Smart Inventory Management System

Compte Rendu de Projet

Technologies : Python / Django / MySQL

Auteurs :
AYMANE BENNASSADIA
SIMO EL AYOUCHI

1. Introduction

Smart Inventory Management System est une application web developpee en Python avec le framework Django. Elle permet la gestion complete d'un inventaire : produits, clients et commandes. Le systeme integre egalement un module d'analyse de donnees base sur Pandas et SQLAlchemy pour fournir des statistiques en temps reel (chiffre d'affaires, produits les plus vendus, frequence d'achat des clients, etc.).

2. Architecture du Projet

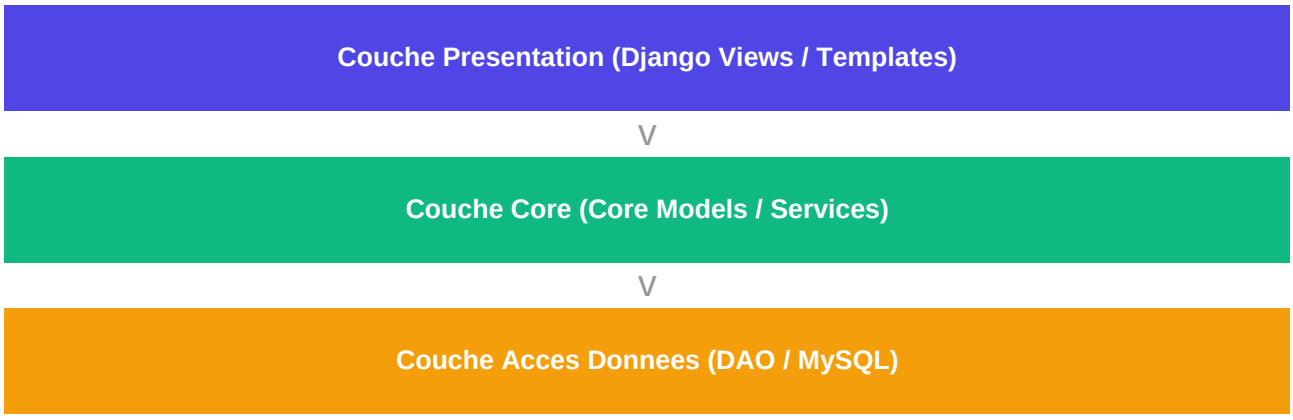
Le projet suit une architecture en couches (Layered Architecture) separant clairement la logique core, l'acces aux donnees et la presentation web.

2.1 Structure des repertoires

```
smart_inventory/
|-- core/
|   |-- models/
|   |-- exceptions/
|   |-- services/
|   |-- test_core.py
|-- database/
|   |-- schema.sql
|   |-- dao/
|       |-- db_config.py
|       |-- populate_db.py
|-- web/django_project/
|   |-- inventory/
|   |-- smart_inventory_web/
|-- analytics/
|-- requirements.txt
```

2.2 Diagramme des couches

L'architecture se decompose en 3 couches principales :



3. Base de Donnees

3.1 SGBD utilise

Le projet utilise MySQL comme systeme de gestion de base de donnees relationnelle.

3.2 Schema de la base de donnees

La base de donnees smart_inventory_db contient 4 tables principales :

Table	Colonnes principales	Cle primaire	Relations
products	name, category, price, quantity_in_stock	id (AUTO_INCREMENT)	-
customers	name, email (UNIQUE)	id (AUTO_INCREMENT)	-
orders	customer_id, order_date	id (AUTO_INCREMENT)	FK -> customers(id)
order_items	order_id, product_id, quantity	id (AUTO_INCREMENT)	FK -> orders, products

3.3 Script SQL

```
CREATE TABLE IF NOT EXISTS products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    category VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    quantity_in_stock INT NOT NULL DEFAULT 0
);

CREATE TABLE IF NOT EXISTS customers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE
);

CREATE TABLE IF NOT EXISTS orders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);

CREATE TABLE IF NOT EXISTS order_items (
    id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

4. Couche Core

4.1 Modele Product

Le modele Product represente un produit en stock. Il contient des methodes pour ajouter/retirer du stock et calculer la valeur totale en stock. Des exceptions personnalisees sont levees en cas de quantite invalide ou de stock insuffisant.

