Exercice Technique – Application de Gestion de Notes Collaboratives

Objectif

Concevoir une application de **gestion de notes** multi-plateforme avec :

- Authentification JWT
- CRUD, recherche/filtrage, tags
- **Partage** d'une note (lecture seule par lien ou avec un autre utilisateur)
- Front web moderne
- **App mobile** offline-first avec synchronisation
- Dockerisation et documentation de bout en bout
  (sur la base de l'exercice précédent, étendu aux 3 couches)

Livrables attendus (monorepo recommandé)

```
/notes-suite/
  backend-spring/
  web-frontend/        # React OU Angular (option BFF NestJS)
  mobile-app/          # Flutter OU React Native
  docker/
    docker-compose.yml
  README.md
```

- **README racine** : prérequis, architecture, commande unique "up".
- **README par partie** : setup, scripts, tests, comptes de démo.
- **Docker Compose** opérationnel (API + DB + web). La mobile app n'est pas conteneurisée mais documentée (émulateur/build).
- **Qualité** : validation inputs, gestion d'erreurs, logs structurés, tests unitaires/minimaux.

Modèle de données (commun)

- **User**(id, email, password_hash, created_at)
- **Note**(id, owner_id, title, content_md, visibility: `PRIVATE|SHARED|PUBLIC`, created_at, updated_at)
- **Tag**(id, label), **NoteTag**(note_id, tag_id)
- **Share**(id, note_id, shared_with_user_id, permission: `READ`)
- **PublicLink**(id, note_id, url_token, expires_at? — optionnel)

Remarque : l'édition collaborative en temps réel est **bonus** ; le partage de base reste **lecture seule**, comme l'énoncé initial.

Partie A — Backend Spring Boot

Stack : Spring Boot 3, Spring Security (JWT), Spring Data JPA, PostgreSQL, OpenAPI/Swagger, MapStruct (option), Testcontainers (option).

Exigences

- **Auth** : inscription, login, refresh token, middleware JWT.
- **Endpoints** (prefix `/api/v1`):
  - `POST /auth/register`, `POST /auth/login`, `POST /auth/refresh`
  - `GET /notes?query=&tag=&visibility=&page=&size=` (pagination tri par `updated_at`)
  - `POST /notes` (title, content_md, tags[]), `GET /notes/{id}`, `PUT /notes/{id}`, `DELETE /notes/{id}`
  - `POST /notes/{id}/share/user` (email du destinataire → permission READ)
  - `POST /notes/{id}/share/public` (génère un `url_token`)
  - `DELETE /shares/{shareId}`, `DELETE /public-links/{id}`
  - **Public** : `GET /p/{url_token}` (retourne la note si publique/valide)
- **Filtres & recherche** : par `title` (LIKE, insensible à la casse) et par `tag`.
- **Validation** : Bean Validation (titre min 3, content max 50k chars, etc.)
- **Sécurité** : ownership sur notes, 403 si accès interdit.
- **Docs** : OpenAPI via springdoc.
- **Tests** : unités services + 1 test d'API heureux (MockMvc).

Partie B — Front web React ou Angular

(**Option avancée** : intercaler un **BFF NestJS** entre le front et l'API pour gérer SSR, caches, composition d'API. Si choisi, documenter la valeur ajoutée.)

Exigences UI/UX

- **Auth** : écrans Login / Register (JWT stocké de façon sécurisée).
- **Liste des notes** avec :
  - Recherche (titre), filtres par visibilité & tags, pagination.
  - Onglets ou chips de visibilité (Privé / Partagé / Public).
- **Édition** : création/édition de note avec **éditeur Markdown** (prévisualisation).
- **Partage** :
  - Par utilisateur (saisie email → appel backend).

    - Lien public (génération/ révocation + copie du lien).
- **Détails** : page publique lisible via `/p/{url_token}`.
- **État & routing** :
    - React : React Router + état global (Context/Zustand/Redux au choix).
    - Angular : Router + service state + HttpClient Interceptor (JWT).
- **Qualité** : toast notifications, gestion erreurs API, skeleton loaders.
- **Tests** : au moins 2 tests composants + 1 test e2e "heureux".

Partie C — App Mobile Flutter ou React Native

Exigences

- **Auth** (écran Login/Register).
- **Liste & détail** des notes (lecture, recherche, filtres).
- **Création/Édition** de note (éditeur texte simple + préview Markdown basique).
- **Partage** : visualiser les métadonnées de partage (sans gestion utilisateur côté mobile, OK pour V1).
- **Offline-first** :
    - Cache local (SQLite/Room/Hive) + file d'**ops en attente**.
    - Stratégie : lecture depuis cache, sync en arrière-plan à la reconnexion.
    - Conflits : **Last-Write-Wins** (documenté).
- **UX** : pull-to-refresh, empty states, indicateurs hors-ligne.
- **Build** : scripts pour lancer l'app (Android émulateur ou device).

Docker & Démarrage

- **Services** : `api` (Spring), `db` (Postgres), `web` (serveur statique Nginx pour build du front).
- **Réseau** : `api` exposée sur `http://localhost:8080`, `web` sur `http://localhost:3000` (React/Angular dev) ou `http://localhost:8081` (Nginx).
- **Scripts** :
    - `docker compose up -d` (DB + API)
    - `web-frontend`: `npm i && npm run dev` (ou `build && docker up web`)
    - `mobile-app`: instructions émulateur.

Sécurité & bonnes pratiques

- **JWT** en `Authorization: Bearer`.
- Cookies **non** requis (SPA).
- Validation backend stricte, messages d'erreurs normalisés `{code, message, details}`.
- CORS : restreindre aux origines locales attendues.
- Migrations DB : Flyway/Liquibase (option).
- Logs JSON (option), corrélation `X-Request-Id`.

🗀 Structure de rendu demandée

L'exercice a été estimé pour 4j par notre CTO

Un repo sous github composé

- Codes

- README.md complet avec instructions d'installation & de test

- Dockerisation (docker compose opérationnel)

- Bonnes pratiques : structure de projet claire, gestion d'erreurs, validation des inputs