PROJET APPLI-VISITES

Application d'enregistrement des visites entre les visiteurs et les praticiens

---

I. Présentation de l'entreprise

GSB est une entreprise pharmaceutique qui a été créée par la fusion entre le géant américain Galaxy et le conglomérat européen Swiss Bourdin. Elle se spécialise dans les médicaments pour les maladies virales et conventionnelles, avec son siège administratif à Paris et son siège social à

Philadelphie. Après la fusion, l'entreprise vise à améliorer ses opérations en réalisant des économies d'échelle grâce à des restructurations et des licenciements, tout en préservant les atouts des deux laboratoires.
Chez GSB, les représentants médicaux doivent conseiller et informer les professionnels de la santé sur les produits de l'entreprise sans réaliser des ventes directes. Les délégués médicaux sont organisés selon le modèle existant chez Galaxy, avec une hiérarchie par région et par secteur géographique.
L'entreprise vise à gérer efficacement les dépenses liées aux déplacements et aux actions sur le terrain tout en préservant son image de marque. Elle veut unifier la gestion des dépenses et des remboursements pour tous les visiteurs.

II. Présentation de l'application

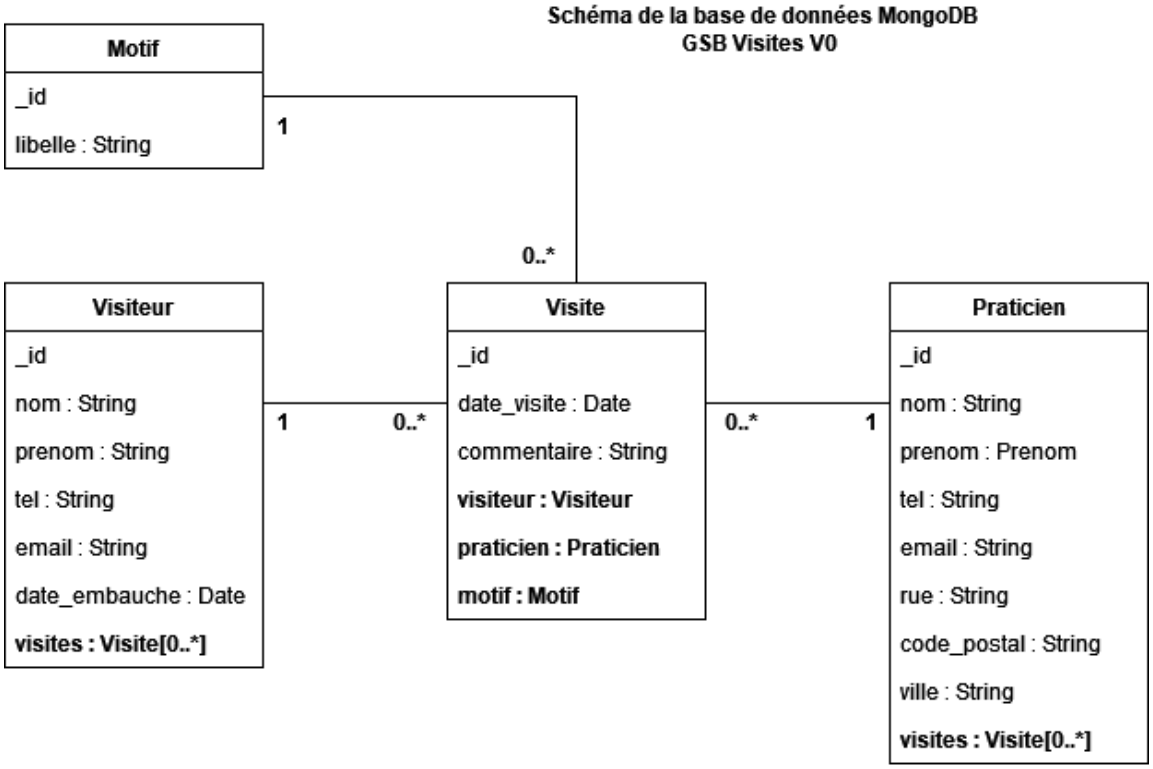
Une application Android a été développée pour gérer les visites des visiteurs médicaux chez les praticiens. Cette application permet aux utilisateurs d'accéder à des informations sur les praticiens, les visites et les visiteurs du laboratoire, facilitant ainsi la communication au sein de la force commerciale.

