Description du Processus de Développement - Projet EVIMERIA

Introduction

Ce document détaille le processus de développement adopté pour la création du site e-commerce EVIMERIA. Il couvre les méthodologies, les étapes clés de la conception et de l'implémentation, ainsi que les défis potentiels et les solutions envisagées. L'objectif est de fournir une vue d'ensemble claire de la manière dont le projet a été structuré et mené à bien.

1. Méthodologie et Approche de Développement

Le développement du projet EVIMERIA a suivi une approche itérative et incrémentale, s'inspirant des principes Agile. Cela a permis une flexibilité dans la gestion des fonctionnalités et une adaptation continue aux besoins du projet.

- **Planification Initiale :**
    - Définition des besoins et des objectifs du site (basée sur le `cahier_de_charge.Md`).
    - Identification des fonctionnalités clés (gestion des produits, utilisateurs, commandes, panier, etc.).
    - Choix de la stack technologique (Django pour le backend, React pour le frontend, PostgreSQL pour la base de données, Docker pour la conteneurisation).
    - `[Insérer capture d'écran du cahier des charges ou d'un tableau de planification initial]`
- **Choix d'Architecture et Justifications :**
    - **Backend (Django & Django REST Framework) :**
        - **Robustesse et Écosystème :** Django est un framework Python "batteries included", offrant de nombreuses fonctionnalités prêtes à l'emploi (ORM, admin, sécurité), ce qui accélère le développement initial. Son écosystème mature (DRF, Celery, etc.) est idéal pour construire des API RESTful complexes et des applications web solides.
        - **Scalabilité :** Bien que monolithique au départ, Django peut être structuré en applications modulaires, facilitant une future évolution vers des microservices si nécessaire. Il est capable de gérer une charge importante avec une configuration appropriée.
        - **Sécurité :** Django intègre des protections contre les vulnérabilités web courantes (XSS, CSRF, SQL Injection).
        - **Python :** La popularité et la lisibilité de Python facilitent la collaboration et la maintenance.
    - **Frontend (React & Vite) :**
        - **Performance et Interactivité :** React, avec son Virtual DOM, permet de créer des interfaces utilisateur dynamiques et performantes, cruciales pour une expérience e-commerce fluide.

        - **Écosystème Riche :** React dispose d'une vaste communauté et de nombreuses bibliothèques (Redux pour la gestion d'état, React Router pour la navigation, Framer Motion pour les animations) qui enrichissent les possibilités.
        - **Vite :** Choisi pour sa rapidité de build et son expérience de développement améliorée (Hot Module Replacement instantané).
        - **SPA (Single Page Application) :** Offre une navigation plus rapide et une sensation d'application native.
    - **Base de Données (PostgreSQL) :**
        - **Fiabilité et ACID :** PostgreSQL est réputé pour sa robustesse, son respect des propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité), essentielles pour un site e-commerce gérant des transactions.
        - **Fonctionnalités Avancées :** Supporte des types de données complexes, des indexations performantes, et des fonctionnalités avancées utiles pour des requêtes analytiques ou des recherches complexes.
        - **Scalabilité :** Offre de bonnes options de scalabilité verticale et horizontale (via réplication, sharding).
    - **Conteneurisation (Docker & Docker Compose) :**
        - **Portabilité et Reproductibilité :** Assure que l'environnement de développement est identique à l'environnement de production, réduisant les problèmes de type "ça marche sur ma machine".
        - **Isolation des Services :** Permet de gérer chaque composant de l'application (backend, frontend, BDD) comme un service isolé.
        - **Déploiement Simplifié :** Facilite le déploiement sur des plateformes cloud comme Railway.
    - `[Insérer capture d'écran d'un schéma d'architecture global du projet]`

- **Développement par Modules/Fonctionnalités :**
    - Décomposition du projet en modules distincts (ex: authentification, catalogue produits, gestion du panier, processus de commande).
    - Développement itératif de chaque module avec des phases de conception, codage, et tests unitaires.

- **Collaboration et Gestion de Code :**
    - Utilisation de Git pour le contrôle de version, avec des branches pour les nouvelles fonctionnalités (`feature-branches`) et les corrections (`bugfix-branches`).
    - Hébergement du code sur GitHub pour faciliter la collaboration et le suivi des modifications.
    - Revues de code (pull requests) pour assurer la qualité et la cohérence.
    - `[Insérer capture d'écran d'une pull request sur GitHub ou de l'arborescence des branches Git]`

- **Tests :**
    - Mise en place de tests unitaires pour le backend (avec Django Test) et le frontend (potentiellement Jest/React Testing Library).
    - Tests d'intégration pour vérifier l'interaction entre le backend et le frontend.
    - Tests manuels pour valider l'expérience utilisateur (UX) et l'interface utilisateur (UI).

