



## **Rapport du TP : Analyse des ventes d'un magasin**

PHILIPPE  
OGOUBI KOMIVI  
L3 DA

## Introduction

Dans un contexte où la donnée est devenue un levier stratégique pour les entreprises, l'analyse des ventes joue un rôle central dans la prise de décisions commerciales. Ce travail pratique s'inscrit dans une démarche d'apprentissage complet autour de la **gestion, l'analyse et la prévision de données commerciales** à l'aide de **Python** et d'une base de données **SQLite**.

Le projet repose sur la création d'un système de gestion des ventes structuré autour de trois entités principales : **les clients, les produits et les ventes**. Ces données ont été modélisées dans une base SQLite, puis enrichies par un jeu de données représentatif d'une activité commerciale réelle, intégrant des ventes réparties sur plusieurs semaines.

À travers ce TP, nous avons manipulé des outils puissants comme `pandas` pour la manipulation des données, `matplotlib/seaborn` pour la visualisation, et `scikit-learn` pour l'entraînement d'un modèle prédictif. L'objectif final était de **prévoir les montants des ventes des jours à venir** à l'aide d'un algorithme de type **Random Forest Regressor**, en se basant sur des variables temporelles extraites des données.

Ce document retrace l'ensemble du processus réalisé : de la structuration de la base de données jusqu'à la génération de prévisions automatisées, en passant par l'analyse descriptive et la validation du modèle.

## I. Le fichier Crud operation

### Structure Générale du Code

#### 1. Importations

```
import pandas as pd
from db_config import connect_db
from datetime import datetime
import numpy as np
```

➤ Explication :

- ✓ `pandas as pd` : bibliothèque puissante pour manipuler des données tabulaires (comme des tableaux ou des tables SQL).
- ✓ `connect_db` : fonction personnalisée importée du fichier `db_config.py`. Elle établit une connexion à la base de données.

- ✓ datetime : permet de manipuler les objets de type date/heure.
- ✓ numpy as np : utilisée ici pour détecter les types numériques dans les DataFrames.

## 2. Fonction utilitaire : conversion des types

```
def _convert_types(df): 5 usages  Philippe-Coder
    """Convertit les types numpy en types Python natifs"""
    for col in df.columns:
        if pd.api.types.is_integer_dtype(df[col]):
            df[col] = df[col].astype(int)
        elif pd.api.types.is_float_dtype(df[col]):
            df[col] = df[col].astype(float)
        elif pd.api.types.is_object_dtype(df[col]):
            df[col] = df[col].astype(str)
    return df
```

### ➤ Explication :

- ✓ Cette fonction prend un DataFrame (df) et parcourt toutes les colonnes.
- ✓ Si une colonne est de type entier ou flottant (int, float) version numpy, elle est convertie vers des types standards Python (int, float).
- ✓ Utile pour éviter les problèmes de compatibilité dans les affichages ou traitements ultérieurs.

## 3. Fonctions liées aux ventes

### ❖ get\_ventes()

```
def get_ventes():
    """Récupère toutes les ventes avec jointures"""
    conn = connect_db()
    try:
        query = """
        SELECT
            v.id,
            v.date_vente,
            v.produit_id,
            v.client_id,
            p.nom as produit,
            p.categorie,
```

```

        c.nom as client,
        v.quantite,
        v.montant
    FROM ventes v
    LEFT JOIN produits p ON v.produit_id = p.id
    LEFT JOIN clients c ON v.client_id = c.id
    ORDER BY v.date_vente DESC
    """

    df = pd.read_sql(query, conn, parse_dates=['date_vente'])
    return _convert_types(df)
except Exception as e:
    raise ValueError(f"Erreur lors de la récupération des ventes: {str(e)}")
finally:
    conn.close()

```

➤ Explication :

- ✓ Requête SQL avec LEFT JOIN sur les tables produits et clients pour obtenir les noms à partir des id.
- ✓ Trié par date décroissante.
- ✓ La colonne date\_vente est automatiquement convertie en objet datetime.
- ✓ Résultat nettoyé avec \_convert\_types().

❖ insert\_vente(...)

```

def insert_vente(date, produit_id, client_id, quantite, montant):
    """Insère une nouvelle vente avec validation"""
    conn = connect_db()
    try:
        # Si la date a une heure, on la convertit au format 'YYYY-MM-DD HH:MM:SS'
        if date:
            # Assurer que la date est au format correct 'YYYY-MM-DD HH:MM:SS'
            date_str = date.strftime('%Y-%m-%d %H:%M:%S') # Format complet avec l'heure
        else:
            date_str = None

        print(f"Insertion de la vente avec la date : {date_str}") # Debug print

        # Paramètres pour l'insertion
        params = (
            date_str,
            int(produit_id),
            int(client_id),
            int(quantite),
            float(montant)
        )

        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO ventes
            (date_vente, produit_id, client_id, quantite, montant)

```

```

VALUES (?, ?, ?, ?, ?)
""" , params)
conn.commit()
return cursor.lastrowid
except Exception as e:
    conn.rollback()
    raise ValueError(f"Erreur insertion vente: {str(e)}")
finally:
    conn.close()

```

➤ Explication :

- ✓ Vérifie que la date est bien au bon format.
- ✓ Prépare une requête SQL d'insertion avec des paramètres typés.
- ✓ Exécute la requête et retourne l'ID de la nouvelle vente insérée.
- ✓ En cas d'erreur : rollback + exception.

❖ update\_vente(...)

```

def update_vente(vente_id, date, produit_id, client_id, quantite, montant):
    """Met à jour une vente existante"""
    conn = connect_db()
    try:
        date_str = date.strftime('%Y-%m-%d %H:%M:%S') if date else None
        params = (
            date_str,
            int(produit_id),
            int(client_id),
            int(quantite),
            float(montant),
            int(vente_id)
        )

        cursor = conn.cursor()
        cursor.execute("""
UPDATE ventes SET
    date_vente=?,
    produit_id=?,
    client_id=?,
    quantite=?,
    montant=?
WHERE id=?
""", params)
        conn.commit()
        return cursor.rowcount
    except Exception as e:
        conn.rollback()
        raise ValueError(f"Erreur mise à jour vente: {str(e)}")

```

```
finally:
    conn.close()
```

- Explication :
  - ✓ Met à jour les colonnes principales d'une vente : date, produit, client, quantité, montant.
  - ✓ Utilise un WHERE id=? pour cibler la vente.
  - ❖ delete\_vente(vente\_id)

```
def delete_vente(vente_id): 1 usage  🧑 Philippe-Coder
    """Supprime une vente"""
    conn = connect_db()
    try:
        cursor = conn.cursor()
        cursor.execute("DELETE FROM ventes WHERE id=?", (int(vente_id),))
        conn.commit()
        return cursor.rowcount
    except Exception as e:
        conn.rollback()
        raise ValueError(f"Erreur suppression vente: {str(e)}")
    finally:
        conn.close()
```

- Explication :
  - ✓ Supprime une vente de la base via son identifiant.
  - ✓ Retourne le nombre de lignes supprimées.

#### 4. Fonctions produits

```
def get_produits():
    """Récupère tous les produits"""
    conn = connect_db()
    try:
        df = pd.read_sql("SELECT * FROM produits ORDER BY nom", conn)
        return _convert_types(df)
    except Exception as e:
        raise ValueError(f"Erreur récupération produits: {str(e)}")
    finally:
        conn.close()

def insert_produit(nom, categorie, prix_unitaire):
    """Ajoute un nouveau produit"""
    conn = connect_db()
    try:
        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO produits (nom, categorie, prix_unitaire)
```

```

VALUES (?, ?, ?)
"""', (str(nom), str(categorie), float(prix_unitaire)))
conn.commit()
return cursor.lastrowid
except Exception as e:
    conn.rollback()
    raise ValueError(f"Erreur insertion produit: {str(e)}")
finally:
    conn.close()

def update_produit(produit_id, nom, categorie, prix_unitaire):
    """Met à jour un produit"""
    conn = connect_db()
    try:
        cursor = conn.cursor()
        cursor.execute("""
UPDATE produits SET
    nom=?,
    categorie=?,
    prix_unitaire=?
WHERE id=?
""", (str(nom), str(categorie), float(prix_unitaire), int(produit_id)))
        conn.commit()
        return cursor.rowcount
    except Exception as e:
        conn.rollback()
        raise ValueError(f"Erreur mise à jour produit: {str(e)}")
    finally:
        conn.close()

def delete_produit(produit_id):
    """Supprime un produit"""
    conn = connect_db()
    try:
        cursor = conn.cursor()
        cursor.execute("DELETE FROM produits WHERE id=?", (int(produit_id),))
        conn.commit()
        return cursor.rowcount
    except Exception as e:
        conn.rollback()
        raise ValueError(f"Erreur suppression produit: {str(e)}")
    finally:
        conn.close()

```

get\_produits(), insert\_produit(), update\_produit(), delete\_produit()

- ✓ get\_produits() récupère tous les produits triés par nom.
- ✓ insert\_produit() ajoute un produit avec nom, catégorie et prix.

