ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Desenvolvimento de Aplicações Móveis
Mobile Application Development
DAM
Tutorial 2 - Weather App

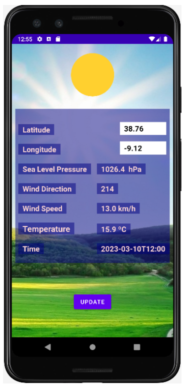
Ana Duarte Correia
ana.correia@isel.pt

Pedro Fazenda
pedro.fazenda@isel.pt

Abstract

This tutorial aims to guide students in the creation of an Android application that can adjust its interface to any screen size and exhibit information related to the present local weather. To acquire the required weather information, students will utilize a REST API. Throughout the tutorial, students will learn how to design diverse layouts, incorporate themes, enable multi-language support, enhance their proficiency in Kotlin programming, and optionally explore the optimal strategies for building Android applications using the MVVM design pattern.

Deadline: March 24th, 2024



Latitude	38.76
Longitude	-9.12
Sea Level Pressure	1026.4 hPa
Wind Direction	214
Wind Speed	13.0 km/h
Temperature	15.9 °C
Time	2023-03-10T12:00

Cool Weather Application

March 11th, 2024

Contents

List of Figures

1 Introduction

This work aims to introduce the application development environment for Android devices. The tutorial introduces the anatomy of an Android program. From the application "Hello World" a new application with a more elaborate user interface is built. The CoolWeatherApp aims to create a user-friendly weather application with a customizable interface suitable for any screen size and themed content. It displays a range of meteorological data for a specific location defined by its latitude and longitude coordinates, including **Sea Level Pressure**, **Wind Direction**,**Wind Speed**, and **Temperature**. The application also presents the date and time the data was obtained, along with an image representing the current weather conditions at the location. For instance, a sun icon symbolizes clear skies during the daytime. By default, the app displays weather information for Lisbon (latitude: 38.076, longitude: -9.12). Users can update the location by entering new coordinates and pressing the "update" button to view the corresponding weather information.

This tutorial is designed to walk you through the development of the application, providing useful guidance on how to proceed. Along with a few code snippets, it aims to equip students with the necessary knowledge and resources to actively fill in any missing pieces of the puzzle. As such, it is highly recommended that learners take advantage of the provided hints and proactively read the relevant documentation to implement the missing code effectively. You can find all the required documentation at the following location:

Android Developers Guide: `https://developer.android.com/guide`

2 Laboratory Work - Weather Application

2.1 Obtaining Weather Information

1. **Testing the Open-Meto API**. In this project we will be using **Open-Meteo** keyless APIs that are free for non-commercial use. Go to the following website: `https://open-meteo.com/en/docs`, select the city of Lisbon and test the API by selecting Hourly and Daily Weather variables. You will be able to preview the variables you select on a chart and, bellow that chart, you will find the corresponding API URL that you can use to make a HTTP request and obtain a JSON response with all the information you selected. This URL will look something like this:

   ```
   https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41&current_
   weather=true&hourly=temperature_2m
   ```

   Play with the different variables, read the documentation, and see how the API URL changes. Pay special attention to the information regarding WMO Weather interpretation codes (bottom of the page). We will be using these codes in our application to describe the current weather in our location (clear sky, Fog, etc.).

2. **Obtain Weather Information** Copy and Paste the API URL on youre browser's address bar, execute the request and you should receive a JOSN response with the weather

information you requested in the URL string. To have a prettier view of this information, copy the JSON string and use the following JSON Pretty Print Online tool: https://jsonformatter.org/json-pretty-print.

2.2 Create a Project and set the Background

1. In this section we will create a **New Project** in Android Studio to display weather information in the main activity. Go ahead and create an **Empty Views Activity**, with the name "Cool Weather APP'", in package dam_AXXXX.coolweatherapp (replace XXXX with your student number).

2. Lets use as a starting reference the Google Pixel 3 AVD. The Google Pixel 3 has a 5.5-inch OLED display with a resolution of 1080 x 2160 pixels and a pixel density of 443 pixels per inch (ppi). It has a viewport size is 393 x 786 PX with 161 actual pixel density, which means it have 2.75 xxhdpi display pixel density. Search the Web for a suitable background image with at least 393 x 786 PX. If you have to edit your images, you can use, for example, Gimp (https://www.gimp.org/). Resize and crop the image if the original size is larger than 393 x 786 and convert the image to png format. Copy the image to the drawable folder of the project. Image names should only contain lower case a-z, 0-9, or underscore or else they will not appear in the Pick a Resource window. Images should be placed in the drawables directory (not any other like drawable-v24).

3. Set the image as the background of your main activity. Open the "activity_main.xml" in the res/layout folder. Open the "Split" tab in the lower left corner of the editor to switch to the Layout Editor. Select the <constraintLayout> element and in the properties panel (lower right corner) put the name of "container" in the attribute id of the element. In the "background" attribute, place the image you copied in the previous point, by pressing the right most vertical rectangular/rounded shape to open the Pick a Resource window and select the previous image. Run the program on the Pixel 3 AVD created in the previous tutorial.

