Introduction

Blechem APP connects with BLE devices and communicates with them. The application is crafted in such a way that it sends and receives data from “DF Robot Bluno” which is a BLE device. This app is intended for users who are interested in stoichiometry.

Getting Started

To run this application in your native environment you need to set up your environment.

Click here to setup- Here.

Dependencies [version included]used in this project

Click on the dependencies to see official documentation of the dependencies.

```
cupertino_icons: ^1.0.2
permission_handler: ^10.2.0
csv: ^5.0.2
file_saver: ^0.2.8
path_provider: ^2.0.15
device_info_plus: ^9.0.2
flutter_blue_plus: ^1.13.4
flutter_email_sender: ^6.0.0
app_settings: ^5.0.0
location: ^4.4.0
get: ^4.6.5
```

Cloning the Repository

- Open terminal or command prompt
- Navigate to the desired directory
- Run : “git clone [HTTPS_repository_url]”

Project Structure

Blechem follows standard flutter structure. The main and editable directories are mentioned below.

- ‘lib/’: Contains all the dart files. The dart files contains source code for different functionalities and UI’s. lib is architectured in MVC pattern.
- ‘android/’: This folder contains several crucial file to handle android specific configuration, gradle properties, permission handling etc.
- ‘ios/’: This is same as ‘android/’ but for iOS-specific congigurations.
- ‘Pubspec.yaml’ : All the dependencies are added in this file. This file is **indentation sensitive .**

Architecture

Blechem follows MVC using the GetX state management solution.

Why MVC and GetX?

MVC pattern is a good software design culture that enables any developer to write clean , maintainable and reusable code.

Get state management is easy to understand state management solution which updates app data all over the app without any hastle.

Dataflow

- User interacts with the UI
- UI sends events to controller
- Controller uses model to handle event based response
- Controller notifies the UI of the changes.

Brief Description of Controllers Classes

DeviceController()

This controller class is responsible for BLE device scan, Update, Connect , Disconnect , Building communication with specific

Bluno (eg. pH,EC) . Codes are commented to describe the methods used. Also, the dataflow is handled by this controller.

CsvController()

This controller class used to create and save csv file on ‘Downloads’ folder as well as on temporary directory. Also the file is attached to email through this controller instance.

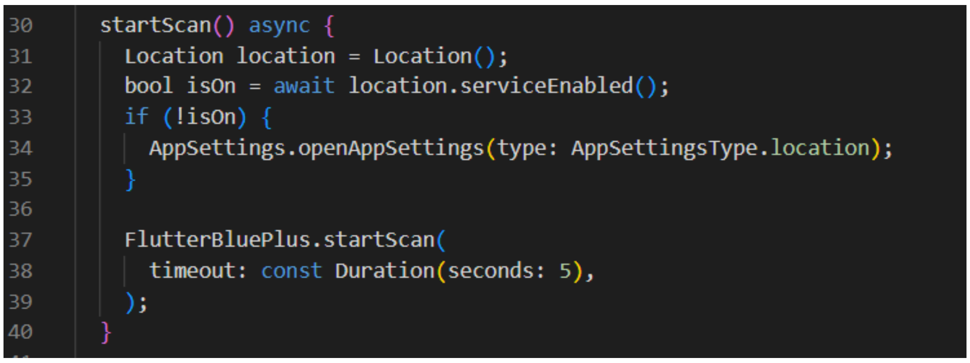
HomePageController()

HomePageController, combines the UI with the controllers.

Features and Functionality

Feature 1: BLE Scan

This feature is used from the package ‘flutter_blue_plus’. This feature scans for ble devices and assigns them to ScanResults variable.