- ✓ `update_produit()` met à jour les infos d'un produit existant.
- ✓ `delete_produit()` supprime un produit par son ID.

## 5. Fonctions clients

```
def get_clients():  
    """Récupère tous les clients"""  
    conn = connect_db()  
    try:  
        df = pd.read_sql("SELECT * FROM clients ORDER BY nom", conn)  
        return _convert_types(df)  
    except Exception as e:  
        raise ValueError(f"Erreur récupération clients: {str(e)}")  
    finally:  
        conn.close()
```

```
def insert_client(nom, email, ville):  
    """Ajoute un nouveau client"""  
    conn = connect_db()  
    try:  
        cursor = conn.cursor()  
        cursor.execute("""  
            INSERT INTO clients (nom, email, ville)  
            VALUES (?, ?, ?)  
            """, (str(nom), str(email), str(ville)))  
        conn.commit()  
        return cursor.lastrowid  
    except Exception as e:  
        conn.rollback()  
        raise ValueError(f"Erreur insertion client: {str(e)}")  
    finally:  
        conn.close()
```

```
def update_client(client_id, nom, email, ville):  
    """Met à jour un client"""  
    conn = connect_db()  
    try:  
        cursor = conn.cursor()  
        cursor.execute("""  
            UPDATE clients SET  
                nom=?,  
                email=?,  
                ville=?  
            WHERE id=?  
            """, (str(nom), str(email), str(ville), int(client_id)))  
        conn.commit()  
        return cursor.rowcount  
    except Exception as e:  
        conn.rollback()
```



```

        raise ValueError(f"Erreur mise à jour client: {str(e)}")
    finally:
        conn.close()

def delete_client(client_id):
    """Supprime un client"""
    conn = connect_db()
    try:
        cursor = conn.cursor()
        cursor.execute("DELETE FROM clients WHERE id=?", (int(client_id),))
        conn.commit()
        return cursor.rowcount
    except Exception as e:
        conn.rollback()
        raise ValueError(f"Erreur suppression client: {str(e)}")
    finally:
        conn.close()

```

get\_clients(), insert\_client(), update\_client(), delete\_client()

- ✓ get\_client () récupère tous les clients triés par nom.
- ✓ insert\_client () ajoute un client avec ses attributs.
- ✓ update\_client () met à jour les infos d'un client existant.
- ✓ delete\_client () supprime un client par son ID.
- ✓ clients ont comme attributs : nom, email, ville.

## 6. Fonctions utilitaires : recherche par ID

❖ get\_produit\_by\_id(id)

```

def get_produit_by_id(produit_id):  & Philippe-Coder
    """Récupère un produit par son ID"""
    conn = connect_db()
    try:
        df = pd.read_sql( sql: "SELECT * FROM produits WHERE id=?", conn, params=(int(produit_id),))
        return _convert_types(df).iloc[0] if not df.empty else None
    except Exception as e:
        raise ValueError(f"Erreur récupération produit: {str(e)}")
    finally:
        conn.close()

```

➤ Explication :

- ✓ Exécute une requête SQL pour retrouver un seul produit via son ID.
- ✓ Retourne la première ligne du DataFrame si trouvée, sinon None.

❖ get\_client\_by\_id(id)

```
def get_client_by_id(client_id):  Ⓜ Philippe-Coder
    """Récupère un client par son ID"""
    conn = connect_db()
    try:
        df = pd.read_sql(sql="SELECT * FROM clients WHERE id=?", conn, params=(int(client_id),))
        return _convert_types(df).iloc[0] if not df.empty else None
    except Exception as e:
        raise ValueError(f"Erreur récupération client: {str(e)}")
    finally:
        conn.close()
```

- ✓ Récupère tous les clients par son ID.

## II. Fichier db\_config.py

### 1. Importations

```
import sqlite3
from sqlite3 import Error
import os
```

- Explication :

sqlite3 : module natif de Python pour gérer une base SQLite (sans serveur, tout tient dans un seul fichier).

Error : utilisé pour attraper les erreurs spécifiques à SQLite.

os : permet de gérer le système de fichiers, ici pour s'assurer que le dossier contenant la BDD existe.

### 2. Déclaration du chemin vers la base de données

```
DB_PATH = 'ventes.db'
```

- Explication :

La base sera stockée dans un fichier nommé ventes.db dans le dossier courant.

### 3. Fonction de connexion à la base

```
def connect_db(): 19 usages  👤 Philippe-Coder
    """Établit une connexion à la base SQLite"""
    try:
        if os.path.isdir(DB_PATH):
            os.makedirs(os.path.dirname(DB_PATH), exist_ok=True)

        conn = sqlite3.connect(DB_PATH)
        conn.execute("PRAGMA foreign_keys = ON")
        return conn
    except Error as e:
        print(f"Erreur de connexion SQLite: {e}")
        raise
```

➤ Explication :

La DB\_PATH contient un répertoire (data/ventes.db), il est créé au besoin (os.makedirs()).

sqlite3.connect(DB\_PATH) ouvre une connexion avec la base.

PRAGMA foreign\_keys = ON : active les clés étrangères (désactivées par défaut en SQLite).

La fonction retourne un objet conn pour interagir avec la base.

#### 4. Fonction d'initialisation de la base

```
def init_db():
```

```
    """Initialise la structure de la base avec mise à jour de la table ventes"""
```

```
    conn = connect_db()
```

```
    try:
```

```
        cursor = conn.cursor()
```

➤ Explication :

Connecte à la base et prépare un curseur pour exécuter des requêtes SQL.

Création de la table clients

```
# Table clients
cursor.execute("""
```

```
CREATE TABLE IF NOT EXISTS clients (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  nom TEXT NOT NULL,
  email TEXT,
  ville TEXT
)""")
```

- Structure :
- ✓ id : identifiant unique (clé primaire auto-incrémentée).
- ✓ nom : obligatoire.
- ✓ email et ville : facultatifs.

Création de la table produits

```
# Table produits
cursor.execute("""
CREATE TABLE IF NOT EXISTS produits (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  nom TEXT NOT NULL,
  categorie TEXT,
  prix_unitaire REAL
)""")
```

- Structure :
- ✓ nom : nom du produit.
- ✓ categorie : libre (ex: "Alimentaire", "Électronique"...).
- ✓ prix\_unitaire : prix à l'unité.
- Détection d'une ancienne table ventes

```
# Vérifie si la table ventes existe déjà
cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND
name='ventes'")
if cursor.fetchone():
  # Renomme l'ancienne table
  cursor.execute("ALTER TABLE ventes RENAME TO ventes_old")
```

- Explication :

Vérifie si une ancienne table ventes existe déjà.

Si oui, elle est renommée ventes\_old pour éviter les conflits.

Création de la nouvelle table ventes

```

# Crée la nouvelle table ventes
cursor.execute("""
CREATE TABLE IF NOT EXISTS ventes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    date_vente TEXT NOT NULL,
    produit_id INTEGER,
    client_id INTEGER,
    quantite INTEGER,
    montant REAL,
    FOREIGN KEY (produit_id) REFERENCES produits(id),
    FOREIGN KEY (client_id) REFERENCES clients(id)
)""")

```

➤ Structure :

- ✓ date\_vente est de type TEXT, stockée en format YYYY-MM-DD HH:MM:SS.
- ✓ produit\_id, client\_id sont des clés étrangères liées aux tables produits et clients.
- ✓ quantite : nombre d'articles vendus.
- ✓ montant : total de la vente.

### Migration des anciennes données

```

# Si l'ancienne table existe, on récupère les données
cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND
name='ventes_old'")
if cursor.fetchone():
    cursor.execute("""
INSERT INTO ventes (id, date_vente, produit_id, client_id, quantite, montant)
SELECT id, date_vente, produit_id, client_id, quantite, montant FROM ventes_old
""")
    cursor.execute("DROP TABLE ventes_old")
print("Migration des données réussie.")

```

➤ Explication :

Si une ancienne version de la table ventes existait, ses données sont copiées vers la nouvelle table.

