



University of Parma
Department of Engineering and Architecture

Mobile Systems Programming Project

E-Motion Application

David Reverberi

Index

General purpose.....	3
Application requirements	4
Installation and operation guide	6
Starting and Managing the Web Service	6
Installation of the application	7
Operation of E-Motion App	9
Technologies Used	12
Bluetooth implementation.....	12
MongoDB Database	14
OpenWeather API.....	14
Software Architecture	15
Application structure	15
Server structure.....	18
Useful Links	20

General purpose

The E-Motion application is developed with the aim of providing users with an intuitive and functional interface for remote control of a robot specialized in cleaning solar panels. It aims to improve and make cleaning operations easier and more efficient, thus contributing to the optimization of solar energy production and the reduction of maintenance costs.

Main objectives:

1. Robot control

Providing users with the ability to directly control the movement of the robot tracks via interactive sliders. Inclusion of functional buttons to start, stop, and manage cleaning operations.

2. User management

Provide functionality to enable registration and subsequent authentication of users who intend to use the 'application.

3. Weather forecast consultation

To make it possible for registered and authenticated users to consult the weather forecast for the city of interest in real time to obtain information for the hours following the time of consultation. The ultimate goal is to plan cleaning operations optimally.

Overall, the 'app is designed to be used by small- to medium-sized solar system owners, maintenance technicians, and other industry workers. It is also developed to ensure ease of use through an immediate and simple interface that can be accessed via smartphone in a few simple steps.

E-Motion thus represents an innovative and strategic support tool to enable greater control in solar panel maintenance operations and efficient planning of the same operations, based on the weather.

Application requirements

It is specified that the development of E -Motion is aimed at the realization of a prototype model, not suitable for public release, due to some information security issues and the implementation of the associated Web Service, not currently ready for 'exposure on the Internet.

Functional Requirements

1. Motion control

The 'application must provide two interactive sliders that take values from 0 (stationary) to 100 (maximum power output) and transmit them to the corresponding motors, one for controlling the vertical (forward/backward) movement of the robot and the other for horizontal (right/left) control. Both sliders are to return to the neutral position in the middle (value 50) when the user releases them. A dedicated button for emergency shutdown of the motors is also provided .

2. Control of cleaning operations

The 'application must provide two buttons for control of cleaning operations: one button dedicated to releasing water and the other dedicated to activating the robot brushes.

3. User management

The 'application must ensure that the user is able to access the robot controls following an 'appropriate authentication. It must also be possible for new users to register, allowing them to create personal accounts identified by username, email and password.

4. Consultation of the weather forecast

The 'user must be able to access a dedicated interface for retrieving weather information about the entered city. The 'application must show the forecast for the next few hours in a clear and understandable manner to facilitate the user in planning cleaning operations.

Non-functional requirements

1. General performance

The graphical elements of the controls screen should respond almost instantaneously to user interactions, with no more than a half-second delay. Buttons that handle login, registration, and weather requests should respond in reasonable time, no more than 20 seconds, and that an 'error is reported to the user if the operation is unsuccessful.

2. Web Service Performance

The web service associated with E- Motion must respond to the http requests it receives quickly, so it is necessary not to burden the API calls to the weather service with unsolicited information or excessive processing, as well as the consultation of the online database to retrieve information about user account data.

3. Using Bluetooth

The 'user should only be able to access the control interface if authenticated and successfully connected to the Bluetooth device associated with the robot. The 'authentication occurs only after the user has been notified of the connection status with the Bluetooth device.

4. Usability and accessibility

The 'application must be intuitive and easy to use, with clear navigation between screens. At least half of the users worldwide who have smartphones with the Android operating system must be able to download the application and use it effectively.

5. Maintainability and scalability

A modular structure must be provided to allow the 'addition of new functionality that does not impact existing components. The realized prototype must be prepared for the introduction of security protocols and for the online distribution of the realized web server for future public release. The 'application must be able to handle an increasing number of users without degrading performance.

