Guide de Développement

Application Mobile

GeoNDGR-Collecte

Réalisé par :

Dahdouh Ayoub

Moutamanni Abdourahman

Encadré par :

El Imami Ayoub

ETAFAT

NDGR (Direction Nationale du Génie Rural)

2025

Table des matières

Chapitre 1

Introduction

1.1 Objectif du document

Ce document constitue le **guide de développement officiel** de l'application mobile **GeoNDGR-Collecte**. Il est destiné aux développeurs, chefs de projet et techniciens impliqués dans la conception, le développement, la maintenance et l'évolution de l'application.

Information

Ce guide couvre l'ensemble du cycle de vie du développement, depuis la configuration de l'environnement jusqu'au déploiement en production sur un serveur Windows avec `NGINX` et `Gunicorn`.

Le présent document décrit de manière exhaustive :

- ✓ L'architecture technique globale du système
- ✓ Les technologies et outils utilisés
- ✓ La structure et l'organisation du code source
- ✓ Les procédures d'installation et de configuration
- ✓ Les mécanismes de communication API
- ✓ Le déploiement serveur (NGINX + Gunicorn sur Windows)
- ✓ Les bonnes pratiques de développement et de maintenance

1.2 Présentation du projet

1.2.1 Contexte

Le projet **GeoNDGR-Collecte** s'inscrit dans le cadre du **Projet de désenclavement des zones de production pisci-rizicole** en Basse Guinée et en Guinée Forestière. Ce programme est financé par l'AFD (Agence Française de Développement) et mis en œuvre par la **Direction Nationale du Génie Rural (NDGR)**.

1.2.2 Objectifs de l'application

L'objectif principal est de doter les équipes de terrain d'un outil mobile performant permettant de :

1. **Collecter** des données géoréférencées sur les pistes, chaussées, ouvrages et infrastructures rurales

2. **Synchroniser** les données avec une base centrale via une API REST sécurisée
3. **Visualiser** et valider les données collectées directement sur le terrain
4. **Travailler hors-ligne** avec synchronisation différée lors du retour en zone connectée

1.2.3 Composants du système

Le système **GeoNDGR-Collecte** repose sur une architecture à trois tiers :

Composant	Description	Technologie
Application Mobile	Interface de collecte et de consultation des données, fonctionnant sur Android	Flutter / Dart
API Backend	Serveur REST exposant les endpoints pour la communication avec l'application mobile	GeoDjango / DRF
Base de données	Stockage et gestion des données attributaires et géométriques	PostgreSQL / PostGIS

TABLE 1.1 – Composants principaux du système GeoNDGR-Collecte

1.3 Portée du document

1.3.1 Public cible

Ce guide s'adresse principalement aux :

— **Développeurs Flutter** : pour comprendre la structure du code mobile et les patterns utilisés
— **Développeurs Backend** : pour la maintenance de l'API GeoDjango
— **Administrateurs système** : pour le déploiement et la configuration serveur
— **Chefs de projet** : pour avoir une vue d'ensemble technique du système

1.3.2 Prérequis de lecture

Pour tirer pleinement profit de ce document, le lecteur devrait avoir des connaissances de base en :

— Développement mobile (concepts généraux)
— Programmation orientée objet
— API REST et protocole HTTP
— Bases de données relationnelles
— Ligne de commande (bash/PowerShell)

Chapitre 2

Architecture Globale

2.1 Vue d'ensemble

L'architecture de **GeoNDGR-Collecte** suit un modèle **client-serveur** classique, adapté aux contraintes du terrain (connectivité intermittente, géolocalisation, données spatiales).

2.2 Flux de données

2.2.1 Collecte et synchronisation

Le flux de données dans **GeoNDGR-Collecte** suit un cycle bien défini :

Étape 1 : **Authentification** — L'utilisateur se connecte via l'application mobile. Un token JWT est généré et stocké localement.

Étape 2 : **Collecte terrain** — L'agent saisit les données (pistes, chaussées, points GPS) via les formulaires de l'application. Les données sont stockées dans la base SQLite locale.

Étape 3 : **Synchronisation** — Lorsque la connectivité est disponible, l'application envoie les données au serveur via l'API REST (requêtes POST).

