Dossier Technique

Projet Trottinette Electrique

Établissement Saint Jean-Baptiste de La Salle

9 Rue Notre Dame des 7 Douleurs - Avignon

Année scolaire 2018-2019

Sy Somphon (BTS SN-IR)

La Salle
Avignon
Frères des Écoles Chrétiennes

Sommaire

Présentation générale du projet

Expression du besoin

Il s'agit de réaliser une application pour une trottinette électrique équipée de capteurs afin que le client puisse voir en temps réel les données de fonctionnement de la trottinette électrique telle que le pourcentage de batterie de la trottinette , l'inclinaison de la trottinette et la vitesse.

Présentation du projet

La trottinette sera équipée de :

- d'un microcontrôleur équipé d'une liaison Bluetooth
- d'un module ADXL335 pour mesurer l'inclinaison
- deux modules responsable de la régulation de la tension du moteur (antivol et régulation courant d'entrée)
- d'un module responsable de la mesure de la vitesse de la trottinette
- d'un module responsable de la mesure de la tension

L'application permettra à l'utilisateur de :

- Se connecter avec la trottinette
- Visualiser sur une carte sa position
- Visualiser le pourcentage de batterie actuelle de la trottinette
- Saisir un trajet
- Visualiser le trajet de sa position actuelle jusqu'à la destination
- Visualiser le pourcentage de batterie que la trottinette consommeras durant le trajet
- Visualiser durant un déplacement avec la trottinette la vitesse de la trottinette
- Verrouiller sa trottinette

Exigences

L'application doit être simple d'utilisation et doit pouvoir afficher les données de fonctionnement de la trottinette ainsi que de pouvoir l'arrêter.

Identification du travail à réaliser

Étudiants chargés du projet

Option EC :

- Étudiant 1 : GRAS Thomas
- Étudiant 2 : HALLIER Kévin

Option IR :

- Étudiant 3 : **SY Somphon**

Répartition des tâches entre étudiants

Étudiant IR : SY Somphon

Module de visualisation - terminal mobile

- Visualiser les données de fonctionnement et la géolocalisation de la TTE
- Visualiser l'autonomie pour un parcours
- Géolocaliser la TTE
- Visualiser le trajet (en option)
- Verrouiller la trottinette à distance (en option)

Mise en oeuvre :

- L'environnement de développement QT Creator en lien avec le terminal mobile

Configuration :

- Le terminal mobile, la liaison Bluetooth

Réalisation :

- Les diagrammes UML
- L'IHM du module
- Le code source de l'application

Documentation :

- Le dossier technique et les documents relatifs au module
- Un guide de mise en route et d'utilisation du module

Étudiant EC : GRAS Thomas

Module de télémétrie

- Acquisition de l'état de charge de la batterie
- Mesure de la vitesse de déplacement
- Mesure de la distance parcourue
- Mettre en forme les mesures
- Transmission des mesures via une liaison sans fil (Bluetooth)

Mise en oeuvre :

- L'environnement de développement Arduino et la trottinette électrique

Configuration :

- La liaison sans fil Bluetooth

Réalisation :

- Les diagrammes SysML
- Le code source
- Les schémas du module

Documentation :

- Le dossier technique et les documents relatifs au module
- Un guide de mise en route et d'utilisation du module

Étudiant EC 2 : HALLIER Kévin

Module d'assistance et d'anti-vol

- Mesure de l'inclinaison
- Mise en forme des mesures
- Immobilisation de la trottinette
- Alerte d'un risque de chute
- Communication avec le terminal mobile via une liaison sans fil Bluetooth

Mise en oeuvre :

- L'environnement de développement Arduino en lien avec le module de prévention de chute

Configuration :

- Le système d'immobilisation, la liaison Bluetooth

Réalisation :

- Les diagrammes SysML
- Le code source
- Les schémas du module

Documentation :

- Le dossier technique et les documents relatifs au module
- Un guide de mise en route et d'utilisation du module

Contraintes fonctionnelles et techniques

Matérielles

- Microcontrôleur ESP-WROOM-32
- Inclinomètre 3 axes GY-61 ADXL335
- Capteur de vitesse LTH301
- Abaisseur de courant LM2596HVS
- Capteur de courant ACS772
- Tablette samsung SM-T813
- Trottinette électrique SXT 1000 Turbo

Logicielles

- QT Creator 4.7.2
- Système d'exploitation de la tablette : Android 7.0 Api 24
- Bouml 7.8
- Compilateur C++ pour Windows : MinGW 5.3.0 32 bits for C++
- Compilateur C pour Windows : MinGW 5.3.0 32 bits for C
- Compilateur C++ pour Android : Android GCC (C++, arm-4.9)
- Compilateur C pour Android : Android GCC (C, arm-4.9)

Module de visualisation - SY Somphon (étudiant 3)

Objectifs

Mon rôle dans ce projet consiste à développer une application pour terminal mobile sous **Android** permettant de visualiser en temps réels les données de fonctionnement de la trottinette.

Le terminal mobile communiquera avec le système embarquée de trottinette électrique via une liaison sans-fil **Bluetooth**.

Ressources matérielles et logicielles

- Langage de programmation : C++ / QML / JavaScript
- IDE : QT Creator 5.7.2
- Framework : QT 5.11.2
- Compilateur C++ pour Windows : MinGW 5.3.0 32 bits for C++
- Compilateur C pour Windows : MinGW 5.3.0 32 bits for C
- Compilateur C++ pour Android : Android GCC (C++, arm-4.9)
- Compilateur C pour Android : Android GCC (C, arm-4.9)
- Système d'exploitation : Windows 10 64 bits
- Système d'exploitation de la tablette : Android 7.0 Api 24
- Modèle de la tablette : Samsung SM-T813
- Version de java : Java 8
- Rédaction des documents : Google Document
- Espace de travail collaboratif pour les documents : Google Drive
- Gestionnaire de version pour le code : subversion (RiouxSVN)

Ressources Documentaire

- http://tvaira.free.fr/
- https://doc.qt.io/qtcreator/index.html

Qt pour Android

Qt est un ensemble de bibliothèques orientées objet et développées en **C++**. Qt peut aussi être vu comme *framework* lorsqu'on l'utilise pour concevoir des interfaces graphiques ou que l'on conçoit l'architecture de son application en utilisant les mécanismes des signaux et slots par exemple. Qt permet de faire du développement multiplateforme : Windows / Linux / Mac / **Android** / ios.

Qt Creator est un environnement de développement intégré pour Qt.

Sous **Android**, la majorité des applications sont codées en Java, mais grâce à Qt, il est possible de coder en **C++** la partie “métier” de l’application et de réaliser son aspect graphique en **QML**.

QML (*Qt Modeling Language*) est un langage déclaratif qui permet de décrire les interfaces utilisateur en termes de composants visuels et d’interaction entre elles. Les fichiers écrits en QML portent l’extension **.qml**. QML intégre aussi un environnement **JavaScript** pour assurer l’interaction de l’interface graphique.

Les cas d'utilisation

L’acteur de ce système est l’utilisateur et le système est la trottinette électrique.

L’utilisateur peut :

- Visualiser sur son terminal mobile les données de fonctionnement de la trottinette (pourcentage de batterie, vitesse, inclinaison, distance parcourue depuis le démarrage).

- Avoir la possibilité de saisir un trajet, de le visualiser sur une carte et pourra s’assurer de la faisabilité du trajet en terme d’autonomie.

- Verrouiller sa trottinette via son terminal mobile.

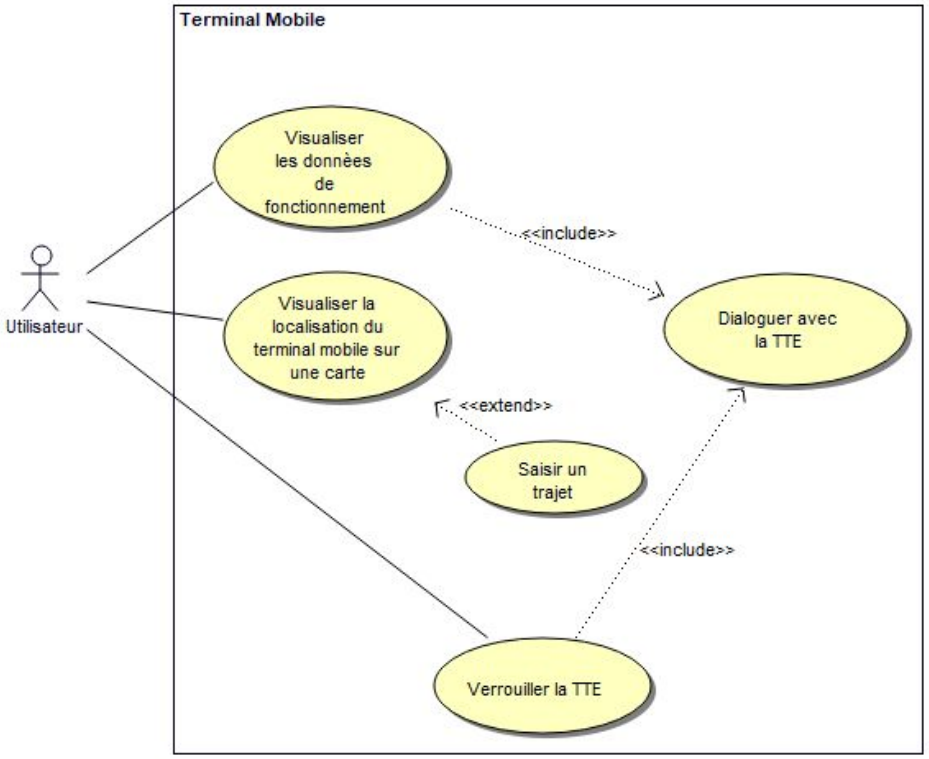
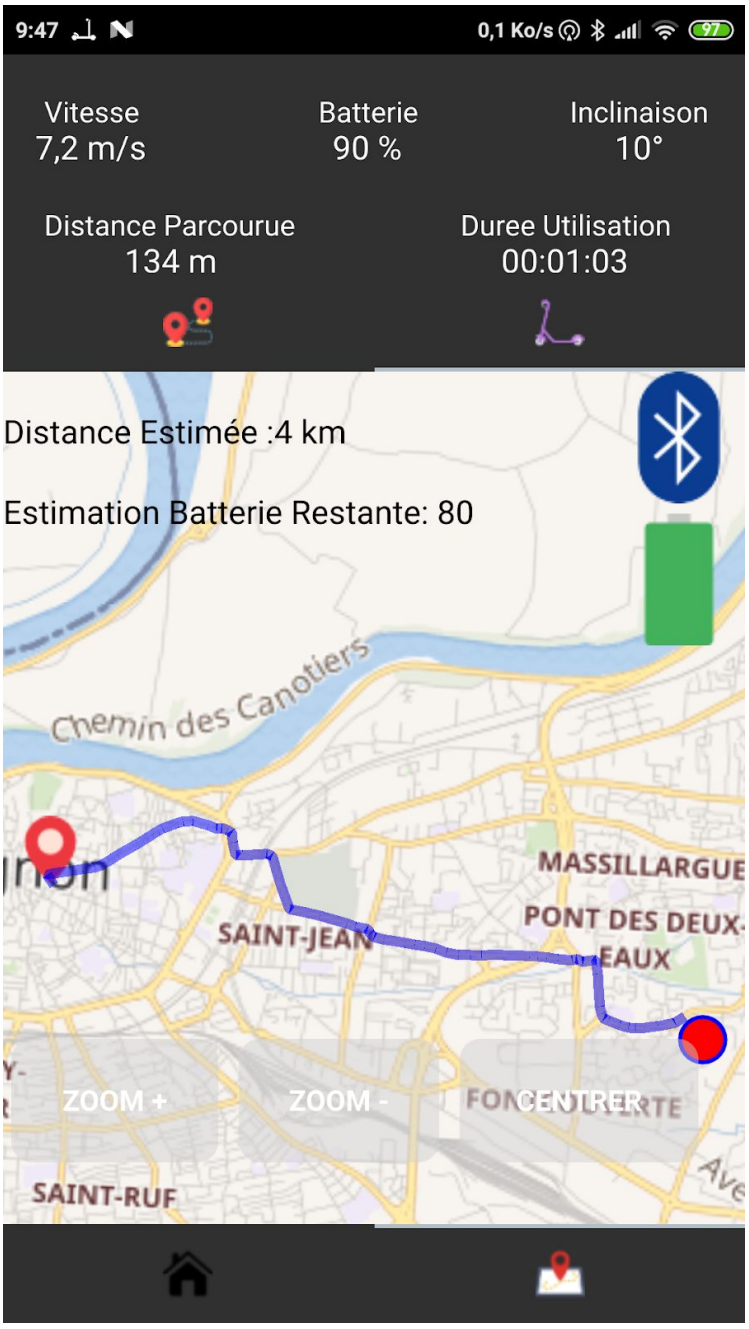


