Révision de Java, Docker et Git

I311 — Bot Discord

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-01

Table des matières

1 Introduction 1
2 Objectifs pédagogiques 1
3 Objectifs Techniques 2
4 Choix du framework 3
5 Modélisation des données pour Discord 3
6 Dockerisation de l'application et de la base de données 3
7 Développement de l'API REST avec JAX-RS 3
8 Intégration Discord (optionnelle / avancée) 4
9 Extensions et travaux optionnels 4
9.1 Grille d'évaluation Indicative 4

1 Introduction

Ce document présente un sujet de TP fil-rouge et projet de semestre pour réviser et approfondir les notions essentielles de Java modernes (POO avancée, persistance, tests), les bonnes pratiques d'ingénierie logicielle (Git, CI, qualité) ainsi que la containerisation avec Docker. Le thème concret est la conception d'un bot et d'une API de gestion d'utilisateurs et de groupes inspirée d'un serveur Discord : ce contexte fournit des cas d'usage réalistes (événements, permissions, synchronisation d'état) tout en restant accessible.

Ce projet **ne s'arrête pas à cette révision** : il constituera le socle évolutif sur lequel nous viendrons greffer les nouvelles notions de microservices et d'architecture distribuée au fil des cours de M2.

2 Objectifs pédagogiques

- Consolidation des fondamentaux Java (conception orientée objet, interfaces, génériques, exceptions, concurrence basique).

- Mise en pratique de la persistance relationnelle avec JPA: mapping, migrations, relations (OneToOne, OneToMany, ManyToMany), transactions.
- Conception d'une API REST ergonomique et testable (contrats, DTOs, validation, gestion d'erreurs).
- Tests : rédaction de tests unitaires, tests d'intégration.
- Automatisation : build Maven reproductible, intégration continue, packaging et containerisation (Dockerfile, Docker Compose).
- Qualité : documentation Javadoc minimale, utilisation d'un linter/sonarlint et logs structurés. Intégration avec SonarQube si possible.
- Architecture logicielle : séparation des couches, principes SOLID, utilisation de patterns simples.

3 Objectifs Techniques

L'objectif de ce TP est de réviser les concepts de base du langage Java que vous devez maîtriser en partant du projet starter : https://github.com/ebpro/starter-quarkus-rest. Il permettra aussi une révision de la mise en pratique de Git et de Maven.

Travail attendu

- Lire attentivement le sujet et initialiser un dépôt GitHub (fork via le lien fourni) : créez une roadmap (milestones) et des tickets (issues) correspondant aux questions et fonctionnalités.
- Documenter au fur et à mesure : Javadoc pour les classes publiques, README avec instructions d'installation et d'exécution.
- Tests obligatoires : couvrir la logique métier avec JUnit + Mockito, et fournir au moins un test d'intégration qui démarre l'application (profil test) avec une base H2 (ou PostgreSQL via testcontainers).
- Build et packaging : utiliser Maven pour compiler, exécuter les tests, générer la Javadoc et produire un artefact exécutable (jar). Fournir un Dockerfile multi-étages pour construire l'image.
- CI locale (suggestion) : un script `ci.sh` ou un `docker-compose` pour lancer les vérifications (build, tests, lint).
- CI distante : configurer GitHub Actions pour automatiser le build et les tests à chaque push.
- Logging : utiliser SLF4J + Logback (ou équivalent) et configurer des niveaux de logs.

4 Choix du framework

Le sujet propose Quarkus pour sa simplicité d'intégration cloud-native (dev mode rapide, build natif optionnel). Si vous préférez Spring Boot, c'est acceptable mais vous devez l'indiquer dans le README et garder les mêmes contrats API et tests.

5 Modélisation des données pour Discord

Le thème du projet est l'intégration avec Discord, une plateforme de communication populaire. Vous allez modéliser les entités principales de Discord, telles que les utilisateurs, les groupes, les rôles et les canaux.

Premières étapes recommandées

1. Concevez un modèle de domaine minimal pour commencer : `User`, `Guild` (serveur), `Role`, `Channel`. Pour chacun, identifiez un petit jeu d'attributs (ex : `User(id, username, displayName, joinedAt)`).
2. Rédigez un diagramme simple (UML) montrant les cardinalités, relations, rôles et contraintes.
3. Implémentez les entités Java et les mappings JPA. Favorisez des identifiants UUID et un schéma simple permettant des inserts/updates faciles.
4. À terme vous fournirez des scripts de migration (Flyway ou Liquibase). **Au démarrage laissez Hibernate générer le schéma pour la phase d'implémentation initiale.**

6 Dockerisation de l'application et de la base de données

Livrables liés à la dockerisation

- Un `Dockerfile` multi-étages pour l'application Java.
- Un `docker-compose.yml` fournissant au moins : l'application, une base PostgreSQL (ou H2 pour tests) et, optionnellement, un service Redis (ou une autre base NoSQL) si utilisé.
- Des instructions claires dans le README pour construire et démarrer l'environnement : `mvn package && docker build -t mon-bot .` et `docker-compose up --build`.

7 Développement de l'API REST avec JAX-RS

Contrats API (exigences minimales)

- Endpoints CRUD pour `User`, `Guild`, `Role`, `Channel` suivant des conventions REST (ex : `GET /api/users`, `POST /api/users`).
- Paginations et filtres simples pour les listes.
- Validation des payloads et code d'erreurs standardisés (400, 404, 500).
- Utiliser DTOs séparés des entités si besoin.

8 Intégration Discord (optionnelle / avancée)

En vous appuyant sur la puissante bibliothèque Discord4J https://discord4j.com/, vous allez créer un bot Discord personnalisé capable d'interagir de manière dynamique avec votre serveur.

- Fournir une abstraction (Service / Adapter) qui encapsule les appels à Discord4J pour faciliter les tests (mockable).
- Commencer par une commande simple `/sayhello` ou un handler d'événement `MemberJoin` qui enregistre l'utilisateur en base (mode offline possible avec des tokens de test).
- Synchronisation : concevoir un mécanisme idempotent pour appliquer des changements de rôles depuis la DB vers Discord (et réciproquement si nécessaire).
- Consignes de sécurité : ne jamais committer de tokens ; utiliser des variables d'environnement et un fichier `.env` exemple.

9 Extensions et travaux optionnels

Vous pourrez, si le temps le permet, proposer des modules additionnels (plugins) et des intégrations LLM.

Exemples d'extensions :

1. Agents déclenchés par commandes

- `/teach` – explique un concept technique demandé.
- `/translate` – traduit un texte.
- `/summarize` – résume un thread ou un texte donné.
- `/qa` – recherche une réponse dans un index local (RAG) et la renvoie.

2. Agents intégrés au channel

- Auto-summarize – résumé périodique d'un channel.
- Auto-translate – traduction automatique inter-channel.
- Auto-moderation – détection simple de contenus inappropriés.
- Standup bot – relances et résumé des réponses quotidiennes.

9.1 Grille d'évaluation Indicative

- Fonctionnalités de base et qualité du code : 45%
- Tests et documentation : 20%
- Dockerisation et reproducibility : 10%
- Architecture et séparation des responsabilités : 15%
- Extensions (LLM, plugins) et créativité : 10% (bonus possible)