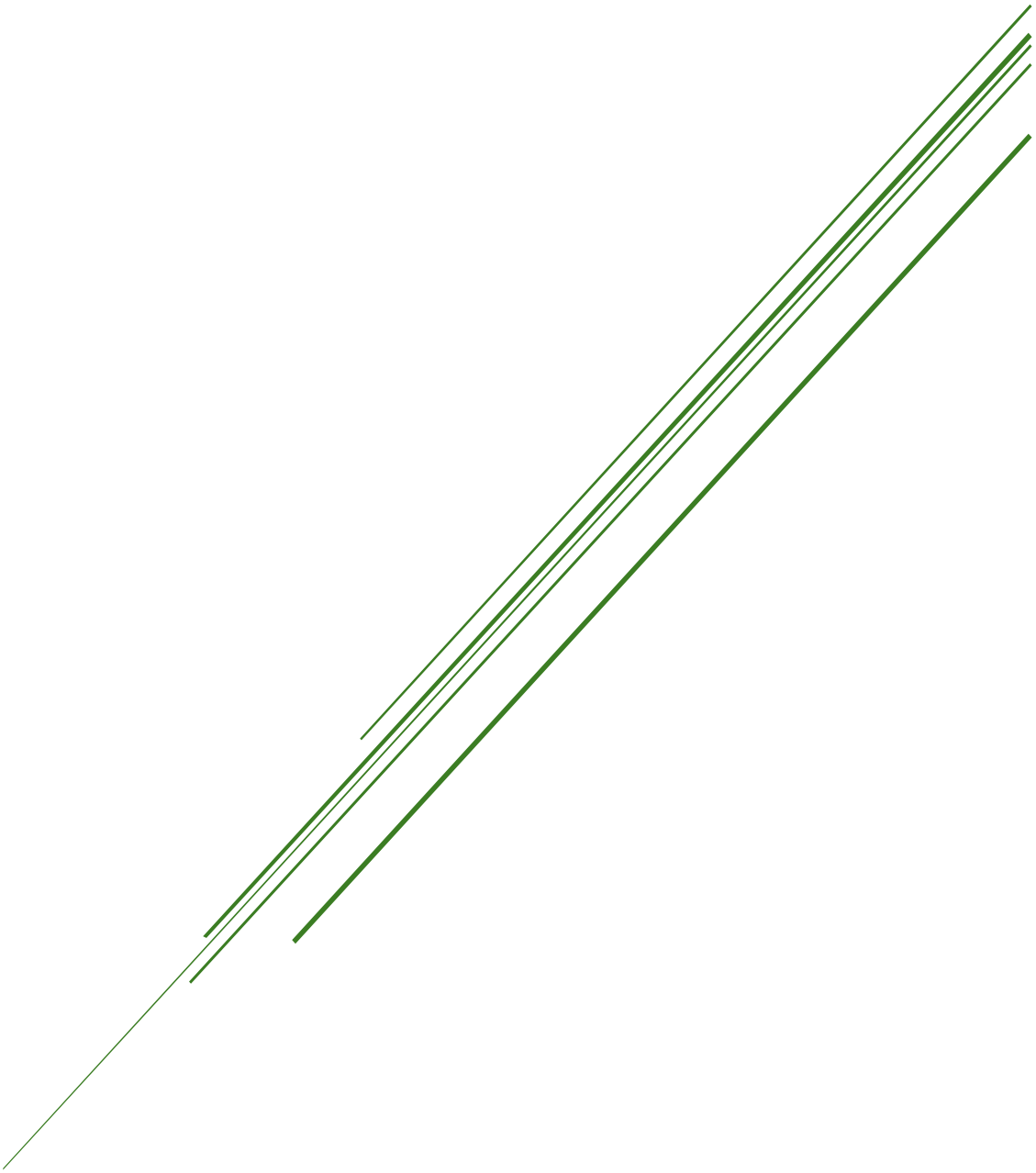
DOCUMENTATION TECHNIQUE

Na’el Benaïssa



ESP
2025

Table des matières

1. Introduction

1.1 Objectif de la documentation

Cette documentation technique a pour objectif de présenter en détail la conception, l'architecture, les choix technologiques et les étapes nécessaires à l'installation et à l'exécution du projet LeafGuard. Elle s'adresse principalement aux développeurs, aux techniciens et aux parties prenantes techniques souhaitant comprendre le fonctionnement interne de l'application, contribuer à son développement, ou la déployer sur d'autres environnements.

Dans le cadre de ce projet académique, cette documentation est également destinée au professeur encadrant, afin de faciliter la prise en main complète de l'application, son évaluation, et la vérification de son bon fonctionnement, aussi bien côté mobile que côté serveur IA.

1.2 Présentation globale du projet

LeafGuard est une application mobile dédiée à l'assistance pour le soin des plantes, à destination des amateurs de jardinage. Son objectif principal est d'offrir une solution simple, accessible et intelligente pour diagnostiquer les maladies végétales à partir d'une simple photo, tout en proposant des conseils personnalisés pour améliorer l'entretien des plantes.

L'application s'appuie sur une intelligence artificielle développée en Python à l'aide de TensorFlow et déployée via une API REST construite avec FastAPI, conteneurisée à l'aide de Docker et hébergée sur Google Cloud Platform. Sur le plan mobile, l'application est développée en Flutter pour garantir une compatibilité multiplateforme (Android/iOS).

Parmi les principales fonctionnalités développées dans cette première version :

- **Diagnostic d'images par IA** : reconnaissance automatisée de maladies sur les plantes (tomate, poivron, pomme de terre) à partir d'images.
- **Recommandations de soins** : suggestions contextuelles pour l'entretien (arrosage, fertilisation, traitements).
- **Rappels personnalisés** : système de notifications pour programmer les soins à effectuer.
- **Base de connaissances végétale** : accès à une documentation enrichie, alimentée notamment par l'API Trefle.
- **Interface fluide et intuitive** : onboarding, gestion des permissions, thème clair/sombre, animations personnalisées.

L'ensemble des données utilisateurs (authentification, stockage cloud) est géré à travers Supabase, qui centralise les fonctions de backend as a service (BaaS), notamment :

- Authentification sécurisée
- Stockage de l'historique de diagnostics
- Gestion de profils utilisateurs

L'objectif à long terme du projet est de pouvoir reconnaître un plus large panel de plantes et maladies, mais cette première version se concentre exclusivement sur trois cultures principales, afin de valider la faisabilité du système.

1.3 Stack technologique

Le projet repose sur une stack moderne et complète, composée des éléments suivants :

Composant	Technologie / Outil utilisé
Frontend mobile	Flutter (Dart SDK 3.5.3)
Backend IA	Python 3.9, TensorFlow, FastAPI, Uvicorn
API REST IA	FastAPI déployée via Docker
Base de données & auth	Supabase
API botanique externe	Trefle API
IDE principaux	Android Studio (Flutter), VS Code (Python/API)
Conteneurisation	Docker
Déploiement cloud	Google Cloud Platform (via Docker container)
Outils de test	mockito, mocktail, flutter_test, build_runner
Environnements d'exécution	Android Emulator, tablette Android

Cette stack permet un développement modulaire, une maintenance facilitée et une capacité d'évolution future grâce à des composants largement documentés et compatibles avec des pratiques DevOps modernes.