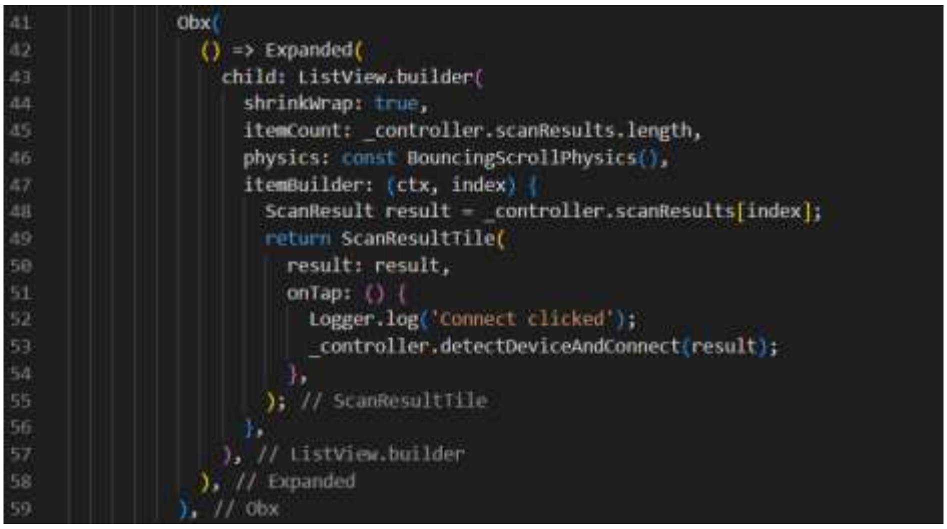


```
startScan() async {
  Location location = Location();
  bool isOn = await location.serviceEnabled();
  if (!isOn) {
    AppSettings.openAppSettings(type: AppSettingsType.location);
  }

  FlutterBluePlus.startScan(
    timeout: const Duration(seconds: 5),
  );
}
```

Feature 2: Show all the available devices

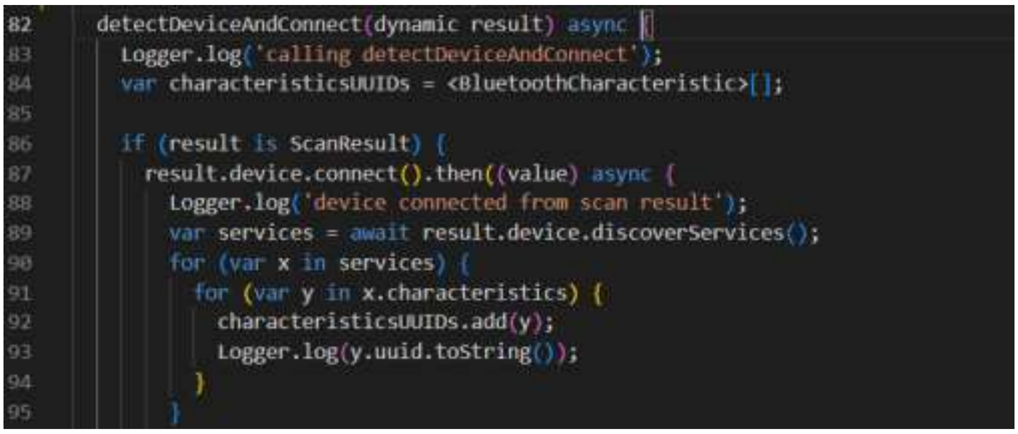
Using list view builder and help of some of the custom widget list of devices are shown on the app.



```
Obx(
  () => Expanded(
    child: ListView.builder(
      shrinkWrap: true,
      itemCount: _controller.scanResults.length,
      physics: const BouncingScrollPhysics(),
      itemBuilder: (ctx, index) {
        ScanResult result = _controller.scanResults[index];
        return ScanResultTile(
          result: result,
          onTap: () {
            Logger.log('Connect clicked');
            _controller.detectDeviceAndConnect(result);
          },
        ); // ScanResultTile
      },
    ), // ListView.builder
  ), // Expanded
), // Obx
```

