Dossier Professionnel

Concepteur Développeur d’Application

Session : 2022-2023

Candidat : Saïd Chamchi

Tuteur Pédagogique : Mathieu Capon

Tuteur Technique : Jean-François Beachler



**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 a 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-Frederic 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

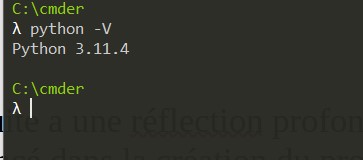
Suite à une réflexion, et la réalisation des digrammes, j’ai pu me lancer dans la création du projet.

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 ?

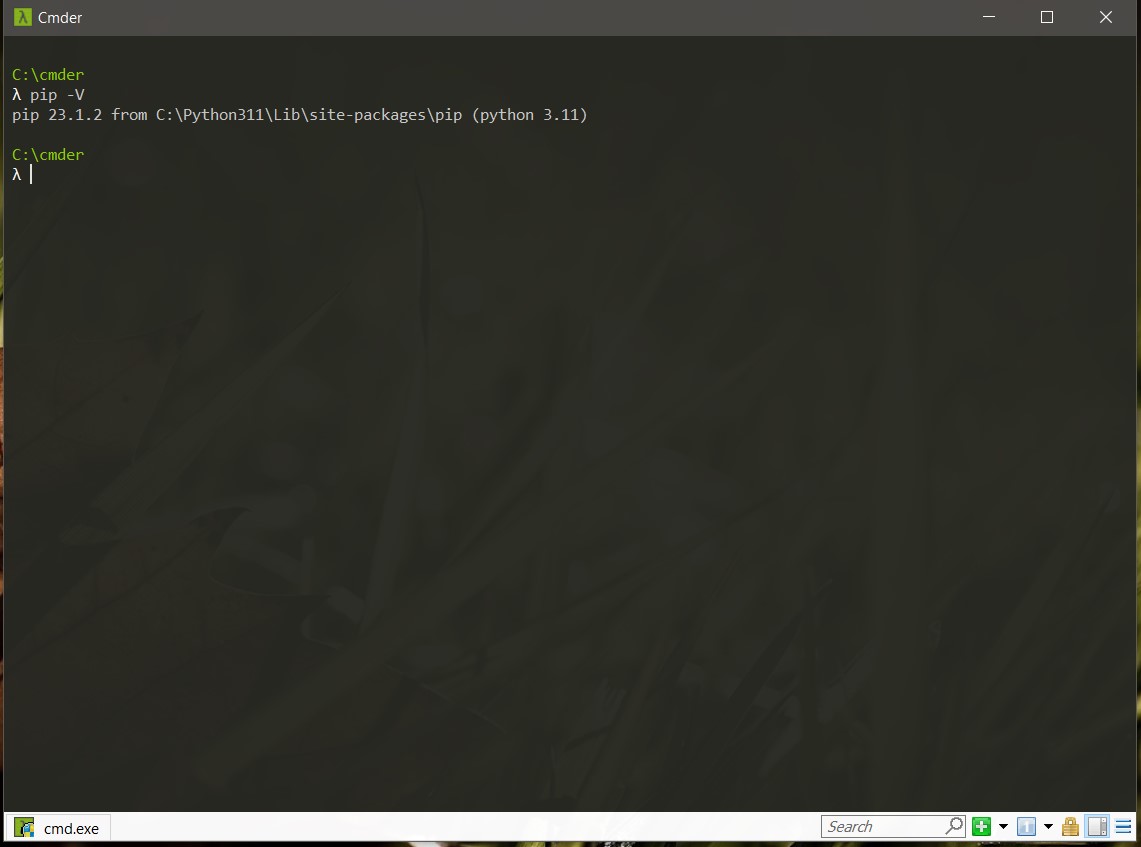
1 - Un environnement virtuel permet de créer un environnement complètement isolé du système

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.

