



Groupe 6 : Salma Desmazieres et Mohamed Djebali

## I. INTRODUCTION :

Réalisation d'un système de gestion d'inventaire pour un entrepôt ou un magasin qui permet de gérer les produits, les quantités disponibles, les ajouts et les retraits de stocks ainsi que la génération de rapports

## II. Fonctionnalités à implémenter :

- Interface en ligne de commande avec menus (afficher l'inventaire, ajouter/supprimer des produits, mettre à jour les quantités).
- Sauvegarde de l'inventaire dans un fichier CSV ou JSON.
- Système d'alertes en cas de stock faible ou critique.
- Gestion des erreurs liées aux saisies utilisateur et aux fichiers.
- Génération de rapports (par exemple : produits en rupture, produits les plus vendus).

### Aspects avancés :

- Implémentation d'une gestion multi-utilisateurs avec des rôles (administrateur et employé).
- Système de suivi en temps réel avec une mise à jour automatique des stocks en fonction des ventes et des approvisionnements simulés.

## III. Conception de la base de données :

### a. Règles de gestion :

Un employé peut afficher toutes les informations d'un produit.

\* Un employé gère 0 ou plusieurs produits et un produit est géré par 1 ou plusieurs employés

Un produit a un id, un nom, une référence et une quantité

\* Un produit est lié à 0 ou plusieurs stocks

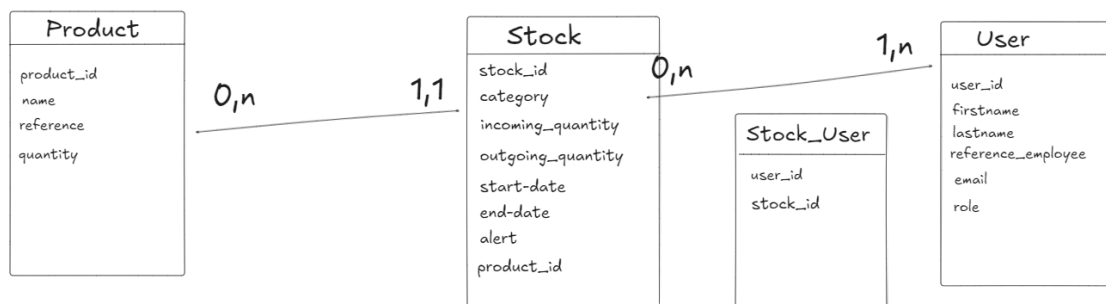
Un stock a un id, une catégorie, une date d'entrée dans le stock, une date de sortie du stock, une date d'entrée, une date de sortie et une alerte en cas de quantité basse.