Feature 3: Connect and Detect specific Bluno

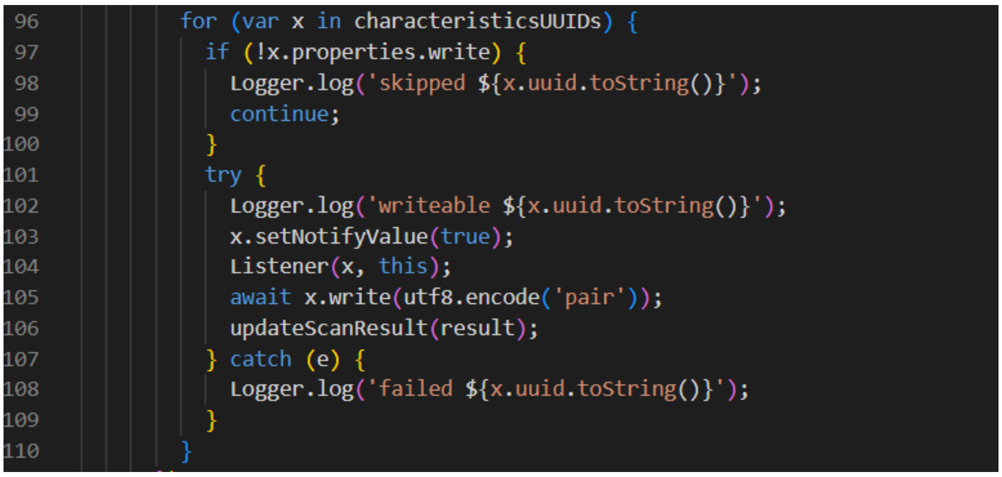
Connection is made using method **device.connect()** provided by 'flutter_blue_plus'.



```
  detectDeviceAndConnect(dynamic result) async {
    Logger.log('calling detectDeviceAndConnect');
    var characteristicsUUIDs = <BluetoothCharacteristic>[];

    if (result is ScanResult) {
      result.device.connect().then((value) async {
        Logger.log('device connected from scan result');
        var services = await result.device.discoverServices();
        for (var x in services) {
          for (var y in x.characteristics) {
            characteristicsUUIDs.add(y);
            Logger.log(y.uuid.toString());
          }
        }
```

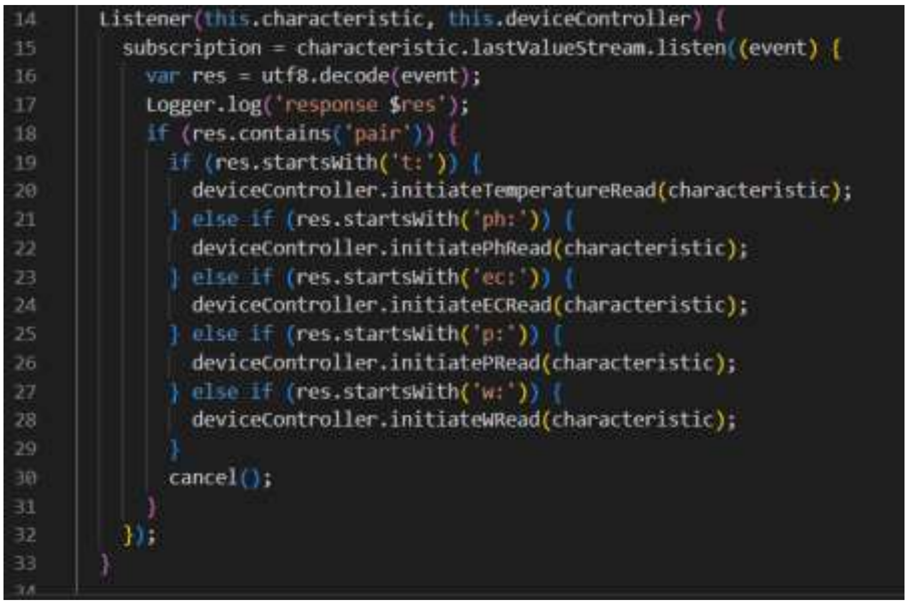
Also detection of valid bluno device's specific read and write characteristics are searched under the method "detctDeviceAndConnect()"