Ensuite, l'ancienne table est supprimée.

Très utile quand on modifie le schéma (comme ici pour date\_vente).

### Validation des modifications

```

conn.commit()

print("Base de données initialisée avec succès.")

```

Enregistre toutes les modifications dans le fichier ventes.db.

## Gestion des erreurs

```
except Error as e:  
    print(f"Erreur d'initialisation: {e}")  
    raise  
finally:  
    conn.close()
```

Affiche l'erreur en cas de souci et ferme toujours la connexion.

### III. Fichier init.py

```
from db_config import init_db
```

➤ Explication :

Ce fichier importe la fonction `init_db` depuis le fichier `db_config.py`.

`init_db()` sert à :

Créer les tables clients, produits, ventes.

Migrer les données si la table ventes existait déjà.

Gérer toutes les erreurs SQL.

```
if __name__ == "__main__":  
    init_db()  
    print("Base de données initialisée.")
```

➤ Explication :

`if __name__ == "__main__":` permet de lancer le script uniquement s'il est exécuté directement (et pas importé ailleurs).

Exécute `init_db()` pour initialiser la base.

Affiche "Base de données initialisée." à la fin, pour confirmer que tout s'est bien passé

### IV. Fichier ml\_forecasting.py

#### 1. Importations

```
import pandas as pd  
from datetime import timedelta
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
```

- ✓ timedelta : pour créer les dates futures.
- ✓ RandomForestRegressor : modèle d'apprentissage automatique utilisé pour la prédiction.
- ✓ train\_test\_split : division des données en entraînement/test.
- ✓ mean\_absolute\_error : mesure d'évaluation de la performance du modèle.

## 2. Fonction principale : forecast\_sales(df\_ventes, periods=6)

### ➤ Objectif :

Prédire le montant total des ventes pour les periods prochains jours (par défaut 6 jours).

Bloc try : pour gérer proprement les erreurs

```
df = df_ventes.copy()
```

```
df['date_vente'] = pd.to_datetime(df['date_vente'])
```

```
df = df.set_index('date_vente').resample('D').sum(numeric_only=True).reset_index()
```

- ✓ Copie le DataFrame pour ne pas modifier l'original.
- ✓ Convertit la colonne date\_vente en format date.
- ✓ Fait un resample journalier (somme des montants par jour) → important pour que chaque jour ait une ligne.

## 3. Ajout de nouvelles colonnes temporelles (features)

```
df['jour'] = df['date_vente'].dt.day
```

```
df['mois'] = df['date_vente'].dt.month
```

```
df['annee'] = df['date_vente'].dt.year
```

```
df['jour_semaine'] = df['date_vente'].dt.dayofweek
```

Ces colonnes représentent les caractéristiques temporelles utilisées comme variables d'entrée pour l'apprentissage.

## 4. Préparation des données pour le modèle

```
X = df[['jour', 'mois', 'annee', 'jour_semaine']]
```

```
y = df['montant']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

X = variables explicatives (date décomposée).

y = variable à prédire : le montant des ventes.

On divise 80% pour l'entraînement, 20% pour le test.

## 5. Entraînement du modèle Random Forest

```
model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
model.fit(X_train, y_train)
```

100 arbres dans la forêt.

Le modèle apprend à prédire les montants à partir des dates.

#### 6. Création des dates futures à prédire

```
last_date = df['date_vente'].max()
```

```
future_dates = [last_date + timedelta(days=i) for i in range(1, periods + 1)]
```

On génère des dates pour les periods jours à venir.

#### 7. Préparation du DataFrame pour la prédiction future

```
future_df = pd.DataFrame({  
    'date_vente': future_dates,  
    'jour': [d.day for d in future_dates],  
    'mois': [d.month for d in future_dates],  
    'annee': [d.year for d in future_dates],  
    'jour_semaine': [d.weekday() for d in future_dates]  
})
```

Même format que l'entraînement : on extrait les infos temporelles pour les dates futures.

#### 8. Prédiction + évaluation du modèle

```
future_df['prediction'] = model.predict(future_df[['jour', 'mois', 'annee', 'jour_semaine']])
```

```
mae = mean_absolute_error(y_test, model.predict(X_test))
```

On prédit les ventes pour chaque jour à venir.

On calcule l'erreur absolue moyenne sur les données test pour évaluer la qualité du modèle.

#### 9. Retour des résultats

```
return future_df[['date_vente', 'prediction']], mae
```

Retourne un DataFrame avec les dates futures et les montants prédits, ainsi que l'erreur (MAE).

En cas d'erreur

```
except Exception as e:
```

```
    return None, str(e)
```

Gestion simple des exceptions : retourne une erreur sous forme de chaîne.



## V. Fichier seed.py

### 1. Importation des modules nécessaires

```
from db_config import connect_db
```

`connect_db` : Une fonction qui établit une connexion avec la base de données SQLite. Elle est importée depuis le fichier `db_config.py`.

### 2. Fonction principale : `seed_data()`

```
def seed_data():
```

Cette fonction gère l'insertion des données dans la base de données. Voici les étapes qu'elle exécute :

### 3. Insertion des produits

```
produits = [  
    ("Stylo bleu", "Papeterie", 1.50),  
    ("Carnet A5", "Papeterie", 3.00),  
    ("Calculatrice", "Électronique", 15.00),  
    ("Clé USB 16GB", "Informatique", 8.00),  
    ("Ramette de papier", "Papeterie", 5.50),  
    ("Souris sans fil", "Électronique", 12.00),  
    ("Agrafeuse", "Bureau", 4.00),  
    ("Enveloppes x50", "Papeterie", 2.50),  
    ("Tapis de souris", "Bureau", 6.00),  
    ("Tableau blanc", "Bureau", 25.00),  
]
```

```
cursor.executemany("INSERT INTO produits (nom, categorie, prix_unitaire) VALUES (?, ?, ?)", produits)
```

Une liste de produits est définie. Chaque produit contient son nom, sa catégorie et son prix unitaire.

La méthode `executemany` permet d'insérer plusieurs enregistrements en une seule requête SQL.

### 4. Insertion des clients

```
clients = [
```

("Alice Dupont", "alice@example.com", "Paris"),  
("Bob Martin", "bob@example.com", "Lyon"),  
("Claire Morel", "claire@example.com", "Marseille"),  
("David Leroy", "david@example.com", "Toulouse"),  
("Emma Dubois", "emma@example.com", "Nice"),  
("François Petit", "francois@example.com", "Nantes"),  
("Gabrielle Chevalier", "gabrielle@example.com", "Strasbourg"),  
("Hugo Bernard", "hugo@example.com", "Montpellier"),  
("Isabelle Fontaine", "isabelle@example.com", "Bordeaux"),  
("Julien Lefevre", "julien@example.com", "Lille"),

]

cursor.executemany("INSERT INTO clients (nom, email, ville) VALUES (?, ?, ?)", clients)

Une liste de clients est définie, chaque client ayant un nom, un email et une ville.

L'insertion se fait de la même manière que pour les produits.

## 5. Insertion des ventes

ventes = [

("2025-02-03", 1, 1, 5, 7.50),  
("2025-02-05", 2, 2, 2, 6.00),  
("2025-02-10", 3, 3, 1, 15.00),  
("2025-02-15", 4, 4, 3, 24.00),  
("2025-02-20", 5, 5, 1, 5.50),  
("2025-02-25", 6, 6, 1, 12.00),  
("2025-03-01", 7, 7, 2, 8.00),  
("2025-03-05", 8, 8, 1, 2.50),  
("2025-03-09", 9, 9, 1, 6.00),  
("2025-03-12", 10, 10, 1, 25.00),  
("2025-03-15", 3, 1, 1, 15.00),  
("2025-03-20", 2, 4, 3, 9.00),  
("2025-03-22", 5, 6, 2, 11.00),  
("2025-03-27", 6, 7, 1, 12.00),

```

("2025-04-01", 1, 2, 10, 15.00),
("2025-04-04", 4, 5, 1, 8.00),
("2025-04-07", 7, 6, 1, 4.00),
("2025-04-08", 8, 3, 2, 5.00),
("2025-04-08", 9, 8, 1, 6.00),
("2025-04-08", 10, 9, 1, 25.00),
]

cursor.executemany("INSERT INTO ventes (date_vente, produit_id, client_id, quantite,
montant) VALUES (?, ?, ?, ?, ?)", ventes)

```

Une liste de ventes est définie avec les dates de vente, les identifiants des produits et clients, la quantité vendue et le montant total.

Ici aussi, executemany insère plusieurs ventes d'un coup.

## 6. Commit et fermeture de la connexion

