateur associé.

En résumé, ce fichier définit une série de sérialiseurs pour les modèles de l'application, qui sont utilisés pour convertir les objets Python en formats plus simples pour la transmission sur le web (JSON). Les sérialiseurs spécifient les champs à inclure dans la sérialisation et peuvent également inclure des champs en lecture seule pour obtenir des informations supplémentaires à partir d'objets associés.

**Test de L’api**

Afin d’assurer le bon fonctionnement de notre api basé sur Django Rest Framework. Je vais effectuer quelques testes.

Pour ce faire je vais utiliser le module de test que nous fournis Django Rest Framework, « APITestCase »

Ce module nous permet d’effectuer diffèrent tests :

* Test d'intégration
* Test Unitaire
* Test d'autorisation.

Voyons plus en détails a quoi consiste ces différents tests.

* Test d’intégration :

Les tests d'intégration sont une méthode de test qui permet de vérifier que différents composants d'un système fonctionnent correctement ensemble. Contrairement aux tests unitaires qui se concentrent sur des parties isolées du code, les tests d'intégration visent à tester le système dans son ensemble. Ces tests sont importants pour s'assurer que toutes les parties du système interagissent correctement entre elles et produisent les résultats attendus. En général, les tests d'intégration sont effectués dans un environnement de test qui simule les conditions réelles d'utilisation du système.

* Test unitaire :

Les tests unitaires sont une pratique de développement logiciel qui consiste à tester des unités de code isolées, comme des fonctions ou des méthodes, pour s'assurer qu'elles fonctionnent correctement. Ces tests sont généralement automatisés et effectués par les développeurs eux-mêmes. Les tests unitaires permettent de vérifier que chaque unité de code fonctionne correctement de manière indépendante, avant de l'intégrer dans le système global. Cela permet de détecter rapidement les erreurs et les bogues, et de faciliter la maintenance et l'évolution du code. Les tests unitaires sont donc essentiels pour garantir la qualité et la fiabilité d'un logiciel.

* Test d’autorisation :

Les tests d'autorisation sont une pratique de test logiciel qui vise à vérifier que les utilisateurs n'ont accès qu'aux fonctionnalités et ressources pour lesquelles ils sont autorisés. Ces tests permettent de s'assurer que les contrôles d'accès sont correctement mis en place et que les utilisateurs ne peuvent pas accéder à des données sensibles ou effectuer des actions non autorisées. Les tests d'autorisation sont généralement effectués en simulant des scénarios d'utilisation avec différents niveaux d'autorisation et en vérifiant que les résultats attendus sont obtenus. Ces tests sont importants pour garantir la sécurité et la confidentialité des données d'une application.

Maintenant il serait judicieux de regarder les différents tests que nous avons effectué pour vérifier le bon fonctionnement de notre application Back end (API) Django Framework.

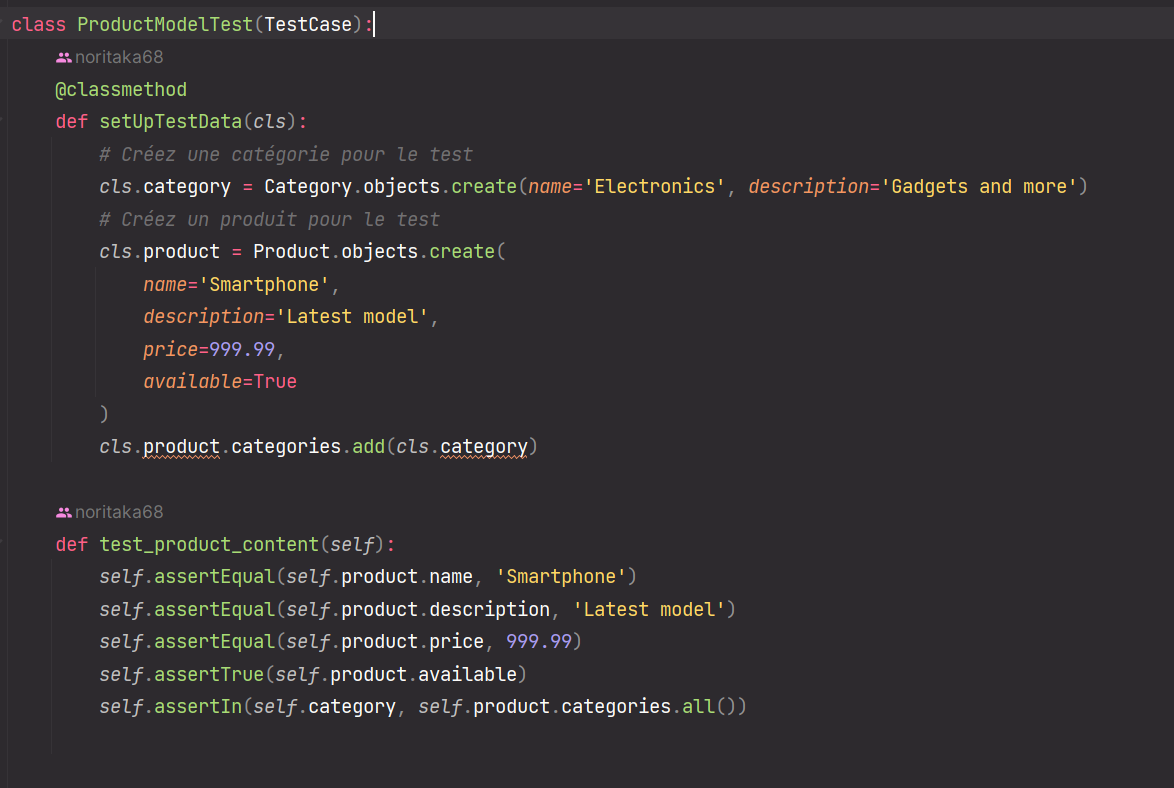
* + **Test Unitaire du Model :**

Dans ce test, nous testons le modèle Product de notre application. Nous créons une catégorie et un produit pour les besoins du test en utilisant la méthode setUpTestData. Ensuite, nous définissons une méthode de test appelée "test\_product\_content" pour vérifier le contenu du produit.

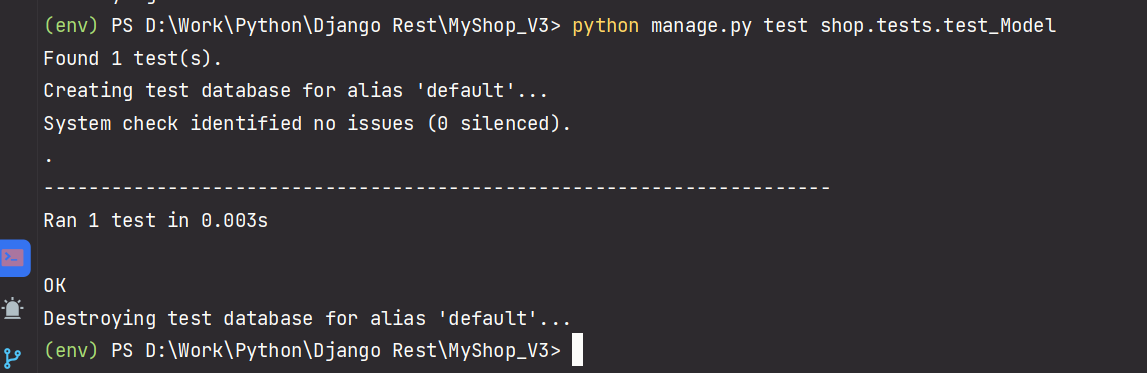
Dans cette méthode, nous utilisons la méthode "assertEqual" pour vérifier que les attributs du produit correspondent aux valeurs attendues. Nous vérifions que le nom du produit est "Smartphone", que la description est "Latest model", que le prix est de 999,99 et que le produit est disponible.

Enfin, nous utilisons la méthode "assertIn" pour vérifier que la catégorie créée précédemment est bien associée au produit. Cette méthode vérifie que la catégorie fait partie des catégories associées au produit en utilisant la méthode "all()" sur l'attribut "categories" du produit.

Ce test permet donc de vérifier que le modèle Product fonctionne correctement et que les données associées sont bien enregistrées dans la base de données.



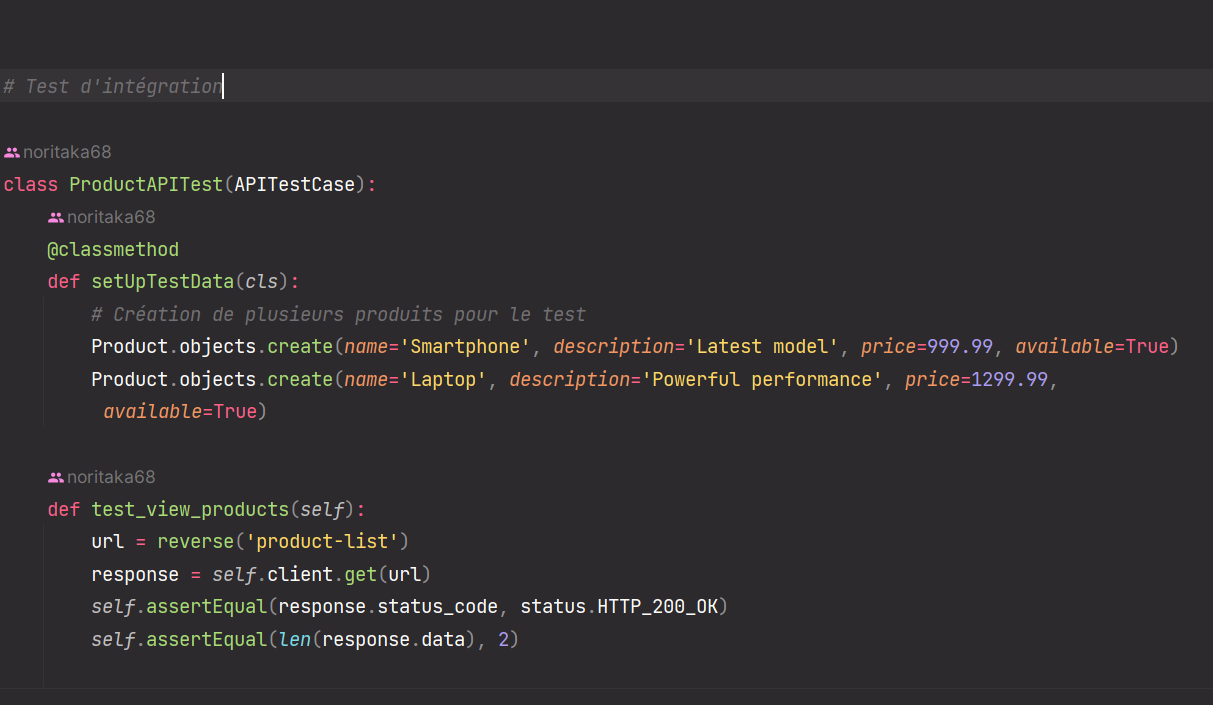
Maintenant je vais exécuter le test pour vérifier que tout fonctionne, en utilisent cette commande : python manage.py test shop.tests.test\_Model .