2. Architecture du projet

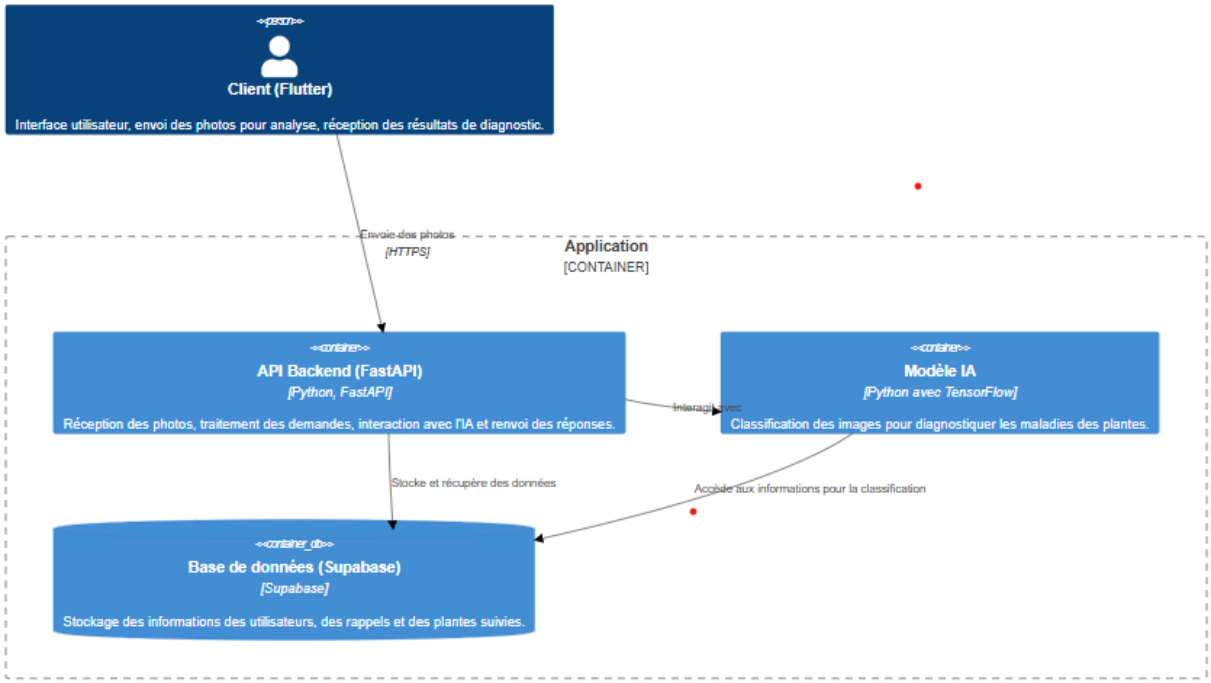
2.1 Vue d'ensemble

Le projet LeafGuard repose sur une architecture modulaire en trois couches principales :

- **Frontend mobile (Flutter)** : L'interface utilisateur de l'application développée avec Flutter. Elle permet aux utilisateurs de prendre des photos, recevoir un diagnostic, consulter des fiches de soins, programmer des rappels et gérer leurs plantes.
- **Backend IA (FastAPI + TensorFlow)** : Une API d'intelligence artificielle déployée sur Google Cloud, qui traite les images envoyées par les utilisateurs pour identifier les maladies des plantes grâce à un modèle de deep learning.
- **Base de données (Supabase)** : Supabase est utilisé comme backend as a service. Il gère l'authentification, le stockage des utilisateurs, des données liées aux plantes, des rappels, ainsi que des journaux d'activités.

Cette séparation claire entre les composants permet une meilleure maintenance, un déploiement indépendant des modules et une évolution progressive du projet.

2.2 Schéma d'architecture générale



2.3 Technologies utilisées

Composant	Technologie	Rôle principal
Mobile	Flutter (Dart)	Développement multiplateforme Android / iOS
Backend IA	FastAPI, TensorFlow	API de diagnostic via un modèle d'apprentissage profond
Authentification	Supabase Auth	Gestion des utilisateurs et sessions
Base de données	Supabase PostgreSQL	Stockage structuré des données utilisateurs et de contenu
Stockage distant	Supabase Storage	Sauvegarde des images (si activé)
Notifications	flutter_local_notifications + timezone	Rappels programmés dans l'application mobile
Interface	go_router, provider	Navigation dynamique et gestion d'état dans Flutter
Autres	Docker, Google Cloud	Conteneurisation et déploiement de l'API

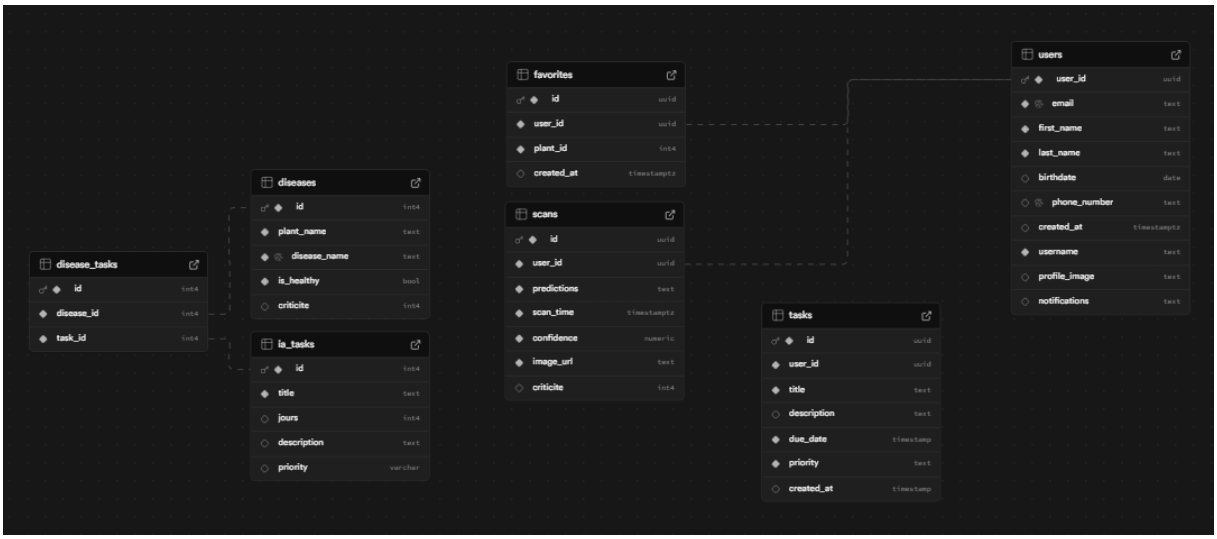
2.4 Communication entre les composants

- **1. Application mobile → API IA :**
  L'utilisateur capture une photo via l'application. Cette image est envoyée en POST au

backend IA via l'endpoint /predict/. La réponse contient le nom de la maladie et un taux de confiance.