b. Dictionnaires des données

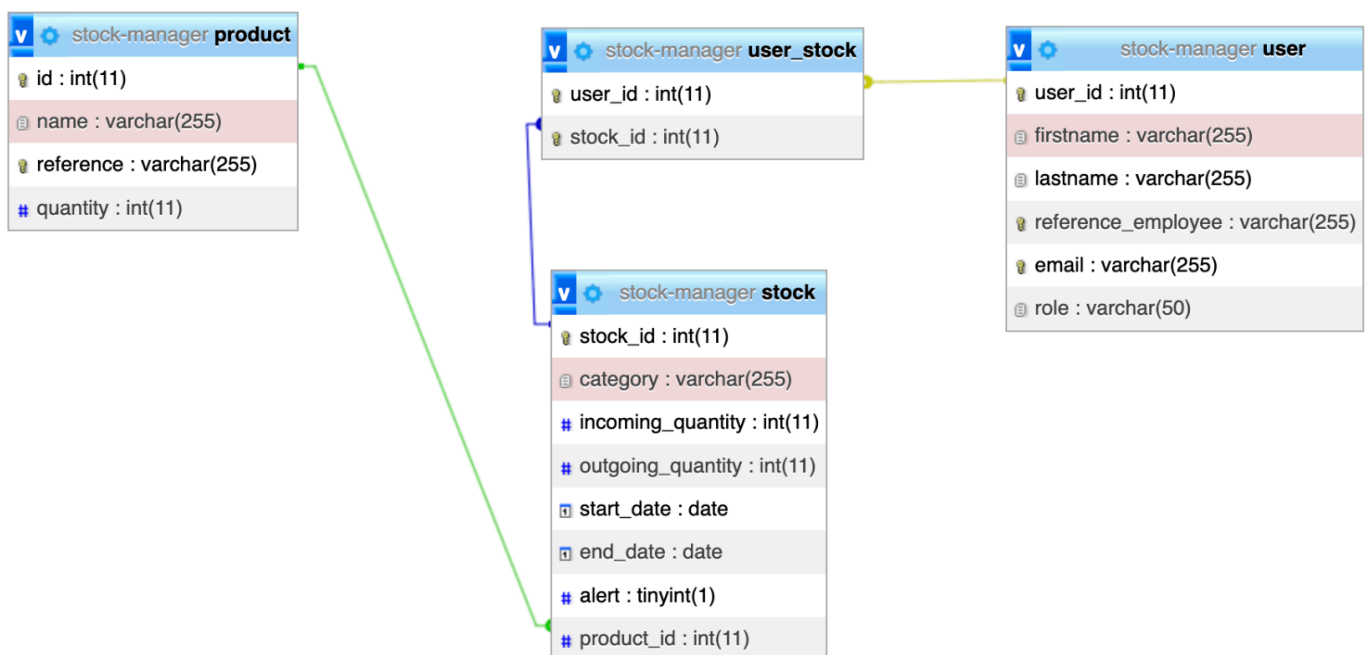
Code mnémonique	Description	Type de données	Table	Commentaire
product_id		numérique	Product	
name	nom du produit	texte	Product	
reference	référence de chaque produit	alphanumérique	Product	
quantity	quantité restante dans le stock	numérique	Product	
stock_id	id	numérique	Stock	
category	nom du stock	texte	Stock	
incoming_quantity	quantité de produit entrant dans le stock		Stock	
outgoing_quantity	quantité de produit sortant du stock		Stock	
start-date	date d'entrée du produit dans le stock		Stock	
end-date	date de sortie du produit dans le stock		Stock	
alert	seuil de stock minimum qui déclenche l'alerte	Numérique	Stock	
user_id		Numérique	user	

firstname	prénom de l'employé	texte	user	
lastname	nom	texte	user	
reference_employee			user	
email	mail de l'employé	texte	user	
rôle	employé ou administrateur	texte	user	

c. MCD :



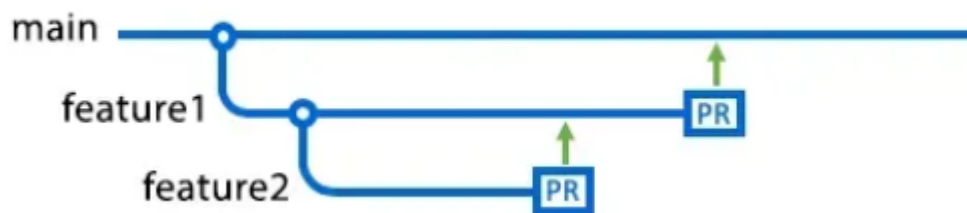
d. MPD :



#### IV. Organisation :

Code hébergé dans un dépôt Github à cette adresse :

<https://github.com/Nouster/stock-manager>



[le versionning](#)

#### V. Lancement du projet et techniques :

##### a. Mode d'utilisation :

- Pré-requis :  
Avoir "python >= 3.12" et une base de données "mysql"
- Cloner le projet
- Se positionner dans le dossier du projet
- Installer les dépendances :  
pip install
- Ajouter les accès à la base de données :  
Dans le fichier .env, changer les valeurs de "user", "password" et "port" dans le fichier d'environnement
- Lancer le projet :