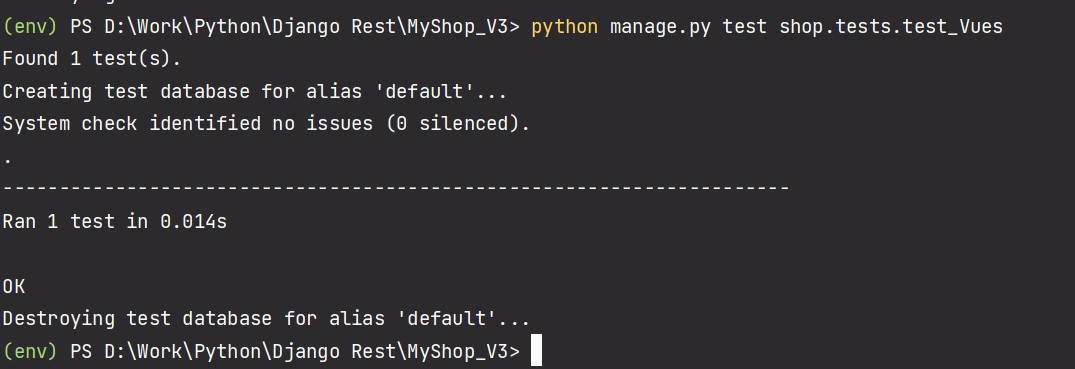
Comme on peut le constater le test c’est bien passé.

* + **Test d’intégration de notre Vue**

Ce test est une vérification de l'API de la boutique en ligne pour s'assurer que la liste des produits est correctement affichée. Dans un premier temps, deux produits sont créés dans la base de données grâce à la méthode setUpTestData. Ensuite, la méthode test\_view\_products vérifie que l'URL de la liste des produits renvoie bien une réponse HTTP 200 OK et que les deux produits créés précédemment sont bien présents dans la réponse. Ce test permet donc de s'assurer que l'API fonctionne correctement et que les produits sont bien affichés dans la liste.



Maintenant nous allons exécuter le test.



Comme on peut le constater le test c’est bien déroulé.

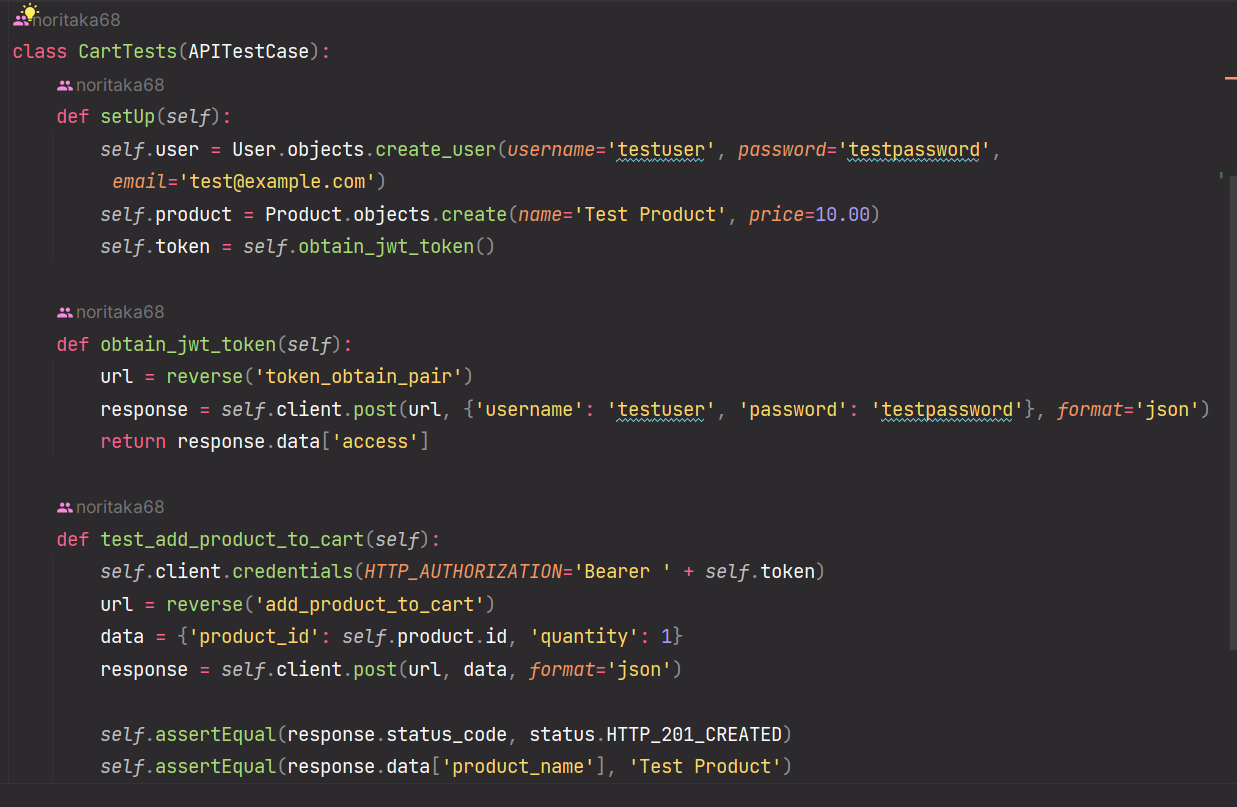
* + **Test d’intégration du panier**

Ce test est un test d'intégration pour le panier de l'application de commerce électronique. Il utilise la classe APITestCase de Django Rest Framework pour tester l'API du panier. Dans un premier temps, le test crée un utilisateur et un produit dans la base de données. Ensuite, il obtient un jeton d'authentification pour l'utilisateur en utilisant l’Endpoint d'authentification JWT.

Cela nous permet de test en même temps le système d'authentification JWT. Et de s'assurer que l'application est bien sécurisé.

Le test principal est la méthode test\_add\_product\_to\_cart, qui teste l'ajout d'un produit au panier. Pour ce faire, le test envoie une requête POST à l'endpoint d'ajout de produit au panier en utilisant les informations d'identification de l'utilisateur et les détails du produit. Ensuite, le test vérifie que la réponse a le bon code d'état HTTP (201 Created) et que les données renvoyées correspondent aux détails du produit ajouté.

En résumé, ce test vérifie que l'API du panier fonctionne correctement en testant l'ajout d'un produit au panier. Il utilise des données de test créées dans la base de données et des requêtes HTTP pour simuler les interactions avec l'API.



Maintenant je vais exécuter le test afin d’assure que notre méthode add\_product\_to\_cart fonctionne bien ainsi que notre système d’authentification.

Pour ce faire je vais utiliser la commande : python manage.py test shop.tests.test\_Cart



Comme on peut le constater le test c’est bien déroulé.

Maintenant que nous avons vérifié, que notre API fonctionne bien nous pouvons créer notre interface graphique.

**Création de l’application Front End**

Afin de fournir une interface a notre application je vais développer une application React.

React est une bibliothèque JavaScript populaire utilisée pour construire des interfaces utilisateur interactives et performantes. En choisissant React pour notre application, nous pouvons bénéficier de ses performances élevées et de ses nombreux outils, tels que les hooks, qui facilitent le développement d'une application front-end. Les hooks sont des fonctions qui permettent de gérer l'état et le cycle de vie des composants React de manière plus simple et plus efficace, ce qui réduit la complexité du code et améliore la maintenabilité de l'application. En somme, React est un choix judicieux pour développer une interface utilisateur moderne et réactive pour notre application.

Alors, pour construire notre application React, j'ai choisi d'utiliser un outil appelé Vite JS. C'est un outil de construction de nouvelle génération qui permet de créer des applications React rapidement et facilement. Avec Vite JS, nous pouvons créer une application React en quelques minutes à peine, sans avoir à nous soucier des configurations complexes. Cela nous permettra de nous concentrer sur le développement de notre application plutôt que sur la configuration de notre environnement de développement. De plus, Vite JS est connu pour ses excellentes performances, ce qui signifie que notre application se chargera rapidement et fonctionnera de manière fluide et efficace. En somme, Vite JS est un outil idéal pour créer une application React dans les meilleures conditions possibles.

**1. Créations de l’application React :**

Afin de créer notre projet React je vais utiliser la commande Vite js.

« npm create vite@latest my-react-app -- --template react ».

Apres l’exécution de cette commande viteJS nous crées une application React prêt à l’utilisation.

Lorsque nous créons une application React avec ViteJS, plusieurs éléments sont générés pour nous aider à structurer notre projet. Voici les principaux éléments :

1. Le dossier "src" : c'est ici que nous allons écrire notre code source. Il contient déjà un fichier "index.html" qui est le point d'entrée de notre application.

2. Le fichier "main.jsx" : c'est le fichier principal de notre application React. C'est ici que nous allons écrire notre code React.

3. Le fichier "vite.config.js" : c'est le fichier de configuration de ViteJS. Il nous permet de configurer les différents paramètres de notre projet, comme les dépendances, les chemins, etc.

4. Le dossier "public" : c'est ici que nous pouvons placer nos fichiers statiques, comme les images ou les feuilles de style.