Étape 4 : **Validation et stockage** — Le serveur GeoDjango valide les données, effectue les transformations de coordonnées nécessaires et stocke les informations dans PostgreSQL/PostGIS.

Étape 5 : **Consultation** — Les données synchronisées peuvent être consultées depuis l'application (requêtes GET) ou depuis d'autres interfaces de visualisation.

FIGURE 2.1 – Flux de données : de la collecte à la synchronisation

2.3 Rôles des composants

2.3.1 Application Mobile Flutter

L'application mobile constitue l'interface principale avec les utilisateurs de terrain. Ses responsabilités incluent :

- **Interface utilisateur** : Formulaires de saisie, cartes interactives, listes de données
- **Géolocalisation** : Acquisition des coordonnées GPS en temps réel
- **Stockage local** : Persistance des données hors-ligne via SQLite
- **Synchronisation** : Envoi et réception des données via l'API REST
- **Visualisation cartographique** : Affichage des entités géographiques sur Google Maps

2.3.2 API GeoDjango (Backend)

Le backend GeoDjango joue le rôle de **couche intermédiaire** entre l'application mobile et la base de données :

- **Exposition REST** : Endpoints sécurisés pour les opérations CRUD
- **Authentification** : Gestion des tokens JWT et des sessions utilisateurs
- **Validation** : Contrôle de la cohérence des données entrantes
- **Transformation spatiale** : Conversion des systèmes de coordonnées (SRID)
- **Sérialisation** : Conversion JSON/GeoJSON pour les échanges de données

2.3.3 PostgreSQL / PostGIS

La base de données assure le **stockage centralisé** de toutes les données du projet :

- **Données attributaires** : Informations descriptives (codes, états, types, etc.)
- **Données géométriques** : Points, lignes et polygones avec leurs coordonnées
- **Intégrité référentielle** : Relations entre les différentes entités
- **Requêtes spatiales** : Analyses géographiques avancées (intersections, buffers, etc.)

2.4 Schéma de déploiement

⚠ Attention

Le déploiement de l'API est réalisé sur un **serveur Windows** en utilisant `NGINX` comme reverse proxy et `Gunicorn` comme serveur WSGI. Cette configuration sera détaillée dans le chapitre dédié au déploiement.

Composant	Rôle	Port
NGINX	Reverse proxy, terminaison SSL, load balancing	82 / 443
Gunicorn	Serveur WSGI pour Django	8001
PostgreSQL	Base de données	5432

TABLE 2.1 – Configuration des ports de déploiement

Chapitre 3

Technologies et Outils

Ce chapitre présente en détail les technologies utilisées dans le développement de **GeoNDGR-Collecte**, ainsi que les outils recommandés pour la maintenance et l'évolution du projet.

3.1 Stack technique

3.1.1 Côté Mobile (Frontend)

Flutter

Flutter SDK

Version	3.4.1
Langage	Dart
Plateforme cible	Android (API Level 35)
Site officiel	`https://flutter.dev`

Flutter est le framework de développement mobile choisi pour **GeoNDGR-Collecte**. Ses avantages principaux :

- **Hot Reload** : Rechargement instantané du code pendant le développement
- **Compilation AOT** : Performances natives grâce à la compilation Ahead-Of-Time
- **Widgets riches** : Bibliothèque complète de composants UI personnalisables
- **Écosystème mature** : Large communauté et nombreux packages disponibles

Packages Flutter essentiels

Package	Utilisation
`http`	Communication HTTP avec l'API REST
`google_maps_flutter`	Affichage et interaction avec les cartes Google Maps
`location`	Acquisition des coordonnées GPS en temps réel
`sqflite`	Base de données SQLite locale pour le mode hors-ligne
`permission_handler`	Gestion des permissions Android (GPS, stockage)
`shared_preferences`	Stockage de préférences utilisateur (tokens, paramètres)
`provider`	Gestion d'état et injection de dépendances

TABLE 3.1 – Packages Flutter utilisés dans GeoNDGR-Collecte

3.1.2 Côté Serveur (Backend)

GeoDjango

GeoDjango

Framework	Django avec extension géospatiale
API	Django REST Framework (DRF)
Sérialisation	GeoFeatureModelSerializer
Site officiel	https://docs.djangoproject.com

GeoDjango étend Django avec des capacités géospatiales complètes :

