Inclusive Trip Planner

Résumé de Projet pour l'Admission en M2

David Cuahonte Cuevas

Juillet 2025

Contents

1. Introduction

Organiser un voyage accessible représente encore aujourd'hui un véritable défi pour de nombreuses personnes en situation de handicap. Si des progrès notables ont été réalisés dans la conception d'interfaces numériques ou dans l'adaptation de certains lieux publics, la majorité des plateformes de planification de trajets ne prennent que partiellement en compte les contraintes liées à la mobilité, à la perception sensorielle ou aux besoins cognitifs spécifiques. Cette insuffisance se traduit bien souvent par un sentiment d'incertitude, de frustration, et parfois par un renoncement pur et simple à voyager de manière autonome.

C'est dans cette perspective que j'ai conçu le projet *Inclusive Trip Planner*. Cette application mobile, pensée dès l'origine pour répondre aux enjeux de l'inclusion, propose des itinéraires accessibles, préconçus, et enrichis de métadonnées explicites liées à l'accessibilité. L'interface, sobre et intuitive, a été développée pour rester compréhensible et utilisable par tous, y compris par des personnes peu familières avec les outils numériques ou présentant certaines limitations physiques.

Au-delà de la réponse fonctionnelle qu'il apporte, ce projet m'a permis de mobiliser l'ensemble des compétences techniques acquises au cours de ma formation. Il repose sur un socle technologique moderne et robuste : le **frontend** a été réalisé en **Jetpack Compose** (Kotlin), tandis que le **backend** s'appuie sur une architecture modulaire basée sur **Java Spring Boot**, **PostgreSQL** et **Docker**.

D'abord développé dans le cadre de mon mémoire de fin d'études, le projet a ensuite été consolidé pour répondre aux attendus du Mastère 2. À ce titre, j'y ai intégré des tests unitaires côté **frontend**, mis en place une chaîne d'intégration continue (**CI/CD**), et préparé l'ensemble à un déploiement en production. Le présent document en retrace les grandes étapes, en détaillant la conception, les choix techniques, les mises en œuvre concrètes, ainsi que les perspectives d'évolution du système vers un usage à plus grande échelle.

2. Objectifs du projet

Inclusive Trip Planner poursuit une double finalité : répondre à un besoin sociétal concret en matière de mobilité inclusive, tout en démontrant une mise en œuvre rigoureuse d'un projet logiciel full-stack, depuis la définition des exigences jusqu'aux tests et au déploiement.

Du point de vue fonctionnel, l'application vise à proposer un outil accessible et pertinent pour les utilisateurs en situation de handicap. L'objectif principal est de permettre à chacun de découvrir et planifier des itinéraires adaptés à ses contraintes spécifiques. Cela implique une interface simple à prendre en main, la possibilité de filtrer les parcours selon des critères d'accessibilité, et un espace personnel pour sauvegarder ses préférences.

Sur le plan technique, le projet s'est donné comme objectifs :

- D'**analyser les limites des solutions existantes** en matière de voyage accessible ;
- De **définir une architecture complète** et modulaire intégrant frontend et backend de manière fluide ;
- De **développer un MVP opérationnel** avec persistance des données, authentification sécurisée, et navigation réactive côté mobile ;
- De **valider les parcours critiques** de l'utilisateur via des tests automatisés couvrant l'ensemble des entités et interactions essentielles ;
- De **documenter l'ensemble des choix** techniques et méthodologiques dans une optique de pérennisation et d'ouverture future.

Plutôt que d'accumuler des fonctionnalités avancées non finalisées, le projet s'est concentré sur une couverture fonctionnelle cohérente et robuste. La suite envisagée s'inscrit dans une logique de continuité : ajout de feedback utilisateur, enrichissement des métadonnées, et exploration de nouveaux contextes applicatifs autour de la mobilité inclusive.

3. Méthodologie

Le développement de *Inclusive Trip Planner* s'est appuyé sur une approche itérative, pragmatique et orientée test, centrée sur les besoins réels des utilisateurs finaux. L'objectif principal était de livrer un MVP pleinement fonctionnel, tout en posant des bases solides en matière de maintenabilité et d'évolutivité.

Le travail a été structuré en trois phases principales :

- **Phase 1: Mise en place de l'architecture :** Initialisation du backend avec **Spring Boot**, **PostgreSQL** et **Docker**, en parallèle de la création d'un squelette mobile en **Jetpack Compose** (Kotlin). Les choix architecturaux clés notamment l'organisation en couches, le mappage via DTOs, et le flux de données unidirectionnel, ont été définis dès cette étape.

- **Phase 2: Implémentation des fonctionnalités principales :** Développement des parcours critiques tels que l'inscription, la navigation parmi les itinéraires et la personnalisation du profil. Chaque fonctionnalité a été développée de manière full-stack, puis validée manuellement avant d'être soumise à des tests.

- **Phase 3: Tests, intégration CI et amélioration continue :** Ajout de tests d'intégration côté backend et de tests unitaires côté frontend. L'UX et la cohérence des APIs ont été affinées progressivement, en réponse à des cycles courts de retours utilisateurs et de validations sur l'outil.

L'ensemble du développement a été réalisé par un seul développeur, ce qui a permis de garantir une cohérence complète entre la logique métier, les contrats d'API et le comportement de l'interface utilisateur. La communication entre le frontend et le backend repose sur **Retrofit** et des **ViewModels**, en s'appuyant sur les **coroutines** pour gérer l'état de manière réactive dans Jetpack Compose.

L'**accessibilité** a guidé les choix dès les premières étapes, aussi bien dans la modélisation des données que dans l'UX. À titre d'exemple, le parcours d'onboarding a été conçu pour prioriser la collecte des besoins utilisateurs, afin de permettre un filtrage pertinent des itinéraires dès les premiers écrans.

Le projet a également accordé une attention particulière à la modularité et à la testabilité dès le départ. Le code backend suit une architecture strictement en couches (Controller $\rightarrow$ Service $\rightarrow$ Repository), tandis que le frontend repose sur une structure pilotée par les **ViewModels**. Le contrôle de version via **Git** a permis une itération sécurisée, avec un isolement rigoureux des branches fonctionnelles et une validation locale systématique avant tout merge.

