Document de Qualité Logicielle pour le Développement d' ESEOSPORT

Introduction

Ce document a pour but de fournir des lignes directrices et des bonnes pratiques à adopter avant et pendant le développement de l'application mobile ESEOSPORT basée sur le framework **Flutter**. Il met l'accent sur l'organisation, la structure du code, les choix technologiques, et les pratiques recommandées. Certaines parties n'ont pas été mises en place sur le projet, elles sont annotées avec *"TO DO".*

Sommaire

1. Technologies et outils utilisés

Framework : Flutter (framework open-source de Google pour les applications multiplateformes).

Langage : Dart (utilisé par Flutter).

Moteur de rendu : Skia (permet la transformation du code Dart en code natif lors de la compilation).

IDE : Android Studio (choisi pour son intégration avec Flutter et ses outils de développement).

Base de données : MongoDB (optionnelle selon les besoins d'historique ou d'évolution).

Outils de versionnage : GitLab ou GitHub (pour la gestion du code source).

2. Conception et Architecture

2.1 Structure du Projet (Frontend)

Clean Architecture est un concept architectural proposé par Robert C. Martin qui vise à rendre les applications **modulaires**, **testables**, et **indépendantes des frameworks**. Bien que **Clean Architecture** présente de nombreux avantages, il est important de noter que son implémentation peut introduire une complexité supplémentaire, souvent disproportionnée pour des applications plus simples.

Dans le cas de notre application, qui se concentre sur la récupération de données via Bluetooth, leur affichage et la gestion d'un historique, le pattern **MVVM (Model-View-ViewModel)** est plus approprié.

- **Model** : représente vos données (Bluetooth, historique).
- **View** : gère uniquement l'affichage des données à l'utilisateur.
- **ViewModel** : contient la logique d'interface et interagit avec le **Model** pour fournir les données à la **View** sans qu'elles soient directement liées.

Cette architecture permet un code maintenable et facilement évolutif sans la surcharge que pourrait entraîner une architecture plus complexe.

```
/lib
├── /models                    // Modèles de données (Bluetooth, historique)
│   ├── bluetooth_data.dart    // Modèle pour les données Bluetooth
│   ├── history_entry.dart     // Modèle pour l'historique
│
├── /viewmodels                // Logique d'interface utilisateur
│   ├── bluetooth_viewmodel.dart  // ViewModel pour les données Bluetooth
│   ├── history_viewmodel.dart    // ViewModel pour l'historique
│
├── /views                     // Composants d'affichage
│   ├── bluetooth_screen.dart     // Vue pour les données Bluetooth
│   ├── history_screen.dart       // Vue pour l'historique
└── main.dart                  // Point d'entrée de l'application
```

Exemple d'architecture pour ESEOSPORT

2.2 Pattern et Gestion de l'état

L'intégration du pattern **BLoC (Business Logic Component)** et de l'injection de dépendances dans notre architecture MVVM améliore considérablement la structure, la testabilité et la scalabilité de notre application.

L'utilisation de BLoC garantit une séparation claire entre la logique métier et la présentation, ce qui contribue à améliorer la maintenabilité et la testabilité de l'application.

```
/lib
├── /models                    // Modèles de données (Bluetooth, historique)
│   ├── bluetooth_data.dart    // Modèle pour les données Bluetooth
│   ├── history_entry.dart     // Modèle pour l'historique
│
├── /blocs                     // Dossier pour les blocs BLoC
│   ├── bluetooth_bloc.dart    // Bloc pour la gestion des données Bluetooth
│   ├── history_bloc.dart      // Bloc pour la gestion de l'historique
│
├── /viewmodels                // Logique d'interface utilisateur
│   ├── bluetooth_viewmodel.dart  // ViewModel pour les données Bluetooth (facultatif)
│   ├── history_viewmodel.dart    // ViewModel pour l'historique (facultatif)
│
├── /views                     // Composants d'affichage
│   ├── bluetooth_screen.dart  // Vue pour les données Bluetooth
│   ├── history_screen.dart    // Vue pour l'historique
└── main.dart                  // Point d'entrée de l'application
```

