Università degli Studi di Napoli Federico II


Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

# Implementazione e validazione di un sistema On-Board per l'ausilio al conducente su veicoli connessi tramite Bluetooth Low Energy e App Flutter

Relatore:
Prof. Porfirio Tramontana

Correlatori:
Dott. Gennaro Galante
Ing. Enrico Landolfi

Moriello Valeria N86002139

# Contestualizzazione del Problema

- Il progetto si colloca all'interno di un contesto di innovazione, affrontando le sfide della mobilità urbana attraverso la realizzazione di un sistema avanzato di comunicazione tra autoveicoli e infrastrutture.
- In un'epoca in cui la tecnologia guida la trasformazione dei trasporti, l'obiettivo primario è facilitare la condivisione tempestiva e intelligente di informazioni tra veicoli, con l'intento di migliorare la sicurezza stradale.

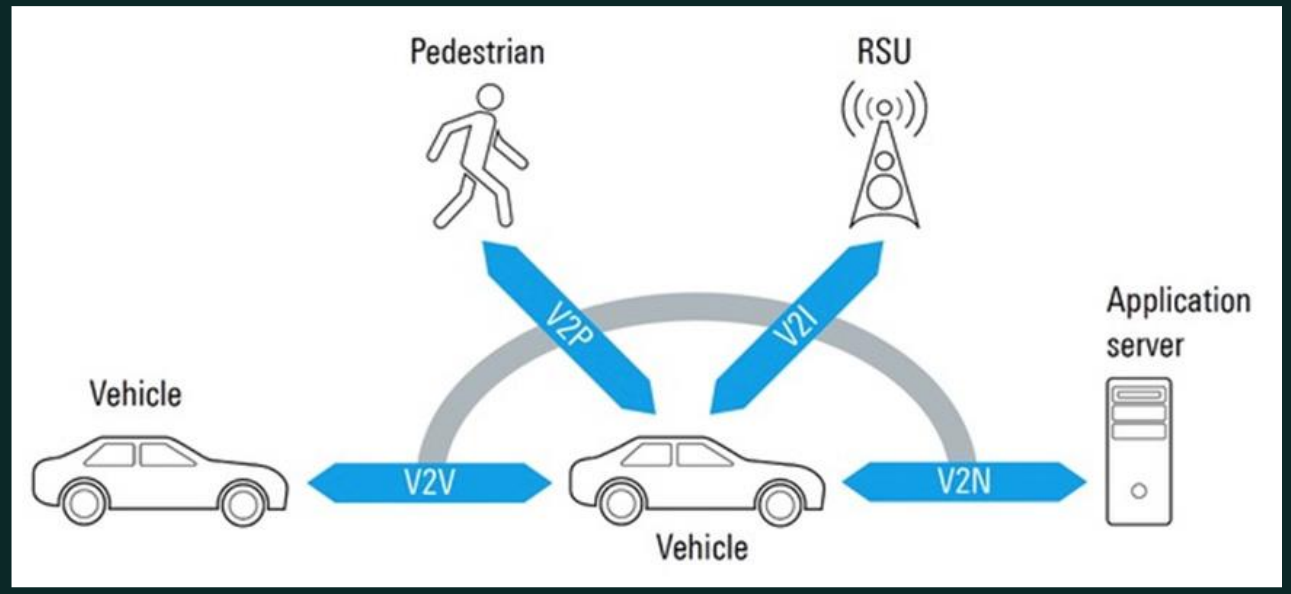
# Elementi chiave

- **V2X**: comunicazione
- **OBU**: agenti attivi

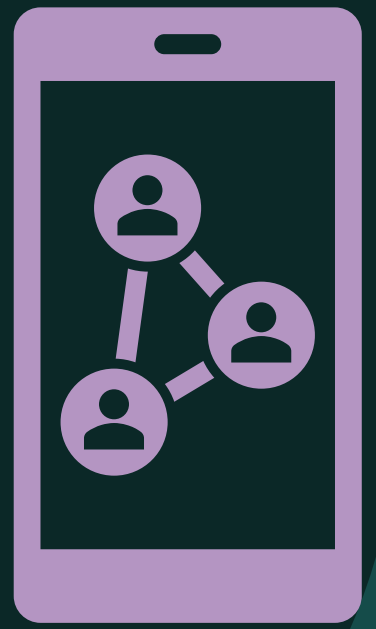
**Vehicular-to-Everything (V2X)**: un concetto che abbraccia varie forme di comunicazione, inclusa **Vehicle-to-Vehicle (V2V)** e la **Vehicle-to-Infrastructure (V2I)**. Questa visione crea una rete collaborativa in cui veicoli e infrastrutture condividono dati in tempo reale, sfruttando la potenza della connettività per migliorare la sicurezza stradale, ottimizzare il flusso del traffico e promuovere soluzioni di mobilità intelligenti.

**On Board Unit (OBU)**: dispositivi messi a disposizione dalla **Unex Corporation**, installati in ciascun veicolo partecipante. Esse operano come nodi intelligenti, acquisendo dati rilevanti sul veicolo e l'ambiente circostante e trasmettendo queste informazioni agli altri autoveicoli presenti nella rete.