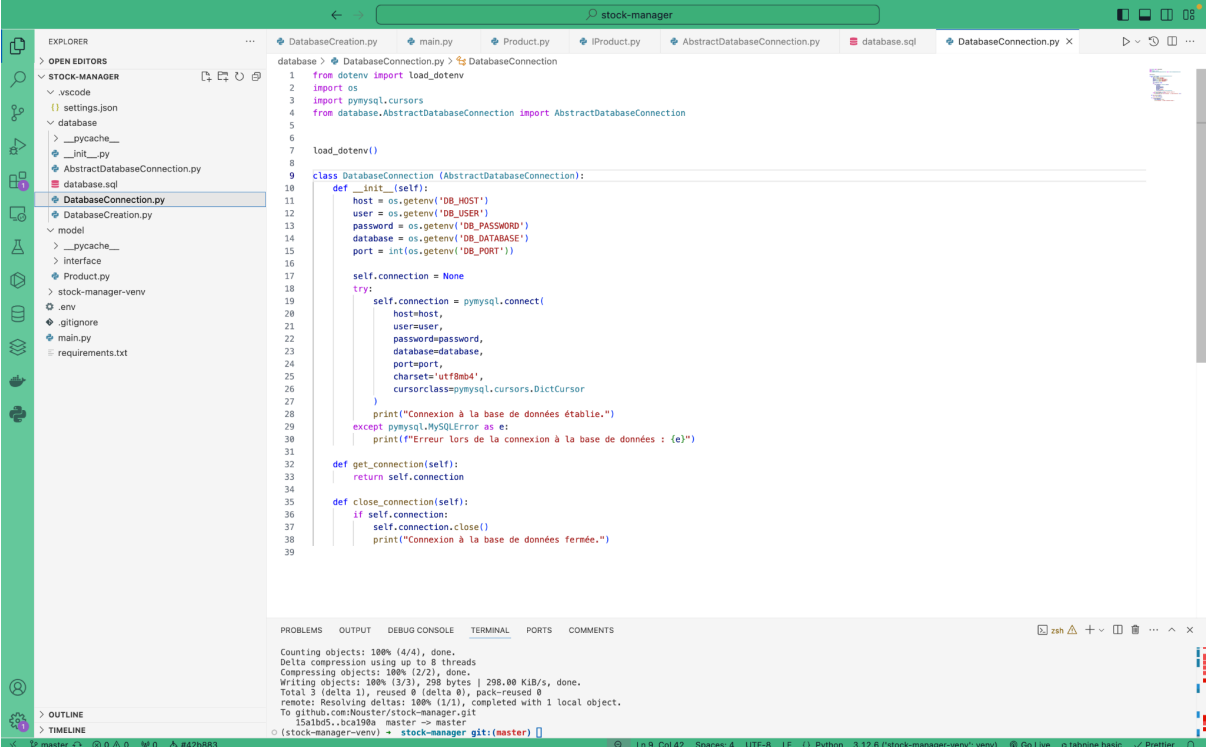
Sur Widows : python main.py

Sur système Unix : python3 main.py

b. Code :

## 1/ Établir une connexion avec notre serveur de base de données

Afin d'être raccord avec notre modèle de conception, nous avons décidé d'abandonner l'idée d'exploiter nos données via un fichier Json. En effet, nous avons un modèle relationnel entre nos différentes tables et nous avons opté pour l'intégration d'un client MySql dans notre projet.