Architecture Frontend pour ESEOSPORT

2.3 Structure du projet (Backend)

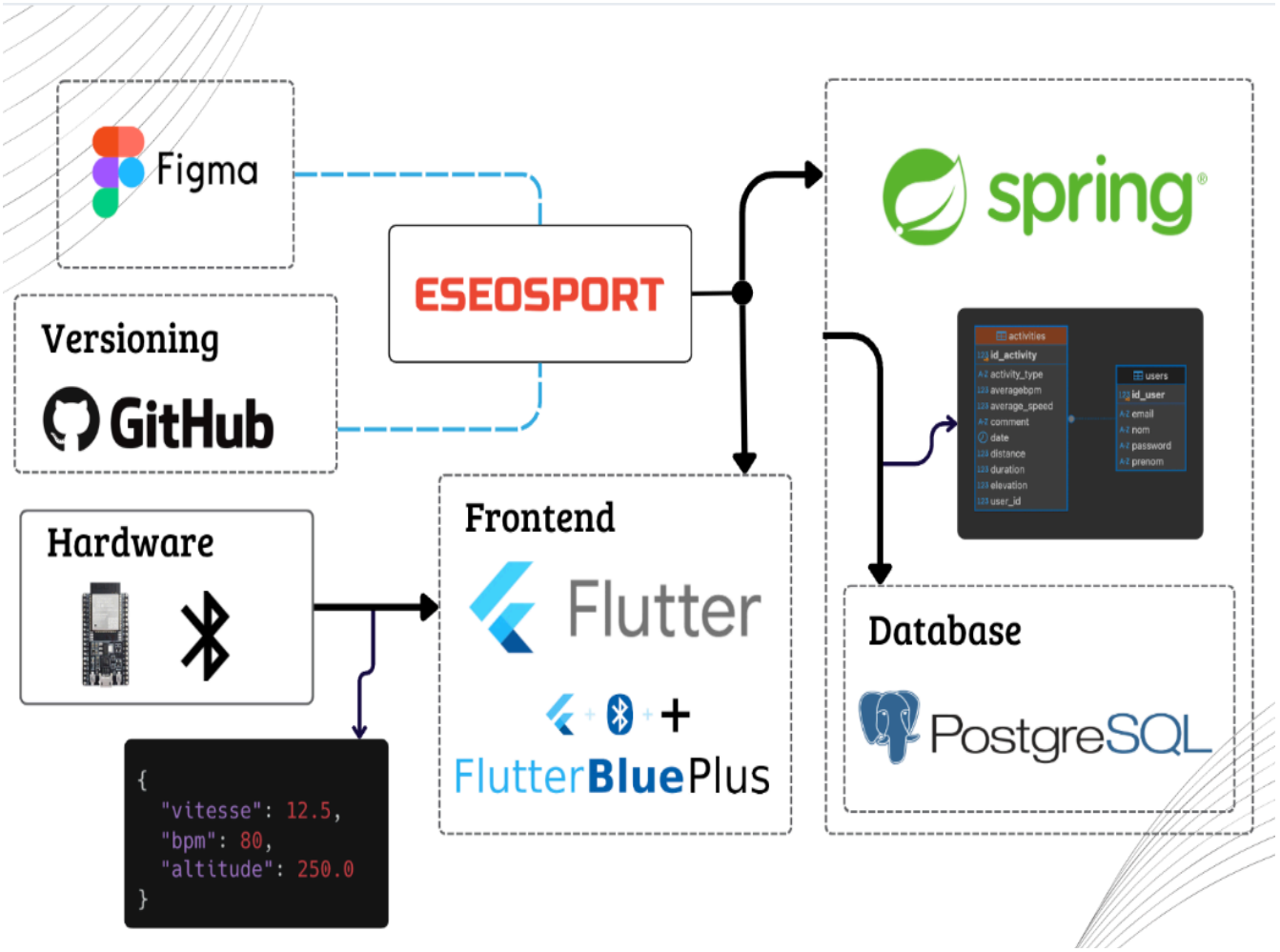
L’application a besoin de stocker des activités, générer des statistiques ou permettre un mode hors ligne ce qui nécessite l’adoption d’une **base de données.** L'utilisation d'une **base de données comme MongoDB** est envisageable pour une **possible évolution de l’application**. Étant donné qu’elle traite des données en Json/Bson, la transmission des données peut se faire plus efficacement. Pour un **stockage léger** pour les données temporaires entre sessions il est possible d’utiliser (*SharedPreferences*) .

