Charte de Développement et contraintes Technologiques

Dans ce document nous présentons la charte de développement (code et base de données) et les contraintes/exigences technologiques.

- Autor: Thomas Djotio Ndié, Prof Dr_Eng.
- thomas.djotio@yowyob.com/+237 675 518 880
- Github id: tdjotio@gmail.com

- Doc date and version: 30.09.25, V0.1

1. Convention de nommage

- Variables : en anglais, style snake_case (ex. user_name, order_date).
- Constantes : en majuscules avec séparateur _ (ex. MAX_SIZE, DEFAULT_TIMEOUT).
- Classes : en anglais, style PascalCase (ex. CustomerService, OrderManager).
- Méthodes : en anglais, style camelCase (ex. calculateTotal(), getUserName()).
- Packages : en minuscules, sans accents, séparés par des points (ex. com.yowyob.flashshop.service).
- Interfaces : pas de préfixe I, nom descriptif en PascalCase (ex. PaymentProcessor).

2. Organisation du code

- Une classe par fichier, fichier nommé comme la classe publique principale.
- Longueur de ligne maximale : 120 caractères.
- Indentation : 4 espaces (pas de tabulations).
- Accolades {} toujours ouvertes à la fin de la ligne de déclaration (ex. if (condition) {).
- Importations : utiliser uniquement les classes nécessaires (pas de import java.util.*;).

3. Documentation et commentaires

- Chaque classe doit contenir un header Javadoc avec :
- Description concise
- Auteur
- Date de création / mise à jour
- Les méthodes publiques doivent avoir une Javadoc précisant paramètres, retour, exceptions.
- Commentaires en anglais, clairs et concis.
- Éviter les commentaires superflus (préférer un code auto-explicatif).

4. Gestion des exceptions

- Toujours utiliser des exceptions spécifiques plutôt que Exception générique.
- Propager les exceptions uniquement si pertinent, sinon les gérer localement.
- Les messages d'erreur doivent être explicites et en anglais.

5. Bonnes pratiques

- Respecter le principe SOLID et les design patterns adaptés.
- Séparer clairement couches Controller / Service / Repository.
- Pas de valeurs "magiques" dans le code → utiliser des constantes.
- Préférer les types primitifs aux objets enveloppe sauf nécessité (ex. int plutôt que Integer).
- Utiliser Optional pour gérer l'absence de valeur au lieu de null.
- Respecter les règles de sécurité : ne pas loguer de données sensibles (mots de passe, tokens, etc.).

6. Tests et qualité

- Chaque nouvelle fonctionnalité doit être accompagnée de tests unitaires (JUnit 5).
- Respecter un coverage minimum de 80%.
- Nommage des méthodes de test explicite (ex. shouldReturnErrorWhenInputIsInvalid).
- Utiliser des outils de vérification automatique (Checkstyle, SonarLint, PMD).

7. Contrôle de version (Git)

- Une fonctionnalité = une branche.
- Convention de nommage des branches :
- feature/nom_fonctionnalite
- bugfix/description_bug
- hotfix/description_rapide
- Commits en anglais, message clair et concis (impératif) :
- Ex. Add payment gateway integration
- Ex. Fix order calculation bug

2. Objectifs pédagogiques

- Concevoir et implémenter un système distribué basé sur les microservices.
- Intégrer des **meilleures pratiques DevOps** (CI/CD, tests, monitoring).

- Mettre en œuvre des **design patterns** et principes de génie logiciel.
- Développer des compétences sur un **stack technologique moderne** :
    - Backend : **Java/Spring Boot (WebFlux, Liquibase, Kafka)**
    - Bases de données : **PostgreSQL/PostGIS, Redis**
    - Recherche : **Elasticsearch**
    - Frontend : **Next.js (SSR/SEO)**
    - Carte interactive: **OpenStreetMap, Maptiler, Maplibre, Leafflet**
    - Outils de déploiement : **Docker, GitHub Actions**
    - Application **PWA**
    - Application à **faible latence (low latence apps; e.g local-first enable app avec DuckDB, DuckDB Wasm)**
- Développer les compétences de développeurs d'applications conscientes des **réseaux résilients**, c'est à dire capables d'Utilisation locale des données par les applications web et mobiles client: DuckDB et **DuckDB-Wasm** avec la couplage **DuckDB–Postgresql**