Choix technologiques : React pour le web, Python pour le back-end, React Native pour le mobile

Pour ce projet, nous avons fait le choix d'une architecture moderne articulée autour de trois technologies principales :

- **React** pour l'interface web,
- **Python** pour la partie back-end,
- **React Native** pour les applications mobiles.

Ce trio forme un ensemble cohérent, largement utilisé dans l'industrie, qui permet de concilier **productivité de développement**, **performance**, **maintenabilité** et **évolutivité**.

1. React pour le front-end web

1.1. Un standard de fait pour les interfaces modernes

React n'est pas un framework complet mais une **bibliothèque JavaScript dédiée à la construction d'interfaces utilisateur**, basée sur des composants réutilisables. (React)

Son approche déclarative (« on décrit l'UI en fonction de l'état ») rend le code plus **prévisible**, plus **facile à tester** et plus **simple à maintenir**, ce qui est un point clé pour un projet qui va évoluer et s'enrichir de nouvelles fonctionnalités. (React)

React est aujourd'hui l'une des bibliothèques front-end les plus utilisées au monde, portée par Meta et par une très large communauté open-source. (Wikipedia) Cela garantit un écosystème riche (composants UI, outils de test, frameworks comme Next.js, etc.) et une grande facilité à trouver de la documentation, des exemples ou du support.

1.2. Comparaison avec Angular et AngularJS

Angular (la version moderne en TypeScript) est un **framework complet** côté front : il fournit un ensemble très structuré de concepts (modules, services, injection de dépendances, etc.). (Angular)

Dans l'absolu, Angular est très puissant, mais pour un projet étudiant ou une équipe réduite, il présente plusieurs inconvénients :

- **Courbe d'apprentissage plus raide** : Angular impose une architecture et une quantité de concepts plus importante que React (modules, decorators, DI, pipes, etc.).

- **Boîte très “opinionated”** : on gagne en structure, mais on perd en flexibilité.

- **Overkill** pour certaines applications : sur des projets où l’on veut itérer vite et tester plusieurs idées, la légèreté de React est souvent un avantage.

Concernant **AngularJS** (Angular 1.x), c’est aujourd’hui une technologie **obsolète** :

- Google a officiellement arrêté le support et la sécurité d’AngularJS au 31 décembre 2021, ce qui en fait un framework en fin de vie.

- Continuer à développer un nouveau projet sur AngularJS poserait des problèmes de maintenance, de sécurité et de recrutement de développeurs.

Face à cela, **React** présente plusieurs atouts pour notre contexte :

- Une **API plus simple à appréhender** pour des développeurs qui connaissent déjà JavaScript.

- Une architecture par **composants réutilisables**, très adaptée aux interfaces riches (dashboards, formulaires dynamiques, etc.)

- La possibilité de le combiner librement avec d’autres bibliothèques (React Router, Zustand/Redux, Tailwind, etc.), sans être enfermé dans un écosystème trop rigide.

1.3. Comparaison avec d’autres bibliothèques (Vue, Svelte, etc.)

D’autres solutions modernes existent, comme **Vue.js** ou **Svelte**, qui sont également très intéressantes. Cependant :

- React reste **le plus répandu** dans l’industrie, ce qui facilite l’**embauche**, l’**onboarding de nouveaux développeurs** et la réutilisation future des compétences dans d’autres projets.
- L’existence de **React Native** (pour le mobile) est un argument fort : choisir React pour le web permet de mutualiser une grande partie des concepts et parfois une portion de la logique UI entre web et mobile.

2. Python pour le back-end

2.1. Un langage lisible, polyvalent et “batteries included”

Python est un **langage de haut niveau, interprété et généraliste**, dont la philosophie met l’accent sur la lisibilité du code et la simplicité.

Quelques points clés :

- La syntaxe de Python est souvent décrite comme du **pseudo-code exécutable**, ce qui améliore énormément la **compréhension du code en équipe**.

- Python adopte une approche **"batteries included"** : la bibliothèque standard est très riche (fichiers, réseau, JSON, compression, etc.), ce qui permet de réaliser beaucoup de choses sans dépendre de dizaines de packages externes.