Installation and operation guide

All the files specified in the following paragraphs are either contained in the project folder along with this report or can be downloaded [here](#) from the project's official Git Hub repository. To get to the 'working application you must first start the server containing the Web Service and poi later install the application on the device.

Starting and Managing the Web Service

The 'server startup is necessary as the application is developed in its prototype version. In the final version it is intended to make this step no longer necessary by making a server distribution public and reachable on the Internet by all users.

Before starting the server on the computer or on the generic Host device at hand, it should be noted that if you intend to install and test the application on one of the supported emulators available on Android Studio, the server must be started on the same device on which the application runs with the IDE.

To start `ilserver` in Windows, you must locate to the path where the web service Python file (`WebService.py`) resides and type the following commands after opening the console:

```
$env: FLASK_APP = "Web Service"  
flask --app Web Service run --host 0.0.0.0 --port= 5000
```

In case you have UNIX as your operating system you have to specify only the second command.

Once started, the server will be listening on all interfaces of the referenced network, so it will be discoverable and reachable by all devices connected to the same network via http protocol.

Installation of the application

The 'app' is developed for Android versions greater than or equal to 12 (Snow Cone). Attempting to run E-Motion on devices with older versions could lead to various bugs or even inability to launch the app itself.

If you do not have an Android device with a supported version, you can take advantage of one of the various emulators made available on Android Studio, after importing and properly running the project renamed as " **E- MotionApp-Emulator** " in the IDE. However, this mode of use is restrictive and valid only for a general verification of the application, due to the impossibility of integrating the Bluetooth (and therefore robot control) functionality of the emulator.

Two modes of installation with a physical device are identified:

1. Installation and startup from the Android Studio IDE

After properly connecting the Android device to the computer with Android Studio installed, you need to import the " **E- MotionApp** " project into the IDE and start it using your personal device as the target.

Before running the application, however, it is necessary to change in the **WebService.kt** file the IP address that identifies the device on which the server was started, by going to replace the string " **http://10.0.2. 2:5000** " with " **http://<Ip- address>:5000** " where <Ip- Address> must be replaced with the IP of the device, of the same network, on which the server was started.

```
val retrofit = Retrofit.Builder() Retrofit.Builder
    .baseUrl(" http://192.168.1.56:5000 ") //Change this with the IP of the server Device
    .addConverterFactory(MoshiConverterFactory.create(moshi)) Retrofit.Builder
    .client(okHttpClient)
    .build()
```

Example where the IP address 192. 168. 1. 56 was used.

This step is necessary for the client (the application) to be able to correctly reach the server (computer or generic device on which the Web Service is running).

If the 'application does not start correctly, please refer to the next section of E-Motion App operation, for the part related to permissions. This mode of using the 'app also allows you to check for a Bluetooth connection with the robot (identified by the generic Bluetooth module HC-05).

2. Installation and startup from the application apk

The 'application installation is done through the apk file contained in the project folder. Also in this mode it is possible to use Bluetooth for connection with i l HC -05 module, if available to the user. However, with this latest version of the application, it will not be possible to contact the server for login, registration and weather information operations, but only to verify the Bluetooth operation.

The 3 installation methods and features are summarized below:

- **E-MotionApp-Emulator** project : To test only server functions.
- **E-MotionApp** project : To test all functions with own device
- **Apk E- Motion App** : To test only functionality with Bluetooth

If you do not have the HC-05 Bluetooth module to connect, we recommend the first mode (with emulator), as it still provides the ability in the 'application to navigate all screens even without a Bluetooth connection.

Operation of E-Motion App

Once the 'app has been properly installed on the Android device, you need to verify that the required permissions have been accepted. If an automatic pop-up requesting Bluetooth and location permissions does not appear when the application is launched and the application fails to start, you must manually enable the permissions from the device settings, in the appropriate section of application management, for the E-Motion application.