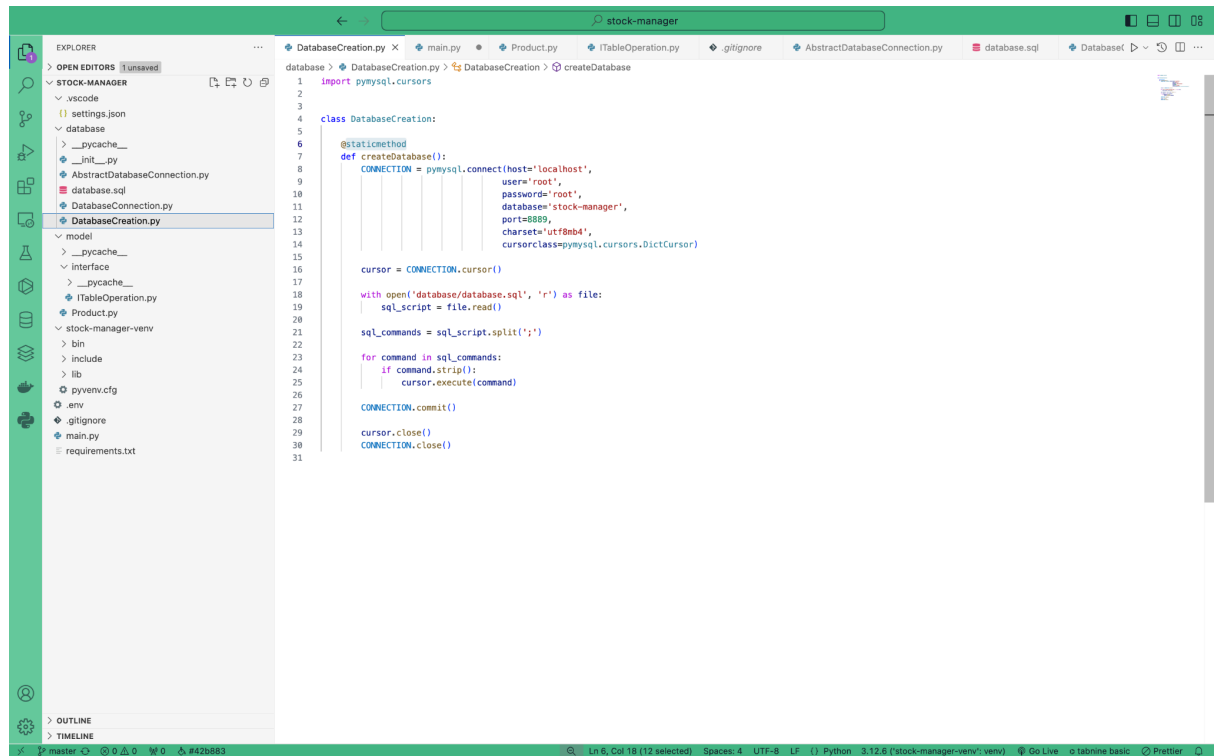


```
1 from dotenv import load_dotenv
2 import os
3 import pymysql.cursors
4 from database.AbstractDatabaseConnection import AbstractDatabaseConnection
5
6
7 load_dotenv()
8
9 class DatabaseConnection (AbstractDatabaseConnection):
10     def __init__(self):
11         host = os.getenv('DB_HOST')
12         user = os.getenv('DB_USER')
13         password = os.getenv('DB_PASSWORD')
14         database = os.getenv('DB_DATABASE')
15         port = int(os.getenv('DB_PORT'))
16
17         self.connection = None
18         try:
19             self.connection = pymysql.connect(
20                 host=host,
21                 user=user,
22                 password=password,
23                 database=database,
24                 port=port,
25                 charset='utf8mb4',
26                 cursorclass=pymysql.cursors.DictCursor
27             )
28             print("Connexion à la base de données établie.")
29         except pymysql.MySQLError as e:
30             print(f"Erreur lors de la connexion à la base de données : {e}")
31
32     def get_connection(self):
33         return self.connection
34
35     def close_connection(self):
36         if self.connection:
37             self.connection.close()
38             print("Connexion à la base de données fermée.")
39
```

## 2/ Création des tables

Salma et moi avons généré un fichier SQL contenant toutes les requêtes nécessaires afin d'être exécutées en BDD. Dans la classe DatabaseCreation, nous allons lire ce fichier et stocker chacune des commandes (après le point virgule) dans un tableau. Grâce à une structure de contrôle nous allons parcourir ce tableau et grâce au cursor, exécuter ces différentes entrées. Ce cursor ( que nous découvrons) semble se comporter de la même manière que PDO (en PHP) et son query.

⚠ Notre méthode est statique. Nous n'avons nullement besoin d'une instance d'objet pour cela.