Elle offre les fonctionnalités suivantes :

- Authentification des visiteurs avec un système de connexion sécurisé.
- Enregistrement des comptes-rendus de visite, incluant la date, le motif de la visite, un commentaire, ainsi que les détails du visiteur et du praticien associés.
- Affichage des détails des praticiens liés à chaque visiteur connecté.

L'application est développée pour l'environnement OS Android en utilisant le langage de programmation Java et l'IDE Android Studio.Elle respecte les conventions de l'architecture POO. Pour assurer la sécurité des données, une authentification des utilisateurs est nécessaire pour accéder à ses fonctionnalités.
Côté serveur, une API a été créée à l'aide du framework Node.js, notamment Express.js. Cette API définit des routes répondant à des requêtes HTTP spécifiques (GET, POST, PUT, DELETE) et renvoie des données formatées en JSON à l'application mobile.

Les données sont stockées dans une base de données NoSQL MongoDB. Cette dernière établit un lien avec l'API Node.js, permettant la connexion à la base de données, la définition de modèles de données, l'exécution d'opérations CRUD et le traitement des résultats, rendant les données disponibles via des requêtes HTTP.



III. Détails de l'application

L'application a donc plusieurs fonctionnalités :

A. Authentification :

- Côté serveur :
  Le controller *userController* permet à l'utilisateur de se connecter. Les informations d'identification (email et mot de passe) sont envoyées au serveur via une requête HTTP. Le contrôleur de l'API recherche l'utilisateur dans la base de données en utilisant l'adresse e-mail fournie.

Si l'utilisateur n'est pas trouvé, une réponse d'erreur est renvoyée indiquant que l'utilisateur n'existe pas.
Si l'utilisateur est trouvé, le mot de passe saisi est comparé au mot de passe haché stocké dans la base de données à l'aide de la fonction *bcrypt.compare()*. Si les mots de passe correspondent, un token JWT est généré et renvoyé à l'utilisateur avec des informations sur l'utilisateur, telles que son ID, nom, prénom et email.
Ce token JWT contient l'ID de l'utilisateur et est signé avec une clé secrète à l'aide de la fonction *jwt.sign()*.
Le token JWT est envoyé dans la réponse, et l'utilisateur peut l'utiliser pour authentifier ses requêtes ultérieures. Le middleware auth vérifie la validité du token JWT envoyé avec la requête.
Le middleware extrait le token JWT de l'en-tête *Authorization* de la requête et le décode à l'aide de la clé secrète spécifiée.
Si le token est valide, l'ID de l'utilisateur est ajouté à l'objet req pour permettre l'accès à l'utilisateur authentifié dans les routes suivantes, sinon il renvoie un message d'erreur

- Côté application :
  L'activité *MainActivity* gère l'interface utilisateur de la page de connexion dans l'application Android. Il utilise le View Binding pour accéder aux éléments de l'interface utilisateur définis dans le fichier XML. Lorsque l'utilisateur clique sur le bouton de connexion, le controller récupère les informations saisies dans les champs d'email et de mot de passe. Ensuite, il crée une instance de Visiteur avec ces informations et utilise Retrofit pour effectuer une requête API de connexion asynchrone. Si la connexion réussit, l'application redirige vers une autre activité avec les informations de l'utilisateur. En cas d'échec, un message d'erreur est affiché à l'utilisateur.

B. Visualisation de la liste des praticiens du visiteur :

- Côté serveur :
  Le controller *visiteurController* permet qu'une fois authentifié, le visiteur est identifié par son ID, qui est stocké dans le token JWT.

