Technical Documentation

Sixth Sense Project
Multisensory Input Puck

Client
Engineering Students

Created By
Albert Koyikalorthe Siby
Zeel Manvendrakumar Patel
Andrew Bourgeois
Kuang Xu
Parag Salgotra
Abdelrahman Amer
Shady Hussein

Table of Contents

1.0 Backend - Data Processing

The following section will walkthrough how the data is received and processed in the app.

1.1 Communication to the Puck

The Communication process is an initial step for the puck to connect to Android Application. In this technical documentation, we will explore the process of establishing a USB serial connection with an Arduino device, including the necessary hardware and software components, and provide a step-by-step guide for sending and receiving data through this connection.

1.1.1 USB Communication Set-up

Communication to Puck is initiated in MainActivity.kt File where the app is triggered for a start-up. And we have used a serial USB connection dependency: “com.github.felHR85:UsbSerial:6.1.0”.We have a set of variables initialized for establishing the connection using dependency components which is as follows:

```
class MainActivity : ComponentActivity()
{
  //variable to pass the vendor-ID of device
  val VENDORID: Int = 9025

  //variable to initialize BaudRate
  val BAUDRATE: Int = 115200

  //initialize variable of type UsbDevice
  private var mDevice: UsbDevice? = null

  //initialize variable of type UsbSerialDevice connection using cable
  private var mSerial: UsbSerialDevice? = null

  //initialize variable of type UsbDeviceConnection
  private var mConnection: UsbDeviceConnection? = null

  //initialize variable for usb serial permission
  private val actionUsbPermission = "com.android.example.USB_PERMISSION"

  //initialize variable usb connection management
  private lateinit var mUsbManager: UsbManager

  //initialize the variable for buffer data storing
  private var message = ""


  //variable default value is 0 when no device connected and changed based on connection
success(1)/failure(-1)
  private var statusConnectionIdentifier: Int = 0


}
```

Further we have the method which shows the dialogue alert based on the connection failure and success which as following

```
class MainActivity : ComponentActivity()
{
  //function we show the Alert based on device connection
  fun ShowConnectionWarning(value: Int) {
   if (value == -1) {
       DialogAlert().AlertDialogMessage(title = "Alert!", message = "Failed to Connect
Device")
   } else if (value == 0) {
       DialogAlert().AlertDialogMessage(title = "Alert!", message = "No Device Found")

   } else if (value == 1) {
       DialogAlert().AlertDialogMessage(
           title = "Alert!",
           message = "Device Connection Established"
       )
   }
}
}
```

Now, the main connection process starts with a call to intiateFilters(), which passes the device list, its permissions and broadcast receiver to register the receiver which then trigger the call to broadcast receiver function for registering it. which is as follows:

```
private val broadcastReceiver = object : BroadcastReceiver() {

   override fun onReceive(context: Context, intent: Intent) {
       //if device serial is not open then it open for the connection with extra permission
       if (intent?.action!! == actionUsbPermission) {
           val granted: Boolean =
               intent.extras!!.getBoolean(UsbManager.EXTRA_PERMISSION_GRANTED)
           if (granted) {
               mConnection = mUsbManager.openDevice(mDevice)

               mSerial = UsbSerialDevice.createUsbSerialDevice(mDevice, mConnection)
               if (mSerial != null) {
                   if (mSerial!!.open()) {
                       mSerial!!.setBaudRate(BAUDRATE)
                       mSerial!!.setDataBits(UsbSerialInterface.DATA_BITS_8)
                       mSerial!!.setStopBits(UsbSerialInterface.STOP_BITS_1)
                       mSerial!!.setParity(UsbSerialInterface.PARITY_NONE)
                       mSerial!!.setFlowControl(UsbSerialInterface.FLOW_CONTROL_OFF)
                       mSerial!!.read(mCallback)
                   }
               } else {
                   Log.i("Serial", "Port is Null")
               }
           }
       } else if (intent.action == UsbManager.ACTION_USB_DEVICE_ATTACHED) {
         //it will start the connection
         startUsbConnection()
       } else if (intent.action == UsbManager.ACTION_USB_DEVICE_DETACHED) {
          //terminate the connection
```

```
            disconnect()
        }
    }
}
```

If the serial connection is open and ready to initiate the connection to the puck then StartUsbConnection() function is triggered and if the connection fails serial is closed by triggering the disconnect() function. which is as follows:

```
private fun startUsbConnection() {
    val usbDevices: HashMap<String, UsbDevice>? = mUsbManager.deviceList

    if (!usbDevices?.isEmpty()!!) {
        var keep = true
        usbDevices.forEach { entry ->
            mDevice = entry.value
            val deviceVendorId: Int? = mDevice?.vendorId

            if (deviceVendorId == VENDORID) {
                //device connection success
                val intent: PendingIntent =
                    PendingIntent.getBroadcast(this, 0, Intent(actionUsbPermission), 0)
                mUsbManager.requestPermission(mDevice, intent)
                statusConnectionIdentifier = 1
                keep = false

            } else {
                //device connection failed
                mConnection = null
                mDevice = null
                Log.i("Serial", "Connection to Device was Failed")

                statusConnectionIdentifier = -1
            }
        }

        if (!keep) {
            return
        }
    } else {
        //no device found variable updating
        statusConnectionIdentifier = 0
    }
}
```

We have disconnect function which terminates the usb serial session which is as followed:

```
private fun disconnect() {
    //terminate the serial session
    mSerial?.close()
}
```

Furthermore after a successful connection is established between the Puck and Android App. Then we trigger the function to commence data reading from the Puck buffer and send it for processing. which is as follows:

```
private val mCallback = UsbSerialInterface.UsbReadCallback { arg0 ->
   try {
       val buffer = String(arg0, charset("UTF-8")).trim()
       Log.i("Buffer Data: ", buffer)
       if (buffer != '!'.toString()) {
           message += buffer
       } else {
           //send data for the process
           sendTheDataForProcess(message)
           message = "" //re-initialize the variable to empty string
       }
   } catch (e: IOException) {
       e.printStackTrace()//trace down the error while reading
   }
}
```

1.1.2 Sort the incoming data

The sortSensorData is a method in the main activity to sort the data according to the name of the sensor and add that data of that particular sensor to the database.

This method receives the data in the form of the string, then split the string and the first line of data contains the indicator of the sensor (For example : “Lo” it is the indicator for the sensor longitude) in the same way all the sensor have their indicators and using the when statement according the indicator the data for that sensor is added to the database. The method is declared as follows (with only some of the indicators all other are declared in the same way) :

```
private fun SortSensorsData(data: String) {
   val DataArray: List<String> = data.split(",")

   if (DataArray.isEmpty()) {
       println(" Data is Empty ")
   }

   when (DataArray[0]) {
       "Lo" -> {

           myLongitude.addToDatabase(1,
myLongitude.getName(),DataArray[1].toDouble(),
myLongitude.checkWarnings(DataArray[1].toInt()),
            LocalDateTime.now(),this@MainActivity)
       }
       "La" -> {
           myLatitude.addToDatabase(1, myLatitude.getName(),DataArray[1].toDouble(),
myLatitude.checkWarnings(DataArray[1].toInt()),
               LocalDateTime.now(),this@MainActivity)
       }
       "t" -> {
           myTime.addToDatabase(1, myTime.getName(),DataArray[1].toDouble(),
myTime.checkWarnings(DataArray[1].toInt()),
               LocalDateTime.now(),this@MainActivity)
       }
   }
}
```

1.2 Sensor Object

The sensor object is where we process all of the data received. This is the meeting point and gateway of three main contributors, the incoming data from puck, the database, and the UI. We use the sensor object to check for warning of the incoming data, to generate data for a sensor, to store the data to the database, retrieve it from the database, and to turn it into different types to print on UI.

1.2.1 Sensor Class Set-up

The sensor class set-up is a simple and elegant way to process the data based on which sensor the data falls under. We have an abstract sensor class called the Sensors to do some basic processing that is common for all the sensors. Then we have a class for each sensor to break down the data individually.

The abstract sensor class declares the abstract methods that are used in each sensor class along with common methods that are implemented for all the sensors.
It is declared as the following:

```
abstract class Sensors(id: Int, name: String, data: Any, warning: Int, time: LocalDateTime?)
{
    //abstract methods that can be edited for each sensor class
    abstract fun addToDatabase(id: Int, name: String, data: Double, warning: Int,
                               time: LocalDateTime?, context: Context)
    abstract fun getFromDatabase(): String
    abstract fun getWarningFromDatabase(): Int
    abstract fun checkWarnings(checkData: Int)
    abstract fun getHistorical(): ArrayList<String>
    abstract fun dummyProvider(personID: Int, context: Context)

    //method to get the id of the person
    fun getID(): Int {}

    //method to get the name of sensor
    fun getName(): String {}

    //method to get the data
    fun getData(): Any {}

    //method to get the warning int of the data
    fun getWarning(): Int {}

    //method to get the time the data came in
    fun getTime(): LocalDateTime? {}

}
```

The abstract methods are then expanded in each of the sensor classes for further breakdown of the data. When a new sensor is added, a new sensor class is created.
An example of the sensor class is as follows:

```
//class for a sensor
```

```
class SensorName(id: Int, name: String, data: Int, warning: Int, time:
LocalDateTime?)
                : Sensors(id, name, data, warning, time)
{

    //method to add the incoming data of a sensor to database
    override fun addToDatabase(id: Int, name: String, data: Double, warning: Int,
                               time: LocalDateTime?, context: Context) {}

    //method to get the latest data of a sensor from the database
    override fun getFromDatabase(): String {}

    //method to get the warning of a data for the sensor from the database
    override fun getWarningFromDatabase(): Int {}

    //method to check the warning of incoming data for the sensor
    override fun checkWarnings(checkData: Int) {}

    //method to get the historical data of the sensor from database
    override fun getHistorical(): ArrayList<String> {}

    //method to randomly generate the data for the sensor
    //use only for teammate data or fake sensors
    override fun dummyProvider(personID: Int, context: Context) {}
}
```

1.2.2 Adding Incoming Data to Database

Inside the sensor class, we have a method to add each incoming data to the database.
The method is set-up to open the database using the DBHandler then use the addNewSensorData method to insert a record to the database. The method requires an id, name of sensor, data as a double, warning indicator as an integer, and time of the data creation. The method will also check for warnings and insert the data into the warning table if the warning indicator indicates that the data is of the warning manner.
An example of this method:

```
//method to add the data of a sensor to database
    @RequiresApi(Build.VERSION_CODES.O)
    fun addToDatabase(id: Int, name: String, data: Double, warning: Int, time:
LocalDateTime?, context: Context) {
        val dbHandler = DBHandler(context)
        //val index = sensorNames.getIndexOf(name)

dbHandler.addNewSensorData(id.toString(),name,data.toString(),warning.toString(),tim
e.toString())

        /**val index = sensorNames.indexOf(name)
        if (warning == 1 && id ==1) {
            addWarningData(id,name,data,warning,time,context)
        }
        else if (sensorNamesWarningCheck[index] == 1) {
            addWarningData(id,name,data,warning,time,context)
        }*/

    }
```

1.2.3 Getting Latest Data from Database

Inside each of the sensor classes, we have a method to get the latest data. The contents of this method will differ based on the sensor.
The method is set-up to call on the dbHandler method to open the database and use the readLatestPerSensor to get the latest data. Once the data is received we will have to format it to a string based and add the unit at the end. The return data of the method will be as follows: 25 °C.
Note: we will need a context passed into this method to open the database.
An example of this method based on temperature sensor:

```
//method to get the data of a sensor from the database
   @RequiresApi(Build.VERSION_CODES.O)
   override fun getFromDatabase(context: Context, personID: Int): String {
       var returnData = ""

       try{
           val dbHandler: DBHandler = DBHandler(context)
           var latestData: ArrayList<SensorModelDB> = ArrayList<SensorModelDB>()
           latestData = dbHandler.readLatestPerSensor("Temperature", personID)
           //If condition to return empty string if the table has size 0
           if (dbHandler.readLatestPerSensor("Temperature", personID).size == 0)
           {
               return returnData;
           }
           returnData = latestData.get(0).data + " °C"
       }
       catch (e: Exception){
           //handle exception
       }

       return returnData
   }
```

1.2.4 Getting Historical Data from Database

Inside each of the sensor classes, we have a method to get the historical data. The contents of this method will differ based on the sensor.
The method is set-up to call on the dbHandler method to open the database and use the readAllPerSensor to get the historical data. We will receive an arraylist of data. Once the data is received, we save it to the return value and return an arraylist of strings to put on the graphs.
Note: we will need a context passed into this method to open the database.
An example of this method based on temperature sensor:

```
//method to get the historical data of the sensor
    override fun getHistorical(context: Context, personID: Int): ArrayList<String> {
        val tempDataArray: ArrayList<String> = arrayListOf()
        val tempWarningArray: ArrayList<Int> = arrayListOf()
        try{
            val dbHandler: DBHandler = DBHandler(context)
            var latestData: ArrayList<SensorModelDB> = ArrayList<SensorModelDB>()
            latestData = dbHandler.readAllPerSensor("Temperature", personID)

            val size = latestData.size - 1
```

```
            for (i in 0..size){
                val m = latestData[i]
                if (m.sensorName == "Temperature")
                {
                    tempDataArray.add(m.data)
                    tempWarningArray.add(m.warning.toInt())
                }
            }
        }
        catch (e: Exception){
            //handle exception
        }

        return tempDataArray
    }
```

1.2.5 Warning

Inside each of the sensor classes, we have a checkWarnings() method to check the warning of each incoming data. The contents of this method will differ based on the sensor. The client will have to provide the developer with boundaries to set up in order for this to work.
The method is set-up to check if the data provided is safe or not. It will take in the incoming data as the parameter. It will then check if the data is out of the boundary provided, if so return a 1. If it is safe, it will return a 0. If all the values are safe, or if the client fails to provide the boundaries, then it should always return a 0.
An example of this method based on temperature sensor:

```
override fun checkWarnings(checkData: Int): Int
{
        if(checkData > 50){
            tempWarning = 1
        }
        else if(checkData < -40){
            tempWarning = 1
        }
        else {
            tempWarning = 0
        }
        return tempWarning
}
```

1.2.6 Dummy Provider

Inside each of the sensor classes, we have a dummyProvider method to generate data for each sensor. This will be used to generate data, if the sensor is a fake sensor, or if data needs to be generated for each teammate. The contents of this method will also differ based on the sensor. The upper and lower bound for the sensor needs to be provided by the client. The method will take in the team member ID and the context of where it is called as the parameters. It will then generate the data and call on another method to add to the database.
An example of this method based on temperature sensor:

```
override fun dummyProvider(personID: Int, context: Context)
```

```
{
        val current = LocalDateTime.now()
        val temperature = (-40..80).random()
        checkWarnings(temperature)
        val enter = Temperature(personID, "Temperature", temperature,
                                tempWarning, current)
        tempHistoricalDatabase.add(enter)
}
```

1.3 Teammates

In this section, we will discuss how to set up a team, initialize it, and generate data for it.

1.3.1 Initialize a Team

Initializing and generating a team is done in the teammates.kt file. This is done through arrays, which will help for further data processing in the future. The name of the team members will be stored in an array. There will be IDs associated with the teammate, which will be stored in another array. Both arrays are then put into a single array to create a 2D array.
An example of this process is:

```
val teammateName = arrayOf("Me", "Baker", "Carter", "Dean", "John", "Tim")
val teammateID = arrayOf(1, 2, 3, 4, 5, 6)
val squad  = arrayOf(teammateID,teammateName)
```

1.3.2 Generate Data for a Team

First, we will have to initialize all the sensor classes inside the teammates.kt file and save it to an array. These could be used throughout the app for other data processing. Each time a new sensor is added, we will have to initialize it here.
An example of this process is:

```
val sensorArray: ArrayList<Sensors> = arrayListOf<Sensors>()
val myTemp = Temperature(1, "Temperature", 0, 0,null)
sensorArray.add(myTemp)
```

After completing the above steps, generating the data for each team member could be done through the generateTeamData() method. This method will take in the context as the parameter. It should be called in the Main Activity to generate the data at app startup.
An example of this process is:

```
fun generateTeamData(context: Context)
{
	//get the number of teammates
	val size2 = squad[0].size - 1
	//generate 4 data for each teammate
	for (i in 1..4)
	{
		//generate data for each of the teammate
		for (j in 0..size2)
		{
			//get the teammate ID
			val id: Int = squad[0][j] as Int
			//generate data for each sensor
			for (tempSensor in sensorArray)
			{
				//call on dummyProvider to generate data
				tempSensor.dummyProvider(id, context)
			}
		}
	}
}
```

2.0 Backend - Database

The following section will walkthrough how to set up the database in the app.

For the database setup, we used the methods found in the book "Android Programming for Beginners" written by John Horton, and published by Packt. This is the github repository for the code used in the book

2.1 Database Set-up

First of all, we created the class DBHandler, which is essentially our SQLiteOpenHelper. The SQLiteOpenHelper is a class used for creating the database and for version management. It provides us with the onCreate and onUpgrade methods, which take care of opening the database if it does exist, creating it if it does not, and upgrading it when necessary. These are all essential steps in order to perform any database operation.