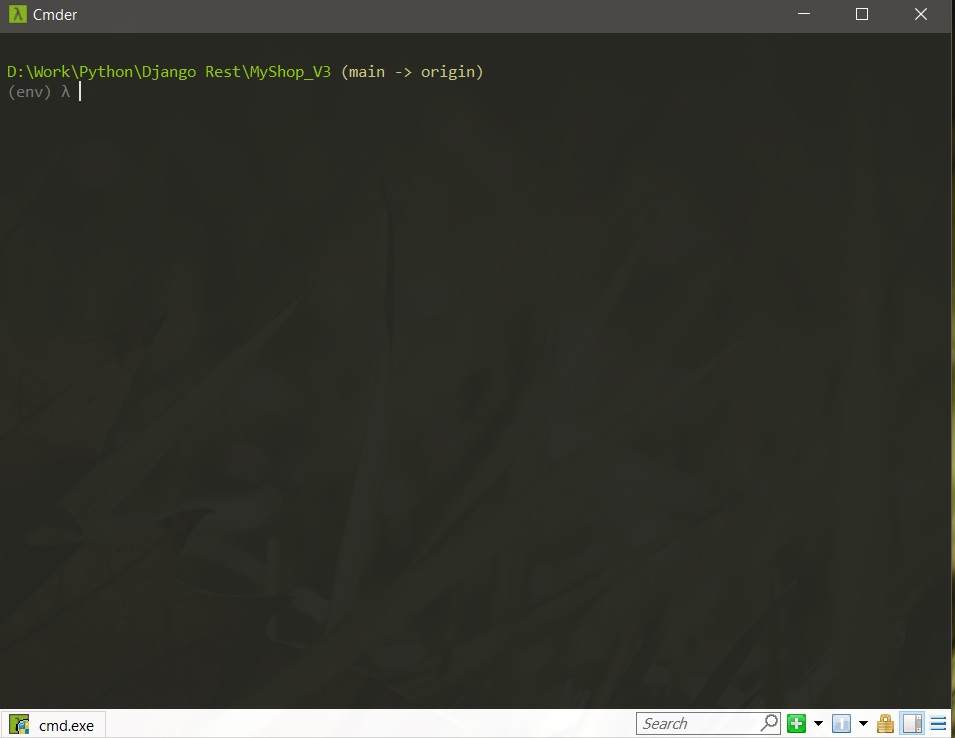
Après ces différentes versifications, je créais mon environnement virtuel avec la commande « python -m venv env ».

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.

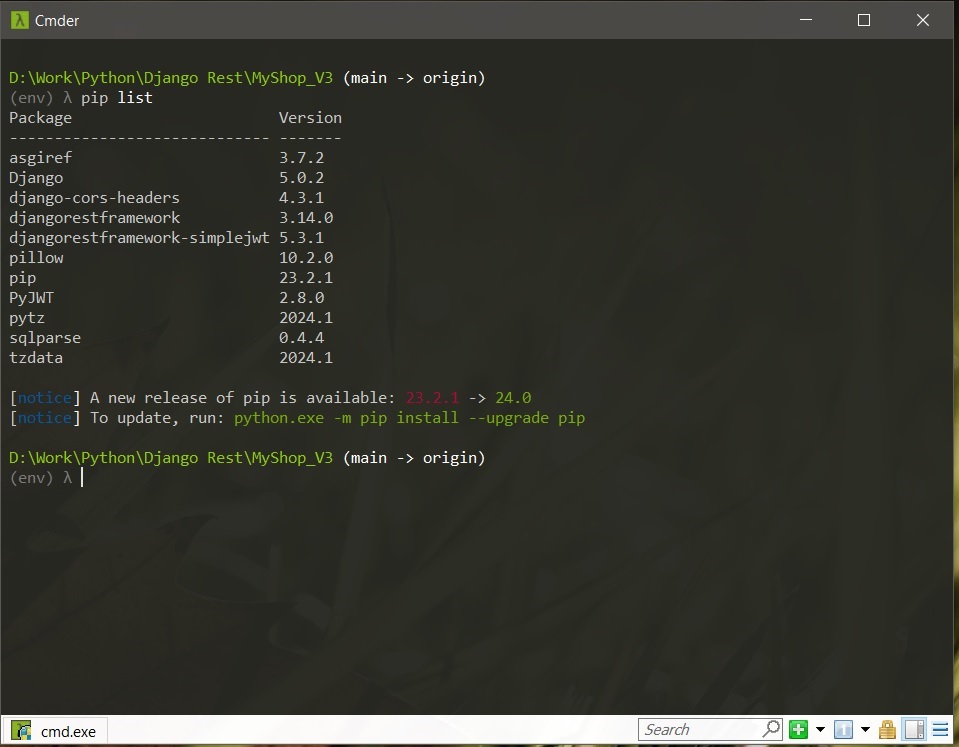
- Apres l’installation de « Pillow » j’installe Django et Django Rest Framework. Il est important de remarque que Django et Django Rest Framework sont de dépendances différentes. Pour ce faire je vais utiliser la commande «pip install Django » et « pip install djangorestframework ». Dans un second temps 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, 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 Language 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 sont 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