5. Le dossier "node\\_modules" : c'est ici que sont stockées toutes les dépendances de notre projet. (React router dom, bootstrap et etc..)

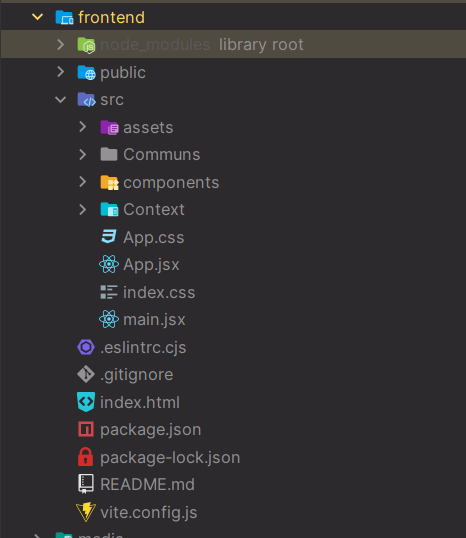
6. Le fichier "package.json" : c'est le fichier qui contient toutes les informations sur notre projet, comme les dépendances, les scripts, etc.

En développant notre application React, nous allons créer des composants React, qui sont des éléments réutilisables de notre interface utilisateur.

Nous allons également utiliser des hooks (useState), qui sont des fonctions spéciales qui nous permettent de gérer l'état et les effets secondaires de notre application.

Enfin, nous allons utiliser des outils de développement, comme le débogueur React, pour nous aider à tester et déboguer notre application.

En résumé, un projet React construit avec ViteJS est composé de plusieurs éléments clés, notamment le dossier "src", le fichier "main.jsx", le fichier "vite.config.js", le dossier "public", le dossier "node\\_modules" et le fichier "package.json". Nous allons créer des composants React et utiliser des hooks pour gérer l'état et les effets secondaires de notre application, ainsi que des outils de développement pour tester et déboguer notre application.



* 1. **Installation de nos dépenses.**

Pour le bon fonctionnement de notre application nous aurons besoin d’installer différentes dépendances.

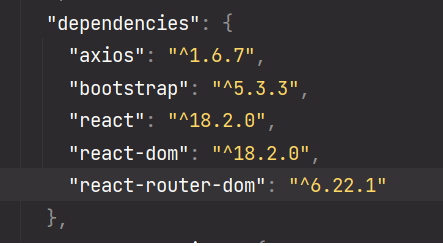
* Pour le Style et afin d’avoir une interface responsive je vais utiliser Bootstrap.
* Afin de pouvoir exécuter des requetés dans les meilleures conditions je vais utiliser Axios.
* Pour pouvoir naviguer entre les différentes pages de notre application React il nous faudra un système de routage.

Pour ce faire je vais utiliser React-router-dom.

Je vais utiliser la commande « npm i bootstrap axios react-router-dom ».

Avec cette unique commande je vais pouvoir installer toutes mes dépendances.

Pour vérifier que l’installation c’est bien faites je vais vérifier le fichier « package.json ».



Comme on peut le constaté l’installation des dépendances c’est bien passé.

* 1. **Structure du Projet.**

Comme on a pu le constater notre projet React sera construit principalement dans le répertoire « src ».

C’est la que nous trouvons les principaux composants de notre application.

Afin d’avoir une architecture solide et propres. Je vais créer plusieurs sous répertoires qui contiendront nos différents composants.

Comme vous pourez le constaté dans l’image ci-dessous je vais créer un repertoire « Communs » qui contiendra la navbar et le footer de l’application.

Le répertoire « components », quand a lui comporte plusieurs sous répertoires. Ils continents les composent principaux de notre application.

* Cart : Le panier
* Home : La page d’accueil de notre application.
* Login : Notre page d’authentification.
* Products : Le composent qui affichera nos produits dans l’application.
* SignUpPage : La page d’inscription.

Dans le dossier « src » nous avons aussi le répertoire « context ».

Ce répertoire contient des éléments importants dan notre applications, comme la logique d’authentification, « AuthContext », la gestion de l’état du panier « CartContext » et enfin la fonction de recherche des produits « SearchContext ».

Nous trouvons dans ce répertoire aussi le fichier « App.jsx ».

Ce composant React contient le routage. Il y a six routes pour six pages différentes : la page d'accueil, la page de connexion, la page d'inscription, la page des produits, la page de détails du produit et la page du panier. Chaque page est rendue par un composant spécifique. Enfin, le composant entier est exporté pour être utilisé dans d'autres parties de l'application.

Dans le répertoire « src » toujours nous avons aussi le fichier « main.jsx ».

Ce composent React est un peu le chef d’orchestre de l’application. Il contient les contexts (AuthContext, CartContext et SearchContext).

Il importe aussi la librairie Bootstrap ainsi que le CSS.

**Les Composent de notre application**

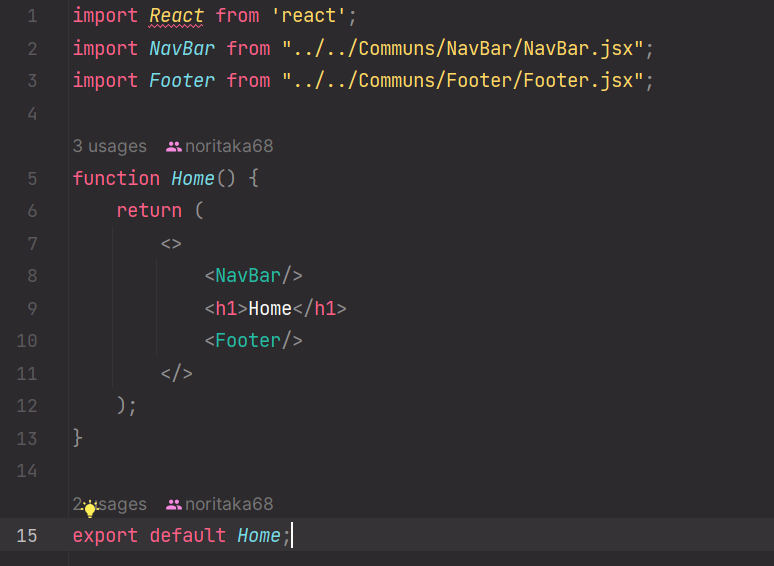
**1. La page d’accueil :**

Dans le cadre du développement de notre application, le composant React désigné pour représenter l'interface principale joue un rôle crucial. Ce dernier intègre deux modules essentiels, à savoir NavBar et Footer, dont l'usage s'étend à divers segments de l'application.

Le composent Home est spécifiée afin de générer un fragment,lequel encapsule l'ensemble des éléments constitutifs de l'interface d'accueil. Initialement, le module NavBar est invoqué pour assurer l'affichage de la barre de navigation en position supérieure de la page.

En outre, un titre de premier niveau « h1 » est présenté, indiquent la mention "Home". Cette indication a pour objectif d'informer clairement l'utilisateur de sa localisation au sein de l'interface principale de l'application. Enfin , l'implémentation du module Footer permet l'insertion du pied de page en bas de l'interface, contribuant ainsi à l'homogénéité de l'expérience utilisateur.

En résumé, ce composant s'avère être l'épicentre d'accueil de notre application, harmonisant l'utilisation de modules récurrents pour forger une interface utilisateur intégrée et intuitive, tout en signalant explicitement à l'utilisateur sa présence sur la page principale



1. **Login Page :**

Pour commencer, notre composant de connexion ne laisse rien au hasard. Il fait appel à une série de hooks et de composants essentiels pour son bon fonctionnement, incluant React pour la base, des hooks tels que useState et useNavigate pour gérer les états et la navigation, useAuth pour l'authentification, et bien sûr, les éléments NavBar et Footer pour une cohérence visuelle à travers l'application.

La fonction LoginPage prend le relais en nous présentant la page de connexion. C'est là que useState entre en jeu, permettant de stocker et de suivre les informations saisies par l'utilisateur, comme le nom d'utilisateur et le mot de passe.

Lorsque l'utilisateur soumet ses informations via le formulaire de connexion, la fonction handleSubmit est mise en action. Elle fait appel à login, une méthode d'authentification qui vérifie les identifiants fournis. Comme login est une promesse, handleSubmit utilise le mot-clé await pour attendre le résultat de cette tentative de connexion.  « promesse »

Si tout se passe bien et que la connexion est établie, l'utilisateur est redirigé vers la page d'accueil grâce à navigate. Si, par contre, la tentative échoue, une erreur est enregistrée dans la console pour diagnostic. C'est une manière simple et efficace de gérer les flux d'authentification.

En résumé, le composant de connexion se distingue par son utilisation habile de useState pour gérer les entrées utilisateur, de useNavigate pour rediriger après une connexion réussie, et de useAuth pour le processus d'authentification. handleSubmit joue un rôle clé en orchestrant la logique d'attente et de traitement de la réponse de la fonction login, assurant ainsi une transition fluide vers la page d'accueil pour l'utilisateur authentifié.





1. **Produit List :**

Ce composant fonctionne tel une vitrine numérique, offrant aux utilisateurs la possibilité de naviguer à travers un éventail de produits et de les filtrer selon leur nom ou description. Pour faciliter cette fonctionnalité, il s'appuie sur diverses bibliothèques et composants, notamment `useState`, `useEffect`, `axios`, `Link`, `useCart`, ainsi que `NavBar` et `Footer`, pour une expérience utilisateur enrichie et une navigation fluide.