4. Gestion de projet

Le suivi du développement de *Inclusive Trip Planner* a été assuré à l'aide d'une **Jira board** structurée, permettant de visualiser et organiser l'ensemble des tâches du projet dans une logique agile. Cette organisation a facilité la priorisation, la traçabilité et la progression continue tout au long des phases de conception, d'implémentation, de test et de documentation.

Les tâches ont été centralisées sous forme de *tickets*, regroupées par **epics** thématiques (par exemple : *Testing and Quality Assurance*, *DevOps and Deployment*, *Final Documentation*) et réparties dans un **sprint principal** courant jusqu'à la fin juillet 2025. Chaque ticket était associé à une catégorie fonctionnelle (*frontend*, *backend*, *CI/CD*, *documentation*) et évoluait dans un pipeline de type *kanban* comportant les étapes suivantes :

- **To Do** : tâches planifiées et prêtes à être entamées ;
- **In Progress** : tickets en cours de développement actif ;
- **In Review** : éléments terminés, en attente de validation ou de test ;
- **Done** : fonctionnalités finalisées, testées et intégrées.

L'usage de Jira a permis d'assurer une transparence totale sur l'avancement des travaux, tout en structurant efficacement les livrables liés aux attendus du Mastère (objectifs C2.2.x et C2.3.x). La gestion du cycle de vie des tickets a été couplée à une stratégie Git rigoureuse (feature branches, pull requests validées après test), garantissant une stabilité du projet et une organisation propice à l'intégration continue.

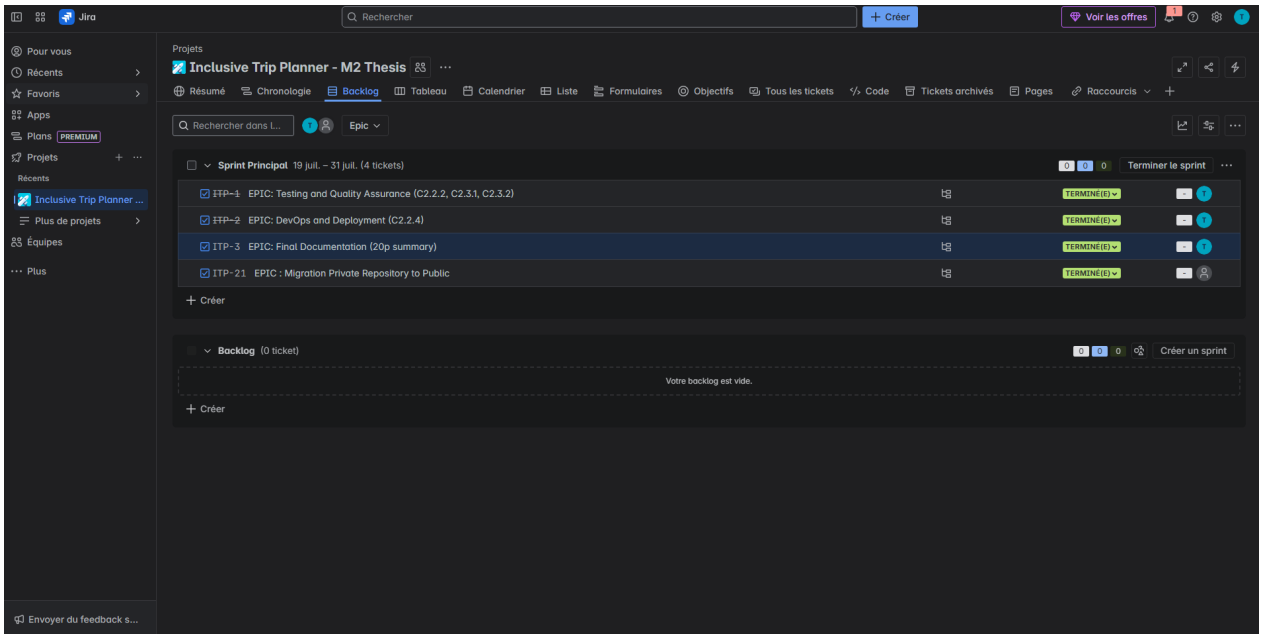


Figure 1: Sprint principal suivi sur Jira, épics livrés le 30 juillet 2025

Estimation budgétaire

Le projet a été développé par une seule personne, en phase MVP. L'estimation ci-dessous se base sur un coût journalier proche du salaire minimum dans le secteur du développement en France. Chaque poste est accompagné d'une remarque sur son évolution potentielle en cas de montée en charge.

Estimation budgétaire indicative

Poste	**Détail**	**Coût estimé**
Temps de développement	Travail de consolidation et documentation, estimé sur la base de 130 €/jour (SMIC développeur)	1 300 €
Hébergement backend (Render)	7 €/mois pendant 3 mois – *Variera selon le nombre de PCU et MAU*	21 €
CI/CD (GitHub Actions)	Plan gratuit – *Valable pour la version actuelle uniquement*	0 €
Stockage base de données (Docker)	Utilisation locale – *À migrer vers le cloud à l'échelle*	0 €
Tests et documentation	Rédaction des tests, validations fonctionnelles et documentation technique, estimée sur la base de 130 €/jour (SMIC QA)	390 €
Coût total estimé		**3 661 €**

Table 1: Estimation budgétaire indicative du projet *Inclusive Trip Planner*

Remarques sur la montée en charge :

- Ce budget correspond à un développement en solo, sans sous-traitance.
- En cas de montée en charge, il faudrait prévoir l'ajout d'un **développeur fullstack**, d'un **QA dédié**, et d'un **chef de projet**.
- L'infrastructure (Render, base de données, CI/CD) devra évoluer selon la charge (nombre d'utilisateurs actifs, scalabilité horizontale).
- Un nouveau chiffrage devra être effectué selon les besoins métiers et techniques futurs.

5. Architecture logicielle

Le projet *Inclusive Trip Planner* repose sur une architecture client-serveur en couches, conçue pour favoriser la modularité, la testabilité et une séparation claire des responsabilités.