2. Étapes Principales de la Création du Site

La création du site EVIMERIA a été divisée en plusieurs grandes phases, impliquant le développement du backend, du frontend, leur intégration, et enfin le déploiement.

2.1. Développement du Backend (Django)

- **Configuration du Projet Django :**
  - Initialisation du projet et des applications Django (ex: `users`, `products`, `orders`).
  - Configuration de la base de données PostgreSQL.
  - Mise en place des variables d'environnement (sécurisation des clés et configurations).
  - `[Insérer capture d'écran de la structure du projet Django ou du fichier settings.py]`
- **Modélisation des Données :**
  - Définition des modèles Django (ORM) pour les produits, catégories, utilisateurs, commandes, adresses, etc.
  - Génération et application des migrations de base de données.
  - `[Insérer capture d'écran d'un fichier models.py ou du schéma de la base de données]`

  Par exemple, un modèle `Product` pourrait ressembler à ceci (extrait de `products/models.py`) :

```
from django.db import models
from categories.models import Category # En supposant une app categories

class Product(models.Model):
    category = models.ForeignKey(Category, related_name='products',
on_delete=models.CASCADE)
    name = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255, unique=True)
    description = models.TextField(blank=True, null=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock_quantity = models.PositiveIntegerField(default=0)
    image_url_cloudinary = models.CharField(max_length=500, blank=True,
null=True) # Ou ImageField avec CloudinaryStorage
    is_available = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ('-created_at',)
        verbose_name = 'Produit'
        verbose_name_plural = 'Produits'

    def __str__(self):
        return self.name
```

- **Développement des APIs RESTful avec DRF :**
    - Création des `serializers` pour convertir les modèles Django en JSON et vice-versa.
    - Développement des `ViewSets` ou `APIViews` pour exposer les endpoints de l'API (ex: `/api/products/`, `/api/users/register/`, `/api/orders/`).
    - Mise en place de l'authentification (JWT avec `djangorestframework-simplejwt`) et des permissions.
    - Gestion du filtrage, de la pagination et de la recherche pour les listes de données.
    - Documentation de l'API avec `drf-yasg` (Swagger).
    - `[Insérer capture d'écran d'un serializer, d'une vue DRF, ou de l'interface Swagger]`

    Voici un exemple de `ProductSerializer` (extrait de `products/serializers.py`) :

```
from rest_framework import serializers
from .models import Product
from categories.serializers import CategorySerializer # En supposant un
serializer pour Category

class ProductSerializer(serializers.ModelSerializer):
    category = CategorySerializer(read_only=True) # Afficher les détails
de la catégorie
    category_id = serializers.PrimaryKeyRelatedField(
        queryset=Category.objects.all(), source='category',
write_only=True
    ) # Permettre de lier par ID à la création/MAJ

    class Meta:
        model = Product
        fields = (
            'id', 'category', 'category_id', 'name', 'slug',
'description',
            'price', 'stock_quantity', 'image_url_cloudinary',
'is_available',
            'created_at', 'updated_at'
        )
        read_only_fields = ('slug', 'created_at', 'updated_at')

    def validate_price(self, value):
        if value <= 0:
            raise serializers.ValidationError("Le prix doit être
supérieur à zéro.")
        return value

    def validate_stock_quantity(self, value):
        if value < 0:
            raise serializers.ValidationError("La quantité en stock ne
peut être négative.")
        return value
```

- **Logique Métier :**
    - Implémentation des fonctionnalités spécifiques : gestion du panier, calcul des totaux, création des commandes, gestion des stocks.
    - Intégration avec des services tiers (ex: Cloudinary pour le stockage des images produits).