— **Modèles géométriques** : Support natif des types Point, LineString, Polygon, etc.
— **Requêtes spatiales** : Opérations géographiques (contains, intersects, distance)
— **Intégration PostGIS** : Communication optimisée avec la base spatiale
— **Transformation SRID** : Conversion automatique entre systèmes de coordonnées

3.1.3 Base de données

PostgreSQL + PostGIS

PostgreSQL / PostGIS

SGBD	PostgreSQL 14+
Extension spatiale	PostGIS 3.x
SRID projet	4326 (WGS 84)
Outil d'administration	PgAdmin 4

3.2 Formats d'échange de données

3.2.1 JSON

Le format JSON standard est utilisé pour les données non spatiales :

```
{
    "id": 42,
    "code_piste": "PST_001",
    "etat_piste": "Bon",
    "largeur_emprise": 4.5
}
```

Listing 3.1 – Exemple de réponse JSON

3.2.2 GeoJSON

Le format GeoJSON est utilisé pour les données spatiales :

```
{
    "type": "Feature",
```

```
    "geometry": {
        "type": "MultiLineString",
        "coordinates": [
            [
                [-13.6773, 9.6412],
                [-13.6758, 9.6421]
            ]
        ]
    },
    "properties": {
        "id": 12,
        "code_piste": "PST_001"
    }
}
```

Listing 3.2 – Exemple de réponse GeoJSON

Information

Les coordonnées sont exprimées dans le système de référence géographique **WGS 84 (EPSG :4326)**, utilisé par les systèmes GPS et le format GeoJSON.

3.3 Outils de développement

Outil	Utilisation	Requis
Visual Studio Code	IDE principal pour Flutter et Python	Recommandé
Android Studio	SDK Android, émulateurs, débogage	Obligatoire
Git	Gestion de versions	Obligatoire
Postman	Test des endpoints API	Recommandé
PgAdmin 4	Administration PostgreSQL	Recommandé

TABLE 3.2 – Outils de développement recommandés

3.4 Serveur de déploiement

3.4.1 NGINX

NGINX est utilisé comme **reverse proxy** devant Gunicorn. Il gère la terminaison SSL/TLS, la mise en cache des ressources statiques et le load balancing.

3.4.2 Gunicorn

Gunicorn est le serveur WSGI Python qui exécute l'application Django avec gestion des workers pour la concurrence.

⚠ Attention

Sur Windows, Gunicorn ne fonctionne pas nativement. Il est nécessaire d'utiliser **WSL (Windows Subsystem for Linux)** ou une solution alternative comme `waitress`.

Chapitre 4

Environnement de Développement

Ce chapitre détaille les étapes nécessaires pour configurer un environnement de développement complet.

4.1 Prérequis logiciels

Composant	Description	Version
Flutter SDK	Framework de développement mobile	3.4.1+
Dart SDK	Langage de programmation (inclus avec Flutter)	3.x
Android Studio	IDE et SDK Android	Latest
Android SDK	Kit de développement Android	API Level 35
Java JDK	Kit de développement Java	11 ou 17
Git	Gestion de versions	2.x+

TABLE 4.1 – Prérequis logiciels pour le développement

4.2 Installation de Flutter

4.2.1 Téléchargement et installation

Étape 1 : **Télécharger Flutter SDK** depuis `https://docs.flutter.dev/get-started/install`

Étape 2 : **Extraire l'archive**

Commande

```
# Linux/macOS
tar xf flutter_linux.tar.xz -C ~/development

# Windows - Extraire vers C:\dev\flutter
```

Étape 3 : **Configurer le PATH**

Commande

```
# Linux/macOS
export PATH="$PATH:~/development/flutter/bin"
```

Étape 4 : Vérifier l'installation

Commande

```
flutter --version
flutter doctor
```

4.3 Configuration du projet

4.3.1 Clonage du dépôt

Commande

```
git clone https://github.com/Ayoub101010/PPRCollecte_Flutter.git
cd PPRCollecte_Flutter
flutter pub get
```

4.3.2 Configuration de la clé Google Maps

Modifiez le fichier `android/app/src/main/AndroidManifest.xml` :

```
1 <meta-data
2     android:name="com.google.android.geo.API_KEY"
3     android:value="VOTRE_CLE_API_GOOGLE_MAPS"/>
```

Listing 4.1 – Configuration de la clé Google Maps

Attention

Ne jamais commiter la clé API dans le dépôt Git ! Utilisez des variables d'environnement.