Architecture backend

Le backend est développé en **Java Spring Boot** et structuré selon des couches distinctes : *Controller*, *Service*, *Repository* et *DTOs*. Chaque couche assure un rôle bien défini dans l'architecture globale du système :

- **Controllers** : exposent des endpoints REST et délèguent la logique métier aux services.
- **Services** : contiennent la logique métier principale et orchestrent les appels aux repositories.
- **Repositories** : assurent la persistance des données via **Spring Data JPA**.
- **DTOs** : permettent de découpler les modèles internes des contrats d'API exposés.

Exemple : `UserController.java`

```java
@GetMapping("/{id}")
public UserResponse getUserById(@PathVariable UUID id) {
    User user = userService.getUserById(id);
    return UserResponse.from(user);
}
```

Listing 1: Couche Controller exposant les endpoints utilisateur

Exemple : `UserService.java` **(extrait)**

```java
@Transactional
public User createUser(UserRequest dto) {
    String platform = dto.getPlatform() != null ? dto.getPlatform().
       toUpperCase() : "EMAIL";
    if (platform.equals("EMAIL")) {
        if (dto.getPasswordHash() == null || dto.getPasswordHash().isBlank
           ()) {
            throw new IllegalArgumentException("Password is required for
               EMAIL platform users");
        }
    }
    ...
    return userRepository.save(user);
}
```

Listing 2: Couche Service traitant la création d'un utilisateur

Tests d'intégration

Les tests d'intégration sont rédigés à l'aide de l'annotation `@SpringBootTest` et valident le bon fonctionnement des flux complets, incluant les interactions avec la base de données et les mécanismes d'authentification.

```
@Test
void shouldCreateUserAndFetchItById() throws Exception {
    UserRequest request = UserRequest.builder()
        .name("Alice")
        .email("alice@example.com")
        .passwordHash("secret")
        .phone("123456")
        .build();

    MvcResult result = mockMvc.perform(post("/api/users")
        .header("Authorization", "Bearer " + token)
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(request)))
        .andExpect(status().isOk())
        .andReturn();
}
```

Listing 3: Test vérifiant la création et la récupération d'un utilisateur

Architecture frontend

Le **frontend mobile** a été conçu en **Kotlin** à l'aide du framework **Jetpack Compose**. L'architecture repose sur l'utilisation de **ViewModels**, de **Retrofit** pour les appels API, et d'une gestion d'état réactive fondée sur les **coroutines**.

Exemple : `ItineraryViewModel.kt`

```
fun fetchItineraries() {
    viewModelScope.launch {
        try {
            val response = api.getAllItineraries()
            _itineraries.value = response
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

Listing 4: ViewModel récupérant la liste des itinéraires

Exemple de composant : `ItineraryDetailsScreen.kt` (extrait)