- Il est largement utilisé pour le **web**, la **data science**, le **machine learning** et l'**automatisation**, ce qui ouvre la porte à des fonctionnalités avancées par la suite (recommandations, scoring, IA, etc.).

Dans notre projet, Python permet de développer une API back-end claire (par exemple avec **FastAPI** ou **Django REST Framework**), tout en gardant la possibilité d'intégrer plus tard des modules d'analyse de données ou de machine learning, sans changer de langage.

2.2. Comparaison avec Node.js (JavaScript côté serveur)

L'alternative naturelle à Python est souvent **Node.js**, puisque cela permet d'utiliser JavaScript **à la fois côté front et côté back**.

Node.js est très performant pour des applications fortement orientées **I/O** (beaucoup de connexions simultanées, websockets, etc.), mais :

- Le code Node/JavaScript peut rapidement devenir **verbeux et complexe** à lire, notamment avec la gestion intensive des promesses et de l'asynchrone.

- Node.js ne bénéficie pas du même écosystème "naturel" pour la **data science** et le **machine learning** que Python, qui reste aujourd'hui la référence dans ces domaines.

Dans notre contexte, où le back-end doit encapsuler de la **logique métier**, des **règles de gestion** et potentiellement des traitements de données plus avancés, Python offre un très bon compromis entre :

- **Productivité** (développement rapide, peu de code pour faire beaucoup de choses),

- **Lisibilité** (onboarding plus facile des nouveaux membres),

- **Richesse de l'écosystème** (frameworks web, bibliothèques scientifiques, IA, etc.).

2.3. Comparaison avec Java / Spring, PHP / Laravel, etc.

D'autres langages back-end sont très solides, en particulier :

- **Java / Spring** : excellent pour des architectures d'entreprise très grandes, avec des contraintes fortes de typage statique et d'outillage, mais la verbosité de Java allonge le temps de développement pour un projet étudiant ou en petite équipe.

- **PHP / Laravel** : très utilisé pour les applications web classiques, mais moins naturel dès que l'on souhaite intégrer des briques de data science ou de machine learning.

En résumé, **Python** combine :

- une **courbe d'apprentissage douce**,

- une excellente **rapidité de développement**,

- la possibilité de faire évoluer le projet vers des fonctionnalités plus avancées sans changer de stack.

3. React Native pour le mobile

3.1. Mutualisation des compétences et de la logique UI

React Native est une technologie mobile qui **s'appuie sur React** pour construire des applications Android et iOS à partir d'un code JavaScript/TypeScript partagé.

Concrètement :

- On retrouve les mêmes concepts que dans React web : composants, props, état, hooks…

- Une partie de la logique (gestion de l'état, appels à l'API, règles métier) peut être **factorisée** entre le web (React) et le mobile (React Native).

- On obtient des applications **proches du natif** en termes de performance, tout en gardant une base de code unique pour les deux plateformes.

Ce choix est particulièrement pertinent dans un contexte de projet où **la même équipe** gère le web et le mobile : cela évite de devoir recruter des spécialistes distincts pour Kotlin/Java (Android) et Swift (iOS).

3.2. Comparaison avec Flutter, Ionic et le natif

Quelques alternatives :

- **Flutter** (Dart) : très bon framework cross-platform, mais il impose l'apprentissage d'un **nouveau langage (Dart)** et d'un nouvel écosystème. Pour une équipe qui utilise déjà massivement JavaScript/TypeScript côté front, cela représente un coût d'entrée supplémentaire.

- **Ionic / Capacitor** : basé sur des technologies web encapsulées dans une webview. C'est simple à mettre en place, mais l'intégration native et les performances peuvent être en dessous de ce que propose React Native pour des interfaces plus complexes.

- **Développement natif (Kotlin/Java pour Android et Swift pour iOS)** : c'est l'option avec les meilleures performances brutes et l'accès le plus direct aux API natives, mais au prix d'un **double développement** (une application par plateforme) et donc d'un temps de développement et de maintenance beaucoup plus élevé.