```
class Product:
    def __init__(self, id, name, category, price, quantity_in_stock):

    def add_stock(self, qty):
    def remove_stock(self, qty):
    def get_value_in_stock(self):
```

4.2 Modele Customer

Le modele Customer represente un client. L'email est valide par une expression reguliere lors de la creation de l'objet. Une InvalidEmailException est levee si l'email est invalide.

```
class Customer:
    def __init__(self, id, name, email):
        self.validate_email(email)
```

4.3 Modele Order et OrderItem

Le modele Order represente une commande associee a un client. Il contient une liste d'OrderItem (produit + quantite). La methode calculate_total() calcule le montant total de la commande.

```
class Order:
    def __init__(self, id, customer: Customer):
        self.items = []
    def add_item(self, product, quantity):
    def calculate_total(self):

class OrderItem:
    def __init__(self, product, quantity):
    def get_subtotal(self):
```

4.4 Exceptions personnalisees

Le module core/exceptions/base.py definit 3 exceptions :

- OutOfStockException : stock insuffisant pour retirer la quantite demandee
- InvalidEmailException : format d'email invalide
- InvalidQuantityException : quantite <= 0 ou negative

5. Couche Acces aux Donnees (DAO)

Le pattern DAO (Data Access Object) est utilise pour separer la logique d'acces a la base de donnees de la logique core. Chaque entite dispose de son propre DAO.

5.1 BaseDAO

Classe abstraite qui centralise la configuration et la connexion a MySQL via mysql-connector-python. La configuration est lue depuis db_config.py.

```
class BaseDAO:
```

```
    def __init__(self):
        self.db_config = { ... }
    def get_connection(self):
        return mysql.connector.connect(**self.db_config)
```

5.2 ProductDAO

Operations CRUD completes sur les produits :

Methode	Description	Requete SQL
save(product)	Inserer un nouveau produit	INSERT INTO products ...
update(product)	Mettre a jour un produit	UPDATE products SET ... WHERE id=?
delete(product_id)	Supprimer un produit	DELETE FROM products WHERE id=?
find_by_id(id)	Trouver un produit par ID	SELECT ... FROM products WHERE id=?

5.3 CustomerDAO

Operations CRUD sur les clients (save, update, delete, find_by_id).

5.4 OrderDAO

Le OrderDAO gere les commandes avec leurs items dans une transaction. La methode save() utilise start_transaction() et rollback() pour garantir l'integrite des donnees. Elle insere la commande, ses items, et met a jour le stock.