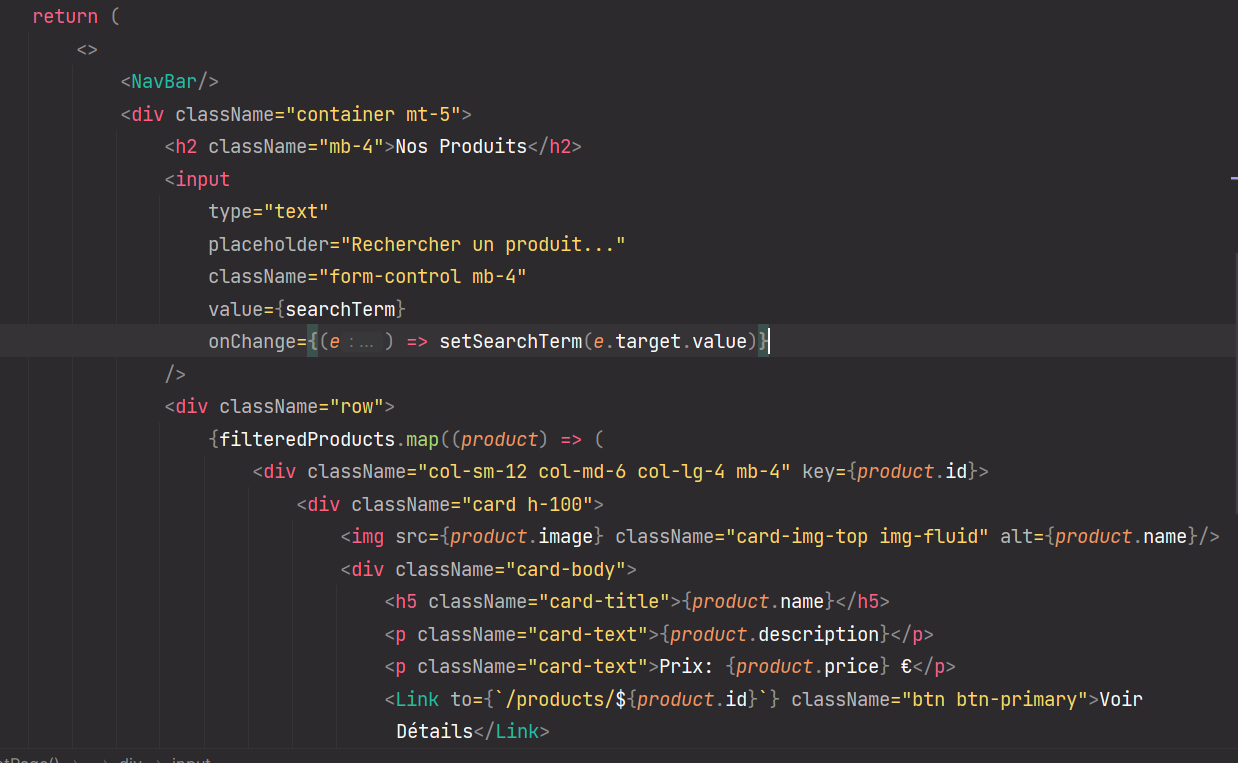
À l'ouverture de la page listant les produits, le composant `NavBar` se présente en premier lieu, établissant une navigation unifié à traver l'application. Suit un conteneur intégrant une barre de recherche intuitive, conçue pour permettre aux utilisateurs de rechercher des produits en fonction de critères spécifiques. Le hook `useState` est employé pour gérer les données saisies dans cette barre de recherche.

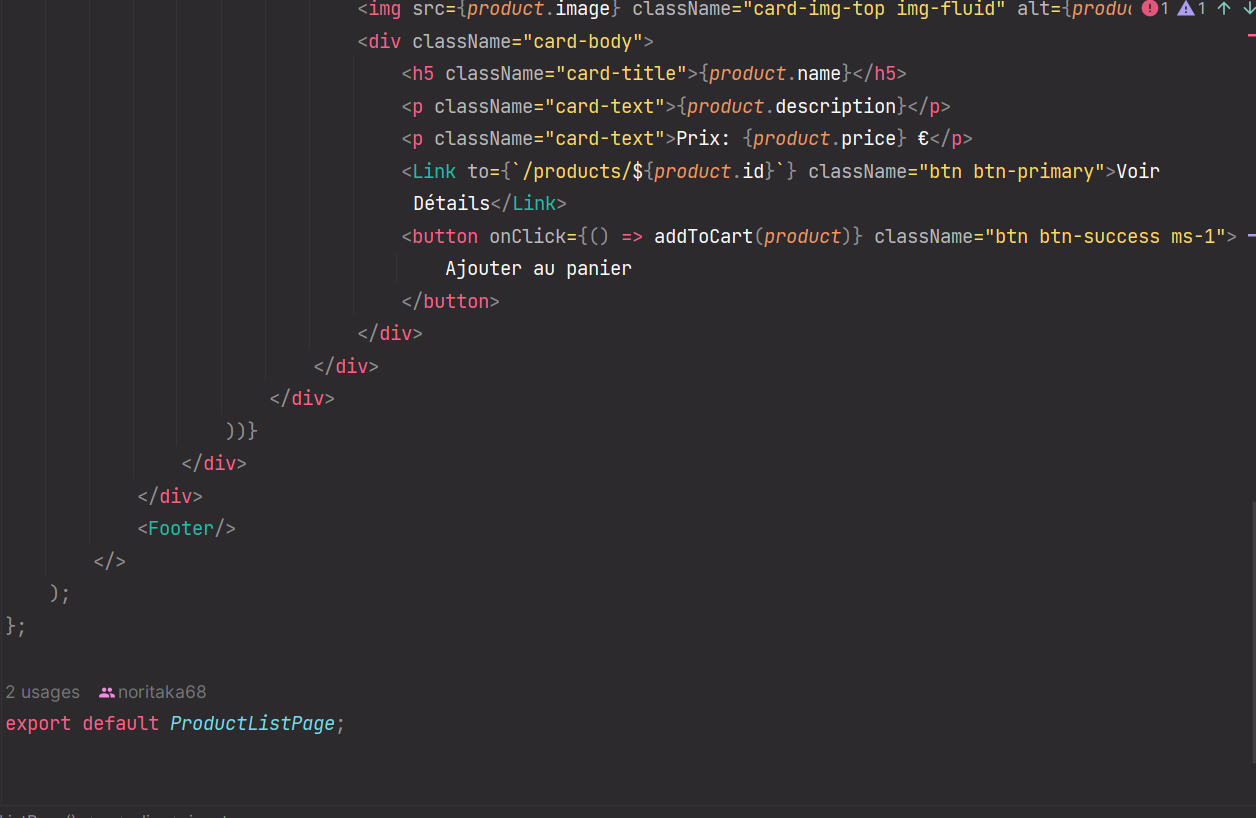
Les informations relatives aux produits sont extraites via une API grâce à la bibliothèque `axios`, les données étant par la suite stockées localement au moyen de `useState`. L'adresse de base de l'API est conservée dans une variable `baseUrl`. L'actualisation des données des produits lors du chargement du composant est assurée par `useEffect`.

Une fois les produits chargés, ils sont filtrés en accord avec le terme de recherche utilisé, à travers la fonction `filter`. Les produits ainsi sélectionnés sont affichés sous forme de grille, chaque carte produit proposant des détails tels que le nom, la description, le prix, une image, et deux boutons. Le premier bouton, "Voir Détails", redirige vers la page de détail du produit concerné, tandis que le second, "Ajouter au panier", permet d'ajouter le produit au panier via la fonction `addToCart` fournie par le contexte du panier.

En résumé, ce composant crée une interface utilisateur intuitive et engageante pour l'exploration et l'acquisition de produits en ligne. Il offre aux utilisateurs la capacité de filtrer les produits selon leurs préférences, d'accéder aux détails d'un produit spécifique et d'ajouter des articles au panier, facilitant ainsi une expérience d'achat en ligne à la fois pratique et agréable.