2.2. Développement du Frontend (React)

- **Initialisation du Projet React :**
    - Mise en place du projet avec Vite pour un environnement de développement rapide.
    - Configuration de TypeScript pour un typage statique.
    - Installation des dépendances clés (React Router, Redux Toolkit, Axios, Tailwind CSS).
    - `[Insérer capture d'écran de la structure du projet React ou du fichier package.json]`
- **Conception de l'Interface Utilisateur (UI) et de l'Expérience Utilisateur (UX) :**
    - Maquettage des différentes pages (accueil, liste produits, détail produit, panier, checkout, profil utilisateur).
    - Développement des composants React réutilisables (boutons, cartes produit, formulaires, etc.).
    - Intégration de Tailwind CSS pour un styling moderne et responsive.
    - `[Insérer capture d'écran d'une maquette, d'un composant React ou d'une page stylée avec Tailwind]`

  Exemple de composant `ProductCard.tsx` simplifié :

```
import React from 'react';

interface Product {
  id: number;
  name: string;
  price: string; // Ou number, formatté ensuite
  image_url_cloudinary?: string;
  slug: string;
}

interface ProductCardProps {
  product: Product;
  onAddToCart: (productId: number) => void;
}

const ProductCard: React.FC<ProductCardProps> = ({ product, onAddToCart
}) => {
  return (
    <div className="border rounded-lg p-4 shadow-lg hover:shadow-xl
transition-shadow duration-300">
      <img
        src={product.image_url_cloudinary ||
```

```
'https/via.placeholder.com/300'}
        alt={product.name}
        className="w-full h-48 object-cover rounded-md mb-4"
      />
      <h3 className="text-lg font-semibold mb-2">{product.name}</h3>
      <p className="text-gray-700 mb-3">{product.price} €</p>
      {/* Link vers la page détail produit avec React Router */}
      <a href={`/products/${product.slug}`} className="text-indigo-600
hover:text-indigo-800 mb-3 block">
        Voir détails
      </a>
      <button
        onClick={() => onAddToCart(product.id)}
        className="w-full bg-blue-500 hover:bg-blue-700 text-white font-
bold py-2 px-4 rounded"
      >
        Ajouter au Panier
      </button>
    </div>
  );
};

export default ProductCard;
```

- **Gestion de l'État :**
  - Utilisation de Redux Toolkit pour gérer l'état global de l'application (panier, informations utilisateur, état de l'authentification).
  - Création des `slices` et des `reducers` pour les différentes parties de l'état.
  - `[Insérer capture d'écran d'un slice Redux]`

  Extrait d'un `cartSlice.ts` pour la gestion du panier :

```
import { createSlice, PayloadAction } from '@reduxjs/toolkit';

interface CartItem {
  productId: number;
  name: string;
  price: number;
  quantity: number;
  image?: string;
}

interface CartState {
  items: CartItem[];
  totalQuantity: number;
  totalAmount: number;
}

const initialState: CartState = {
```

```
  items: [],
  totalQuantity: 0,
  totalAmount: 0,
};

const cartSlice = createSlice({
  name: 'cart',
  initialState,
  reducers: {
    addItemToCart(state, action: PayloadAction<Omit<CartItem,
'quantity'>>) {
      const newItem = action.payload;
      const existingItem = state.items.find(item => item.productId ===
newItem.productId);
      state.totalQuantity++;
      state.totalAmount += newItem.price;

      if (!existingItem) {
        state.items.push({ ...newItem, quantity: 1 });
      } else {
        existingItem.quantity++;
      }
    },
    removeItemFromCart(state, action: PayloadAction<{ id: number }>) {
      const idToRemove = action.payload.id;
      const existingItem = state.items.find(item => item.productId ===
idToRemove);
      if (existingItem) {
        state.totalQuantity--;
        state.totalAmount -= existingItem.price;
        if (existingItem.quantity === 1) {
          state.items = state.items.filter(item => item.productId !==
idToRemove);
        } else {
          existingItem.quantity--;
        }
      }
    },
    clearCart(state) {
      state.items = [];
      state.totalQuantity = 0;
      state.totalAmount = 0;
    }
    // Autres reducers : updateQuantity, etc.
  },
});

export const { addItemToCart, removeItemFromCart, clearCart } =
cartSlice.actions;
export default cartSlice.reducer;
```

- **Interaction avec l'API Backend :**

- Utilisation d'Axios pour effectuer les requêtes HTTP vers l'API Django.
- Gestion des réponses, des erreurs, et mise à jour de l'état de l'application en conséquence.
- Stockage sécurisé des tokens d'authentification.

- **Routage :**
  - Mise en place de React Router DOM pour la navigation entre les différentes pages de l'application (SPA - Single Page Application).
  - Définition des routes publiques et protégées (nécessitant une authentification).

2.3. Intégration Backend et Frontend

- **Configuration des CORS :**
  - Utilisation de `django-cors-headers` pour permettre les requêtes depuis le domaine du frontend.
- **Tests d'Intégration :**
  - Vérification du bon fonctionnement des flux de données entre le frontend et le backend (inscription, connexion, ajout au panier, passage de commande).
  - Débogage des problèmes de communication et de format de données.