2.1.1 Table Creations

Our application's database consists of two tables. One is called mySensor, which is our main table where we store all our data except for the warning data. The warning data is stored in its own separate table called mywarnings. We create the two mentioned tables via two SQL queries,

```
//creating a constructor for our database handler that handles all the database code
class DBHandler (context: Context?): SQLiteOpenHelper(context, DB_NAME, null, DB_VERSION) {

    //method for creating a database by running a sqlite query
    override fun onCreate(db: SQLiteDatabase) {

        //create an sqlite query and we are setting our column names along with their data types
        //main table to store all data
        val query = ("CREATE TABLE " + TABLE_NAME_SENSORS + " ("
                + ID_COL + " INTEGER PRIMARY KEY AUTOINCREMENT,"
                + USERID_COL + " TEXT,"
                + NAME_COL + " TEXT,"
                + DATA_COL + " TEXT,"
                + WARNING_COL + " TEXT,"
                + TIME_COL + " TEXT)")

        //create an sqlite query and we are setting our column names along with their data types
        //sub table to store only the warning data
        val query1 = ("CREATE TABLE " + TABLE_NAME_WARNINGS + " ("
                + ID_COL_1 + " INTEGER PRIMARY KEY AUTOINCREMENT,"
                + WARNING_USER_COL + " TEXT,"
                + WARNING_NAME_COL + " TEXT,"
                + WARNING_DATA_COL + " TEXT,"
                + WARNING_WARNING_COL + " TEXT,"
                + WARNING_TIME_COL + " TEXT)")

        //call a exec sql method to execute above sql queries
        db.execSQL(query)
        db.execSQL(query1)
    }
```

"query" is the one that creates the main table, and "query1" creates the warning table

2.1.2 Column Creation

Our database's columns are created within the DBHandler method above. Our main table has 6 columns, which are:

1. ID Column, which is the unique ID generated for each sensor, and is auto incremented.
2. UserID column, which is the unique ID each user has. This helps our application connect the sensor reading of each sensor to its respective user.
3. Name column, which is the name of the sensor
4. Data column, which is the data value read by the sensor
5. Warning Column, where the warning status is stored

6. Time column, which is the time in which the data value is read and saved into the database

Our second table also has 6 columns, which are as follows:

1. ID column, which is auto incremented to give a unique ID for each warning value
2. Warning_User column, which stores which user has the warning displayed
3. Warning_Name, which is the name of the sensor giving the warning
4. Warning_Data, which is the value of the sensor that caused the warning
5. Warning_Warning, which is the name of the warning. For example, if the warning is caused by a low heart rate, the value saved in this column would be heart rate.
6. Warning_time, which is the column where we save the time the warning got saved into the database.

2.2 Database Methods Set-up

Our way of implementing the database is similar to what is found in the book mentioned above. Each database operation has its own method, such as inserting, updating, retrieving and updating the data. Our application required us to have extra methods, such as fetching data saved only in the last 10 minutes, etc. These methods are explained in more detail below.

2.2.1 Adding data to database

The below method is called upon whenever we need to save new data into the main table.
It takes the 6 column fields of the table as a parameter, we then use ContentValues, which is a "maplike class that matches a value to a String key", to insert the values into their respective column

```
fun addNewSensorData( userID: String?, sensorName: String?, data: String?, warning: String?, Time: String? ):
Long {

    //variable for our sqlite database and calling writable method as we are writing data in our database
    val db = this.writableDatabase

    //variable for content values
    val values = ContentValues()

    //pass all values along with its key and value pair
    values.put(USERID_COL, userID)
    values.put(NAME_COL, sensorName)
    values.put(DATA_COL, data)
    values.put(WARNING_COL, warning)
    values.put(TIME_COL, Time)

    //pass content values to our table
    val returnValue = db.insert(TABLE_NAME_SENSORS, null, values)

    //close our database
    db.close()

    //return value - if (-1) then error; if > 0 then success
    return returnValue
}
```

2.2.2 Adding warning data to database

The below method is called upon whenever we need to save new data into the warning table. Much like the method above, it takes 6 parameters which represent the 6 columns in the warning table, and it then uses ContentValues to insert the values into their respective column

```
fun addNewWarningData( userID: String?, sensorName: String?, data: String?, warning: String?, Time: String? ):
Long {

    //variable for our sqlite database and calling writable method as we are writing data in our database
    val db = this.writableDatabase

    //variable for content values
    val values = ContentValues()

    //pass all values along with its key and value pair
    values.put(WARNING_USER_COL, userID)
    values.put(WARNING_NAME_COL, sensorName)
    values.put(WARNING_DATA_COL, data)
    values.put(WARNING_WARNING_COL, warning)
    values.put(WARNING_TIME_COL, Time)

    //pass content values to our table
    val returnValue = db.insert(TABLE_NAME_WARNINGS, null, values)

    //close our database
    db.close()

    //return value - if (-1) then error; if > 0 then success
    return returnValue
}
```

2.2.3 Getting data from database

We have multiple methods to fetch data from the database based on different requirements. These can all be found below

This is the method used to read ALL of the data in our main table. It returns an arrayList SensorModelArrayList, which contains all the data.

```
//method for reading all the sensor data regardless of sensor name
    fun readAllData(): ArrayList<SensorModelDB>? {

        //database variable for reading our database
        val db = this.readableDatabase

        //cursor with query to read data from database
        val sensorCursor: Cursor = db.rawQuery("SELECT * FROM $TABLE_NAME_SENSORS", null)

        //array list for storing the data
        val sensorModelArrayList: ArrayList<SensorModelDB> = ArrayList()

        //moving our cursor to first position
        if (sensorCursor.moveToFirst()) {
            do {
                //add the data from cursor to our array list
                sensorModelArrayList.add(
                    SensorModelDB(
                        sensorCursor.getString(1),
                        sensorCursor.getString(2),
                        sensorCursor.getString(3),
                        sensorCursor.getString(4),
                        sensorCursor.getString(5)
                    )
                )}
                while (sensorCursor.moveToNext())
            //move our cursor to next
        }

        //close our cursor
        sensorCursor.close()

        //returning our array list with all the data
        return sensorModelArrayList
    }
```

This is the method used to fetch the latest data from the main table based on the name of a sensor. It takes the name of the sensor as a parameter, and returns an ArrayList called sensorModelArrayList with all the values

```
fun readLatestPerSensor(sensorName: String): ArrayList<SensorModelDB> {

    //database variable for reading our database
    val db = this.readableDatabase

    //call on generateLatestData to generate a query because doing it here gave errors on the WHERE
    clause val tempQ = generateLatestData(sensorName) + " ORDER BY $ID_COL DESC LIMIT 1"

    //cursor with query to read data from database based on query generated above
    val sensorCursor: Cursor = db.rawQuery(tempQ, null)

    //array list for storing the data
    val sensorModelArrayList: ArrayList<SensorModelDB> = ArrayList()

    //moving our cursor to first position
    if (sensorCursor.moveToFirst()) {
       do {
          //add the data from cursor to our array list
          sensorModelArrayList.add(
             SensorModelDB(
                sensorCursor.getString(1),
                sensorCursor.getString(2),
                sensorCursor.getString(3),
                sensorCursor.getString(4),
                sensorCursor.getString(5)
             )
          )
       } while (sensorCursor.moveToNext())
       //move our cursor to next
    }

    //close our cursor
    sensorCursor.close()

    //returning our array list with all the data
    return sensorModelArrayList
}
```

Below is the method used to fetch data based off of a user's ID and a sensor's name. This method is used when we want the specific sensor value of a specific user. For example, if we want to fetch the heart rate value of the user "Baker". This method takes the name of the sensor and the userID as parameters, and returns an arrayList with the results.

```
fun retrieveDataBySenorAndUser(sensorName: String,userID: String):
    ArrayList<SensorModelDB>{ val sensorModelArrayList: ArrayList<SensorModelDB> =
    readAllData()!!
    val tempList: ArrayList<SensorModelDB> = ArrayList()
    val retrieveDataList: ArrayList<SensorModelDB> = ArrayList()
    var i = 0
    while (i <sensorModelArrayList.size){
      if (sensorModelArrayList[i].userID==userID){
        tempList.add(sensorModelArrayList[i])
      }
      i++
    }
    i =0
    while (i <tempList.size){
      if (tempList[i].sensorName==sensorName){
        retrieveDataList.add(tempList[i])
      }
      i++
    }
    return retrieveDataList
  }
```

2.2.4 Getting warning data from database

The following method reads the latest warning of a specific sensor, based on that sensor's name. It takes the name of the sensor as a parameter and returns string returnValue

```
//method for reading latest warning of one sensor each
    fun readLatestWarningPerSensor(sensorName: String): String {

        //database variable for reading our database
        val db = this.readableDatabase

        //call on generateLatestData to generate a query because doing it here gave errors on the WHERE
        clause val tempQ = generateLatestData(sensorName) + " ORDER BY $ID_COL DESC LIMIT 1"

        //cursor with query to read data from database based on query generated above
        val sensorCursor: Cursor = db.rawQuery(tempQ, null)

        //string for storing the data
        var returnValue = ""

        //moving our cursor to first position
        if (sensorCursor.moveToFirst()) {
            do {
                //add the data from cursor to our string
                returnValue = sensorCursor.getString(4)
            } while (sensorCursor.moveToNext())
            //move our cursor to next
        }

        //close our cursor
        sensorCursor.close()

        //returning our string with the data
        return returnValue
    }
```

2.2.5 Getting historical data from database

Historical data in our application is retrieved inside of the sensor object. Each sensor object has a method called “getHistorical” inside of it. The readAllPerSensor method is called, and the result is added into an arrayList called tempDataArray. In order to fetch only the historical data of that sensor, we start adding after decreasing the array size by one, to ignore the latest added data for that sensor. The code for the altitude sensor can be seen below, the method is nearly identical for each sensor

```
override fun getHistorical(context: Context, personID: Int): ArrayList<String> {
    val tempDataArray: ArrayList<String> = arrayListOf()
    val tempWarningArray: ArrayList<Int> = arrayListOf()
    try{
        val dbHandler: DBHandler = DBHandler(context)
        var latestData: ArrayList<SensorModelDB> = ArrayList<SensorModelDB>()
        latestData = dbHandler.readAllPerSensor("Altitude", personID)

        val size = latestData.size - 1
        for (i in 0..size){
            val m = latestData[i]
            if (m.sensorName == "Altitude")
            {
                tempDataArray.add(m.data)
                tempWarningArray.add(m.warning.toInt())
            }
        }
    }
    catch (e: Exception){
        //handle exception
    }

    return tempDataArray
}
```

To retrieve data saved only within the last 10 minutes, we generate a SQLite query based on the name of the sensor the user has chosen. Part of this code is found below, the rest of the code are only if cases, one for each sensor in our application

```
fun generateTenMinutesData(sensorName: String, usersID: Int): String {

    //variable for storing the query
    var tempQ = ""
    val tempUserID = usersID.toString()

    //change the query based on the sensor name input
    if (sensorName == "Temperature"){
        tempQ = "SELECT * FROM $TABLE_NAME WHERE $NAME_COL = 'Temperature' AND $USERID_COL = $tempUserID" +
                " AND $TIME_COL > datetime('now', '-10 minutes')"
    }
    else if(sensorName == "Humidity"){
        tempQ = "SELECT * FROM $TABLE_NAME WHERE $NAME_COL = 'Humidity' AND $USERID_COL = $tempUserID" +
                " AND $TIME_COL > datetime('now', '-10 minutes')"
    }
    else if(sensorName == "Pressure"){
        tempQ = "SELECT * FROM $TABLE_NAME WHERE $NAME_COL = 'Pressure' AND $USERID_COL = $tempUserID" +
                " AND $TIME_COL > datetime('now', '-10 minutes')"
    }
    else if(sensorName == "Altitude"){
        tempQ = "SELECT * FROM $TABLE_NAME WHERE $NAME_COL = 'Altitude' AND $USERID_COL = $tempUserID" +
                " AND $TIME_COL > datetime('now', '-10 minutes')"
    }
```

The generated SQLite query is then executed in a method called readTenMinutes, found below.

```
fun readTenMinutes(sensorName: String, usersID: Int): ArrayList<SensorModelDB> {
    //database variable for writing our database
    val db = this.writableDatabase

    val tempQ = generateTenMinutesData(sensorName, usersID) + " ORDER BY $ID_COL DESC LIMIT 100"
    //cursor with query to read data from database based on query generated above
    val sensorCursor: Cursor = db.rawQuery(tempQ, null)

    //array list for storing the data
    val sensorArrayList: ArrayList<SensorModelDB> = ArrayList();
    //moving our cursor to first position
    if (sensorCursor.moveToFirst()) {
        do {
            //add the data from cursor to our array list
            sensorArrayList.add(
                SensorModelDB(
                    sensorCursor.getString(1),
                    sensorCursor.getString(2),
                    sensorCursor.getString(3),
                    sensorCursor.getString(4),
                    sensorCursor.getString(5)
                )
            )
        } while (sensorCursor.moveToNext())
        //move our cursor to next
    }

    //close our cursor
    sensorCursor.close()
    //close the database
    db.close()
    //returning our array list with all the data
    return sensorArrayList
}
```

The readTenMinutes method executes the generated SQLite query generated by the generateTenMinutesData method, and adds the result into an array called sensorArrayList, which it returns. The method returns only the data saved within the last 10 minutes because of the part after the AND clause in the generateTenMinutesData.” >datetime(‘now’, ‘-10 minutes’)” ensures that the SQLite query only returns data generated within the last 10 minutes.

2.2.6 Getting historical warning data from database

Historical warning data is similar to the retrieval of sensor data mentioned above. In our application, waning data is retrieved inside of the readAllPerSensorobject. Each sensor object has a method called “getHistoricalWarning” inside of it. The readAllPerSensor method is called, and the result is added into an arrayList called tempWarningArray. In order to fetch only the historical warning data of that sensor, we start adding after decreasing the array size by one, to ignore the latest added data for that sensor. The code inside of the altitude sensor can be seen below, the method is nearly identical for each sensor

```
override fun getHistoricalWarning(context: Context, personID: Int): ArrayList<Int> {
    val tempDataArray: ArrayList<String> = arrayListOf()
    val tempWarningArray: ArrayList<Int> = arrayListOf()
    try{
        val dbHandler: DBHandler = DBHandler(context)
        var latestData: ArrayList<SensorModelDB> = ArrayList<SensorModelDB>()
        latestData = dbHandler.readAllPerSensor("Altitude", personID)

        val size = latestData.size - 1
        for (i in 0..size){
            val m = latestData[i]
            if (m.sensorName == "Altitude")
            {
                tempDataArray.add(m.data)
                tempWarningArray.add(m.warning.toInt())
            }
        }
    }
    catch (e: Exception){
        //handle exception
    }

    return tempWarningArray
}
```

2.2.7 Deleting all data from tables

We have two methods that delete all the data, one for each table in our application. The method used to delete all data from the main table is found below

```
fun deleteAll() {

    //database variable for writing our database
    val db = this.writableDatabase

    //delete all data in the table
    db.execSQL("delete from "+ TABLE_NAME)

    //close database
    db.close()
}
```

And the method used to delete all data from the warning table is as follows:

```
fun deleteAllWarning() {

    //database variable for writing our database
    val db = this.writableDatabase

    //delete all data in the table
    db.execSQL("delete from "+ TABLE_NAME_1)

    //close database
    db.close()
}
```

2.2.8 Exporting all data from tables

We created a dedicated function that helps facilitate the process of exporting data from tables, these functions allow the users to export data from the application to internal storage in "sixthSenseImports" directory an auto generated directory that contains excel files generated. Additionally there is one helper method that we implemented to help with exporting the data.

Bellow you can find the code for exportToCsv()helper method that is responsible for getting data from the database

```
fun exportToCSV(context: Context) {
    try{
        val dbHandler: DBHandler = DBHandler(context)
        userList = dbHandler.readAllData()!!
        if (userList.size > 0) {
            Toast.makeText(context, "starting export", Toast.LENGTH_SHORT).show()
            createXlFile(context)
        } else {
            Toast.makeText(context, "list are empty", Toast.LENGTH_SHORT).show()
        }
    }
    catch (e: Exception){
        //handle exception
        Toast.makeText(context, "failed", Toast.LENGTH_SHORT).show()
    }
}
```

As shown above the exportToCSV function is responsible for getting the data from the database by calling the readAllData function, that reads all sensor data from the database, then checks whether the list is empty or not, if the userlist is not empty the function displays a message to the user saying “Starting export” using Toast.makeText method then calls the createXlFile function that creates the actual excel file and the sixthsenseImports directory on the Android device.