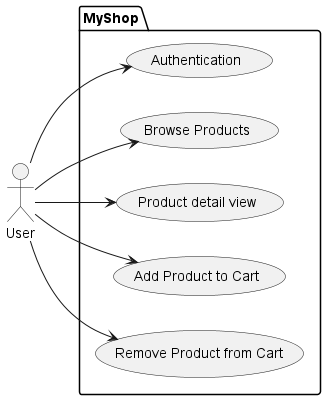
- Connexion

- Interface de gestion

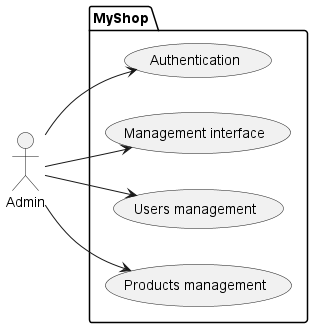
- Gestion des Utilisateurs

- Gestion des Produits

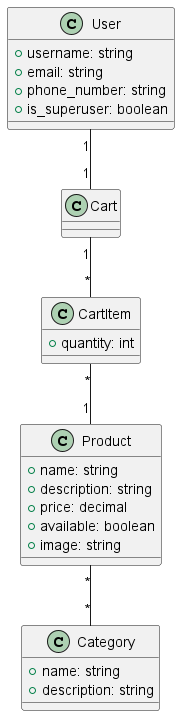
Diagramme « Use Case Utilisateur » :



**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 éléments, mais chaque élément 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.

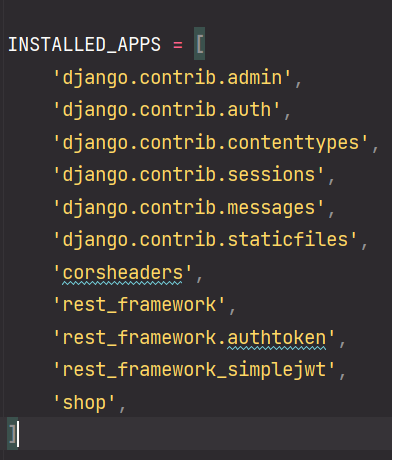
**Configuration du Fichier Setting**

Afin d’assure 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 differents parametre 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 », « 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 ton projet. Ils traitent les requêtes et les réponses.

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 médias.

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 qui la consomme 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. Elle ajoute les champs suivants :

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

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

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

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

phone\_number : Un champ 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 CharField pour le nom de la catégorie. Il peut contenir jusqu'à 255 caractères.

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

description : Un champ TextField pour la description du produit.

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

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

image : Un champ 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 entités 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.