4.4 Lancement de l'application

Commande

```
# Verifier les appareils connectes
flutter devices

# Lancer sur l'appareil par defaut
flutter run

# Lancer sur un appareil specifique
flutter run -d emulator-5554
```

4.4.1 Raccourcis utiles

Touche	Action
r	Hot Reload
R	Hot Restart
q	Quitter

TABLE 4.2 – Raccourcis clavier en mode debug

Chapitre 5

Structure du Code Flutter

Ce chapitre présente l'organisation du code source de l'application **GeoNDGR-Collecte**.

5.1 Vue d'ensemble de l'arborescence

```
lib/
|-- main.dart            # Point d'entree
|-- config.dart          # Configuration
|-- models/              # Modeles de donnees
|   |-- piste_model.dart
|   |-- chaussee_model.dart
|-- services/            # Services metier
|   |-- api_service.dart
|   |-- location_service.dart
|   |-- sync_service.dart
|-- pages/               # Ecrans
|   |-- home_page.dart
|   |-- login_page.dart
|-- widgets/             # Composants UI
|   |-- map_widget.dart
|-- helpers/             # Utilitaires
    |-- database_helper.dart
```

Listing 5.1 – Arborescence du dossier lib/

5.2 Fichiers principaux

5.2.1 main.dart — Point d'entrée

```dart
import 'package:flutter/material.dart';
import 'pages/login_page.dart';

void main() {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'GeoNDGR-Collecte',
```

```
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primarySwatch: Colors.blue,
        useMaterial3: true,
      ),
      home: const LoginPage(),
    );
  }
}
```

Listing 5.2 – Structure de main.dart

5.2.2 config.dart — Configuration

```
class AppConfig {
  static const String baseUrl = 'https://api.example.com:82';
  static const String loginEndpoint = '/api/auth/login/';
  static const String pistesEndpoint = '/api/pistes/';
  static const int connectionTimeout = 30;
  static const int srid = 4326;
}
```

Listing 5.3 – Exemple de config.dart

5.3 Couche Models

```
class PisteModel {
  final int? id;
  final String codePiste;
  final String etatPiste;
  final Map<String, dynamic>? geometry;

  PisteModel({
    this.id,
    required this.codePiste,
    required this.etatPiste,
    this.geometry,
  });

  factory PisteModel.fromJson(Map<String, dynamic> json) {
    return PisteModel(
      id: json['id'],
      codePiste: json['properties']['code_piste'],
      etatPiste: json['properties']['etat_piste'],
      geometry: json['geometry'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'code_piste': codePiste,
      'etat_piste': etatPiste,
      'geom': geometry,
    };
  }
}
```

Listing 5.4 – Exemple de piste_model.dart

5.4 Couche Services

5.4.1 ApiService — Communication HTTP

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;

class ApiService {
  static String? _token;

  static Future<bool> login(String username, String password) async {
    final response = await http.post(
      Uri.parse('${AppConfig.baseUrl}/api/auth/login/'),
      headers: {'Content-Type': 'application/json'},
      body: jsonEncode({
        'username': username,
        'password': password,
      }),
    );