- **2. Application mobile → Supabase** :
  Authentification des utilisateurs, enregistrement des journaux, stockage des rappels, récupération des fiches informatives, tout passe par des appels directs à Supabase via les SDK fournis.
- **3. API IA ← Docker ← Google Cloud** :
  L'API est conteneurisée avec Docker puis déployée sur une instance cloud (Cloud Run ou VM). Cela permet d'assurer une haute disponibilité et une facilité de mise à jour du modèle.

2.5 Schéma de la base de données Supabase

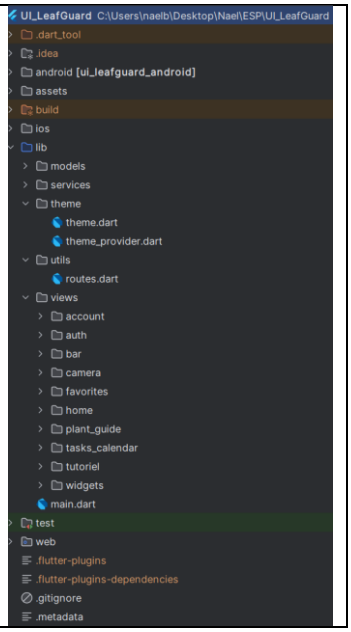


2.6 Structure des fichiers du projet Flutter

Le code source de l'application est organisé de manière modulaire pour assurer clarté et maintenabilité :

- **lib/models/** : Contient les modèles de données utilisés dans l'application.
- **lib/services/** : Regroupe les services métiers (API IA, Supabase, notifications...).
- **lib/theme/** : Gère le thème clair/sombre de l'application.
- **lib/utils/** : Contient les utilitaires, notamment la gestion centralisée des routes avec go_router.
- **lib/views/** : Organisé par fonctionnalités (authentification, caméra, guide, tâches, favoris...), chaque écran a son propre dossier.
- **lib/widgets/** : Composants réutilisables pour l'interface.
- **main.dart** : Point d'entrée de l'application, initialisation des services et lancement de l'UI.

Les ressources (images, SVG...) sont placées dans le dossier assets/, et les tests dans test/ avec mockito/mocktail.



Cette organisation facilite l'évolutivité, la navigation dans le code et la séparation des responsabilités.

3. Dépendances & Packages

3.1 Liste des packages utilisés

L’ensemble des dépendances utilisées dans le projet Flutter sont déclarées dans le fichier pubspec.yaml. Elles couvrent divers aspects du développement : navigation, services backend, UI, accès matériel, gestion de l’état et stockage.

```yaml
dependencies:
  flutter:
    sdk: flutter
  go_router: ^14.4.1
  supabase_flutter: ^2.7.0
  cupertino_icons: ^1.0.8
  http: ^1.3.0
  table_calendar: ^3.2.0
  sqflite: ^2.4.2
  path_provider: ^2.1.5
  path: ^1.9.1
  intl: ^0.20.2
  camera: ^0.11.1
  image_picker: ^1.1.2
  introduction_screen: ^3.1.17
  cached_network_image: ^3.4.1
  provider: ^6.1.2

  flutter_local_notifications: ^19.2.1
  timezone: ^0.10.1
  permission_handler: ^12.0.0+1

  shared_preferences: any
  flutter_svg: ^2.1.0
```

3.2 Justification des choix

Package	Rôle dans l'application
go_router	Gestion avancée et déclarative des routes de navigation
supabase_flutter	Intégration à Supabase (authentification, base de données, session)
http	Requêtes API (communication avec l'API LeafGuard et l'API Trefle)
sqflite, path_provider	Stockage local (favoris, rappels, historique)
path	Manipulation des chemins pour stockage local
camera, image_picker	Capture d'images et import depuis la galerie
cached_network_image	Chargement d'images avec mise en cache
table_calendar	Affichage des rappels dans une vue calendrier
flutter_local_notifications, timezone	Planification des notifications et gestion des fuseaux horaires
intl	Internationalisation, gestion des formats de date/heure
provider	Gestion d'état simple et performante
shared_preferences	Stockage de petites données (thème, préférences)
flutter_svg	Affichage d'icônes et illustrations vectorielles
introduction_screen	Présentation de l'application à l'ouverture (tutoriel)

3.3 Dépendances de développement et de test

Déclarées dans la section dev_dependencies, ces bibliothèques sont utilisées pour le testing, la génération de code et l'amélioration de la qualité du code :

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  mockito: ^5.4.6
  build_runner: ^2.4.15
  mocktail: ^1.0.4
  flutter_lints: ^4.0.0
  http: any

  flutter_launcher_icons: ^0.14.3
```

Package	Utilité
flutter_test	Tests unitaires et tests de widgets
mockito, mocktail	Mocking de services pour tests
build_runner	Génération automatique de code (JSON, modèles, etc.)
flutter_lints	Règles de style pour garantir un code propre
flutter_launcher_icons	Configuration des icônes de lancement pour Android/iOS

3.4 Mise à jour des dépendances

Il est conseillé de vérifier régulièrement les mises à jour disponibles pour chaque dépendance. Pour cela, deux commandes peuvent être utilisées :

- Pour mettre à jour automatiquement : flutter pub upgrade
- Pour voir les versions disponibles : flutter pub outdated

Ces mises à jour permettent de bénéficier des dernières fonctionnalités, corrections de bugs et améliorations de performance.

4. Navigation

4.1 Gestion via go_router

L'application Flutter utilise le package go_router pour la gestion de la navigation. Ce package permet une approche déclarative des routes, une meilleure organisation du code et une gestion efficace des redirections et des transitions de pages.

Le routeur est configuré dans une classe centralisée appelée Routes, qui expose une méthode statique routerConfiguration. Cette méthode initialise le GoRouter en tenant compte d'un paramètre showOnboarding pour déterminer la page initiale (écran d'accueil ou tutoriel).

4.2 Définition des routes

Voici les principales routes définies dans l'application, avec leur destination et les éventuelles restrictions d'accès :

Chemin	Page affichée	Accès conditionnel
/	Page d'accueil	Aucun
/onboarding	Tutoriel	Affiché au premier lancement
/calendar	Calendrier des tâches	Requiert une authentification
/camera	Caméra	Aucun
/favorites	Plantes favorites	Requiert une authentification
/plantsguide	Guide des plantes	Aucun

/account	Mon compte	Requiert une authentification
/auth	Page de connexion	Aucun

L'authentification est vérifiée via la méthode isUserAuthenticated(), qui utilise le client Supabase pour déterminer si une session active est en cours.

4.3 Redirections conditionnelles

Certaines routes sensibles (comme /calendar, /favorites, ou /account) sont protégées. Si un utilisateur tente d'y accéder sans être authentifié, il est automatiquement redirigé vers la page /auth.