- **NetCom Group** non solo si propone di implementare la tecnologia **V2X** attraverso le **OBU**, ma aspira a ridefinire il panorama della mobilità urbana, trasformando ogni veicolo in un nodo consapevole e proattivo di una rete di comunicazione avanzata.



# Obiettivi principali: dallo sviluppo di un'interfaccia grafica allo sviluppo from scratch dell'applicazione mobile



- Il progetto ha avuto inizio con la richiesta da parte dell'azienda di sviluppare un'interfaccia per un'**app Android esistente**, la quale riceve dati da una **Raspberry Pi 3 Model B+** sulla quale è in esecuzione uno script **Python**, responsabile dell'invio di dati. Questa fase iniziale ha fornito le basi del progetto, focalizzandosi sull'aspetto visuale e funzionale dell'applicazione.
- Durante la fase di **fattibilità**, però, le richieste e le esigenze del progetto hanno portato alla scelta di realizzare l'applicazione da capo per più ragioni.

# Confronto tra il progetto esistente e le nuove esigenze

## STATO ATTUALE DEL PROGETTO

- Applicazione realizzata con **Android**
- Protocollo di comunicazione **Bluetooth** mediante **RFCOMM**

## OBIETTIVI DA RAGGIUNGERE

- Volontà di realizzare un'applicazione cross-platform (framework **Flutter**)
- Migrazione da **Bluetooth** a **Bluetooth Low Energy**

# Tecnologie adottate per lo sviluppo

**PERIPHERAL**

- Programmazione di un nuovo script Python utilizzando il **Server GATT (Generic Attribute Profile)**, responsabile della connessione **BLE** e della trasmissione dei dati.

**CENTRAL**

- Sviluppo di un'applicazione mobile mediante framework **Flutter** per la visualizzazione dei dati.

# Analisi dei requisiti

## REQUISITI FUNZIONALI

- L'app deve presentare in modo chiaro e comprensibile i **dati del veicolo**.
- Lo script Python **deve generare dati casuali e continui**, impacchettati in un messaggio JSON, per simulare situazioni di traffico realistiche.

## REQUISITI NON FUNZIONALI

- L'applicazione deve essere sviluppata in **Flutter** per garantire la compatibilità cross-platform, permettendo l'esecuzione su dispositivi **Android** e **iOS**.
- Lo script Python deve essere progettato per effettuare la **comunicazione Bluetooth Low Energy** garantendo la trasmissione dei dati tra il veicolo e l'app.

# Dati del veicolo

- **Velocità del vento**: indica la velocità del vento in km/h.
- **Pendenza**: l'inclinazione della strada espressa in gradi.
- **Velocità**: la velocità del veicolo espressa in km/h.
- **Distanza relativa**: la distanza dal veicolo frontale espressa in metri.
- **Comportamento dell'utente alla guida**: un valore in percentuale, che rappresenta il comportamento dell'utente alla guida.
- **Collisione** riporta il rischio di una collisione:

  0: indica nessun rischio;

  1: rischio di collisione.
- **Pedone vulnerabile** indica la presenza di un pedone in strada:

  0: nessun pedone in strada;

  1: pedone in strada.
- **Veicolo d'emergenza** indica la ricezione di una richiesta di priorità da parte di un autoveicolo d'emergenza:

  0: nessuna richiesta di priorità;

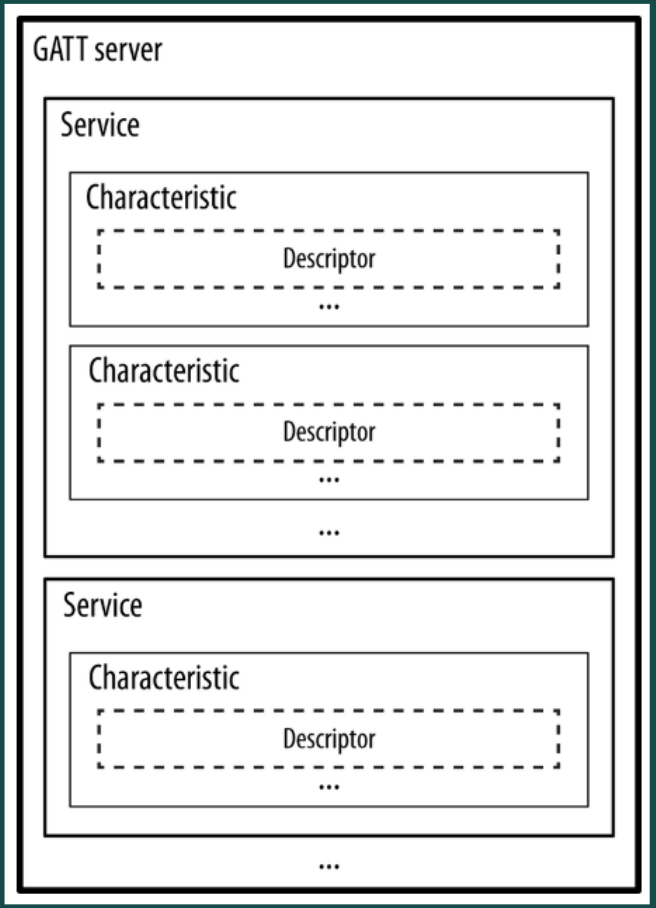
  1: presenza di una richiesta di priorità.