    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      _token = data['token'];
      return true;
    }
    return false;
  }
}
```

Listing 5.5 – Extrait de api_service.dart

5.5 Diagramme des dépendances

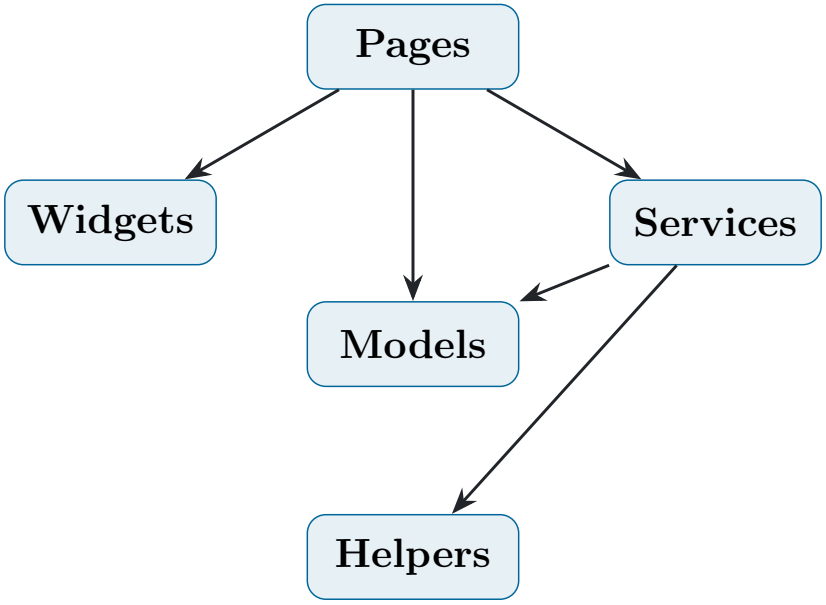

FIGURE 5.1 – Dépendances entre les couches de l'application

Chapitre 6

API REST et Communication

Ce chapitre détaille les mécanismes de communication entre l'application mobile et le backend GeoDjango.

6.1 Endpoints disponibles

Méthode	Endpoint	Description
POST	`/api/auth/login/`	Authentification
GET	`/api/pistes/`	Liste des pistes
POST	`/api/pistes/`	Créer une piste
GET	`/api/pistes/{id}/`	Détail d'une piste
PUT	`/api/pistes/{id}/`	Modifier une piste
DELETE	`/api/pistes/{id}/`	Supprimer une piste
GET	`/api/chaussees/`	Liste des chaussées
POST	`/api/chaussees/`	Créer une chaussée

TABLE 6.1 – Endpoints de l'API GeoNDGR-Collecte

6.2 Authentification JWT

L'API utilise l'authentification par **JSON Web Token (JWT)**.

6.2.1 Requête de login

```
{
    "username": "agent_terrain",
    "password": "motdepasse123"
}
```

Listing 6.1 – Requête POST /api/auth/login/

6.2.2 Réponse de login

```
{
    "token": "eyJhbGciOiJIUzI1NiIs...",
    "user_id": 42,
    "username": "agent_terrain"
}
```

Listing 6.2 – Réponse du serveur

6.2.3 Utilisation du token

Toutes les requêtes authentifiées doivent inclure le token :

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

6.3 Codes de réponse HTTP

Code	Statut	Signification
200	OK	Requête réussie (GET, PUT)
201	Created	Ressource créée (POST)
400	Bad Request	Données invalides
401	Unauthorized	Token invalide
404	Not Found	Ressource introuvable
500	Server Error	Erreur serveur

TABLE 6.2 – Codes de réponse HTTP de l'API

6.4 Cycle de synchronisation

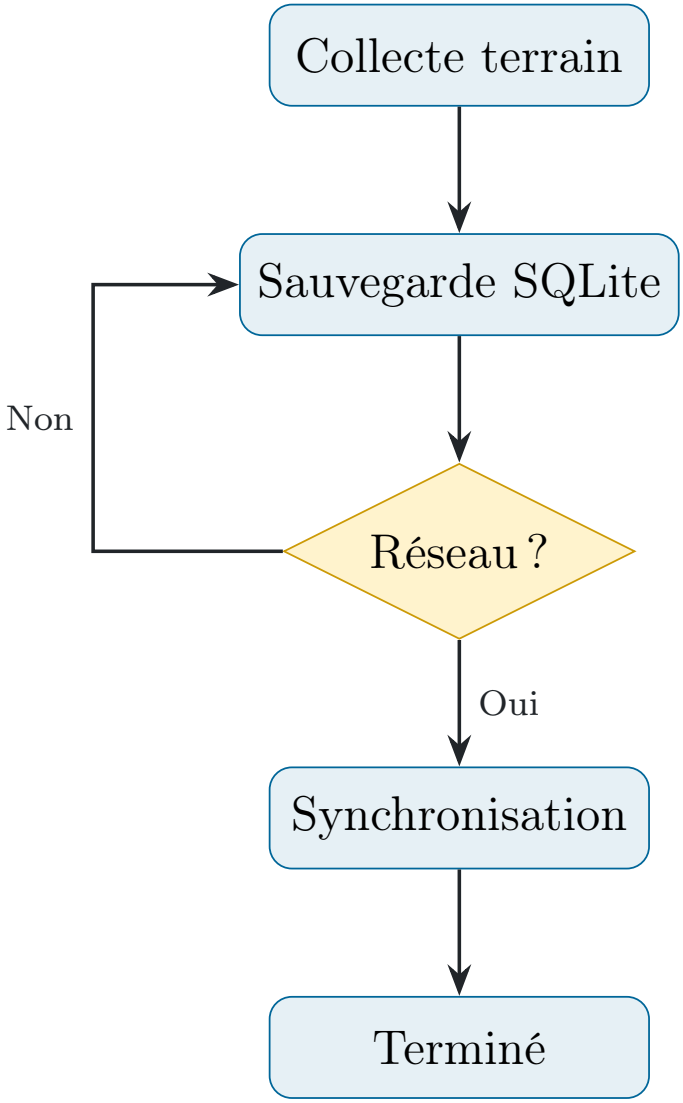

FIGURE 6.1 – Flux de synchronisation des données

6.5 Sécurité et bonnes pratiques

Bonne pratique

Recommandations de sécurité :

- Toujours utiliser HTTPS en production
- Ne jamais stocker les mots de passe en clair
- Renouveler régulièrement les tokens JWT
- Valider toutes les données côté serveur

Chapitre 7

Base de Données

Ce chapitre présente l'architecture des données de **GeoNDGR-Collecte**.

7.1 Architecture des données

Aspect	SQLite (Local)	PostGIS (Serveur)
Usage	Stockage temporaire hors-ligne	Stockage permanent
Données	Non synchronisées	Toutes les données
Géométries	JSON texte	Types natifs

TABLE 7.1 – Comparaison des deux systèmes de stockage

7.2 Schéma PostgreSQL/PostGIS

7.2.1 Table : pistes