```
conn.commit()
```

```
conn.close()
```

### ➤ Explication :

- ✓ conn.commit() valide les modifications dans la base de données.
- ✓ conn.close() ferme la connexion à la base de données une fois que toutes les données ont été insérées.

## 7. Message de succès

```
print("Données insérées avec succès avec des dates variées !")
```

Affiche un message pour confirmer que les données ont bien été insérées.

## 8. Exécution du script

```
if __name__ == "__main__":
    seed_data()
```

Vérifie que ce script est exécuté directement (et non importé dans un autre script).

Appelle la fonction seed\_data() pour insérer les données de test.

# VI. Explication du Code : Dashboard d'Analyse des Ventes

## 1. Importations

Le code utilise plusieurs bibliothèques :

```
import streamlit as st
import pandas as pd
import plotly.express as px
from datetime import datetime, timedelta
from db_config import connect_db, init_db
from crud_operations import *
from ml_forecasting import forecast_sales
from st_aggrid import AgGrid, GridOptionsBuilder, GridUpdateMode
import numpy as np
from fpdf import FPDF
import io
from PIL import Image
import base64
from io import BytesIO
```

- streamlit pour l'interface web
- pandas pour la manipulation des données
- plotly pour les visualisations
- FPDF pour générer des rapports PDF
- Modules personnalisés ('db\_config', 'crud\_operations', 'ml\_forecasting') pour la base de données et les opérations métier

## 2. Initialisation

- La base de données est initialisée avec init\_db()
- La configuration de la page est définie avec : set\_page\_config()

### **Fonctionnalités Principales**

#### ➤ Authentification :login()

- Un système simple d'authentification dans la sidebar
- Identifiants codés en dur ("admin"/"admin123")

```
def login(): 1 usage  Philippe-Coder
    st.sidebar.subheader("Connexion Admin")
    user = st.sidebar.text_input("Nom d'utilisateur")
    password = st.sidebar.text_input("Mot de passe", type="password")
    if st.sidebar.button("Se connecter"):
        if user == "admin" and password == "admin123":
            st.session_state["logged_in"] = True
            st.rerun()
        else:
            st.sidebar.error("Identifiants invalides")
```

`st.sidebar.subheader("Connexion Admin")`

Affiche un sous-titre "Connexion Admin" dans la barre latérale de l'application Streamlit.

`user = st.sidebar.text_input("Nom d'utilisateur")`

`password = st.sidebar.text_input("Mot de passe", type="password")`

Crée deux champs de saisie dans la barre latérale :

Un pour entrer le nom d'utilisateur

Un pour entrer le mot de passe (masqué grâce à `type="password"`)

`if st.sidebar.button("Se connecter"):`

Affiche un bouton "Se connecter".

Quand ce bouton est cliqué, le code à l'intérieur du bloc `if` est exécuté.

`if user == "admin" and password == "admin123":`

`st.session_state["logged_in"] = True`

`st.rerun()`

Vérifie que les identifiants entrés correspondent à "admin" et "admin123".

Si les identifiants sont corrects :

On stocke `True` dans `st.session_state["logged_in"]` → ce qui permet de garder l'utilisateur connecté entre les pages ou après rechargement.

`st.rerun()` recharge l'application Streamlit, pour refléter le fait que l'utilisateur est maintenant connecté (par exemple, afficher du contenu réservé à l'admin).

`else:`

`st.sidebar.error("Identifiants invalides")`

Si les identifiants sont incorrects, un message d'erreur est affiché dans la barre latérale.

### 3. Tableau de Bord : show\_dashboard()

**Fonction** show\_dashboard()

```
def show_dashboard():
```

Déclaration de la fonction qui affichera le tableau de bord des ventes dans Streamlit.

#### 3.1 Titre de la page

```
st.header("Tableau de bord avancé des ventes")
```

Affiche un en-tête sur la page. C'est le titre principal visible en haut du tableau de bord.

#### 3.2 Chargement des données de ventes

```
df_ventes = get_ventes()
```

Appelle une fonction personnalisée get\_ventes() qui va récupérer les données de ventes depuis la base de données (probablement MySQL). Le résultat est un DataFrame Pandas.

#### 3.3 Traitement des dates

```
df_ventes['date_vente'] = pd.to_datetime(df_ventes['date_vente'], errors='coerce')
```

Convertit la colonne date\_vente en objet datetime.

Si des valeurs ne peuvent pas être converties (par exemple, une chaîne invalide), elles deviennent NaT grâce à errors='coerce'.

```
df_ventes = df_ventes.dropna(subset=['date_vente'])
```

Supprime les lignes où date\_vente est vide ou invalide (NaT).

#### 3.4 Vérification de la disponibilité des données

```
if df_ventes.empty:
```

```
    st.warning("Aucune donnée de vente valide disponible")  
    return
```

Si le DataFrame est vide (après filtrage des dates invalides), un message d'avertissement s'affiche, et la fonction s'arrête.

### 4. Définition des bornes de dates

```
min_date = df_ventes['date_vente'].min().to_pydatetime()
```

```
max_date = df_ventes['date_vente'].max().to_pydatetime()
```

Récupère les dates minimum et maximum du DataFrame pour définir les bornes possibles du filtre de dates.

`to_pydatetime()` convertit les objets `Timestamp` en objets `datetime.datetime` natifs de Python.

## 5. Filtres dynamiques dans un expander Streamlit

```
with st.expander(" Filtres", expanded=True):
```

Création d'un menu déroulant contenant les filtres, ouvert par défaut.

Filtre de dates

```
date_debut = st.date_input("Date de début", value=min_date, min_value=min_date,
max_value=max_date)
```

```
date_fin = st.date_input("Date de fin", value=max_date, min_value=min_date,
max_value=max_date)
```

L'utilisateur choisit une plage de dates. Les bornes sont limitées aux valeurs réelles dans la base.

Filtre de catégories de produits

```
categories = df_ventes['categorie'].dropna().unique()
```

```
selected_categories = st.multiselect("Catégories", options=categories,
default=categories)
```

Sélection d'une ou plusieurs catégories via un menu à choix multiple. Par défaut, toutes les catégories sont sélectionnées.

Sélection de la période d'analyse

```
freq = st.selectbox("Période", options=["D", "W", "M", "Q", "Y"], format_func=lambda
x: {"D": "Jour", "W": "Semaine", "M": "Mois", "Q": "Trimestre", "Y": "Année"}[x])
```

Choix de la fréquence d'analyse pour les graphes d'évolution (D = Day, W = Week, etc.). `format_func` permet d'afficher un nom lisible pour l'utilisateur.

## 6. Application des filtres sur les données

```
mask = (
    (df_ventes['date_vente'].dt.date >= date_debut) &
    (df_ventes['date_vente'].dt.date <= date_fin)
)
```

On crée un **masque booléen** : il retourne `True` pour les lignes dans la plage de dates choisie.

```
filtered_df = df_ventes[mask]
```

On filtre le DataFrame selon ce masque.

```
if selected_categories:  
    filtered_df = filtered_df[filtered_df['categorie'].isin(selected_categories)]
```

On applique un second filtre pour ne garder que les lignes dont la catégorie est dans la sélection de l'utilisateur.

## 7. Calcul des KPI

```
total = filtered_df['montant'].sum()  
count = len(filtered_df)  
avg = filtered_df['montant'].mean()
```

Calcul des indicateurs :

- Total des ventes
- Nombre de ventes
- Panier moyen

```
top_product = filtered_df['produit'].value_counts().idxmax() if not filtered_df.empty else  
"Aucun"
```

On détermine le produit le plus vendu (celui avec le plus de lignes).

```
col1, col2, col3, col4 = st.columns(4)
```

Affiche 4 colonnes horizontales pour organiser les KPI côte à côte.

## 8. Graphique d'évolution des ventes

```
period_sales = filtered_df.groupby(pd.Grouper(key='date_vente',  
freq=freq)).agg({'montant': 'sum', 'produit': 'count'}).rename(columns={'montant': 'Total des  
ventes', 'produit': 'Nombre de ventes'})
```

Regroupe les ventes par période choisie (freq), puis calcule :

- Le total des ventes par période
- Le nombre de ventes

```
st.line_chart(period_sales)
```

Affiche le graphique de l'évolution temporelle.