L'application peut fonctionner sans une **API** mais l'intégration d'une **API** via le framework **Spring** présente de nombreux avantages qui amélioreront la fonctionnalité, la sécurité et la scalabilité de votre projet. L**’API** expose les **endpoints pour gérer les données** (par exemple, récupération des données Bluetooth et historique).

```
/src
├── /models                    // Modèles de données (Bluetooth, historique)
│   ├──
├── /repositories
├── /services                  // Services métier pour la logique de l'application
│   ├──
├── /controllers               // Contrôleurs REST pour gérer les requêtes HTTP
│   ├──
├── /config                    // Configuration de l'application
│   ├── WebConfig.java         // Configuration des routes, CORS, etc.
│   ├── DatabaseConfig.java    // Configuration de la base de données
├── /exceptions                // Gestion des exceptions personnalisées
│   ├── ResourceNotFoundException.java // Exception pour les ressources non trouvées
│   ├── BadRequestException.java      // Exception pour les requêtes incorrectes
└── Application.java           // Point d'entrée de l'application
```

Architecture Backend choisie pour ESEOSPORT

2.3 Structure du projet et solutions techniques



Structure et solutions techniques de l'application ESEOSPORT

3. Standards et Bonnes Pratiques de Développement

3.1 Conventions de Nommage

Respecter les conventions de nommage Dart et Flutter pour assurer la lisibilité et la cohérence du code :

Classes et types : CamelCase (exemple : `ActivityManager`, `UserProfile`).

Variables et fonctions : lowerCamelCase (exemple : `calculateSpeed`, `userProfile`).

Constantes : MAJUSCULES avec underscores (exemple : `MAX_SPEED`, `DEFAULT_BPM`).

3.2 Commentaires et Documentation

Utilisation de **commentaires explicatifs** pour les blocs de code complexes. Utilisation des **doc comments** (///) pour générer une documentation automatique avec DartDoc.

3.3 Gestion des Exceptions

Encadrez les appels réseau, les accès à la base de données et les interactions Bluetooth avec des blocs `try-catch` pour éviter les crashs de l'application en cas d'échec.

3.4 Tests

Mise en place des tests efficaces pour assurer la qualité de l'application :

Tests unitaires : pour valider la logique métier (par exemple, calculer les statistiques).

Tests d'intégration : pour valider l'interaction entre les différentes parties de l'application (par exemple, lecture des données Bluetooth).

Tests UI : pour valider l'affichage et les interactions utilisateur (utilisez des outils comme `integration_test`).

3.5 Versionnement

Utilisation de Git pour la gestion des versions du code avec la création de branches spécifiques pour chaque fonctionnalité. (branches : dev et main).

Lien du repository : https://github.com/msbduno/eseosport_app.git

Feat	Ajout d'une fonctionnalité
Fix	Réparation d'un bug
Refactor	Simplification du code

Convention de nommage GIT

4. Qualité du Code

4.1 Les principes de qualité du code

Lisibilité : Écrire du code simple et compréhensible pour faciliter la maintenance.

Modularité : Diviser le code en petites fonctions ou classes pour éviter les monolithes.

Réutilisabilité : Créez des composants réutilisables (par exemple, des **widgets**).

```
/lib
├── /models                    // Modèles de données
│   ├── bluetooth_data.dart    // Modèle pour les données Bluetooth
│   ├── history_entry.dart     // Modèle pour l'historique
├── /viewmodels                // ViewModels pour la logique d'interface
│   ├── bluetooth_viewmodel.dart  // ViewModel pour les données Bluetooth
│   ├── history_viewmodel.dart    // ViewModel pour l'historique
├── /views                     // Composants d'affichage
│   ├── bluetooth_screen.dart     // Vue pour les données Bluetooth
│   ├── history_screen.dart       // Vue pour l'historique
├── /widgets                   // Dossier pour les widgets réutilisables
│   ├── custom_button.dart     // Widget personnalisé pour les boutons
│   └── loading_indicator.dart    // Widget pour un indicateur de chargement
└── main.dart                  // Point d'entrée de l'application
```

Architecture *pour ESEOSPORT* avec utilisation de widgets

5. Lancement de l'application

5.1 Pré-requis

Installation de Flutter et de Dart.

Mise en place de l'environnement de développement (Android Studio ou VS Code avec Flutter plugin).