2.4. Conteneurisation avec Docker

- **Création des Dockerfiles :**
  - Un `Dockerfile` pour le backend Django (incluant Gunicorn, les dépendances Python).
  - Un `Dockerfile` pour le frontend React (incluant Node.js, la compilation des assets statiques, et un serveur comme Nginx ou Serve pour servir les fichiers buildés).
  - `[Insérer capture d'écran d'un des Dockerfiles]`

  Exemple de `Dockerfile` pour le backend Django :

```
# Étape 1: Builder l'environnement Python
FROM python:3.10-slim-buster AS builder

WORKDIR /app

# Installer les dépendances système nécessaires pour certaines
bibliothèques Python
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential libpq-dev \
    && rm -rf /var/lib/apt/lists/*

# Copier le fichier des dépendances et les installer
COPY ./backend/requirements.txt .
RUN pip wheel --no-cache-dir --wheel-dir /app/wheels -r requirements.txt

# Étape 2: Créer l'image finale
FROM python:3.10-slim-buster

WORKDIR /app
```

```
# Créer un utilisateur non-root
RUN groupadd -r appuser && useradd --no-log-init -r -g appuser appuser

# Copier les roues Python pré-compilées de l'étape builder
COPY --from=builder /app/wheels /wheels
COPY --from=builder /app/requirements.txt .

# Installer les roues Python (plus rapide que de recompiler)
RUN pip install --no-cache /wheels/*

# Copier le reste du code de l'application backend
COPY ./backend/ .

# Donner la propriété du répertoire de l'application à l'utilisateur non-
root
RUN chown -R appuser:appuser /app
USER appuser

# Exposer le port que Gunicorn utilisera
EXPOSE 8000

# Commande par défaut pour lancer Gunicorn
# Assurez-vous que jaelleshop.wsgi est le bon chemin vers votre fichier
WSGI
CMD ["gunicorn", "--bind", ":8000", "--workers", "3",
"jaelleshop.wsgi:application"]
```

- **Orchestration avec Docker Compose :**
  - Création d'un fichier `docker-compose.yml` pour définir et lier les services : backend, frontend, et base de données PostgreSQL.
  - Configuration des réseaux, des volumes pour la persistance des données (PostgreSQL), et des variables d'environnement.
  - Facilitation du lancement de l'environnement de développement complet avec une seule commande (`docker-compose up`).
  - `[Insérer capture d'écran du fichier docker-compose.yml]`

  Extrait du `docker-compose.yml` :

```
version: '3.8'

services:
  backend:
    build:
      context: .
      dockerfile: ./backend/Dockerfile
    restart: unless-stopped
    env_file:
      - .env # Contient les variables d'environnement pour Django
```

```
    volumes:
      - ./backend:/app # Montage pour le développement local
    ports:
      - "8000:8000"
    depends_on:
      db:
        condition: service_healthy # Attendre que la BDD soit prête

  frontend:
    build:
      context: .
      dockerfile: ./frontend/Dockerfile
    restart: unless-stopped
    ports:
      - "3000:3000" # Port Vite en dev, ou 80 si Nginx en prod dans le
conteneur
    volumes:
      - ./frontend:/app # Montage pour le développement local
      - /app/node_modules # Éviter d'écraser node_modules du conteneur
    depends_on:
      - backend
    environment:
      - VITE_API_BASE_URL=http://localhost:8000/api # Exemple pour Vite

  db:
    image: postgres:15-alpine
    restart: unless-stopped
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - .env # Contient POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_DB
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d
${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5

volumes:
  postgres_data:
```

2.5. Déploiement

- **Choix de la Plateforme :**
    - Utilisation de Railway pour le déploiement simplifié des services conteneurisés.
- **Configuration du Déploiement sur Railway :**
    - Liaison du dépôt GitHub au projet Railway.

