**Choix des technologies**

Pour ce projet, le langage choisi est **Python**. Ce choix est motivé par sa simplicité d’écriture, sa rapidité de développement et la richesse de son écosystème. De nombreuses bibliothèques, telles que *requests*, *polyline* ou *sqlite3*, permettent de gérer facilement des requêtes HTTP, le traitement de données géographiques et la manipulation d’une base de données.

Côté interface, l’utilisation de **Kivy** et **KivyMD** permet de créer des interfaces réactives et multi-plateformes. Ces frameworks facilitent la mise en place d’écrans dynamiques, de menus interactifs et d’autres éléments graphiques essentiels pour une application mobile ou de bureau.

Pour la partie Back-End, le projet s’appuie sur **SQLite** comme système de gestion de base de données (SGBD). SQLite est particulièrement adapté pour des projets de taille modeste : il est léger, ne nécessite pas de serveur dédié et s’intègre parfaitement avec Python. On retrouve ainsi dans les modules de gestion de base de données (comme dans *DatabaseManager.py* et *DatabaseGenerator.py*) les opérations d’insertion, de mise à jour et de suppression des données.

**Structure du projet**

La structure adoptée vise à séparer clairement les responsabilités, ce qui facilite la maintenance et l’évolution du projet. On distingue ainsi plusieurs répertoires et fichiers :

* **Dossier pour les vues (interface utilisateur) :**  
  Les fichiers de l’interface (notamment dans *main.py*) contiennent la logique de présentation, la gestion des écrans, des menus et des interactions avec l’utilisateur.
* **Dossier pour les modèles de données :**  
  Les classes et modules qui représentent les entités du projet, comme les bâtiments et les salles, sont regroupés dans des fichiers tels que *DatabaseManager.py* et *DatabaseGenerator.py*. Ces modules gèrent également la communication avec la base de données SQLite.
* **Dossier pour les fonctionnalités spécifiques (itinéraires et géodonnées) :**  
  Les fonctionnalités relatives au calcul d’itinéraires et au traitement des données géographiques sont implémentées dans *itineraire.py* et *geojson.py*. Ces modules interagissent avec des API externes (comme Valhalla) et permettent de générer des fichiers GeoJSON à afficher sur la carte.

Cette organisation en modules distincts permet de maintenir une séparation nette entre la logique de présentation (front-end), la gestion des données (back-end) et le traitement spécifique des itinéraires.

**Développement des fonctionnalités**

Le projet inclut diverses fonctionnalités essentielles, notamment :

* **Ajout, suppression et modification des enregistrements :**  
  Les opérations CRUD sont mises en œuvre dans les modules gérant la base de données. Par exemple, *DatabaseManager.py* définit des méthodes pour insérer des bâtiments et des salles, mettre à jour les coordonnées ou le chemin vers les fichiers GeoJSON, et supprimer des entrées si nécessaire.
* **Calcul d’itinéraires et affichage sur carte :**  
  La fonctionnalité de calcul d’itinéraire est implémentée dans *itineraire.py* via un appel à l’API Valhalla. Une fois récupéré, l’itinéraire est décodé, converti en coordonnées utilisables et sauvegardé sous forme de fichier GeoJSON. Ce fichier est ensuite chargé et affiché sur la carte dans l’interface (voir le module *main.py*).

**Interface utilisateur interactive :**  
 Grâce à Kivy et KivyMD, l’application permet à l’utilisateur de rechercher un bâtiment ou une salle, d’obtenir des suggestions, puis de visualiser l’itinéraire sur une carte. L’interface gère aussi l’actualisation dynamique des données, par exemple lors de la saisie d’une adresse ou d’une sélection dans le menu.  
  
  
Pour garantir la robustesse et la fiabilité de chaque fonctionnalité développée, il est indispensable de mettre en place des tests unitaires.

**Choix de l'outil de test**

* **Bibliothèque standard :** Python propose le module *unittest* qui permet de définir des classes de tests, d’organiser des cas de tests et d’exécuter des assertions pour vérifier que les fonctions se comportent comme prévu.

**Structure des tests**

Chaque fichier de test correspond à un module du projet, comme :

* test\_itineraire.py pour vérifier la récupération et le décodage des itinéraires via l’API Valhalla,
* test\_database.py pour tester les opérations CRUD sur la base de données SQLite,
* test\_geojson.py pour s’assurer que le traitement des fichiers GeoJSON se déroule correctement.

**Cas de test :** Pour chaque fonctionnalité, les tests couvrent :

* Les cas d’usage normaux (entrée valide et résultats attendus),
* Les cas limites (exemple : données manquantes ou incorrectes),
* Les cas d’erreurs, afin de vérifier que les exceptions sont bien gérées (par exemple, simuler une réponse erronée de l’API).

**Rédaction et exécution**

* **Assertions :** Chaque test doit comporter des assertions claires pour comparer les résultats obtenus aux résultats attendus. Cela permet d’identifier rapidement les écarts et les dysfonctionnements.
* **Exécution automatisée :** Les tests peuvent être exécutés via la commande python -m unittest .

En résumé, le projet est structuré de manière modulaire avec Python, en utilisant Kivy/KivyMD pour le front-end, SQLite pour la gestion des données, et des modules dédiés pour traiter et afficher les itinéraires. Chaque technologie et chaque structure choisie s’inscrivent dans une démarche de développement rapide, maintenable et adaptée aux besoins du projet.