Diagramme de cas d'utilisation

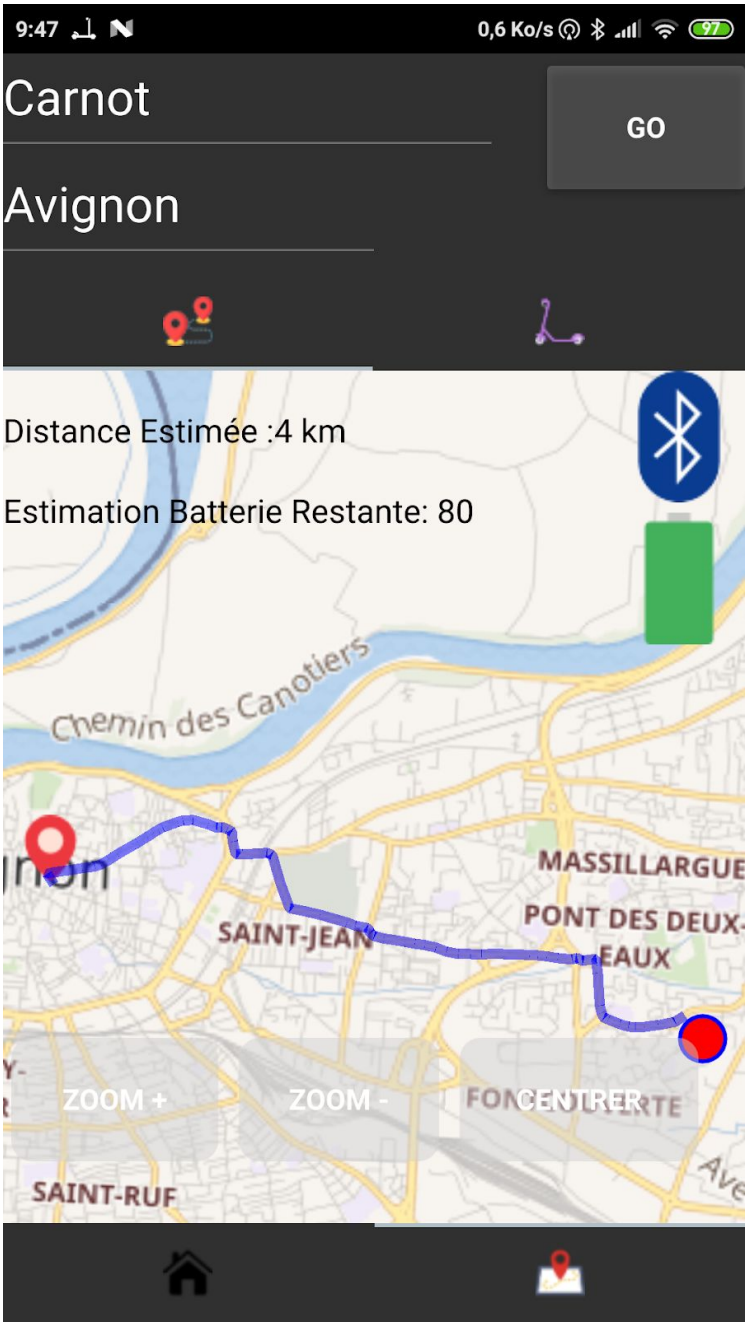
Prototypage et maquette de l'IHM

Comme nous pouvons le voir sur la capture d'écran ci-dessous, la page principale de l'application possède deux onglets. Un onglet **données** qui permet à l'utilisateur de visualiser les données en temps réel de la trottinette (Vitesse , Inclinaison , Pourcentage de charge , Distance Parcourue , Durée D'utilisation).



Capture d'écran de l'IHM

Le second onglet **itinéraire** va permettre à l'utilisateur de saisir un trajet et de le voir sur une carte. Il pourra aussi voir la distance totale du trajet ainsi qu'une estimation du pourcentage de batterie restante à l'arrivée.



Capture d'écran de l'IHM

Plan de test de validation

Etudiant 1 : SY Somphon

Désignation	Résultat attendu	Oui/Non
Visualiser la vitesse de la trottinette en temps réel	On voit sur l'IHM la vitesse de la trottinette en temps réel	
Visualiser le pourcentage de batterie de la trottinette en temps réel	On voit le pourcentage de batterie de la trottinette en temps réel	
Visualiser un trajet entre notre position actuelle et une position désirée	On voit sur la carte le trajet entre notre position actuelle et une destination	
Visualiser la distance du trajet	On visualise la distance du trajet en km ou m	
Visualiser une estimation de la batterie restante à la fin du trajet	On visualise une estimation de la batterie restante à la fin du trajet	
Visualiser en cas de risque de chute un avertissement	On visualise une image nous avertissant d'un éventuelle risque de chute avec la direction	
Verrouillage de la trottinette	La trottinette est verrouillé , on ne pas accélérer .	

Répartition des tâches

Cas d'utilisation	Priorité	Itération
Structure de l'IHM	Haute	1
Géolocaliser la TTE	Moyenne	1
Visualiser le trajet	Basse	1
Visualiser les données de fonctionnement de la TTE	Haute	2
Visualiser l'autonomie pour un parcours	Moyenne	3
Verrouiller la trottinette à distance	Moyenne	3

Diagramme de gantt

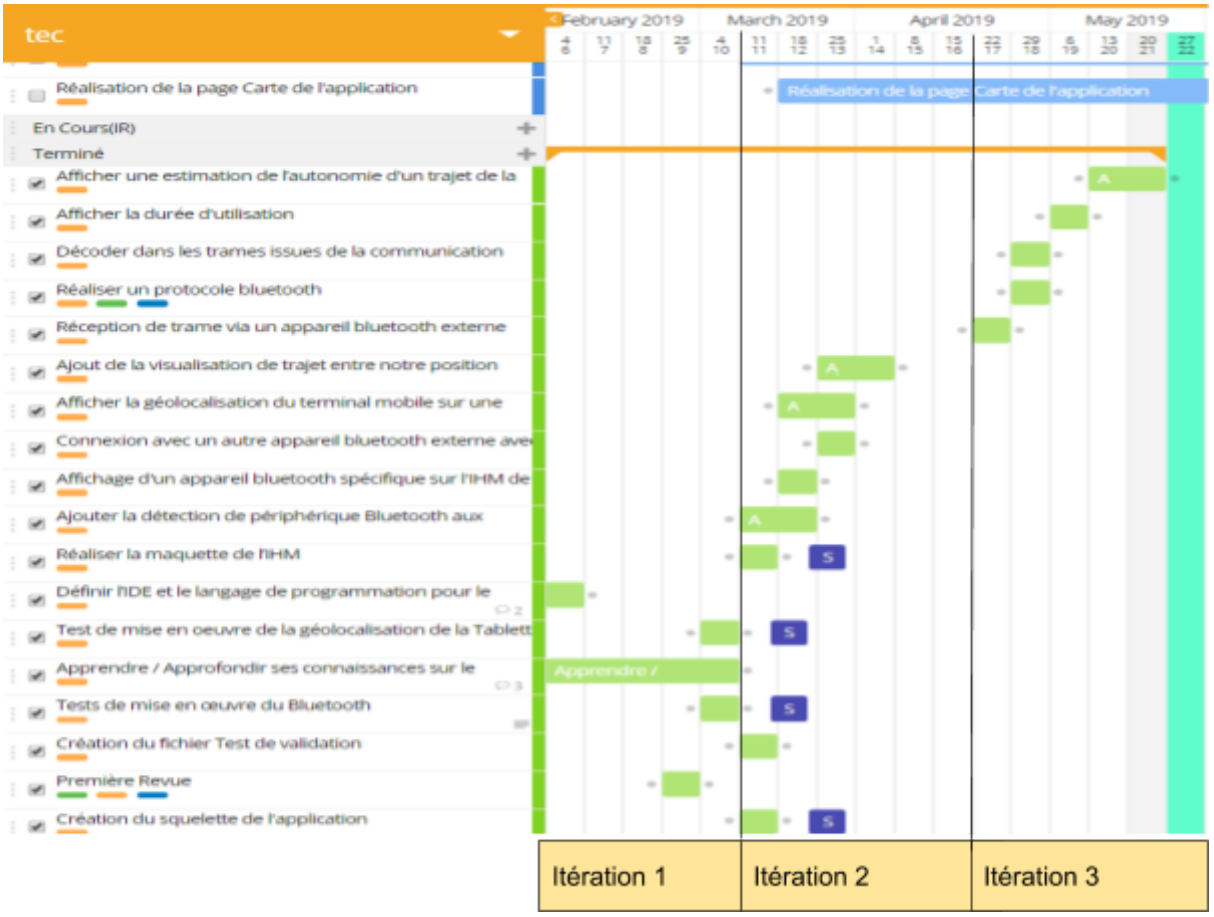
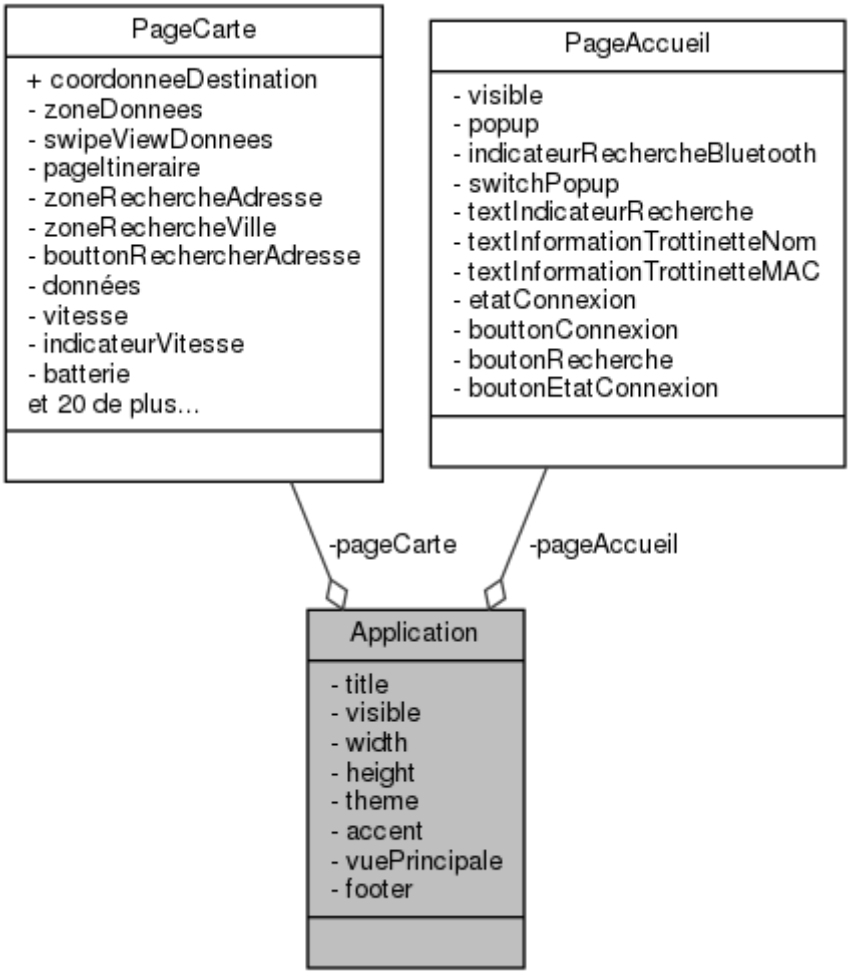


Diagramme de gantt

Réalisation de l'application

Pour l'application développée avec le framework Qt, la partie programmation est divisée en deux parties, la partie graphique qui concerne l'IHM en **QML**, et la partie fonctionnalités (connexion bluetooth, traitement de trames, chronomètre) en **C++**.

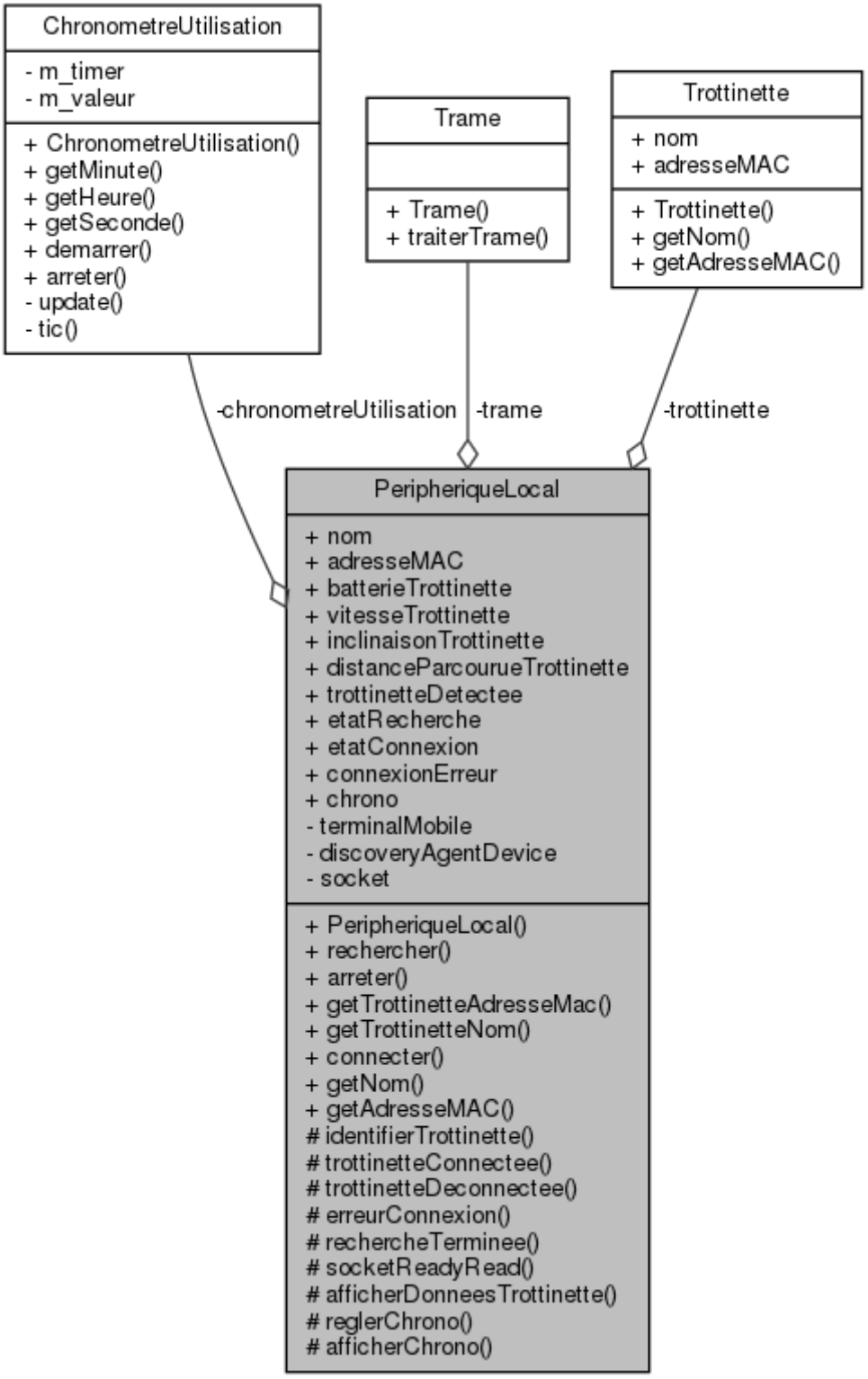
Diagramme de classe (partie QML)



La page **Accueil** permettra de rechercher, détecter et se connecter à une trottinette en Bluetooth.