    - Configuration des services (backend, frontend, base de données) en utilisant les Dockerfiles.
    - Gestion des variables d'environnement pour la production (clés API, secret key, configuration de la base de données de production).
    - Configuration des domaines et du HTTPS.
    - `[Insérer capture d'écran du dashboard Railway montrant les services déployés]`

- **Mise en Place de la CI/CD (Continuous Integration/Continuous Deployment) :**
    - Utilisation de GitHub Actions pour automatiser le build et le déploiement de l'application à chaque `push` sur la branche principale (ou une branche de déploiement).
    - Automatisation des tests (unitaires, intégration) dans le pipeline de CI/CD pour garantir la non-régression.
    - `[Insérer capture d'écran d'un workflow GitHub Actions avec ses étapes]`

3. Défis Rencontrés et Solutions

Au cours du développement d'un projet e-commerce complexe comme EVIMERIA, plusieurs défis peuvent survenir. Voici quelques exemples courants et comment ils ont pu être abordés ou anticipés :

- **Défi : Gestion de l'état complexe dans le frontend (Panier, Filtres, Authentification).**
    - **Solution :** Adoption de Redux Toolkit pour une gestion centralisée, structurée et prévisible de l'état. Utilisation de `selectors` optimisés (ex: `reselect`) pour éviter les re-calculs inutiles. Persistence de certaines parties de l'état (panier, session utilisateur) dans le `localStorage` pour une meilleure UX.
    - `[Insérer capture d'écran illustrant l'utilisation de Redux DevTools, la structure des stores, ou un exemple de slice complexe]`

- **Défi : Sécurisation de l'API et gestion fine des accès (Authentification/Autorisation).**
    - **Solution :**
        - **Authentification :** Tokens JWT (via `djangorestframework-simplejwt`) avec des durées de vie courtes pour les `access tokens` et des `refresh tokens` pour maintenir la session. Stockage sécurisé des tokens côté client (ex: `HttpOnly cookies` si possible, sinon `localStorage` avec des mesures additionnelles).
        - **Autorisation :** Mise en place de classes de permission personnalisées dans Django REST Framework pour contrôler l'accès aux endpoints et aux objets spécifiques (ex: un utilisateur ne peut modifier que ses propres commandes).
        - **Protection contre les attaques courantes :** Validation des entrées, utilisation de l'ORM Django pour prévenir les injections SQL, configuration de `django-cors-headers` de manière restrictive, protection CSRF activée par Django.
        - **HTTPS :** Chiffrement systématique des communications.
    - `[Insérer capture d'écran d'une configuration de permissions DRF, d'un exemple de requête authentifiée, ou des réglages de sécurité JWT]`

- **Défi : Configuration du déploiement multi-conteneurs et gestion des environnements (Développement, Staging, Production).**

- **Solution :**
  - **Docker Compose :** Utilisé pour définir et orchestrer l'ensemble des services localement.
  - **Variables d'environnement :** Utilisation intensive de variables d'environnement (gérées via des fichiers `.env` localement et les secrets de la plateforme de déploiement comme Railway) pour distinguer les configurations (clés API, paramètres de base de données, niveaux de DEBUG).
  - **Builds Docker optimisés :** Utilisation de `multi-stage builds` dans les Dockerfiles pour réduire la taille des images finales et améliorer la sécurité (ne pas inclure les dépendances de build dans l'image de production).
- `[Insérer capture d'écran montrant les services interconnectés dans Docker, une configuration de variables d'environnement sur Railway, ou un extrait de Dockerfile avec multi-stage build]`

- **Défi : Optimisation des performances (Temps de chargement des pages, Requêtes base de données).**
  - **Solutions Côté Backend :**
    - **Requêtes optimisées :** Utilisation de `select_related` et `prefetch_related` de Django ORM pour réduire le nombre de requêtes SQL (N+1 problem).
    - **Indexation :** Ajout d'index pertinents sur les champs fréquemment interrogés en base de données (ex: clés étrangères, champs de filtre).
    - **Mise en cache :** Potentielle utilisation de Redis ou Memcached pour mettre en cache les résultats de requêtes coûteuses ou les données fréquemment accédées.
  - **Solutions Côté Frontend :**
    - **Optimisation des images :** Utilisation de Cloudinary pour le stockage, l'optimisation (compression, format WebP, redimensionnement à la volée) et la distribution (via CDN).
    - **Lazy Loading :** Chargement différé des images et des composants non visibles initialement.
    - **Code Splitting :** Division du code JavaScript en plus petits morceaux chargés à la demande (fonctionnalité native de Vite/React).
    - **Minification et Compression :** Minification des assets (JS, CSS) et compression Gzip/Brotli au niveau du serveur web (Nginx).
  - `[Insérer capture d'écran de l'interface Cloudinary, du code implémentant le lazy loading, ou d'un outil d'analyse de performance web comme Lighthouse montrant les améliorations]`