The createXlFile function uses Apache POI library using the HSSFWorkbook function to create the excel file as shown bellow

```
val wb: Workbook = HSSFWorkbook()
var cell: Cell? = null
var sheet: Sheet? = null
sheet = wb.createSheet("Export")
```

Then it creates a header row for columns (index, UserID, Sensor Name, Warning Indicator, Time) using the createRow() function as shown bellow.

```
val row: Row = sheet.createRow(0)
        cell = row.createCell(0)
        cell.setCellValue("Index")
        cell = row.createCell(1)
        cell.setCellValue("User ID")
        cell = row.createCell(2)
        cell.setCellValue("Sensor Name")
        cell = row.createCell(3)
        cell.setCellValue("Data")
        cell = row.createCell(4)
        cell.setCellValue("Warning Indicator")
        cell = row.createCell(5)
        cell.setCellValue("Time")
'
```

Then sets the size for the columns using the setColumnWidth method as shown bellow

```
  sheet.setColumnWidth(0, 20 * 200)
  sheet.setColumnWidth(1, 30 * 200)
  sheet.setColumnWidth(2, 30 * 200)
  sheet.setColumnWidth(3, 30 * 200)
  sheet.setColumnWidth(4, 30 * 200)
  sheet.setColumnWidth(5, 30 * 200)
```

After setting the the columns width, it iterates over userlist then creates a row in the excel file populating the cell with user object then resize the columns

```
 for (i in 0 until userList.size) {
            val row1: Row = sheet.createRow(i + 1)
            cell = row1.createCell(0)
            cell.setCellValue(i.toString())
            cell = row1.createCell(1)
            cell.setCellValue(userList.get(i).userID)
            cell = row1.createCell(2)
            cell.setCellValue(userList.get(i).sensorName)
            cell = row1.createCell(3)
            cell.setCellValue(userList.get(i).data)
            cell = row1.createCell(4)
            cell.setCellValue(userList.get(i).warning)
            cell = row1.createCell(5)
            cell.setCellValue(userList.get(i).Time)
            sheet.setColumnWidth(0, 20 * 200)
            sheet.setColumnWidth(1, 30 * 200)
            sheet.setColumnWidth(2, 30 * 200)
            sheet.setColumnWidth(3, 30 * 200)
            sheet.setColumnWidth(4, 30 * 200)
            sheet.setColumnWidth(5, 30 * 200)
        }
```

Then creates a directory on the Android device named “sixthsenseImports” and names the excel file name based on the device current date and time using LocalDatetime and DateTimeFormater

```
val folderName = "SixthSenseImports"
 val fileName = folderName + LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyyMMMdd_HH-mm-ss")) + ".xls"
```

Then the function creates a new excel file and a File object corresponding to it using specified path, then checks if the directory for the excel file exists. Next it creates a FileOutputStream and writes the workbook to the stream using the write method.

```
val path: String =
            Environment.getExternalStorageDirectory().toString() + File.separator.toString() + folderName + File.separator.toString() + fileName
        val path2: String =
            File.separator.toString() + folderName + File.separator.toString() + fileName
        val file =
            File(Environment.getExternalStorageDirectory().toString() + File.separator.toString() + folderName)
        if (!file.exists()) {
            file.mkdirs()
        }
        var outputStream: FileOutputStream? = null
        try {
            outputStream = FileOutputStream(path)
            wb.write(outputStream)
            // ShareViaEmail(file.getParentFile().getName(),file.getName());
            Toast.makeText(context, "Excel Created in $path2", Toast.LENGTH_SHORT).show()
        } catch (e: IOException) {
            e.printStackTrace()
            Toast.makeText(context, "Not OK", Toast.LENGTH_LONG).show()
            try {
                if (outputStream != null) {
                    outputStream.close()
                }
            } catch (ex: Exception) {
                ex.printStackTrace()
            }
```

3.0 Frontend - UI

The following section will walkthrough how to set up the UI in the app.

3.1 Real-time Set-up

Real-Time Screen with its efficient and user-friendly interface, Realtime Screen provides an intuitive platform for monitoring, analyzing, and visualizing real-time data from various sensors. This documentation serves as a guide for developers to know the structure of the implementation of the Respective UI Screen.This Class contains the three function which are RealTimeScreen(),RealTimeMainUser() and RealTimeTeammate().Furthermore, RealTimeScreen() setup the screen panel to accommodate the other two Screen For the MainUser and Teammate Panels.

3.1.1 RealTimeScreen

Overview

This is a composable function written in Kotlin that creates a Real-Time Screen. The screen has a top app bar, a list of teammates represented by their name, a dropdown menu, and action buttons that can be used to export data, change the view of the graph, and display a consent dialog.

Function signature

```
@RequiresApi(Build.VERSION_CODES.Q)
@SuppressLint("UnusedMaterialScaffoldPaddingParameter")
@Composable
 fun RealTimeScreen()
```

- This function is annotated with @RequiresApi to specify that the minimum API level required to run this code is Q (29).
- @SuppressLint("UnusedMaterialScaffoldPaddingParameter") is used to suppress a lint warning.
- This function is a composable function that returns a UI element.

Parameters

This function has no parameters.

Returns

This function returns a UI element.

Dependencies

This function uses the following classes:

- Scaffold: A Material Design layout component that implements the basic material design visual structure, such as top app bar, floating action button, and drawer.
- TopAppBar: A Material Design component that represents a toolbar with text and icon buttons.
- Text: A composable function that displays text on the screen.
- IconButton: A Material Design component that represents a clickable icon button.

- Icon: A composable function that displays an icon.
- Intent: A class that provides an intent that can be used to start another activity or service.

Code flow

1. The Scaffold composable function is called with a modifier that sets the background color to transparent and rounds the corners of the element.
2. The topBar parameter of the Scaffold function is set to a TopAppBar composable function that has a title, elevation, and actions.
3. The title parameter of the TopAppBar function is set to a Text composable function that displays the text "Live Data".
4. The actions parameter of the TopAppBar function is set to a Row composable function that contains a Column and two IconButtons.
5. The Column composable function contains a LazyHorizontalGrid composable function that displays a list of teammate names represented by icons.
6. The IconButton composable function that is the second child of the Row composable function has an onClick listener that toggles the view of a sub-graph.
7. The third IconButton composable function displays an icon for exporting data and has an onClick listener that displays a consent dialog.
8. If the consent dialog is displayed, it is created using a custom DialogAlert composable function that has two onClick listeners for the "Yes" and "No" buttons.
9. When the "Yes" button of the consent dialog is clicked, data is exported to a CSV file using a custom DBHandler class.
10. If the teammate name icon is clicked, the selected teammate's data is displayed on the graph.

```
@RequiresApi(Build.VERSION_CODES.Q)
@SuppressLint("UnusedMaterialScaffoldPaddingParameter")
@Composable
fun RealTimeScreen() {
   Scaffold(
       modifier = Modifier
           .background(Color.Transparent)
           .clip(RoundedCornerShape(10.dp)),
       topBar = {
           TopAppBar(modifier = Modifier
               .padding(3.dp)
               .fillMaxWidth()
               .fillMaxHeight(.10f)
               .graphicsLayer {
                   shape = RoundedCornerShape(12.dp)
                   clip = true
               }, elevation = 13.dp, title = {
               Text(
                   text = "Live Data",
                   color = Color.White,
                   fontFamily = customfont,
                   modifier = Modifier.padding(3.dp),
                   fontSize = 18.sp,
```

```
                    overflow = TextOverflow.Visible,
                    letterSpacing = 1.sp

                )
            },
                actions = {

                    //create the row layout to arrange the action button and dropdownMenu
                    Row(
                        modifier = Modifier
                            .fillMaxWidth(.85f)
                            .fillMaxHeight()
                            .padding(1.dp)
                            .background(Color.Transparent),
                        horizontalArrangement = Arrangement.End,
                        verticalAlignment = Alignment.Top
                    ) {

                        Column(
                            modifier = Modifier
                                .fillMaxWidth(.80f)
                                .fillMaxHeight()
                                .padding(1.dp)
                                .clip(RoundedCornerShape(10.dp)),
                            verticalArrangement = Arrangement.Center,
                            horizontalAlignment = Alignment.CenterHorizontally
                        ) {
                            LazyHorizontalGrid(
                                GridCells.Fixed(1), Modifier
                                    .fillMaxSize(), horizontalArrangement =
Arrangement.Start
                            ) {
                                items(teammateName.size) { index ->
                                    IconButton(modifier = Modifier
                                        .size(95.dp, 45.dp)
                                        .padding(1.dp)
                                        .background(Color.Transparent), onClick = {
                                        selectedMember = teammateName[index]
                                        changeView = 0
                                        if (selectedMember == "Me") {
                                            closeMyData = 0
                                            closeTeammate = 1
                                        } else {
                                            closeTeammate = 0
                                        }
                                        val navigate = Intent(
                                            this@HomeActivity, HomeActivity::class.java
                                        )
                                        startActivity(navigate)
                                    }) {
                                        Row {
                                            val pin = Teammates().returnPins(
                                                teammateName[index],
                                                this@HomeActivity
                                            )
                                            val color =

Teammates().returnPinsColor(teammateName[index])
                                            Icon(
                                                painter = painterResource(pin),
                                                "Pins",
                                                modifier = Modifier
```

```
                                                .padding(1.dp)
                                                .size(20.dp),
                                            tint = color
                                        )
                                        //create the space for size of 3.Dp
                                        Spacer(modifier = Modifier.size(3.dp))

                                        Text(
                                            teammateName[index], color = Color.White
                                        )
                                        //create the space for size of 3.Dp
                                        Spacer(modifier = Modifier.size(3.dp))

                                        val warningSign =
SensorsUI().GetWarningLightInt(
                                            this@HomeActivity, teammateName[index]
                                        )
                                        if (warningSign == 0) {
                                            Box(
                                                modifier = Modifier
                                                    .clip(CircleShape)
                                                    .background(Color.Green)
                                                    .size(15.dp, 15.dp)
                                                    .align(Alignment.CenterVertically)
                                            )
                                        } else if (warningSign == 1) {
                                            Box(
                                                modifier = Modifier
                                                    .clip(CircleShape)
                                                    .background(Color.Red)
                                                    .size(15.dp, 15.dp)
                                                    .align(Alignment.CenterVertically)

                                            )
                                        }
                                    }
                                }
                            }
                        }

                        IconButton(modifier = Modifier
                            .size(55.dp, 45.dp)
                            .padding(1.dp)
                            .background(Color.Transparent), onClick = {

                            if (subGraphScreen == 0 && subGraphScreen2 == 0) {
                                subGraphScreen = 1
                                subGraphScreen2 = 1
                            } else {
                                subGraphScreen = 0
                                subGraphScreen2 = 0
                            }

                            val navigate = Intent(
                                this@HomeActivity, HomeActivity::class.java
                            )
                            startActivity(navigate)
                        }) {
                            Icon(

                                painter = painterResource(R.drawable.eyeicon),
```

```
                            "Change View",
                            modifier = Modifier
                                .fillMaxSize()
                                .padding(4.dp),
                            tint = Color.White
                        )
                    }
                    //create the space for size of 3.Dp
                    Spacer(modifier = Modifier.size(3.dp))
                    val showConsentDialog = remember { mutableStateOf(false) }
                    if (showConsentDialog.value) {
                        DialogAlert().ConsentDialog(
                            onYesClick = {
                                val dbHandler = DBHandler(this@HomeActivity)
                                dbHandler.exportToCSV(this@HomeActivity)
                                showConsentDialog.value = false
                            },
                            onNoClick = { showConsentDialog.value = false }
                        )
                    }

                    IconButton(modifier = Modifier
                        .size(55.dp, 45.dp)
                        .padding(1.dp)
                        .background(Color.Transparent),
                        onClick = {
                            showConsentDialog.value = true
                        }
                    )

                    {
                        Icon(

                            painter = painterResource(R.drawable.export),
                            "Export",
                            modifier = Modifier
                                .fillMaxSize()
                                .padding(4.dp),
                            tint = Color.White
                        )
                    }
                    //create the space for size of 3.Dp
                    Spacer(modifier = Modifier.size(3.dp))
                }
            }


        )
    })
{
    MapboxScreen()
    Row(
        modifier = Modifier
            .fillMaxSize()
            .background(Color.Transparent)

            .padding(4.dp)
            .clip(RoundedCornerShape(15.dp))
    ) {
        Column(
            Modifier
                .fillMaxHeight()
```

```
                    .background(Color.Transparent)
                    .weight(0.5F)
                    .padding(2.dp)
                    .clip(RoundedCornerShape(15.dp))
            ) {
                if (closeMyData == 0) {
                    RealTimeMainUser()
                }
            }

            //create the space for size of 3.Dp
            Spacer(modifier = Modifier.size(4.dp))

            Column(
                Modifier
                    .fillMaxHeight()
                    .background(Color.Transparent)
                    .weight(0.5F)
                    .padding(2.dp)
                    .clip(RoundedCornerShape(15.dp))
            ) {
                if (closeTeammate == 0) {
                    RealTimeTeammate()
                }
            }
        }
    }

}
```

3.1.2 RealTimeMainUser

The code provided is a Kotlin function named RealTimeMainUser(). It is annotated with @SuppressLint("UnusedMaterialScaffoldPaddingParameter") to suppress warnings for an unused parameter in the Scaffold function. The function is also annotated with @Composable, indicating that it is a composable function that builds a UI element.

The function defines a Row composable that contains a Scaffold composable as its only child. The Scaffold composable is used to create a layout with a top app bar, which includes a title and some action buttons.
The top app bar is created using the TopAppBar composable. It has a modifier that sets its size and padding, and applies a RoundedCornerShape with a clip. The elevation parameter specifies the height of the app bar's shadow, and the title parameter sets the title text, font family, font size, and overflow behavior.

The actions parameter of the TopAppBar composable is a lambda that creates a Row composable. The Row contains four IconButton composables, each representing a different action button, and a Spacer composable between each button.

Each IconButton has a modifier that sets its size, weight, and padding, and an onClick lambda that specifies the behavior when the button is clicked. The first button has an Icon composable as its content, and the remaining three have a Text composable.

The Icon and Text composables use conditional logic based on the values of two mutable variables, changeViewMe and changeMapMe. If changeViewMe is equal to 0, the button's content is displayed with a green tint. If it is not 0, the button's content is displayed with a white tint. The Text composables also specify an overflow behavior.

The RealTimeMainUser() function does not return any values, as it is used to build a UI element with a Row and a Scaffold composable. The purpose of this function is to create a top app bar with action buttons, and provide functionality when each button is clicked.

```
  @SuppressLint("UnusedMaterialScaffoldPaddingParameter")
   @Composable
   fun RealTimeMainUser() {
       Row {
           Scaffold(topBar = {
               TopAppBar(modifier = Modifier
                   .fillMaxWidth()
                   .fillMaxHeight(.13f)
                   .padding(top = 3.dp, start = 3.dp, end = 3.dp, bottom = 3.dp)
                   .graphicsLayer {
                       shape = RoundedCornerShape(12.dp)
                       clip = true
                   }, elevation = 10.dp, title = {
                   Text(
                       text = "My Data",
                       color = Color.White,
                       fontFamily = customfont,
                       modifier = Modifier
                           .padding(2.dp)
                           .align(Alignment.CenterVertically),
                       fontSize = 15.sp,
                       overflow = TextOverflow.Ellipsis,

                       )
               }, actions = {

                   //create the row layout to arrange the action button and dropdownMenu
                   Row(
                       modifier = Modifier
                           .fillMaxWidth(0.80f)
                           .fillMaxHeight()
                           .padding(2.dp)
//                                 .border(BorderStroke(Dp.Hairline,Color.Red))
                       ,
                       horizontalArrangement = Arrangement.End,
                       verticalAlignment = Alignment.CenterVertically
                   ) {
                       // Health Action button
                       IconButton(modifier = Modifier
```

```
                            .padding(4.dp)
                            .size(60.dp, 60.dp)
                            .weight(0.5F),
                            onClick = {
                                changeViewMe = 0
                                changeMapMe = 0
                                val navigate = Intent(
                                    this@HomeActivity, HomeActivity::class.java
                                )
                                startActivity(navigate)
                            }) {
                            if (changeViewMe == 0) {

                                Icon(
                                    imageVector = Icons.Default.Home,
                                    contentDescription = "Home",
                                    tint = Color.Green
                                )

//
                            } else {
                                Icon(
                                    imageVector = Icons.Default.Home,
                                    contentDescription = "Home",
                                    tint = Color.White
                                )
                            }
                        }
                        //create the space for size of 3.Dp
                        Spacer(modifier = Modifier.size(1.dp))

                        // Health Action button
                        IconButton(modifier = Modifier
                            .align(Alignment.CenterVertically)
                            .size(140.dp, 60.dp)
                            .weight(1F), onClick = {
                            changeViewMe = 1
                            changeMapMe = 0
                            val navigate = Intent(
                                this@HomeActivity, HomeActivity::class.java
                            )
                            startActivity(navigate)
                        }) {
                            if (changeViewMe == 1) {
                                Text(
                                    text = "Health",
                                    color = Color.Green,
                                    fontFamily = customfont,
                                    fontSize = 15.sp,
                                    overflow = TextOverflow.Visible,

                                    )
                            } else {
                                Text(
                                    text = "Health",
                                    color = Color.White,
                                    fontFamily = customfont,
                                    fontSize = 15.sp,
                                    overflow = TextOverflow.Visible,

                                    )
                            }
```

```
                        }
                        //create the space for size of 3.Dp
                        Spacer(modifier = Modifier.size(1.dp))

                        // Environmental button
                        IconButton(modifier = Modifier
                            .align(Alignment.CenterVertically)
                            .size(140.dp, 60.dp)
                            .weight(1.5F), onClick = {
                            changeViewMe = 2
                            changeMapMe = 0
                            val navigate = Intent(
                                this@HomeActivity, HomeActivity::class.java
                            )
                            startActivity(navigate)
                        }) {
                            if (changeViewMe == 2) {
                                Text(
                                    text = "Environment",
                                    color = Color.Green,
                                    fontFamily = customfont,
                                    fontSize = 15.sp,
                                    overflow = TextOverflow.Ellipsis
                                )
                            } else {
                                Text(
                                    text = "Environment",
                                    color = Color.White,
                                    fontFamily = customfont,
                                    fontSize = 15.sp,
                                    overflow = TextOverflow.Ellipsis
                                )
                            }
                        }
                        //create the space for size of 3.Dp
                        Spacer(modifier = Modifier.size(1.dp))

                        // Environmental button
                        IconButton(modifier =
Modifier.padding(3.dp).align(Alignment.CenterVertically).size(60.dp, 60.dp).weight(0.5F),
                            onClick = {
                                if (closeTeammate == 1) {
                                    closeMyData = 1
                                    changeMapMe = 0
                                    val navigate = Intent(
                                        this@HomeActivity, MapActivity::class.java
                                    )
                                    startActivity(navigate)
                                } else {
                                    closeMyData = 1
                                    changeMapMe = 0
                                    val navigate = Intent(
                                        this@HomeActivity, HomeActivity::class.java
                                    )
                                    startActivity(navigate)
                                }
                            }

                        ) {
                            Icon(

                                painter = painterResource(R.drawable.close_button),
```

```
                                "Close Button",
                                modifier = Modifier.fillMaxSize(),
                                tint = Color.White
                            )
                        }
                        //create the space for size of 3.Dp
                        Spacer(modifier = Modifier.size(1.dp))
                    }
                })
            }) {

            }
        }
    }
```

Now,the sub snippet inside the above method which call the different Realtime screens and it represents a function that returns a Composable UI screen displaying the health-related data of a user in real-time. The function contains a conditional statement that checks if the changeViewAnotherOne variable is equal to 0. If yes, it returns a Column composable that occupies the maximum available size and has a padding of 3dp.