La page **Carte** permettra de visualiser la géolocalisation de la trottinette sur une carte ainsi que les données de fonctionnement.

Diagramme de classe (partie C++)



La partie C++ de l'application est composée de quatre classes . La classe principale est la classe **PeripheriqueLocal** , cette classe est responsable de la liaison entre la partie fonctionnalité (C++) et la partie IHM(QML) . Cette classe contient toutes les fonctionnalité responsable de la détection et de la connexion à la trottinette électrique connectée .
Elle est composée de trois autre classe :

- **ChronometreUtilisation** : cette classe s'occupera de la mesure du temps d'utilisation de la TEC à partir de la connexion avec le terminal mobile
- **Trame** : cette classe s'occupera du traitement des trames reçues par la TEC
- **Trottinette** : cette classe contient les données de la TEC , (adresse , nom)

Structure du langage QML

L'interface homme-machine de l'application est générée à partir de fichiers QML . Au niveau du code , un fichier QML est composée en deux parties , la partie importation de module et la partie de déclarations d'objets . Ces objets sont hiérarchisés de la façon suivante :

```
import QtQuick 2.9
import QtQuick.Controls 2.2

Page {
visible: true
...
Popup {
          id: popup
          x: (parent.width - width) / 2
          y: (parent.height - height) / 2
          width: Screen.desktopAvailableWidth
          height: Screen.desktopAvailableHeight / 2
}
}
```

Qt Quick Controls

Qt Quick Controls est un module qui fournit un ensemble de composants pour créer des interfaces très complètes :

- Application fenêtrée : ApplicationWindow, MenuBar, StatusBar, ToolBar, Action, Page, …
- Navigation et vues : ScrollView, SplitView, StackView, TabView, TableView, TreeView, SwipeView, …
- Contrôles : Button, CheckBox, ComboBox, Label, ProgressBar, Slider, SpinBox, BusyIndicator, …
- Menus : Menu, MenuItem et MenuSeparator …

Positionnement des objets

Pour positionner les objets , QT Quick a ajouté en plus du système traditionnel (Grille , rangée ,colonne) le système d'anchors. Ce système est simple , chaque objet peuvent-être imaginé avec 7 lignes invisible .

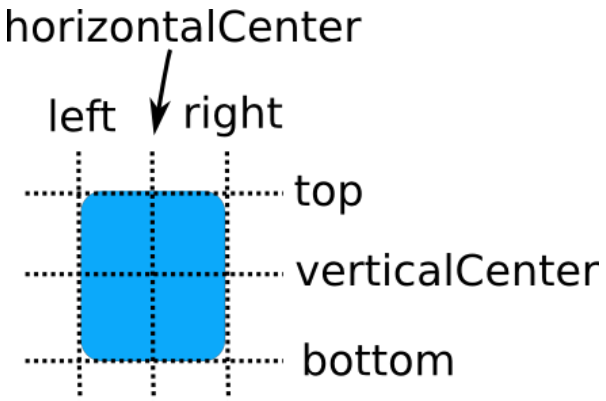
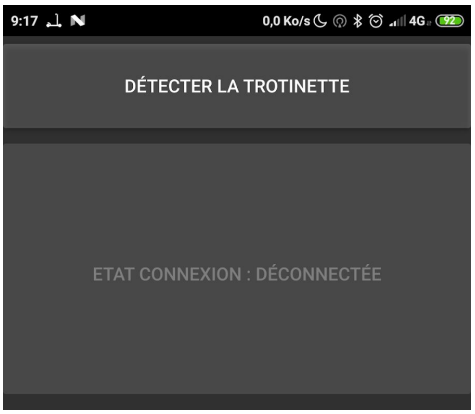


Schéma illustrant le système d'anchor

Le système d'anchors vas permettre ensuite de positionner les objets en fonction de leur ligne d'anchors .

```
    Button{
        id: boutonRecherche;
        width: Screen.desktopAvailableWidth
        height: Screen.desktopAvailableHeight / 8
        anchors.horizontalCenter: parent.horizontalCenter // Centre le
bouton en fonction de son parent
        // ....
    }
    Button{
        id: boutonEtatConnexion;
        anchors.top : boutonRecherche.bottom // Le bord du haut du bouton
Etat Connexion se trouvera en dessous du bord du bas du bouton Recherche
        topPadding:
        anchors.horizontalCenter: parent.horizontalCenter
        // ...
    }
```



Capture d'écran de l'IHM

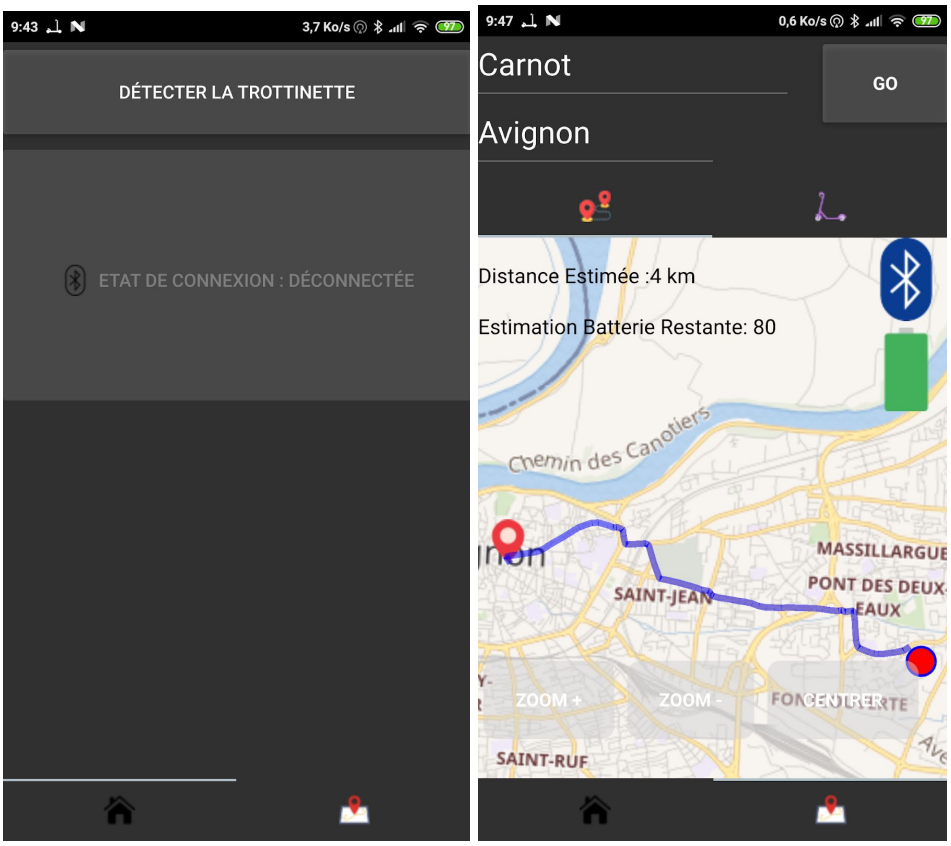
SwipeView

Le **SwipeView** est composé de **Page**s et permet à l'utilisateur de naviguer entre ses pages. Dans ce projet, le changement de page en slidant sera bloqué et on le réalisera en appuyant sur des boutons. Cela est possible en associant un **SwipeView** avec un **TabBar**. Ce qui donne le code suivant :

```
SwipeView {
        id: vuePrincipale
        interactive: false  // Désactive l'interaction du swipe
        currentIndex: onglets.currentIndex // Affiche la page en
fonction de l'index du tabBar
        anchors.fill: parent
        PageAccueil { // Première page du SwipeView
            id: pageAccueil
        }
        PageCarte { // Deuxième page du SwipeView
            id: pageCarte
        }
    }
    footer: TabBar { // TabBar
        id: onglets
        width: parent.width
        anchors.horizontalCenter: parent.horizontalCenter
        currentIndex: vuePrincipale.currentIndex
        TabButton { // Premier Bouton du TabBar
            text: "Accueil"
```

```
        }
        TabButton { // Deuxième Bouton du TabBar
            text: "Carte"
        }
    }
}
```

Suite à cela on obtient au niveau de l'application une barre en bas de l'application qui va permettre de changer de page.



Capture d'écran de l'IHM

Les Boutons (button)

Sous QML, lorsqu'un utilisateur active un bouton , un signal clicked est émis. Pour pouvoir réaliser des actions en fonctions de l'action de l'utilisateur il faudra alors se connecter au signal émis. De ce faite lorsque le signal sera émis, une action sera commise. Au niveau QML, la connexion à un signal se fera de la manière suivante : préfixe "**on**" + nom du signal avec une majuscule "**Clicked**".

```
Button{
    id: boutonRecherche;
    // ...
    onClicked: {
        popup.open();
    }
}
```

Association d'un objet C++ au document principal QML

La liaison QML avec un objet C++ se fait dans le **main.cpp** de l'application :

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "peripheriquelocal.h"

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.rootContext()->setContextProperty("peripheriqueLocal", new
PeripheriqueLocal());
    engine.load(QUrl(QStringLiteral("qrc:/Application.qml")));
    return app.exec();
}
```

Explications :
`engine.rootContext()->setContextProperty()` , Ajoute un objet c++ au contexte QML que l'on instanciera dynamiquement
`engine.load()`, Charge le document principal QML dans l'application

Dans le fichier QML , on pourra dorénavant appelée la classe **PeripheriqueLocal** via l'objet dynamique **peripheriqueLocal**.

Appel d'une méthode d'un objet C++ à partir de QML

Pour appeler une méthode **C++** à partir d'un fichier **QML** , il faudra ajouter la macro `Q_INVOKABLE` dans une classe qui hérite de `QObject` pour pouvoir l'appeler à partir de **QML.**

```
class PeripheriqueLocal : public QObject
{
public:
   explicit PeripheriqueLocal(QObject *parent = nullptr);
   Q_INVOKABLE void rechercher();
}
```

Une fois la macro ajoutée, il suffit juste d'appeler la méthode de l'objet **peripheriqueLocal** à partir de QML.

```
Switch {
         text: "Rechercher Trotinette"
         onClicked:{
            peripheriqueLocal.rechercher();
         }
}
```

Accéder au attribut d'une classe C++ sans accesseurs à partir de QML

Pour accéder aux attributs d'une classe C++ qui hérite de QObject sans accesseurs depuis QML, il suffit d'ajouter une propriété à la classe avec la macro Q_PROPERTY de la manière suivante :

```
Q_PROPERTY(QString nom MEMBER nom NOTIFY trottinetteTrouvee)
```

Le Bluetooth

Qu'est-ce que le Bluetooth ?

Le Bluetooth est une technologie sans fil qui permet à des appareils électroniques d'échanger des données à courte distance. Le Bluetooth utilise des ondes radio sur la bande de fréquences de 2,4 GHz (la même que le Wifi) pour connecter des équipements entre eux (smartphone, enceinte, oreillette, objets connectés…) afin de leur permettre d'échanger des données ou des fichiers (documents, photos, musique…).

Pourquoi utiliser le Bluetooth dans notre projet ?

Le module ESP32-WROOM-32 dispose du Wifi et du Bluetooth .Comparé au Wifi , le Bluetooth est plus adaptée pour transmettre des petites quantités de données . Ce qui est idéal pour récupérer les données de nos capteurs pour les afficher sur notre IHM. Au niveau de la distance , comme le terminal mobile est poser sur le support téléphone de la TEC , la portée de 100 mètre du bluetooth est suffisant .
Le tableau ci-dessous est un comparatif entre le Bluetooth et le Wifi avec des données issue de la datasheet de l'ESP32-WROOM-32.

	Bluetooth	Wifi
Protocole	Bluetooth v4.2	802.11 b/g/n
Bit rate	1 Mbps	150 Mbps
Fréquence	2.4 GHz ~ 2.5 GHz	2.4 GHz ~ 2.5 GHz
Distance	50-100m	~100 m
Consommation d'énergie	Faible	Élevée
Besoin d'internet	Non	Oui

Ce tableau montre bien que pour notre projet , le Bluetooth est plus intéressant car les données échangées ne sont pas volumineux donc 1 Mbps est suffisant , et comme le module est alimentée directement via la batterie de la trottinette , gagner en consommation est important donc le bluetooth répond parfaitement à nos critères .

Détection du Bluetooth

La détection d'appareil Bluetooth se fera dans la partie C++ de l'application.
Pour détecter les autres appareils bluetooth il faut tout d'abord disposer niveau physique d'un appareil disposant d'une interface bluetooth.
Avant de commencer à coder, il faut ajouter les modules suivants dans le fichier **.pro** :