```
if len(period_sales) >= 3:
```



```
period_sales['Tendance'] = period_sales['Total des ventes'].rolling(window=3).mean()  
st.line_chart(period_sales[['Total des ventes', 'Tendance']])
```

Si on a assez de points, on ajoute une courbe de tendance (moyenne glissante sur 3 périodes).

## 9. Répartition des ventes

```
category_sales = filtered_df.groupby('categorie').agg({'montant': 'sum', 'produit':  
'count'}).rename(columns={'montant': 'Total des ventes', 'produit': 'Nombre de ventes'})
```

Regroupement par catégorie de produit.

```
st.pyplot(...) # graphique circulaire (camembert)  
st.bar_chart(...) # graphique à barres
```

Affichage :

- Pie chart (répartition du chiffre d'affaires par catégorie)
- Histogramme (nombre de ventes par catégorie)

## 10. Performance produite

```
product_performance = filtered_df.groupby('produit').agg({'quantite':  
'sum'}).sort_values(by='quantite', ascending=False)
```

Classement des produits selon la quantité vendue.

```
top5 = product_performance.head(5)  
bottom5 = product_performance.tail(5)
```

Top 5 et flop 5 produits.

## 11. Détails complets (tableau)

```
st.dataframe(filtered_df.sort_values(by='date_vente', ascending=False))
```

Affiche toutes les ventes filtrées dans un tableau dynamique, trié par date décroissante.

## 12. Export Excel

with pd.ExcelWriter(buffer, engine='xlsxwriter') as writer:

```
writer.save()  
st.download_button(...)
```

Génère un fichier Excel avec plusieurs onglets :

- Données filtrées
- Répartition par catégorie
- Performances produits

Utilise un buffer mémoire temporaire (io.BytesIO()).

### 13. Export PDF

```
pdf = FPDF()
pdf.set_font("Arial", size=12)
...
st.download_button(label=" Télécharger le rapport PDF",
data=pdf.output(dest='S').encode('latin1'), ...)
```

Création d'un rapport PDF avec :

- Titre
- KPI
- Données tabulaires

Utilise la bibliothèque fpdf pour générer le document.

### 14. Gestion des erreurs

except Exception as e:

```
st.error(f"Erreur lors du chargement des données: {str(e)}")
```

Capture toute erreur éventuelle dans la fonction, et affiche un message clair à l'utilisateur.


## ❖ Fichier Excel des analyses de ventes

Fonction create\_sales\_excel() – Export Excel Multi-Onglets

```
def create_sales_excel(filtered_df, period):
    "Crée un fichier Excel des analyses de ventes"
```

La fonction prend deux paramètres :

- filtered\_df : un DataFrame Pandas contenant les données de ventes filtrées par date, catégorie, etc.
- period : la période d'analyse choisie par l'utilisateur (même si ici elle n'est pas utilisée dans la fonction, on pourrait l'utiliser pour nommer les fichiers ou les onglets dynamiquement).

 Préparation d'un buffer mémoire

```
output = io.BytesIO()
```

Crée un **flux mémoire binaire** (plutôt qu'un fichier physique).

Cela permet d'envoyer le fichier Excel directement dans le navigateur avec Streamlit, sans le stocker sur le disque.

#### Création du fichier Excel avec ExcelWriter

```
with pd.ExcelWriter(output, engine='openpyxl') as writer:
```

Ouvre un **contexte d'écriture Excel** à l'aide de openpyxl, un moteur compatible avec les fichiers .xlsx.

On lie cet écrivain au buffer output précédemment créé.

#### Onglet 1 – Ventes détaillées

```
filtered_df.to_excel(writer, sheet_name='Ventes détaillées', index=False)
```

Enregistre le **DataFrame complet** dans un premier onglet appelé "Ventes détaillées".  
index=False évite d'écrire l'index Pandas dans le fichier (inutile ici).

#### Onglet 2 – Synthèse par catégorie

```
summary_cat = filtered_df.groupby('categorie').agg({  
    'montant': ['sum', 'count'],  
    'quantite': 'sum'  
})
```

On regroupe les ventes **par catégorie** et on calcule :

- sum des montants → chiffre d'affaires total par catégorie
- count des montants → nombre de ventes
- sum des quantités → volume de produits vendus

```
summary_cat.to_excel(writer, sheet_name='Par catégorie')
```

On écrit le tableau dans un onglet Excel nommé "**Par catégorie**".

#### Onglet 3 – Synthèse par produit

```
summary_prod = filtered_df.groupby('produit').agg({  
  
    'montant': ['sum', 'mean'],  
    'quantite': 'sum'  
})
```


On regroupe par **produit** avec les statistiques suivantes :

- sum des montants → chiffre d'affaires par produit
- mean des montants → panier moyen par produit
- sum des quantités → nombre total de produits vendus

```
summary_prod.to_excel(writer, sheet_name='Par produit')
```


Ajoute un troisième onglet "**Par produit**" avec cette synthèse.

```

 Remise à zéro du pointeur
output.seek(0)
```

Positionne le curseur du buffer au **début** pour que le fichier soit lisible depuis le début lors du téléchargement.

```

 Retour du fichier généré

return output
```

Retourne le fichier binaire prêt à être utilisé par un bouton Streamlit (st.download\_button).

## 15. Gestion des Données

❖ Ventes gestion\_ventes()

```
def gestion_ventes():
```

```
    st.header("Gestion des Ventes")
```

Affiche un titre principal en haut de la page.

Sécurité d'exécution multiple :

```
    current_time = datetime.now()
```

On récupère l'heure actuelle, ce qui peut être utile pour des logs ou pour éviter que certaines actions ne soient rejouées plusieurs fois par inadvertance.

✓ Création des onglets

```
    tab1, tab2, tab3 = st.tabs(["Voir les ventes", "Ajouter une vente", "Modifier/Supprimer"])
```

On crée trois onglets Streamlit pour séparer les fonctionnalités :

tab1 : Visualisation

tab2 : Ajout

tab3 : Modification ou suppression

a) Onglet 1 : Voir les ventes

```
    with tab1:
```

```
        df_ventes = get_ventes()
```

Récupère les données des ventes depuis la base de données via une fonction personnalisée `get_ventes()`.

```
gb = GridOptionsBuilder.from_dataframe(df_ventes)
```

Initialise les options de la grille interactive AgGrid à partir du DataFrame.

```
gb.configure_pagination(paginationAutoPageSize=True)
```

```
gb.configure_side_bar()
```

```
gb.configure_selection('single', use_checkbox=True)
```

Ajoute :la pagination automatique, une barre latérale pour filtrer/organiser, une sélection unique par case à cocher.

```
grid_options = gb.build()
```

Construit les paramètres de la grille.

```
grid_response = AgGrid(  
    df_ventes,  
    gridOptions=grid_options,  
    update_mode=GridUpdateMode.SELECTION_CHANGED,  
    theme='streamlit' if st.session_state.get("theme", "light") == "light" else 'dark',  
    height=400,  
    reload_data=True  
)
```

Affiche le tableau AgGrid interactif dans l'interface : Adapté au thème clair/sombre, Actualisation automatique, Hauteur définie à 400px

b) Onglet 2 : Ajouter une vente

with tab2:

```
form_ajout = st.form(key='form_ajout_vente')
```

On crée un formulaire structuré nommé `form_ajout_vente`.

Sélection des données à insérer

with form\_ajout:

```
col1, col2 = st.columns(2)
```

Découpe le formulaire en deux colonnes.

Colonne 1 : date + produit

```
date = st.date_input("Date de vente", value=datetime.now())
```

```
df_produits = get_produits()
```

```
produit_id_map = {row['nom']: int(row['id']) for _, row in df_produits.iterrows() }
```

```
produit = st.selectbox("Produit", list(produit_id_map.keys()))
```

```
produit_id = produit_id_map[produit]
```

Affiche :

Un champ de date

Une liste déroulante des produits récupérés de la base de données

On mappe chaque nom à son id via un dictionnaire

Colonne 2 : client + quantité + montant

```
df_clients = get_clients()
client_id_map = {row['nom']: int(row['id']) for _, row in df_clients.iterrows()}
client = st.selectbox("Client", list(client_id_map.keys()))
client_id = client_id_map[client]
quantite = st.number_input("Quantité", min_value=1, step=1, value=1)
montant = st.number_input("Montant", min_value=0.0, value=0.0)
```

Identique aux produits mais avec les clients.

Ajout de champs numériques pour quantité et montant.

#### ❖ Soumission du formulaire

```
submit = form_ajout.form_submit_button("Ajouter la vente")
if submit:
    try:
        insert_vente(date, produit_id, client_id, quantite, montant)
        st.success("Vente ajoutée avec succès !")
        st.rerun()
```

Quand on clique sur Ajouter la vente :

Les données sont envoyées à la fonction insert\_vente()

Message de succès + actualisation automatique de la page (st.rerun())

```
except Exception as e:
    st.error(f"Erreur lors de l'ajout: {str(e)}")
```

Affiche une erreur si l'insertion échoue (ex. : contrainte SQL).

#### c) Onglet 3 : Modifier / Supprimer une vente

with tab3:

```
st.subheader("Modifier ou supprimer une vente")
df_ventes = get_ventes()
```

On récupère à nouveau toutes les ventes.

Vérification s'il y a des ventes

```
if len(df_ventes) == 0:  
    st.warning("Aucune vente à afficher")
```

Si aucune vente n'existe, on affiche un message d'alerte.

Sélection d'une vente à modifier

```
else:  
    df_ventes['id'] = df_ventes['id'].astype(int)  
    selected = st.selectbox(  
        "Sélectionner une vente à modifier",  
        df_ventes.apply(  
            lambda x: f"{x['date_vente']} - {x['produit']} - {x['client']} - {x['montant']}CFA",  
            axis=1  
        )  
    )
```

L'utilisateur sélectionne une vente sous forme textuelle.

On génère une liste à partir du DataFrame.