Within this Column composable, there is another conditional statement that checks if the changeViewMe variable is equal to 0. If yes, it returns a Column composable with a padding of 1dp that occupies the maximum available size and has a weight of 1F.

Inside this Column, there is another conditional statement that checks if the nonExpandedView variable is equal to 0. If yes, it returns two Row composables.

The first Row composable occupies the maximum available width and height and has a weight of 1.3F. It contains a Column composable that occupies the maximum available size and has a weight of 1.3F. Inside this Column, there is a TextButton composable with the text "Warnings" and an onClick listener that sets the nonExpandedView variable to 1 and navigates to the HomeActivity.

The second Row composable occupies the maximum available size, has a weight of 3F, and has a RoundedCornerShape of 10dp. It contains a variable warningSign that stores the warning light intensity for the user. If warningSign is equal to 0, it returns a Text composable with the text "No Warnings for the User". Otherwise, if warningSign is equal to 1, it returns the RealTimeSingleWarningMe function from the SensorsUI class.

The second Column composable has a weight of 1F and occupies the maximum available size with a padding of 1dp. It contains two Row composables.

The first Row composable occupies the maximum available width and height with a weight of 1.2F. It contains a Column composable that occupies the maximum available size and has a weight of 1F. Inside this Column, there is a TextButton composable with the text "Vitals" and an onClick listener that sets the nonExpandedView variable to 2 and navigates to the HomeActivity.

The second Row composable has a weight of 3F, a RoundedCornerShape of 10dp, and occupies the maximum available width and height. It returns the RealTimeVitalSubGraphMe function from the SensorsUI class.

If the nonExpandedView variable is equal to 1, the function returns a Column composable with a padding of 1dp that occupies the maximum available size and has a weight of 1F.

Inside this Column, there are two Row composables.

The first Row composable occupies the maximum available width and height with a weight of 0.7F. It contains a Column composable that occupies the maximum available size and has a weight of 1F. Inside this Column, there is a TextButton composable with the text "Warnings" and an onClick listener that sets the nonExpandedView variable to 0 and navigates to the HomeActivity. Which is as follow:

```
 if (changeViewAnotherOne == 0) {
   Column(
       modifier = Modifier.fillMaxSize().padding(3.dp)
   ) {
       if (changeViewMe == 0) {


           if (nonExpandedView == 0) {
               Column(
                   modifier = Modifier.fillMaxSize().padding(1.dp).weight(1F)
               ) {

                   Row(
                       modifier = Modifier
                           .fillMaxWidth().fillMaxHeight().weight(1.3F)
.background(Color.Transparent),
                       verticalAlignment = Alignment.Top
                   ) {

                       Column(
                           modifier = Modifier
                               .fillMaxSize().padding(2.dp).weight(1.3F),
horizontalAlignment = Alignment.Start
                       ) {
                           TextButton(onClick = {

                               nonExpandedView = 1
                               val navigate = Intent(
```

```
                            this@HomeActivity, HomeActivity::class.java
                        )
                        startActivity(navigate)
                    }) {
                        Text(
                            text = "Warnings",
                            fontWeight = FontWeight.Bold,
                            color = Color.White,
                            fontSize = 13.sp
                        )
                    }
                }

            }

            Row(
                modifier = Modifier
                    .clip(RoundedCornerShape(10.dp))
                    .fillMaxSize()
                    .weight(3F),
                verticalAlignment = Alignment.CenterVertically,
                horizontalArrangement = Arrangement.Center
            ) {

                val warningSign = SensorsUI().GetWarningLightInt(
                    this@HomeActivity, "Me"
                )
                if (warningSign == 0) {
                    Text(
                        text = "  No Warnings for the User",
                        modifier = Modifier
                            .padding(bottom = 2.dp)
                            .align(Alignment.CenterVertically),
                        color = Color.White,
                        fontFamily = FontFamily.SansSerif,
                        fontWeight = FontWeight.SemiBold,
                        fontStyle = FontStyle.Italic,
                        fontSize = 18.sp,

                        )
                } else if (warningSign == 1) {
                    SensorsUI().RealTimeSingleWarningMe(this@HomeActivity)
                }

            }


        }

        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(1.dp)
                .weight(1F)
        ) {

            Row(
                modifier = Modifier
                    .fillMaxWidth()
                    .fillMaxHeight()
                    .weight(1.2F)
                    .background(Color.Transparent),
```

```
                    verticalAlignment = Alignment.Top
                ) {

                    Column(
                        modifier = Modifier
                            .fillMaxSize()
                            .weight(1F),
                        horizontalAlignment = Alignment.Start
                    ) {
                        TextButton(
                            onClick = {
                                nonExpandedView = 2

                                val navigate = Intent(
                                    this@HomeActivity, HomeActivity::class.java
                                )
                                startActivity(navigate)
                            },
                        ) {
                            Text(
                                text = "Vitals",
                                fontWeight = FontWeight.Bold,
                                color = Color.White,
                                fontSize = 13.sp
                            )
                        }
                    }
                }

                Row(
                    modifier = Modifier
                        .clip(RoundedCornerShape(10.dp))
                        .fillMaxWidth()
                        .fillMaxHeight()
                        .weight(3F),
                    verticalAlignment = Alignment.CenterVertically
                ) {

                    SensorsUI().RealTimeVitalSubGraphMe(this@HomeActivity)

                }

            }
        } else if (nonExpandedView == 1) {
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(1.dp)
                    .weight(1F)
            ) {

                Row(
                    modifier = Modifier
                        .fillMaxWidth()
                        .fillMaxHeight()
                        .weight(0.7F)
                        .background(Color.Transparent),
                    verticalAlignment = Alignment.Top
                ) {

                    Column(
                        modifier = Modifier
```

```
                            .fillMaxSize()
                            .padding(2.dp)
                            .weight(1F), horizontalAlignment = Alignment.Start
                    ) {
                        TextButton(onClick = {
                            nonExpandedView = 0

                            val navigate = Intent(
                                this@HomeActivity, HomeActivity::class.java
                            )
                            startActivity(navigate)
                        }) {
                            Text(
                                text = "Warnings",
                                fontWeight = FontWeight.Bold,
                                color = Color.White,
                                fontSize = 13.sp
                            )
                        }
                    }
                }
                Row(
                    modifier = Modifier
                        .clip(RoundedCornerShape(10.dp))
                        .fillMaxSize()
                        .weight(3.3F),
                    horizontalArrangement = Arrangement.Center,
                    verticalAlignment = Alignment.CenterVertically
                ) {

                    val warningSign = SensorsUI().GetWarningLightInt(
                        this@HomeActivity, "Me"
                    )
                    if (warningSign == 0) {
                        Text(
                            text = "  No Warnings for the User",
                            modifier = Modifier
                                .padding(bottom = 2.dp)
                                .align(Alignment.CenterVertically),
                            color = Color.White,
                            fontFamily = FontFamily.SansSerif,
                            fontWeight = FontWeight.SemiBold,
                            fontStyle = FontStyle.Italic,
                            fontSize = 18.sp,


                            )
                    } else if (warningSign == 1) {
                        SensorsUI().RealTimeSingleWarningMe(this@HomeActivity)
                    }
                }
            }
        } else if (nonExpandedView == 2) {
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(1.dp)
                    .weight(1F)
            ) {

                Row(
                    modifier = Modifier
```

```
                        .fillMaxWidth()
                        .fillMaxHeight()
                        .weight(0.7F)
                        .background(Color.Transparent),
                    verticalAlignment = Alignment.Top
                ) {

                    Column(
                        modifier = Modifier
                            .fillMaxSize()
                            .weight(1F),
                        horizontalAlignment = Alignment.Start
                    ) {
                        TextButton(
                            onClick = {
                                nonExpandedView = 0
                                val navigate = Intent(
                                    this@HomeActivity, HomeActivity::class.java
                                )
                                startActivity(navigate)
                            },
                        ) {
                            Text(
                                text = "Vitals",
                                fontWeight = FontWeight.Bold,
                                color = Color.White,
                                fontSize = 13.sp
                            )
                        }
                    }
                }

                Row(
                    modifier = Modifier
                        .clip(RoundedCornerShape(10.dp))
                        .fillMaxWidth()
                        .fillMaxHeight()
                        .weight(3.3F),
                    verticalAlignment = Alignment.CenterVertically
                ) {

                    SensorsUI().RealTimeVitalSubGraphMe(this@HomeActivity)

                }
            }
        }
    } else if (changeViewMe == 1) {

        if (subGraphScreen == 0) {
            Row(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(1.dp)
                    .weight(1F)
            ) {
                SensorsUI().RealTimeHealthTwoPointOMe(this@HomeActivity)
            }
        } else if (subGraphScreen == 1) {
            Row(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(1.dp)
```

```
                        .weight(1F)
                ) {
                    SensorsUI().RealTimeHealthSubGraphMe(this@HomeActivity)
                }
            }
        } else if (changeViewMe == 2) {
            if (subGraphScreen == 0) {
                Row(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(1.dp)
                        .weight(1F)
                ) {
                    SensorsUI().RealTimeEnvTwoPointOMe(this@HomeActivity)
                }
            } else if (subGraphScreen == 1) {
                Row(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(1.dp)
                        .weight(1F)
                ) {
                    SensorsUI().RealTimeEnvSubGraphMe(this@HomeActivity)
                }
            }
        }
    }
}
```