```
QT  += qml quick bluetooth
```

On va ensuite créé deux classes, une pour le périphérique local que l'on nommera peripheriqueLocal et une autre pour la trottinette. Dans la déclaration de la classe peripheriqueLocal on va importer les déclarations des bibliothèques suivantes :

```
#include  <QBluetoothLocalDevice>
#include  <QBluetoothDeviceInfo>
#include  <QBluetoothDeviceDiscoveryAgent>
```

QBluetoothLocalDevice permet l'accès au Bluetooth de l'appareil.

QBluetoothDeviceInfo permet d'avoir accès à certaines informations telles que le nom du périphérique Bluetooth et son adresse.
QBluetoothDeviceDiscoveryAgent permet à partir d'un appareil bluetooth de détecter les appareils Bluetooth aux alentours.

Dans la déclaration on va ajouter en attribut privé, un objet de type **QBluetoothLocalDevice** afin d'avoir accès au Bluetooth de l'appareil et un pointeur de type **QBluetoothDeviceDiscoveryAgent** qui va permettre de découvrir les périphériques bluetooth à proximité.

Dans le constructeur de la classe **peripheriqueLocale** on va vérifier si l'appareil dispose d'une interface Bluetooth avec la condition suivante :

```
if(!peripheriqueLocal.isValid())
{
     qCritical("Bluetooth  désactivé  !");
     return;
}
```

Si le périphérique possède le Bluetooth alors on va l'activer avec la commande suivante :

```
peripheriqueLocal.powerOn();
```

Et on va pouvoir initialiser les attributs du périphérique local telles que le nom , l'adresse mac comme ceux-ci :

```
nom = peripheriqueLocal.name();
adresse = peripheriqueLocal.address().toString();
```

On va initialiser l'objet de type **QBluetoothDeviceDiscoveryAgent** dans le constructeur par défaut et faire les connexions signaux/slots pour la recherche de périphériques Bluetooth :

```
 //  Slot  pour  la  recherche  de  périphériques  Bluetooth
    connect(discoveryAgentDevice,SIGNAL(deviceDiscovered(QBluetoothDeviceInfo)),
this, SLOT(identifierTrottinette(QBluetoothDeviceInfo)));
    connect(discoveryAgentDevice, SIGNAL(finished()), this,
SLOT(rechercheTerminee()));
```

Lorsque le **discoveryAgentDevice** découvre un périphérique , il déclenchera la méthode **identifierTrottinette(QBluetoothDeviceInfo)** qui contient les informations sur les périphériques découverts.

```
void PeripheriqueLocal::identifierTrottinette(const QBluetoothDeviceInfo &info)
{
    //qDebug() << Q_FUNC_INFO << info.name() << info.address().toString();
    if(info.name() == "tec")
    {
        qDebug() << Q_FUNC_INFO << "Trottinette" << info.name() <<
```

```
info.address().toString();
        trottinette = new Trottinette(info.name(),info.address().toString());
        trottinetteDetectee = true;
        discoveryAgentDevice->stop();
        emit trottinetteTrouvee();
    }
```

Ici, on identifiera la trottinette à partir du nom du périphérique bluetooth externe.

Connexion à un autre appareil bluetooth

Pour la connexion avec un autre appareil Bluetooth on va utiliser la classe **QBluetoothSocket** pour cela on va déclarer un pointeur de **QBluetoothSocket** dans la classe **peripheriqueLocal** et on l'initialisera lorsque l'on fera la connexion. Ici on utilisera un Bouton pour réaliser la connexion. Donc pour connecter deux périphériques bluetooth entre eux il faut seulement posséder l'adresse Bluetooth de l'autre machine.

```
Q_INVOKABLE void connecter());
```

Ce qui donnera :

```
void PeripheriqueLocal::connecter()
{
    //...
    if (!socket)
    {
        socket = new QBluetoothSocket(QBluetoothServiceInfo::RfcommProtocol);
        connect(socket, SIGNAL(connected()), this,
SLOT(trottinetteConnectee()));
        connect(socket, SIGNAL(disconnected()), this,
SLOT(trottinetteDeconnectee()));
    }
    if(etatConnexion == false)
    {
        QBluetoothUuid uuid = QBluetoothUuid(QBluetoothUuid::SerialPort);

socket->connectToService(QBluetoothAddress(this->getTrottinetteAdresseMac()),
uuid);
        socket->open(QIODevice::ReadWrite);
    }
}
```

Ici on voit que tout d'abord on a initialisé un objet `QBluetoothSocket` en donnant le protocole de communication ici le protocole est le **Rfcomm**. Le protocole Rfcomm est un protocole de transport permettant l'émulation d'un port série RS-232.
On va ensuite se connecter à l'autre appareil en se connectant à un service via le socket en spécifiant l'adresse. Le service est spécifié grâce à l'uuid.

Maintenant on pourra communiquer avec l'autre appareil bluetooth via le socket .

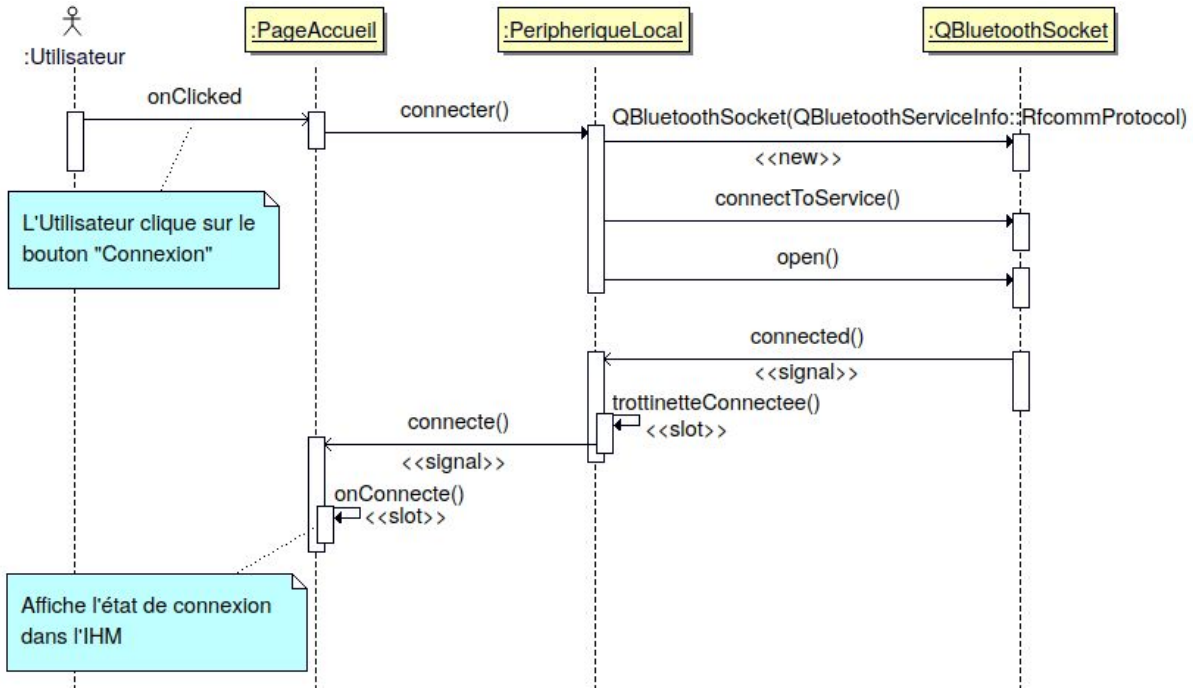


Diagramme de séquence <<Connexion à la trottinette >>

Sur ce diagramme de séquence on peut voir que c'est l'action clic sur le bouton connexion qui va appeler la fonction connecter de la classe PeripheriqueLocal .

Protocole de communication

Trame d'Acquisition

Les trames d'acquisition sont transmises par l'ESP32 de la TTE et contiennent toutes les données issues des capteurs du système.

Exemple de trame

```
TEC;10;80;25;5;rcd\n
```

Les différents champs de la trame

```
Début : TEC
Délimiteur : ;
Fin de la trame : \n
```

5 champs de valeurs qui correspondent aux différentes acquisitions issues des capteurs du système de la TTE et un champ qui correspond à la prévention d'un risque de chute .

Voici l'ordre des champs :

- L'inclinaison : en degrés , min -90 , max 90 , précision : à l'unité
- Batterie : en pourcentage , min: 0 , max 100 , précision : à l'unité
- Vitesse : en m/s , min:0 , max:100 , précision : arrondi au centième
- Distance parcourue : en km , min: 0 , max ?? , arrondi au centième
- Risque de chute : rcd → Risque de chute à droite
  rcg → Risque de chute à gauche
  acg→ Aucun risque de chute

Réception de trame à travers une communication Bluetooth

Une fois la connexion effectuée entre les deux appareils, le socket de la classe **peripheriqueLocal** émettra le signal **readyRead** à chaque fois que des données sont disponibles.
Sur ce signal, nous allons connecter un slot qui va émettre à son tour les données disponibles avec un signal pour le traitement de trames.

Dans PeripheriqueLocal.cpp :

```
connect(socket, SIGNAL(readyRead()), this, SLOT(socketReadyRead()));
void PeripheriqueLocal::socketReadyRead()
{
   QByteArray donnees;

   while (socket->bytesAvailable())
   {
       donnees += socket->readAll();
   }
   emit trameRecue(QString(donnees));
}
```

Traitement de trame

Le traitement est effectué dans la classe `Trame` qui est dédiée à cette tâche. Suite au signal émis contenant la trame par la méthode socketReadyRead() de la classe PeripheriqueLocal , nous allons connecter ce signal à un slot de la classe Trame qui va traiter la trame et renvoyer les données des capteurs à la classe PeripheriqueLocal.

Dans PeripheriqueLocal :

```
connect(this,SIGNAL(trameRecue(QString )), trame ,SLOT(traiterTrame(QString)));
 // Envoie de la trame à la classe Trame
connect(trame,SIGNAL(donneesTrottinette(QString,QString,QString,QString)), this,
SLOT(afficherDonneesTrottinette(QString , QString , QString , QString)));
// Envoie des données issue de la trame à la classe PeripheriqueLocal
connect(trame,SIGNAL(risqueDeChute(QString)),SLOT(afficherRisqueDeChute(QString)));
// Envoie le risque de chute à la classe PeripheriqueLocal
```

Le traitement de trame est réalisé par le slot **traiterTrame(QString Trame)**.
Comme les champs des trames sont délimités par **';'** , nous avons utilisé la fonction **section** de la classe **QString** de Qt.

La fonction **section** fonctionne de la manière suivante il faut saisir d'abord le caractère délimiteur, sa position parmi les autres délimiteurs et ensuite la position du dernier délimiteur. La fonction retournera alors la section du String du premier délimiteur jusqu'au dernier délimiteur choisi en s'incluant lui aussi.

Prenons comme exemple la trame suivante `TEC;10;80;25;5;rcd\n`

```
inclinaison = trame.section(';',1,1);
```

Ici la fonction section retournera **10**. On a indiqué 1 dans le premier champ pour préciser que nous voulons les données à partir du premier délimiteur , et 1 dans le dernier champ pour indiquer que nous voulons uniquement les données situées entre le premier et le deuxième délimiteur.

Affichage des données de fonctionnement sur l'IHM

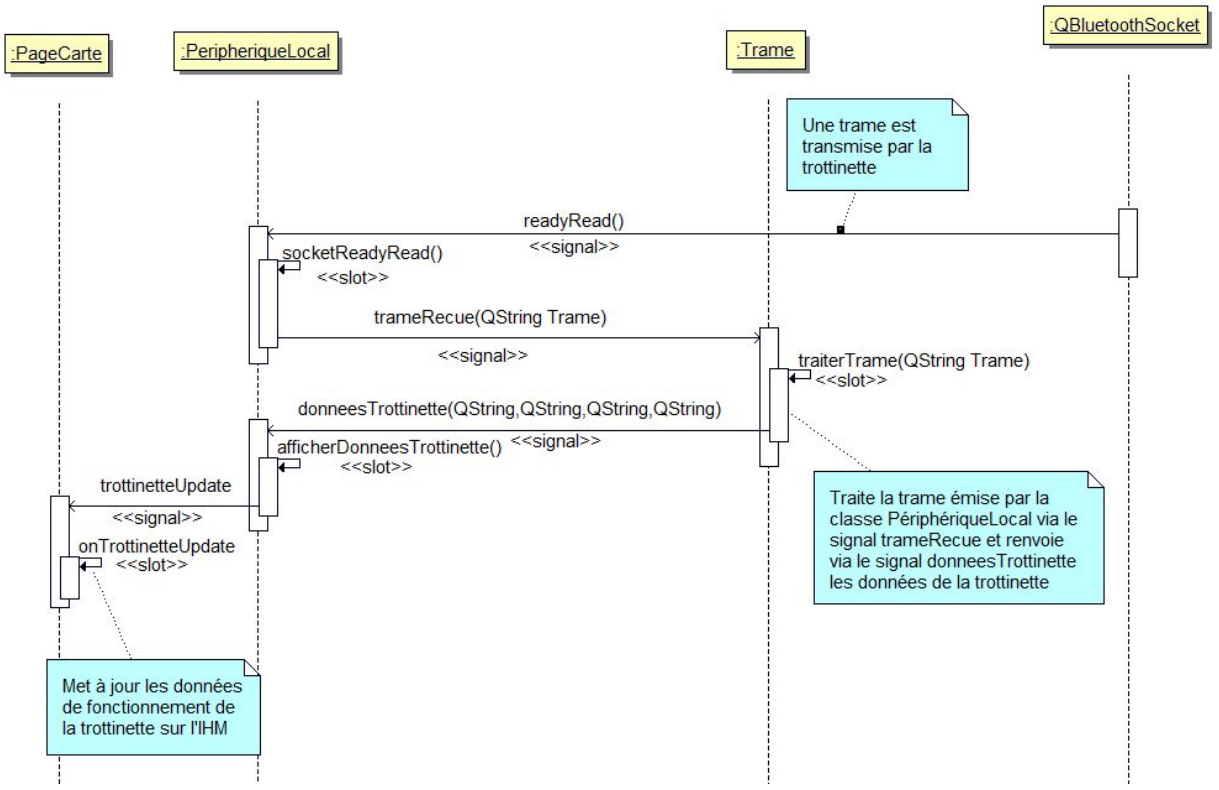


Diagramme de séquence << Affichage des données de fonctionnement de la trottinette >>