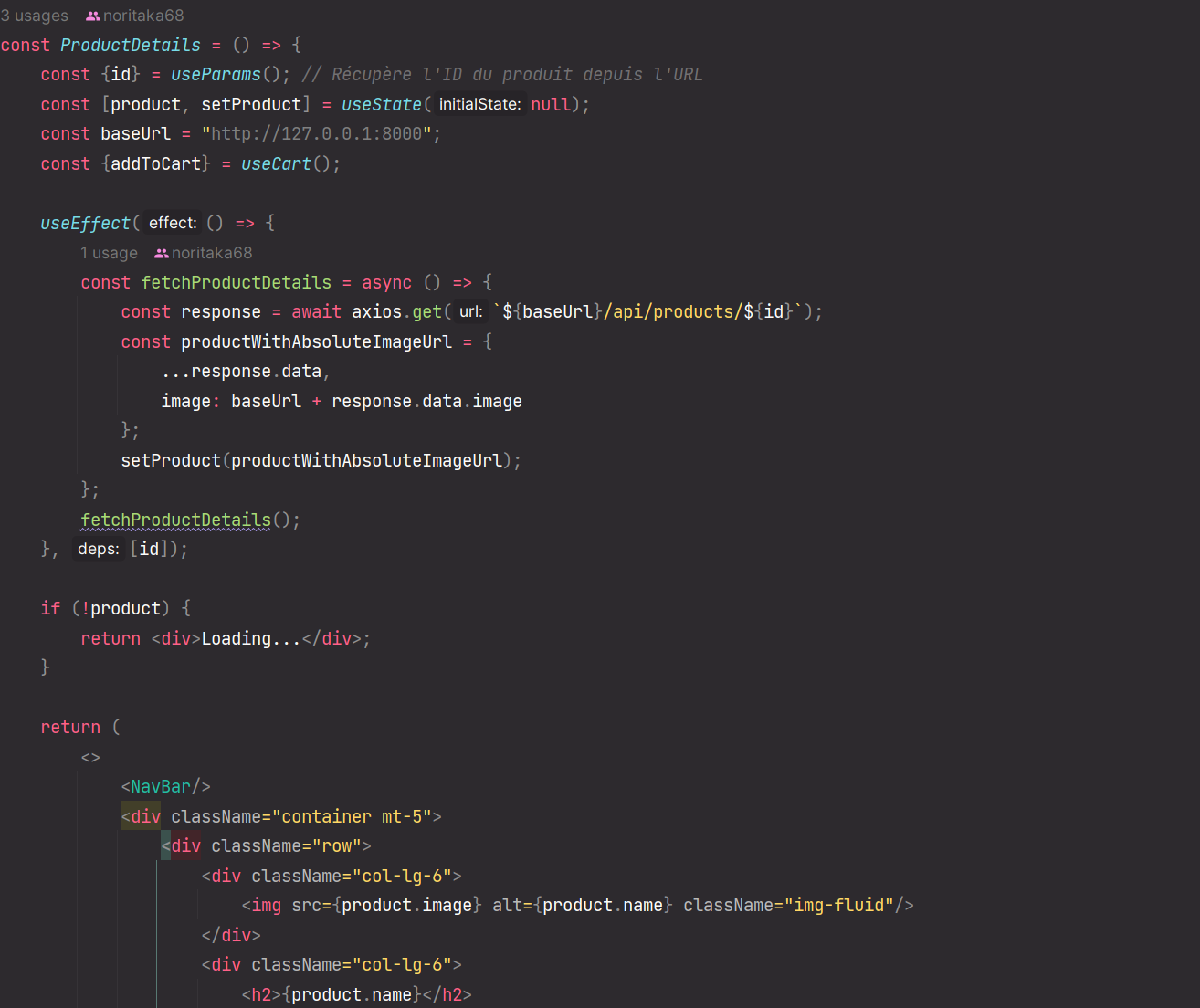
4.**Product Details :**

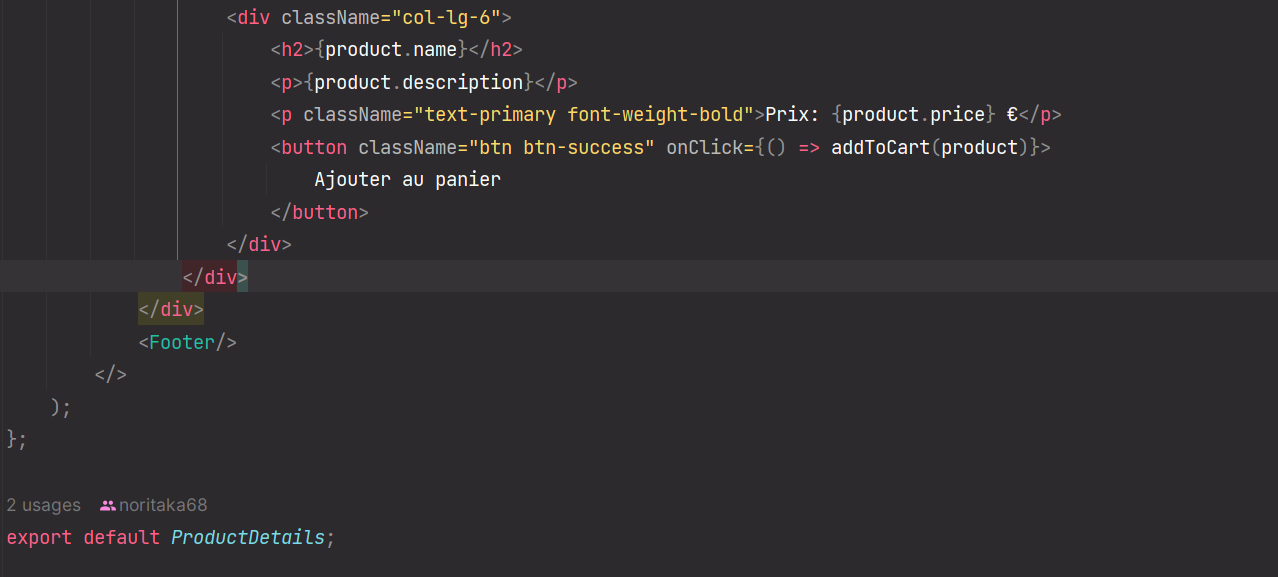
Ce composant React affiche les détails d'un produit spécifique et permet de l'ajouter au panier. Il utilise plusieurs hooks et composants, tels que React, useState, useEffect, axios, useParams, useCart.

Lorsque l'utilisateur accède à la page de détails du produit, le composant récupère l'ID du produit à partir de l'URL à l'aide de la fonction useParams. Ensuite, il récupère les détails du produit à partir d'une API en utilisant la bibliothèquee axios et les stocke dans l'état local de la page a l'aide de useState. L'URL de base de l'API est stocké dans une constante baseUrl.

Pendant le chargement des détails du produit, un message "Loading..." est affiché. Une fois les détails chargés, ils sont affichés dans des « card » bootstrap. La première colonne affiche l'image du produit, tandis que la deuxième colonne affiche les détails du produit, tels que son nom, sa description, son prix et un bouton "Ajouter au panier".

Le bouton "Ajouter au panier" utilise la fonction addToCart du contexte de panier pour ajouter le produit au panier. Lorsque l'utilisateur clique sur ce bouton, le produit est ajouté au panier et l'état du panier est mis à jour.





1. Sign Up Page :

Cette page permet aux utilisateurs de s'inscrire en fournissant un nom d'utilisateur, une adresse e-mail et un mot de passe. Il utilise plusieurs hooks et composants, tels que, useState, axios, useNavigate, NavBar et Footer.

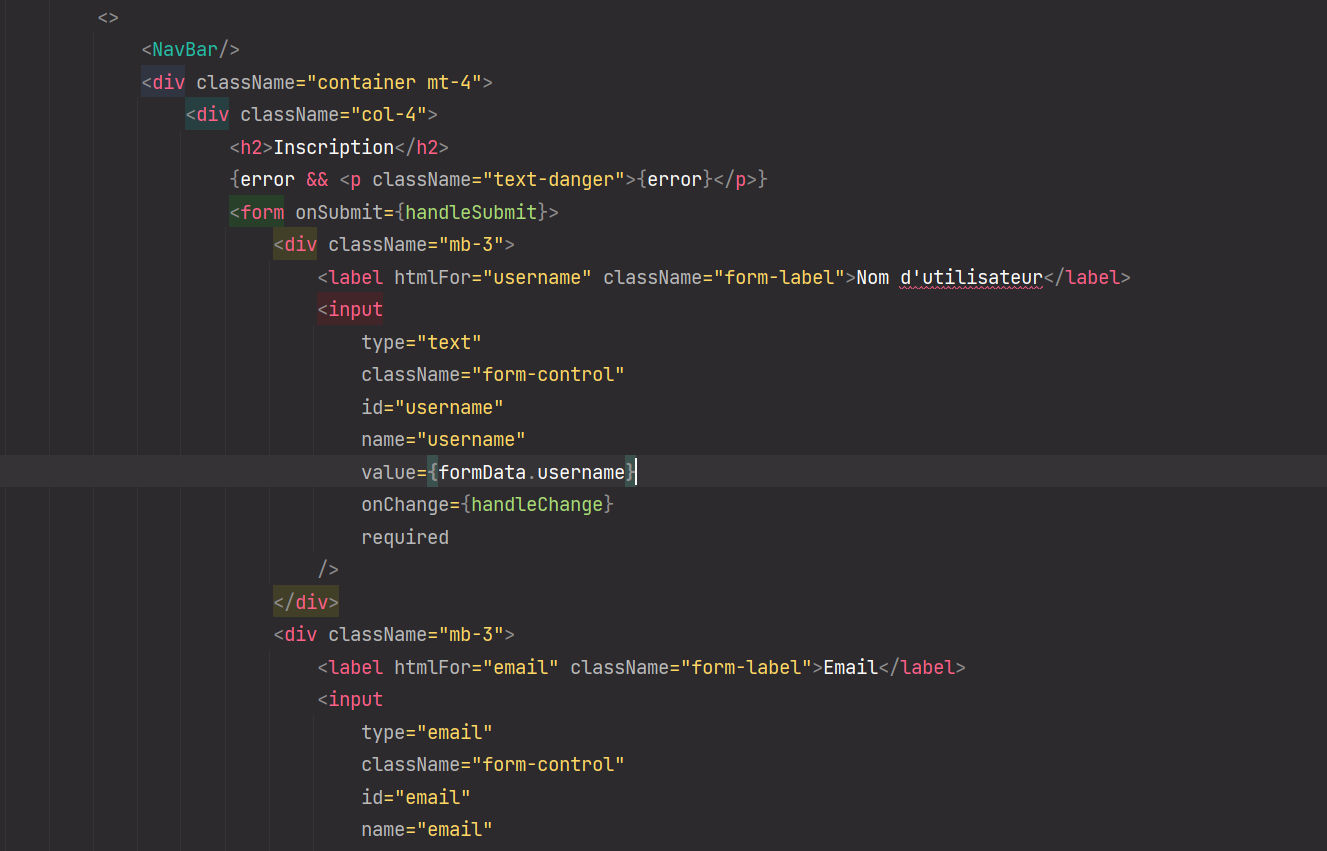
Le composant utilise le hook useState pour stocker les données du formulaire d'inscription dans l'état local de la page. Les données du formulaire sont envoyées à une API à l'aide de la bibliothèque axios lorsque l'utilisateur soumet le formulaire. Si l'inscription réussit, l'utilisateur est redirigé vers la page de connexion à l'aide du hook useNavigate. Si l'inscription échoue, un message d'erreur est affiché à l'utilisateur.