```
Text(
    text = itinerary?.title ?: "Loading...",
    fontSize = 26.sp,
    fontWeight = FontWeight.Bold,
    color = Color.Black
)
```

Listing 5: Composable affichant les informations d'un itinéraire

Cette organisation permet de maintenir une séparation claire des responsabilités entre les composants frontend et backend, tout en assurant une forte testabilité et une maintenabilité à long terme.

6. Diagrammes UML et maquettes visuelles

Afin de soutenir la conception fonctionnelle de l'application, deux diagrammes UML ont été réalisés : un diagramme de cas d'utilisation pour illustrer les interactions entre l'utilisateur et le système, ainsi qu'un diagramme d'activités décrivant un scénario type d'onboarding. Trois captures d'écran issues de l'application mobile fonctionnelle sont également fournies à titre de maquettes visuelles.

Diagramme de cas d'utilisation

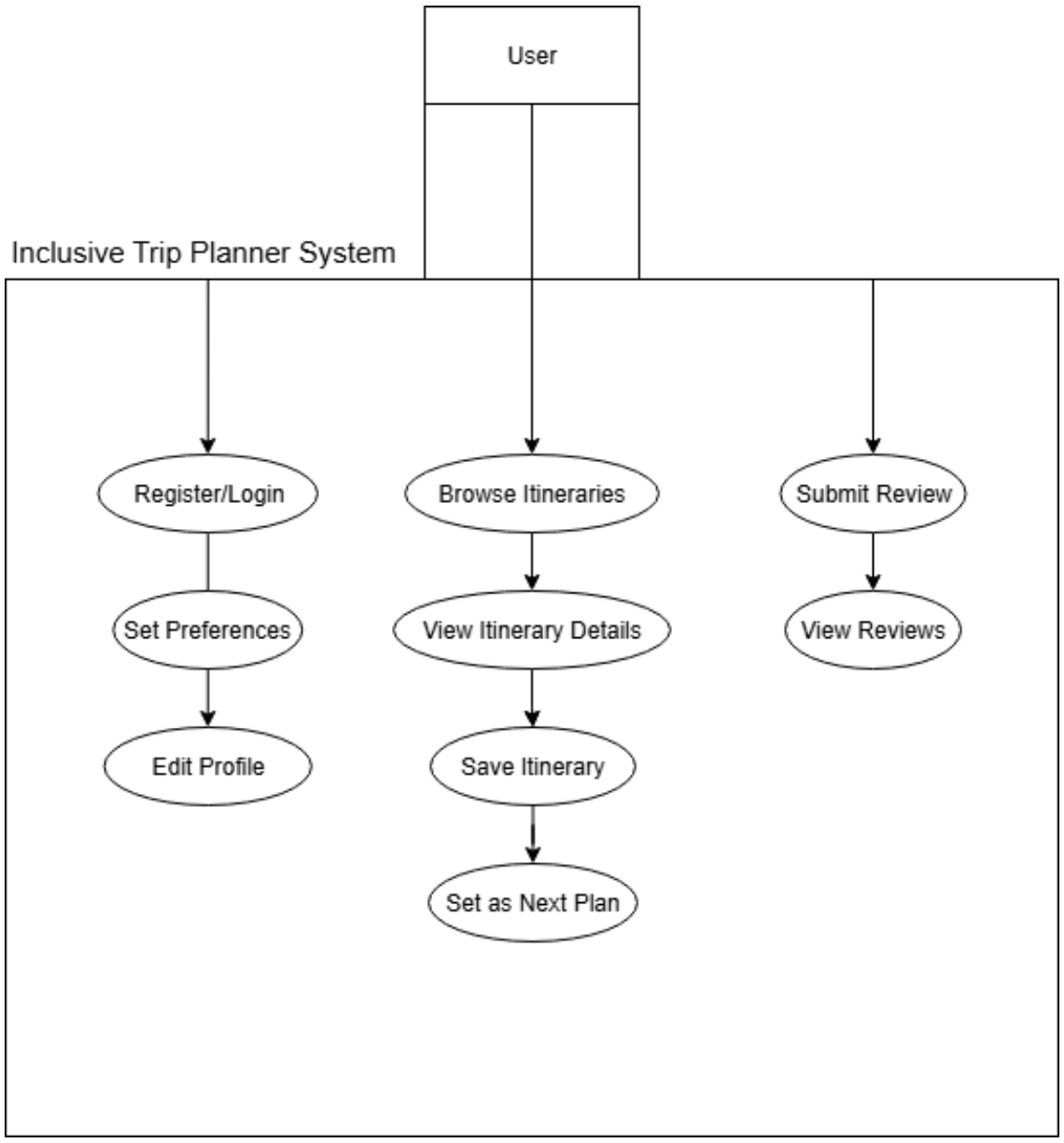


Figure 2: Diagramme de cas d'utilisation: Interactions fonctionnelles entre l'utilisateur et le système

Diagramme d'activités

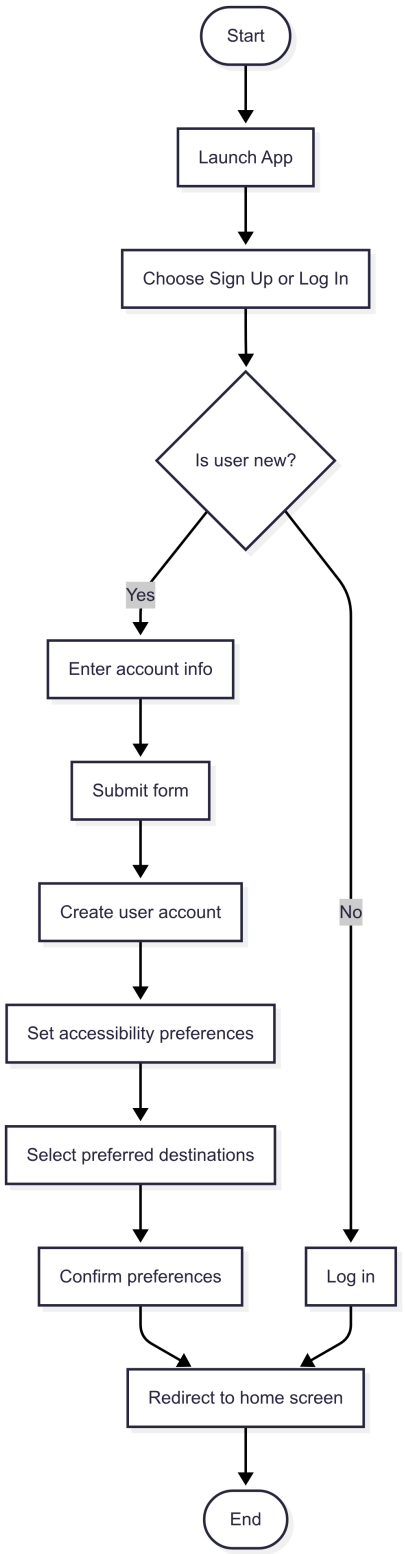
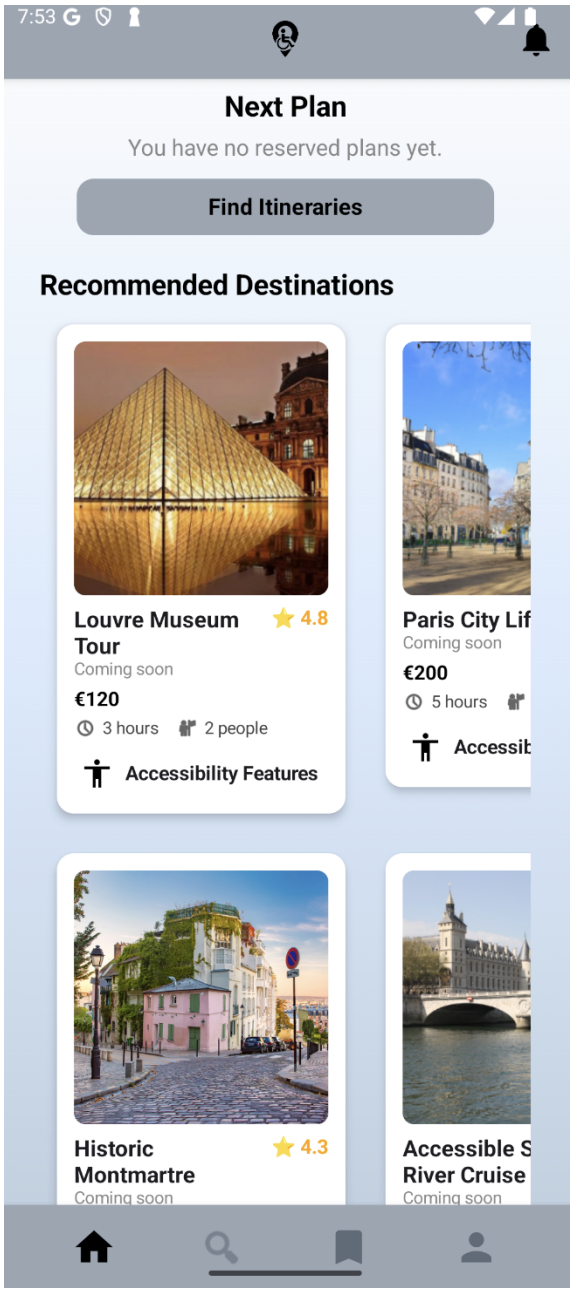


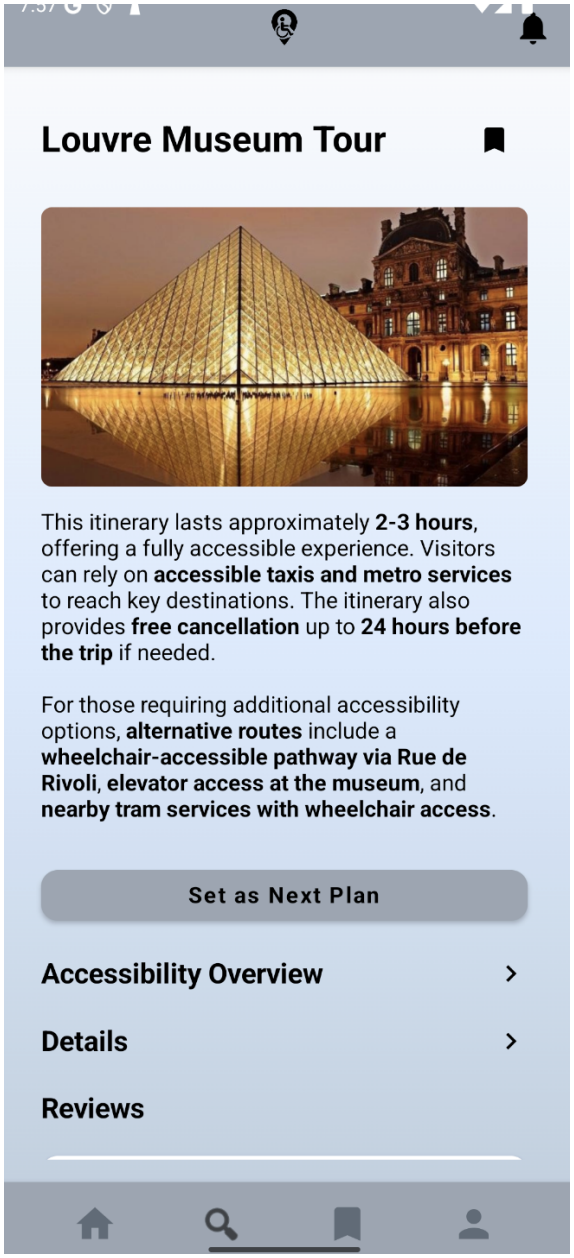
Figure 3: Diagramme d'activités: Parcours d'inscription et sélection des préférences d'accessibilité

Écrans mobiles – Maquettes visuelles

Écrans clés de l'interface mobile: accueil et détails d'itinéraire



(a) Écran d'accueil affichant les itinéraires recommandés et le plan actif



(b) Écran de détails d'un itinéraire avec les informations d'accessibilité

Figure 4: Maquettes mobiles de l'application

Écran de préférences d'accessibilité

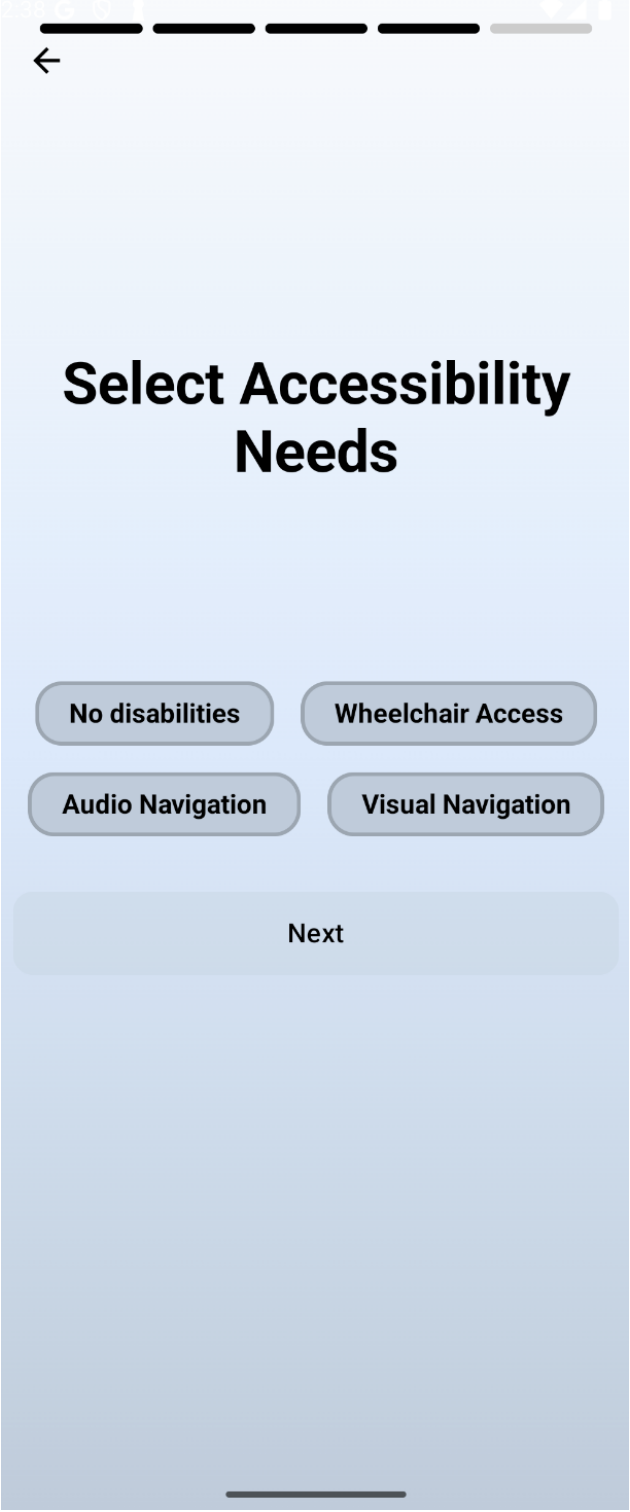


Figure 5: Sélection des préférences d'accessibilité pendant l'onboarding

7. Tests et assurance qualité

La fiabilité de *Inclusive Trip Planner* a constitué un objectif central tout au long du développement. La stratégie de test a couvert à la fois les composants backend et frontend, avec un accent particulier mis sur la validation fonctionnelle, la prévention des régressions, et la préparation aux évolutions futures.

Le **backend** intègre des **tests d'intégration** pour l'ensemble des entités, en s'appuyant sur le framework de test de *Spring Boot* et une base de données H2 en mémoire. Ces tests couvrent les cycles complets de requêtes/réponses, incluant les opérations CRUD, les relations entre entités et les mécanismes d'authentification. Les tests sont basés sur l'annotation `@SpringBootTest`, ce qui permet de simuler des scénarios d'usage réalistes sans avoir recours à des mocks. Un exemple de ces tests a été présenté dans la section dédiée à l'architecture logicielle.