```
Connections {
   target: peripheriqueLocal
   onChronoUpdated:
   {
       indicateurdureeUtilisation.text = peripheriqueLocal.chrono
   }

   onTrottinetteUpdate:
   {
       indicateurVitesse.text=peripheriqueLocal.vitesseTrottinette + "
m/s"

indicateurInclinaison.text=peripheriqueLocal.inclinaisonTrottinette + "
°"
       indicateurBatterie.text=peripheriqueLocal.batterieTrottinette + "
%"
```

```
        indicateurDistanceParcourue.text =
peripheriqueLocal.distanceParcourueTrottinette + " m"
  }
}
```

Voici le code issue de l'onglet données de la partie **QML**, on réceptionne le signal émis par la classe **périphériqueLocal** et lorsque le signal est émis, on modifie l'affichage des données de fonctionnement avec les attributs de la classe **périphériqueLocal**.

Le slot en QML se fait de la manière suivante , il y a d'abord le **Connections** pour indiquer que c'est une connexion et ensuite on doit saisir le signal qu'on attend. La saisie d'un signal se fait de la manière suivante : on fixe le mot-clef '**on**' suivie du nom du signal avec une lettre majuscule , ici "**TrotinetteUpdate**".

Afficher le temps d'utilisation de la trottinette sur l'IHM

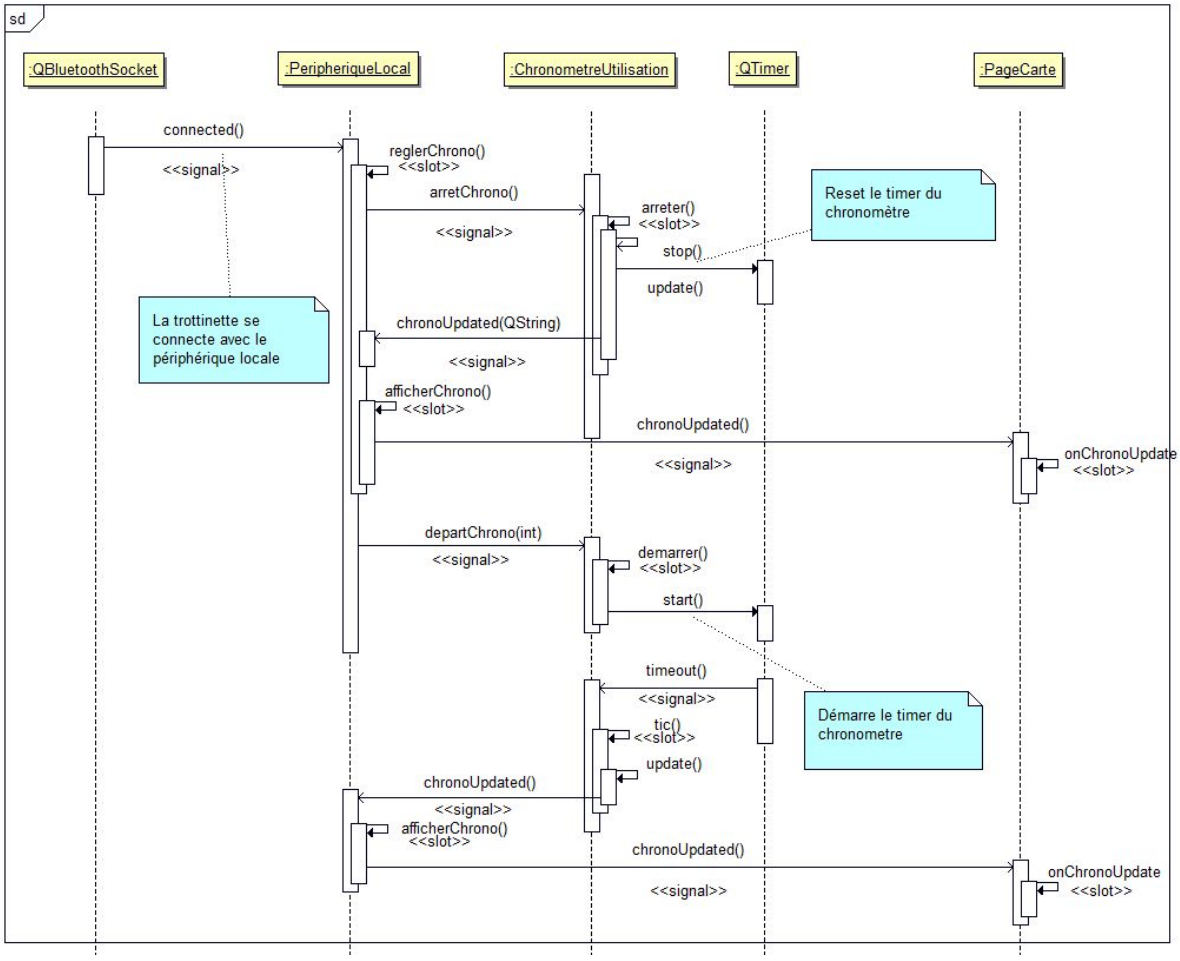


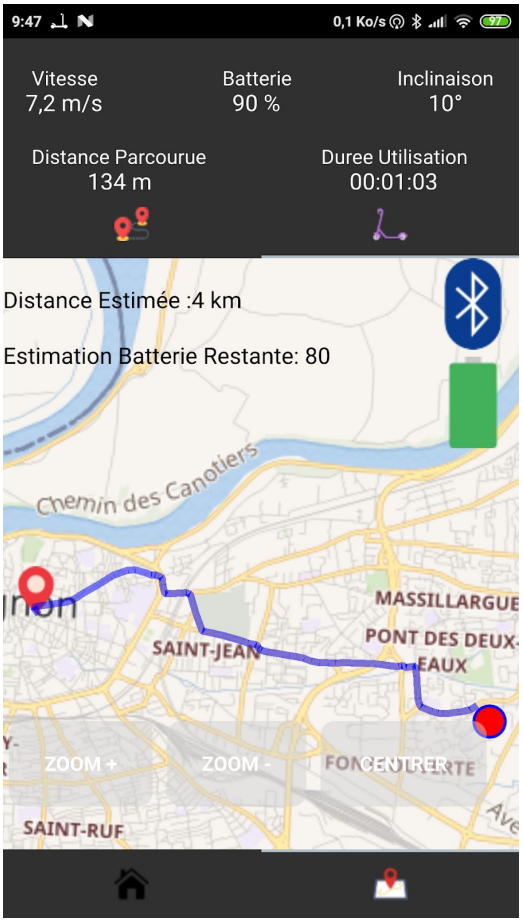
Diagramme de séquence << Afficher le temps d'utilisation de la trottinette >>

Sur ce diagramme de séquence, on peut voir que la connexion entre l'appareil locale et la trottinette est le déclencheur des fonctionnalitées liées au chronomètre. Lors de la connexion , la classe **PeripheriqueLocal** va émettre le signal **reglerChrono()** qui va émettre à son tour deux signaux, le signal **arretChrono()** pour reset le chronomètre et le signal **departChrono(int)** qui va recevoir en argument la vitesse du compteur en ms afin de démarrer le chronomètre.

Dans le fichier **ChronometreUtilisation.cpp**

```
void ChronometreUtilisation::arreter()
{
    m_valeur = 0 ;
    m_timer->stop();
    update();
    qDebug() << "ChronometreUtilisation::arreter()";
}
```

La variable **m_valeur** est la valeur du compteur de tic de l'horloge et **m_timer** le timer de l'horloge. Voici le rendue finale de l'affichage des données dans l'IHM.



Capture d'écran de l'onglet données de la partie de l'IHM

Affichage du risque de chute sur l'IHM

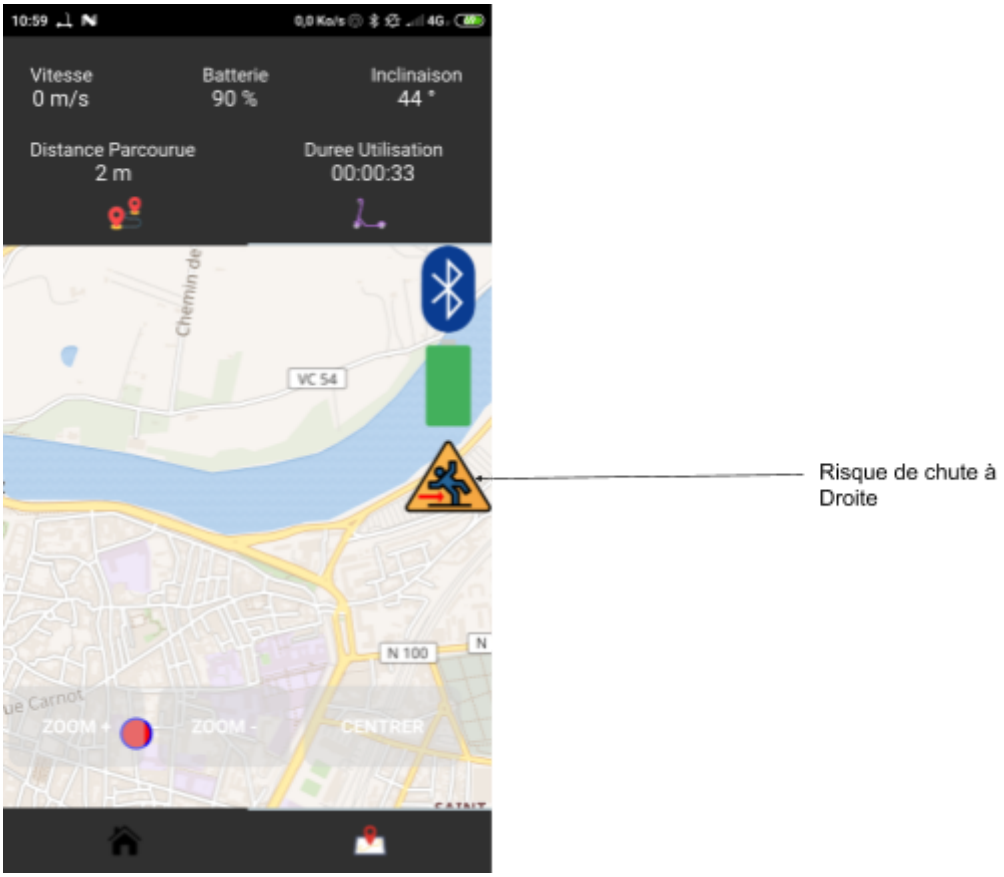
Pour indiquer à l'utilisateur la présence d'un risque de chute à gauche ou à droite , nous avons décidé d'afficher sur l'écran une icône indiquant la direction de la chute. Lorsque le risque de chute est nulle , il n'y a en conséquence aucune icône concernant le risque de chute visible.

```
onRisqueDeChuteUpdate:
{
    if(peripheriqueLocal.risqueDeChute == "rcd")
    {
        indicateurRisqueDeChute.visible=true
        indicateurRisqueDeChute.source="src/icons/32x32/rcd.png"
    }
    else
    {
       if(peripheriqueLocal.risqueDeChute =="rcg")
       {
           indicateurRisqueDeChute.visible=true

indicateurRisqueDeChute.source="src/icons/32x32/rcg.png"
       }
       else
       {
           indicateurRisqueDeChute.visible=false
       }
    }
}
```

L'objet **indicateurRisqueDeChute** est une Image.

```
Image {
    id: indicateurRisqueDeChute
    visible: false
    source: ""
    cache:false
...
}
```



Affichage de la carte sur l'IHM

L'affichage de la carte est réalisée directement dans la partie graphique de l'IHM , c'est à dire la partie QML. Pour cela nous avons utilisée le module **QtLocation** fourni par le framework QT. Le module **QtLocation** dispose du type **Map** ainsi que de plusieur type pouvant interagir avec des objets de type **Map**. Concernant l'affichage de la carte via un objet de type **Map** , nous avons besoins d'un fournisseur de carte pour pouvoir l'afficher dans l'IHM . De ce fait , Qt a prévu le coup en laissant la possibilité au développeur de saisir le fournisseur de carte à travers la propriété Plugin du type **Map**. Il va falloir au préalable déclarer un objet Plugin avec tous les informations concernant le fournisseur de carte. Voici un exemple de la déclaration d'un objet Plugin :

```
        Plugin {
            id: mapPlugin
            locales: "fr_FR"
            name: "osm" // OpenStreetMap
            PluginParameter { name: "osm.geocoding.host"; value:
"https://nominatim.openstreetmap.org" }
        }
```

On voit ici que le fournisseur de carte est **OpenStreetMap**.

Une fois le plugin déclaré, il nous suffit juste de déclarer l'objet de type **Map** en précisant le plugin.