```dart
/// Vérifie si un utilisateur est actuellement connecté via Supabase.
bool isUserAuthenticated() {
  final session = Supabase.instance.client.auth.currentSession;
  return session != null;
}
```

Cette vérification assure que seuls les utilisateurs connectés ont accès aux fonctionnalités sensibles ou personnalisées.

4.4 Transitions personnalisées

L'application utilise une classe NoTransitionPage pour désactiver les animations de transition entre les vues, afin d'offrir une navigation fluide et rapide :

```dart
/// Page personnalisée sans animation de transition pour le routage.
class NoTransitionPage extends CustomTransitionPage<void> {
  NoTransitionPage({required super.child}) : super(
    transitionsBuilder: (context, animation, secondaryAnimation, child) => child,
  );
```

Cela permet de garder une navigation cohérente, sans effets visuels distrayants.

4.5 Exemple de code – Déclaration des routes

```dart
GoRoute(
  path: '/calendar',
  builder: (context, state) => const TasksCalendarPage(),
  pageBuilder: (context, state) => NoTransitionPage(child: const TasksCalendarPage()),
  redirect: (context, state) => isUserAuthenticated() ? null : '/auth',
),  // GoRoute
```

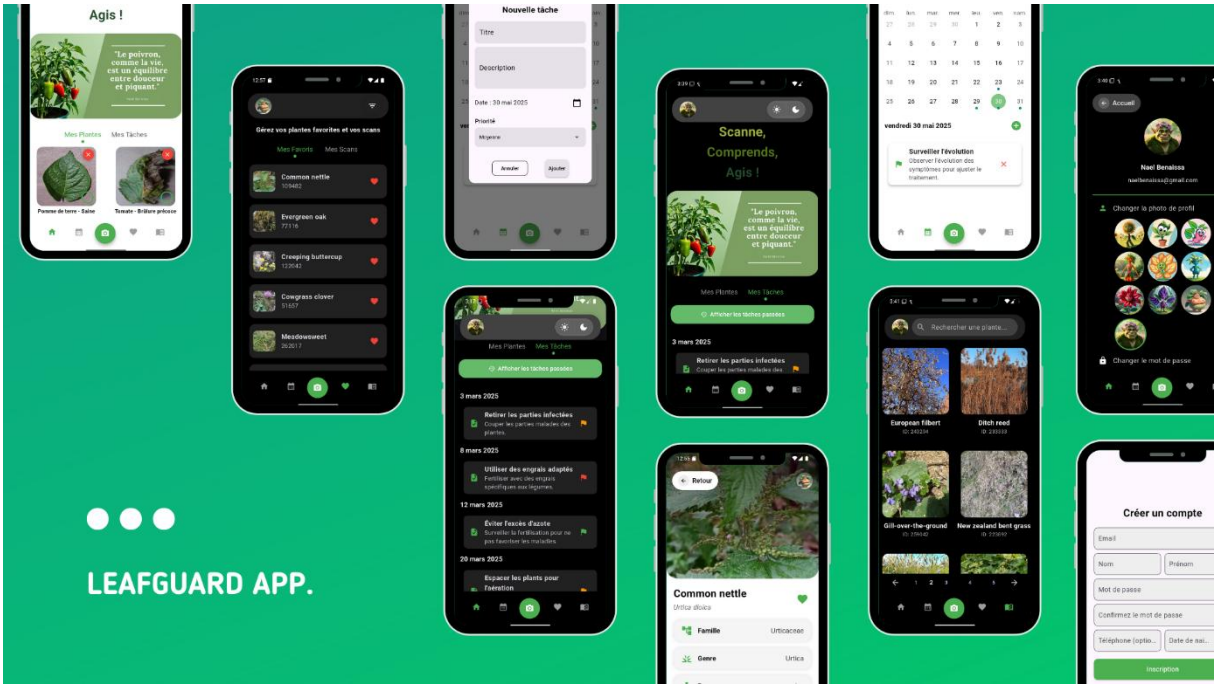
5. Interface utilisateur (UI)

L'interface de LeafGuard a été conçue pour offrir une expérience utilisateur fluide, intuitive et cohérente, adaptée aussi bien aux débutants qu'aux passionnés de jardinage. Elle intègre un design moderne, la prise en charge du thème sombre, et un affichage responsive.

5.1 Écrans principaux

L'application se compose de plusieurs écrans fonctionnels majeurs, chacun étant géré dans son propre dossier sous lib/views/ :

- **Accueil (Home)** : Vue synthétique des plantes suivies, rappels à venir, et accès rapide aux fonctionnalités.
- **Camera** : Permet de capturer une image pour analyse par l'IA, ou d'en sélectionner une depuis la galerie.
- **Calendrier** : Présente un aperçu chronologique des soins planifiés, des rappels et des événements liés aux plantes (arrosage, fertilisation, traitements). Il aide à anticiper les tâches à effectuer.
- **Guide de plantes** : Fournis des informations sur plus de 100 000 plantes via l'API Trefle.



5.2 Thème clair et sombre (theme.dart)

LeafGuard propose un système de thème dynamique, avec alternance entre mode clair et sombre, basé sur les préférences de l'utilisateur.

Les thèmes sont définis dans lib/theme/theme.dart. Chaque aspect visuel est centralisé : couleurs principales, AppBar, navigation, texte, boutons, cartes...