- **Défi : Cohérence de l'interface utilisateur sur différents appareils (Responsive Design).**
  - **Solution :** Utilisation intensive de Tailwind CSS (approcche "mobile-first"), un framework "utility-first" qui facilite la création de designs responsives grâce à ses classes utilitaires pour les breakpoints, flexbox, grid. Tests réguliers sur différents navigateurs et tailles d'écran (outils de développement des navigateurs, simulateurs).
  - `[Insérer capture d'écran montrant le site sur mobile et desktop, ou un exemple de code Tailwind pour le responsive]`

- **Défi : Synchronisation entre les développements backend et frontend et gestion des contrats d'API.**
    - **Solution :**
        - **Documentation d'API :** Utilisation de `drf-yasg` (Swagger/OpenAPI) pour générer une documentation interactive et à jour de l'API. Cela sert de référence unique pour l'équipe frontend.
        - **Communication :** Réunions courtes et régulières pour discuter des besoins en API, des modifications et des blocages.
        - **Développement Parallèle :** Une fois le contrat d'API défini pour une fonctionnalité, les équipes backend et frontend peuvent travailler en parallèle, le frontend utilisant des données mockées en attendant l'implémentation réelle de l'API.
        - **Versioning de l'API (anticipation) :** Prévoir une stratégie de versioning (ex: `/api/v1/...`, `/api/v2/...`) pour introduire des changements majeurs sans casser les versions existantes du frontend.
    - `[Insérer capture d'écran de l'interface Swagger de l'API]`
- **Défi : Gestion des transactions et intégrité des données pour les commandes.**
    - **Solution :**
        - **Transactions Atomiques :** Utilisation des transactions de base de données Django (`transaction.atomic`) pour s'assurer que toutes les opérations liées à une commande (ex: création de la commande, mise à jour du stock, enregistrement du paiement) sont complétées avec succès ou annulées en bloc en cas d'erreur.
        - **Validation Rigoureuse :** Validation des données à plusieurs niveaux (frontend, backend serializers, modèles Django) pour garantir la cohérence.
        - **Gestion des Concurrences :** Potentielle utilisation de mécanismes de verrouillage optimiste ou pessimiste pour la mise à jour des stocks si la concurrence est élevée.
    - `[Insérer un extrait de code montrant l'utilisation de transaction.atomic dans Django]`

    Exemple de gestion de création de commande avec transaction atomique (simplifié) :

```
from django.db import transaction
from django.shortcuts import get_object_or_404
# Supposons que Order, OrderItem, Product sont vos modèles Django
# et Cart est une classe ou un mécanisme pour gérer le panier en
session/BDD

def create_order_from_cart(user, cart_data, shipping_address_instance):
    try:
        with transaction.atomic():
            # 1. Créer l'objet Commande
            order = Order.objects.create(
                user=user,
                shipping_address=shipping_address_instance,
                # ... autres champs de commande (total initial, etc.)
                total_amount=0 # Sera calculé ensuite
```

```
            )

            total_order_amount = 0

            # 2. Créer les LignesDeCommande (OrderItem) à partir du
panier
            for item_in_cart in cart_data.items(): # item_in_cart
pourrait être {product_id, quantity}
                product = get_object_or_404(Product,
pk=item_in_cart['product_id'])

                if product.stock_quantity < item_in_cart['quantity']:
                    raise ValueError(f"Stock insuffisant pour
{product.name}")

                order_item = OrderItem.objects.create(
                    order=order,
                    product=product,
                    quantity=item_in_cart['quantity'],
                    price_at_purchase=product.price # Prix au moment de
l'achat
                )
                total_order_amount += order_item.price_at_purchase *
order_item.quantity

                # 3. Décrémenter le stock du produit
                product.stock_quantity -= item_in_cart['quantity']
                product.save()

            # 4. Mettre à jour le montant total de la commande
            order.total_amount = total_order_amount
            order.status = 'pending_payment' # ou 'processing' si
paiement déjà fait
            order.save()

            # 5. Potentiellement vider le panier ici
            # cart.clear()

            return order # Retourner la commande créée

    except ValueError as e: # Gérer spécifiquement les erreurs de stock
ou autres erreurs métier
        # Logger l'erreur, notifier l'utilisateur, etc.
        # La transaction sera automatiquement annulée (rollback) grâce au
with transaction.atomic()
        print(f"Erreur métier lors de la création de la commande : {e}")
        return None
    except Exception as e:
        # Gérer les autres exceptions inattendues
        # Logger l'erreur
        print(f"Erreur inattendue lors de la création de la commande :
{e}")
        return None
```

- **Défi : Scalabilité de la base de données et gestion d'un catalogue produit grandissant.**
    - **Solution (Anticipation & Stratégies) :**
        - **Modélisation Efficace :** Conception de schémas de base de données optimisés pour les requêtes courantes.
        - **Archivage :** Stratégies d'archivage pour les anciennes données (ex: vieilles commandes) si nécessaire.
        - **Réplication :** Utilisation de réplicas en lecture pour décharger la base de données principale pour les opérations de lecture intensives.
        - **Sharding (si extrême) :** Division de la base de données en plusieurs instances plus petites (solution plus complexe, pour des besoins très importants).
        - **Monitoring :** Surveillance continue des performances de la base de données pour identifier les goulots d'étranglement.