Côté **frontend**, des tests unitaires ont été écrits pour chaque **ViewModel**. Ils s'appuient sur des *mock APIs* et des *coroutine test dispatchers* pour simuler le comportement des appels réseau, tout en vérifiant que les transitions d'état (`collectAsState()`, indicateurs de chargement, etc.) réagissent correctement aux interactions de l'utilisateur. Ces tests sont rédigés en Kotlin à l'aide des outils de test natifs, et sont pleinement compatibles avec l'environnement **Jetpack Compose**.

Le contrôle de version avec **Git** renforce cette approche orientée qualité grâce à un modèle de branches stable. Chaque fonctionnalité est développée et testée de manière isolée, et le code non testé est systématiquement bloqué avant toute fusion vers la branche principale.

Par ailleurs, le projet intègre un pipeline **CI** (basé sur **GitHub Actions**) qui exécute automatiquement les tests d'intégration backend et les tests unitaires frontend à chaque *push* ou *pull request*. Ce mécanisme garantit une validation continue de la stabilité fonctionnelle du projet et contribue activement à prévenir les régressions tout au long du cycle de développement.

Couche	**Type de test**	**Périmètre de couverture**
Backend	Tests d'intégration	Cycle complet pour User, Itinerary, AccessibilityFeature, Settings, Auth
Frontend	Tests unitaires	Logique des ViewModels, gestion d'état, flux API
CI/CD	Exécution du pipeline	Tests automatisés à chaque push/pull request.

Table 2: Résumé de la couverture de tests selon les différentes couches de l'application

La combinaison de tests locaux et d'un pipeline automatisé s'est révélée essentielle pour assurer la stabilité du projet. En particulier, les validations CI ont permis d'éviter des régressions lors des ajouts de dernière minute, et ont renforcé la fiabilité des livrables, y compris dans des cycles d'itération courts et intensifs.

8. Sécurité et authentification

L'application *Inclusive Trip Planner* repose sur un système d'authentification sans état (*stateless*), basé sur des **JSON Web Tokens (JWT)**, afin d'assurer un contrôle d'accès à la fois sécurisé et évolutif. Ce choix d'architecture élimine la nécessité de stocker des sessions côté serveur, tout en facilitant l'intégration avec des clients frontend tels que des applications mobiles.

La sécurité est assurée à l'aide de `Spring Security`, configuré à travers une chaîne de filtres personnalisée. Le backend définit des règles d'autorisation granulaires, protégeant chaque endpoint en fonction du rôle utilisateur et de l'état d'authentification. Les composants clés de cette architecture sont :

- Un **filtre d'authentification JWT**, qui intercepte les requêtes, extrait et valide le token, puis alimente le contexte de sécurité de Spring.
- Un **mécanisme de hachage des mots de passe**, utilisant un algorithme sécurisé tel que `BCrypt`, afin de garantir que les identifiants ne soient jamais stockés en clair.