Le controller utilise cet ID pour récupérer les informations du visiteur à l'aide de la fonction *getOneVisiteur*.
Lors de la récupération du visiteur, le controller utilise populate pour inclure les données des praticiens dans le résultat. Cela est réalisé en utilisant le champ *portefeuillePraticiens* du model *Visiteur*, qui est une référence aux praticiens associés à ce visiteur. Une fois les informations récupérées et les praticiens inclus, une réponse HTTP est renvoyée au client contenant les détails du visiteur, y compris les détails de son portefeuille de praticiens.

- Côté application :
  L'activité *ListPraticiensActivity* permet d'afficher la liste des praticiens pour le visiteur connecté. L'instance de la classe *ActivityListPraticiensBinding* est utilisée pour accéder aux vues définies dans le fichier XML de mise en page *activity_list_praticiens.xml*. Des variables privées sont déclarées pour gérer la RecyclerView et son adapteur (*PraticiensRecyclerViewAdapter*). L'objet Visiteur est récupéré en tant qu'extra d'intention de l'activité précédente (*MainActivity*), il contient les informations du visiteur, y compris le token d'authentification. L'instance *GsbVisitesServices* s'occupe d'effectuer des appels API à l'aide de Retrofit. Un appel API est créé pour récupérer les informations du visiteur, en incluant la liste des praticiens associés à ce visiteur.
  Dans la méthode *onResponse*, si la réponse de l'API est réussie, la liste des praticiens est mise à jour avec les praticiens associés au visiteur connecté.
  Dans la méthode *onFailure*, si la réponse échoue, un message d'erreur est affiché à l'utilisateur à l'aide d'un Toast.

C. Visualisation des informations d'un praticien

- Côté serveur :
  Le controller *praticienController* permet de rechercher le praticien dans la base de données grâce à la méthode *findOne*, l'ID du praticien est passé en paramètre de la requête.
  Si le praticien est trouvé, il est renvoyé avec toutes ses informations, y compris les visites associées (via la méthode

populate).Tous les praticiens sont récupérés de la base de données avec la méthode find.
Chaque praticien est renvoyé avec ses informations, y compris les visites associées (via la méthode populate).

- Côté application :
  L'activité *InfosPraticienActivity* affiche les informations détaillées d'un praticien sélectionné. Cette activité récupère les informations du praticien sélectionné depuis l'activité précédente (*ListPraticiensActivity*) et affiche ces informations à l'utilisateur.
  La classe générée *ActivityInfosPraticienBinding* est utilisée pour accéder directement aux vues définies dans le fichier XML de mise en page *activity_infos_praticien.xml*. L'activité récupère l'objet *Praticiens* passé en tant qu'extra d'intention de l'activité précédente pour obtenir les détails du praticien à afficher.
  La méthode *onCreate* est utilisée pour initialiser l'interface utilisateur de l'activité et récupérer les informations du praticien. Les références aux vues *TextView* sont récupérées à partir du fichier de mise en page XML, puis le texte de ces vues est mis à jour avec les informations spécifiques du praticien telles que son nom, son prénom et son mail. L'affichage des visites n'est pas finalisé.

IV. Fonctionnement du code

Il y a deux parties de code différentes, le code côté serveur(API) et le code côté application(Android). Le code se constitue donc de la manière suivante:

A. API

1. Controllers

- visiteController :