3.1.3 RealTimeMainTeammate

It follows the same procedure as above for the RealTimeMainUser() and generate the same topBar and scaffold with just different teammates data.which is as follow:

```
@OptIn(ExperimentalMaterialApi::class)
@SuppressLint("UnusedMaterialScaffoldPaddingParameter")
@Composable
fun RealTimeTeammate() {
   Row() {
       Scaffold(topBar = {
           TopAppBar(modifier = Modifier
               .fillMaxWidth()
               .fillMaxHeight(.13f)
               .padding(top = 3.dp, start = 3.dp, end = 3.dp, bottom = 3.dp)
               .graphicsLayer {
                   shape = RoundedCornerShape(12.dp)
                   clip = true
               }, elevation = 10.dp, title = {
               Text(
                   text = "$selectedMember",
                   color = Color.White,
                   fontFamily = customfont,
                   modifier = Modifier
                       .padding(2.dp)
                       .align(Alignment.CenterVertically),
                   fontSize = 15.sp,
                   overflow = TextOverflow.Visible,

                   )
           }, actions = {

               //create the row layout to arrange the action button and dropdownMenu
               Row(
                   modifier = Modifier
                       .fillMaxWidth(.85f)
                       .fillMaxHeight()
                       .padding(2.dp),
                   horizontalArrangement = Arrangement.End,
                   verticalAlignment = Alignment.CenterVertically

               ) {

                   // Home Action button
                   IconButton(modifier = Modifier
                       .padding(4.dp)
                       .align(Alignment.CenterVertically)
                       .size(60.dp, 60.dp)
                       .weight(0.5F),
                       onClick = {
                           changeView = 0
                           changeMap = 0
                           val navigate = Intent(
                               this@HomeActivity, HomeActivity::class.java
                           )
                           startActivity(navigate)
                       }) {
                       if (changeView == 0) {
                           Icon(
                               imageVector = Icons.Default.Home,
                               contentDescription = "Home",
                               tint = Color.Green
                           )
                       } else {
                           Icon(
```

```
                imageVector = Icons.Default.Home,
                contentDescription = "Home",
                tint = Color.White
            )
        }
    }
    //create the space for size of 3.Dp
    Spacer(modifier = Modifier.size(1.dp))

    // Health Action button
    IconButton(modifier = Modifier
        .align(Alignment.CenterVertically)
        .size(140.dp, 60.dp)
        .weight(1F), onClick = {
        changeView = 1
        changeMap = 0
        val navigate = Intent(
            this@HomeActivity, HomeActivity::class.java
        )
        startActivity(navigate)
    }) {
        if (changeView == 1) {
            Text(
                text = "Health",
                color = Color.Green,
                fontFamily = customfont,
                fontSize = 15.sp,
                overflow = TextOverflow.Visible,

                )
        } else {
            Text(
                text = "Health",
                color = Color.White,
                fontFamily = customfont,
                fontSize = 15.sp,
                overflow = TextOverflow.Visible,

                )
        }
    }
    //create the space for size of 3.Dp
    Spacer(modifier = Modifier.size(1.dp))

    // Environmental button
    IconButton(modifier = Modifier
        .align(Alignment.CenterVertically)
        .size(140.dp, 60.dp)
        .weight(1.5F), onClick = {
        changeView = 2
        changeMap = 0
        val navigate = Intent(
            this@HomeActivity, HomeActivity::class.java
        )
        startActivity(navigate)
    }

    ) {
        if (changeView == 2) {
            Text(
                text = "Environment",
                color = Color.Green,
```

```
                            fontFamily = customfont,
                            fontSize = 15.sp,
                            overflow = TextOverflow.Ellipsis
                        )
                    } else {
                        Text(
                            text = "Environment",
                            color = Color.White,
                            fontFamily = customfont,
                            fontSize = 15.sp,
                            overflow = TextOverflow.Ellipsis
                        )
                    }
                }
                //create the space for size of 3.Dp
                Spacer(modifier = Modifier.size(1.dp))

                // close button
                IconButton(modifier = Modifier
                    .padding(3.dp)
                    .align(Alignment.CenterVertically)
                    .size(60.dp, 60.dp)
                    .weight(0.5F),
                    onClick = {
                        if (closeMyData == 1) {
                            closeTeammate = 1
                            changeMapMe = 0
                            val navigate = Intent(
                                this@HomeActivity, MapActivity::class.java
                            )
                            startActivity(navigate)
                        } else {
                            closeTeammate = 1
                            changeMapMe = 0
                            val navigate = Intent(
                                this@HomeActivity, HomeActivity::class.java
                            )
                            startActivity(navigate)
                        }
                    }

                ) {
                    Icon(

                        painter = painterResource(R.drawable.close_button),
                        "Close Button",
                        modifier = Modifier.fillMaxSize(),
                        tint = Color.White
                    )
                }
                //create the space for size of 3.Dp
                Spacer(modifier = Modifier.size(1.dp))
            }
        })
    }) {
        Row {
            if (changeViewAnotherOne == 0) {
                Column(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(3.dp)
                ) {
```

```
                if (changeView == 0) {

                    if (nonExpandedView2 == 0) {
                        Column(
                            modifier = Modifier
                                .fillMaxSize()
                                .padding(1.dp)
                                .weight(1F)
                        ) {


                            Row(
                                modifier = Modifier
                                    .fillMaxWidth()
                                    .fillMaxHeight()
                                    .weight(1.3F)
                                    .background(Color.Transparent),
                                verticalAlignment = Alignment.Top
                            ) {

                                Column(
                                    modifier = Modifier
                                        .fillMaxSize()
                                        .weight(1F),
                                    horizontalAlignment = Alignment.Start
                                ) {
                                    TextButton(
                                        onClick = {

                                            nonExpandedView2 = 1
                                            val navigate = Intent(
                                                this@HomeActivity,
                                                HomeActivity::class.java
                                            )
                                            startActivity(navigate)
                                        },
                                    ) {
                                        Text(
                                            text = "Warnings",
                                            fontWeight = FontWeight.Bold,
                                            color = Color.White,
                                            fontSize = 13.sp
                                        )
                                    }
                                }
                            }


                            Row(
                                modifier = Modifier
                                    .clip(RoundedCornerShape(10.dp))
                                    .fillMaxSize()
                                    .weight(3F),
                                verticalAlignment = Alignment.CenterVertically,
                                horizontalArrangement = Arrangement.Center
                            ) {

                                val warningSign = SensorsUI().GetWarningLightInt(
                                    this@HomeActivity, selectedMember
                                )
                                if (warningSign == 0) {
                                    Text(
```

```
                                            text = "  No Warnings for the User",
                                            modifier = Modifier
                                                .padding(bottom = 2.dp)
                                                .align(Alignment.CenterVertically),
                                            color = Color.White,
                                            fontFamily = FontFamily.SansSerif,
                                            fontWeight = FontWeight.Normal,
                                            fontSize = 20.sp
                                        )
                                    } else if (warningSign == 1) {

SensorsUI().RealTimeSingleWarning(this@HomeActivity)
                                    }

                                }


                            }

                            Column(
                                modifier = Modifier
                                    .fillMaxSize()
                                    .padding(1.dp)
                                    .weight(1F)
                            ) {

                                Row(
                                    modifier = Modifier
                                        .fillMaxWidth()
                                        .fillMaxHeight(0.10f)
                                        .weight(1.2F)
                                        .background(Color.Transparent),
                                    verticalAlignment = Alignment.Top
                                ) {

                                    Column(
                                        modifier = Modifier
                                            .fillMaxSize()
                                            .weight(1F),
                                        horizontalAlignment = Alignment.Start
                                    ) {
                                        TextButton(
                                            onClick = {
                                                nonExpandedView2 = 2
                                                val navigate = Intent(
                                                    this@HomeActivity,
                                                    HomeActivity::class.java
                                                )
                                                startActivity(navigate)
                                            },
                                        ) {
                                            Text(
                                                text = "Vitals",
                                                fontWeight = FontWeight.Bold,
                                                color = Color.White,
                                                fontSize = 13.sp
                                            )
                                        }
                                    }
                                }
```

```
                    Row(
                        modifier = Modifier
                            .clip(RoundedCornerShape(10.dp))
                            .fillMaxSize()
                            .weight(3F),
                        verticalAlignment = Alignment.CenterVertically
                    ) {

                        SensorsUI().RealTimeVitalSubGraph(this@HomeActivity)

                    }


                }
            } else if (nonExpandedView2 == 1) {
                Column(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(1.dp)
                        .weight(1F)
                ) {


                    Row(
                        modifier = Modifier
                            .fillMaxWidth()
                            .fillMaxHeight()
                            .weight(0.7F)
                            .background(Color.Transparent),
                        verticalAlignment = Alignment.Top
                    ) {

                        Column(
                            modifier = Modifier
                                .fillMaxSize()
                                .weight(1F),
                            horizontalAlignment = Alignment.Start
                        ) {
                            TextButton(
                                onClick = {

                                    nonExpandedView2 = 0
                                    val navigate = Intent(
                                        this@HomeActivity,
                                        HomeActivity::class.java
                                    )
                                    startActivity(navigate)
                                },
                            ) {
                                Text(
                                    text = "Warnings",
                                    fontWeight = FontWeight.Bold,
                                    color = Color.White,
                                    fontSize = 13.sp
                                )
                            }
                        }
                    }


                    Row(
                        modifier = Modifier
```

```
                                        .clip(RoundedCornerShape(10.dp))
                                        .fillMaxSize()
                                        .weight(3.3F),
                                    verticalAlignment = Alignment.CenterVertically
                                ) {

                                    val warningSign = SensorsUI().GetWarningLightInt(
                                        this@HomeActivity, selectedMember
                                    )
                                    if (warningSign == 0) {
                                        Text(
                                            text = "  No Warnings for the User",
                                            modifier = Modifier
                                                .padding(bottom = 2.dp)
                                                .align(Alignment.CenterVertically),
                                            color = Color.White,
                                            fontFamily = FontFamily.SansSerif,
                                            fontWeight = FontWeight.Normal,
                                            fontSize = 20.sp
                                        )
                                    } else if (warningSign == 1) {

SensorsUI().RealTimeSingleWarning(this@HomeActivity)
                                    }

                                }


                            }
                        } else if (nonExpandedView2 == 2) {
                            Column(
                                modifier = Modifier
                                    .fillMaxSize()
                                    .padding(1.dp)
                                    .weight(1F)
                            ) {

                                Row(
                                    modifier = Modifier
                                        .fillMaxWidth()
                                        .fillMaxHeight(0.10f)
                                        .weight(0.7F)
                                        .background(Color.Transparent),
                                    verticalAlignment = Alignment.Top
                                ) {

                                    Column(
                                        modifier = Modifier
                                            .fillMaxSize()
                                            .weight(1F),
                                        horizontalAlignment = Alignment.Start
                                    ) {
                                        TextButton(
                                            onClick = {
                                                nonExpandedView2 = 0
                                                val navigate = Intent(
                                                    this@HomeActivity,
                                                    HomeActivity::class.java
                                                )
                                                startActivity(navigate)
                                            },
                                        ) {
```

```
                                            Text(
                                                text = "Vitals",
                                                fontWeight = FontWeight.Bold,
                                                color = Color.White,
                                                fontSize = 13.sp
                                            )
                                        }
                                    }
                                }


                                Row(
                                    modifier = Modifier
                                        .clip(RoundedCornerShape(10.dp))
                                        .fillMaxSize()
                                        .weight(3.3F),
                                    verticalAlignment = Alignment.CenterVertically
                                ) {

                                    SensorsUI().RealTimeVitalSubGraph(this@HomeActivity)

                                }


                            }
                        }


                    } else if (changeView == 1) {

                        if (subGraphScreen2 == 0) {
                            Row(
                                modifier = Modifier
                                    .fillMaxSize()
                                    .padding(1.dp)
                                    .weight(1F)
                            ) {
                                SensorsUI().RealTimeHealthTwoPointO(this@HomeActivity)
                            }
                        } else if (subGraphScreen2 == 1) {
                            Row(
                                modifier = Modifier
                                    .fillMaxSize()
                                    .padding(1.dp)
                                    .weight(1F)
                            ) {

SensorsUI().RealTimeHealthSubGraphTeammate(this@HomeActivity)
                            }
                        }


                    } else if (changeView == 2) {

                        if (subGraphScreen2 == 0) {
                            Row(
                                modifier = Modifier
                                    .fillMaxSize()
                                    .padding(1.dp)
                                    .weight(1F)
                            ) {
                                SensorsUI().RealTimeEnvTwoPointO(this@HomeActivity)
```

```
                                }
                            } else if (subGraphScreen2 == 1) {
                                Row(
                                    modifier = Modifier
                                        .fillMaxSize()
                                        .padding(1.dp)
                                        .weight(1F)
                                ) {

SensorsUI().RealTimeEnvSubGraphTeammate(this@HomeActivity)
                                }
                            }


                        }
                    }
                }
            }
        }
    }
}
```

3.1.4 Real-Time Screen Components

This section give details about the components used for the above three method to develop the Real-Time Screen which are developed in RealTimeSensorUI.kt file which are as follow:

1. Function Name: CardMakerTwoPointO

   This function is used to create a card component in Jetpack Compose with a custom design and functionality. It takes four parameters as input and returns a Composable Card object.

   Parameters

   - sensorName: A string representing the name of the sensor to be displayed on the card.
   - sensorData: A string representing the sensor data to be displayed on the card.
   - warningSign: An integer value representing the warning sign status. If the value is 0, the sensor data will be displayed in white color, and if the value is 1, the sensor data will be displayed in red color.
   - context: A Context object representing the application context.

   Return

   - A Composable Card object with a custom design and functionality.

   Annotations

   - @OptIn(ExperimentalMaterialApi::class): This annotation is used to enable experimental Material Design APIs.

   Composable Annotation

   - @Composable: This annotation is used to indicate that this function is a composable function and can be used to create a UI component.

Functionality

The CardMakerTwoPointO function creates a card component with the following design and functionality:

- The card has a width of 120.dp and a height of 55.dp.
- The card has a rounded corner shape with a radius of 15.dp.
- The card background color is based on the Material Theme background color.
- The card elevation is 2.dp.
- The card contains three rows with the sensor name, sensor data, and data unit.
- The sensor name is displayed in a shortened form with a font size of 9.sp, a bold font weight, and white color.
- The sensor data is split into two parts: the data and the unit. The data is displayed in a font size of 15.sp, a bold font weight, and either white or red color based on the warning sign value.
- The data unit is displayed in a font size of 9.sp, a bold font weight, and white color.
- The card is clickable, and when clicked, it sets the selectedSensor, selectedSensorTwoPoint0, and selectedMemberTwoPoint0 variables and navigates to the HistoricalUI activity.

Code

```
@OptIn(ExperimentalMaterialApi::class)
@Composable
fun CardMakerTwoPointO(
   sensorName: String, sensorData: String, warningSign: Int, context: Context
) {
   Card(modifier = Modifier
        .width(120.dp)
        .height(55.dp)
        .testTag(stringResource(id = R.string.card1))
        .padding(3.dp),
        backgroundColor = MaterialTheme.colors.background,
        elevation = 2.dp,
        shape = RoundedCornerShape(15.dp),
        onClick = {
            selectedSensor = sensorName
            selectedSensorTwoPoint0 = selectedSensor
            selectedMemberTwoPoint0 = selectedMember
            //Navigate to the HistoricalUI activity on click of card
            val navigate = Intent(context, HistoricalUI::class.java)
            context.startActivity(navigate)
        }) {

        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(2.dp),
            verticalArrangement = Arrangement.Center

        ) {
            //Row to display the sensor name
            Row(
```

```
                modifier = Modifier
                    .padding(2.dp)
                    .weight(1F)
                    .fillMaxWidth()
                    .fillMaxHeight(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Top
            ) {
                //Shorten the sensor name and display it
                val tempSensorName = shortSensorNames[sensorNames.indexOf(sensorName)]
                Text(
                    text = tempSensorName,
                    fontFamily = customfont,
                    fontWeight = FontWeight.Bold,
                    fontSize = 9.sp,
                    letterSpacing = 1.sp,
                    color = Color.White,
                    overflow = TextOverflow.Ellipsis
                )
            }
            //Get the sensor data and split it into data and unit
            var wholeDate = sensorData
            var delDate = " "
            var justData = " "
            var dataUnit = " "

            try {
                wholeDate = sensorData
                delDate = " "
                val splitDate = wholeDate.split(delDate)
                justData = splitDate[0]
                dataUnit = splitDate[1]
            } catch (e: Exception) {

            }
            //Row to display the sensor data
            Row(
                modifier = Modifier
                    .padding(2.dp)
                    .weight(1.5F)
                    .fillMaxWidth()
                    .fillMaxHeight(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Top

            ) {
                //Display the sensor data in red color if the warning sign is 1, else
in white color
                if (warningSign == 0) {
                    Text(
                        text = "$justData",
                        modifier = Modifier.align(Alignment.CenterVertically),
                        fontFamily = customfont,
                        fontWeight = FontWeight.Bold,
                        fontSize = 15.sp,
                        letterSpacing = 1.sp,
                        color = Color.White,
                        overflow = TextOverflow.Ellipsis

                    )
                } else if (warningSign == 1) {
                    Text(
```

```
                        text = "$justData",
                        modifier = Modifier.align(Alignment.CenterVertically),
                        fontFamily = customfont,
                        fontWeight = FontWeight.Bold,
                        fontSize = 15.sp,
                        letterSpacing = 1.sp,
                        color = Color.Red,
                        overflow = TextOverflow.Ellipsis

                    )

                }
            }
            //Row to display the sensor data unit
            Row(
                modifier = Modifier
                    .padding(2.dp)
                    .weight(1F)
                    .fillMaxWidth()
                    .fillMaxHeight(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Top

            ) {
                Text(
                    text = "$dataUnit",
                    fontFamily = customfont,
                    fontWeight = FontWeight.Bold,
                    fontSize = 9.sp,
                    letterSpacing = 1.sp,
                    color = Color.White,
                    overflow = TextOverflow.Ellipsis

                )
            }}
    }}
```

2. Function Name: CardMakerTwoPointOMe
   The CardMakerTwoPointOMe function is a composable function that generates a Material Design Card widget with the given parameters. It takes in a sensorName string, sensorData string, warningSign integer, and context object as input parameters. It uses the ExperimentalMaterialApi annotation to indicate that it is using experimental Material Design components.

   The function creates a Material Design Card widget that displays sensor data, such as temperature or humidity, along with a sensor name and unit of measurement. It also includes a warning sign that changes color depending on the value of the warningSign parameter. The user can click on the card to navigate to a historical data view for the selected sensor.
   Parameters
   - sensorName: A string representing the name of the sensor whose data is being displayed.
   - sensorData: A string representing the data of the sensor being displayed, along with its unit of measurement.

- warningSign: An integer that determines the color of the warning sign. A value of 0 indicates no warning, while a value of 1 indicates a warning.
- context: An Android context object used to start the historical data view activity.

Composable Components

The CardMakerTwoPointOMe function uses the following composable components to create the Material Design Card widget:

- Card: A Material Design Card widget that displays sensor data.
- Column: A composable component that arranges its children vertically.
- Row: A composable component that arranges its children horizontally.
- Text: A composable component that displays text.
- Spacer: A composable component that creates empty space between components.
- Modifier: A utility class used to modify the layout and behavior of composable components.
- MaterialTheme: A composable component used to define the colors and typography of Material Design components.

Code

```
class MainActivity : ComponentActivity()
@OptIn(ExperimentalMaterialApi::class)
@Composable
fun CardMakerTwoPointOMe(
   sensorName: String, sensorData: String, warningSign: Int, context: Context
) {

   Card(modifier = Modifier
       .width(120.dp)
       .testTag(stringResource(id = R.string.card2))
       .height(55.dp)
       .padding(3.dp),
       backgroundColor = MaterialTheme.colors.background,
       elevation = 5.dp,
       shape = RoundedCornerShape(15.dp),
       onClick = {
           selectedSensor = sensorName
           selectedMemberTwoPoint0 = "Me"
           selectedSensorTwoPoint0 = selectedSensor
           val navigate = Intent(context, HistoricalUI::class.java)
           context.startActivity(navigate)
       }) {

       Column(
           modifier = Modifier
               .fillMaxSize()
               .padding(2.dp),
           verticalArrangement = Arrangement.Center

       ) {


           Row(
               modifier = Modifier
```

```
                    .padding(2.dp)
                    .weight(1F)
                    .fillMaxWidth()
                    .fillMaxHeight(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Top
            ) {
                val tempSensorName =
shortSensorNames[sensorNames.indexOf(sensorName)]
                Text(
                    text = tempSensorName,
                    modifier = Modifier.align(Alignment.CenterVertically),
                    fontFamily = customfont,
                    fontWeight = FontWeight.Bold,
                    fontSize = 9.sp,
                    letterSpacing = 1.sp,
                    color = Color.White,
                    overflow = TextOverflow.Ellipsis
                )

            }

            var wholeDate = sensorData
            var delDate = " "
            var justData = " "
            var dataUnit = " "

            try {
                wholeDate = sensorData
                delDate = " "
                val splitDate = wholeDate.split(delDate)
                justData = splitDate[0]
                dataUnit = splitDate[1]
            } catch (e: Exception) {

            }
            Row(
                modifier = Modifier
                    .padding(2.dp)
                    .weight(1.5F)
                    .fillMaxWidth()
                    .fillMaxHeight(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Top
            ) {

                if (warningSign == 0) {
                    Text(
                        text = "$justData",
                        modifier = Modifier,
                        fontFamily = customfont,
                        fontWeight = FontWeight.Bold,
                        fontSize = 15.sp,
                        letterSpacing = 1.sp,
                        color = Color.White,
                        overflow = TextOverflow.Ellipsis

                    )
                } else if (warningSign == 1) {
                    Text(
                        text = "$justData",
                        modifier = Modifier,
```

```
                        fontFamily = customfont,
                        fontWeight = FontWeight.Bold,
                        fontSize = 15.sp,
                        letterSpacing = 1.sp,
                        color = Color.Red,
                        overflow = TextOverflow.Ellipsis

                    )

                }
            }
            Spacer(modifier = Modifier.size(1.dp))
            Row(
                modifier = Modifier
                    .padding(2.dp)
                    .weight(1F)
                    .fillMaxWidth()
                    .fillMaxHeight(),
                horizontalArrangement = Arrangement.Center,
                verticalAlignment = Alignment.Top
            ) {
                Text(
                    text = "$dataUnit",
                    modifier = Modifier,
                    fontFamily = customfont,
                    fontWeight = FontWeight.Bold,
                    fontSize = 10.sp,
                    letterSpacing = 1.sp,
                    color = Color.White,
                    overflow = TextOverflow.Ellipsis

                )
            }
        }

    }
}
```

Then, you can call the CardMakerTwoPointOMe function and pass in the required parameters: ***CardMakerTwoPointOMe("Temperature", "22 °C", 0, context)***

This will create a Material Design Card widget that displays the name "Temperature", the value "22", and the unit of measurement "°C", with no warning sign. Clicking on the card will navigate to the historical data view for the selected sensor.