Dans notre cas, **React Native** est un compromis idéal :

- Un **seul paradigme** (React) pour le web et le mobile,

- Un **time-to-market plus court**,

- Une courbe d'apprentissage uniformisée pour toute l'équipe.

4. Cohérence globale de la stack et avantages pour l'équipe

En combinant **React**, **Python** et **React Native**, on obtient une stack :

- **Cohérente** : la même logique de composants côté front et mobile, un back-end clair et lisible.

- **Évolutive** : on peut facilement ajouter de nouvelles fonctionnalités, exposer de nouvelles routes API, développer une deuxième application mobile, etc.

- **Apprenante** : chaque membre de l'équipe peut monter en compétence sur des technologies très demandées sur le marché (React, Python, React Native).

- **Pérenne** : contrairement à AngularJS, aujourd'hui en fin de vie, toutes les briques que nous utilisons sont activement maintenues et soutenues par des communautés très dynamiques.

5. Choix des services, actions et réactions

-

🔹 Service Google (Gmail / YouTube)

- **Actions :**

1. **Gmail – Nouveau mail contenant un mot-clé**
   Déclenche l'AREA lorsqu'un mail reçu contient un mot-clé spécifique dans l'objet (ex. `[URGENT]`).

2. **YouTube – Nouvelle vidéo sur une chaîne donnée**
   Déclenche l'AREA lorsqu'une nouvelle vidéo est publiée sur une chaîne YouTube configurée.

- **Réactions :**

1. **Gmail – Envoyer un email**
   Envoie un mail à un destinataire avec sujet et corps configurables.

2. **YouTube – Ajouter une vidéo à une playlist**
   Ajoute une vidéo (par ID) dans une playlist YouTube donnée (ex : sauvegarder automatiquement des vidéos importantes).

🔹 Service GitHub

- **Actions :**
  3. **GitHub – Nouvelle issue créée sur un dépôt**
  Déclenche l'AREA dès qu'une issue est créée sur un repository configuré.

4. **GitHub – Nouvelle pull request ouverte**
   Déclenche l'AREA lorsqu'une nouvelle pull request est ouverte sur le dépôt.

- **Réactions :**
  3. **GitHub – Créer une issue**
  Crée automatiquement une issue avec un titre et un corps définis par l'AREA (ex. pour remonter une alerte).

4. **GitHub – Commenter une issue ou une pull request**
   Ajoute un commentaire automatique (notification, statut, lien, etc.) sur une issue ou une PR ciblée.

🔹 Service Microsoft (Outlook)

- **Actions :**
  5. **Outlook – Nouveau mail reçu dans un dossier spécifique**
  Déclenche l'AREA lorsqu'un mail arrive dans un dossier précis (ex. "Projets", "Support").

  6. **Outlook – Nouvel événement ajouté au calendrier**
  Déclenche l'AREA dès qu'un nouvel événement est créé dans le calendrier (réunion, rendez-vous, etc.).

- **Réactions :**
  5. **Outlook – Envoyer un mail**
  Envoie un email via le compte Outlook (réponse automatique, notification interne, etc.).

  6. **Outlook – Créer un événement de calendrier**
  Crée un événement (titre, date, durée, description) à partir des données de l'AREA.

🔹 Service interne "Timer"

- **Actions :**
  7. **Timer – Déclenchement tous les jours à une heure donnée**
  Exécute l'AREA chaque jour à une heure fixe (ex. 09h00).

  8. **Timer – Déclenchement toutes les X minutes**
  Exécute l'AREA à intervalles réguliers (ex. toutes les 15 minutes).

- **Réactions :**
  7. **Timer – Envoyer une requête HTTP (webhook)**
  Envoie une requête HTTP POST vers une URL configurée (pour notifier un système externe).

  8. **Timer – Enregistrer un log d'activité**
  Crée une entrée de log en base (ou dans un fichier) pour tracer l'exécution d'un scénario (utile pour le monitoring).