```
Map {
            id: map
            anchors.fill: parent
            plugin: mapPlugin
            center: QtPositioning.coordinate(43.95, 4.8167)
            minimumZoomLevel: 5
            zoomLevel: 15
}
```

Affichage de notre localisation sur la carte de l'IHM

Le module **QtLocation** fournit des objets pour interagir avec la carte. Pour pouvoir voir en temps réel notre position sur la carte, nous allons associer le module **QtLocation** pour l'affichage et le module **QtPositioning** pour tout ce qui est en rapport avec la localisation.

Le module **QtPositioning** fournit plusieur type concernant la localisation dont le type **PositionSource**. Le type **PositionSource** fournit la position actuelle du système.

```
    PositionSource {
        id: src
        updateInterval: 1000
        active: true
        preferredPositioningMethods: PositionSource.AllPositioningMethods
        onPositionChanged: {
            var coord = src.position.coordinate;
            console.log("Coordinate:", coord.longitude, coord.latitude);
        }
    }
```

Dans notre cas, nous allons mettre à jour la position actuelle de l'appareil toutes les secondes.

Au niveau de l'affichage, nous allons utilisé le type **MapQuickItem** pour afficher sur la carte une image en fonction des coordonnées de l'appareil.

```
        Map {
            id: map
            /...
            MapQuickItem{
```

```
                id:indicateurPosition
                sourceItem: Rectangle {width: 24; height: 24; color:
"red"; border.width: 2; border.color: "blue"; smooth: true; radius: 15;}
                coordinate: src.position.coordinate
            /...
             }
```

Saisir un trajet

Pour la suite, les fonctionnalitées liées à la saisie d'un trajet et au calcul d'un itinéraire entre un point A et un point B seront dépendant d'un fournisseur de map en ligne, dans notre cas ce sera **OpenStreetMap**. Le type **GeocodeModel** du module **QtLocation** permet d'obtenir via une adresse ses coordonnées géographique (latitude , longitude , altitude) et vice-versa. Niveau code, la propriété **query** du type **GeocodeModel** pourra contenir l'adresse en **String** ou un objet de type **Address**, si nous voulons obtenir les coordonnées géographiques à partir d'une Adresse ou directement un type Coordinate pour obtenir l'adresse en fonction des coordonnées géographiques. Nous allons donc un **GeocodeModel** qui va détenir les information liée à l'adresse de destination .

```
GeocodeModel {
      id: adresseRecherche
      plugin: mapPlugin
}
```

Pour cela nous avons décidé, afin de limiter les erreurs lors de la saisie de l'adresse par l'utilisateur de diviser l'adresse en deux parties , le nom de la rue / avenue et le nom de la ville.

```
TextField {
            id: zoneRechercheAdresse
            placeholderText: "Rue ou Avenue"
}

TextField {
            id: zoneRechercheVille
            placeholderText: "Ville"
}
```

Et pour que l'utilisateur puisse affirmer la saisie de son trajet nous avons ajouté un bouton qui va saisir dans le **GeocodeModel**, l'adresse de la destination afin d'obtenir les coordonnées géographique.

```
Button {
                    id: bouttonRechercherAdresse
              /...
                    onClicked:
                    {
                        if(zoneRechercheVille.text != "")
                        {
                            adresseRecherche.query =
zoneRechercheAdresse.text+","+zoneRechercheVille.text+" ,France"
                            adresseRecherche.update()
                        }
                    }
                }
```

Une fois la localisation du **GeocodeModel**, un signal **localisationChanged()** est émis.

Visualiser un trajet

Le type **RouteModel** du module **QtLocation** permet de récupérer des informations géographiques concernant les routes à partir d'un fournisseur de carte. On peut par exemple à partir de deux coordonnées géographiques, connaitre le trajet entre les deux points , la direction , et récupérer diverses informations liées au trajet. Les coordonnées géographiques des trajets seront stockées dans la propriété **query** qui est un **RouteQuery**. En plus des coordonnées géographiques, on peut dans le **RouteQuery** rajouter des informations sur le trajet , par exemple le type de véhicules ou encore l'optimisation du trajet (trajet plus chère , trajet plus rapide ,etc ...).

```
RouteModel {
        id: itineraire
        property bool actif: false
        plugin: mapPlugin
        autoUpdate: true
        query: RouteQuery {
            id: choixItineraire
            travelModes: RouteQuery.BicycleTravel
            routeOptimizations: RouteQuery.ShortestRoute
        }
```

Nous avons vu précédemment que lorsque l'on change la localisation d'un objet de type **GeocodeModel**, un signal est émis. Nous allons utiliser ce signal pour ajouter les coordonnées géographique de la destination, et de notre position actuelle dans le **RouteQuery**. Nous allons tout d'abord nous connecter au signal, vérifier que les coordonnées sont valides.

Si les coordonnées sont valides, nous allons saisir dans une variable coordonneeDestination les coordonnées géographiques de l'adresse de destination et nous allons aussi ajouter notre position actuelle et l'adresse de destination dans le **RouteQuery.**

```
GeocodeModel {
        id: adresseRecherche
        plugin: mapPlugin
        onErrorChanged: {
            console.log("<adresseRecherche> onErrorChanged GeocodeModel :
" + error + " - " + errorString)
        }
        onLocationsChanged: {
            if(error)
                console.log("<adresseRecherche> Erreur GeocodeModel : " +
error + " - " + errorString)
            if (count>=1)
            {
                if(get(0).coordinate.isValid)
                {
                    console.log("<adresseRecherche> : " +
get(0).address.text + "\n" + get(0).coordinate + "\n")
                    coordonneeDestination = get(0).coordinate
                    marqueurArrivee.coordinate = coordonneeDestination

                    choixItineraire.clearWaypoints()
                    choixItineraire.addWaypoint(src.position.coordinate)
                    choixItineraire.addWaypoint(coordonneeDestination)
                    map.update()
                }
            }
        }
    }
```

Le **get(0)** , ici est une fonction qui retourne un objet de type **Location** issue du query.
Maintenant que nous avons le trajet dans le **RouteModel**, il ne nous reste plus qu'à l'afficher sur la carte .
Nous allons utiliser un **MapItemView** pour l'affichage du trajet.

```
MapItemView {
    model: itineraire
    delegate: MapRoute {
        route: routeData
        line.color: "blue"
        line.width: 5
        smooth: true
```

```
        opacity: 0.5    }
}
```

Le **MapItemView** fonctionne de la manière suivante , on saisit les données que l'on souhaite représentée à travers le model, et ensuite on saisit la manière dont on souhaite représenter les données sur la carte. Ici on a représenté le trajet avec le type MapRoute qui affiche une route obtenue à travers un objet de type **RouteModel**. Pour avoir une visualisation de la destination , nous avons ajouté un objet **MapQuickItem** marqueurArrivee qui aura pour coordonnées les coordonnées de la destination que l'on a mis à jour dans **adresseRecherche** .

```
MapQuickItem {
                id: marqueurArrivee
                anchorPoint {
                    x: imageM.width/2
                    y: imageM.height
                }
                sourceItem: Column {
                    Image { id: imageM; source:
"src/icons/iconeDestination.png" }
                }
            }
```

Récupérer la distance d'un trajet

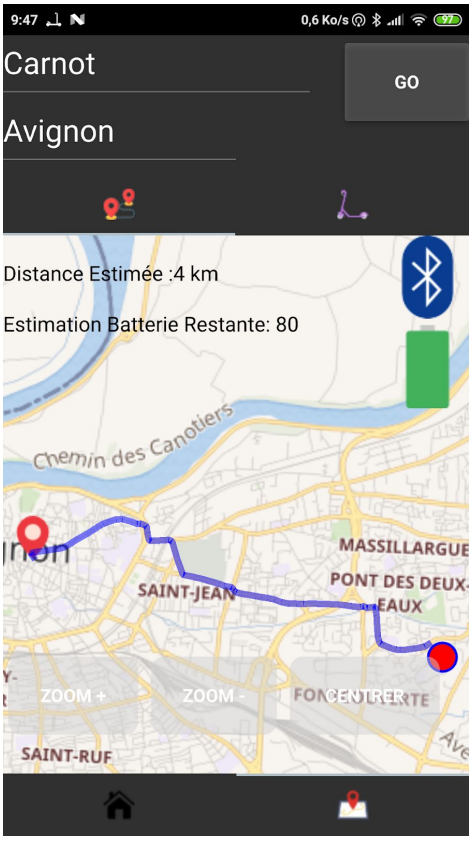
Pour extraire la distance d'un trajet stockée dans un **RouteModel**, nous avons créé une variable **totalDistance**. Le **RouteModel** dispose d'un objet de type **Route** contenant le trajet qu'on aura préalablement saisi. À partir d'un objet de type **Route**, on peut obtenir la distance d'un trajet en mètres et le temps de trajets en secondes. La fonction get() du type **RouteModel** retourne un objet de type **Route**. Donc à partir du **RouteModel**, on obtiendra la distance du trajet de la manière suivante :

```
RouteModel {
        id: itineraire
        property bool actif: false
        ...
        onStatusChanged: {
            console.log("RouteModel status : " + status)
            if (status == RouteModel.Ready)
            {
                if (count>=1)
                {
                    console.log("RouteModel : " + count)
                    actif = true
```

```
                totalDistance = itineraire.count == 0 ? "" :
Helper.formatDistance(itineraire.get(0).distance)