```
// Style des cartes (Cards)
static CardTheme _cardTheme(Color color) {
  return CardTheme(
    elevation: 4,
    shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(15)),
    margin: const EdgeInsets.symmetric(vertical: 8, horizontal: 10),
    color: color,
  );  // CardTheme
}
```

L’adoption de ThemeData.light() et ThemeData.dark().copyWith() permet une personnalisation fine, tout en assurant la compatibilité avec les composants Flutter natifs.

5.3 Responsive design

L'application est entièrement responsive, s’adaptant automatiquement aux différentes tailles et orientations d’écrans (smartphones, tablettes...).

Cela est géré à l’aide de :

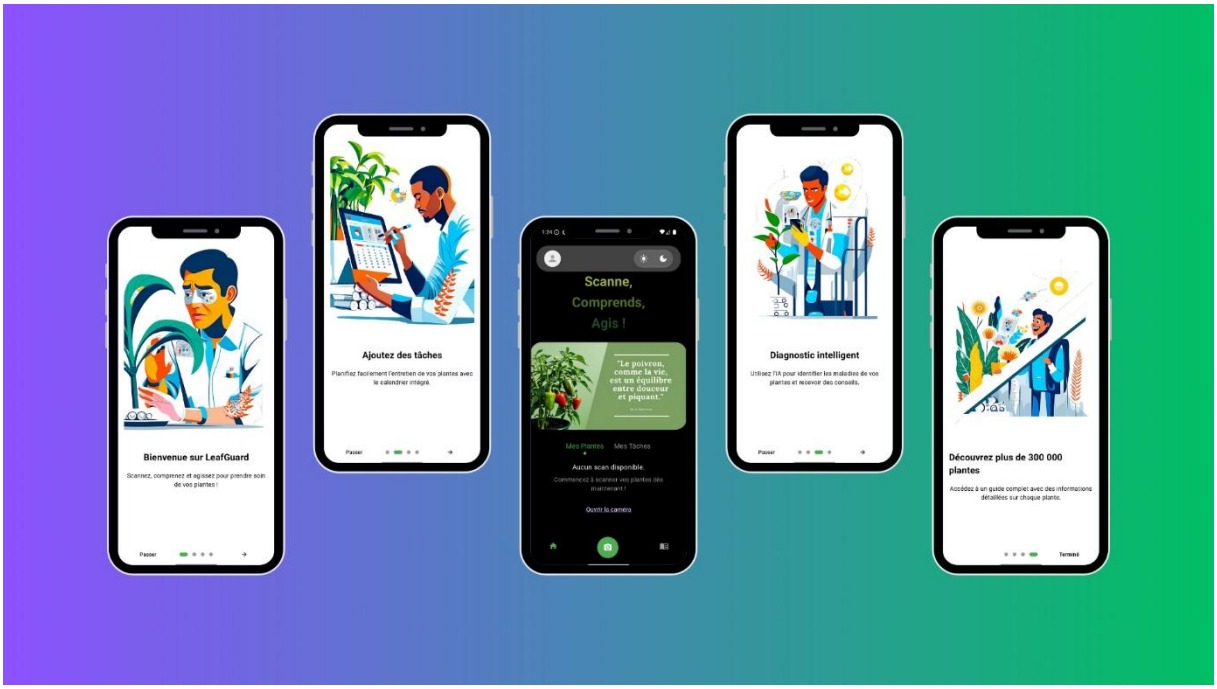
- MediaQuery pour adapter dynamiquement les marges, tailles de texte, paddings.
- LayoutBuilder pour afficher des structures conditionnelles selon la largeur disponible.
- Flexible, Expanded, Wrap, etc. pour organiser les composants de façon fluide.

```
// Calcul de la hauteur totale de l'appbar (barre d'état + toolbar + marge personnalisée)
double appBarHeight = MediaQuery.of(context).padding.top + kToolbarHeight + 23;
```

Ce comportement responsive est testé sur différents formats, garantissant une ergonomie optimale quel que soit le support.

5.4 Onboarding (introduction_screen)

L’onboarding, s’intègre naturellement à l’UI globale. Il utilise la librairie introduction_screen avec un design illustré, des transitions animées, et une navigation intuitive.



6. Fonctionnalités principales

L'application LeafGuard regroupe plusieurs fonctionnalités essentielles centrées sur l'assistance intelligente au soin des plantes. Ces fonctionnalités reposent sur une intégration fluide entre les services Flutter, Supabase, Trefle API et les composants natifs du mobile.

6.1 Authentification avec Supabase

L'application intègre un système d'authentification sécurisé via Supabase, permettant à chaque utilisateur d'avoir un espace personnel pour ses plantes, historiques de diagnostics, et rappels.

- Validation côté client : champs obligatoires, format d'e-mail, mot de passe sécurisé.
- Stockage sécurisé des informations sur Supabase.
- Redirection automatique après connexion.

```dart
/// Vérifie si le mot de passe respecte les critères de sécurité
bool _isPasswordValid(String password) {
  final regex = RegExp(r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[!@#\$&*~%?^+=.,:;_\-])[A-Za-z\d!@#\$&*~%?^+=.,:;_\-]{8,15}$');
  return regex.hasMatch(password);
}

/// Calcule l'âge à partir de la date de naissance
int _calculateAge(DateTime birthDate) {
  final today = DateTime.now();
  int age = today.year - birthDate.year;
  if (today.month < birthDate.month || (today.month == birthDate.month && today.day < birthDate.day)) {
    age--;
  }
  return age;
}

/// Affiche un message d'erreur sous forme de Snackbar rouge
void _showSnackbar(String message) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text(message), backgroundColor: Colors.red, duration: const Duration(seconds: 2)),
  );
}
```

6.2 Diagnostic via caméra ou galerie

L'utilisateur peut effectuer un diagnostic végétal en prenant une photo avec l'appareil photo ou en sélectionnant une image depuis la galerie.

- Utilisation du package image_picker.
- Redirection automatique vers l'écran de pré-analyse.
- Transmission de l'image à l'API IA pour prédiction.

```dart
/// Envoie une image à l'API d'IA pour prédire la maladie de la plante.
/// Retourne un Map contenant les résultats décodés en JSON.
/// Lance une exception en cas d'erreur réseau ou code HTTP != 200.
Future<Map<String, dynamic>> predictDisease(File imageFile) async {
  try {
    var request = http.MultipartRequest("POST", Uri.parse(_predictUrl));

    // Ajout du fichier image dans la requête multipart/form-data
    request.files.add(await http.MultipartFile.fromPath('file', imageFile.path));

    // Envoi de la requête
    var streamedResponse = await client.send(request);