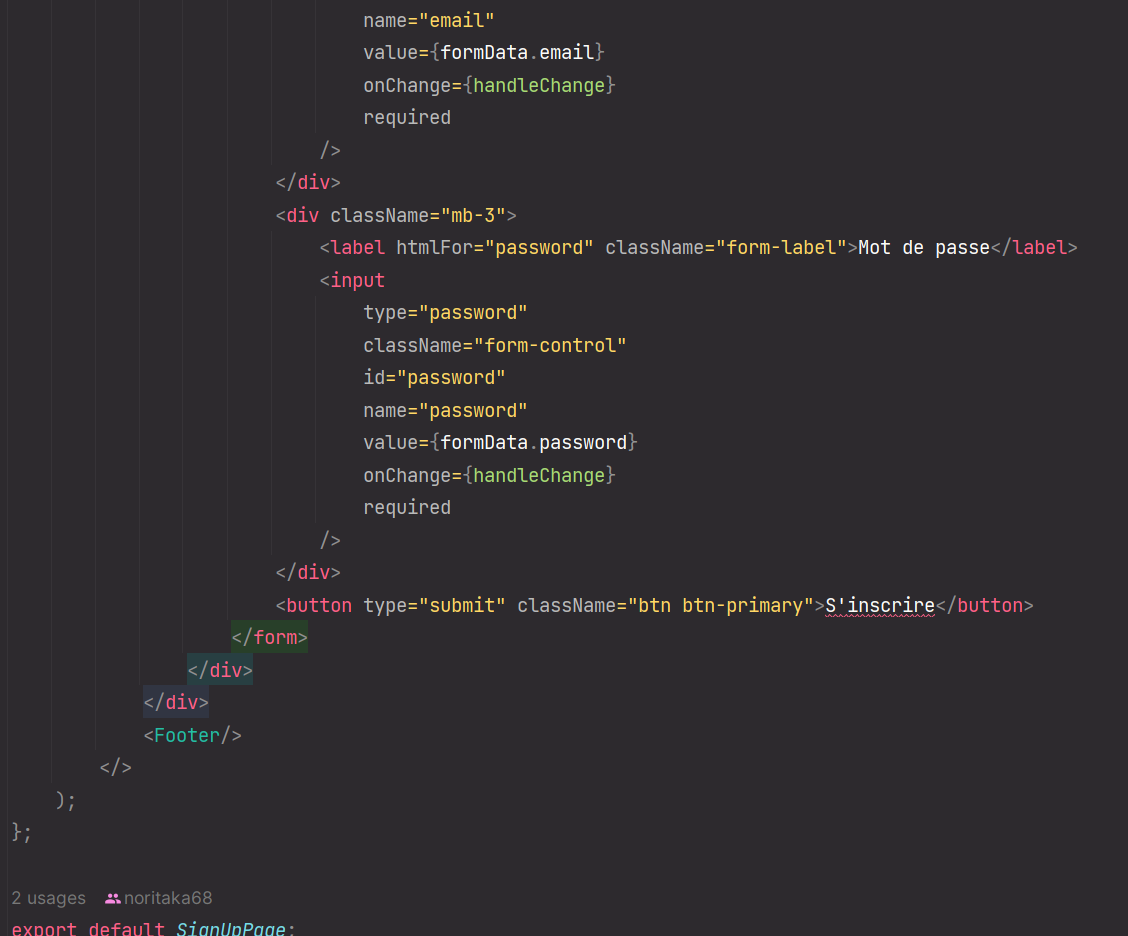
Le composant affiche également les composants NavBar et Footer pour une expérience utilisateur cohérente.

En résumé, ce composant est un formulaire d'inscription en ligne. Il permet aux utilisateurs de s'inscrire en fournissant un nom d'utilisateur, une adresse e-mail et un mot de passe, et affiche un message d'erreur en cas d'échec de l'inscription. Il utilise également des composants communs pour créer une expérience utilisateur cohérente.









6 Les Contextes :

Afin d’avoir une bonne gestion de l’état de notre application je vais utiliser l’api Contexte.

Cet outil formidable nous permet de gère les données de l’application et de pouvoir les transmettre a tout les composents de notre choix.

On peut le décrire comme un « service » .

Leurs utilisations est relativement simple. Dans un premier temps nous créons un contexte en utilisent le fonction « createContext » de React.

Cette fonction contient un objet contenant deux composants, un Provider et un Consumer.

Le composant Provider est utilisé pour encapsuler une arborescence de composants à laquelle vous souhaitez donner accès à la valeur du Context. Le Provider accepte une prop value qui sera disponible pour tous les composants enfants qui consomment ce Context.

Pour accéder à la valeur du Context dans un composant, vous pouvez utiliser le composant Consumer ou le Hook useContext. Le Hook useContext permet une consommation plus directe et bien plus simple. C’est ainsi que nous allons procéder.

* Le context AuthContexte :

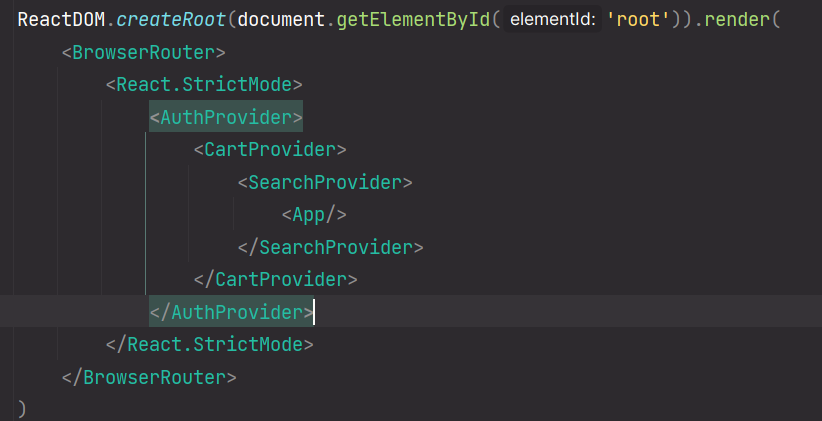
Ce composant, permet de fournir un système d'authentification qui permet de gérer l'état de connexion d'un utilisateur dans une application. Il utilise le hook useState pour stocker le jeton d'authentification dans l'état local de l'application et le hook useEffect pour mettre à jour le jeton d'authentification à partir du localStorage lorsque l'application est chargée.

Le composant fournit également des fonctions de connexion et de déconnexion pour permettre aux utilisateurs de se connecter et de se déconnecter de l'application. La fonction de connexion envoie une requête POST à une API pour obtenir un jeton d'authentification, qui est ensuite stocké dans l'état local de l'application et dans le localStorage. La fonction de déconnexion supprime le jeton d'authentification de l'état local de l'application et du localStorage.

Le composant définit également une propriété isAuthenticated pour vérifier facilement l'état de connexion de l'utilisateur dans l'application.

En utilisant ce composant, on peut facilement ajouter une fonctionnalité d'authentification à notre application React. Il suffit d'envelopper les composants qui nécessitent une authentification dans le composant AuthProvider, et d'utiliser le hook useAuth pour accéder aux fonctions de connexion et de déconnexion ainsi qu'à l'état de connexion de l'utilisateur.

On peut voir ce composant comme un garde du corps pour notre application React, protégeant les parties sensibles de l'application en vérifiant l'état de connexion de l'utilisateur et en fournissant des fonctions de connexion et de déconnexion faciles à utiliser.



* Le context CartContext :

Ce context nous permet de fournir un panier d'achat qui permet aux utilisateurs d'ajouter des produits à leur panier et de soumettre leur commande. Il utilise le hook useState pour stocker les éléments du panier dans l'état local de l'application et le hook « useEffect » pour mettre à jour le panier dans le localStorage lorsque les éléments du panier changent.

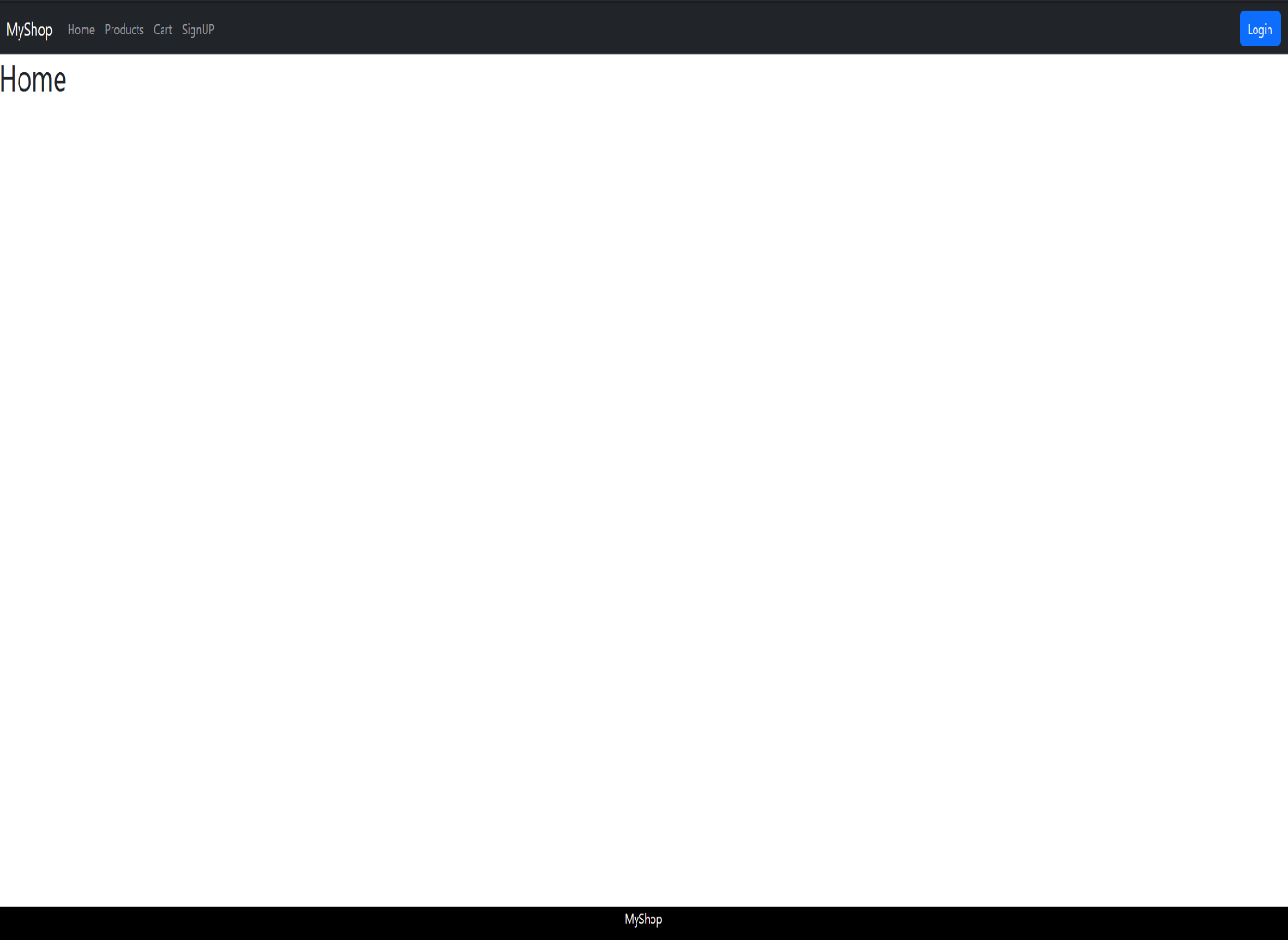
Le composant fournit également des fonctions pour ajouter des produits au panier, supprimer des produits du panier et vider complètement le panier. La fonction « addToCart » prend un objet produit et une quantité en paramètres et ajoute le produit au panier avec la quantité spécifiée. La fonction « removeFromCart » prend un identifiant de produit en paramètre et supprime le produit du panier. La fonction « clearCart » vide complètement le panier.