```
CREATE TABLE pistes (
    id SERIAL PRIMARY KEY,
    code_piste VARCHAR(50) UNIQUE NOT NULL,
    nom_origine_piste VARCHAR(255),
    nom_destination_piste VARCHAR(255),
    largeur_emprise DOUBLE PRECISION,
    frequence_trafic VARCHAR(50),
    etat_piste VARCHAR(50),
    geom GEOMETRY(MultiLineString, 4326),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_pistes_geom ON pistes USING GIST(geom);
```

Listing 7.1 – Structure de la table pistes

7.2.2 Table : chaussees

```
CREATE TABLE chaussees (
    id SERIAL PRIMARY KEY,
    code_chaussee VARCHAR(50) UNIQUE NOT NULL,
    piste_id INTEGER REFERENCES pistes(id),
    type_chaussee VARCHAR(100),
```

```
    etat_chaussee VARCHAR(50),
    geom GEOMETRY(MultiLineString, 4326),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_chaussees_geom ON chaussees USING GIST(geom);
```

Listing 7.2 – Structure de la table chaussees

7.3 Schéma SQLite local

```
CREATE TABLE pistes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    code_piste TEXT NOT NULL,
    nom_origine_piste TEXT,
    nom_destination_piste TEXT,
    geometry TEXT,
    synced INTEGER DEFAULT 0,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);
```

Listing 7.3 – Schéma SQLite local

7.4 Requêtes spatiales courantes

```
-- Calculer la longueur totale des pistes
SELECT SUM(ST_Length(geom)) AS longueur_totale
FROM pistes;

