RAPPORT DE PROJET : ARCHITECTURE D'UNE PLATEFORME E-COMMERCE

Module : Web Application Architectures (MACSIN4A0625)

Enseignant référent : M. Arnaud NAUWYNCK

Date : Janvier 2026

Étudiants :

- Ahmat ROUCHAD
- Chanez KHELIFA

1. Introduction et Contexte

Dans le cadre du module **Web Application Architectures**, nous avons conçu et implémenté une plateforme e-commerce complète. L'objectif de ce projet n'était pas seulement de livrer une application fonctionnelle, mais de mettre en œuvre une **architecture logicielle robuste, modulaire et maintenable**, répondant aux standards industriels actuels.

Notre approche a été guidée par la volonté de découpler les responsabilités (Backend/Frontend), d'assurer la qualité du code via une pipeline CI/CD rigoureuse, et d'intégrer des concepts de modélisation avancés inspirés du **Domain Driven Design (DDD)**.

2. Choix Architecturaux et Stack Technologique

Pour répondre aux exigences de scalabilité et de maintenabilité, nous avons opté pour une architecture **Headless** séparant la logique métier (API) de l'interface utilisateur.

2.1 Backend : Robustesse et Sécurité (Django 6.0)

Nous avons choisi l'écosystème Python avec **Django 6.0** et **Django REST Framework (DRF)**.

- **Architecture Modulaire :** Le backend est structuré en applications distinctes (users, shop, orders, payments, core). Cette approche "Modular Monolith" permet une séparation claire des contextes métier (Bounded Contexts).
- **Base de données :** Utilisation de **PostgreSQL 16** pour garantir l'intégrité des données relationnelles et la conformité ACID, essentielle pour les transactions financières (commandes/paiements).
- **Sécurité :** Implémentation de JWT pour l'authentification stateless et intégration sécurisée de **Stripe** via Webhooks pour la gestion des paiements.

2.2 Frontend : Performance et Expérience Utilisateur (React 18)

L'interface client est une Single Page Application (SPA) développée en **React 18** avec **TypeScript**.

- **Type Safety :** L'utilisation stricte de TypeScript assure une robustesse du code et réduit les erreurs au runtime.
- **State Management :** Nous avons privilégié **Zustand** (plus léger et moderne que Redux) pour gérer l'état global de l'application de manière fluide.
- **Build Tooling :** Utilisation de **Vite** pour un Hot Module Replacement (HMR) instantané et des builds de production optimisés.

2.3 DevOps et Conteneurisation

Pour garantir l'isomorphisme entre les environnements de développement et de production, l'ensemble de la stack est conteneurisé via **Docker** et orchestré avec **Docker Compose**.

- Services isolés : backend, frontend, db (PostgreSQL).
- Volumes persistants pour la base de données.

3. Méthodologie et Qualité Logicielle

3.1 Domain Driven Design (DDD) & Ubiquitous Language

Comme illustré dans nos phases de conception, nous avons tenté d'appliquer les principes du DDD :

- **Ubiquitous Language :** Définition d'un lexique commun (anglais) strict entre le code et le métier pour éviter toute ambiguïté (ex: Order, Customer, ProductCatalog).
- **Modélisation :** Les schémas de base de données reflètent fidèlement les relations métier, avec une attention particulière portée aux agrégats dans les modules orders et shop.

3.2 Workflow Git et Intégration Continue (CI/CD)

Nous avons mis en place une stratégie de gestion de versions stricte pour faciliter la collaboration :

- **Git Flow :** Utilisation de branches par fonctionnalité (feat/, fix/, refactor/) avec protection de la branche main.
- **Configuration Git :** Application de pull.rebase = true pour maintenir un historique linéaire et propre.

Pipeline GitHub Actions : L'automatisation est au cœur de notre processus de qualité. À chaque Push ou Pull Request, une pipeline CI se déclenche :

1. **Linting & Formatage :** Vérification du style de code via Black, Flake8 et isort (Python) ainsi que ESLint(JS/TS).
2. **Tests Automatisés :** Exécution de la suite de tests unitaires et d'intégration via **pytest**.
3. **Reporting :** Génération de rapports de couverture de code (pytest-cov).

Seul un code passant intégralement ces étapes peut être fusionné, garantissant une "Dette Technique" minimale.

4. Fonctionnalités Clés Implémentées

- **Gestion Utilisateurs :** Inscription, Login, Profils étendus.
- **Catalogue Produits :** Liste, détails, recherche et filtrage performant côté serveur.
- **Tunnel de Commande :** Panier (géré par le state manager), validation de commande.
- **Paiement :** Intégration complète de l'API Stripe (mode test) avec gestion des succès/échecs.
- **Documentation API :** Génération automatique de la documentation via Swagger/OpenAPI, facilitant la consommation de l'API par le frontend.

5. Conclusion et Perspectives

Ce projet nous a permis de consolider nos compétences en architecture web moderne. Nous avons réussi à livrer une application fonctionnelle tout en respectant des contraintes techniques élevées (Dockerisation, CI/CD, Typage fort).

Les défis principaux ont résidé dans la configuration fine de l'environnement Docker pour le hot-reload et la synchronisation des modèles de données entre le backend Django et le frontend TypeScript.

Pour aller plus loin, nous pourrions envisager :

- L'ajout d'un cache Redis pour optimiser les requêtes du catalogue.
- Le déploiement de l'infrastructure sur un fournisseur Cloud (AWS ou Azure) via Terraform.