Il est nécessaire d'ajouter des SDK (Software Development Kit). Une aide à l'installation est proposer par Flutter : https://docs.flutter.dev/get-started/install

5.2 Commandes de base

Vérification de l'installation : `flutter doctor`

Mettre à jour Flutter : `flutter upgrade`

Installer les dépendances : `flutter pub get`

Lancer l'émulateur ou un appareil physique : Démarrer un émulateur Android ou connecter un téléphone via USB. Pour la connexion USB il faut passer en mode développeur sur votre smartphone android.

Lancer l'application : `flutter run` pour exécuter l'application à partir du fichier `main.dart`.

5.2 Clonage GitHub

Pour pouvoir lancer le projet, vous avez besoin de cloner ce lien dans un dossier sur votre machine : https://github.com/msbduno/eseosport_app_v2.git

Ensuite vous devez ouvrir le projet “backend” sur intelliJ et le projet “eseosport_app” sur Android Studio.

6. Diagrammes UML

6.1 Script SQL

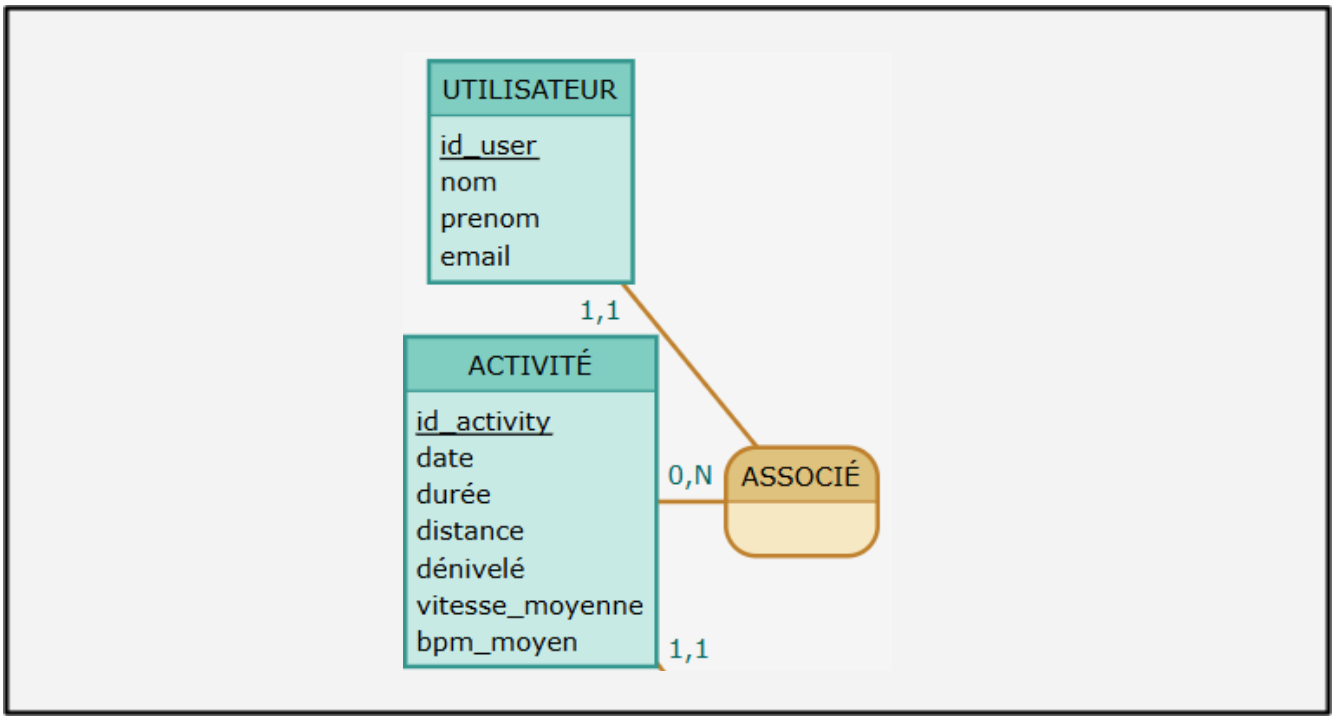
En cas d'utilisation d'une base de données, un **MCD** et un **MLD** est nécessaire pour organiser les entités et leurs relations. Cela inclut la modélisation des activités, des utilisateurs, et des sessions Bluetooth.