  Ce contrôleur gère les opérations CRUD pour les visites dans l'application. La fonction *createVisite* permet de créer une visite en récupérant les données de la requête, créant

une nouvelle visite, et l'enregistre en base de données pui met à jour le praticien associé, et retourne une réponse HTTP 201 en cas de succès.
La fonction *getOneVisite* récupère une visite spécifique, peuple les références visiteur et praticien, et retourne la visite avec une réponse HTTP 200 ou une réponse HTTP 404 si elle n'est pas trouvée. La fonction *modifiyVisite* récupère les données de la requête, met à jour la visite en base de données, et retourne une réponse HTTP 201 en cas de succès. La fonction *deleteVisite* supprime une visite spécifique et retourne une réponse HTTP 200 en cas de succès. Enfin la fonction *getAllVisites* récupère toutes les visites de la base de données, peuple les références visiteur et praticien, et retourne toutes les visites sous forme de tableau.
Les fonctions utilisent express-async-handler pour gérer les erreurs asynchrones et renvoient les réponses HTTP appropriées en cas de succès ou d'échec de l'opération.

- motifController :

  Ce contrôleur gère les opérations CRUD pour les motifs de l'application. La fonction *createMotif* permet de créer un motif en récupérant son libellé. Elle crée ensuite un nouvel objet Motif, l'enregistre en base de données, et retourne une réponse HTTP 201 en cas de succès. La fonction *getOneMotif* récupère un motif spécifique, et retourne le motif avec une réponse HTTP 200 ou une réponse HTTP 404 si non trouvé. La fonction *modifyMotif* récupère le libellé mis à jour du motif, le met à jour en base de données, et retourne une réponse HTTP 201 en cas de succès.La fonction *deleteMotif* supprime un motif spécifique, et retourne une réponse HTTP 200 en cas de succès.La fonction *getAllMotifs* récupère tous les motifs de la base de données, et retourne tout les motifs sous forme de tableau avec une réponse HTTP 200.
  Les fonctions utilisent express-async-handler pour gérer les erreurs asynchrones et renvoient les réponses HTTP appropriées en cas de succès ou d'échec de l'opération.

2. Models

- motif :

  Ce fichier définit un modèle Mongoose pour l'entité *Motif*.Il importe la bibliothèque mongoose, qui est utilisée pour interagir avec la base de données.
  Le schéma du motif est défini à l'aide de mongoose.Schema(même chose pour tout les modèles), il contient un champ *libellé* de type String qui est requis.
  Le modèle Motif est créé à partir du schéma défini et exporté en tant que module.

- praticien :

  Le schéma du praticien contient plusieurs champs tels que *nom, prenom, tel, email, rue, code_postal, ville,* et *visites*. Certains de ces champs sont requis (required: true), tandis que d'autres ne le sont pas. Le champ *visites* est un tableau d'identifiants d'objets de type *Visite*, référencés par leur ID.
  Le modèle Praticien est créé à partir du schéma défini et exporté en tant que module.

- visite :

  Le schéma de la visite contient plusieurs champs tels que *date_visite, commentaire, visiteur, praticien,* et *motif*. Certains de ces champs sont requis, tandis que d'autres ne le sont pas. Les champs *visiteur* et *praticien* sont des références à des documents dans d'autres collections de la base de données.
  Le modèle *Visite* est créé à partir du schéma défini et exporté en tant que module.

- visiteur :

  Le schéma du *visiteur* contient plusieurs champs tels que *nom, prenom, tel, email, password, date_embauche, visites* et *portefeuillePraticiens*. Certains de ces champs sont requis, tandis que d'autres ne le sont pas. Les champs *visites* et *portefeuillePraticiens* sont des tableaux d'identifiants d'objets de type *Visite* et *Praticien*, référencés par leur ID.
  Le modèle *Visiteur* est créé à partir du schéma défini et exporté en tant que module.

3. Routes

- motif :

  Ce fichier définit les routes CRUD pour les *motifs*. Il importe d'abord d'express et crée un routeur, puis il importe le contrôleur *motif*.
  Il définit ensuite des routes GET (*getAllMotifs, getOneMotif*), POST (*createMotif*), PUT (*modifyMotif*),DELETE (*deleteMotif*) pour les *motifs*.
  Enfin il exporte le routeur.