3. Function Name: CardMakerRealTimeScreen2TwoPointO
   Function Signature
   @OptIn(ExperimentalMaterialApi::class)
   @Composable
   fun CardMakerRealTimeScreen2TwoPointO(sensorName: String, sensorData: String, warningSign: Int, context: Context)
   Description

This is an experimental composable function that displays a card UI for real-time sensor data. The function takes four parameters, sensorName, sensorData, warningSign, and context. The sensorName parameter is a string that represents the name of the sensor. The sensorData parameter is a string that represents the current sensor data. The warningSign parameter is an integer that represents the warning sign for the sensor data. The context parameter is the context of the current state of the application.

Parameters

- sensorName : A string that represents the name of the sensor.
- sensorData : A string that represents the current sensor data.
- warningSign : An integer that represents the warning sign for the sensor data.
- context : The context of the current state of the application.

Composable Functions used

- Card: A pre-defined composable function provided by Jetpack Compose that displays a Material Design Card.
- Row: A pre-defined composable function provided by Jetpack Compose that arranges its children in a horizontal row.
- Column: A pre-defined composable function provided by Jetpack Compose that arranges its children in a vertical column.
- Text: A pre-defined composable function provided by Jetpack Compose that displays text.

Return Value

This function doesn't return anything. It is a composable function and its return type is Unit.

Code

```
@OptIn(ExperimentalMaterialApi::class)
@Composable
fun CardMakerRealTimeScreen2TwoPointO(
   sensorName: String, sensorData: String, warningSign: Int, context: Context
) {

   Card(
       modifier = Modifier
           .fillMaxWidth()
           .height(58.dp)
           .padding(2.dp)
           .border(BorderStroke(0.8.dp, Color.White), shape =
RoundedCornerShape(10.dp)),
       backgroundColor = MaterialTheme.colors.background,
       elevation = 1.dp,
       shape = RoundedCornerShape(10.dp),
       onClick = {
           selectedSensor = sensorName
           selectedSensorTwoPoint0 = selectedSensor
           selectedMemberTwoPoint0 = selectedMember
           val navigate = Intent(context, HistoricalUI::class.java)
           context.startActivity(navigate)
       },
```

```
        content = {
            Row(
                modifier = Modifier.fillMaxWidth(),

                )
            {

                Column(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(3.dp)
                        .clip(RoundedCornerShape(10.dp))
                        .weight(1F)
                        .background(MaterialTheme.colors.background),
                    verticalArrangement = Arrangement.Center
                )
                {
                    Column(
                        modifier = Modifier
                            .align(Alignment.CenterHorizontally)
                            .padding(3.dp)
                    ) {
                        val tempSensorName =
shortSensorNames[sensorNames.indexOf(sensorName)]
                        Text(
                            text = tempSensorName,
                            modifier = Modifier.align(Alignment.CenterHorizontally),
                            fontFamily = customfont,
                            fontWeight = FontWeight.Bold,
                            fontSize = 13.sp,
                            letterSpacing = 1.sp,
                            color = Color.White,
                            overflow = TextOverflow.Ellipsis
                        )
                        if (warningSign == 0) {
                            Text(
                                text = sensorData,
                                modifier =
Modifier.align(Alignment.CenterHorizontally),
                                fontFamily = customfont,
                                fontWeight = FontWeight.Bold,
                                fontSize = 13.sp,
                                letterSpacing = 1.sp,
                                color = Color.White,
                                overflow = TextOverflow.Ellipsis
                            )
                        } else if (warningSign == 1) {
                            Text(
                                text = sensorData,
                                modifier =
Modifier.align(Alignment.CenterHorizontally),
                                fontFamily = customfont,
                                fontWeight = FontWeight.Bold,
                                fontSize = 13.sp,
                                letterSpacing = 1.sp,
                                color = Color.Red,
                                overflow = TextOverflow.Ellipsis
                            )
                        }
                    }
                }
```

```
                Column(
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(1.dp)
                        .clip(RoundedCornerShape(10.dp))
                        .weight(2.5F),
                    verticalArrangement = Arrangement.Center
                )
                {
                    // call 10 min graph
                    tenMinGraphSensorArray(sensorName, selectedMember, context)
                }
            }
        }
    )
}
```

4. Function Name: CardMakerRealTimeScreen2TwoPointOMe
   This function is similar to above function just using the user data to be displayed

5. Function Name: RealTimeEnvTwoPointO
   The RealTimeEnvTwoPointO function is a composable function in Kotlin that takes a Context object as a parameter. This function creates a column layout with modifications and displays sensor data in a grid format using CardMakerTwoPointO.
   Parameters
   - context - The Context object is used to access resources related to the application.

   Functionality
   - Column - Creates a composable column layout with various modifications such as size, padding, rounded corners, and test tags.
   - tempEnvironmentSensorsArray - An array list of Sensors objects used to store the sensors without warnings.
   - tempWarningEnvironmentSensorsArray - An array list of Sensors objects used to store the sensors with warnings.
   - environmentSensorsArray - An array list of Sensors objects used to retrieve data for display in the column.
   - getWarningFromDatabase - A function of Sensors object used to get the warning status of the sensor from the database.
   - getIDFromName - A function of Teammates object used to get the ID of the selected team member.
   - LazyHorizontalGrid - A composable that displays the cards in a grid format.
   - items - A function used to display cards for sensors with warnings first and sensors without warnings last.
   - CardMakerTwoPointO - A function used to create the cards with sensor data.

   Code

```
@Composable
fun RealTimeEnvTwoPointO(context: Context) {
   // Create a column layout with various modifications
   Column(
       modifier = Modifier
           .fillMaxSize()
           .padding(7.dp)
           .clip(RoundedCornerShape(12.dp))
           .testTag(stringResource(id = R.string.RTEnv))
           .background(Color.Transparent)
   ) {
       // Retrieve data to be displayed in cards in the column
       val tempEnvironmentSensorsArray: ArrayList<Sensors> =
arrayListOf<Sensors>()
       val tempWarningEnvironmentSensorsArray: ArrayList<Sensors> =
arrayListOf<Sensors>()
       for (index in environmentSensorsArray) {
           val id = Teammates().getIDFromName(selectedMember)
           val tempWarning = index.getWarningFromDatabase(context, id)
           if (tempWarning == 1) {
               tempWarningEnvironmentSensorsArray.add(index)
           } else {
               tempEnvironmentSensorsArray.add(index)
           }
       }
       // Use a LazyHorizontalGrid to display the cards in a grid format
       LazyHorizontalGrid(GridCells.Fixed(3)) {
           // Display cards for sensors with warnings first
           items(tempWarningEnvironmentSensorsArray.size) { index ->
               val id = Teammates().getIDFromName(selectedMember)
               CardMakerTwoPointO(
                   sensorName =
tempWarningEnvironmentSensorsArray[index].getName(),
                   sensorData =
tempWarningEnvironmentSensorsArray[index].getFromDatabase(
                       context, id
                   ),
                   warningSign =
tempWarningEnvironmentSensorsArray[index].getWarningFromDatabase(
                       context, id
                   ),
                   context = context
               )
           }
           // Display cards for sensors without warnings last
           items(tempEnvironmentSensorsArray.size) { index ->
               val id = Teammates().getIDFromName(selectedMember)
               CardMakerTwoPointO(
                   sensorName = tempEnvironmentSensorsArray[index].getName(),
                   sensorData =
tempEnvironmentSensorsArray[index].getFromDatabase(
                       context, id
                   ),
                   warningSign =
tempEnvironmentSensorsArray[index].getWarningFromDatabase(
                       context, id
                   ),
                   context = context
               )
           }
```

```
        }
    }

}
```

6. Function Name: RealTimeHealthTwoPointO
   The RealTimeHealthTwoPointO function is a Composable function that takes a Context object as a parameter. It displays real-time health data in a grid format. It fetches the respective sensor data to be displayed and separates them into two arrays - tempHealthSensorsArray and tempWarningHealthSensorsArray based on whether they have any warnings. The function then displays the data in a grid format using the LazyHorizontalGrid composable.
   Parameters
   - context: Context - The context of the application.

   Composables Used
   - Column - A vertical column that places its children in a top to bottom sequence.
   - LazyHorizontalGrid - A horizontally scrolling grid with fixed number of columns and dynamic number of rows.
   - CardMakerTwoPointO - A Composable function that displays the data in a card format.

   Return Value
   This function does not return any value. It simply displays the real-time health data in a grid format.

   Code

```
 @Composable
   fun RealTimeHealthTwoPointO(context: Context) {

       Column(
           //modified column
           modifier = Modifier
               .fillMaxSize()
               .testTag(stringResource(id = R.string.RTHealth))
               .padding(7.dp)
//
               .clip(RoundedCornerShape(12.dp))
               .background(MaterialTheme.colors.background)


       ) {
           //get the respective sensor data to be displayed
           val tempHealthSensorsArray: ArrayList<Sensors> = arrayListOf<Sensors>()
           val tempWarningHealthSensorsArray: ArrayList<Sensors> =
arrayListOf<Sensors>()
           for (index in healthSensorsArray) {
               val id = Teammates().getIDFromName(selectedMember)
```

```
                val tempWarning = index.getWarningFromDatabase(context, id)
                if (tempWarning == 1) {
                    tempWarningHealthSensorsArray.add(index)
                } else {
                    tempHealthSensorsArray.add(index)
                }
            }
            //grid format to be displayed
            LazyHorizontalGrid(GridCells.Fixed(3)) {
                items(tempWarningHealthSensorsArray.size) { index ->
                    val id = Teammates().getIDFromName(selectedMember)
                    CardMakerTwoPointO(
                        sensorName = tempWarningHealthSensorsArray[index].getName(),
                        sensorData =
tempWarningHealthSensorsArray[index].getFromDatabase(
                            context, id
                        ),
                        warningSign =
tempWarningHealthSensorsArray[index].getWarningFromDatabase(
                            context, id
                        ),
                        context = context
                    )
                }
                items(tempHealthSensorsArray.size) { index ->
                    val id = Teammates().getIDFromName(selectedMember)
                    CardMakerTwoPointO(
                        sensorName = tempHealthSensorsArray[index].getName(),
                        sensorData =
tempHealthSensorsArray[index].getFromDatabase(context, id),
                        warningSign =
tempHealthSensorsArray[index].getWarningFromDatabase(
                            context, id
                        ),
                        context = context
                    )
                }
            }
        }
    }
```

7. Function Name: RealTimeHealthTwoPointOMe
   This function is similar to above ***RealTimeHealthTwoPointO*** function just using the user data to be displayed
8. Function Name: RealTimeEnvTwoPointOMe
   This function is similar to above ***RealTimeEnvTwoPointO*** function just using the user data to be displayed

9. Function Name: RealTimeEnvSubGraphTeammate
   The function RealTimeEnvSubGraphTeammate is a Kotlin Composable function that creates a column layout with various modifications to display environmental data for a selected teammate in a grid format using a LazyColumn.

Input

- context: an instance of the Context class that provides access to application-specific resources and classes.

Output

This function returns nothing. It only composes the UI elements for displaying the environmental data for a selected teammate.

Implementation

- The function starts by creating a Column layout with various modifications such as padding, clipping, test tag, and background color.
- Next, it retrieves the environmental data to be displayed in cards in the column by iterating through the environmentSensorsArray. It populates two arrays, tempEnvironmentSensorsArray and tempWarningEnvironmentSensorsArray, with sensors that have no warnings and sensors that have warnings, respectively.
- It then uses a LazyColumn to display the cards in a grid format.
- It first displays the cards for sensors with warnings, followed by cards for sensors without warnings, using the items function of LazyColumn. Each card is created using the CardMakerRealTimeScreen2TwoPointO function, passing the required sensor data as parameters.

Code

```
@Composable
fun RealTimeEnvSubGraphTeammate(context: Context) {
   // Create a column layout with various modifications
   Column(
       modifier = Modifier
           .fillMaxSize()
           .padding(7.dp)
           .clip(RoundedCornerShape(12.dp))
           .testTag(stringResource(id = R.string.RTEnv))
           .background(Color.Transparent)
   ) {
       // Retrieve data to be displayed in cards in the column
       val tempEnvironmentSensorsArray: ArrayList<Sensors> = arrayListOf<Sensors>()
       val tempWarningEnvironmentSensorsArray: ArrayList<Sensors> =
arrayListOf<Sensors>()
       for (index in environmentSensorsArray) {
           val id = Teammates().getIDFromName(selectedMember)
           val tempWarning = index.getWarningFromDatabase(context, id)
           if (tempWarning == 1) {
               tempWarningEnvironmentSensorsArray.add(index)
           } else {
               tempEnvironmentSensorsArray.add(index)
           }
       }
       // Use a LazyHorizontalGrid to display the cards in a grid format
       LazyColumn() {
           // Display cards for sensors with warnings first
           items(tempWarningEnvironmentSensorsArray.size) { index ->
               val id = Teammates().getIDFromName(selectedMember)
               CardMakerRealTimeScreen2TwoPointO(
                   sensorName = tempWarningEnvironmentSensorsArray[index].getName(),
```

```
                    sensorData =
tempWarningEnvironmentSensorsArray[index].getFromDatabase(
                        context, id
                    ),
                    warningSign =
tempWarningEnvironmentSensorsArray[index].getWarningFromDatabase(
                        context, id
                    ),
                    context = context
                )
            }
            // Display cards for sensors without warnings last
            items(tempEnvironmentSensorsArray.size) { index ->
                val id = Teammates().getIDFromName(selectedMember)
                CardMakerRealTimeScreen2TwoPointO(
                    sensorName = tempEnvironmentSensorsArray[index].getName(),
                    sensorData = tempEnvironmentSensorsArray[index].getFromDatabase(
                        context, id
                    ),
                    warningSign =
tempEnvironmentSensorsArray[index].getWarningFromDatabase(
                        context, id
                    ),
                    context = context
                )
            }
        }
    }

}
```

10. Function Name: RealTimeHealthSubGraphTeammate

RealTimeHealthSubGraphTeammate() is a Kotlin function written using Jetpack Compose library. The function takes a Context object as a parameter.

Parameters

- context : Context - The context of the caller.

Returns

This function does not return any value. It only composes and renders the UI components.

Functionality

This function is responsible for rendering the real-time health subgraph of a teammate. It creates a Column Composable that contains a LazyColumn. It then retrieves the sensor data of the selected teammate from the database and separates them into two arrays, tempWarningHealthSensorsArray and tempHealthSensorsArray, based on whether they have a warning or not. It then creates UI components using the CardMakerRealTimeScreen2TwoPointO Composable for each sensor in the arrays.

Code

```
@Composable
   fun RealTimeHealthSubGraphTeammate(context: Context) {

       Column(
           //modified column
           modifier = Modifier
               .fillMaxSize()
               .testTag(stringResource(id = R.string.RTHealth))
               .padding(7.dp)
//
               .clip(RoundedCornerShape(12.dp))
               .background(MaterialTheme.colors.background)


       ) {
           //get the respective sensor data to be displayed
           val tempHealthSensorsArray: ArrayList<Sensors> = arrayListOf<Sensors>()
           val tempWarningHealthSensorsArray: ArrayList<Sensors> =
arrayListOf<Sensors>()
           for (index in healthSensorsArray) {
               val id = Teammates().getIDFromName(selectedMember)
               val tempWarning = index.getWarningFromDatabase(context, id)
               if (tempWarning == 1) {
                   tempWarningHealthSensorsArray.add(index)
               } else {
                   tempHealthSensorsArray.add(index)
               }
           }
           //grid format to be displayed
           LazyColumn() {
               items(tempWarningHealthSensorsArray.size) { index ->
                   val id = Teammates().getIDFromName(selectedMember)
                   CardMakerRealTimeScreen2TwoPointO(
                       sensorName = tempWarningHealthSensorsArray[index].getName(),
                       sensorData =
tempWarningHealthSensorsArray[index].getFromDatabase(
                           context, id
                       ),
                       warningSign =
tempWarningHealthSensorsArray[index].getWarningFromDatabase(
                           context, id
                       ),
                       context = context
                   )
               }
               items(tempHealthSensorsArray.size) { index ->
                   val id = Teammates().getIDFromName(selectedMember)
                   CardMakerRealTimeScreen2TwoPointO(
                       sensorName = tempHealthSensorsArray[index].getName(),
                       sensorData =
tempHealthSensorsArray[index].getFromDatabase(context, id),
                       warningSign =
tempHealthSensorsArray[index].getWarningFromDatabase(
                           context, id
                       ),
                       context = context
                   )
               }
           }
       }
```

```
    }
```

11. Function Name: RealTimeEnvSubGraphMe
    This function is similar to above ***RealTimeEnvSubGraphTeammate*** function just using the user data to be displayed

12. Function Name: RealTimeHealthSubGraphMe
    This function is similar to above ***RealTimeHealthSubGraphTeammate*** function just using the user data to be displayed

13. Function Name: RealTimeVitalSubGraph
    Description
    This function is responsible for displaying real-time vital subgraph data in the form of a column. It retrieves data for each vital sensor and displays it in a Card using the CardMakerRealTimeScreen2TwoPointO() function.
    Input Parameters
    - context: An instance of the Android Context class that is used to navigate to other activities.