- Un **contrôle d'accès basé sur les rôles** (*role-based access control*), pour restreindre l'exposition des routes et appliquer des règles d'autorisation strictes au sein de l'API.

Les JWT sont signés à l'aide d'un secret généré côté serveur, et contiennent des *claims* comme l'identifiant utilisateur, le rôle et une date d'expiration. Lorsqu'un utilisateur se connecte ou crée un compte, le backend lui retourne un token signé, que l'application mobile stocke localement (par exemple via les *shared preferences*) et envoie ensuite dans l'en-tête `Authorization` de chaque requête suivante.

Résumé du flux d'authentification

1. L'utilisateur envoie ses identifiants à l'endpoint `/api/auth/login`.
2. Le backend vérifie les informations fournies, puis renvoie un JWT signé.
3. L'application mobile stocke ce token localement, de manière sécurisée.
4. À chaque appel API, le token est ajouté aux en-têtes de la requête.
5. Un filtre Spring dédié valide le token JWT et autorise l'accès aux routes protégées.

Ce fonctionnement permet de disposer d'une API sécurisée, sans état, et capable de monter en charge horizontalement. Il ouvre également la voie à l'intégration de fournisseurs d'identité externes (comme Google ou Facebook via *Firebase Authentication*) ou à des mécanismes d'authentification multi-facteurs.

L'implémentation actuelle reste volontairement ouverte à l'extension. Une stratégie de connexion OAuth2 a d'ailleurs été amorcée à l'aide de tokens Firebase, et l'architecture backend prévoit déjà la possibilité d'ajouter de nouveaux filtres ou fournisseurs au besoin.

En résumé, ce système d'authentification par JWT trouve un juste équilibre entre simplicité d'usage, robustesse technique et évolutivité, tout en préparant l'application à une gestion d'identité plus avancée si nécessaire à l'avenir.

9. Pipeline CI/CD

Le projet *Inclusive Trip Planner* s'appuie sur une chaîne CI/CD complète afin de garantir que chaque modification du code soit automatiquement testée, validée et prête pour la production. Cette automatisation repose sur **GitHub Actions**, **Docker**, et un hébergement cloud sur **Render** pour le backend.

CI backend : tests et déploiement conteneurisé

Le pipeline backend est déclenché à chaque `push` ou `pull request` vers `main` ou `develop`. Il exécute les étapes suivantes :

- Compilation du code sous Java 21 ;
- Construction d'une image Docker avec `docker build` ;
- Lancement de l'architecture complète (Spring Boot + PostgreSQL) via `docker compose`
- Attente active de la disponibilité de l'API via une boucle `curl` sur `/health` ;
- Exécution des tests d'intégration avec `mvn test` ;
- Envoi conditionnel de l'image vers Docker Hub si les tests passent.