- praticien :

  Ce fichier définit les routes CRUD pour les *praticiens*. Il importe d'abord express et crée un routeur, puis il importe le contrôleur *praticien*. Ensuite, il définit des routes GET (*getAllPraticiens, getOnePraticien*), POST (*createPraticien*), PUT (*modifyPraticien*), DELETE (*deletePraticien*) pour les praticiens. Enfin, il exporte le routeur.

- user :

  Ce fichier définit les routes pour l'inscription et la connexion des utilisateurs. Il importe d'abord express et crée un routeur. Ensuite, il importe le contrôleur *user*. Il définit une route POST (/*signup*) pour l'inscription des utilisateurs et

une route POST (*//login*) pour la connexion des utilisateurs. Enfin, il exporte le routeur.

- visite :

  Ce fichier définit les routes CRUD pour les *visites*. Il importe d'abord express et crée un routeur, puis il importe le contrôleur *visite*. Ensuite, il définit des routes GET (*getAllVisites, getOneVisite*), POST (*createVisite*), PUT (*modifyVisite*), DELETE (*deleteVisite*) pour les visites. Enfin, il exporte le routeur.

- visiteur :

  Ce fichier définit les routes CRUD pour les *visiteurs*. Il importe d'abord express et crée un routeur, puis il importe le contrôleur *visiteur*. Ensuite, il définit des routes GET (*getAllVisiteurs, getOneVisiteu*r), PUT (*modifyVisiteur*), DELETE (*deleteVisiteur*) pour les visiteurs. De plus, il définit une route POST (*addPraticien*) pour ajouter un praticien à un visiteur spécifique. Enfin, il exporte le routeur.

4. App.js

Ce fichier permet de configurer :

- L'importation des modules express et mongoose.
- L'importation du middleware d'authentification(auth).
- L'importation des routes pour les motifs, les visites, les visiteurs, les praticiens et les utilisateurs.
- La connexion à la base de données MongoDB.
- L'utilisation de express.json() pour parser les données JSON des requêtes.
- La configuration des en-têtes CORS pour autoriser les requêtes depuis n'importe quelle origine.
- L'utilisation des routes avec les middlewares d'authentification pour sécuriser les endpoints.

- L’exportation de l'application express configurée pour être utilisée dans d'autres fichiers.

5. Server.js

Ce fichier permet de configurer :

- L'importation des modules http et app (notre application Express).
- La normalisation du port pour le serveur.
- La définition du port sur lequel l'application va écouter.
- La gestion des erreurs de démarrage du serveur.
- La création du serveur HTTP avec l'application Express.
- La gestion des erreurs et des événements du serveur.
- L'affichage du port sur lequel le serveur écoute lorsqu'il est démarré.

B. Android

1. Classes

- Praticiens :

  La classe Praticiens implémente l'interface Serializable de Java. Elle représente un praticien dans l'application et contient quatre attributs : nom, prenom, email et visites(liste des visites associées au praticien).
  Elle a un constructeur qui prend le nom, le prénom et l'email du praticien en paramètres et initialise les attributs correspondants. Il initialise également la liste des visites en tant que nouvelle ArrayList. Elle contient également des méthodes getter pour chaque attribut, qui renvoient la valeur de l'attribut correspondant.

- Visiteur :

La classe Visiteur est implémente l'interface Serializable de Java. Elle représente un visiteur (utilisateur) dans l'application. Elle contient plusieurs attributs
token (token d'authentification du visiteur), visiteurId, email, password, nom, prenom et portefeuillePraticiens(liste des praticiens associés au visiteur).
Elle a un constructeur qui prend l'email et le mot de passe du visiteur en paramètres et initialise les attributs correspondants. Il initialise également la liste des praticiens en tant que nouvelle ArrayList. Elle contient également des méthodes getter et setter pour chaque attribut, qui renvoient la valeur de l'attribut correspondant ou définissent la valeur de l'attribut correspondant.