            }
    ...
        }
}
```

`Helper.formatDistance` va formater la distance en mètres si la distance est inférieur à 1000s mètres sinon en kilomètres. C'est une fonction JavaScript. On va ensuite utiliser la distance totale pour l'afficher et aussi déduire l'estimation du pourcentage de batterie restante pour un trajet.



Capture d'écran de l'onglet itinéraire de la partie carte de l'IHM

Tests de validation

Etudiant 1 : SY Somphon

Désignation	**Résultat attendu**	**Oui/Non**
Visualiser la vitesse de la trottinette en temps réel	On voit sur l'IHM la vitesse de la trottinette en temps réel	**Oui**
Visualiser le pourcentage de batterie de la trottinette en temps réel	On voit le pourcentage de batterie de la trottinette en temps réel	**Oui**
Visualiser un trajet entre notre position actuelle et une position désirée	On voit sur la carte le trajet entre notre position actuelle et une destination	**Oui**
Visualiser la distance du trajet	On visualise la distance du trajet en km ou m	**Oui**
Visualiser une estimation de la batterie restante à la fin du trajet	On visualise une estimation de la batterie restante à la fin du trajet	**Oui**
Visualiser en cas de risque de chute un avertissement	On visualise une image nous avertissant d'un éventuelle risque de chute avec la direction	**Oui**
Verrouillage de la trottinette	La trottinette est verrouillé , on ne peut pas accélérer .	Non

Annexes

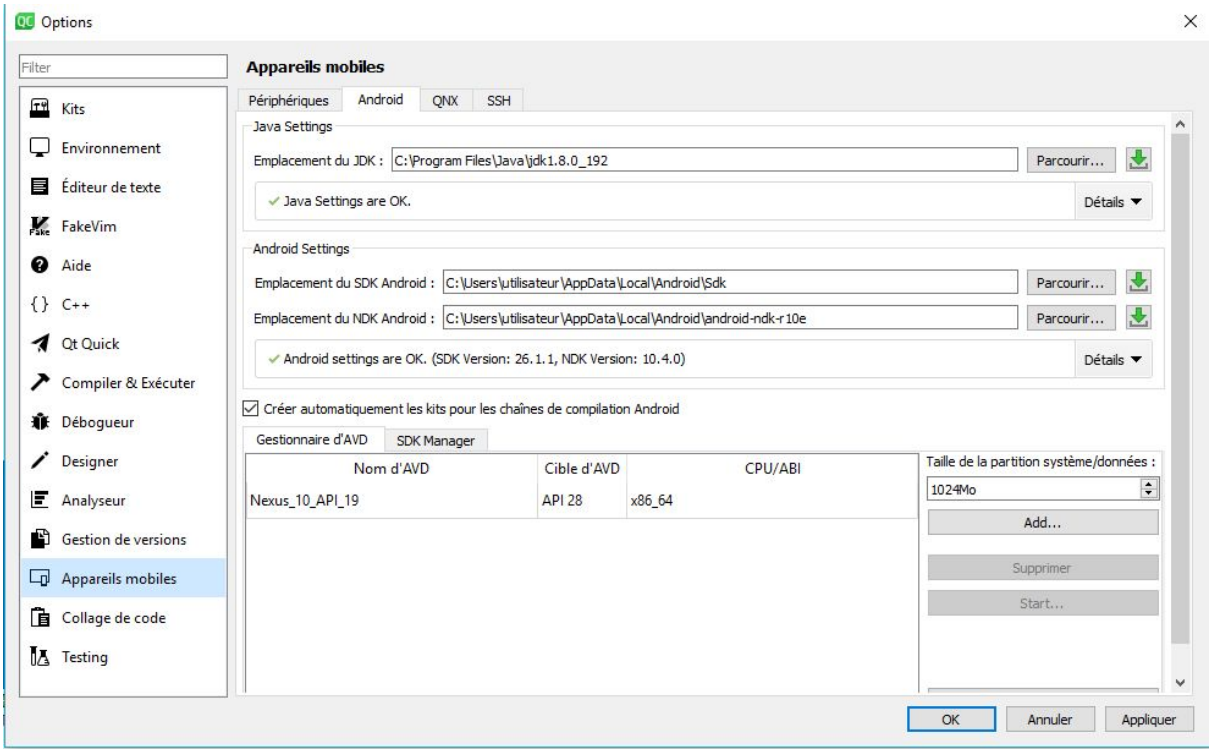
Guide de mise en route de l'application TEC

Configuration de QT Creator sous Windows pour le déploiement sur la plateforme Android.

Pré-Requis :

- Il faudra au préalable avoir installer QT Creator avec les composants responsables de la création d'un kit Android . (QT propose lors de l'installation , le choix sur les composants que l'on souhaite ajouter à l'IDE) .
- Il faut disposer d'Android SDK
- D'Android NDK en version r10e
- Java SE Development kit en version v8.
- Un système Android avec le mode développeur et ayant activée le débogage usb

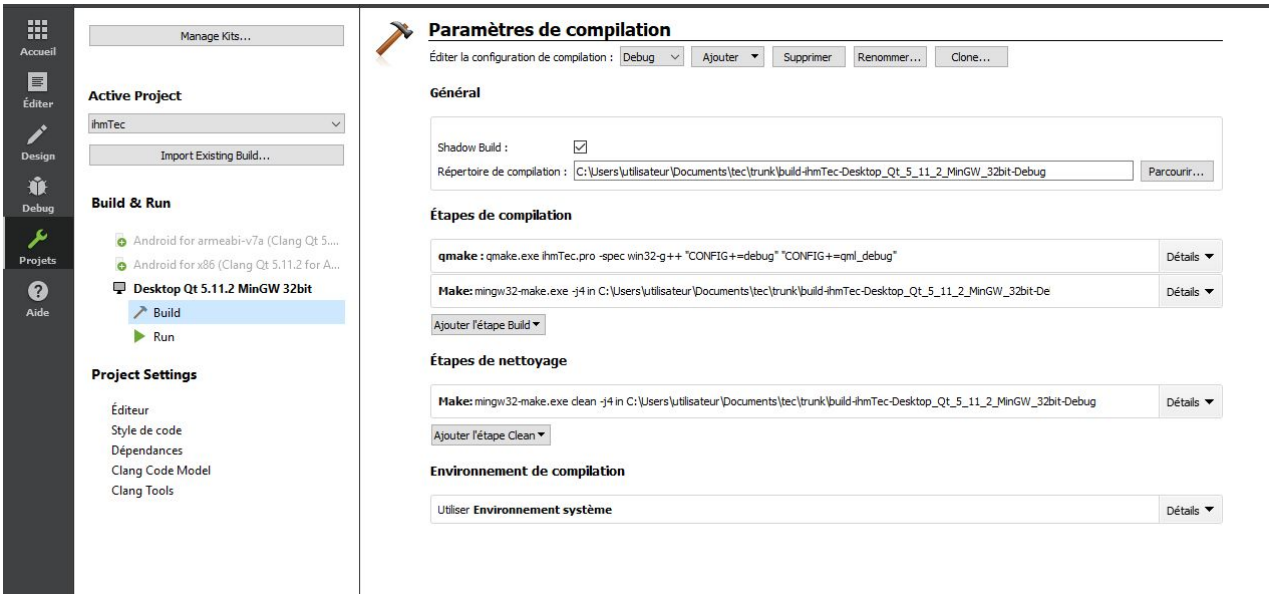
Une fois les outils obtenus , pour les configurée sur QT il suffit d'aller dans l'onglet Outils→ Options → Android . Et saisir l'emplacement des dossiers des outils obtenue précédemment.



Nous pouvons maintenant passer à la partie suivante qui concernent le déploiement de l'application sur une plateforme Android.

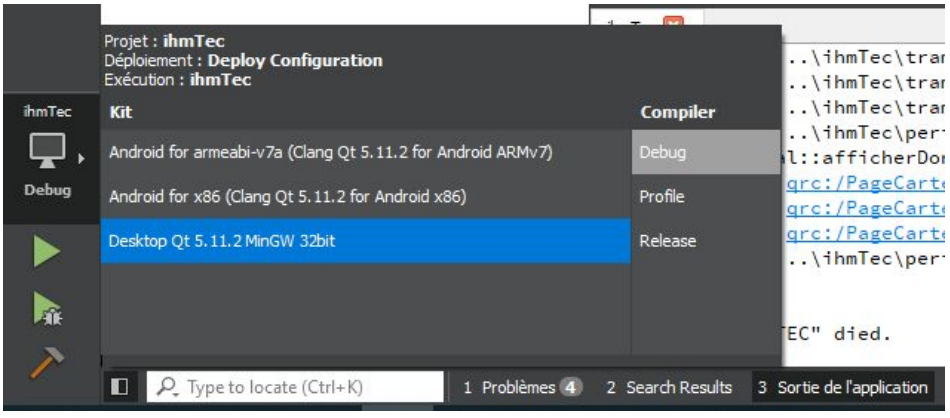
Ajout d'un Kit Android à un projet existant

Pour ajouter un Kit Android à un projet existant , il faudra au préalable ouvrir son projet dans QT Creator , ensuite il faudra ouvrir l'onglet projet située sur la gauche et si tout a été configuré correctement vous verrez apparaître les kits Android.

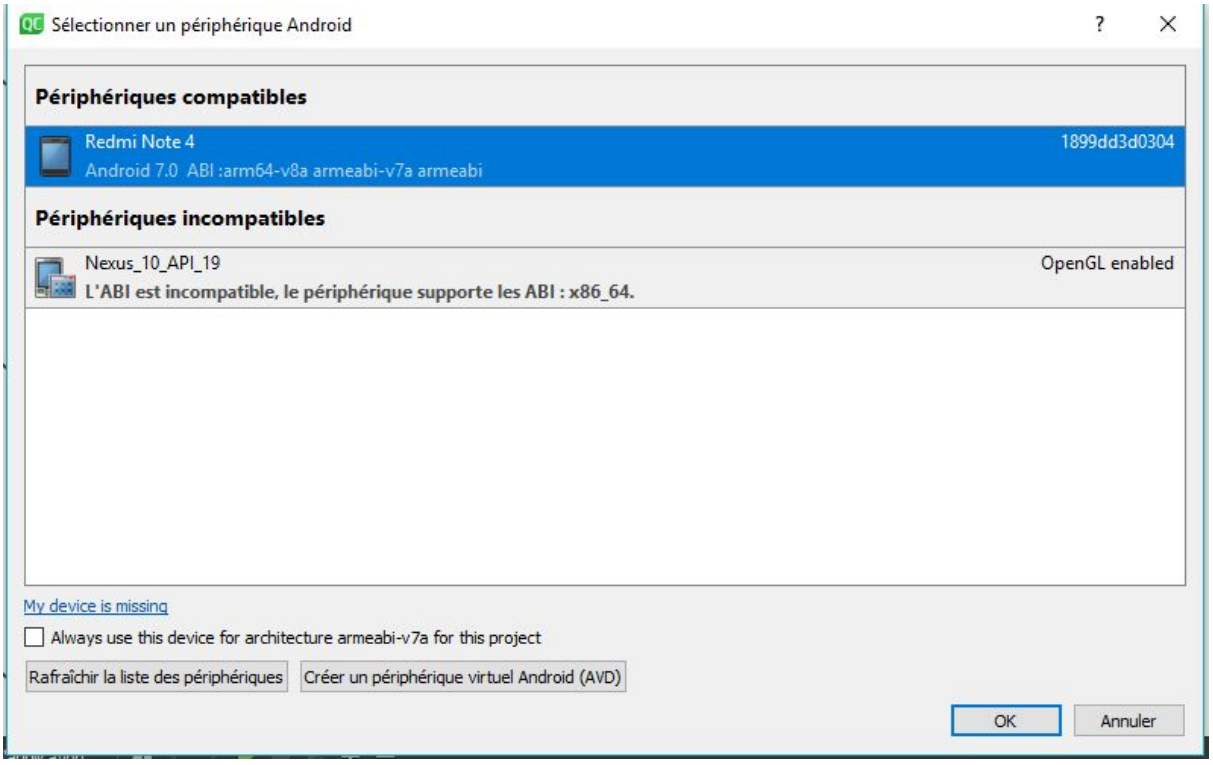


En cliquant sur les kits transparents , vous allez ajouter les kits dans votre projet.

Déployer l'application sur un kit Android

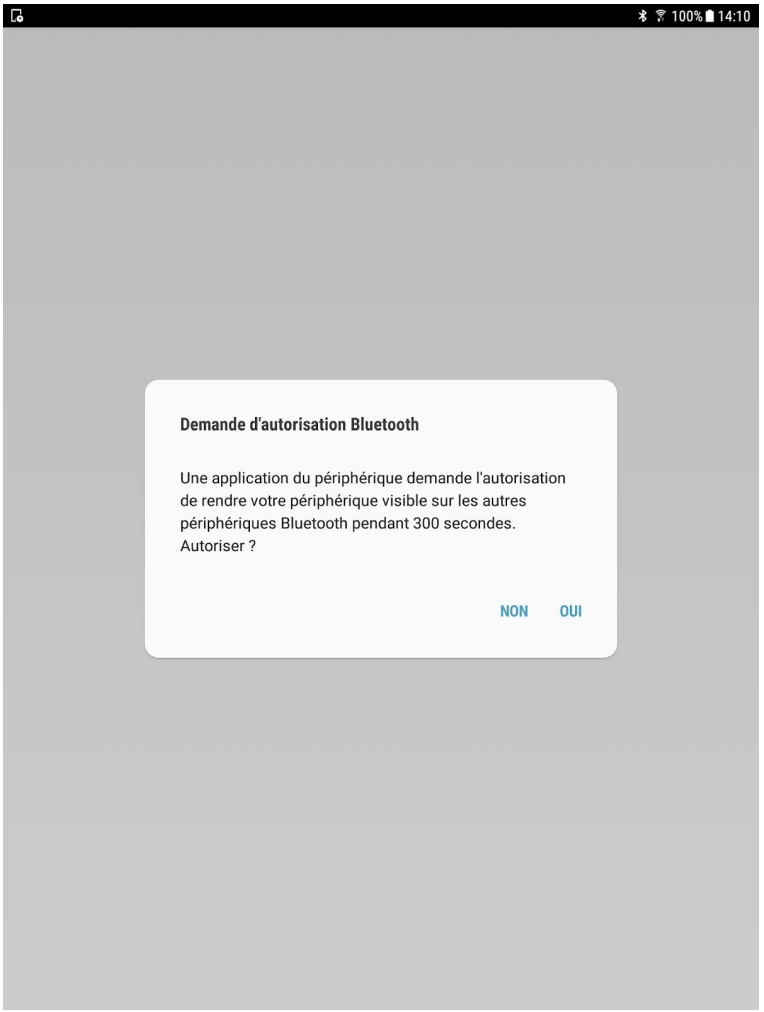


Avant la compilation , il faudra simplement saisir le kit correspondant à l'architecture système de votre terminal sous Android .
Lors de la compilation , une fenêtre vas apparaître et vas vous proposer une liste d'appareil sous Android qui est reliée à votre PC.

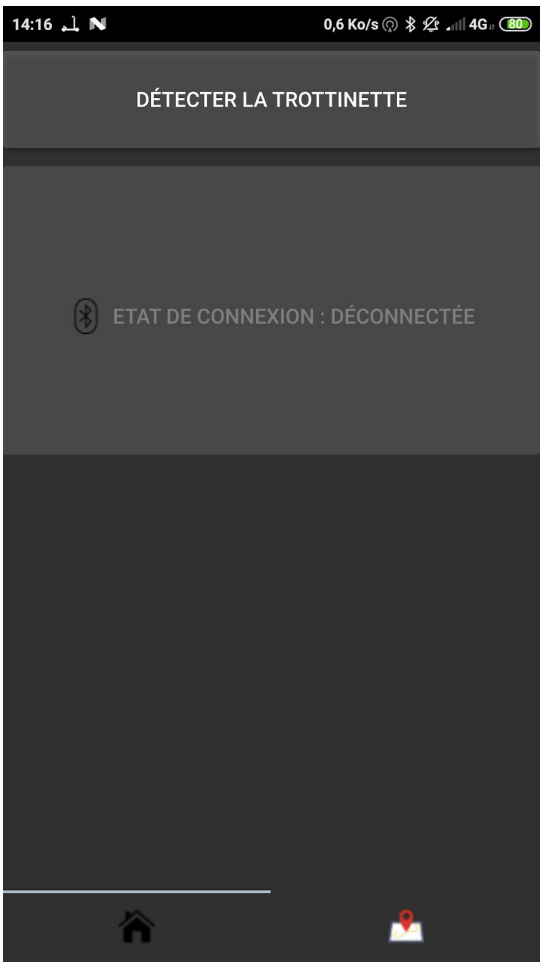

Sélectionner un périphérique Android
Périphériques compatibles
Redmi Note 4
1899dd3d0304
Android 7.0 ABI :arm64-v8a armeabi-v7a armeabi
Périphériques incompatibles
Nexus_10_API_19
OpenGL enabled
L'ABI est incompatible, le périphérique supporte les ABI : x86_64.
My device is missing
Always use this device for architecture armeabi-v7a for this project
Rafraîchir la liste des périphériques