Voici un script SQL qui modélise nos besoins en termes de base de données, ce dernier reste à être adapté si une technologie telle que MongoDB est utilisée.

```
1.  CREATE TABLE Utilisateur (
2.    id_user INTEGER PRIMARY KEY AUTOINCREMENT,
3.    nom TEXT NOT NULL,
4.    prenom TEXT NOT NULL,
5.    email TEXT NOT NULL UNIQUE,
6.  );
7.
8.  CREATE TABLE Activite (
9.    id_activity INTEGER PRIMARY KEY AUTOINCREMENT,
10.   date DATETIME NOT NULL,
11.   duree INTEGER NOT NULL,
12.   distance REAL NOT NULL,
13.   denivele REAL,
14.   vitesse_moyenne REAL,
15.   bpm_moyen INTEGER,
16.   utilisateur_id INTEGER,
17.   FOREIGN KEY (utilisateur_id) REFERENCES Utilisateur(id_user)
18. );
19.
20. CREATE TABLE DataPoint (
21.   id_data_point INTEGER PRIMARY KEY AUTOINCREMENT,
22.   timestamp DATETIME NOT NULL,
23.   vitesse REAL NOT NULL,
24.   bpm INTEGER,
25.   altitude REAL,
26.   activite_id INTEGER,
27.   FOREIGN KEY (activite_id) REFERENCES Activite(id_activity)
28. );
```

Script SQL *pour ESEOSPORT*

6.2 Modèle Conceptuel des Données (MCD)



MCD *pour ESEOSPORT*

Le projet actuel utilise une base de données PostgreSQL.
Pour lancer le projet il sera donc nécessaire de télécharger Postgre sur votre machine.
https://www.postgresql.org/download/

6.3 Diagramme de Classe

TO DO :

Utilisez l'outil `dcdg` (https://pub.dev/packages/dcdg) pour générer des diagrammes de classe à partir du code Dart. Ce diagramme permet de visualiser les relations entre les différents objets de l'application.

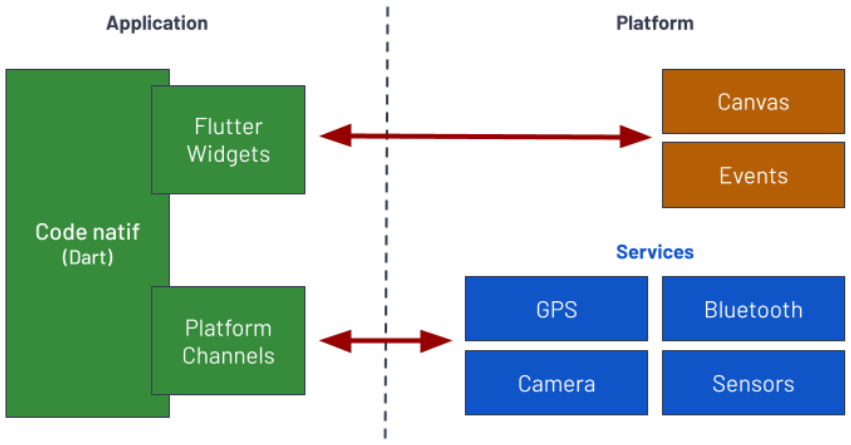
7. Solutions Techniques

7.1 Les particularité de Dart

Développé par Google, 1ère version en 2011

- Initialement créé comme une **alternative à JavaScript**
- 2 modes de fonctionnement : **interprété ou compilé**
- Compilation pour du natif : **mobile**, **desktop**, **backend**
- Compilation pour du web : **JavaScript**
- **Langage fortement typé**
- **Langage orienté objet**