- Visites :

  La classe Visites implémente l'interface Serializable de Java. Elle représente une visite dans l'application et contient plusieurs attributs : date_visite,commentaire, praticien, visiteur et motif.
  La classe a un constructeur qui prend la date de la visite, le commentaire, le praticien, le visiteur et le motif en paramètres et initialise les attributs correspondants. Elle contient également des méthodes getter et setter pour chaque attribut, qui renvoient la valeur de l'attribut correspondant ou définissent la valeur de l'attribut correspondant.

2. RecyclerViews

- PraticiensRecyclerTouchListener :

  Cette classe implémente l'interface *RecyclerView.OnItemTouchListener*, ce qui lui permet de détecter les interactions utilisateur sur un RecyclerView contenant des praticiens. Son but principal est de détecter les clics sur les éléments de la liste des praticiens et de déclencher une action en réponse à ces clics. Elle utilise un GestureDetector pour détecter les gestes. Lorsqu'un clic

est détecté sur un praticien, la méthode *onClick* est appelée sur l'interface *PraticiensRecyclerViewClickListener*, passant ainsi la vue cliquée et sa position dans la liste.

- PraticiensRecyclerViewAdapter :

  Cette classe, étend *RecyclerView.Adapter* et est utilisée pour lier les données de la liste des praticiens à une vue RecyclerView. Son but principal est de fournir une structure pour afficher les éléments de la liste des praticiens de manière efficace.
  Elle utilise une liste de praticiens pour stocker les données à afficher. Lors de la création d'un nouvel élément de vue (*onCreateViewHolder*), elle crée une vue à partir d'un fichier de mise en page XML spécifié et la lie à un objet *RecyclerViewHolder*. Lors de la liaison des données à la vue (*onBindViewHolder*), elle récupère les données du praticien à la position spécifiée dans la liste et les affiche dans les vues correspondantes. Enfin, elle indique le nombre total d'éléments dans la liste (*getItemCount*).

- PraticiensRecyclerViewClickListener :

  Cette interface définit un contrat pour gérer les événements de clic sur les éléments de la vue RecyclerView. Son but principal est de permettre la détection des clics sur les éléments de la liste des praticiens et de définir une action à effectuer en réponse à ces clics.
  Elle contient une méthode *onClick* qui prend en paramètres la vue cliquée et sa position dans la liste. Cette méthode sera implémentée par les classes qui souhaitent “écouter” les événements de clic sur les éléments de la liste des praticiens.

3. Instance

RetroFitClientInstance, fournit une instance de Retrofit, un client HTTP pour les requêtes réseau dans une application Android. Son objectif principal est de fournir une instance configurée de

Retrofit pour effectuer des appels réseau à l'URL de base spécifiée.
Elle utilise un modèle de conception Singleton pour garantir qu'une seule instance de Retrofit est créée pendant le cycle de vie de l'application. La méthode *getRetrofitInstance* retourne cette instance de Retrofit si elle existe déjà, sinon elle en crée une nouvelle en spécifiant l'URL de base et en configurant le convertisseur Gson pour convertir les réponses JSON en objets Java.

V. Tests fonctionnels

A. Authentification d'un utilisateur à l'application

Description :
Assurer que le système d'authentification fonctionne correctement en permettant l'accès uniquement aux utilisateurs avec des identifiants valides et en bloquant l'accès pour les utilisateurs avec des identifiants invalides.

Conditions préalables :

- L'application Android est accessible et fonctionnelle.
- L'API est accessible et opérationnelle.
- L'utilisateur dispose d'un compte enregistré dans la base de données.

Étapes :

1) Accéder à l'écran de connexion :
   Ouvrir l'application Android sur le périphérique.
   Saisir l'URL de l'API dans la barre d'adresse et lancer le serveur.
   Démarrer l'application Android sur un appareil.
   L'écran de connexion de l'application doit s'afficher.
2) Saisir les identifiants de connexion :

Dans les champs prévus à cet effet sur l'écran de connexion, saisir un email et un mot de passe valides associés à un compte enregistré dans la base de données.
Appuyer sur le bouton de connexion.