```
for i in {1..10}; do
  if curl --fail http://localhost:8080/health; then exit 0; fi
  sleep 3
done
exit 1
```

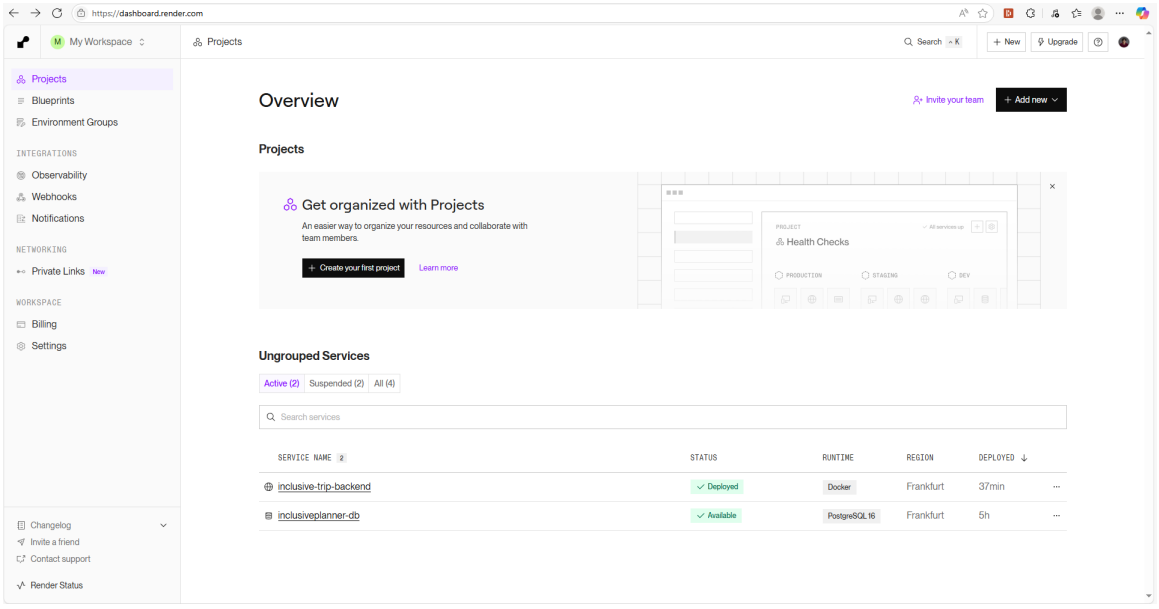


Figure 6: Déploiement automatisé du backend sur Render depuis Docker Hub

CI frontend : compilation, tests unitaires et couverture de code

Le frontend mobile, développé en **Jetpack Compose**, dispose de sa propre pipeline CI définie dans un dépôt GitHub séparé. Celle-ci s'exécute à chaque `push` ou `pull request` vers `main` ou `develop`, et repose sur trois *jobs* indépendants exécutés en parallèle :

- **Build APK** : compilation du projet Android via `./gradlew assembleDebug`, sur un environnement configuré avec JDK 17 et Android SDK. Cela permet de s'assurer que l'application est générable à tout moment.
- **Unit Tests** : lancement des tests unitaires Kotlin via `testDebugUnitTest`, avec simulation des appels API par *Mock APIs* et utilisation de *coroutine test dispatchers*. Les artefacts sont archivés pour vérification.
- **Lint et Couverture** : exécution de `lintDebug` pour détecter les erreurs de style ou de logique, suivie d'une génération de rapports de couverture de code avec **JaCoCo** (`:app:jacocoTestReport`). Les résultats sont exportés au format HTML.

