**1. Test unitaire (Unit Test)**

**Objectif :** Vérifier le bon fonctionnement de chaque fonction ou méthode isolée.

**Exemples de tests :**

* **Fonction de décodage de polyline :**  
  Vérifier que la fonction de *itineraire.py* décode correctement la polyline renvoyée par l’API Valhalla et renvoie la liste attendue de coordonnées après correction.
* **Extraction d'adresses :**  
  Tester la fonction extract\_adresse dans *DatabaseManager.py* pour s’assurer qu’elle extrait correctement les coordonnées (longitude, latitude) à partir du dictionnaire, notamment pour les cas valides et les cas où la clé est absente.
* **Fonction de construction de chemins GeoJSON :**

• Vérifier que la fonction path\_suffixe de *geojson.py* identifie correctement les fichiers correspondant à un suffixe donné.

* **Extraction d'adresses (DatabaseManager.py) :**  
  • Clé existante : Passer un dictionnaire contenant une entrée comme "bat3": "43.305403, 5.378269" et s’assurer que la fonction extract\_adresse retourne la liste [43.305403, 5.378269].

• Clé absente : Tester avec un numéro de bâtiment non présent dans le dictionnaire pour vérifier que la fonction retourne None ou gère le cas de manière appropriée.  
• Format incorrect : Fournir une chaîne mal formatée (par exemple "43.305403-5.378269") pour s’assurer que la fonction détecte l’erreur et réagit correctement (par exemple, en levant une exception ou en retournant une valeur par défaut).

* **Construction de chemins GeoJSON (geojson.py) :**

• Vérification du suffixe : Tester la fonction path\_suffixe avec un chemin de fichier se terminant par le suffixe attendu et vérifier qu’elle renvoie True.

• Test négatif : Fournir un chemin dont le nom ne correspond pas au suffixe et s’assurer que la fonction retourne False.

• Cas particulier : Tester avec un répertoire contenant plusieurs fichiers .geojson pour s’assurer que la fonction path() renvoie bien une liste complète et correcte des chemins, même si le répertoire est vide ou n’existe pas.

**2. Test d'intégration**

**Objectif :** Valider l’interaction entre plusieurs modules pour assurer un fonctionnement cohérent de bout en bout.

**Exemples de tests :**

* **Itinéraire complet :**  
  Simuler une requête dans *itineraire.py* pour récupérer un itinéraire, générer le fichier GeoJSON, puis vérifier que ce fichier est correctement chargé et affiché dans la carte via le module d’interface (*main.py*).
* **Intégration base de données – interface utilisateur :**  
  Tester le flux complet depuis la recherche d’un bâtiment via l’interface jusqu’à la récupération des coordonnées depuis la base SQLite (via *DatabaseManager.py*). Vérifier que les suggestions affichées et la sélection d’un bâtiment entraînent le chargement correct de la couche GeoJSON correspondante.
* **Insertion et mise à jour des données :**  
  Valider que les fonctions de création et de mise à jour des enregistrements (batiments et salles) interagissent correctement entre *DatabaseGenerator.py* et *DatabaseManager.py*.

**3. Test fonctionnel**

**Objectif :** Vérifier que le système répond aux exigences fonctionnelles définies par le cahier des charges.

**Exemples de tests :**

* **Recherche de bâtiment :**  
  Tester que la saisie d’un nom ou d’un identifiant de bâtiment dans le champ de recherche affiche bien une liste de suggestions, et qu’en sélectionnant l’un d’eux, l’itinéraire et la carte se mettent à jour correctement.
* **Calcul d’itinéraire :**  
  Vérifier que, lorsqu’un utilisateur saisit des points de départ et d’arrivée valides (soit par coordonnées, soit par nom de bâtiment ou salle), le système récupère et affiche un itinéraire complet (instructions, distance et durée estimée).
* **Navigation et changement d’écran :**  
  Confirmer que les transitions entre l’écran de recherche et l’écran de l’itinéraire fonctionnent sans accroc et que la carte se recentre correctement sur la position recherchée.

**4. Test de performance**

**Objectif :** Mesurer la réactivité et la capacité du système en conditions de charge.

**Exemples de tests :**

* **Temps de réponse API et traitement GeoJSON :**  
  Mesurer le temps nécessaire pour obtenir la réponse de l’API Valhalla, décoder la polyline et générer le fichier GeoJSON. Vérifier que le délai reste acceptable pour une utilisation interactive.
* **Chargement de la carte :**  
  Simuler un scénario où plusieurs couches GeoJSON sont ajoutées successivement à la carte (par exemple, lors d’une recherche rapide de différents bâtiments) et mesurer l’impact sur la fluidité et le temps de rafraîchissement de l’interface.

**5. Test de sécurité**

**Objectif :** S’assurer que le système est robuste face aux attaques et vulnérabilités potentielles.

**Exemples de tests :**

* **Validation des entrées utilisateur :**  
  Tester la saisie de données malicieuses ou inattendues (exemple : injection SQL via le champ de recherche) afin de vérifier que le système échappe correctement aux caractères spéciaux et empêche toute attaque par injection.
* **Gestion des erreurs et journalisation :**  
  Simuler des scénarios d’erreurs (par exemple, indisponibilité de l’API Valhalla ou accès à un fichier GeoJSON inexistant) pour s’assurer que le système gère proprement les exceptions, n’affiche pas de messages sensibles et enregistre les erreurs pour une analyse ultérieure.
* **Accès aux ressources sensibles :**  
  Vérifier que les fichiers sensibles (tels que la base de données ou les logs) ne sont pas accessibles directement par un utilisateur non autorisé, et que le système ne divulgue pas d’informations techniques dans les messages d’erreur.