🔹 Service Trello

- **Actions :**
  9. **Trello – Nouvelle carte créée sur une liste**

Déclenche l'AREA lorsqu'une carte est ajoutée à une liste donnée (ex. "À faire").

10. **Trello – Carte déplacée vers une liste spécifique**
    Déclenche l'AREA lorsqu'une carte passe dans une liste cible (ex. "Terminé").

- **Réactions :**

9. **Trello – Créer une carte**
Ajoute une nouvelle carte avec titre, description et éventuellement des labels sur une liste donnée.

10. **Trello – Mettre à jour une carte**
    Modifie une carte existante (description, labels, échéance) pour refléter l'état d'un scénario (ex. marquer une tâche comme "En retard").

- **Synthèse**

  - 5 services complémentaires (Google, GitHub, Microsoft, Timer, Trello)
  - 14 capacités exploitables dans les AREA (7 actions + 7 réactions)
  - Couverture de cas d'usage variés : mails, vidéos, gestion de projet, calendrier, automatisation temporelle, webhooks.

6. Choix de la base de données : PostgreSQL

- **Contexte et besoin**
    - Besoin d'une base **relationnelle**, robuste et compatible SQL.
    - Données structurées (utilisateurs, services, actions/réactions, logs, tokens, etc.).
    - Nécessité de **transactions fiables**, de contraintes d'intégrité et de bonnes performances.
- **Pourquoi PostgreSQL**
    - SGBD **open source**, très mature et largement utilisé en production.
    - Excellente gestion des **transactions ACID**, des **contraintes** (foreign keys, unique, checks) et des **jointures complexes**.
    - Support de types avancés (JSON/JSONB, UUID, arrays, enum) utile pour stocker des configs d'AREA ou des payloads.
    - Bon équilibre entre **performance**, **fiabilité** et **respect des standards SQL**.
- **Comparaison avec MySQL / MariaDB**
    - MySQL/MariaDB sont aussi très populaires et performants pour des cas simples (CRUD classique, sites web).
    - PostgreSQL est généralement considéré comme plus **strict** et plus riche sur les fonctionnalités avancées (CTE, window functions, JSONB, indexation avancée…).
    - Pour un projet qui peut évoluer vers des requêtes complexes ou des fonctionnalités avancées, PostgreSQL offre plus de **flexibilité**.
- **Comparaison avec MongoDB (NoSQL)**
    - MongoDB est orienté **documents** (NoSQL) avec schéma flexible, pratique quand la structure des données change tout le temps.
    - Dans notre cas, nous avons des **relations fortes** (utilisateur ↔ services ↔ actions ↔ réactions), ce qui se modélise naturellement en SQL.
    - Un SGBD relationnel comme PostgreSQL simplifie les **jointures**, les **requêtes analytiques** et garantit mieux l'**intégrité des données**.

- **Comparaison avec SQLite**
    - SQLite est très léger et simple à déployer (fichier unique), idéal pour des **prototypes** ou des applis locales.
    - Mais il est moins adapté pour une **appli serveur multi-utilisateurs** avec concurrence d'accès et montée en charge.
    - PostgreSQL est plus adapté à un déploiement **dockerisé** / serveur avec plusieurs instances de back-end qui accèdent à la même base.

Nous avons choisi **PostgreSQL** car il offre une base solide, évolutive et conforme aux standards SQL, tout en restant open source et bien intégrée dans notre stack (backend Python + Docker).

7. Critères d'évaluation

Pour votre stack, les critères essentiels sont :

Critère	Importance
Compatibilité FastAPI	Primordial : API orientée REST / async
Support PostgreSQL	Essentiel pour relations complexes et types avancés
Async	Support de l'asynchrone pour performances web
Facilité d'utilisation	Rapidité de développement et maintenabilité
Sécurité	Prévention des injections SQL, requêtes paramétrées