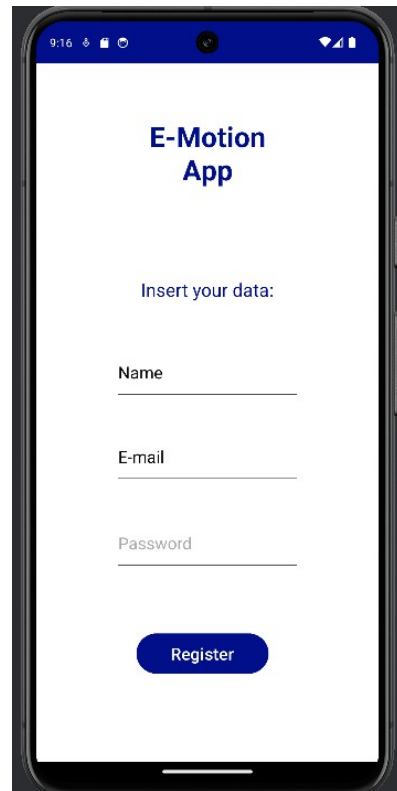
Login and registration screen

Once the 'application has been started and the associated permissions verified the login screen will appear on the display (see image 1 below). Here you will be able to fill in the relevant fields with your credentials in order to access the robot controls. If, on the other hand, you have not already created a personal account, you can do so by pressing the "register" button at the bottom of the screen. A 'registration interface will open (See image 2 below) where you will have to enter your credentials.

In the 'app (except in the version developed for testing on the emulator) it will not be possible to access the controls interface without a connection to the Bluetooth module HC- 05 (present inside the robot, it is therefore necessary to make the association first, then start the app.



1. Login screen



2. Registration screen

To speed up access time to the controls, you can log in with the credentials:
username: *guest* and password: *guestpassword*

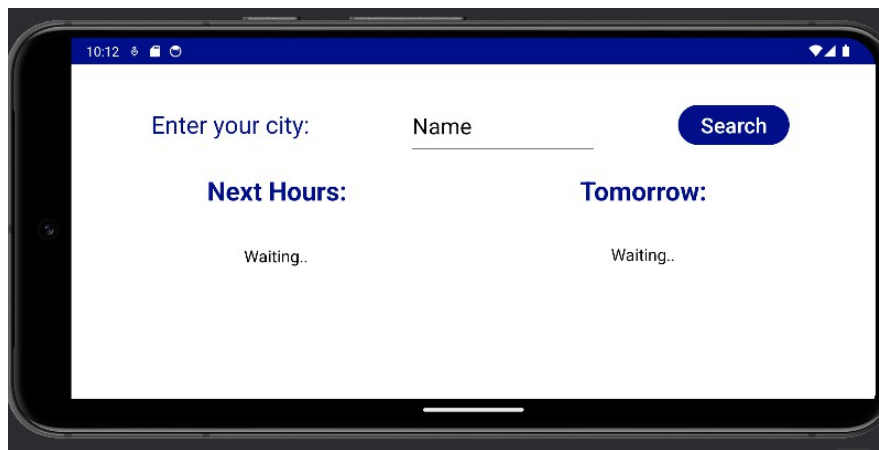
Controls and weather screen

Once logged into the system you will access the 'controls interface (see image 3 next) where you will be able to maneuver the robot via the two interactive sliders, use the clean (clean), water (water) and motor stop (stop) buttons, and access the interface with weather information via the 'weather button.

In the weather forecast browsing interface (see image 4 next) you will be asked to indicate the city of interest for which you are interested in the information. It will be pressing the appropriate Search button that in less than a few seconds (depending on the readability and speed of the server) will bring the information back to the slots below.



3. Controls screen



4. Weather forecast screen

Each of the four interfaces presented provides for handling errors of various kinds that may occur. For example, if the credentials on the login screen are incorrect or the server or database cannot be reached, an error will be reported and access to the 'controls screen will be prevented. Similarly, if all the fields in the registration screen are not filled in or the username indicated is already existing, the user will be informed with an appropriate message shown in the GUI. Searching for a city that is not available among those in the reference weather API will also generate an error that will be graphically communicated to the user.