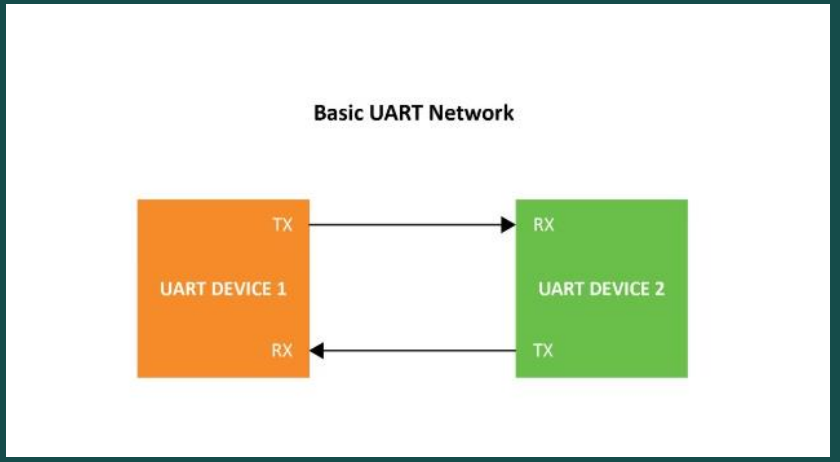
# UART over Bluetooth Low Energy

- Il protocollo **Universal Asynchronous Receiver/Transmitter (UART)** è uno dei protocolli più diffusi per la comunicazione con dispositivi computerizzati attraverso una porta seriale.
- Quando si parla di **UART** in un contesto moderno, ci si riferisce spesso all'emulazione di una porta seriale attraverso la tecnologia Bluetooth Low Energy.
- l'implementazione di un servizio **UART** personalizzato consente di emulare il comportamento di una porta seriale attraverso il **server GATT**.

# Emulazione della porta UART: funzionamento

- Le **caratteristiche GATT** vengono utilizzate per emulare i flussi di dati di una comunicazione UART. Le caratteristiche coinvolte sono:
- **Caratteristica di trasmissione (TX)** utilizzata dal server.
- **Caratteristica di ricezione (RX)** utilizzata dal client.
- Ciascuna caratteristica è identificata da un **Universal Unique Identifier (UUID)**.



# Gestione della comunicazione BLE

- Per abilitare la comunicazione BLE sulla Raspberry Pi, il software interagisce con il sistema **BlueZ** utilizzando **DBus**:
- **BlueZ** è il middleware Bluetooth - si colloca tra il sistema operativo e le applicazioni;
- **DBus** è un sistema di comunicazione interprocesso che consente l'interazione tra il software e **BlueZ**.
- Il metodo **RegisterApplication** registra l'applicazione presso il gestore **GATT**, mentre il metodo **RegisterAdvertisement** registra l'annuncio BLE presso il gestore dell'annuncio.
- Queste operazioni sono fondamentali per rendere **disponibile il servizio GATT e annunciarlo agli altri dispositivi BLE circostanti**.