```
selected_id = int(df_ventes.iloc[  
    df_ventes.apply(  
        lambda x: f"{x['date_vente']} - {x['produit']} - {x['client']} - {x['montant']}CFA",  
        axis=1  
    ).tolist().index(selected)  
    ]['id'])
```

On retrouve l'id de la vente sélectionnée pour la modifier/supprimer.

### ❖ Formulaire de modification

```
form_modif = st.form(key='form_modif_vente')
```

Crée un formulaire de modification.

Remplissage initial des champs avec les données de la vente

```
vente_data = df_ventes[df_ventes['id'] == selected_id].iloc[0]
```

On extrait les données de la vente choisie.

Ensuite, on affiche deux colonnes, avec les champs pré-remplis (date, produit, client, quantité, montant), et on gère le mapping nom ↔ id comme pour l'ajout.

Boutons "Modifier" et "Supprimer"

```
with col1:
    if form_modif.form_submit_button("Modifier"):
        update_vente(...)
        st.success(...)
        st.rerun()
```

En cliquant sur Modifier, les données sont mises à jour via update\_vente().

```
with col2:
    if form_modif.form_submit_button("Supprimer"):
        delete_vente(selected_id)
        st.success(...)
        st.rerun()
```

En cliquant sur Supprimer, la vente est supprimée via delete\_vente().

Gestion globale des erreurs

```
except Exception as e:
    st.error(f"Erreur dans la gestion des ventes: {str(e)}")
```

Permet d'afficher une erreur en cas de plantage général de la fonction.

### ❖ **Fonction : gestion\_produits()**

```
def gestion_produits():
```

C'est la définition de la fonction principale. Elle est appelée lorsqu'on veut afficher et gérer les produits dans l'interface.

1. Titre principal de la page  
`st.header("Gestion des Produits")`

Affiche un gros titre avec une icône sur la page.

Bloc try principal  
`try:`

Permet de capturer toute erreur dans la fonction, pour éviter que l'app Streamlit plante.

Création de trois onglets

```
tab1, tab2, tab3 = st.tabs([" Voir les produits", " Ajouter un produit", " Modifier /  
Supprimer"])
```

Création d'une interface à 3 **onglets** :

#### 1. **Voir les produits**



2. **Ajouter un produit**
3. **Modifier ou Supprimer**

Onglet 1 – Consultation avec AgGrid

with tab1:

```
st.subheader("Liste des produits")
df_produits = get_produits()
```

Récupère tous les produits depuis la base de données dans un DataFrame.

if df\_produits.empty:

```
st.info("Aucun produit trouvé.")
```

Affiche un message si aucun produit n'est enregistré.

else:

```
gb = GridOptionsBuilder.from_dataframe(df_produits)
gb.configure_pagination(paginationAutoPageSize=True)
gb.configure_side_bar()
gb.configure_default_column(editable=False, groupable=True)
gb.configure_selection(selection_mode="single", use_checkbox=True)
grid_options = gb.build()
```

Configuration de la grille interactive avec AgGrid :

- **Pagination automatique**
- **Barre latérale**
- **Colonnes non éditables**
- **Sélection unique avec case à cocher**

```
AgGrid(
    df_produits,
    gridOptions=grid_options,
    update_mode=GridUpdateMode.SELECTION_CHANGED,
    theme="streamlit",
    height=400,
    fit_columns_on_grid_load=True
)
```

Affiche le tableau interactif avec toutes les options définies.

Onglet 2 – Ajout d'un produit

with tab2:

```
st.subheader("Ajouter un produit")
```

Sous-titre de l'onglet 2

```
with st.form("form_ajout_produit"):
    nom = st.text_input("Nom du produit")
    categorie = st.text_input("Catégorie")
    prix_unitaire = st.number_input("Prix unitaire (€)", min_value=0.0, format="%.2f")
```

Formulaire pour entrer :

- Le nom
- La catégorie
- Le prix unitaire (non négatif, formaté en euros)

```
submit = st.form_submit_button(" Ajouter")
```

Bouton de soumission du formulaire

if submit:

```
    try:
        insert_produit(nom, categorie, prix_unitaire)
        st.success("Produit ajouté avec succès !")
        st.rerun()
```

À la soumission, les données sont envoyées à la fonction `insert_produit()` pour les ajouter à la base. Si tout va bien, on affiche un message de succès et on relance l'app (`st.rerun()`).

Onglet 3 – Modifier / Supprimer un produit

```
with tab3:
    st.subheader(" Modifier ou Supprimer un produit")
    df_produits = get_produits()
```

Récupère à nouveau les produits.

if `df_produits.empty`:

```
    st.warning("Aucun produit à modifier ou supprimer.")
```

Message s'il n'y a aucun produit

else:

```
    selected_nom = st.selectbox("Sélectionnez un produit :", df_produits['nom'])
```

L'utilisateur choisit le produit à modifier ou supprimer via une liste déroulante (`selectbox`).

```
    selected_row = df_produits[df_produits['nom'] == selected_nom].iloc[0]
    produit_id = selected_row['id']
```

On récupère toutes les données du produit sélectionné (pour pré-remplir le formulaire) + son identifiant.

Formulaire pour modifier ou supprimer

```
with st.form("form_modif_supp_produit"):
    new_nom = st.text_input("Nom", value=selected_row['nom'])
    new_categorie = st.text_input("Catégorie", value=selected_row['categorie'])
    new_prix = st.number_input("Prix unitaire (€)", min_value=0.0,
value=float(selected_row['prix_unitaire']), format="%.2f")
```

Formulaire pré-rempli avec les données du produit actuel

```
col1, col2 = st.columns(2)
```

On sépare le formulaire en deux colonnes pour afficher deux boutons côte à côte

Modification du produit

with col1:

```
if st.form_submit_button(" Modifier"):
    try:
        update_produit(produit_id, new_nom, new_categorie, new_prix)
        st.success("Produit modifié avec succès !")
        st.rerun()
```

Quand on clique sur "Modifier", on appelle update\_produit() et on actualise la page

Suppression du produit

with col2:

```
if st.form_submit_button(" Supprimer"):
    try:
        delete_produit(produit_id)
        st.success("Produit supprimé avec succès !")
        st.rerun()
```

Quand on clique sur "Supprimer", on appelle delete\_produit() et on actualise aussi la page.

Gestion globale des erreurs

```
except Exception as e:
```

```
    st.error(f"Erreur globale dans la gestion des produits : {e}")
```

Si n'importe quelle partie du bloc plante, cette ligne affiche l'erreur.

## ❖ **Fonction** gestion\_clients()

```
def gestion_clients():
```

Définition de la fonction principale. Elle affiche l'interface pour gérer les clients.

✓ Titre principal

```
st.header("Gestion des Clients")
```

Affiche un grand titre en haut de la page.

Bloc try principal

try:

Permet de capturer les erreurs globales pour éviter que l'application plante.

Création des 3 onglets

```
tab1, tab2, tab3 = st.tabs(["Voir les clients", "Ajouter un client", "Modifier/Supprimer"])
```

Création de trois onglets :

1. Voir tous les clients
2. Ajouter un nouveau client
3. Modifier ou supprimer un client

Onglet 1 – Voir les clients

with tab1:

```
df_clients = get_clients()
st.dataframe(df_clients, height=400)
```

- `get_clients()` : fonction qui récupère tous les clients depuis la base de données.
- `st.dataframe(...)` : affiche les données dans un tableau interactif. Le tableau est scrollable verticalement avec une hauteur de 400px.

Onglet 2 – Ajouter un client

with tab2:

```
with st.form("form_ajout_client"):
```

Crée un **formulaire** nommé `form_ajout_client` dans lequel l'utilisateur peut entrer des infos.