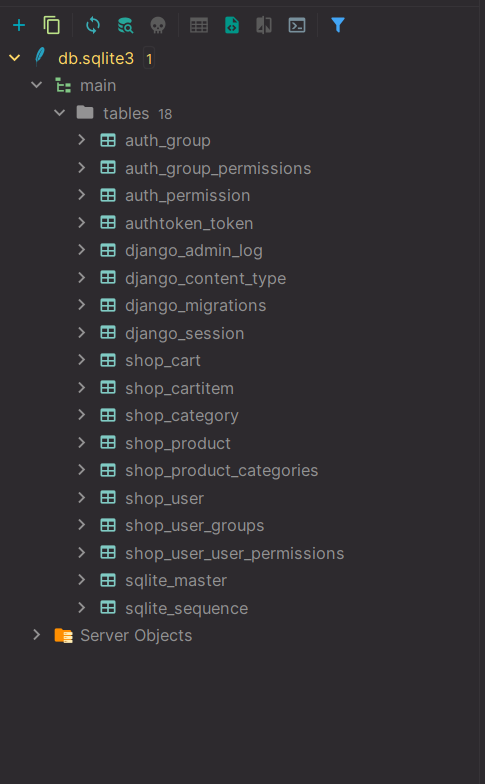
Après cela je vais pouvoir créer les tables de dans ma base de données, avec la commande :

**« 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 qu'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 precedement ce fichier a pour bus principal de configure l’interface d’administraion de Django.

Pour ce faire il a besoin de plusieurs elements.

1. Dans un premier temps je dois importé 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.

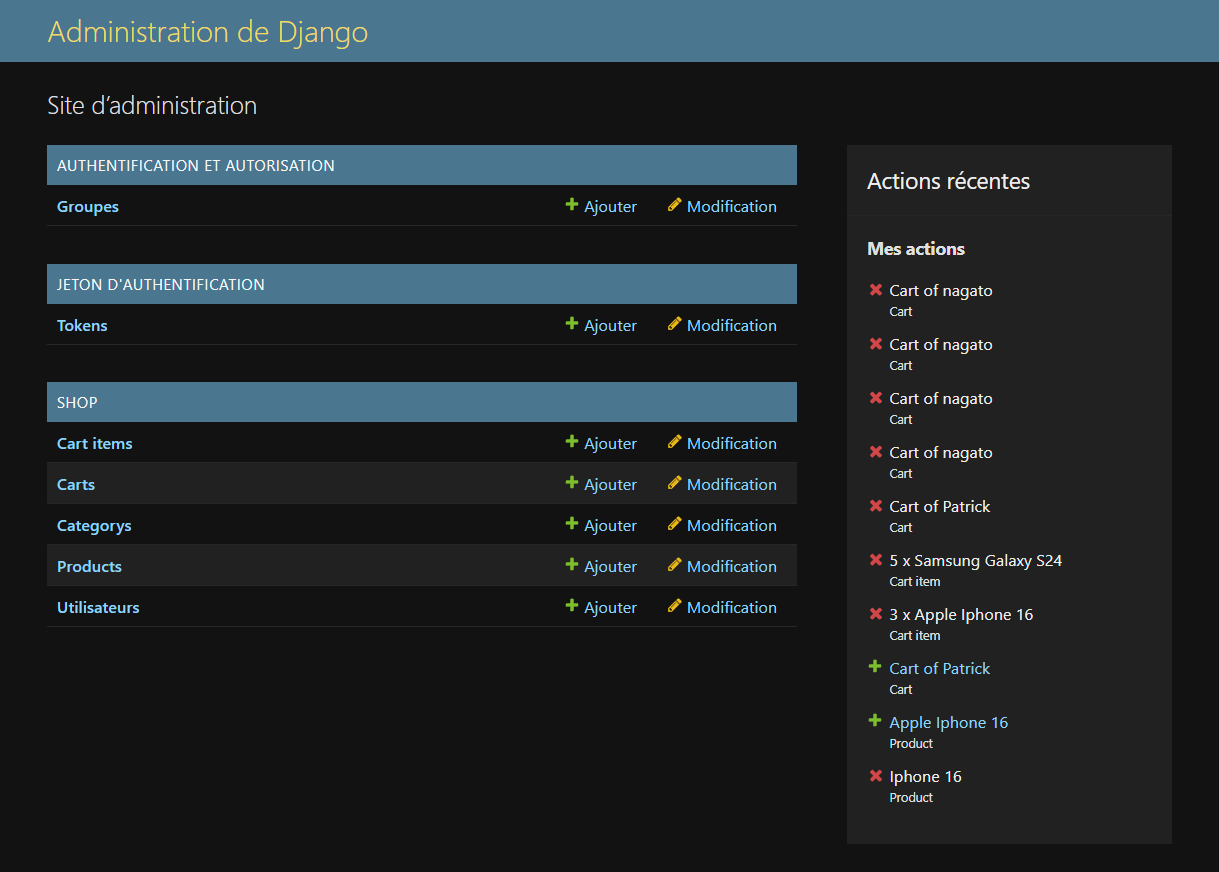
admin.site.register(CartItem, CartItemAdmin):