La fonction « submitCart » est appelée lorsque l'utilisateur soumet sa commande. Elle envoie une requête POST à une API pour ajouter chaque élément du panier au panier de l'utilisateur sur le serveur. La fonction utilise le jeton d'authentification « JWT » stocké dans le « localStorage » pour authentifier la requête. Une fois que tous les éléments ont été ajoutés avec succès, le panier est vidé et un message de succès est affiché à l'utilisateur.

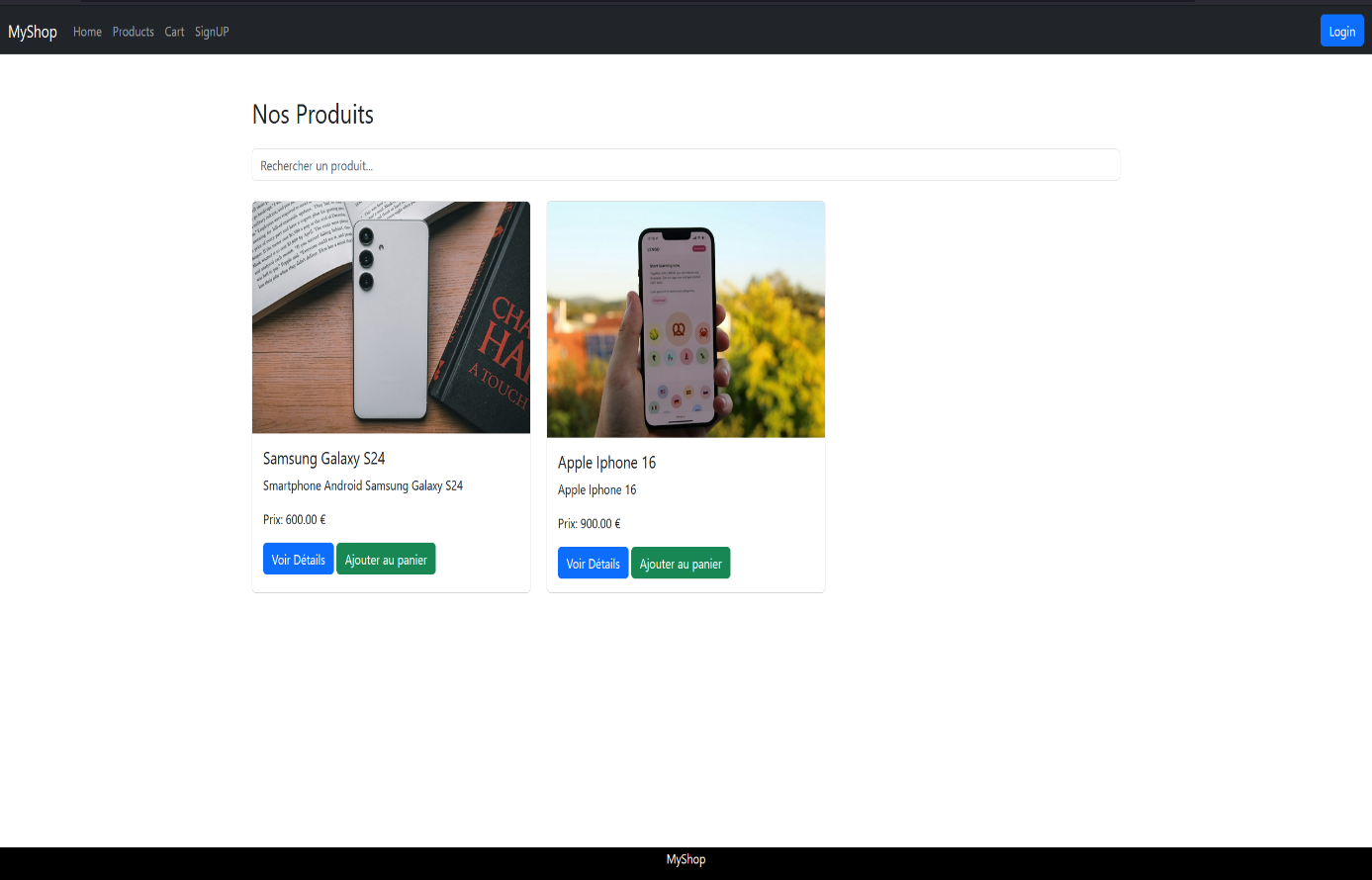
En utilisant ce composant, on peut facilement ajouter un panier d'achat à application React. Il suffit d'envelopper les composants qui nécessitent un panier dans le composant CartProvider, et d'utiliser le hook useCart pour accéder aux fonctions d'ajout, de suppression et de soumission du panier ainsi qu'aux éléments du panier.