    // Vérification du succès de la requête
    if (streamedResponse.statusCode == 200) {
      var responseData = await streamedResponse.stream.bytesToString();
      return jsonDecode(responseData);
    } else {
      throw Exception("Erreur lors de la prédiction : ${streamedResponse.statusCode}");
    }
  } catch (e) {
    // Gestion des erreurs réseau ou exceptions inattendues
    throw Exception("Erreur de connexion à l'IA : $e");
  }
}
```

6.3 Stockage des résultats distant

Chaque diagnostic est sauvegardé sur Supabase avec l'image, la confiance, la criticite, et la maladie détectée. L'historique est ensuite consultable via LA PAGE Home et la page camera.

- Requête d'écriture après chaque prédiction réussie.
- Affichage ordonné dans l'historique utilisateur.

id uuid	user_id uuid	predictions text	scan_time timestamptz	confiden... nume...	image_url text	criticite int4
1809f5df-7a94-4062-8a32-2d9c10c76837	f4dfda0f-231e-4f27-827d-e46444...	Tomate Saine	2025-02-14 00:22:09.240963-	0.95	https://xweiounkhqtchlapjazt.supabase.c	1
27d78bdc-a2d5-489f-b796-21923dd014f9	4ff0e55b-234d-4433-b8d5-4b0e...	Tomate - Brûlure précoce	2025-05-28 23:36:01.211439+(	0.93	https://xweiounkhqtchlapjazt.supabase.c	2
2ef22411-1ee3-4ad5-a376-8cb0a436f264	f4dfda0f-231e-4f27-827d-e46444...	Pomme de terre - Saine	2025-03-28 13:49:36.646114+(	0.71	https://xweiounkhqtchlapjazt.supabase.c	3
3974fe22-357b-44da-8199-8badd39acece	f4dfda0f-231e-4f27-827d-e46444...	Poivron avec tâche bactérienne	2025-02-14 00:21:53.78416+0(	0.95	https://xweiounkhqtchlapjazt.supabase.c	4
4745f651-b965-45cf-ade3-43085502db8(	4ff0e55b-234d-4433-b8d5-4b0e...	Pomme de terre - Saine	2025-05-28 23:35:40.672863-	0.71	https://xweiounkhqtchlapjazt.supabase.c	0
53dba6e5-5b06-47c5-828b-9815ef3063c	dd5be52f-6c00-41e8-8eeb-7c74c...	Tomate - Brûlure tardive	2025-05-27 13:58:44.300454(	0.29	https://xweiounkhqtchlapjazt.supabase.c	3

6.4 Calendrier de soins intelligent (table_calendar)

Le calendrier permet de créer manuellement des tâches de soin (arrosage, traitement, etc.), mais aussi d'ajouter automatiquement des tâches recommandées par l'IA après un diagnostic.

- Interface construite avec table_calendar.
- Ajout de tâches personnalisées ou guidées.
- Notifications planifiées à 9h le jour de la tâche.

id int4	title text	jours int4	description text	priority varchar
7	Éliminer les feuilles infectées	3	Enlever les parties malades pour éviter la	high
8	Appliquer un traitement au cuivre	7	Utiliser un traitement à base de cuivre pou	medium
9	Surveiller l'évolution	1	Observer l'évolution des symptômes pour	low
10	Retirer les parties infectées	2	Retirer les parties malades pour limiter l'ir	high

6.5 Notifications locales (flutter_local_notifications)

L'application utilise des notifications locales pour rappeler les tâches planifiées dans le calendrier. Chaque tâche déclenche une alerte à 9h du matin, le jour prévu.

- Notifications personnalisées avec titre et contenu.
- Compatibilité Android/iOS.

```dart
// Planifie une notification de test après 3 secondes
final scheduledDate = DateTime.now().add(const Duration(seconds: 3));
NotificationService().scheduleNotificationForTask(
  id: 0,
  title: 'Test Notification',
  body: 'Cette notification est déclenchée après 3 secondes.',
  date: scheduledDate,
);
```

6.6 Recherche botanique via Trefle API

L'écran Guide de plantes permet de rechercher des fiches d'informations botaniques détaillées, en s'appuyant sur les données fournies par l'API Trefle :

- Recherche par nom de plante.
- Affichage du nom scientifique, famille, entretien, maladies courantes.

```dart
/// Récupère une liste de plantes, avec pagination et possibilité de recherche par nom.
/// Retourne une Map contenant une liste de données et un total.
Future<Map<String, dynamic>> fetchPlants({int page = 1, String query = ""}) async {
  final url = query.isEmpty
      ? "$_baseUrl?token=$_apiKey&page=$page"
      : "$_baseUrl/search?token=$_apiKey&q=${Uri.encodeComponent(query)}";

  return _getPlantsData(url);
}
```

6.7 Système de permissions (permission_handler)

L'application gère les permissions critiques de manière centralisée grâce au package permission_handler.

- Appareil photo
- Stockage / Galerie
- Notifications

Le système vérifie les autorisations à l'ouverture des écrans concernés, avec affichage de dialogues explicatifs en cas de refus.

7. Backend IA – FastAPI

Le backend de l'application LeafGuard repose sur une API REST développée avec FastAPI, qui permet l'analyse d'images envoyées par les utilisateurs afin de diagnostiquer automatiquement les maladies affectant certaines plantes (tomate, poivron, pomme de terre), grâce à un modèle de deep learning construit avec TensorFlow.

7.1 Structure du projet Python

Le projet est organisé selon une architecture simple et modulaire :

```
IA_LeafGuard/
├── model/
│   └── modele_maladies_plantes.h5   # Modèle pré-entraîné
├── PlantVillage/                    # Dataset d'entraînement
├── app.py                           # Script principal (point d'entrée de l'API)
├── IA_LeafGuard.ipynb               # Notebook de génération/évaluation du modèle
├── requirements.txt                 # Dépendances Python
├── Dockerfile                       # Configuration Docker
```

7.2 Description du modèle IA (TensorFlow)

Le cœur du système repose sur un modèle CNN entraîné via TensorFlow, à partir du dataset open-source PlantVillage. Ce modèle est sauvegardé au format .h5 et chargé dynamiquement lors du lancement de l'API.

- Architecture : Convolutional Neural Network (CNN)
- Classes détectées :
  - Tomate
  - Pomme de terre
  - Poivron
- Format d'entrée : image RGB redimensionnée à 224x224

Accéder au notebook d'entraînement du modèle.

7.3 Point d'entrée – app.py

Le fichier app.py constitue le point d'entrée principal de l'API. Il initialise le modèle à partir du fichier .h5 et expose un endpoint unique /predict/.

- Chargement unique du modèle à l'initialisation
- Traitement d'image via Pillow et NumPy
- Réponse retournée : nom de la maladie + taux de confiance

```
# Chargement du modèle
model = tf.keras.models.load_model("model/modele_maladies_plantes.h5")
```

7.4 Endpoint /predict/ : fonctionnement

L'API expose un unique endpoint POST : /predict/, conçu pour recevoir une image et retourner un diagnostic.

- Méthode : POST
- Paramètre requis : file (image au format JPG/PNG)
- Type : multipart/form-data
- Réponse : objet JSON avec nom de la maladie et taux de confiance

```
# Endpoint API pour l'analyse d'image
@app.post("/predict/")
async def predict_disease(file: UploadFile = File(...)):
    image_bytes = await file.read()
    result = predict(image_bytes)
    return result
```

7.5 Exemple d'appel via cURL + réponse JSON

Ci-dessous un exemple de requête HTTP avec cURL pour tester localement l'API :

```
curl -X 'POST' \
  'http://127.0.0.1:8080/predict/' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@path_to_image.jpg'
```

Réponse JSON type :