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

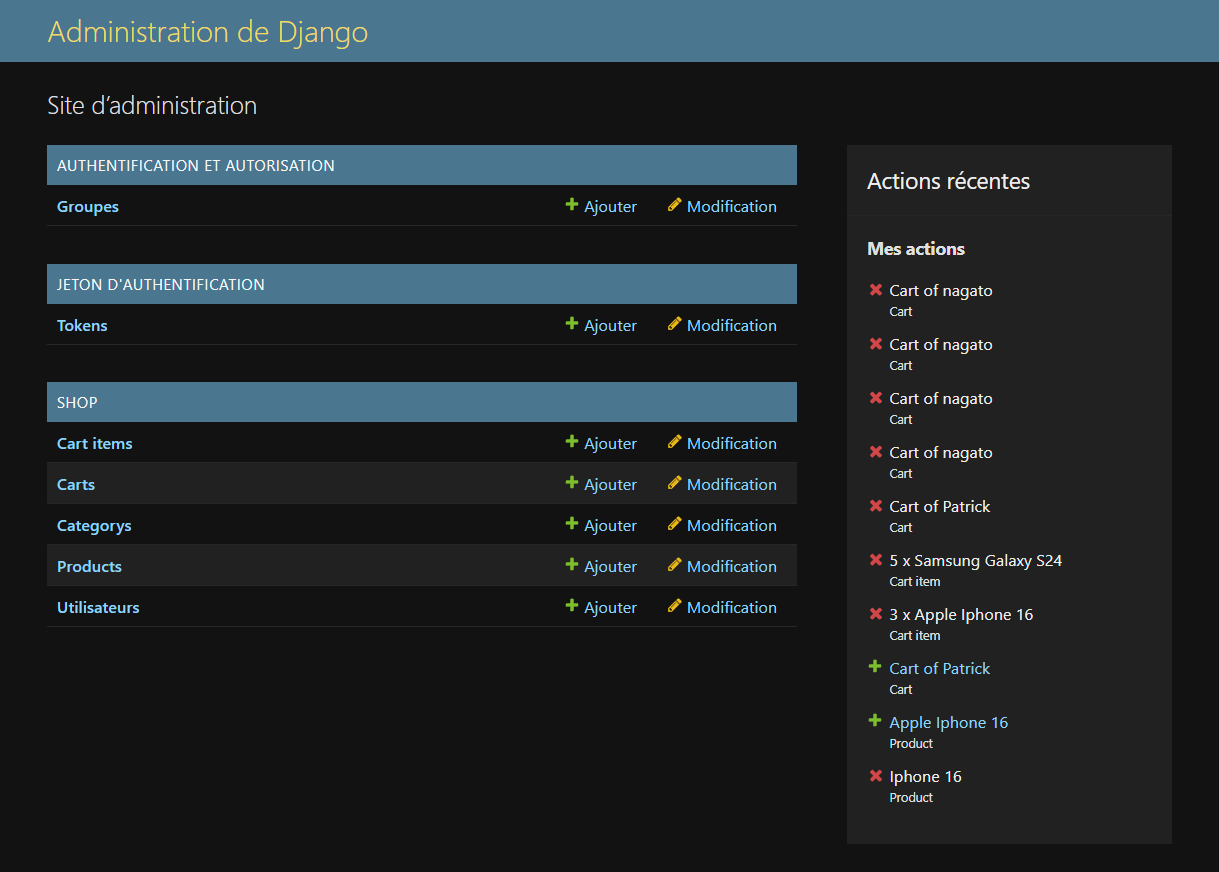
Chaque classe d'administration a des attributs tels que list\_display, list\_filter, search\_fields, etc., qui sont utilisés pour personnaliser l'affichage et le comportement des modèles dans l'interface d'administration. Par exemple, list\_display spécifie les champs à afficher dans la liste des objets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