**Annexes**

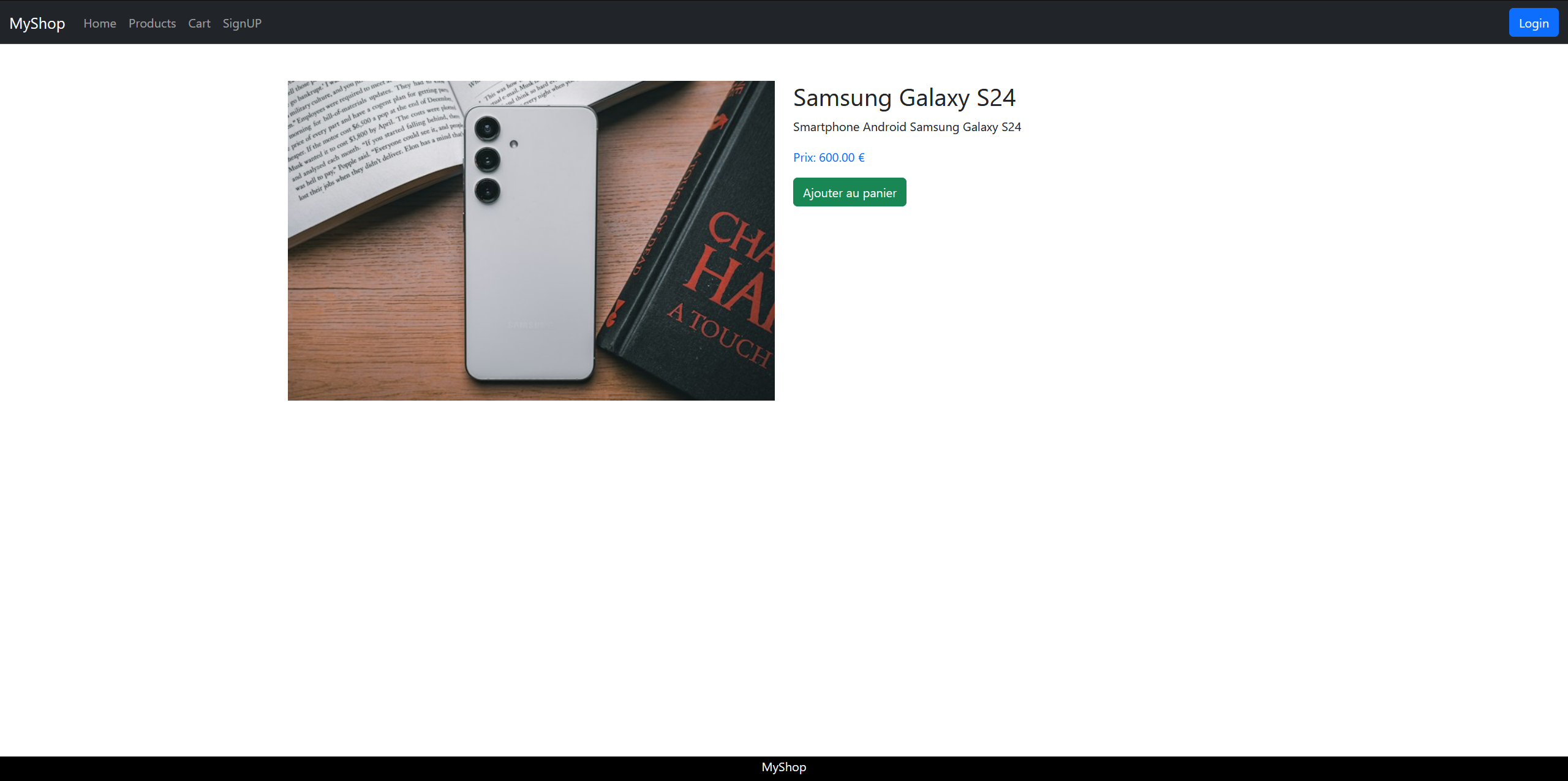
**HomePage**

****

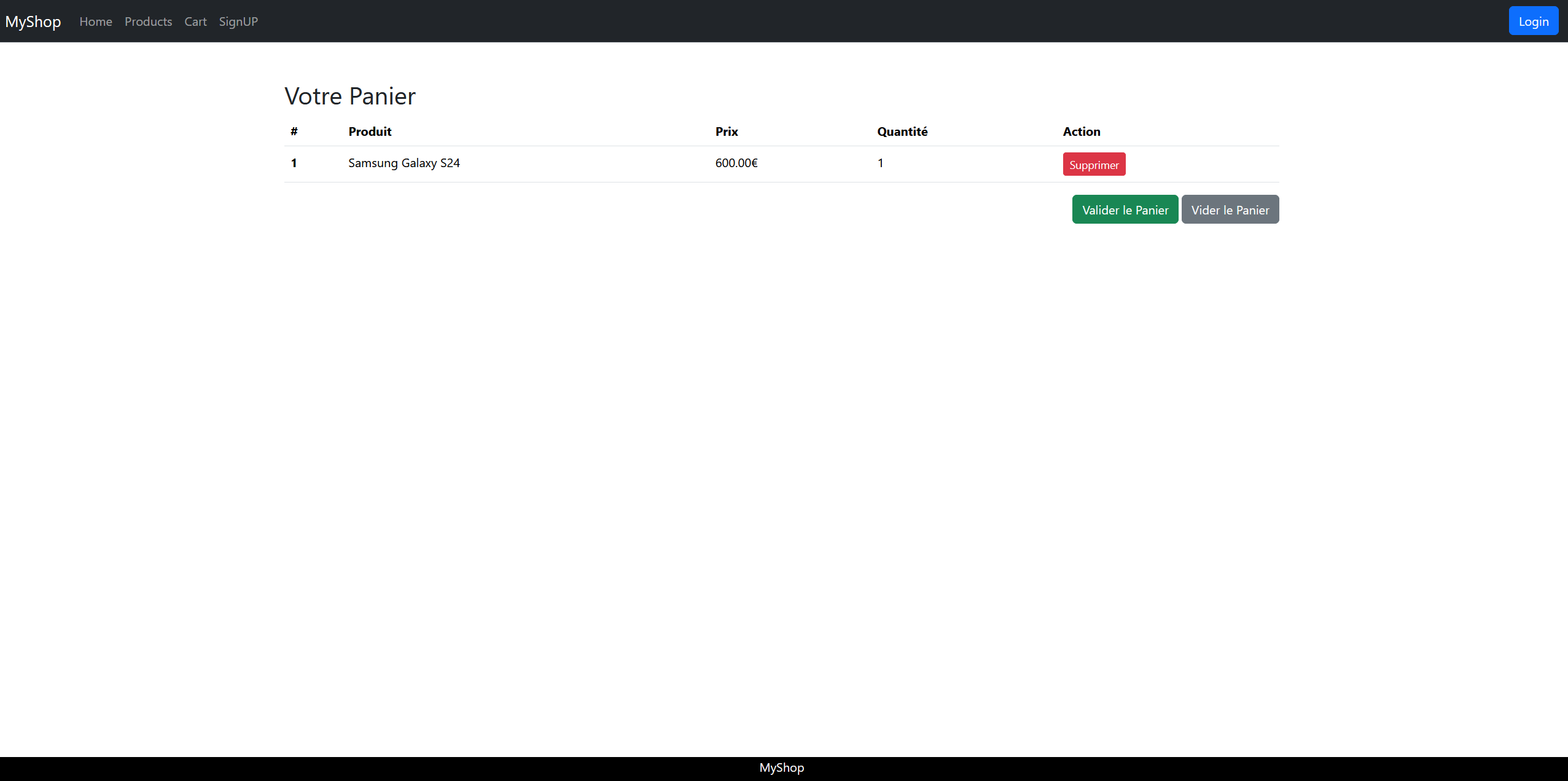
**ProductPage :**

****

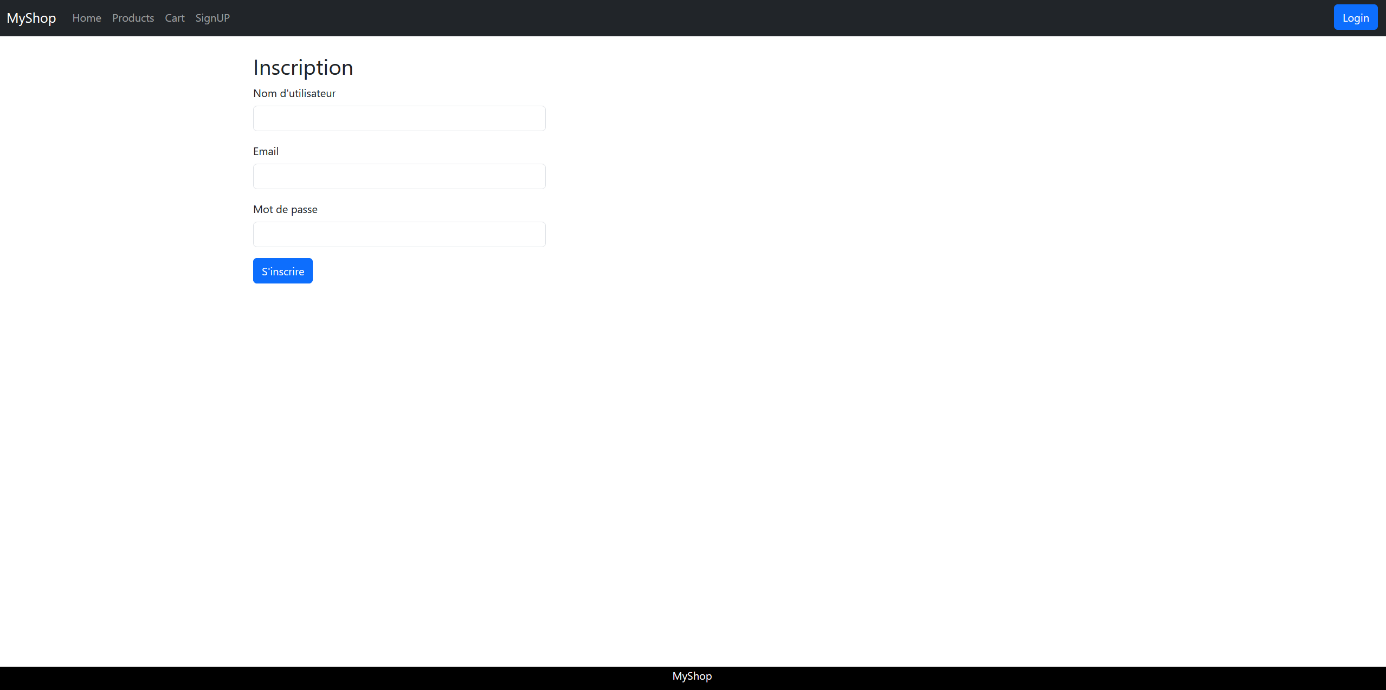
**Details Page :**

****

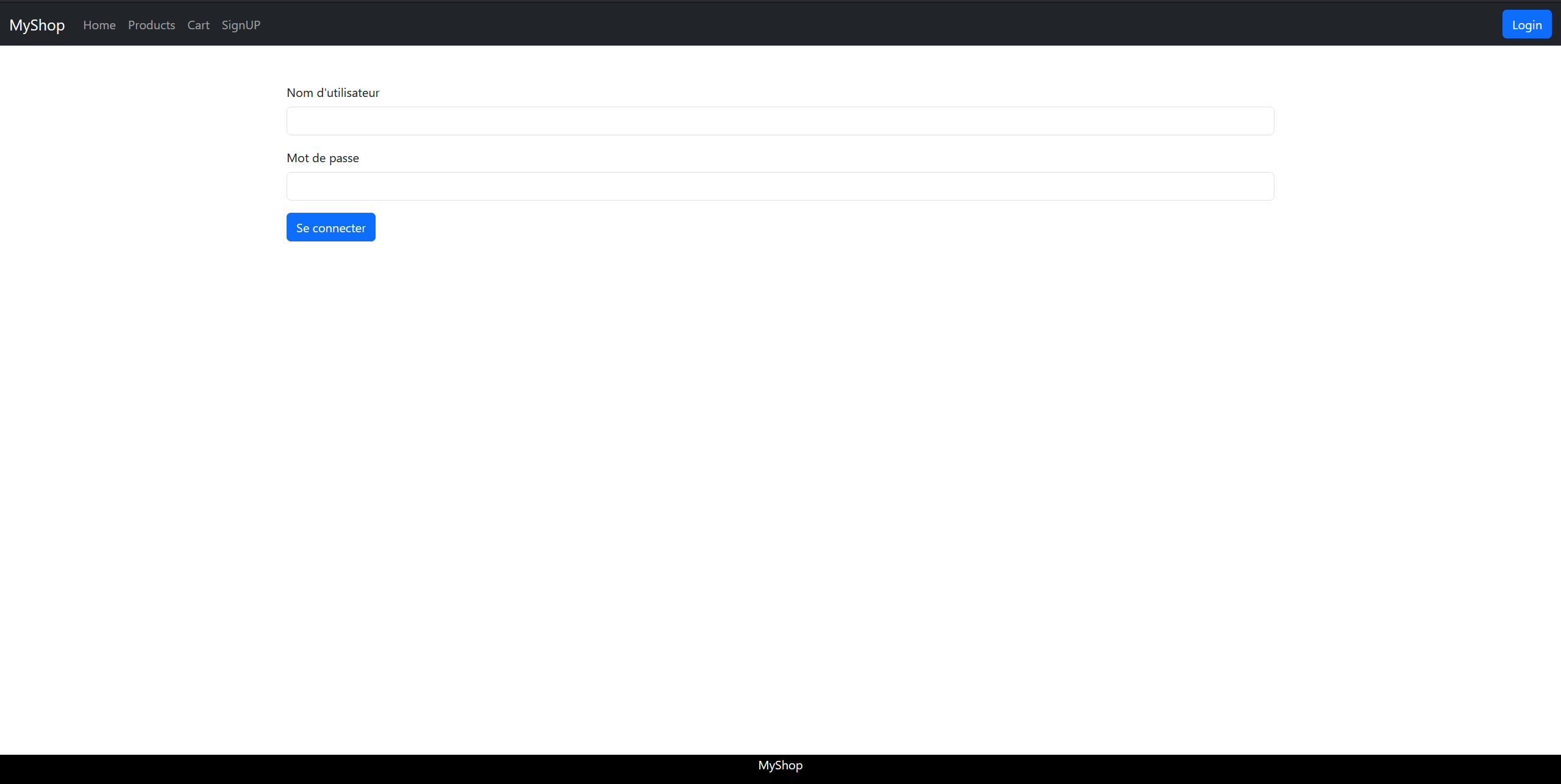
**Cart Page :**

****

**Sign up Page :**

****

**Login Page :**

****