-- Trouver les pistes dans un rayon (ex: ~0.05 deg ~ quelques km
    selon latitude)
SELECT code_piste FROM pistes
WHERE ST_DWithin(
    geom,
    ST_SetSRID(ST_MakePoint(-13.6773, 9.6412), 4326),
    0.05
);
```

Listing 7.4 – Requêtes spatiales utiles

Chapitre 8

Déploiement Serveur

Ce chapitre détaille la procédure de déploiement sur un serveur Windows avec **NGINX** et **Gunicorn**.

Attention

Gunicorn ne fonctionne pas nativement sous Windows. Ce guide utilise **WSL** (Windows Subsystem for Linux).

8.1 Architecture de déploiement

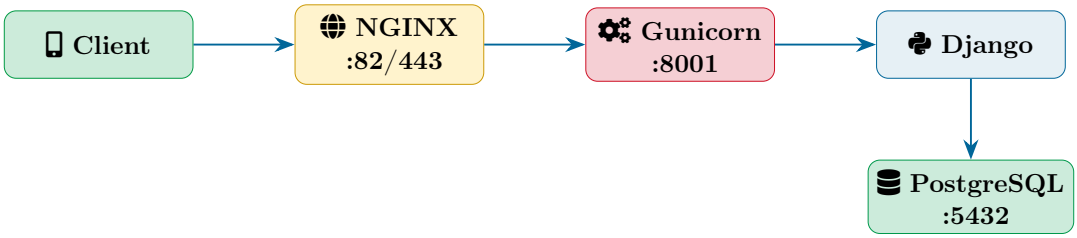

FIGURE 8.1 – Architecture de déploiement

8.2 Installation de WSL

Commande

```
# PowerShell en administrateur
wsl --install -d Ubuntu-22.04
wsl --set-default-version 2
```

8.3 Configuration PostgreSQL

Commande

```
sudo -u postgres psql
CREATE USER geondgr_user WITH PASSWORD 'motdepasse';
CREATE DATABASE geondgr_db OWNER geondgr_user;
\c geondgr_db
CREATE EXTENSION postgis;
```

8.4 Déploiement Django

Commande

```
cd /var/www/geondgr
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
pip install gunicorn
python manage.py migrate
python manage.py collectstatic
```

8.5 Configuration Gunicorn

```
bind = "127.0.0.1:8001"
workers = 4
timeout = 120
accesslog = "/var/log/gunicorn/access.log"
errorlog = "/var/log/gunicorn/error.log"
```

Listing 8.1 – gunicorn.conf.py

8.6 Configuration NGINX

```
upstream gunicorn_backend {
    server 127.0.0.1:8001;
}

server {
    listen 8082;
    server_name 81.192.142.205;
    client_max_body_size 50M;

    location /static/ {
        alias /var/www/geondgr/staticfiles/;
    }

    location / {
        proxy_pass http://gunicorn_backend;
```

```
16          proxy_set_header Host $host;
17          proxy_set_header X-Real-IP $remote_addr;
18      }
19  }
```

Listing 8.2 – Configuration NGINX

8.7 Vérification du déploiement

Bonne pratique

Checklist :

- ✓ PostgreSQL démarre correctement
- ✓ Gunicorn répond sur le port 8001
- ✓ NGINX répond sur le port 82
- ✓ L'API répond aux requêtes

Chapitre 9

Tests et Validation

9.1 Types de tests

Type	Description	Outils
Unitaires	Fonctions individuelles	Flutter Test
Intégration	Interactions composants	Integration Test
UI	Interface utilisateur	Widget Test
API	Endpoints REST	Postman
Terrain	Conditions réelles	Appareil physique

TABLE 9.1 – Types de tests

9.2 Tests en mode Debug

Commande

```
flutter devices
flutter run
flutter run -d emulator-5554
```

9.3 Génération des packages

9.3.1 APK de release

Commande

```
flutter build apk --release
# Sortie: build/app/outputs/flutter-apk/app-release.apk
```

9.3.2 Android App Bundle

Commande

```
flutter build appbundle --release
# Sortie: build/app/outputs/bundle/release/app-release.aab
```

9.4 Tests hors-ligne

1. Activer le Mode Avion
2. Collecter des données
3. Fermer et rouvrir l'app
4. Vérifier les données locales
5. Désactiver Mode Avion
6. Synchroniser et vérifier

9.5 Checklist avant release

Vérification	OK
Code analysé (`flutter analyze`)	
Tests passent (`flutter test`)	
Testé sur émulateur	
Testé sur appareil physique	
Tests hors-ligne effectués	
Aucune clé API exposée	
Version mise à jour	

TABLE 9.2 – Checklist avant release

Chapitre 10

Maintenance et Contribution

10.1 Gestion de versions avec Git

10.1.1 Structure des branches

Branche	Préfixe	Description
`main`	—	Code de production stable
`develop`	—	Branche d'intégration
`feature/*`	`feature/`	Nouvelles fonctionnalités
`fix/*`	`fix/`	Corrections de bugs
`hotfix/*`	`hotfix/`	Corrections urgentes

TABLE 10.1 – Convention de nommage des branches

10.1.2 Workflow de développement

Commande