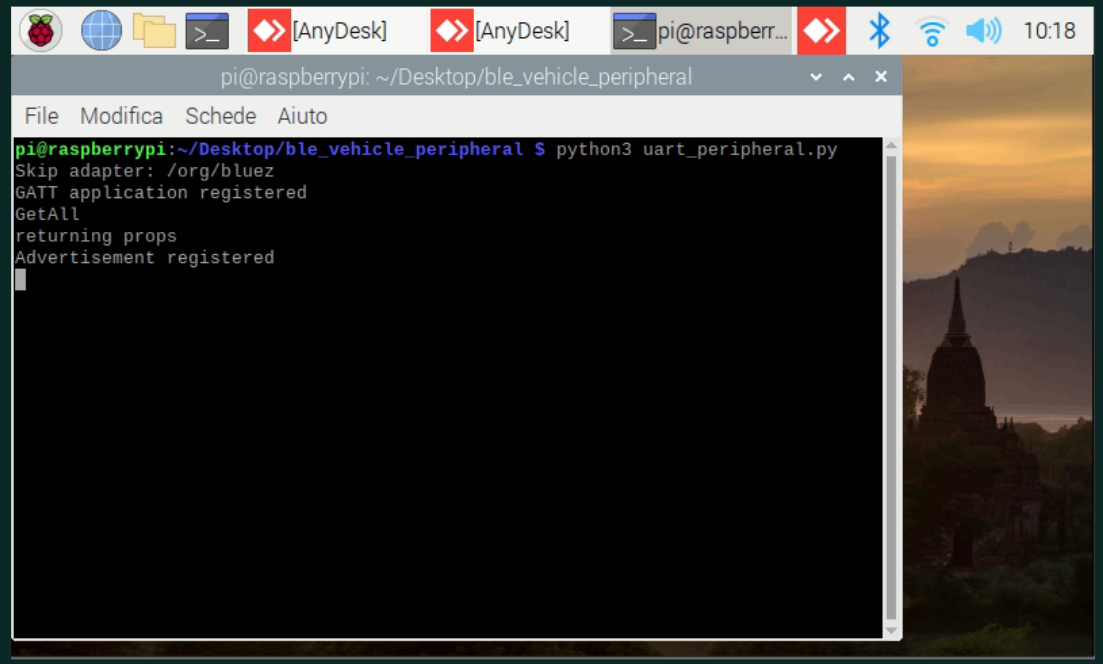
# Generazione e invio dati casuali

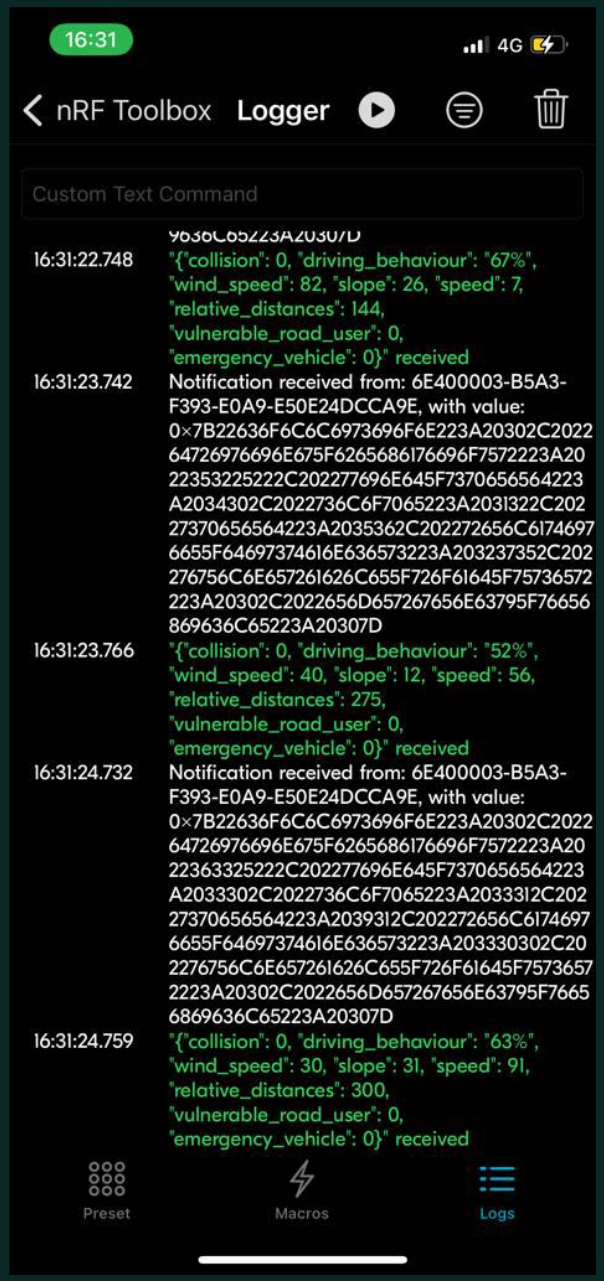
- La classe **TxCharacteristic** implementa i seguenti metodi:
- **send_veicolo_data()** che genera dati casuali;
- **StartNotify()** è responsabile dell'inizio della notifica;
- **StopNotify()** gestisce la terminazione della notifica.

- Questa configurazione assicura che la Raspberry Pi invii dati casuali ed emulati dell'automobile all'applicazione Flutter ogni secondo, attraverso la caratteristica di trasmissione.

```
class TxCharacteristic(Characteristic):
    def __init__(self, bus, index, service):
        Characteristic.__init__(self, bus, index, UART_TX_CHARACTERISTIC_UUID,
                                ['notify'], service)
        self.notifying = False
        self.timeout_id = None

    def send_veicolo_data(self):
        if not self.notifying:
            return

        # Genera dati del veicolo casuali
        collision = int(numpy.random.choice(numpy.arange(0, 2, 1), p=[0.95, 0.05]))
        driving_behaviour = str(random.randrange(0, 100)) + "%"
        wind_speed = random.randrange(0, 100)  # km/h
        slope = random.randrange(0, 50)  # °
        speed = random.randrange(0, 130)  # km/h
        relative_distances = random.randrange(1, 1000)  # m
        vulnerable_road_user = int(numpy.random.choice(numpy.arange(0, 2, 1), p=[0.95, 0.05]))
        emergency_vehicle = int(numpy.random.choice(numpy.arange(0, 2, 1), p=[0.95, 0.05]))

        # Costruisci la stringa JSON con i dati del veicolo
        veicolo_data = {
            "collision": collision,
            "driving_behaviour": driving_behaviour,
            "wind_speed": wind_speed,
            "slope": slope,
            "speed": speed,
            "relative_distances": relative_distances,
            "vulnerable_road_user": vulnerable_road_user,
            "emergency_vehicle": emergency_vehicle
        }

        # Converti il dizionario in una stringa JSON
        veicolo_json = json.dumps(veicolo_data)

        # Converti la stringa JSON in una lista di byte
        value = [dbus.Byte(c.encode()) for c in veicolo_json]

        # Invia i dati del veicolo
        self.PropertiesChanged(GATT_CHRC_IFACE, {'Value': value}, [])

        # Ripeti la chiamata ogni secondo
        GLib.timeout_add_seconds(1, self.send_veicolo_data)

    def StartNotify(self):
        if not self.notifying:
            self.notifying = True
            self.PropertiesChanged(GATT_CHRC_IFACE, {'Notifying': True}, [])
            self.timeout_id = GLib.timeout_add_seconds(1, self.send_veicolo_data)

    def StopNotify(self):
        if self.notifying:
            self.notifying = False
            self.PropertiesChanged(GATT_CHRC_IFACE, {'Notifying': False}, [])
            if self.timeout_id is not None:
                GLib.source_remove(self.timeout_id)
                self.timeout_id = None
```