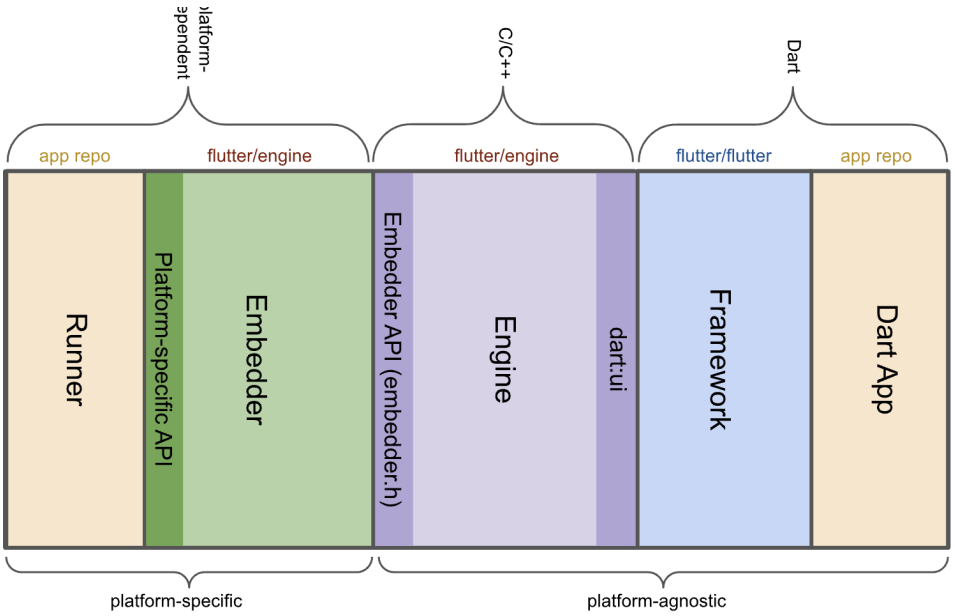
7.2 Approche Flutter



P. Raguin

21

7.3 Architecture Flutter



7.4 Transmission et traitement des données via bluetooth

Frontend (Flutter) :

Récupération des données : L'application Flutter interagit directement avec les périphériques Bluetooth pour récupérer les données. Cela se fait à l'aide de bibliothèques Flutter spécifiques au Bluetooth, comme `flutter_blue`, `flutter_reactive_ble`, ou d'autres.

Traitement des données : Une fois que les données sont reçues par l'application Flutter, elles peuvent être traitées localement dans l'application (par exemple, formatées, filtrées, etc.) avant d'être affichées à l'utilisateur.

Backend (Spring) :

Stockage des données : Si vous souhaitez conserver un historique des données Bluetooth ou effectuer des analyses, vous enverrez ces données au backend. Par exemple, après avoir récupéré les données, l'application Flutter peut envoyer une requête HTTP (POST) à votre API Spring pour stocker les données dans une base de données.

Gestion des requêtes : Le backend peut également être utilisé pour servir d'autres données, comme des historiques d'activités, des statistiques, etc.

7.5 Code permettant la connexion Bluetooth

7.5.1 Architecture de la connexion Bluetooth

L'application utilise une architecture en trois couches pour gérer la connexion Bluetooth :

1. **BluetoothRepository :** Couche de données principale qui gère la connexion Bluetooth
2. **BluetoothServiceManager :** Service utilitaire pour les opérations Bluetooth de bas niveau
3. **LiveDataViewModel :** ViewModel qui fait le lien entre les données Bluetooth et l'interface utilisateur

7.5.2 Composants principaux

BluetoothRepository

Le repository implémente les fonctionnalités suivantes :

- Scan des appareils Bluetooth disponibles
- Connexion à un appareil spécifique
- Gestion du flux de données
- Gestion de l'état de la connexion

```
static const String TARGET_DEVICE_NAME = 'BLE-Random';

static const String SERVICE_UUID = '1111';
static const String CHARACTERISTIC_UUID = '2222';
```

Les constantes définissent les paramètres de connexion spécifiques à notre appareil.

Processus de connexion

1. Initiation du scan :

```
Future<bool> startScan({Duration timeout = const Duration(seconds: 10)})
```

- Vérifie la disponibilité du Bluetooth
- Lance un scan avec un timeout de 10 secondes
- Recherche l'appareil avec le nom spécifié

2. Établissement de la connexion :

```
Future<bool> _connectToDevice(BluetoothDevice device)
```

- Connexion à l'appareil sélectionné
- Découverte des services disponibles
- Configuration des caractéristiques de notification

3. Gestion des données :

```
void _processIncomingData(List<int> data)
```

- Réception des données brutes
- Conversion en format JSON
- Distribution des données via un StreamController

Conclusion

Ce document constitue une base solide pour le développement de l'application Flutter. Il rassemble les bonnes pratiques, l'organisation du projet, et les recommandations techniques afin d'assurer un développement efficace, propre et évolutif.