Should you test the 'application with the emulator you will still be able to access the interface of the controls and interact with them , even without a Bluetooth connection.

Technologies Used

Bluetooth implementation

The Bluetooth protocol was chosen for communication between the robot and the mobile device. B T connectivity was implemented by following a series of steps, from verifying the permissions required for operation, to using client- and server-side sockets for communication between the i l device and the HC-05 module residing in the robot.

Bluetooth activation and configuration

After indicating in the Android Manifest the permissions required by the application, to initialize the resources necessary for Bluetooth func tioning, an instance of the default Bluetooth Adapter() class in the program's MainActivity will be the one to allow all interactions with the components that handle Bluetooth.

Request to enable permissions and connect with associated devices If the user has not enabled permissions, there is an automatic prompt in the software through a pop-up warning to the user:

```
//Checking if the bluetooth permission is granted or not
if (bluetoothAdapter?.isEnabled == false) {
    if (ActivityCompat.checkSelfPermission(
        context: this,
        Manifest.permission.BLUETOOTH_CONNECT
    ) != PackageManager.PERMISSION_GRANTED
    ) {
        // If not granted, permission is requested to the user
        if (Build.VERSION.SDK_INT > 31) {
            ActivityCompat.requestPermissions(
                activity: this,
                arrayOf(android.Manifest.permission.BLUETOOTH_CONNECT),
                requestCode: 100
            )
            return
        }
    }
}
```

Verification and application for necessary permits

Once the permissions are verified, we proceed with the detection of the associated devices by exploiting the bonded Devices() method of the instance of the bluetoothAdapter() class, mentioned earlier. Among all the associated BT devices detected by the system, only the one of interest (identified by the name HC-05) will be detected and its MAC address will be extracted, which is necessary for the next phase of the actual connection.

```
//Permission granted --> Getting all the paired devices
val pairedDevices: Set<BluetoothDevice>? = bluetoothAdapter?.bondedDevices
pairedDevices?.forEach { device ->
    val deviceName = device.name
    val deviceHardwareAddress = device.address // MAC address
    Log.d( tag: "cnt", msg: "Device Name --> $deviceName, Device MAC --> $deviceHardwareAddress")
    if (deviceName == "HC-05") {
        val deviceHMAC = device //Address of interest
        Log.d( tag: "cnt", msg: "HC-05 MAC: --> ${deviceHMAC!!.address}")
    }
}
```

Obtaining all associated devices and identification of the HC -05 module.

Instead, the connection with the BT device is done in a separate thread to avoid congestion in the Main Activity. We initially establish the communication channel with the previously acquired information by going to define an instance of the Bluetooth Socket class that identifies the socket, then connect with the connect() method and identify the output stream with the outputStream() method, also from the BluetoothSocket class. These operations are performed only if the BT device of interest has been detected among all the associated ones, otherwise an error message is returned to the 'user for inability to identify the HC-05 module.

Writing to the output stream

Once the communication channel with the Bluetooth HC - 05 module is started, whenever the user interacts with the widgets used to control the robot, the writeBytes() function of the main Activity() is called, whose task is to write the bytes it receives as a parameter in the communication stream with the BT module.

```

// Call this function to send data to the remote device.
@ Davide Reverberi
fun writeBytes(bytes: ByteArray) {
    try {
        if (OutputStream == null){
            Log.d( tag: "error", msg: "OutputStream is null --> Can't communicate")
        }
        OutputStream?.write(bytes)    //Writing the Bytes in the stream
        //Thread.sleep(50)
    } catch (e: IOException) {
        Log.e( tag: "error", msg: "Error occurred when sending data", e)
    }
}
}

```

Detail of the function of writing bytes to the stream

MongoDB database

MongoDB is a NoSQL database that in E- Motion was used to manage and store user account information. This choice allowed for a robust and scalable user authentication and management system. Each user account is saved in JSON format with name, email and password information and communication with the database is mediated by the Web Service associated with the application, through http calls from the Android application (client) to the server itself (Web Service) , which in turn queries the database.

