Guide Technique NaotY – Youth Computing

1. Qualité de Code & Principes Généraux

Le projet NaotY, destiné à la gestion des notes internes de l'association Youth Computing, doit être réalisé avec un haut niveau d'exigence technique et de clarté. Le code doit être :

- Lisible, maintenable et bien structuré
- Conforme aux principes SOLID (Frontend) et Clean Architecture (Backend)
- Accompagné de commentaires pertinents (en anglais)
- Testable et testé (tests unitaires/requêtes critiques)
- Respectueux de la charte graphique

2. Frontend (React + TypeScript + TailwindCSS)

Architecture

- Utiliser une architecture modulaire avec des dossiers : `components/`, `pages/`, `hooks/`, `services/`, `utils/`, `interfaces/`
- Appliquer **les principes SOLID** pour la construction des composants :
  - Single Responsibility: chaque composant a un seul rôle clair
  - Open/Closed: composants étendus via props, non modifiés directement

Bonnes pratiques

- Typage strict avec TypeScript
- Eviter les composants anonymes ou imbriqués complexes
- Gestion des états via `useState`, `useReducer` ou `Zustand`
- Communication API via `axios` avec interception (token JWT)
- Responsivité obligatoire (mobile first)

UI / UX

- Respecter la **charte graphique Youth Computing** :

- Couleurs : `#010b40` (bleu), `#f13544` (fuchsia), gris moyen (#999999)
- Police : `Ubuntu` en principale, `Century Gothic` en secondaire

- Utilisation cohérente des composants Tailwind avec classes utilitaires centralisées
- Privilégier l'accessibilité (aria-labels, navigation clavier, contraste suffisant)

3. Backend (FastAPI + PostgreSQL)

Architecture : Clean Architecture

Organisation des dossiers :

```
src/
├── domain/         # Entités métier et interfaces
├── application/    # Cas d'usage (services, logique)
├── infrastructure/# Accès base de données, API externes
├── interfaces/     # Contrôleurs FastAPI (routes, DTOs)
```

Bonnes pratiques

- Suivre le principe d'inversion de dépendance
- Aucun accès à la DB ou aux requêtes externes dans les entités
- Utiliser des DTOs pour la validation des entrées
- Gestion centralisée des exceptions
- Authentification JWT à chaque route protégée

Base de données

- Utiliser Alembic pour les migrations
- Respecter la normalisation des tables
- Utiliser des UUID pour les identifiants principaux

4. Git & GitHub : Convention d'utilisation

Branches

- `main` : branche de production (stable, testée)
- `dev` : intègre toutes les fonctionnalités validées (recette)
- `feature/nom-fonctionnalite` : une branche par fonctionnalité

- `hotfix/description` : pour les corrections critiques post-prod

Nommage de commit (anglais, précis)

Format : `type(scope): message`

Exemples :

- `feat(note): add schedule planner with CRON support`
- `fix(auth): resolve token expiration issue`
- `refactor(api): split routes and services`
- `chore(ci): update GitHub action for build`

PR & Revue de code

- Chaque feature doit passer par une Pull Request vers `dev`
- Description claire, checklist de tests
- Relecture obligatoire par un autre membre avant merge

5. Tests & CI

- Ajouter des tests unitaires pour les cas critiques
- Intégrer GitHub Actions pour le linting, les tests et le build
- Possibilité de déployer sur environnement de préproduction après chaque merge sur `dev`

6. Environnement de développement

- Frontend : Vite, ESLint, Prettier configurés
- Backend : FastAPI, dotenv, black pour le formatage
- Docker pour le déploiement local des services (DB, API, UI)