    Output
    This function has no return value, it simply displays the real-time vital subgraph data.

    Code

```
@Composable
fun RealTimeVitalSubGraph(context: Context) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(3.dp)
            .testTag(stringResource(id = R.string.RTVitalSub))
            .clip(RoundedCornerShape(12.dp))
            .background(MaterialTheme.colors.background)
    ) {
        LazyColumn() {
            // Loop through each vital sensor and display its data as a card using
CardMakerRealTimeScreen2TwoPointO().
            items(vitalSensorsArray.size) { index ->
                val id = Teammates().getIDFromName(selectedMember)
                selectedSensor = vitalSensorsArray[index].getName()
                CardMakerRealTimeScreen2TwoPointO(
                    sensorName = vitalSensorsArray[index].getName(),
                    sensorData = vitalSensorsArray[index].getFromDatabase(context, id),
                    warningSign =
vitalSensorsArray[index].getWarningFromDatabase(context, id),
                    context = context
```

```
                )
            }
        }
    }
}
```

14. Function Name: RealTimeVitalSubGraphMe
    This function is similar to above ***RealTimeVitalSubGraphMe*** function just using the user data to be displayed

15. Function Name: RealTimeSingleWarning
    The RealTimeSingleWarning function is a composable function that displays a column of cards containing information about sensors with warning signs for a selected team member in real-time.
    Parameters
    - context: Context - The context of the activity or fragment in which this function is called.

    Functionality
    The RealTimeSingleWarning function first creates a Column composable element that contains the cards. It then uses a LazyHorizontalGrid to display the cards in a horizontal scroll view.
    For each sensor, the function checks if it has a warning sign for the selected team member, and if so, adds it to an ArrayList of Sensors. The function then displays a card for each sensor in the ArrayList, using the CardMakerTwoPointO function.

    Code

```
@Composable
fun RealTimeSingleWarning(context: Context) {
   Column(
       modifier = Modifier
           .fillMaxSize()
           .testTag(stringResource(id = R.string.RTWarning))
           .padding(7.dp)
           .clip(RoundedCornerShape(12.dp))
           .background(MaterialTheme.colors.background)
   ) {
       // Use LazyHorizontalGrid to display cards in a horizontal scroll view
       LazyHorizontalGrid(GridCells.Fixed(1)) {
           val id = Teammates().getIDFromName(selectedMember)
           val tempSensor: ArrayList<Sensors> = arrayListOf<Sensors>()

           // Iterate through all sensors to find those with a warning sign
           for (k in allSensorsArray) {
               val tempWarning = k.getWarningFromDatabase(context, id)
               val tempSensorName = k.getName()
               if (tempWarning == 1) {
                   if (!vitalSensorNames.contains(tempSensorName)) {
```

```
                        tempSensor.add(k)
                    }
                }
            }

            // Display cards for each non-vital sensor with a warning sign
            items(tempSensor.size) { index ->
                val tempData = tempSensor[index].getFromDatabase(context, id)
                val tempWarning = tempSensor[index].getWarningFromDatabase(context, id)
                CardMakerTwoPointO(
                    sensorName = tempSensor[index].getName(),
                    sensorData = tempData,
                    warningSign = tempWarning,
                    context = context
                )
            }
        }
    }
}
```

16. Function Name: RealTimeSingleWarningMe
    This function is similar to above ***RealTimeSingleWarningMe*** function just using the user data to be displayed

17. Function Name: GetWarningLightInt
    Description
    The GetWarningLightInt function returns an integer value that indicates if there are any warning signs for a particular team member. The function takes two parameters, a Context object and a String object that represents the name of the team member.
    Parameters
    - context (type: Context): The Context object is used to access the application's resources and services.
    - member (type: String): The String parameter represents the name of the team member for whom the warning light is being checked.

    Return Value
    - warningCheck (type: Int): The function returns an integer value that indicates if there are any warning signs for the team member. The value is 0 if there are no warning signs and 1 if there are any warning signs.

    Code

```
fun GetWarningLightInt(context: Context, member: String): Int {

    val id = Teammates().getIDFromName(member)
    var warningCheck = 0

    for (k in healthSensorsArray) {
```

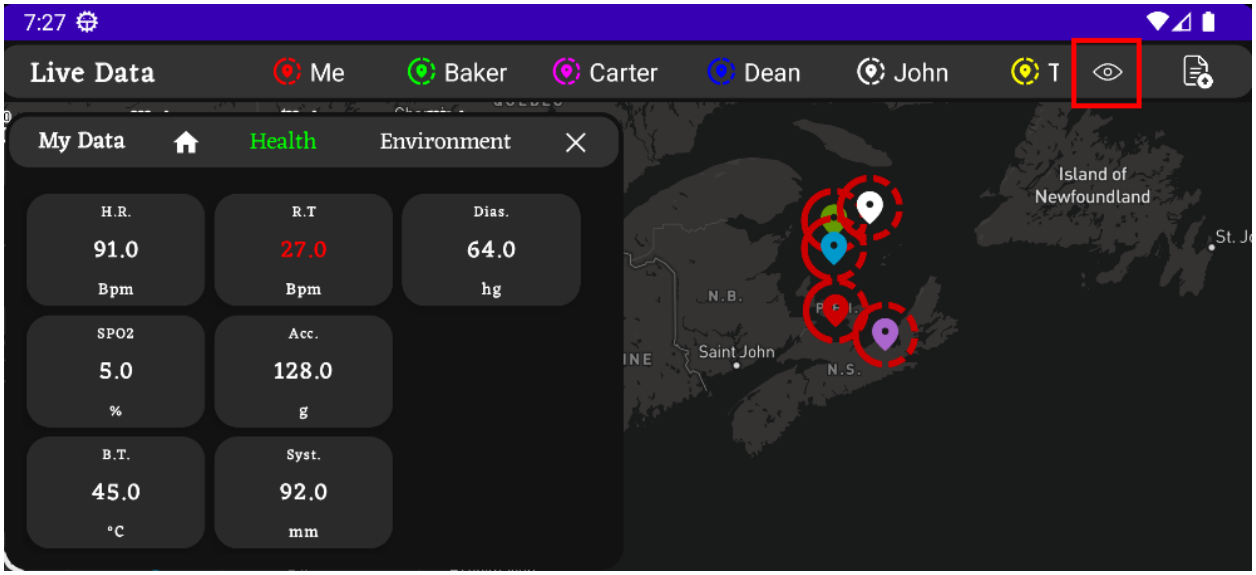
```
        val tempWarning = k.getWarningFromDatabase(context, id)
        if (tempWarning == 1) {
            warningCheck = 1
            break
        }
    }
    for (k in environmentSensorsArray) {
        val tempWarning = k.getWarningFromDatabase(context, id)
        if (tempWarning == 1) {
            warningCheck = 1
            break
        }
    }
    return warningCheck
}
```

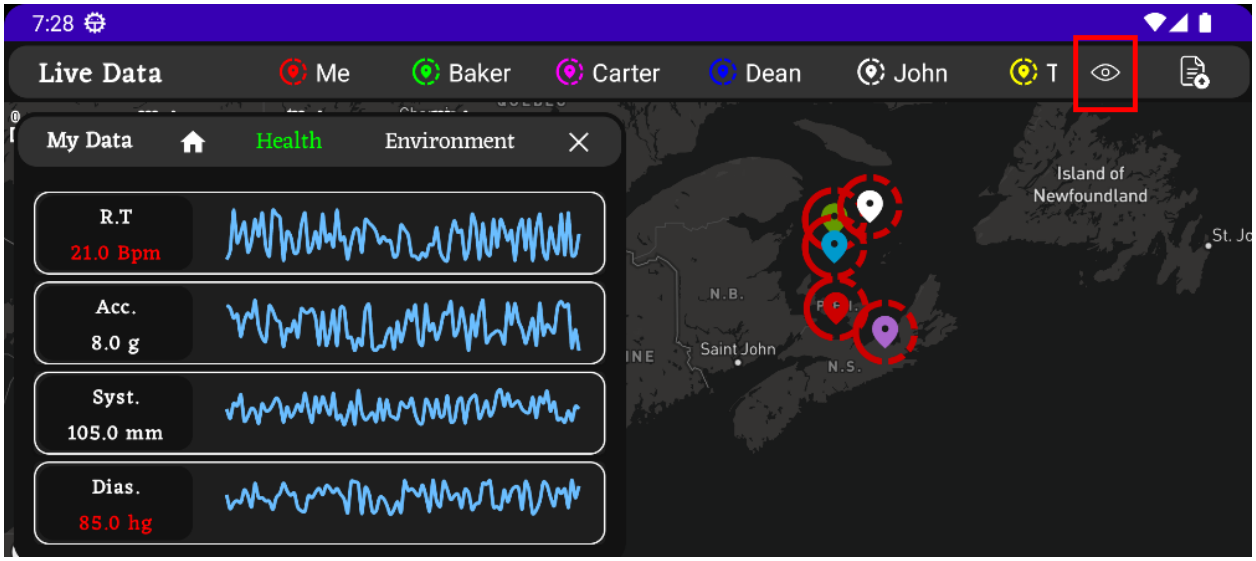
Code Usage

```
val warningLight = GetWarningLightInt(context, "John")
if (warningLight == 1) {
    // Show warning light
} else {
    // Hide warning light
}
```

3.1.5 Ten-Minute Graphs

The small blue graphs throughout the app are ten-minute graphs. They can be toggled globally for the ‘Health’ and ‘Environment’ tabs by clicking the eye icon.





These graphs were implemented with the Charty library. It is available, along with its documentation, here: https://github.com/hi-manshu/Charty

To create a new ten-minute graph which fetches data from the database, the tenMinGraphSensorArray method needs to be called.

Code for Ten-Minute Graph

```
@Composable
	fun TenMinuteGraph (realTimeDataPoint: MutableList<LineData>) {
	if (realTimeDataPoint.isEmpty()) {
		Text(
		text = "No Data",
		modifier = Modifier
			.fillMaxSize()
			.wrapContentSize(Alignment.Center),
		style = TextStyle(
			fontFamily = customfont,
			color = Color.White,
			fontSize = 28.sp,
			textAlign = TextAlign.Center
		)
		)
	}
	else {
		LineChart(
		lineData = realTimeDataPoint,
		color = Color(0xFF6BBEFF),
		modifier = Modifier
			.fillMaxSize()
			.padding(start = 7.dp, end = 5.dp, top = 8.dp, bottom = 5.dp)
		,
		chartDimens = ChartDimens(8.dp),
		axisConfig = AxisConfig(
			false, false,
			false, false,
			xAxisColor = Color(0xFFFFFFFF),
			yAxisColor = Color(0xFFFFFFFF),
			textColor = Color(0xFFFFFFFF)
		),
		lineConfig = LineConfig(true, false)
		)
	}
	}
```

3.2 Historical Set-up

3.2.1 Historical Screen Layout

The historical screen adds two dropdown menus on the navigation bar: the selected person and selected sensor. When selecting one or both of these options, the data in the graph will be updated to show all available datapoints for that sensor as measured by the puck of that person.

The main feature of the historical screen is the graph. This takes up all of the screen space below the navigation bar and shows all of the sensor measurements in the database collected by the selected sensor of the selected person.

- The y-axis shows the recorded value and the x-axis shows the timestamp for when that value was collected.
- Values denoted by a green circle are within the “safe” range for that sensor. Red circles indicate values in the “warning” range.

- The graph is scrollable (by swiping to one side) and zoomable (by pinching the graph).

3.2.2 Generating Line Chart with Madrapps plot

The 'plot' library is an open-source library for Android Compose capable of generating line charts which are scrollable and zoomable. It is available in the following GitHub repository, which also has the most up-to-date setup and usage guide: https://github.com/Madrapps/plot

To create a new historical graph which fetches data from the database, the graphSensorArray method needs to be called.
Code for Historical Graph

```
@Composable
	// TimedDataPoint is simply a pair containing a time object and a Float data
value.
	fun HistoricalGraph(historicalData: MutableList<TimedDataPoint>) {
	val dataPoints: MutableList<DataPoint> = mutableListOf()
	for (i in 0 until historicalData.size) {
		dataPoints.add(DataPoint(i.toFloat(), historicalData[i].data))
	}
	val screenWidth = LocalConfiguration.current.screenWidthDp
	if (historicalData.isEmpty()) Text(
		text = "No Data",
		modifier = Modifier
		.fillMaxSize()
		.wrapContentSize(Alignment.Center),
		style = TextStyle(
		fontFamily = customfont,
		color = Color.White,
		fontSize = 28.sp,
		textAlign = TextAlign.Center
		)
	)
	else {
		val textMeasurer = rememberTextMeasurer()
		var selectedValue: Float by remember {
		mutableStateOf(0f)
		}
		LineGraph(plot = LinePlot(
		listOf(
			LinePlot.Line(
				dataPoints,
				LinePlot.Connection(color = Color(0x77FFFFFF)),
				LinePlot.Intersection { center, point ->

				val warning =
SensorsUI().checkWarningsGraph(point.y.toInt())
				val color = if (warning == 1) Color(0xFFEB0000)
				else Color(0xFF5FEB00)
				drawCircle(
					color,
					6.dp.toPx(),
					center,
				)
				},
				LinePlot.Highlight(color = Color(0xFFFFFFFF), alpha =
0.5f),
				LinePlot.AreaUnderLine(Color(0xFFFFFFFF), alpha = 0.2f)
```

```
                        )
                ),
                grid = LinePlot.Grid(Color(0x44FFFFFF), steps = 5),
                yAxis = LinePlot.YAxis(content = { min, offset, _ ->
                        for (it in 0 until 5) {
                                val value = it * offset + min
                                Text(
                                text = truncateLargeNumber(value.toDouble(), 1),
                                maxLines = 1,
                                overflow = TextOverflow.Ellipsis,
                                style = TextStyle(
                                        fontFamily = customfont,
                                        color = Color.White,
                                        fontSize = 22.sp,
                                        textAlign = TextAlign.Center
                                ),
                                color = MaterialTheme.colors.onSurface
                                )
                        }
                }),
                xAxis = LinePlot.XAxis(content = { _, _, _ ->
                        val numLabels = 10
                        for (it in 0 until numLabels) {
                                Text(
                                text = historicalData[it * (historicalData.size /
numLabels)].time.toString()
                                ,
                                maxLines = 1,
                                overflow = TextOverflow.Visible,
                                style = MaterialTheme.typography.caption,
                                color = MaterialTheme.colors.onSurface
                                )
                        }
                }),
                selection = LinePlot.Selection(
                        detectionTime = 200L,
                        highlight = LinePlot.Connection(
                                draw = { offsetBottom, offsetTop ->

                                val measuredText = textMeasurer.measure(
                                        AnnotatedString(

selectedValue.toDouble().round(0).toLong().toString()
                                        ),
                                        style = TextStyle(
                                        fontFamily = customfont,
                                        color = Color.White,
                                        fontSize = 24.sp,
                                        textAlign = TextAlign.Center
                                        )
                                )
                                var topLeft = offsetTop
                                if (offsetTop.x + measuredText.size.width > screenWidth)
                                        topLeft = Offset(topLeft.x -
measuredText.size.width, topLeft.y)

                                drawLine(
                                        Color.White,
                                        offsetBottom,
                                        offsetTop,
                                        strokeWidth = 7f,
                                        alpha = 0.7f
```

```
                            )
                            drawRect(Color.DarkGray, topLeft,
measuredText.size.toSize())
                            drawText(measuredText, topLeft = topLeft)
                            }
                    )
             )
             ),
             modifier = Modifier
                    .fillMaxSize(),

             onSelection = { xLine, points ->
                    selectedValue = points[0].y
             }
             )
       }
       }
```

3.3 MapSet-up

MapSet-UP :

The map set-up is done using mapbox sdk

Before installing the SDK, you will need to gather the appropriate credentials. The SDK requires two pieces of sensitive information from your Mapbox account.

A public access token: From your account's tokens page, you can either copy your default public token or click the Create a token button to create a new public token.

- A secret access token with the Downloads:Read scope.
    1. From your account's tokens page, click the Create a token button.
    2. From the token creation page, give your token a name and make sure the box next to the Downloads:Read scope is checked.
    3. Click the Create token button at the bottom of the page to create your token.
    4. The token you've created is a secret token, which means you will only have one opportunity to copy it somewhere secure.