Créer un périphérique virtuel Android (AVD)
OK
Annuler

Mise en route de l’application

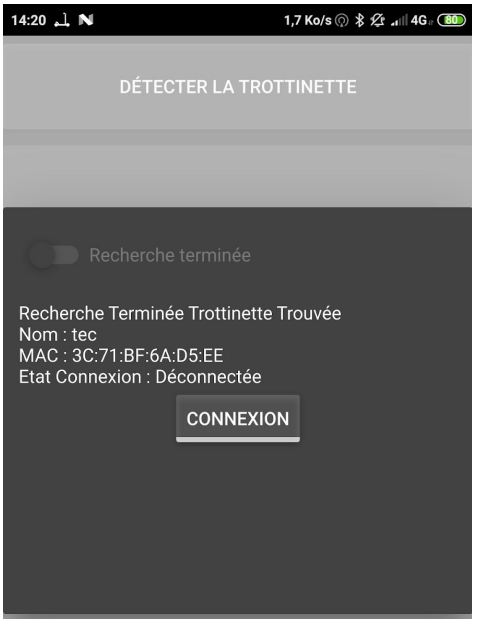
Lors du premier démarrage de l’application , un popup vas apparaître en vous demandant d’autoriser l’application à utiliser les services Bluetooth de l’appareil en refusant toutes les fonctionnalitée liées au bluetooth ne fonctionneront pas . Si jamais le bluetooth est désactivée , en appuyant sur Oui , le bluetooth vas s’activer automatiquement .



Voici la première page de l’application , cette partie de l’application vas permettre à l’utilisateur de se connecter à la trottinette. En appuyant sur le bouton détecter Trottinette , l’utilisateur vas ouvrir un popup.



En appuyant sur le bouton rechercher Trottinette , la recherche de la trottinette vas se lancer. Lorsque la trottinette vas être détectée , dans le popup vas s'afficher l'adresse de la Trottinette ainsi que son nom d'appareil .

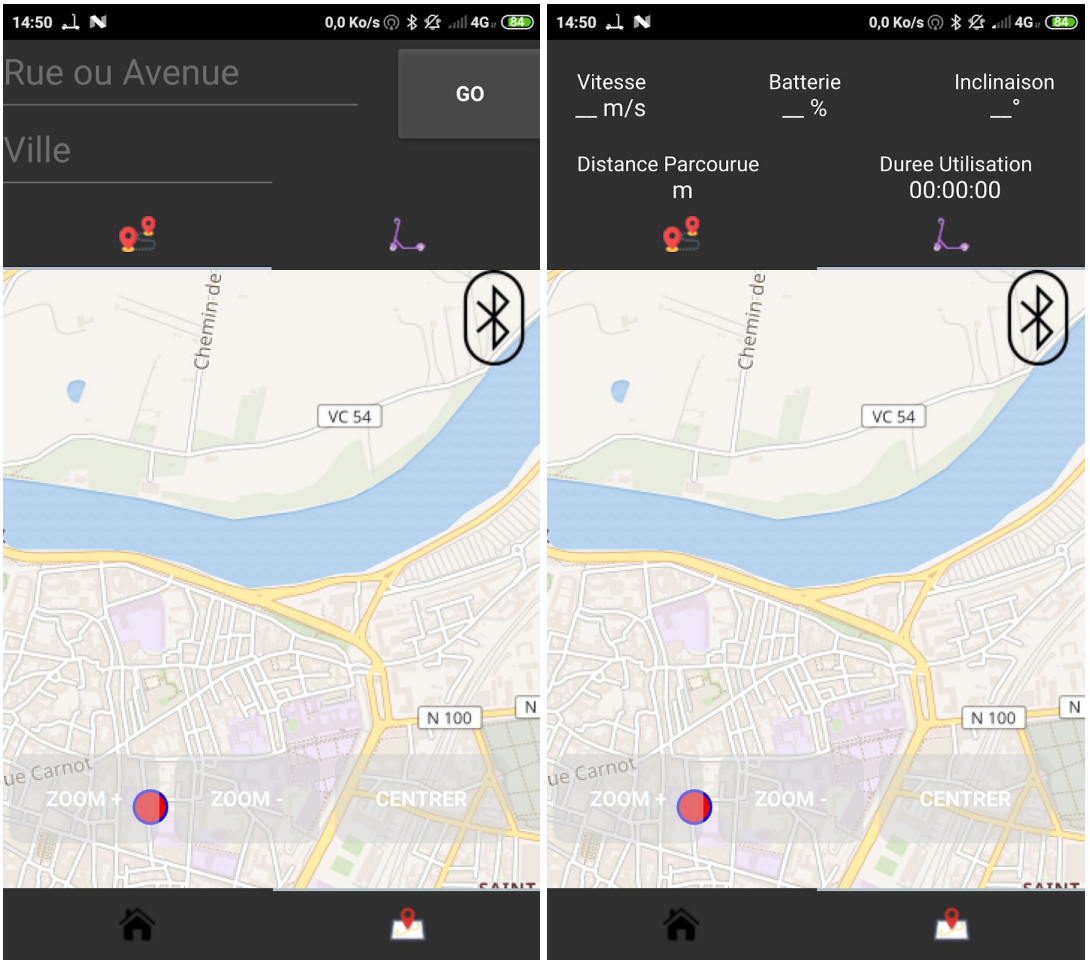


En appuyant sur le bouton connexion , l'appareil vas se connecter à la trottinette .

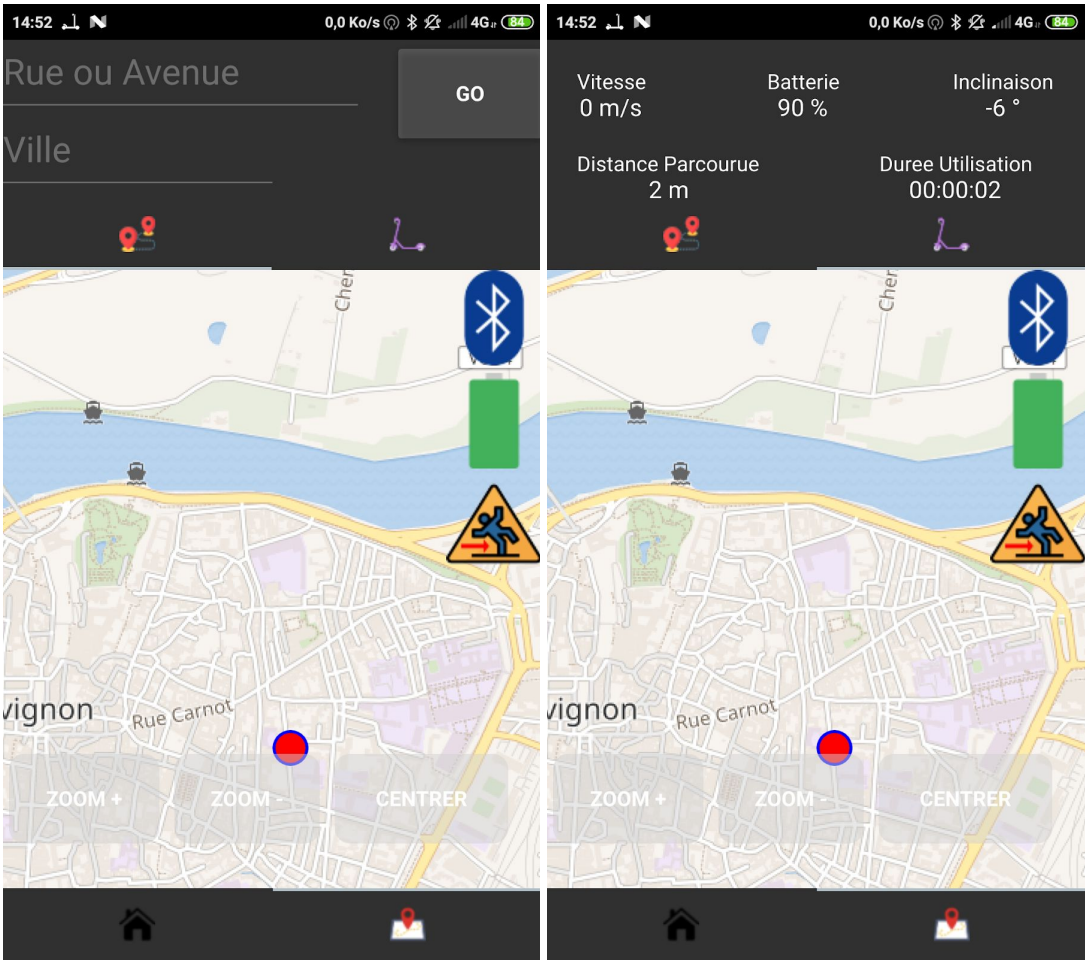
Passons à la partie Carte de l'application. Nous pouvons accéder à la partie Carte sans avoir besoin de se connecter à la trottinette.
Sur la partie Carte de l'application nous pouvons :

- Voir en temps réels notre position
- Voir les données de fonctionnement de la trottinette lorsque la trottinette est connectée avec notre terminal
- Pouvoir saisir un trajet et voir l'estimation de la batterie restante à la fin du trajet
- Avoir un avertissement en cas de risque de chute

Voici un aperçu de la partie Carte de l'application sans être connectée à la trottinette.

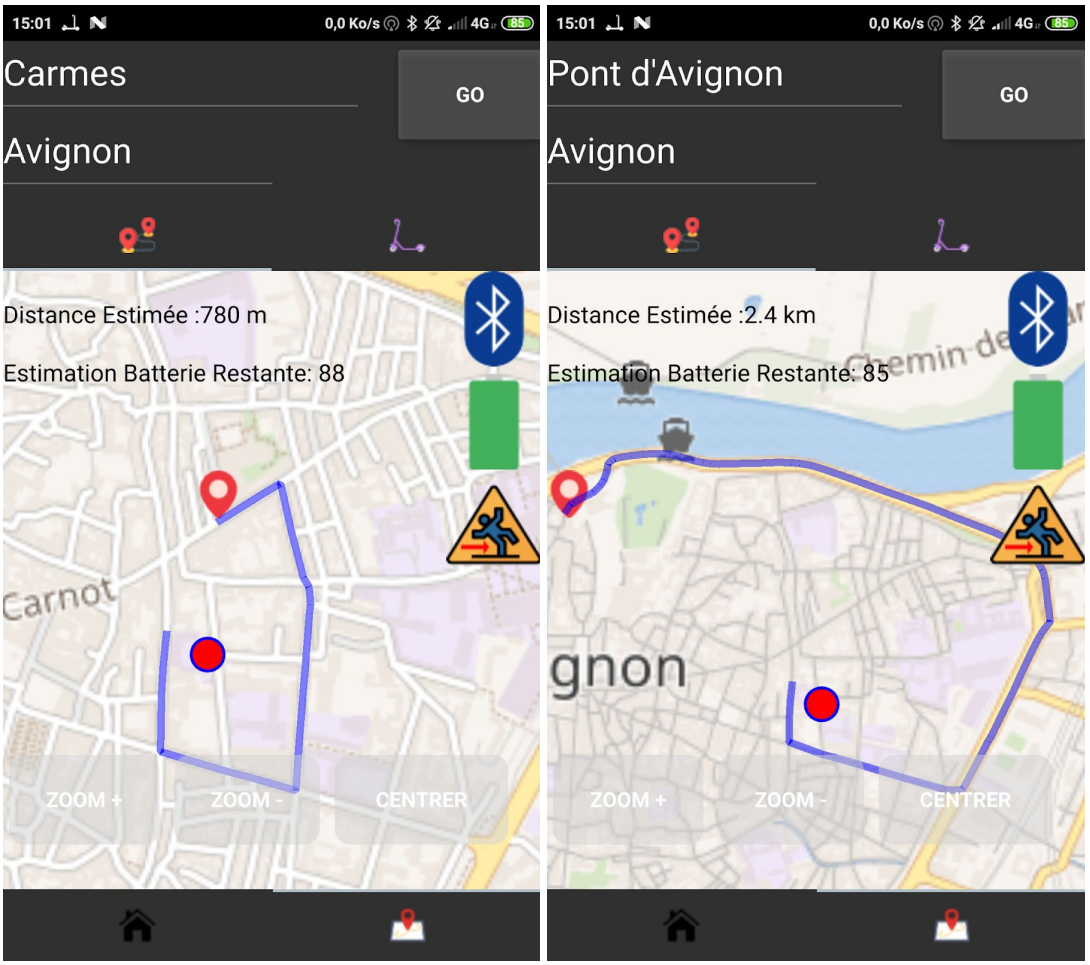


Voici un aperçu de la partie Carte de l'application lorsque l'on est connectée avec la trottinette.



La saisie d'un trajet se réalisera dans l'onglet Itinéraire de la page Carte . Pour cela il suffit de saisir le nom de la rue et le nom de la ville et d'appuyer sur le bouton GO.

Voici des exemples de saisie de trajet.



La distance estimée s'adapte en mètre ou en kilomètre .

Glossaire

TEC : Trottinette Électrique Connectée

IHM : Interface Homme Machine

Framework : Un framework (ou infrastructure logicielle en français) désigne en programmation informatique un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application.

Socket : un socket est une interface logicielle avec les services du système d'exploitation, grâce à laquelle un développeur exploitera facilement et de manière uniforme les services d'un protocole réseau.

RFCOMM: signifie « Radio frequency communication ». Ce service est basé sur les spécifications RS-232, qui émule des liaisons séries.

UUID : Universally Unique Identifier, identifiant universel unique en français est un nombre de 128 bits permettant d'identifier de manière unique.

SDK : Un SDK, pour Sotfware Development Kit, désigne un ensemble d'outils utilisés par les développeurs pour le développement d'un logiciel destiné à une plateforme déterminée (Linux, Windows, Android, etc.). On le traduit en français par kit de développement. Un SDK est composé de :

- D'un traducteur capable de traduire le langage de programmation en langage machine
- D'un éditeur de liens en mesure de relier, en un fichier exécutable
- Différents éléments et de bibliothèques de routines.

Android NDK : L'Android Native Development Kit ou Android NDK est une API du système d'exploitation Android permettant de développer directement dans le langage du matériel cible, par opposition au Android SDK qui est une abstraction en bytecode Java, indépendante du matériel.