```
{
  "maladies": "Tomato - Mosaic Virus",
  "confiance": 0.95
}
```

7.6 Déploiement du backend

Deux méthodes de déploiement sont supportées :

- **Local avec Uvicorn** : uvicorn app:app --host 0.0.0.0 --port 8080
- **Conteneurisé avec Docker** : docker build -t ia_leafguard .

  docker run -p 8080:8080 ia_leafguard

8. Déploiement de l'API IA

L'API de reconnaissance de maladies végétales développée avec FastAPI est conteneurisée à l'aide de Docker, puis déployée sur Google Cloud Platform (GCP). Ce choix permet une portabilité, une scalabilité et une simplicité de gestion adaptées aux exigences du projet.

8.1 Construction d'un conteneur Docker

L'environnement complet (FastAPI, modèle TensorFlow, dépendances) est encapsulé dans un conteneur Docker à l'aide d'un Dockerfile défini à la racine du projet. Ce conteneur permet d'exécuter l'API de manière identique sur tout environnement compatible avec Docker.

8.2 Déploiement sur Google Cloud

Durant le processus de développement et de mise en production de l'API IA LeafGuard, plusieurs commandes Docker ont été utilisées pour assurer la construction, le test local, la gestion des images, ainsi que leur envoi vers un registre distant avant déploiement.

Tout d'abord, l'image Docker de l'application a été construite localement à partir du fichier Dockerfile. Cette image contient le code source de l'API FastAPI, le modèle TensorFlow pré-

entraîné, ainsi que toutes les dépendances nécessaires à l'exécution de l'API. Grâce à cette encapsulation, l'environnement est parfaitement maîtrisé et reproductible.

Une fois l'image construite, elle a été testée localement avec Docker pour s'assurer que l'API fonctionne correctement avant tout déploiement. L'exécution locale permet également un débogage rapide et une itération rapide en cas de problème.

Ensuite, l'image a été taguée et poussée vers Google Artifact Registry, le registre de conteneurs proposé par Google Cloud. Cette étape rend l'image accessible à Google Cloud Run pour un déploiement sans serveur.

8.3 Commandes Docker utilisées

Voici les principales commandes Docker employées lors du développement et du déploiement :

Commande	Description
docker build -t ia_leafguard .	Construction de l'image locale
docker run -p 8080:8080 ia_leafguard	Exécution locale de l'API
docker tag ia_leafguard gcr.io/...	Attribution du tag pour GCP
docker push gcr.io/...	Envoi de l'image sur Google Artifact Registry

8.4 Avantages de Docker pour ce projet

L'utilisation de Docker présente de nombreux bénéfices dans le cadre de LeafGuard :

- **Portabilité** : L'API peut être exécutée sur n'importe quel système compatible Docker.
- **Reproductibilité** : Le conteneur garantit une exécution stable et identique entre les environnements (local, cloud...).
- **Facilité de déploiement** : Intégration fluide avec Google Cloud Run pour un déploiement sans serveur.
- **Isolation** : Le modèle, les dépendances Python et l'API sont encapsulés, réduisant les conflits système.

9. Tests

9.1 Types de tests : unitaires, mocks

Dans LeafGuard, deux grandes catégories de tests sont utilisées :

- **Tests unitaires** : Ils vérifient individuellement le comportement de fonctions ou méthodes, sans dépendances externes. Par exemple, le service IaLeafguardService est testé pour garantir que la méthode predictDisease() retourne les bons résultats à partir de réponses simulées.
- **Mocks** : Utilisés pour simuler des dépendances comme le client HTTP ou le plugin de notifications. Cela permet de tester des composants de manière isolée en contrôlant le comportement des objets externes.

9.2 Packages utilisés

Pour la mise en place des tests, deux bibliothèques Dart sont employées :

- **mockito** : Utilisé principalement pour générer des mocks complexes comme FlutterLocalNotificationsPlugin.
- **mocktail** : Plus flexible pour Dart >= 2.12, utilisé pour mocker des services comme http.Client dans les tests réseau.

9.3 Exemple de test unitaire

Extrait d'un test unitaire sur la méthode predictDisease() :

```dart
test('predictDisease retourne une map avec la réponse de l\'IA', () async {
  // Préparation d'une fausse image locale pour la prédiction
  final fakeImage = File('assets/img/plantes/nest_fern.png');

  // Simulation d'une réponse HTTP réussie avec un corps JSON encodé
  final mockResponse = MockStreamedResponse();
  final responseBody = jsonEncode({"disease": "blight", "confidence": 0.85});

  // Mock du code HTTP 200 OK
  when(() => mockResponse.statusCode).thenReturn(200);
  // Mock du corps de la réponse sous forme de stream de bytes
  when(() => mockResponse.stream)
      .thenAnswer((_) => http.ByteStream.fromBytes(utf8.encode(responseBody)));

  // Mock de l'envoi de la requête HTTP qui renvoie la réponse simulée
  when(() => mockHttpClient.send(any())).thenAnswer((_) async => mockResponse);

  // Appel de la méthode testée
  final result = await service.predictDisease(fakeImage);

  // Vérification que le résultat est bien une Map et contient les données attendues
  expect(result, isA<Map<String, dynamic>>());
  expect(result['disease'], 'blight');
  expect(result['confidence'], 0.85);

  // Vérification que la méthode send du client HTTP a été appelée exactement une fois
  verify(() => mockHttpClient.send(any())).called(1);
});
```

Cet exemple démontre l'utilisation de mocktail pour simuler une réponse HTTP et tester le traitement effectué par le service.

9.4 Bonnes pratiques de test

- **Isoler les composants testés** : Utiliser des mocks pour toutes les dépendances externes.

- **Nommer clairement les tests** : Le nom du test doit décrire précisément le comportement attendu.
- **Tester les cas d'erreur** : Simuler des erreurs réseau, des réponses HTTP invalides, etc.
- **Utiliser setUp() et tearDown()** pour initialiser ou nettoyer les ressources entre les tests.
- **Respecter la couverture fonctionnelle** : Chaque scénario critique de l'application (diagnostic, notifications, authentification) doit être couvert.

10. Configuration des plateformes

10.1 Android – AndroidManifest.xml

Le fichier AndroidManifest.xml a été configuré pour gérer les autorisations nécessaires au bon fonctionnement de l'application :

Permissions utilisées

```
<uses-permission android:name="android.permission.POST_NOTIFICATIONS" />
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_MEDIA_IMAGES" />
<uses-permission android:name="android.permission.SCHEDULE_EXACT_ALARM" />
<uses-permission android:name="android.permission.INTERNET"/>
```

Configuration pour les notifications

```
<!-- Notifications : Receiver pour redémarrage de l'appareil -->
<receiver
    android:name="com.dexterous.flutterlocalnotifications.ScheduledNotificationBootReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <action android:name="android.intent.action.MY_PACKAGE_REPLACED" />
    </intent-filter>
</receiver>

<!-- Notifications : Receiver pour alarmes planifiées -->
<receiver
    android:name="com.dexterous.flutterlocalnotifications.ScheduledNotificationReceiver"
    android:exported="false" />
```

Configuration matérielle

```
<meta-data
    android:name="android.hardware.camera"
    android:value="true" />
<meta-data
    android:name="android.hardware.camera.autofocus"
    android:value="true" />
```

Cette configuration permet de gérer l'appareil photo, les notifications programmées, ainsi que le redémarrage des alarmes après redémarrage du téléphone.

10.2 iOS – Info.plist

Le fichier Info.plist contient les métadonnées essentielles pour iOS :

Clés importantes