OpenWeather API

The developed Android application includes a feature to obtain weather information using the OpenWeather API. This API was used as the main source to obtain accurate and real-time weather data, which is then processed and sent to the client via the specially developed Web Service. When a user enters a valid city name, upon pressing the Search button in the weather control interface, a GET-type http request is generated to the WebService. When the latter 'receives the request, a specific API key is used to authenticate and send a request for the weather information to the OpenWeather API, which will then be processed and returned to the client.


Software Architecture

The overall architecture of the application is divided into two components that can communicate with each other: the client (Android app) and the server (web service).

Application structure

Main Activity Management

The 'overall software architecture of the app is developed to make sure that all operations are started and finished by a single MainActivity. Within it, all necessary procedures are handled for the entire lifecycle of Bluetooth connections, from requesting permissions to connecting and sending data to the module. Similarly, all operations associated with the Web service are handled within the same Activity . Specifically, in the onStart() method of the Main Activity, a binding is performed whose purpose is to "bind" the operation of the service accessing information processed on the server to the Activity's life cycle.

A screenshot of a code editor showing the implementation of the onStart() method in Kotlin. The code is for a class named 'Davide Reverberi'. The method 'override fun onStart()' contains the following logic: it calls 'super.onStart()', comments '//Bind to local WebService', creates an 'Intent' with 'packageContext: this' and 'WebService::class.java', and then calls 'bindService' with the intent, a variable 'connection', and the flag 'Context.BIND_AUTO_CREATE'.

```
▲ Davide Reverberi
override fun onStart() {
    super.onStart()
    //Bind to local WebService
    val boundIntent = Intent( packageContext: this, WebService::class.java)
    bindService(boundIntent,connection, Context.BIND_AUTO_CREATE)
}
```

Detail of the on Start() method, in which the " binding" with the web service is performed

Navigation between the different graphical interfaces provided is made possible by the navigation graph associated with the activity_main.xml file, thanks to a FragmentContainer View. Consequently, again, all developed graphics depend directly on the lifecycle of the main activity.

To avoid congesting the 'activity, the use of threads (as in the case of Bluetooth connes sion) or coroutines is very often used in the software structure.

In particular, user login and registration operations, having to wait for a response from the server, must be handled in an execution unit separate from the one handling the GUI update, to avoid congesting the graphics and compromising the 'user experience'. Thus, coroutines play a key role in the 'overall architecture'.

Management of graphics and user interaction

As already anticipated, each of the four user-navigable interfaces is based on a precise flow graph that is hosted in the xml file associated with the main Activity, with custom connections and graphics depending on the objective of each screen. Each of these interfaces has an xml style file in which the graphical elements are defined, and a kotlin file containing the class that handles interactions with them. When the user interacts with a graphical widget, specific functions are activated that take care of transferring information and reacting to the user's input in real time.

Communication with the server

The service that handles interactions with the actual web service uses the Moshi library for serializing JSON files, Retrofit to support networking and interaction with the server API, and configuration of an Ok Http client with custom settings for connection timeouts, reading and sending data.

```
// Configure OkHttpClient with custom timeout settings --> Registration process may require several seconds
val okHttpClient = OkHttpClient.Builder()
    .connectTimeout( timeout: 30, TimeUnit.SECONDS) // Connection timeout
    .readTimeout( timeout: 30, TimeUnit.SECONDS)    // Read timeout
    .writeTimeout( timeout: 30, TimeUnit.SECONDS)   // Write timeout
    .build()
```

It was chosen to use 30 seconds before a timeout is detected

The WebService defined in the WebService.kt file provides three methods for calling functions to contact the server . The WebServiceAPI interface, defined in the ServiceUtil. kt file, takes care of translating these calls into http requests of the GET and POST types.