```
git checkout develop
git pull origin develop
git checkout -b feature/ma-feature
# ... developpement ...
git add .
git commit -m "feat: description"
git push origin feature/ma-feature
# Creer une Pull Request
```

10.2 Conventions de commit

Type	Description	Exemple
`feat`	Nouvelle fonctionnalité	`feat: add offline sync`
`fix`	Correction de bug	`fix: resolve GPS timeout`
`docs`	Documentation	`docs: update README`
`refactor`	Refactorisation	`refactor: simplify API`
`test`	Ajout de tests	`test: add unit tests`
`chore`	Maintenance	`chore: update deps`

TABLE 10.2 – Types de commits

10.3 Conventions de code

Élément	Convention	Exemple
Variables	camelCase	`userName`
Fonctions	camelCase	`fetchData()`
Classes	PascalCase	`PisteModel`
Fichiers	snake_case	`piste_model.dart`
Constantes	UPPER_SNAKE_CASE	`API_BASE_URL`

TABLE 10.3 – Conventions de nommage

10.4 Checklist avant Pull Request

Vérification	OK
Code formaté (`flutter format .`)	
Aucune erreur (`flutter analyze`)	
Tests passent (`flutter test`)	
Pas de secrets dans le code	
Documentation mise à jour	

TABLE 10.4 – Checklist avant PR

10.5 Versionnage sémantique

Le projet suit le **Semantic Versioning** : `MAJOR.MINOR.PATCH`

Élément	Incrémenté quand...
MAJOR	Changements incompatibles
MINOR	Nouvelles fonctionnalités rétrocompatibles
PATCH	Corrections de bugs

TABLE 10.5 – Règles de versionnage

10.6 Contact et support

Rôle	Nom	Responsabilité
Développeur	Dahdouh Ayoub	Architecture, développement, tests, conception
Développeur	Moutamanni Abdourahman	Architecture, développement, tests, conception
Encadrant	*El Imami Ayoub*	Supervision

TABLE 10.6 – Équipe de développement

Chapitre 11

Annexes

11.1 Glossaire

Terme	Définition
API	Application Programming Interface
APK	Android Package Kit
AAB	Android App Bundle
CRUD	Create, Read, Update, Delete
DRF	Django REST Framework
GeoJSON	Format JSON pour données géographiques
GPS	Global Positioning System
Gunicorn	Serveur HTTP WSGI pour Python
JWT	JSON Web Token
NGINX	Serveur web / Reverse proxy
PostGIS	Extension spatiale PostgreSQL
REST	Representational State Transfer
SRID	Spatial Reference System Identifier
SQLite	Base de données légère intégrée
UTM	Universal Transverse Mercator
WSGI	Web Server Gateway Interface
WSL	Windows Subsystem for Linux

11.2 Références techniques

Technologie	URL
Flutter	https://docs.flutter.dev
Django	https://docs.djangoproject.com
PostgreSQL	https://www.postgresql.org/docs/
PostGIS	https://postgis.net/documentation/
NGINX	https://nginx.org/en/docs/

TABLE 11.2 – Documentation officielle

11.3 Codes EPSG

Code	Nom	Utilisation
4326	WGS 84	Coordonnées GPS / GeoJSON

TABLE 11.3 – Codes EPSG utilisés

11.4 Variables d'environnement

Variable	Description
`DJANGO_SECRET_KEY`	Clé secrète Django
`DB_PASSWORD`	Mot de passe PostgreSQL
`DB_HOST`	Hôte de la base de données
`DEBUG`	Mode debug (True/False)

TABLE 11.4 – Variables d'environnement

11.5 Historique des versions

Version	Date	Changements
1.0.0	Sept. 2025	Version initiale (PPRCollecte)
2.0.0	Déc. 2025	Renommage GeoNDGR-Collecte

TABLE 11.5 – Historique des versions

GeoNDGR-Collecte

Guide de Développement

Réalisé par **Dahdouh Ayoub** & **Moutamanni Abdourahman**

Encadré par ***El Imami Ayoub***

ETAFAT | NDGR

2025