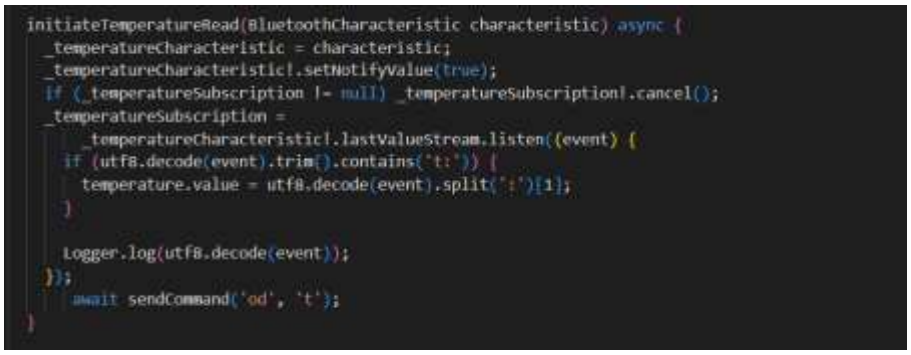
```
        for (var x in characteristicsUUIDs) {
          if (!x.properties.write) {
            Logger.log('skipped ${x.uuid.toString()}');
            continue;
          }
          try {
            Logger.log('writeable ${x.uuid.toString()}');
            x.setNotifyValue(true);
            Listener(x, this);
            await x.write(utf8.encode('pair'));
            updateScanResult(result);
          } catch (e) {
            Logger.log('failed ${x.uuid.toString()}');
          }
        }
```

Feature 4: Subscribing to bluno

Subscription is handled using **model class Listener** and custom method “**initiateRead[specific_sensor_name] ()**”.



```
Listener(this.characteristic, this.deviceController) {
  subscription = characteristic.lastValueStream.listen((event) {
    var res = utf8.decode(event);
    Logger.log('response $res');
    if (res.contains('pair')) {
      if (res.startsWith('t:')) {
        deviceController.initiateTemperatureRead(characteristic);
      } else if (res.startsWith('ph:')) {
        deviceController.initiatePhRead(characteristic);
      } else if (res.startsWith('ec:')) {
        deviceController.initiateECRead(characteristic);
      } else if (res.startsWith('p:')) {
        deviceController.initiatePRead(characteristic);
      } else if (res.startsWith('w:')) {
        deviceController.initiateWRead(characteristic);
      }
      cancel();
    }
  });
}
```



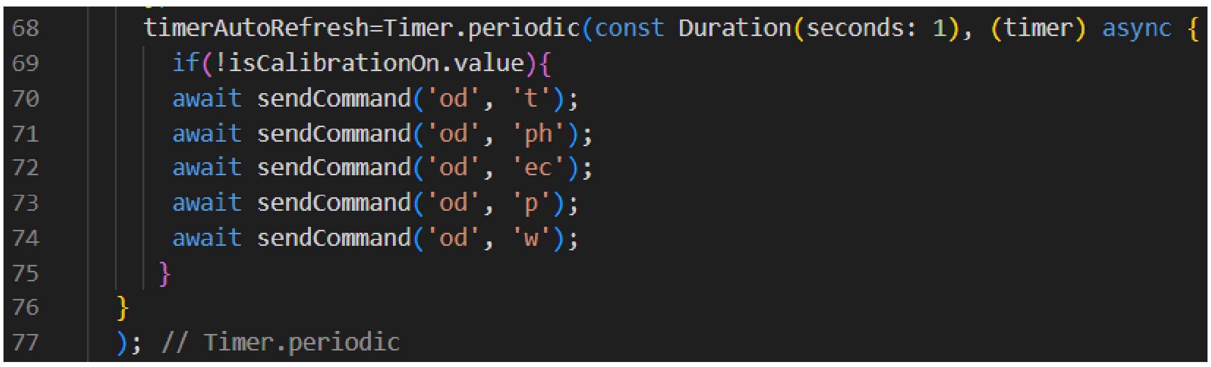
```
initiateTemperatureRead(BluetoothCharacteristic characteristic) async {
  _temperatureCharacteristic = characteristic;
  _temperatureCharacteristic!.setNotifyValue(true);
  if (_temperatureSubscription != null) _temperatureSubscription!.cancel();
  _temperatureSubscription =
      _temperatureCharacteristic!.lastValueStream.listen((event) {
    if (utf8.decode(event).trim().contains('t:')) {
      temperature.value = utf8.decode(event).split(':')[1];
    }

    Logger.log(utf8.decode(event));
  });
   await sendCommand('od', 't');
}
```