```
def save(self, order):
    conn.start_transaction()
    INSERT INTO orders ...
    Pour chaque item :
        INSERT INTO order_items ...
        UPDATE products SET quantity_in_stock = ...
    conn.commit()
```

6. Service d'Analyse (Analytics)

Le module analytics_service.py fournit des fonctions d'analyse de donnees basees sur Pandas et SQLAlchemy. Ces fonctions sont appelees depuis le dashboard Django.

Fonction	Description	Retour
get_total_revenue()	Chiffre d'affaires total	float
get_best_selling_products(limit)	Produits les plus vendus	list[dict]
get_stock_value()	Valeur totale du stock	float
get_avg_order_value()	Valeur moyenne par commande	float
get_monthly_revenue()	Revenus mensuels	list[dict]
get_customer_frequency()	Frequence d'achat clients	dict

Chaque fonction utilise SQLAlchemy pour se connecter a MySQL, execute une requete SQL, charge les resultats dans un DataFrame Pandas, puis effectue les calculs necessaires (groupby, sum, mean, etc.). En cas d'erreur, une valeur par defaut est retournee (0.0 ou []).

7. Interface Web (Django)

7.1 Configuration

L'application Django est configuree dans smart_inventory_web/settings.py. Elle utilise la meme configuration MySQL centralisee (db_config.py). Les modeles Django sont declares avec managed = False car les tables sont gerees directement par le schema SQL.

7.2 Modeles Django

Les modeles Django (Product, Customer, Order, OrderItem) sont des miroirs des tables MySQL. Ils utilisent Meta.managed = False et Meta.db_table pour pointer vers les tables existantes.

7.3 Vues (Views)

Les vues principales de l'application sont :

Vue	URL	Description
dashboard	/	Tableau de bord avec statistiques et graphiques
product_list	/products/	Liste de tous les produits
product_create	/products/add/	Formulaire ajout produit
product_update	/products/<id>/edit/	Modification produit
product_delete	/products/<id>/delete/	Suppression produit
customer_list	/customers/	Liste des clients
customer_create	/customers/add/	Formulaire inscription client
order_list	/orders/	Liste des commandes
order_create	/orders/add/	Creation de commande
order_detail	/orders/<id>/	Detail commande

7.4 Formulaires (Forms)

Les formulaires utilisent la validation core des modeles Core. Par exemple, ProductForm appelle CoreProduct() dans clean() pour valider le prix et la quantite. CustomerForm appelle CoreCustomer() pour valider l'email via regex. Le

formulaire de commande utilise un InlineFormSet pour gerer dynamiquement les items.

7.5 Templates

L'interface utilise Bootstrap pour le design responsive. Templates disponibles :

- base.html - Template de base avec navigation
- dashboard.html - Tableau de bord
- product_list.html / product_form.html - Gestion produits
- customer_list.html / customer_form.html - Gestion clients
- order_list.html / order_form.html / order_detail.html - Gestion commandes
- product_confirm_delete.html - Confirmation de suppression

8. Tests Unitaires

Le fichier core/test_core.py contient des tests fonctionnels couvrant les scenarios suivants :

Test	Description	Resultat attendu
Creation produit	Product avec prix et stock valides	OK - objet cree
add_stock	Ajouter 10 unites au stock	stock = 44
remove_stock	Retirer 5 unites	stock = 39
get_value_in_stock	Calculer valeur stock	89.90 * 39
Prix negatif	Product avec price = -1	ValueError levee
Stock insuffisant	remove_stock(100) sur stock=39	OutOfStockException
Quantite negative	Product avec qty = -5	InvalidQuantityException
Email valide	Customer avec email valide	OK - client cree
Email invalide	Customer avec 'not_an_email'	InvalidEmailException
Commande totale	Calcul total (449+75.5*2)	total = 600.0
Commande vide	calculate_total() sans items	total = 0
Quantite 0	add_item avec qty=0	InvalidQuantityException
Quantite negative	add_item avec qty=-3	InvalidQuantityException

9. Technologies et Dependances

Technologie	Version	Utilisation
Python	3.8+	Langage principal
Django	>= 4.2.28	Framework web
MySQL	-	Base de donnees relationnelle
mysql-connector-python	>= 9.1.0	Connexion MySQL (DAO)
SQLAlchemy	>= 2.0.0	Connexion MySQL (Analytics)
NumPy	>= 1.24.0	Calculs numeriques
Matplotlib	>= 3.7.0	Visualisation (Jupyter)
Seaborn	>= 0.12.0	Visualisation (Jupyter)
Bootstrap	-	Framework CSS / UI

10. Installation et Execution

10.1 Pre-requis

- Python 3.8+
- MySQL Server

10.2 Etapes d'installation

1. Creer et activer l'environnement virtuel :

```
python -m venv .venv
.venv\Scripts\activate
```

2. Installer les dependances :

```
pip install -r requirements.txt
```

3. Configurer la base de donnees dans database/dao/db_config.py :

```
DB_CONFIG = {
    'NAME': 'smart_inventory_db',
    'USER': '',
    'PASSWORD': '',
    'HOST': '',
    'PORT': '',
}
```

4. Executer le schema SQL puis les migrations Django :

```
cd web/django_project
python manage.py migrate
```

5. Lancer le serveur :

```
python manage.py runserver
```

Application accessible sur : http://127.0.0.1:8000/

11. Conclusion

Le projet Smart Inventory Management System est une application complete de gestion d'inventaire qui respecte les

bonnes pratiques de developpement logiciel : separation des responsabilites (architecture en couches), validation core centralisee (exceptions personnalisees), acces aux donnees via le pattern DAO avec gestion transactionnelle, et une interface web moderne avec Django et Bootstrap.

Le module d'analyse base sur Pandas et SQLAlchemy permet d'obtenir des indicateurs de performance cles (KPI) directement dans le tableau de bord : chiffre d'affaires, produits populaires, valeur du stock, et comportement des clients.

Les tests unitaires couvrent les principaux cas d'utilisation et les cas limites (validations, exceptions), garantissant la fiabilite de la logique core.