4. Diagrammes Clés du Projet

Cette section présente les diagrammes essentiels qui illustrent l'architecture fonctionnelle et la structure des données du projet EVIMERIA. Ces représentations textuelles sont destinées à être reproduites avec des outils de modélisation graphique.

4.1. Diagramme de Cas d'Utilisation Général

Ce diagramme montre les interactions principales entre les acteurs (Visiteur, Client Enregistré, Administrateur) et le système EVIMERIA.

- **Acteurs :**
    - `Visiteur` (Utilisateur non authentifié)
    - `Client` (Utilisateur authentifié, hérite des actions du Visiteur)
    - `Administrateur` (Utilisateur avec droits spécifiques pour la gestion du site)
- **Cas d'Utilisation (pour `Visiteur`) :**
    - `(Consulter le catalogue de produits)`
    - `(Rechercher des produits)` (par nom, catégorie)
    - `(Voir les détails d'un produit)` (description, prix, images, avis)
    - `(Ajouter un produit au panier)`
    - `(Voir le panier)`
    - `(S'inscrire pour devenir Client)`
    - `(Se connecter)`
- **Cas d'Utilisation (pour `Client`) (en plus de ceux du Visiteur) :**
    - `(Gérer son profil)`
        - `<<include>> (Modifier ses informations personnelles)`
        - `<<include>> (Gérer ses adresses de livraison/facturation)`
    - `(Passer une commande)`

        - `<<include>> (Valider le panier)`
        - `<<include>> (Sélectionner/Ajouter une adresse de livraison)`
        - `<<include>> (Sélectionner une méthode de paiement)`
        - `<<include>> (Effectuer le paiement)` --> (interagit avec un `Système de Paiement Externe`)
        - `<<include>> (Recevoir une confirmation de commande)`
    - `(Consulter son historique de commandes)`
    - `(Voir les détails d'une commande passée)`
    - `(Laisser un avis sur un produit acheté)`
    - `(Gérer sa liste de souhaits)` (optionnel)

- **Cas d'Utilisation (pour `Administrateur`) :**
    - `(Se connecter en tant qu'administrateur)`
    - `(Gérer le catalogue produits)`
        - `<<include>> (Ajouter un nouveau produit)`
        - `<<include>> (Modifier les informations d'un produit)`
        - `<<include>> (Supprimer un produit)`
        - `<<include>> (Gérer les stocks)`
        - `<<include>> (Gérer les images des produits)`
    - `(Gérer les catégories de produits)`
        - `<<include>> (Ajouter une catégorie)`
        - `<<include>> (Modifier une catégorie)`
        - `<<include>> (Supprimer une catégorie)`
    - `(Gérer les commandes des clients)`
        - `<<include>> (Consulter la liste des commandes)`
        - `<<include>> (Voir les détails d'une commande)`
        - `<<include>> (Mettre à jour le statut d'une commande)` (ex: payée, en préparation, expédiée, livrée, annulée)
    - `(Gérer les comptes clients)`
        - `<<include>> (Consulter la liste des clients)`
        - `<<include>> (Voir les informations d'un client)`
        - `<<include>> (Activer/Désactiver un compte client)`
    - `(Gérer les avis sur les produits)`
        - `<<include>> (Approuver/Rejeter un avis)`
    - `(Gérer les promotions/codes promo)` (optionnel)
    - `(Consulter les statistiques de vente)` (optionnel)

Relations typiques (à représenter avec des flèches appropriées) : `Acteur` --- `(Cas d'Utilisation)`; `(Cas A)` --- `<<include>>` ---> `(Cas B)`; `Client` --|> `Visiteur` (Généralisation/Héritage).

4.2. Diagramme Entité-Association (Modèle Conceptuel de Données pour PostgreSQL)

Ce diagramme décrit la structure de la base de données.

