**🧱 Documentation d’Architecture - Application Microservices (Spring Boot + NestJS)**

**📌 Objectif**

Cette documentation décrit l’architecture globale d’une application basée sur une approche microservices. L’ensemble repose majoritairement sur **NestJS**, avec un **seul service Spring Boot** (facturation). L’architecture suit les principes de modularité, séparation des responsabilités, et communication asynchrone via RabbitMQ.

**🗺️ Vue d’ensemble des services**

|  |  |  |
| --- | --- | --- |
| **Service** | **Technologie** | **Description** |
| api-gateway | NestJS | Point d’entrée principal de l’application |
| auth-service | NestJS | Authentification, gestion des utilisateurs |
| affaire-service | NestJS | Gestion des affaires client |
| agency-service | NestJS | Gestion des agences et entités |
| contrat-service | NestJS | Gestion des contrats |
| facture-service | **Spring Boot** | Génération et gestion des factures |
| operation-service | NestJS | Suivi des opérations et interventions |
| workflow-service | NestJS | Orchestration des processus |
| mailer-service | NestJS | Envoi des notifications par e-mail |

Tous les services communiquent via **RabbitMQ**, en suivant un schéma de messagerie basé sur les événements.

**📂 Structure du projet**

project-root/

├── services/

│ ├── affaire-service/ # NestJS microservice

│ ├── agency-service/ # NestJS microservice

│ ├── api-gateway/ # NestJS gateway

│ ├── auth-service/ # NestJS microservice

│ ├── contrat-service/ # NestJS microservice

│ ├── facture-service/ # Spring Boot microservice

│ ├── mailer-service/ # NestJS microservice

│ ├── operation-service/ # NestJS microservice

│ └── workflow-service/ # NestJS microservice

│

├── Database/

│ ├── config/

│ ├── models/ # Sequelize models (TS + JS)

│ ├── repositories/ # Sequelize repositories

│ ├── migrations/

│ ├── seeders/

│ └── sequelize.ts # Sequelize instance

│

├── application/ # Services transversaux Nest (auth)

│ ├── dtos/

│ ├── mappers/

│ ├── services/

│ ├── usecases/

│ └── validators/

│

├── domain/

│ ├── entité/

│ └── errors/

│

├── infrastructure/ # Interfaces techniques et connecteurs externes

│ ├── controllers/ # Contrôleurs REST exposant les usecases

│ ├── listeners/ # Gestionnaires des événements RabbitMQ

│ ├── modules/ # Assemblage des dépendances NestJS

│ ├── guards/ # Vérifications d’authentification/autorisation

│ ├── interceptors/ # Intercepteurs de requêtes/réponses (log, cache, etc.)

│ ├── filters/ # Gestion centralisée des exceptions (exception filters)

│ └── config/ # Chargement de configuration, validation env, etc.

│

├── docker-compose.yml

└── README.md

Chaque sous-dossier de infrastructure/ a une responsabilité claire :

* **controllers/** : exposent les routes HTTP, orchestrent les cas d’usage (sans contenir de logique métier).
* **listeners/** : abonnés RabbitMQ avec @EventPattern, appellent les cas d’usage en réaction à un message.
* **modules/** : déclarent les dépendances nécessaires pour chaque feature (services, usecases, repo, etc).
* **guards/** : protègent les routes selon rôles ou présence de JWT valide.
* **interceptors/** : exécutent du code autour des requêtes, comme logs ou mise en cache.
* **filters/** : interceptent et personnalisent la gestion des exceptions.
* **config/** : centralise les règles de validation des variables d’environnement, config nest, etc.

**🔄 Communication interservices**

**RabbitMQ (événements asynchrones)**

Tous les services publient ou écoutent des messages RabbitMQ avec un format JSON commun.

* Utilisation de @MessagePattern() dans NestJS
* Spring écoute via @RabbitListener

**HTTP REST (cas ponctuels)**

Certains services (ex. api-gateway) appellent en REST les endpoints internes (auth, users).

**🔐 Authentification & Sécurité**

* Centralisée dans auth-service (NestJS + JWT + Passport)
* Les autres services utilisent des **Guards** pour vérifier les tokens

@Injectable()

export class JwtAuthGuard implements CanActivate {

canActivate(context: ExecutionContext): boolean {

const req = context.switchToHttp().getRequest();

const token = req.headers.authorization?.split(' ')[1];

return validateJwt(token); // Fonction à implémenter

}

}

**⚙️ Technologies utilisées**

|  |  |
| --- | --- |
| Domaine | Stack technique |
| Backend principal | NestJS + TypeScript |
| Facturation uniquement | Spring Boot 3, Java 17+, JPA |
| ORM NestJS | Sequelize (PostgreSQL) |
| Messagerie | RabbitMQ (asynchrone) |
| Authentification | JWT, Passport.js (NestJS) |
| Base de données | PostgreSQL, SQLite pour test, H2 pour Spring |
| Conteneurisation | Docker, docker-compose |

**📡 Exemple de flux : création de contrat → facture**

1. contrat-service crée un contrat client → émet contract\_created
2. facture-service écoute l’événement via RabbitMQ → crée une première facture
3. mailer-service reçoit invoice\_created et envoie un e-mail

**🔧 Déploiement**

* Docker compose local (RabbitMQ, PostgreSQL, microservices)
* Services exposés via api-gateway (NestJS)
* Liveness/Readiness probes sur /health

**📎 Notes spécifiques**

* Le dossier Database/ contient les modèles Sequelize pour tous les services Nest.
* Chaque microservice Nest suit Clean Architecture :
  + application/ : DTOs, services métier, use cases
  + domain/ : entités métier pures, erreurs
  + infrastructure/ : connecteurs HTTP et RabbitMQ, configuration, sécurité, erreurs
* Le facture-service est **le seul** service Spring Boot, utilisé pour des raisons métier spécifiques (interop comptable Java).

**👤 Auteurs**

Yassine BOULAHNINE — Développeur Full Stack

Architecture pensée pour une application scalable, modulaire, maintenable, avec des responsabilités clairement séparées entre services.