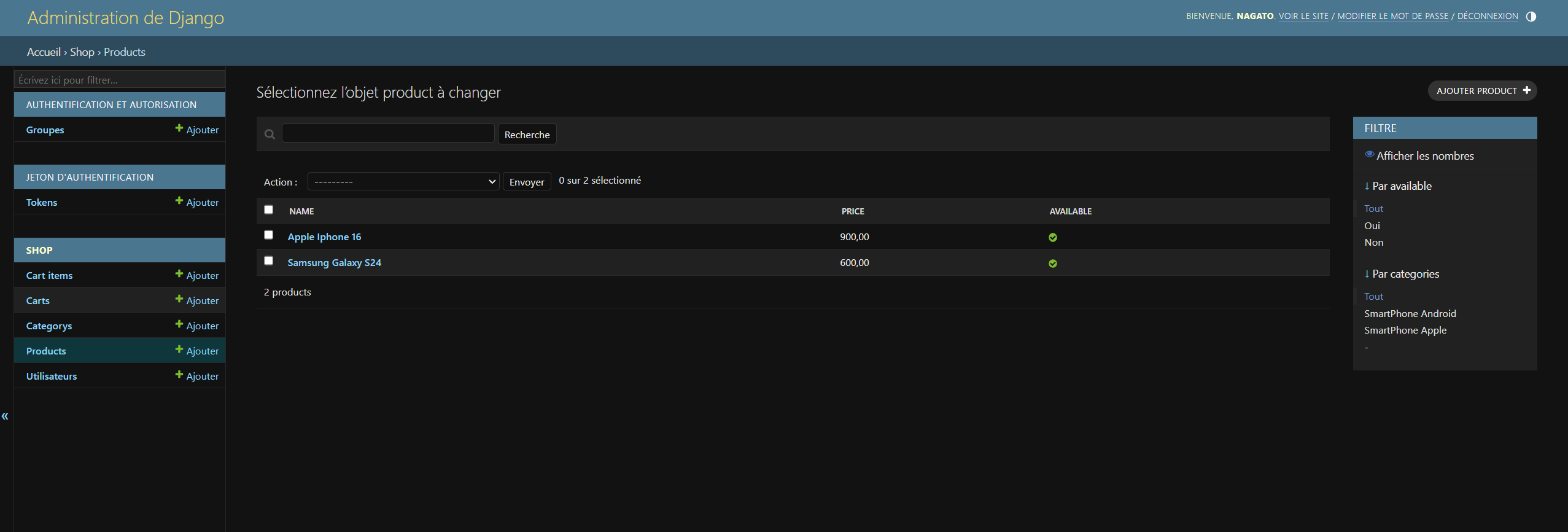
Après cela, j'aurai accès à mes différentes entités directement depuis le panel administrateur.

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 a la création des views.

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

(Models, Views, Templates) similaire au Designe 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 décorateurs « @apiview ».

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

Comme dans les précédents fichier, 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 fais le tour de l’ensemble de nos imports . Nous pouvons coder nos controlleurs

1. **Controlleur user\_details :**

Cette vue 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. **Controlleur « user\_list » :**

Cette vue 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. **Controlleur « 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.