# Test e verifica



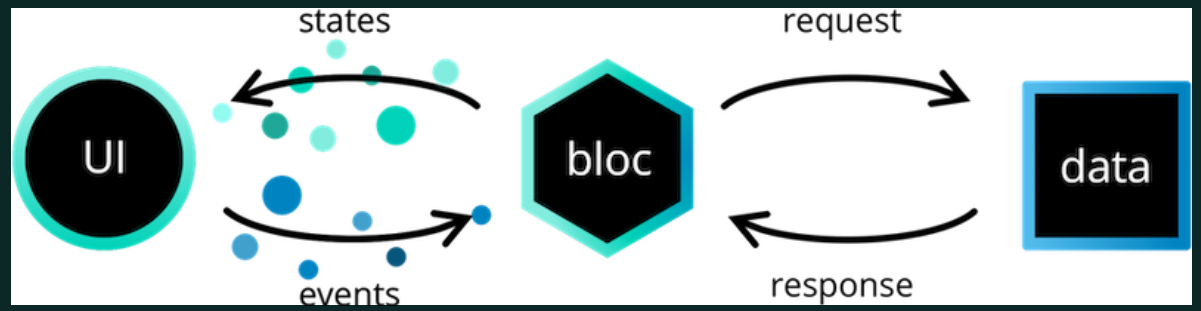

nRF TOOLBOX

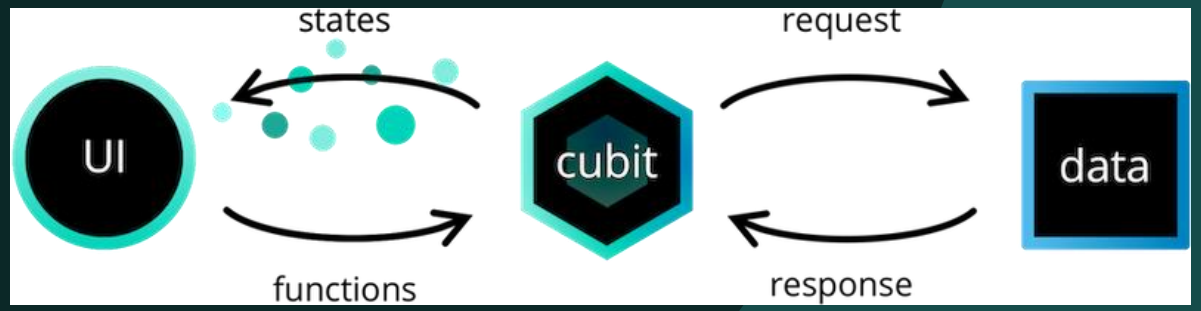
# Flutter

- **Flutter** è un framework open-source sviluppato da Google, progettato per la creazione efficiente di app cross-platform.
- La peculiarità distintiva di Flutter risiede nell'adozione del linguaggio di programmazione **Dart**.
- Questo framework offre agli sviluppatori un ambiente coeso e potente per la costruzione di interfacce utente moderne e ad alte prestazioni.