```
<key>CFBundleDisplayName</key>
<string>Ui Leafguard</string>
<key>CFBundleName</key>
<string>LeafGuard</string>
<key>UILaunchStoryboardName</key>
<string>LaunchScreen</string>
<key>UIMainStoryboardFile</key>
<string>Main</string>
<key>UISupportedInterfaceOrientations</key>
<array>
    <string>UIInterfaceOrientationPortrait</string>
    <string>UIInterfaceOrientationLandscapeLeft</string>
    <string>UIInterfaceOrientationLandscapeRight</string>
</array>
```

Compatibilité et performances

```
<key>LSRequiresIPhoneOS</key>
<true/>
<key>CADisableMinimumFrameDurationOnPhone</key>
<true/>
<key>UIApplicationSupportsIndirectInputEvents</key>
<true/>
```

Cette configuration assure la compatibilité avec les différentes orientations écran et permet une meilleure gestion des performances pour les animations ou les notifications.

11. Sécurité

11.1 Sécurité côté mobile

Les données sensibles, notamment les tokens d'authentification, sont stockées localement à l'aide de solutions simples comme SharedPreferences. Bien que ces mécanismes ne chiffrent pas les données par défaut, l'application évite de stocker des informations critiques ou

personnelles en clair. Les tokens ont une durée de vie limitée et peuvent être automatiquement invalidés côté serveur.

Toutes les communications réseau avec l'API s'effectuent via le protocole HTTPS, garantissant un canal sécurisé contre l'interception et les attaques de type Man-in-the-Middle. L'intégration avec des bibliothèques HTTP permet de gérer les en-têtes d'authentification, les expirations de token et le rafraîchissement des sessions de manière centralisée.

La gestion des permissions (caméra, notifications, stockage) est encadrée par une demande d'autorisation explicite à l'utilisateur. Aucun accès aux ressources sensibles n'est possible sans consentement. En cas de refus, des traitements adaptés sont prévus pour éviter les blocages ou les crashs.

11.2 Sécurité côté API

L'API est hébergée sur Supabase, une plateforme sécurisée qui offre une gestion fine des accès aux données via des règles de sécurité au niveau de la base RLS. Ces règles permettent de restreindre l'accès aux données en fonction de l'utilisateur authentifié, garantissant que chaque requête ne peut accéder qu'aux informations qui lui sont autorisées.

L'authentification est gérée via des tokens JWT émis par Supabase, contenant les informations nécessaires pour valider l'identité et les permissions. Chaque token est signé et vérifié sur le serveur avant l'exécution des requêtes. Les routes API sensibles sont protégées par des middlewares qui appliquent des contrôles d'accès selon le rôle et les droits de l'utilisateur, implémentant une politique RBAC (Role-Based Access Control).

Le backend inclut également des mécanismes de validation stricte des données envoyées par les formulaires. Cela garantit que seules des données correctement formatées et attendues sont traitées, limitant ainsi les risques d'injection ou de corruption. Les entrées utilisateur sont systématiquement nettoyées et validées avant tout traitement.

Enfin, des protections contre les attaques courantes telles que l'injection SQL, CSRF, et XSS sont en place, assurant la robustesse de l'application côté serveur. Un système de limitation de requêtes (rate limiting) est également configuré pour éviter les abus et les attaques par déni de service.

11.3 Permissions et accès restreints

L'application demande uniquement les permissions nécessaires à son fonctionnement, telles que :

- Accès à la caméra (pour la capture de contenu)
- Notifications (pour les rappels et alertes)
- Accès aux fichiers multimédias ou à l'espace de stockage (lecture/écriture de médias)

Ces permissions sont définies dans AndroidManifest.xml (Android) et Info.plist (iOS), et sont activées uniquement après validation par l'utilisateur à l'exécution. Aucun composant critique de l'application n'est accessible par défaut sans accord explicite.

12. Limites & Évolutions possibles

12.1 Fonctionnalités non encore implémentées

Certaines fonctionnalités envisagées ne sont pas encore intégrées dans la version actuelle de l’application. Parmi celles-ci :

- Gestion avancée des profils utilisateurs, incluant des options de personnalisation et de préférences détaillées.
- Support multi-langue complet pour améliorer l’accessibilité à un public international.
- Fonctionnalités collaboratives telles que le partage et la co-édition de diagnostic en temps réel.
- Notifications enrichies avec actions interactives directement depuis la notification (ex : répondre, marquer comme lu).
- Support étendu des périphériques et plateformes, notamment une version web ou desktop.

12.2 Optimisations futures possibles

L’architecture actuelle laisse la place à plusieurs pistes d’amélioration et d’optimisation :

- Amélioration des performances via la mise en place de cache local plus sophistiqué et optimisation des requêtes réseau.
- Renforcement de la sécurité notamment par l’intégration de stockage sécurisé des tokens.
- Optimisation de l’UX/UI en affinant les animations, transitions, et en adaptant mieux l’application aux différentes tailles d’écran.
- Automatisation accrue des tests, incluant tests d’intégration et tests end-to-end pour garantir la stabilité lors de nouvelles mises à jour.
- Monitoring et reporting avec intégration d’outils d’analyse et de remontée d’erreurs pour un suivi en temps réel.

12.3 Perspectives d’évolution

Plusieurs axes d’évolution peuvent être envisagés pour enrichir l’application et répondre aux attentes futures des utilisateurs :

- Partage des scans entre utilisateurs : Permettre aux utilisateurs de partager leurs analyses et résultats de scans avec d’autres membres, favorisant ainsi l’échange d’informations et la création d’une communauté active autour de l’application.
- Élargissement des plantes détectées par l’IA : Améliorer la base de données et les modèles d’intelligence artificielle pour reconnaître un nombre plus important de plantes, incluant des variétés régionales ou moins courantes, afin d’augmenter la pertinence et la couverture des diagnostics.

- Intégration de recommandations personnalisées : Développer un module qui, à partir des scans et des préférences utilisateur, propose des conseils personnalisés sur l’entretien des plantes, la prévention des maladies ou les meilleures pratiques de jardinage.

13. Annexes

Références des API utilisées

Trefle API

La Trefle API est une base de données botanique en ligne offrant un accès à un vaste catalogue d’informations sur les plantes. Elle permet de récupérer des données détaillées telles que les descriptions botaniques, les images, les conditions de croissance, ainsi que les classifications scientifiques.

- **Site officiel :** https://trefle.io
- **Utilisation dans le projet :** Consultation des données botaniques.

Supabase

Supabase est une plateforme open-source backend-as-a-service qui fournit une base de données relationnelle PostgreSQL, des authentifications, du stockage et des fonctions serverless. Elle est utilisée pour héberger et sécuriser les données utilisateur, gérer l’authentification, et assurer la synchronisation en temps réel.

- **Site officiel :** https://supabase.com
- **Utilisation dans le projet :** Gestion sécurisée des utilisateurs, stockage, et synchronisation des données entre appareils.