```
- name: Run Lint
  run: ./gradlew lintDebug

- name: Run Code Coverage
  run: ./gradlew :app:jacocoTestReport
```

Ce découpage en tâches indépendantes permet d'isoler les erreurs plus facilement et de garantir une validation rapide, même en cas de changements fréquents. Les artefacts produits (résultats de test, rapports de lint, couverture) sont ensuite accessibles directement depuis GitHub pour revue.

L'application Android n'est pas encore déployée automatiquement, mais ce pipeline garantit une qualité continue de la base de code mobile. À terme, il est prévu d'y ajouter une étape de publication vers un store ou une distribution interne.

10. Contraintes et limites

Le développement de *Inclusive Trip Planner* a soulevé plusieurs défis, à la fois techniques et organisationnels. En tant que développeur unique en charge de l'ensemble du stack, il a fallu opérer des choix stratégiques concernant la portée fonctionnelle, l'architecture, et la répartition du temps, tout en assurant la cohérence et la fiabilité de la base de code.

L'un des défis majeurs a été de maintenir une cohérence de bout en bout à travers les différentes couches du système : modélisation des données, logique métier côté backend, gestion des états frontend, et réactivité de l'interface utilisateur. Cette exigence a nécessité une planification rigoureuse ainsi qu'une discipline stricte dans l'usage de Git, afin d'éviter les régressions et les changements non testés.

Une autre complexité est apparue dans l'équilibre entre accessibilité et usage général. Faire de l'accessibilité un critère central impliquait d'élargir le modèle de données, de concevoir de nouveaux parcours utilisateurs (comme la collecte de préférences dès l'onboarding), et de gérer des cas limites rarement abordés dans un MVP classique. Cela a renforcé la complexité fonctionnelle et architecturale de la majorité des écrans.

Les tests ont également mis en lumière plusieurs arbitrages. Le développement initial s'est fait sans pipeline automatisé, en s'appuyant uniquement sur des validations manuelles et des workflows Git locaux. Si cette approche a permis de démarrer rapidement, elle a toutefois ralenti la boucle de retour et introduit des frictions. L'ajout progressif de pipelines CI/CD permet aujourd'hui d'améliorer considérablement l'automatisation des tests et la préparation au déploiement.

Limites connues du MVP actuel :

- Les suggestions d'itinéraires sont prédéfinies et ne s'adaptent pas dynamiquement à l'historique ou à la géolocalisation de l'utilisateur.
- Les filtres d'accessibilité reposent sur un schéma de métadonnées fixe, sans données en temps réel ni contributions communautaires.
- La gestion des erreurs et des cas limites est fonctionnelle, mais pas encore entièrement optimisée pour une mise en production à grande échelle.

Malgré ces limitations, le MVP atteint pleinement ses objectifs essentiels : proposer une expérience de planification de voyage accessible, sécurisée et bien structurée sur mobile. Son architecture modulaire et sa couverture de test posent des fondations solides pour une évolution future vers plus de personnalisation, de montée en charge, et d'intégration avec des services tiers liés à la mobilité.

11. Conclusion et perspectives d'évolution

Le projet *Inclusive Trip Planner* aboutit à un MVP robuste, accessible et spécifiquement conçu pour les personnes ayant des besoins en matière de mobilité ou d'accessibilité. Il conjugue une expérience utilisateur soignée, une sélection d'itinéraires adaptés et une architecture full-stack moderne, pour aboutir à une solution fonctionnelle, centrée sur l'utilisateur. L'application permet de découvrir, sauvegarder et suivre des parcours personnalisés, alignés avec les préférences et contraintes propres à chaque profil.

D'un point de vue technique, le projet témoigne d'une maîtrise complète de la pile logicielle : développement réactif du frontend avec **Jetpack Compose**, backend sécurisé et scalable sous **Spring Boot**, tests d'intégration et mise en place d'une chaîne CI pour assurer la maintenabilité à long terme. Les principes fondamentaux d'architecture, la séparation des responsabilités, conception d'API via DTOs, et itérations guidées par les tests ont été respectés tout au long du développement.

L'accessibilité n'a pas été traitée comme une exigence secondaire, mais intégrée dès la phase de modélisation des données et de conception des interfaces. Ce positionnement a permis de poser des fondations qui allient performance technique et inclusion réelle dans l'expérience utilisateur.

Les évolutions futures envisagées portent sur les axes suivants :

- **Génération dynamique d'itinéraires** : passer de parcours statiques à une composition intelligente, assistée par règles ou IA, en fonction des données utilisateur en temps réel.
- **Systèmes de recommandation** : introduire une logique adaptative capable de suggérer des itinéraires en fonction du comportement, du profil d'accessibilité ou de l'historique de retours utilisateur.
- **Enrichissement des métadonnées d'accessibilité** : intégrer des mises à jour en temps réel, des contributions communautaires et des données issues d'APIs externes pour étoffer l'annotation des parcours.
- **Adaptation au secteur de la santé** : réorienter la plateforme pour accompagner les patients à mobilité réduite dans l'organisation de trajets accessibles vers les cabinets médicaux ou hôpitaux, en partenariat avec des professionnels de santé.

Le MVP actuel pose une base modulaire solide, capable d'évoluer aussi bien en termes de domaine d'application que d'échelle. Qu'il s'agisse d'un assistant de voyage personnel ou d'un outil au service de l'inclusion, *Inclusive Trip Planner* se positionne comme une solution porteuse de sens, à la croisée entre excellence technique et conception centrée sur l'humain.

Je remercie Ynov pour l'attention portée à ce projet, et suis pleinement motivé à poursuivre ce travail au sein du Mastère, afin d'approfondir mes compétences et mener ce projet vers de nouveaux horizons.

Annexe

L'annexe ci-dessous présente de manière synthétique les principaux outils techniques utilisés au cours du projet, ainsi que les raisons de leur sélection.

- **Kotlin + Jetpack Compose** – Utilisés pour le développement frontend en raison du support natif Android, de la composition déclarative d'interfaces, et de leur parfaite compatibilité avec une architecture pilotée par ViewModel. Le modèle réactif de Compose, combiné aux *coroutines* intégrées, en fait un choix idéal pour gérer l'état asynchrone dans une application mobile conçue dès l'origine pour ce support.

- **Spring Boot (Java)** – Choisi pour le développement backend grâce à la richesse de son écosystème, à la simplicité de création d'APIs REST, et à son support naturel des architectures en couches. Son intégration avec **Spring Security** et **JPA** en fait une solution robuste pour développer des applications sécurisées et évolutives.

- **PostgreSQL** – Base de données relationnelle retenue pour sa stabilité, ses performances d'indexation, et sa compatibilité avec **Liquibase**. Son support natif des UUIDs et des structures JSON a été particulièrement utile dans la modélisation de certaines entités et préférences utilisateur.

- **Docker** – Utilisé pour conteneuriser les services backend ainsi que la base de données PostgreSQL, garantissant une cohérence d'environnement entre le développement, les tests et la production. **Docker Compose** a permis d'orchestrer facilement les tests d'intégration et les simulations locales de l'architecture complète.

- **GitHub Actions** – Mis en place comme socle de l'intégration continue (**CI**) pour le frontend et le backend. Il assure une automatisation fiable des tests, du linting, de la couverture de code, de la génération d'images Docker, et des workflows de déploiement — avec une validation systématique à chaque *push* avant fusion.

- **Liquibase** – Outil de gestion des migrations de base de données, assurant l'évolution du schéma de manière versionnée et réversible. Il garantit que chaque état de la base est traçable, reproductible, et validé avant le démarrage de l'application, ce qui est essentiel pour la stabilité en test et en production.

- **Render** – Plateforme choisie pour l'hébergement du backend sous forme de conteneur Docker. Elle propose une interface simple pour le déploiement et la gestion des secrets, ainsi qu'un hébergement HTTPS, des journaux de build et des vérifications d'état. Elle offre une alternative légère à des plateformes CI/CD plus complexes.

- **Retrofit** – Librairie utilisée côté frontend pour gérer les appels réseau de manière structurée et typée. Elle permet de transformer automatiquement les réponses JSON du backend en objets Kotlin, tout en s'intégrant nativement avec les *coroutines* et les `ViewModels`, facilitant ainsi la communication entre l'interface et le backend.