Feature 5: Auto Refresh and Start New Project

There are mainly two ways to get data. Autorefresh after every 1s and Start New project.

-Autorefresh feature using a periodic timer only

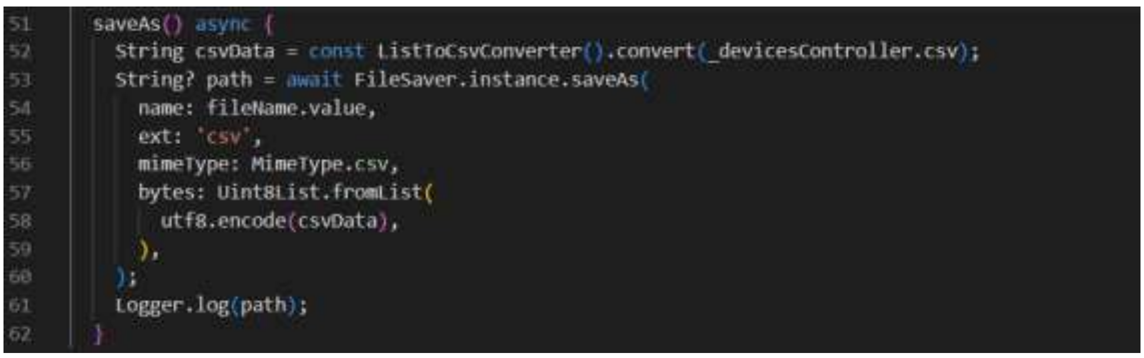


```
  timerAutoRefresh=Timer.periodic(const Duration(seconds: 1), (timer) async {
    if(!isCalibrationOn.value){
    await sendCommand('od', 't');
    await sendCommand('od', 'ph');
    await sendCommand('od', 'ec');
    await sendCommand('od', 'p');
    await sendCommand('od', 'w');
    }
  }
); // Timer.periodic
```

- **Start New Project**
  This feature include some UI and methods. Main method that controls this feature is StartContinuos() method , StopContinuos() Method , CsvController class.
  You can find the description in the comment in the source code.