3) Authentification réussie :
   Si les identifiants sont valides, l'utilisateur doit être redirigé vers l'activité ListPraticiensActivity.
   L'utilisateur doit recevoir un objet Visiteur contenant ses informations personnelles.
4) Échec de l'authentification :
   Si les identifiants ne sont pas valides, l'utilisateur doit rester sur l'écran de connexion.
   L'utilisateur doit recevoir un message d'erreur indiquant que les identifiants sont incorrects.

B. Affichage de la liste des praticiens pour un visiteur connecté

Description :
Assurer que l'application Android peut afficher la liste des praticiens associés à un visiteur connecté. Les praticiens doivent être récupérés depuis l'API et affichés dans une liste sur l'écran de l'application.

Conditions préalables :

- L'application Android est accessible et fonctionnelle.
- L'API est accessible et opérationnelle.
- L'utilisateur est connecté en tant que visiteur.
- Le visiteur a au moins un praticien dans son portefeuille de praticiens

Étapes :

1) Récupération des praticiens :
   L'application Android doit envoyer une requête à l'API pour récupérer la liste des praticiens associés au visiteur connecté.
   L'API doit renvoyer la liste des praticiens liés au visiteur.
2) Affichage des praticiens dans une liste :

Les données des praticiens doivent être récupérées depuis l'API et stockées localement dans l'application Android.
Les informations des praticiens doivent être affichées dans une liste sur l'écran de l'activité.
Chaque praticien doit être représenté par son nom.

3) Gestion des cas d'erreur :
   Si aucun praticien n'est associé au visiteur connecté, l'application doit afficher un message indiquant qu'aucun praticien n'est disponible.
   En cas d'erreur de communication avec l'API ou de traitement des données, l'application doit afficher un message d'erreur approprié.
4) Interaction avec les praticiens :
   L'utilisateur doit pouvoir interagir avec les praticiens affichés dans la liste, en cliquant sur un praticien pour afficher ses détails.
5) Navigation :
   L'utilisateur doit pouvoir naviguer entre les écrans de l'application de manière fluide, en utilisant appuyant sur le bouton arrière ou sur un praticien.

C. Affichage des informations du praticien sélectionné

Description :
Permettre à l'utilisateur de visualiser les informations détaillées d'un praticien spécifique sur l'application Android. Les données du praticien doivent être récupérées depuis l'API et affichées sur l'écran de l'activité.

Conditions préalables :

- L'application Android est accessible et fonctionnelle.
- L'API est accessible et opérationnelle.
- L'utilisateur est connecté en tant que visiteur.

Étapes :

1) Récupération des informations du praticien :

L'application Android doit envoyer une requête à l'API pour récupérer les informations détaillées du praticien sélectionné.
L'API doit renvoyer les données du praticien spécifique identifié par son ID.

2) Affichage des informations du praticien :
   Les informations du praticien, telles que son nom, prénom et email doivent être récupérées depuis l'API et affichées sur l'écran de l'activité dédiée aux informations du praticien.
   Les données récupérées doivent être affichées dans les champs de texte appropriés sur l'écran.
3) Gestion des cas d'erreur :
   Si le praticien spécifié n'est pas trouvé dans la base de données, l'application doit afficher un message indiquant que le praticien n'a pas été trouvé.
   En cas d'erreur de communication avec l'API ou de traitement des données, l'application doit afficher un message d'erreur approprié.
4) Interaction avec les informations du praticien :
   L'utilisateur doit pouvoir visualiser les détails du praticien, tels que son nom, prénom, email et ses visites(pas finalisé)
5) Navigation :
   L'utilisateur doit pouvoir naviguer entre les écrans de l'application de manière fluide en allant en arrière, à la page précédente
6) Gestion des données :
   L'application doit être capable de gérer les données du praticien de manière appropriée, notamment en les affichant de manière lisible et compréhensible pour l'utilisateur.