# Design Patter: BLoC vs Cubit

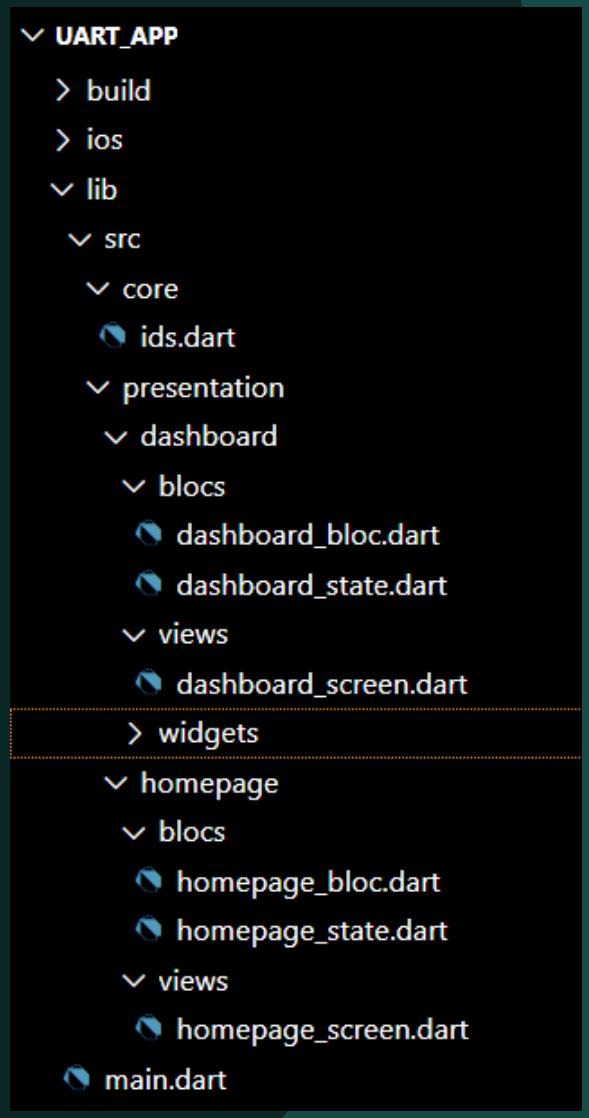
- L'**architettura** dell'app è stata progettata seguendo principi di modularità e facilità di manutenzione, con un'enfasi particolare sulla gestione dello stato attraverso il design pattern **Cubit**, semplificazione del design pattern **BLoC**.
- Questi pattern offrono un approccio dichiarativo e reattivo alla **gestione dello stato** dell'applicazione, consentendo una chiara separazione tra la **logica di business** e la **presentazione** dell'interfaccia utente.



# Architettura dell'app

- **Presentation**
- **homepage** e **dashboard**: queste cartelle rappresentano le due principali sezioni dell'applicazione, ciascuna contenente la propria logica di presentazione.
- Ogni sezione è ulteriormente suddivisa in blocs e views.
- **Blocs**: contiene i Cubits che gestiscono lo stato e la logica di business per le rispettive sezioni. L'utilizzo dei Cubits avviene utilizzando la libreria **bloc**. Per la clean architecture, le cartelle e i file contenenti i Cubits e gli stati vengono chiamate Bloc, proprio perché derivano dalla stessa libreria.
- **Views**: qui sono presenti le interfacce utente di ciascuna sezione. Le views interagiscono con i Cubits per riflettere dinamicamente lo stato dell'applicazione.



# Gestione BLE da parte dell'app Flutter

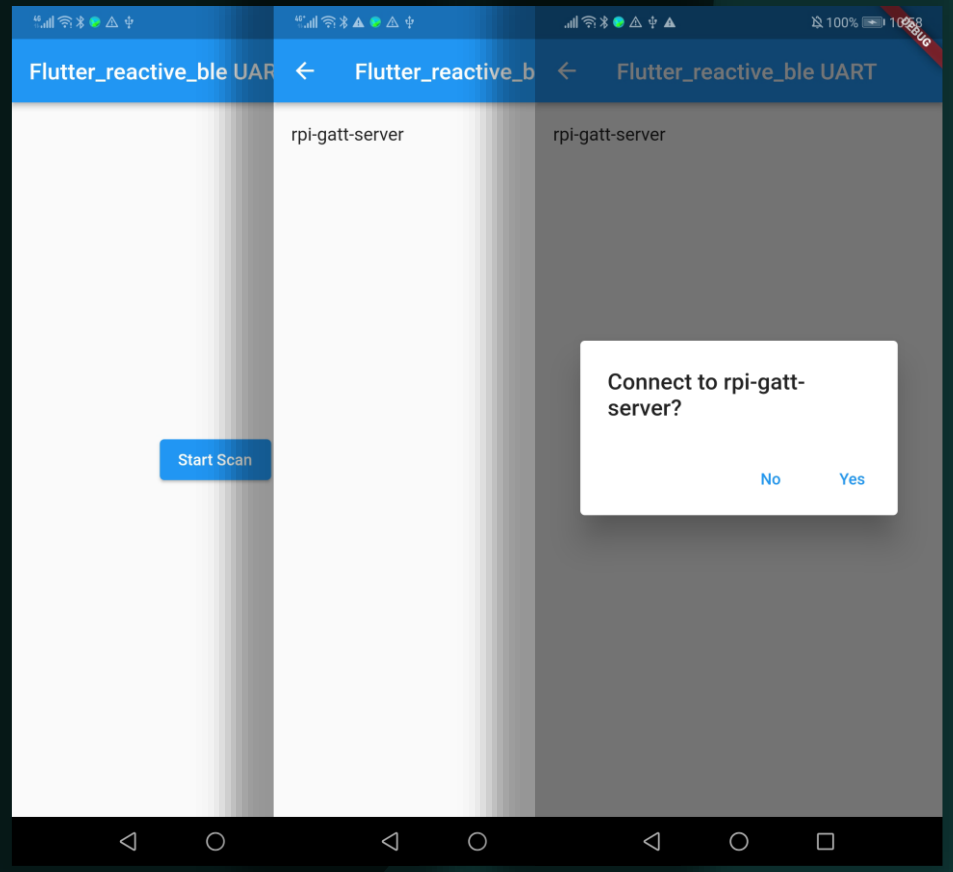
- L'applicazione utilizza la libreria **flutter_reactive_ble** per semplificare la gestione della connessione BLE.
- Gli aspetti chiave della gestione BLE includono:

1. **Inizializzazione del Bluetooth**: la libreria inizializza il modulo Bluetooth e verifica la presenza della funzionalità BLE sul dispositivo.
2. **Scansione dei dispositivi BLE**: l'app effettua la scansione per individuare i dispositivi BLE disponibili, focalizzandosi su quelli che supportano il servizio e la caratteristica UART.
3. **Connessione e disconnessione**: l'app gestisce i processi di connessione e disconnessione con il dispositivo Raspberry Pi, gestendo gli eventi corrispondenti.
4. **Ricezione dei dati**: utilizzando la libreria, l'app riceve dati attraverso il canale BLE.

# Inizializzazione del Bluetooth e scansione dei dispositivi BLE

- Le prime due fasi della gestione del BLE da parte dell'app Flutter, sono gestite dalla cartella **homepage**. Infatti, essa contiene:
- **homepage_bloc.dart** che gestisce il flusso dei dati relativi ai Cubits responsabili della scannerizzazione e alla connessione con il dispositivo con servizio UART, corrispondente all'UUID del servizio presente su script Python in esecuzione su Raspberry. Ogni funzione presente all'interno della classe **HomepageCubit** avrà un **emit** con lo stato associato;
- **homepage_state.dart** in cui ciascuna classe rappresenta uno specifico stato dell'applicazione, gestendo le informazioni necessarie durante le diverse fasi dell'esecuzione;
- **homepage_screen** contiene i widget responsabili della visualizzazione relativa alla parte grafica dell'applicazione. Infatti, la classe **HomeConnector** estende StatelessWidget ed è la classe responsabile di definire le interfacce grafiche degli stati relativi alla homepage, in base a determinati stati dell'applicazione.

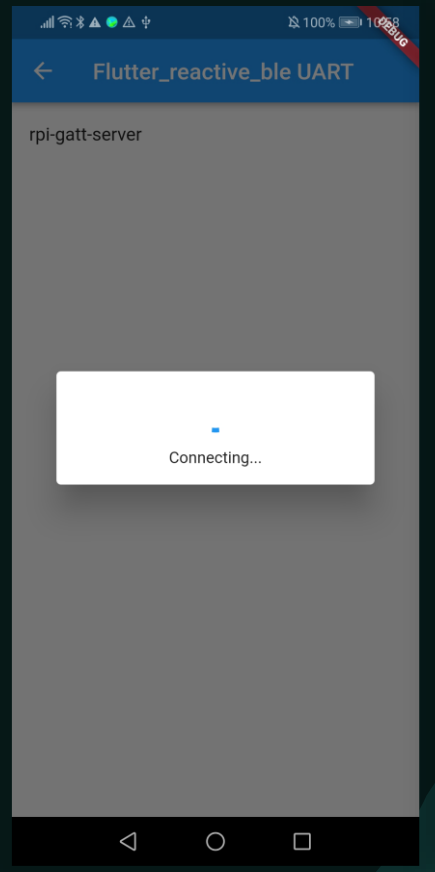
```dart
class HomeConnector extends StatelessWidget {
  const HomeConnector({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (_) => HomepageCubit(),
      child: const HomePageBody(),
    ); // BlocProvider
  }
}
```