The following are the 3 possible calls to the web service API, where the data structures containing the and information to be included in the body of POST requests are also visible.

```
interface WebServiceAPI {  
  
    @GET("weather")  
    suspend fun getWeather(@Query("city") city: String): Response<WeatherReport>  
  
    @POST("register")  
    suspend fun registerUser(@Body userProfile: UserProfile): Response<RegisterResponse>  
  
    @POST("login")  
    suspend fun loginUser(@Body userCredentials: UserCredentials): Response<LoginResponse>  
}
```

Design Patterns

A number of useful design patterns were used in the development of the application . Most of them were used implicitly, not so much because of the application's design choice, but more because of how the logic of the Android API and classes was implemented.

Of these, the **Singleton** is perhaps the most recurring model of all because of the logic with which the various listener methods of the widgets (buttons and seek Bars) are implemented. Whenever the user interacts with these components, a 'single instance of the listener is returned, thus avoiding the creation of multiple instances of the same object. The connection to the web service also uses a Singleton to ensure that there is only one instance of the connection, improving the efficiency and consistency of network operations.

Another design pattern used in software development is **Observer**.

This behavioral type pattern is implicitly used in the interaction between the service connection (ServiceConnection) and the Main Activity. Due to the binding and this design pattern, when the service connects or disconnects, the MainActivity is notified via the onServiceConnected and onServiceDisconnected methods (in the MainActivity. kt file).

```

/** Defines callbacks for service binding, passed to bindService(). */
private val connection = object : ServiceConnection {
    override fun onServiceConnected (className: ComponentName, service: IBinder) {
        // We've bound to service, cast the IBinder and get service instance.
        val binder = service as MyBinder
        customService = binder.getService()
        mBound = true
    }
    override fun onServiceDisconnected (arg0: ComponentName) {
        mBound = false
    }
}

```

Example of the use of Singleton and Observer in the connection with the service

Also recurring in the code is the 'implicit use of the **Adapter** pattern to manage compatibility between the different libraries of Retrofit, Moshi, and OkHttpClient. This pattern allows cooperation between these libraries without changing their internal code in any way.

Server structure

Flask and components

The server is implemented using the Flask framework, a lightweight framework suitable for developing web applications in Python. The server was also developed to be easily extensible, with the 'addition of any new web APIs. The components needed to connect with the MongoDB database and to interact with OpenWeatherAPI were integrated into the server. The keys used to access the above services are given in the .env file but should you wish to create your own server with your own database you must replace them with your own keys.

API Endpoints

The server provides three API endpoints to enable the operations of accessing, recording, and querying the meteorological information:

1. **/weather:** Endpoint that accepts a GET-type request with the parameter "city" specifying the city to be considered for processing the weather information. Weather information for the requested city is returned in response.

2. / register: POST-type endpoint that handles user registration. JSON data is processed and credentials are saved to the MongoDB database if the username is not detected as already existing.

3. /login: POST-type endpoint that handles user access operations to the application. Again, information in JSON format regarding username and password that is received from the client is processed. The server takes care of checking for any matches in the database and returns an error message or authentication confirmation.

An additional endpoint dedicated to testing the 'database activity is made available:

4. /test_db_connection: Endpoint that takes care of testing the connection to the MongoDB database. It returns a success message if the connection is successfully established or an error message if not.

Useful Links

1. Android documentation on Bluetooth:

<https://developer.android.com/develop/connectivity/bluetooth?hl=it>

2. MongoDB Atlas:

<https://www.mongodb.com/it-it/atlas>

3. OpenWeather API:

<https://openweathermap.org/>

4. Flask:

<https://flask.palletsprojects.com/>

5. Git Hub repository of the project:

<https://github.com/DaddaRev/E-Motion-App/tree/Web-Service>