Add the credentials in your settings.gradle file :

```
maven {
   url 'https://api.mapbox.com/downloads/v2/releases/maven'
   authentication {
       basic(BasicAuthentication)
   }
   credentials {
       username = "*********"
       password = ************
```

```
    }
}
```

In your gradle.properties make a string of the token to be used in your code:

```
MAPBOX_DOWNLOADS_TOKEN = "  Your Token "
```

Add the following dependencies in your build.gradle(Module) file :

```
implementation 'com.mapbox.maps:android:10.10.1'

buildFeatures{
   viewBinding true
}
```

Add the following dependencies in your build.gradle(Project) file :

```
id 'com.google.android.libraries.mapsplatform.secrets-gradle-plugin' version '2.0.1'
apply false
```

The following fields need to be set-up :

- offlineManager: An instance of OfflineRegionManager that manages the offline download process.
- offlineRegion: An instance of OfflineRegion that represents the downloaded offline map tiles.
- mapView: An instance of MapView that displays the online or offline map.
- binding: An instance of ActivityMapBinding that provides access to the views in the layout file.

```
private lateinit var offlineManager: OfflineRegionManager
private lateinit var offlineRegion: OfflineRegion
private var mapView: MapView? = null
private lateinit var binding: ActivityMapBinding
```

The following methods need to be set-up :

- onCreate(savedInstanceState: Bundle?): Called when the activity is created. It initializes the Mapbox SDK, sets up the offline download process, and inflates the layout file.

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   binding = ActivityMapBinding.inflate(layoutInflater)
   offlineManager =
OfflineRegionManager(MapInitOptions.getDefaultResourceOptions(this@MapActivity))
```

```
   offlineManager.createOfflineRegion(
       OfflineRegionGeometryDefinition.Builder()
           .geometry(point)
           .pixelRatio(2f)
           .minZoom(zoom - 2)
           .maxZoom(zoom + 2)
           .styleURL(styleUrl)

.glyphsRasterizationMode(GlyphsRasterizationMode.NO_GLYPHS_RASTERIZED_LOCALLY)
           .build(),
       callback
   )
```

- downloadComplete(): Called when the offline download process is completed. It hides the progress bar and shows the "Show Map" button.

```
private fun downloadComplete() {
   binding.downloadProgress.visibility = View.GONE
   binding.showMapButton.visibility = View.VISIBLE
   binding.showMapButton.setOnClickListener {
       it.visibility = View.GONE
       // create mapView
       mapView = MapView(this@MapActivity).also { mapview ->
           val mapboxMap = mapview.getMapboxMap()

mapboxMap.setCamera(CameraOptions.Builder().zoom(zoom).center(point).build())
           mapboxMap.loadStyleUri(styleUrl)
       }
       setContentView(mapView)

       mapView?.onStart()
   }
}
   offlineManager.createOfflineRegion(
       OfflineRegionGeometryDefinition.Builder()
           .geometry(point)
           .pixelRatio(2f)
           .minZoom(zoom - 2)
           .maxZoom(zoom + 2)
           .styleURL(styleUrl)

.glyphsRasterizationMode(GlyphsRasterizationMode.NO_GLYPHS_RASTERIZED_LOCALLY)
           .build(),
       callback
   )
```

- onStart(): Called when the activity is started. It initializes the Mapbox SDK.

```
override fun onStart() {
   super.onStart()
   mapView?.onStart()
}
```

- onStop(): Called when the activity is stopped. It stops the Mapbox SDK.

```
override fun onStop() {
   super.onStop()
   mapView?.onStop()
}
```

- onDestroy(): Called when the activity is destroyed. It invalidates the offline region and destroys the Mapbox SDK.

```
override fun onDestroy() {
   super.onDestroy()
   offlineRegion.invalidate { }
   mapView?.onDestroy()
}
```

OfflineRegionObserver
An interface that provides callbacks for the offline download process.
Methods

- mapboxTileCountLimitExceeded(limit: Long): Called when the maximum number of map tiles has been downloaded.
- statusChanged(status: OfflineRegionStatus): Called when the status of the offline download process changes.
- responseError(error: ResponseError): Called when an error occurs during the offline download process.

```
private val regionObserver: OfflineRegionObserver = object : OfflineRegionObserver {
   override fun mapboxTileCountLimitExceeded(limit: Long) {
       logE(TAG, "Mapbox tile count max (= $limit) has exceeded!")
   }

   override fun statusChanged(status: OfflineRegionStatus) {
       logD(
           TAG,
           "${status.completedResourceCount}/${status.requiredResourceCount}
resources; ${status.completedResourceSize} bytes downloaded."
       )
       if (status.downloadState == OfflineRegionDownloadState.INACTIVE) {
           downloadComplete()
           return
       }
   }

   override fun responseError(error: ResponseError) {
       logE(TAG, "onError: ${error.reason}, ${error.message}")
```

```
offlineRegion.setOfflineRegionDownloadState(OfflineRegionDownloadState.INACTIVE)
    }
}
```

OfflineRegionCreateCallback

An interface that provides a callback for creating an offline region.

Methods

- onCreate(expected: Expected<OfflineRegion, OfflineRegionError>): Called when the offline region is created. It stores the offline region and sets the download state to active.

```
private val callback: OfflineRegionCreateCallback = OfflineRegionCreateCallback {
expected ->
    if (expected.isValue) {
        expected.value?.let {
            offlineRegion = it
            it.setOfflineRegionObserver(regionObserver)
            it.setOfflineRegionDownloadState(OfflineRegionDownloadState.ACTIVE)
        }
    } else {
        logE(TAG, expected.error!!)
    }
}
```

MapboxScreen

A Composable function that displays the online or offline map using Jetpack Compose.

Methods

- MapboxScreen(): Called when the MapboxScreen Composable is first displayed. It creates an instance of MapView and adds an annotation to the map.

```
@Composable
fun MapboxScreen() {
    Box(
        contentAlignment = Alignment.Center,
        modifier = Modifier.fillMaxSize()
                .testTag(stringResource(id = R.string.MBScreen))
    ) {
        AndroidView(
            modifier = Modifier,
            factory = { context ->
                ResourceOptionsManager.getDefault(
                    context,
                    context.getString(R.string.MAPBOX_DOWNLOADS_TOKEN)
                )
                //create map
                val initialCameraOptions = CameraOptions.Builder()
                    .center(point)
                    .zoom(zoom)
                    .build()
                val mapInitOptions = MapInitOptions(context,
MapInitOptions.getDefaultResourceOptions(context),
                    MapInitOptions.getDefaultMapOptions(context),
MapInitOptions.defaultPluginList, initialCameraOptions, true)
                val mapView = MapView(context, mapInitOptions)
                mapView.apply {
```

```
                    getMapboxMap().loadStyleUri(Style.DARK)
                    addAnnotationToMap(mapView)
                }
            }
        )
    }
}
```

3.4 Alertdialog Set-up

This section provide the usage instructions and functionality mentioned in DialogAlert class which includes three functions which are AlertDialogMessage() (*it can be used to show alert if the device connection was established or not*), ConsentDialog() (*it can be used to take consent from user regarding File Export*) and AutoExportToaster() (*it can be used to toast the message when App automatically export the data file*). Which are detailed as below:

3.4.1 AlertdialogMessage Function

Function Description
The AlertDialogMessage function is a composable function written in Kotlin. It displays an alert dialog with a title, message, and a dismiss button. The alert dialog is displayed if the initial status of the dialog is set to true. The function is built using Jetpack Compose, a modern toolkit for building UIs in Android apps.
Function Parameters
The AlertDialogMessage function takes in two parameters:

1. title (required): A string representing the title of the alert dialog.
2. message (required): A string representing the message of the alert dialog.

Function Output
The AlertDialogMessage function returns an alert dialog.
Function Implementation
The AlertDialogMessage function is implemented using the AlertDialog composable function. It takes in various parameters that customize the alert dialog's appearance and behavior, such as the dialog's height, width, background color, dismiss button, and more.

The AlertDialog function takes in the following parameters:

1. modifier: A modifier that is applied to the alert dialog's layout.
2. onDismissRequest: A callback that is invoked when the user dismisses the alert dialog.
3. title: The title of the alert dialog.
4. text: The message of the alert dialog.
5. confirmButton: A button that confirms the alert dialog.

In the implementation of the AlertDialogMessage function, the modifier parameter is used to set the height, width, background color, and shadow of the alert dialog. The onDismissRequest parameter is used to dismiss the alert dialog when the user clicks on the dismiss button.
The title parameter is used to set the title of the alert dialog. The text parameter is used to set the message of the alert dialog. The confirmButton parameter is used to add a dismiss button to the alert dialog. When the user clicks on the dismiss button, the openDialogStatus variable is set to false, and the alert dialog is dismissed.
Example Usage
To use the AlertDialogMessage function, simply call it with the required parameters, like this:

AlertDialogMessage(title = "Title", message = "This is a message.")

This will display an alert dialog with the title "Title" and the message "This is a message." The user can dismiss the alert dialog by clicking on the dismiss button.
Code

```
@Composable
fun AlertDialogMessage(title: String, message: String) {
   // Set the initial status of the dialog to be open
   val openDialogStatus = remember { mutableStateOf(true) }
   // Display the alert dialog if the status is true
   if (openDialogStatus.value) {
       AlertDialog(
           modifier = Modifier
               .height(180.dp)
               .testTag(stringResource(id = R.string.Alert_Dialog_1))
               .fillMaxWidth()
               .shadow(elevation = 20.dp)
               .background(
                   color = MaterialTheme.colors.onPrimary,
                   shape = RoundedCornerShape(25.dp, 10.dp, 25.dp, 10.dp)
               ),
           onDismissRequest = { openDialogStatus.value = false },
           title = {
               Text(
                   text = title,
                   textAlign = TextAlign.Center,
                   modifier = Modifier
                       .padding(top = 10.dp)
                       .fillMaxWidth(),
                   color = Color.White,
                   fontFamily = FontFamily.SansSerif
               )
           },
           text = {
               Text(
                   text = message,
                   textAlign = TextAlign.Center,
                   modifier = Modifier
                       .padding(top = 10.dp, start = 25.dp, end = 25.dp)
                       .fillMaxWidth(),
                   color = Color.White,
                   fontFamily = FontFamily.Default,
```

```
                    )
            },
            confirmButton = {
                // Add a dismiss button to close the alert dialog
                Button(
                    onClick = { openDialogStatus.value = false },
                    elevation = ButtonDefaults.elevation(15.dp),
                    colors = ButtonDefaults.buttonColors(Color.Red),
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(6.dp)
                        .shadow(
                            elevation = 4.dp, // adjust the elevation value to achieve
desired shadow effect
                            shape = RoundedCornerShape(40.dp),
                            spotColor = Color.White
                        ),
                    shape = RoundedCornerShape(40.dp),
                    ) {
                    Text(text = "Dismiss", color = Color.White, fontWeight = FontWeight.Bold)
                }
            },
        )
    }
}
```

3.4.2 Consent Dialog Function

Function Description

The ConsentDialog function is a composable function that creates a custom dialog box in Jetpack Compose. It accepts two lambda functions as parameters, onYesClick and onNoClick, which are invoked when the user clicks the "Yes" and "No" buttons respectively.

Parameters:

- onYesClick: () -> Unit: A lambda function that is invoked when the user clicks the "Yes" button.
- onNoClick: () -> Unit: A lambda function that is invoked when the user clicks the "No" button.

Return Value:

- Unit: The function doesn't return any value.

Usage:

To use this function, simply call it and pass in two lambda functions as parameters that should be executed when the user clicks the "Yes" or "No" button respectively.

ConsentDialog(onYesClick = { / handle "Yes" button click */ }, onNoClick = { /* handle "No" button click */ })*

Implementation Details:

- This function uses Jetpack Compose's AlertDialog composable to create a custom dialog box.
- It sets the height and width of the dialog box using the height() and fillMaxWidth() modifiers respectively.
- It also adds a shadow to the dialog box using the shadow() modifier and a custom background color and shape using the background() modifier.
- The onDismissRequest parameter of the AlertDialog is used to call the onNoClick function when the user dismisses the dialog box.
- The title and text parameters are used to set the title and message of the dialog box respectively.
- Two buttons, "Yes" and "No", are added to the dialog box using the buttons parameter.
- The TextButton composable is used to create the "Yes" and "No" buttons.
- The onClick parameter of the TextButton is used to display a toast message and invoke the onYesClick or onNoClick function when the user clicks the respective button.

Code

```
@Composable
fun ConsentDialog(onYesClick: () -> Unit, onNoClick: () -> Unit) {
   // Get the current context
   val context = LocalContext.current
   // Display an AlertDialog with a custom background color and shape
   AlertDialog(
       modifier = Modifier
           .height(210.dp)
           .fillMaxWidth()
           .shadow(elevation = 20.dp)
           .background(
               color = MaterialTheme.colors.onPrimary,
               shape = RoundedCornerShape(25.dp, 10.dp, 25.dp, 10.dp)
           ),
       // Call the onNoClick function when the user dismisses the dialog
       onDismissRequest = { onNoClick() },
       // Set the title and text of the dialog
       title = { Text("Alert!", color = Color.White, fontWeight = FontWeight.Bold)
},
       text = {
           Text(
               "Would you Like to Export your Data",
               color = Color.White,
               fontWeight = FontWeight.Bold
           )
       },
       // Add two buttons to the dialog
       buttons = {
           Row {
               // "Yes" button
               TextButton(
                   elevation = ButtonDefaults.elevation(15.dp),
                   colors = ButtonDefaults.buttonColors(Color.Red),
                   modifier = Modifier
                       .fillMaxWidth()
                       .padding(6.dp)
                       .weight(1F)
```

```
                        .shadow(
                            elevation = 4.dp, // adjust the elevation value to
achieve desired shadow effect
                            shape = RoundedCornerShape(40.dp),
                            spotColor = Color.White // set the desired shadow color
                        ),
                    shape = RoundedCornerShape(40.dp),
                    // Display a toast message and call the onYesClick function when
the user clicks "Yes"
                    onClick = {
                        Toast.makeText(
                            context,
                            "Your Export request has been Executed!",
                            Toast.LENGTH_SHORT
                        ).show()
                        onYesClick()
                    }
                ) {
                    Text("Yes", color = Color.White, fontWeight = FontWeight.Bold)
                }
                Spacer(modifier = Modifier.size(10.dp))
                // "No" button
                TextButton(
                    elevation = ButtonDefaults.elevation(15.dp),
                    colors = ButtonDefaults.buttonColors(Color.Red),
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(6.dp)
                        .weight(1F)
                        .shadow(
                            elevation = 4.dp, // adjust the elevation value to
achieve desired shadow effect
                            shape = RoundedCornerShape(40.dp),
                            spotColor = Color.White // set the desired shadow color
                        ),
                    shape = RoundedCornerShape(40.dp),
                    // Display a toast message and call the onNoClick function when
the user clicks "No"
                    onClick = {
                        Toast.makeText(
                            context,
                            "Your Export request has been Terminated!",
                            Toast.LENGTH_SHORT
                        ).show()
                        onNoClick()
                    }
                ) {
                    Text("No", color = Color.White, fontWeight = FontWeight.Bold)
                }
            }
        }
    )
}
```

3.4.3 Toast Message Function

Function Description:

The AutoExportToaster function is a Composable function written in Kotlin. It displays a toast message to indicate that a file has been exported successfully to the specified path. The function takes a string parameter called 'Path', which represents the location where the file has been exported.

Parameters:

- Path (String): A string parameter that represents the location where the file has been exported.

Return Type:
This function does not return any value.

Code

```
@Composable
fun AutoExportToaster(Path: String) {
   // Get the context from LocalContext
   val context = LocalContext.current

   // Display a toast message to indicate that the file has been exported
successfully
   Toast.makeText(context, "File Exported Successfully in {$Path}",
Toast.LENGTH_SHORT)
       .show()
}
```

3.5 Navigation Set-up

The navigation set-up is done using the activities and the Intent to navigate between the activities. The following process was used :

First right click on MainActivity,
Then go to new,
Then go to the Activity,
Then click on empty activity.
It will pop up a new window, Change the activity name to the page name,
Then uncheck generate a layout file and click on finish.
This will automatically create a new activity in AndroidManifest.xml.
Then we need to make some changes there for the landscape orientation, For example :

```
<activity
   android:name=".Health"
   android:configChanges="orientation"
   android:exported="false"
   android:screenOrientation="sensorLandscape"
   android:theme="@style/Theme.SixthSenseProjectV1">
   <meta-data
       android:name="android.app.lib_name"
       android:value="" />
</activity>
```

Then change the activity to component activity by editing the outer method.
For example :

```
class Health : ComponentActivity() {
   @RequiresApi(Build.VERSION_CODES.O)
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContent {
           SixthSenseProjectV1Theme {
               // A surface container using the 'background' color from the theme
               Surface(
                   modifier = Modifier.fillMaxSize(),
                   color = MaterialTheme.colors.background
               ) {
                   if (connectionChange == 1){
                       connectionChange = 0
                       MainActivity().ShowConnectionWarning(value =
statusConnectionIdentifier)
                   }
                   RealTimeHealth()
               }
           }
       }
   }
}
```

Then use the Intent in the onClick option of the IconButton. It takes two parameters, the context of the current class and the context of the class which should be shown.
For example :

```
IconButton(modifier = Modifier
   .padding(2.dp)
   .size(140.dp, 60.dp)
   .weight(2F)
   .border(
       border = BorderStroke(
           width = 1.dp,
           color = Color.White
       ),
       shape = RoundedCornerShape(10.dp)
   ),
   onClick = {
       val navigate = Intent(this@Health, Health::class.java)
       startActivity(navigate)
   }
)
```