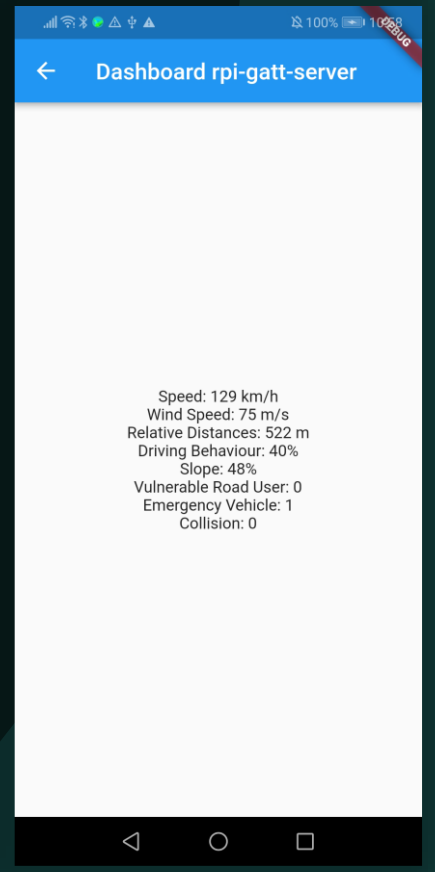


# Connessione/Disconnessione e Ricezione dei dati

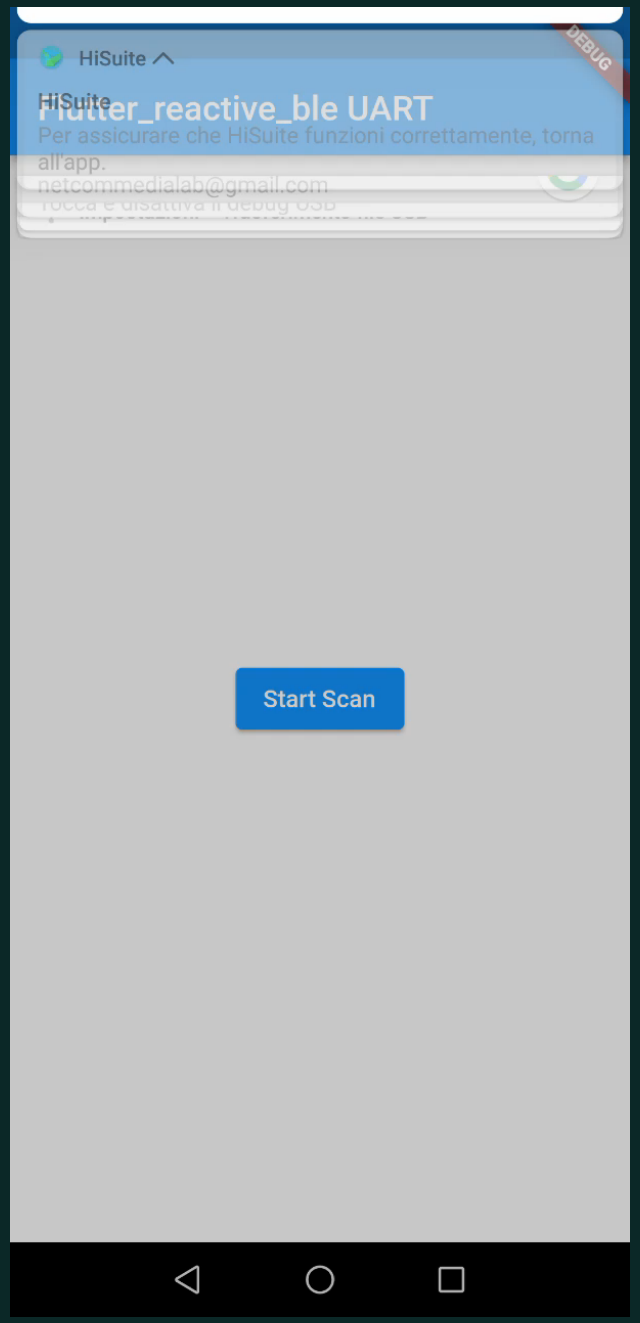
- Le seconde due fasi della gestione del BLE da parte dell'app Flutter, sono gestite dalla cartella **dashboard**. Infatti, essa contiene:
- **dashboard_bloc.dart** che gestisce il flusso dei dati relativi ai Cubits responsabili della connessione (già avvenuta) e della corretta ricezione dei dati dal dispositivo con servizio UART. Ogni funzione presente all'interno della classe **DashboardCubit** avrà un **emit** con lo stato associato;
- **dashboard_state.dart** in cui ciascuna classe rappresenta uno specifico stato dell'applicazione, gestendo le informazioni necessarie durante le diverse fasi dell'esecuzione;
- **dashboard_screen** contiene i widget responsabili della visualizzazione relativa alla parte grafica dell'applicazione. Infatti, la classe **DashboardConnector** estende StatelessWidget ed è la classe responsabile di definire le interfacce grafiche degli stati relativi alla homepage, in base a determinati stati.

```dart
class DashboardConnector extends StatelessWidget {
  const DashboardConnector({Key? key, required this.discoveredDevice})
      : super(key: key);

  final DiscoveredDevice discoveredDevice;

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => DashboardCubit()..onConnectDevice(discoveredDevice),
      child: const _DashboardBody(),
    ); // BlocProvider
  }
}
```



# Registrazione dell’applicazione


HiSuite
HiSuite
Per assicurare che HiSuite funzioni correttamente, torna all'app.
netcommedialab@gmail.com
DEBUG
Start Scan

# Conclusioni

- L'implementazione ha avuto successo nell'instaurare una connessione **BLE** tra la **Raspberry Pi** e l'app mobile realizzata con il framework **Flutter**.
- La comunicazione avviene attraverso un **server GATT** e le relative caratteristiche **UART**.
- La soluzione ha superato con successo i test di interoperabilità, garantendo una corretta comunicazione tra i dispositivi.
- La strutturazione del software, basata sul design pattern **BLoC**, in particolare con l'uso di **Cubit**, ha fornito un'architettura modulare e manutenibile.

# Grazie per l’attenzione

*In memoria di Maurizio Moriello*