Feature 6: Save to Downloads Folder

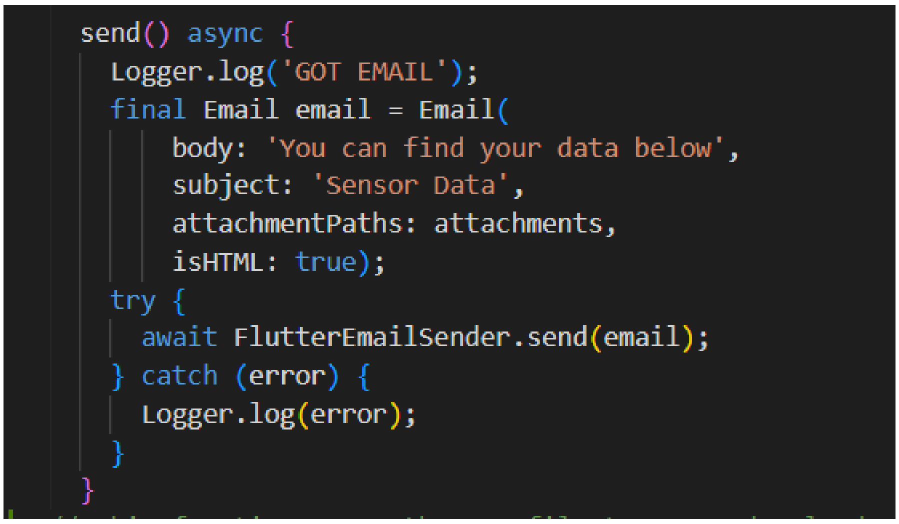
This feature is made using file_saver package. This is a feature under Start New Project.



```
saveAs() async {
  String csvData = const ListToCsvConverter().convert(_devicesController.csv);
  String? path = await FileSaver.instance.saveAs(
    name: fileName.value,
    ext: 'csv',
    mimeType: MimeType.csv,
    bytes: Uint8List.fromList(
      utf8.encode(csvData),
    ),
  );
  Logger.log(path);
}
```

Feature 7: Emailing CSV attachments

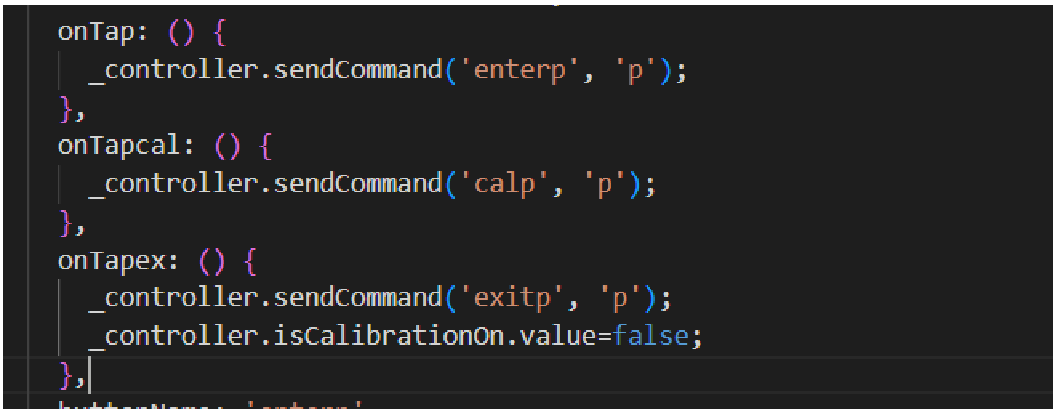
This was made using flutter_email_sender package. Also the attachments are generated in app's temporary directory. This manged bt CsvController class.



```
send() async {
  Logger.log('GOT EMAIL');
  final Email email = Email(
      body: 'You can find your data below',
      subject: 'Sensor Data',
      attachmentPaths: attachments,
      isHTML: true);
  try {
    await FlutterEmailSender.send(email);
  } catch (error) {
    Logger.log(error);
  }
}
```

Feature 8: Calibration

Calibration is done in the settings page. After building connection with device using sendCommand() method of DeviceController() specified commands are send to device for calibration.



```
onTap: () {
  _controller.sendCommand('enterp', 'p');
},
onTapcal: () {
  _controller.sendCommand('calp', 'p');
},
onTapex: () {
  _controller.sendCommand('exitp', 'p');
  _controller.isCalibrationOn.value=false;
},
```

Run The Application

To run the application after all the setup go to the terminal and execute **‘flutter run’**

To build apk’s execute ‘**flutter build apk –split-per-abi**’

Conclusion

Thank you for using and contributing to this project.