```
nom = st.text_input("Nom complet")
email = st.text_input("Email")
ville = st.text_input("Ville")
```

Trois champs pour ajouter un client : son **nom**, **email**, et **ville**.

```
submit = st.form_submit_button("Ajouter le client")
```

Bouton pour envoyer le formulaire.

```
if submit:
    try:
        insert_client(nom, email, ville)
        st.success(" Client ajouté avec succès !")
        st.rerun()
```

Si on clique sur le bouton :

- On appelle la fonction `insert_client(...)` pour enregistrer le client en base.
- `st.success(...)` : affiche un message de succès.
- `st.rerun()` : recharge automatiquement la page Streamlit pour actualiser la liste.

```
except Exception as e:
    st.error(f"Erreur lors de l'ajout: {str(e)}")
```

Gestion d'erreur spécifique à l'ajout d'un client.

Onglet 3 – Modifier / Supprimer un client

```
with tab3:
    st.subheader("Modifier ou supprimer un client")
    df_clients = get_clients()
```

Titre de la section + récupération de la liste des clients

```
if len(df_clients) == 0:
    st.warning("Aucun client à afficher")
```

Si la base est vide, on affiche un message d'alerte.

Sélection d'un client à modifier/supprimer

```
else:
    selected = st.selectbox("Sélectionner un client à modifier", df_clients['nom'])
```

L'utilisateur choisit un client par son nom dans une liste déroulante (selectbox).

```
selected_id = df_clients[df_clients['nom'] == selected]['id'].values[0]
client_data = df_clients[df_clients['id'] == selected_id].iloc[0]
```

On récupère :

- L'id du client sélectionné (pour modification/suppression dans la base)
- Toutes les données du client sélectionné (`client_data`)

Formulaire de modification/suppression

```

with st.form("form_modif_client"):
    new_nom = st.text_input("Nom", value=client_data['nom'])
    new_email = st.text_input("Email", value=client_data['email'])
    new_ville = st.text_input("Ville", value=client_data['ville'])

```

Un formulaire **pré-rempli** avec les informations du client. L'utilisateur peut modifier ce qu'il veut.

```
coll, col2 = st.columns(2)
```

Sépare l'espace du formulaire en deux colonnes côte à côte.

Colonne 1 – Modifier

```

with coll:

    if st.form_submit_button("Modifier"):
        try:
            update_client(selected_id, new_nom, new_email, new_ville)
            st.success(" Client modifié avec succès !")
            st.rerun()

```

Si on clique sur “Modifier” :

- Appel à `update_client(...)` pour mettre à jour les infos en base.
- Message de succès + rechargement de la page.

```

except Exception as e:
    st.error(f"Erreur lors de la modification: {str(e)}")

```

Gestion d'erreur en cas d'échec de la modification.

Colonne 2 – Supprimer

```

with col2:

    if st.form_submit_button("Supprimer"):
        try:
            delete_client(selected_id)
            st.success(" Client supprimé avec succès !")
            st.rerun()

```

Si on clique sur “Supprimer” :

- Appel à `delete_client(...)` pour supprimer ce client.
- Message de succès + rechargement.

```

except Exception as e:
    st.error(f"Erreur lors de la suppression: {str(e)}")

```

Gestion d'erreur si la suppression échoue.

Bloc global except

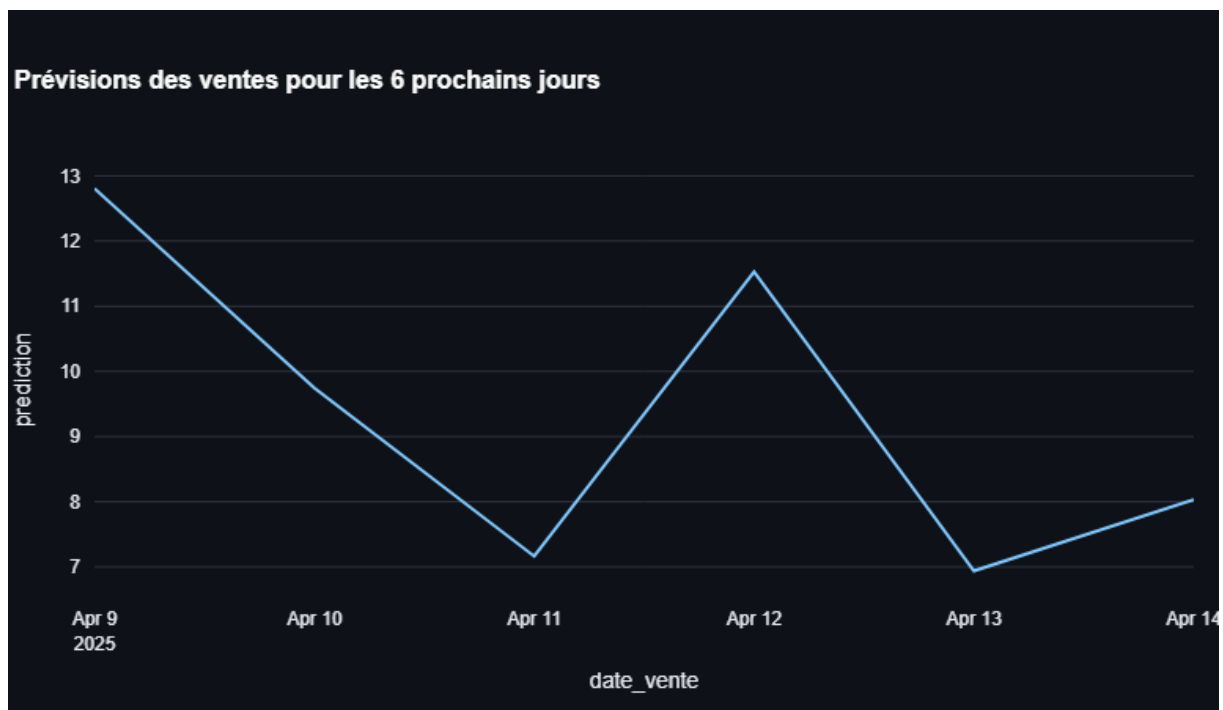
except Exception as e:

```
st.error(f"Erreur dans la gestion des clients: {str(e)}")
```

Capture **toute erreur non prévue** et affiche un message global d'erreur.

### 15. Prévisions : `show_forecasts()`

- Utilise un module de machine learning pour prédire les ventes futures
- Affiche les résultats sous forme de graphique et tableau



	date_vente	prediction
0	2025-04-09 00:00:00	12.805
1	2025-04-10 00:00:00	9.745
2	2025-04-11 00:00:00	7.165
3	2025-04-12 00:00:00	11.53
4	2025-04-13 00:00:00	6.94
5	2025-04-14 00:00:00	8.035

Fonction show\_forecasts()  
def show\_forecasts():

Définition de la fonction principale pour gérer l'interface de prévision des ventes.

1. Titre de la section  
st.header("Prévisions des ventes")

Affiche un **titre en grand** sur la page Streamlit.

2. Récupération des ventes  
df\_ventes = get\_ventes()

Appelle une fonction get\_ventes() qui retourne l'historique des ventes sous forme de **DataFrame** (avec au moins les colonnes date\_vente, quantité, etc.).

3. Vérification du volume de données  
if len(df\_ventes) < 30:  
    st.warning("Pas assez de données historiques pour faire des prévisions (minimum 30 jours)")  
    return

- Si la base contient **moins de 30 lignes** de ventes, on affiche un **message d'avertissement**.
- return arrête l'exécution de la fonction : on ne fait pas de prévision.

4. Chargement de la prévision



```
with st.spinner("Calcul des prévisions..."):
    forecast, mae = forecast_sales(df_ventes)
```

- Affiche un **spinner de chargement** pendant l'exécution.
- Appel à la fonction `forecast_sales(df_ventes)` :
  - Elle retourne deux éléments :
    - `forecast` : un DataFrame contenant les dates futures + les prédictions (prediction).
    - `mae` : la **Mean Absolute Error**, un indicateur d'erreur du modèle.

#### 5. Résultat du modèle

if forecast is not None:

```
    st.success(f"Prévisions calculées (MAE: {mae:.2f})")
```

Si la prévision a fonctionné (non nulle) :

- Affiche un **message de succès**
- Affiche la MAE arrondie à deux décimales.

#### 6. Affichage du graphique

```
fig = px.line(
    forecast,
    x='date_vente',
    y='prediction',
    title='Prévisions des ventes pour les 6 prochains jours'
)
```

Création d'un **graphique en ligne** avec **Plotly Express** :

- **x** = `date_vente` (la date future)
- **y** = `prediction` (valeur prédite)
- Titre : "Prévisions des ventes pour les 6 prochains jours"

```
st.plotly_chart(fig, use_container_width=True)
```

Affiche le graphique dans Streamlit en prenant toute la largeur de la page.

#### 7. Affichage du tableau des prévisions

```
st.dataframe(forecast)
```

Affiche le tableau complet des prévisions sous forme de **DataFrame** Streamlit.

#### 8. En cas d'erreur

else:

```
    st.error("Erreur lors du calcul des prévisions")
```

Si le modèle retourne None, affiche un message d'erreur.

Fonction `forecast_sales(df_ventes)`

```
python
CopierModifier
def forecast_sales(df_ventes):
```

Définition d'une fonction qui prend en entrée un **DataFrame des ventes** et retourne :

- Les prévisions sous forme de DataFrame
- La **MAE** (Mean Absolute Error) du modèle

#### 1. Nettoyage et préparation des données

```
df_ventes['date_vente'] = pd.to_datetime(df_ventes['date_vente'])
df_ventes = df_ventes.sort_values('date_vente')
```

- Convertit les dates en objets datetime si ce n'est pas déjà le cas.
- Trie les ventes par date pour une série chronologique propre.

#### 1. Regrouper les ventes par jour

```
daily_sales = df_ventes.groupby('date_vente')['quantite'].sum().reset_index()
```

Agrège les données par date pour obtenir la **quantité totale vendue par jour**. On évite ainsi les doublons ou ventes multiples dans la même journée.

#### 3. Création de variables

```
daily_sales['day'] = np.arange(len(daily_sales))
```

Ajoute une colonne day qui représente les jours sous forme d'index numériques (0, 1, 2...). Cela permet de faire des prévisions **avec une régression linéaire** ou un modèle time series simple.

#### 4. Entraînement du modèle

```
X = daily_sales[['day']]
y = daily_sales['quantite']
```

```
model = LinearRegression()
model.fit(X, y)
```

- X est la variable indépendante (numéro du jour).
- y est la quantité vendue.
- On entraîne un **modèle de régression linéaire** sur ces données (très simple et rapide pour une démo).

#### 5. Évaluation du modèle

```
y_pred = model.predict(X)
mae = mean_absolute_error(y, y_pred)
```

- On prédit les ventes déjà connues avec le modèle.
- On calcule la **MAE** : erreur moyenne absolue entre les vraies valeurs et les prévisions du modèle.

- Cela donne un **indice de performance** (plus c'est bas, mieux c'est).

## 6. Génération des prévisions futures

python

CopierModifier

```
future_days = np.arange(len(daily_sales), len(daily_sales)+6)
future_dates = [daily_sales['date_vente'].max() + timedelta(days=i+1) for i in range(6)]
```

- future\_days = index des **6 prochains jours**.
- future\_dates = les **dates correspondantes** à partir du dernier jour connu.

python

CopierModifier

```
future_preds = model.predict(future_days.reshape(-1, 1))
```

Utilise le modèle pour **prédire les quantités** pour les 6 prochains jours.

## 7. Format final du résultat

```
forecast_df = pd.DataFrame({
    'date_vente': future_dates,
    'prediction': future_preds
})
```

Crée un DataFrame avec les dates futures et leurs prédictions.

## 8. Retour du résultat

python

CopierModifier

```
return forecast_df, mae
```

La fonction retourne :

- Les prévisions
- L'erreur MAE

## 9. Gestion d'erreur

python

CopierModifier

```
except Exception as e:
    return None, None
```

Si une erreur se produit pendant l'exécution, on retourne deux None pour que la fonction show\_forecasts() affiche un message d'erreur.

importation des bibliothèques

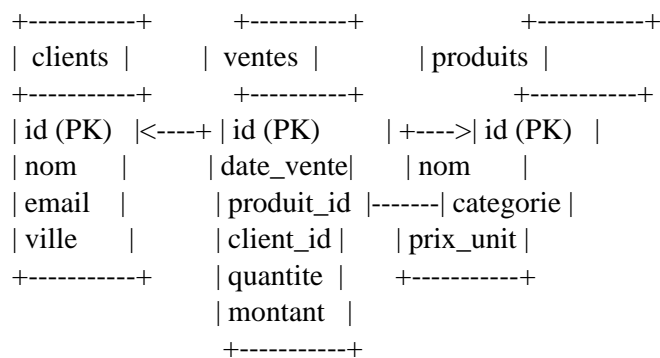
```
import pandas as pd
from datetime import timedelta
from sklearn.ensemble import RandomForestRegressor
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
```

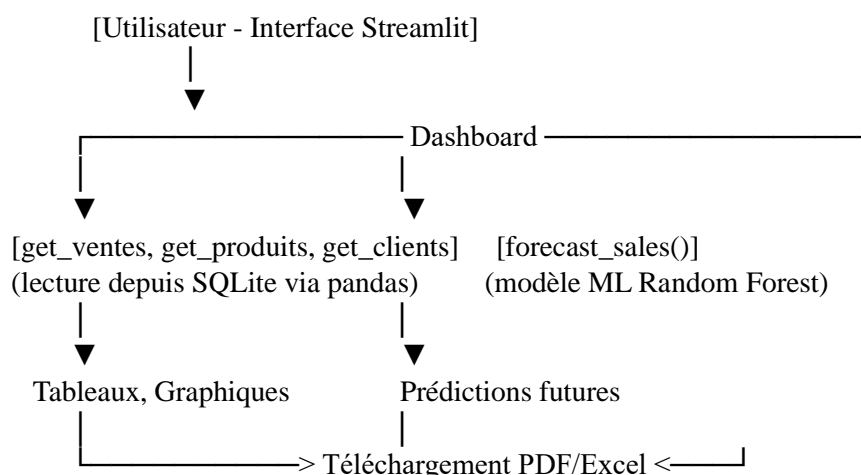
### Objectif :

- pandas : Bibliothèque pour la manipulation et l'analyse de données. Elle est utilisée ici pour traiter les données de ventes.
- timedelta : Module de la bibliothèque datetime, utilisé pour manipuler des dates en ajoutant ou en soustrayant des jours.
- RandomForestRegressor : Un modèle d'arbres décisionnels en forêts aléatoires pour effectuer des régressions, ici utilisé pour prédire le montant des ventes.
- train\_test\_split : Fonction pour diviser les données en ensembles d'entraînement et de test.
- mean\_absolute\_error : Fonction pour calculer l'erreur absolue moyenne entre les prédictions et les valeurs réelles.

## 1. Schéma de la base de données (ERD)

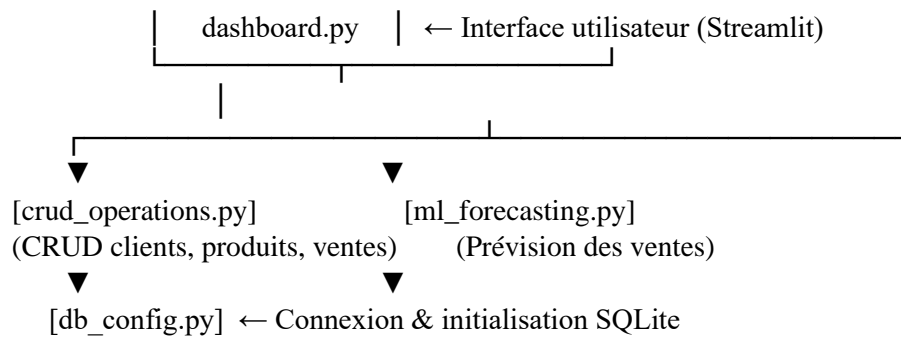


## 2. Diagramme de flux fonctionnel



## 3. Architecture modulaire





## 4. Wireframe de l'interface utilisateur

+-----+	
MENU (Sidebar)	
[Tableau de bord]	
[Gestion des ventes]	
[Gestion des produits]	
[Gestion des clients]	
[Prévisions]	
+-----+	
Affichage central (contenu)	
- Filtres par date, catégorie	
- Graphiques évolutifs (Plotly)	
- Tableaux dynamiques (AgGrid)	
- Export Excel / PDF	
+-----+	