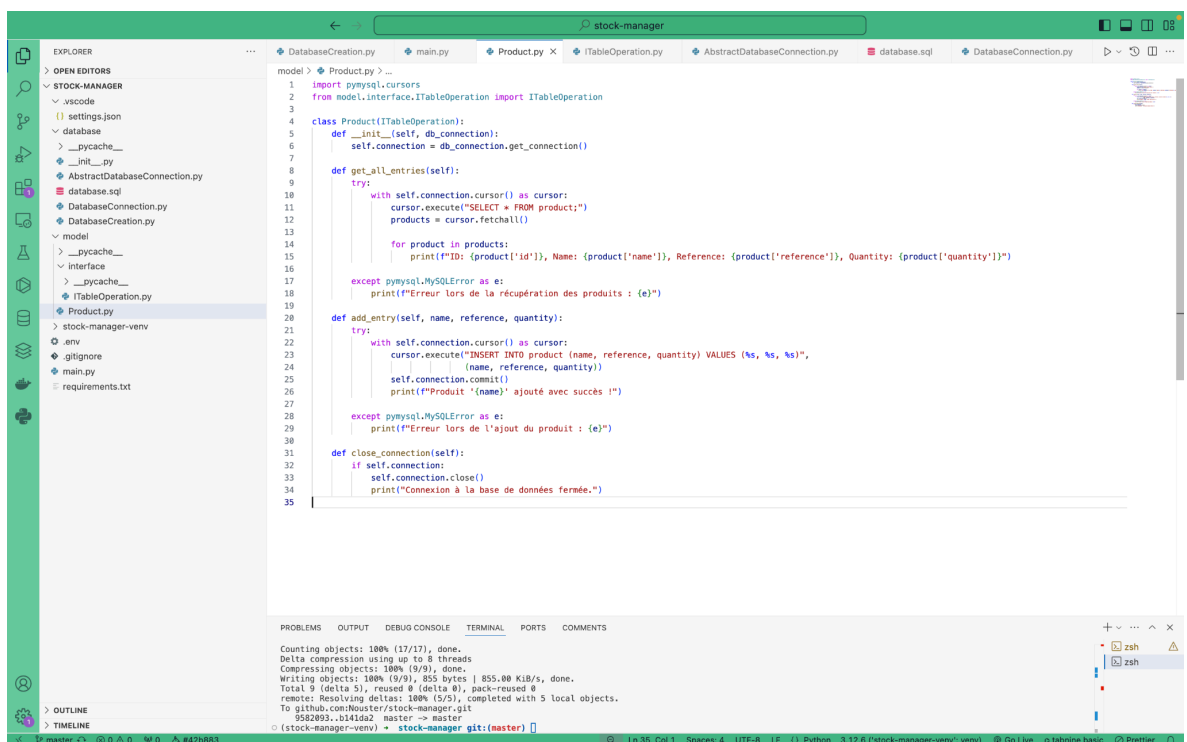
The screenshot shows a VS Code editor window titled 'stock-manager'. The Explorer sidebar on the left shows a project structure with folders like 'stock-manager' and 'model', and files like 'DatabaseCreation.py'. The main editor area displays the code for 'DatabaseCreation.py'. The code defines a class 'DatabaseCreation' with a static method 'createDatabase()'. The method uses 'pymysql' to connect to a database and execute SQL commands from a file.

```
1 import pymysql.cursors
2
3
4 class DatabaseCreation:
5
6     @staticmethod
7     def createDatabase():
8         CONNECTION = pymysql.connect(host='localhost',
9                                     user='root',
10                                    password='root',
11                                    database='stock-manager',
12                                    port=8889,
13                                    charset='utf8mb4',
14                                    cursorclass=pymysql.cursors.DictCursor)
15
16         cursor = CONNECTION.cursor()
17
18         with open('database/database.sql', 'r') as file:
19             sql_script = file.read()
20
21         sql_commands = sql_script.split(';')
22
23         for command in sql_commands:
24             if command.strip():
25                 cursor.execute(command)
26
27         CONNECTION.commit()
28
29         cursor.close()
30         CONNECTION.close()
31
```

### 3/ Découplage du code associé aux fonctionnalités

Au début de notre phase de développement, nous avons l'essentiel de notre code (**programmation fonctionnelle**) contenu dans un seul fichier. Au fur et à mesure de l'avancement, nous nous sommes aperçus qu'il serait difficile pour nous **d'implémenter de nouvelles fonctionnalités** au vu du **couplage fort**.