Communauté / Documentation	Support et évolution sur le long terme

7.2. Comparaison des ORM Python

7.2.1 SQLAlchemy

- **Type** : ORM traditionnel / professionnel
- **Intégration FastAPI** : ✅ Excellente
- **Support PostgreSQL** : ✅ Complet, support des types avancés (JSONB, UUID)
- **Async** : ✅ Support natif via SQLAlchemy 1.4+ avec `asyncio`
- **Facilité** : Moyenne → nécessite un bon design et structuration
- **Sécurité** : ✅ Requêtes paramétrées
- **Avantages** : Très flexible, mature, compatible avec tout projet pro
- **Inconvénients** : Courbe d'apprentissage pour les débutants, un peu verbeux

7.2.2 SQLModel

- **Type** : ORM moderne, basé sur SQLAlchemy + Pydantic
- **Intégration FastAPI** : ✅ Excellente (créé par l'auteur de FastAPI)
- **Support PostgreSQL** : ✅ Complet
- **Async** : Partiellement, suit SQLAlchemy
- **Facilité** : ✅ Très simple, typé, validation automatique
- **Sécurité** : ✅ Requêtes paramétrées via SQLAlchemy
- **Avantages** : Rapidité de développement, forte cohérence avec FastAPI, validation automatique

- **Inconvénients** : Moins mature que SQLAlchemy, fonctionnalités avancées parfois limitées

7.2.3 Tortoise ORM

- **Type** : ORM simple, async, inspiré de Django ORM
- **Intégration FastAPI** : ✅ Possible, bonne pour petits projets
- **Support PostgreSQL** : ✅ Support natif
- **Async** : ✅ Natif
- **Facilité** : ✅ Simple à apprendre
- **Sécurité** : ✅ Requêtes paramétrées
- **Avantages** : Async-friendly, simple
- **Inconvénients** : Moins complet pour des besoins complexes, moins mature pour projets pro

7.2.4 Django ORM

- **Type** : ORM intégré à Django
- **Intégration FastAPI** : ❌ Non recommandée (dépend de Django)

- **Support PostgreSQL** : ✅ Excellent
- **Async** : Partiel (Django 4.x support async limité)
- **Facilité** : Moyenne → dépend du framework Django
- **Sécurité** : ✅ Bonne par défaut
- **Avantages** : Très puissant et complet
- **Inconvénients** : Non compatible avec FastAPI, lourd pour microservices

7.3. Analyse et recommandations

ORM	Avantages	Inconvénients	Convient à la stack FastAPI + PostgreSQL + React
SQLAlchemy	Mature, flexible, complet, async	Courbe d'apprentissage, verbeux	Oui, excellent choix pour projets pro
SQLModel	Simple, typé, validation automatique, natif FastAPI	Moins mature, fonctionnalités avancées limitées	✅ Très recommandé pour RapidAPI + petites équipes
Tortoise ORM	Async natif, simple	Moins complet, moins mature	Adapté pour petits projets, pas idéal pour production complexe

Django ORM	Très puissant, complet	Dépend Django, lourd	❌ Non recommandé pour cette stack

7.4. Conclusion

Pour notre stack :

- **SQLAlchemy** : ✅ Choix professionnel robuste et flexible. Idéal si vous voulez contrôler tous les aspects de vos modèles et de la base.
- **SQLModel** : ✅ Meilleur compromis rapidité/maintenabilité si vous privilégiez la simplicité et l'intégration native avec FastAPI.
- **Tortoise ORM** : option possible pour projets simples, mais limité pour la production.
- **Django ORM** : à éviter, non compatible avec FastAPI.

Conclusion

Le choix **React (web) + Python (back-end) + React Native (mobile)** n'est pas seulement un choix "à la mode" :

C'est une **combinaison mûre, largement éprouvée et cohérente**, qui :

- maximise la **productivité** de l'équipe,
- facilite la **maintenance** et l'**évolution** du projet,
- s'appuie sur des technologies **massivement utilisées dans l'industrie**,
- laisse la porte ouverte à des fonctionnalités avancées (analyse de données, IA, applications mobiles riches) sans changement de stack.

Dans le contexte d'un projet réalisé par une petite équipe, ce trio constitue donc **un excellent compromis entre simplicité, puissance et pérennité**, tout en valorisant des compétences directement réutilisables en stage ou dans le monde professionnel.