- **Entités et leurs principaux attributs (PK = Clé Primaire, FK = Clé Étrangère) :**
    1. **Utilisateur (User)**

- `id_user` (PK, Entier, Auto-incrémenté)
- `email` (Chaîne, Unique, Non nul)
- `password_hash` (Chaîne, Non nul)
- `first_name` (Chaîne, Optionnel)
- `last_name` (Chaîne, Optionnel)
- `is_staff` (Booléen, Défaut: False) (Pour l'accès admin Django)
- `is_active` (Booléen, Défaut: True)
- `date_joined` (Timestamp, Non nul)
- `last_login` (Timestamp, Optionnel)

2. **Adresse (Address)**

- `id_address` (PK, Entier, Auto-incrémenté)
- `user_id` (FK vers `Utilisateur.id_user`, Non nul)
- `street_address` (Chaîne, Non nul)
- `city` (Chaîne, Non nul)
- `postal_code` (Chaîne, Non nul)
- `country` (Chaîne, Non nul)
- `is_default_shipping` (Booléen, Défaut: False)
- `is_default_billing` (Booléen, Défaut: False)

3. **Categorie (Category)**

- `id_category` (PK, Entier, Auto-incrémenté)
- `name` (Chaîne, Unique, Non nul)
- `slug` (Chaîne, Unique, Non nul) (Pour les URLs)
- `description` (Texte, Optionnel)
- `parent_category_id` (FK vers `Categorie.id_category`, Optionnel, pour sous-catégories)

4. **Produit (Product)**

- `id_product` (PK, Entier, Auto-incrémenté)
- `category_id` (FK vers `Categorie.id_category`, Non nul)
- `name` (Chaîne, Non nul)
- `slug` (Chaîne, Unique, Non nul)
- `description` (Texte, Optionnel)
- `price` (Décimal, Non nul)
- `stock_quantity` (Entier, Non nul, Défaut: 0)
- `image_url_cloudinary` (Chaîne, Optionnel)
- `is_available` (Booléen, Défaut: True)
- `created_at` (Timestamp, Non nul, auto_now_add=True)
- `updated_at` (Timestamp, Non nul, auto_now=True)

5. **Commande (Order)**

- `id_order` (PK, UUID ou Entier Auto-incrémenté, Non nul)
- `user_id` (FK vers `Utilisateur.id_user`, Non nul)

- `shipping_address_id` (FK vers `Adresse.id_address`, Non nul)
- `billing_address_id` (FK vers `Adresse.id_address`, Optionnel)
- `created_at` (Timestamp, Non nul)
- `updated_at` (Timestamp, Non nul)
- `total_amount` (Décimal, Non nul)
- `status` (Chaîne, Non nul, ex: 'pending_payment', 'paid', 'processing', 'shipped', 'delivered', 'cancelled', 'refunded')
- `payment_intent_id` (Chaîne, Optionnel, pour suivi Stripe/Paypal)
- `shipping_method` (Chaîne, Optionnel)
- `tracking_number` (Chaîne, Optionnel)

6. **LigneDeCommande (OrderItem)**

- `id_order_item` (PK, Entier, Auto-incrémenté)
- `order_id` (FK vers `Commande.id_order`, Non nul)
- `product_id` (FK vers `Produit.id_product`, Non nul)
- `quantity` (Entier, Non nul)
- `price_at_purchase` (Décimal, Non nul) (Important pour l'historique des prix)

7. **AvisProduit (Review)**

- `id_review` (PK, Entier, Auto-incrémenté)
- `product_id` (FK vers `Produit.id_product`, Non nul)
- `user_id` (FK vers `Utilisateur.id_user`, Non nul)
- `rating` (Entier, Non nul, contrainte: 1-5)
- `comment` (Texte, Optionnel)
- `created_at` (Timestamp, Non nul)

- **Relations et Cardinalités Principales (à représenter avec des connecteurs et symboles appropriés) :**
  - `Utilisateur` (1) -- (0,N) `Adresse` : Un utilisateur peut avoir zéro ou plusieurs adresses.
  - `Utilisateur` (1) -- (0,N) `Commande` : Un utilisateur peut passer zéro ou plusieurs commandes.
  - `Utilisateur` (1) -- (0,N) `AvisProduit` : Un utilisateur peut écrire zéro ou plusieurs avis.
  - `Categorie` (1) -- (0,N) `Produit` : Une catégorie peut contenir zéro ou plusieurs produits.
  - `Categorie` (0,N) -- (0,1) `Categorie` : Relation réflexive pour les sous-catégories (une catégorie peut avoir une catégorie parent).
  - `Produit` (1) -- (0,N) `LigneDeCommande`