Nous débutons sur Python mais nous avons entrepris de nous appuyer sur les piliers fondamentaux de la **POO** et de certains principes **SOLID**. Cela nous permettra de développer plus **aisément de nouvelles fonctionnalités** par la suite.



```
1 import pymysql.cursors
2 from model.interface.ITableOperation import ITableOperation
3
4 class Product(ITableOperation):
5     def __init__(self, db_connection):
6         self.connection = db_connection.get_connection()
7
8     def get_all_entries(self):
9         try:
10             with self.connection.cursor() as cursor:
11                 cursor.execute("SELECT * FROM product;")
12                 products = cursor.fetchall()
13
14                 for product in products:
15                     print(f"ID: {product['id']}, Name: {product['name']}, Reference: {product['reference']}, Quantity: {product['quantity']}")
16
17         except pymysql.MySQLError as e:
18             print(f"Erreur lors de la récupération des produits : {e}")
19
20     def add_entry(self, name, reference, quantity):
21         try:
22             with self.connection.cursor() as cursor:
23                 cursor.execute("INSERT INTO product (name, reference, quantity) VALUES (%s, %s, %s)",
24                               (name, reference, quantity))
25                 self.connection.commit()
26                 print(f"Produit '{name}' ajouté avec succès !")
27
28         except pymysql.MySQLError as e:
29             print(f"Erreur lors de l'ajout du produit : {e}")
30
31     def close_connection(self):
32         if self.connection:
33             self.connection.close()
34             print("Connexion à la base de données fermée.")
35
```

Parmi les principes de la POO, nous avons essayé de mettre en place **l'abstraction, l'héritage et le polymorphisme**.

```

DatabaseCreation.py | main.py | Product.py | ITableOperation.py | AbstractDatabase
database > AbstractDatabaseConnection.py > ...
1  from abc import ABC, abstractmethod
2
3  class AbstractDatabaseConnection(ABC):
4      @abstractmethod
5      def get_connection(self):
6          pass
7
8      @abstractmethod
9      def close_connection(self):
10         pass
11

```

Les méthodes affichées dans cette l'interface ci-dessous sont des méthodes qui vont être implémentées dans les classes qui vont exécuter des requêtes et ces dernières auront donc un contrat d'implémentation qui les forceront à fournir un corps aux différentes méthodes.

```

model > interface > ITableOperation.py > ITableOperation > close_connection
Nouster, il y a 35 minutes | 1 author (Nouster)
1
2  from abc import ABC, abstractmethod
3
4  class ITableOperation(ABC):
5
6      @abstractmethod
7      def get_all_entries(self):
8          pass
9
10     @abstractmethod
11     def add_entry(self, name, reference, quantity):
12         pass
13
14     @abstractmethod
15     def close_connection(self):
16         pass

```

Nouster, il y a 6 heures • refactor(product) new interface for tables

### ITableOperation.py

```

import pymysql.cursors
from model.interface.ITableOperation import ITableOperation

```

```

Nouster, il y a 37 minutes | 1 author (Nouster)
class Product(ITableOperation):
    def __init__(self, db_connection):
        self.connection = db_connection.get_connection()

    def get_all_entries(self):
        try:
            with self.connection.cursor() as cursor:
                cursor.execute("SELECT * FROM product;")
                products = cursor.fetchall()

            for product in products:

```



La classe ci-dessous établit une **connexion** avec la **base de données** grâce à notre client **PyMySQL**. Nous lui fournissons les différentes informations qui proviennent d'un **fichier d'environnement** situé à la racine de notre projet. Nous pouvons aussi fournir un paramètre par défaut à la méthode `getenv()` au cas où nous n'aurions pas de fichier `env`.

```
class DatabaseConnection (AbstractDatabaseConnection):
    def __init__(self):
        host = os.getenv('DB_HOST')
        user = os.getenv('DB_USER')
        password = os.getenv('DB_PASSWORD')
        database = os.getenv('DB_DATABASE')
        port = int(os.getenv('DB_PORT'))

        self.connection = None
        try:
            self.connection = pymysql.connect(
                host=host,
                user=user,
                password=password,
                database=database,
                port=port,
                charset='utf8mb4',
                cursorclass=pymysql.cursors.DictCursor
            )
            print("Connexion à la base de données établie.")
        except pymysql.MySQLError as e:
            print(f"Erreur lors de la connexion à la base de données : {e}")
```

À suivre :

Malheureusement, nous n'avons pas pu fournir plus d'informations sur la documentation. En effet, pour respecter les contraintes de temps, nous nous sommes arrêtés ici.