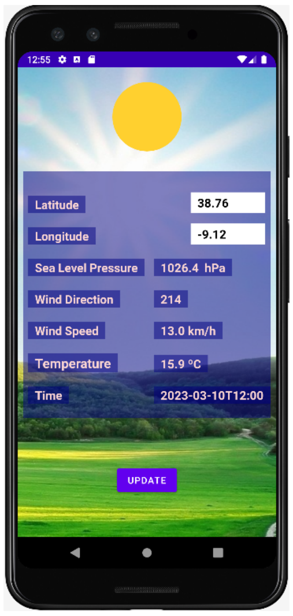
4. Create a landscape layout and adapt the background image so that if fits perfectly to this layout. To accomplish this, crop the higher resolution image that was used in the previous section in a way that is looks good with a size of 786 x 393.

5. Create and add a new portrait and landscape layout for different screen size and resolution (select, for example, a tablet). Create a new AVD and test how this layout look in this new device. Adjust the background images appropriately.

2.3 Designing the layout

To create a layout resembling the portrait and landscape layouts depicted in Figures 1 and 2, choose the required view components from the Palette. Your UI creativity can earn you an additional 10% based on its quality. Follow the steps below to build your layout:

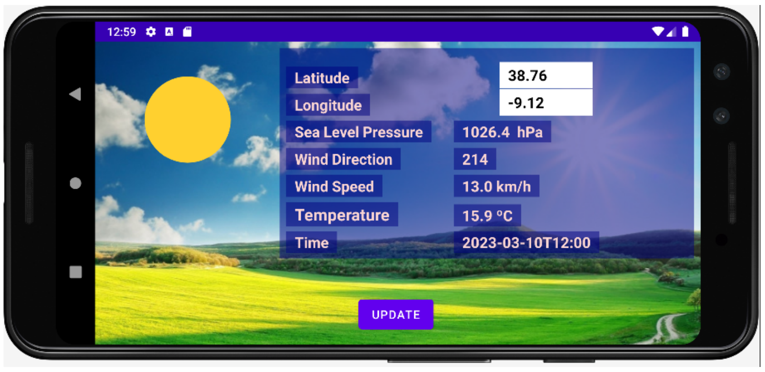
1. To represent the current weather, import the color or black Vector Assets provided in the "Drawables.zip" file accompanying this tutorial. The icons in this file illustrate



12:55

Latitude 38.76
Longitude -9.12
Sea Level Pressure 1026.4 hPa
Wind Direction 214
Wind Speed 13.0 km/h
Temperature 15.9 ºC
Time 2023-03-10T12:00

Figure 1: Portrait layout configuration for the Smart Weather App



12:59

Latitude 38.76
Longitude -9.12
Sea Level Pressure 1026.4 hPa
Wind Direction 214
Wind Speed 13.0 km/h
Temperature 15.9 ºC
Time 2023-03-10T12:00

Figure 2: Landscape layout.

different weather conditions. For instance, in our layout, the sun was representing a "clear day" in Lisbon on the following date: "2023-03-10T12h00". [1]. To add the entire directory, go to "Tools/Resource Manager", press "+"(Add resources to module) and select "Import Drawables".

2. Please review the following documentation:

   `https://developer.android.com/develop/ui/views/theming/themes`

   Your application features two distinct themes: a Light Theme and a Dark Theme. Each theme should be personalized, and you should test them by adjusting the display settings found at:

   `https://developer.android.com/develop/ui/views/theming/darktheme`

   All colors must be defined in the "values/colors.xml" file. In addition, you should expand the Light Theme to include the new themes outlined in listings 1. You can switch between themes programmatically using the code shown in listings 2, where the "day" boolean variable is determined from the "Time" at a specific location, as well as the "Sunrise" and "Sunset" hours obtained from the weather API. We can do this later, for now just declare "day" as a **boolean** variable that you can change to test the different themes.

```xml
<resources xmlns:tools="http://schemas.android.com/tools">
   <!-- Base application theme. -->
   <style name="Theme.CoolWeatherApp"
         parent="Theme.MaterialComponents.DayNight.NoActionBar">
      <!-- Primary brand color. -->
      <item name="colorPrimary">@color/purple_500</item>
      <item name="colorPrimaryVariant">@color/purple_700</item>
      <item name="colorOnPrimary">@color/white</item>
```

[1]Obtained from: https://github.com/Tomorrow-IO-API/tomorrow-weather-codes

```
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->

        <item name="background">@color/viewBG</item>
        <item name="android:textColor">@color/WeatherLabel</item>
        <item name="android:textSize">18sp</item>
        <item name="android:textStyle">bold</item>

    </style>

    <style name="Theme.Day" parent="Theme.CoolWeatherApp">
        <item name="android:windowBackground">@drawable/sunny_bg</item>
    </style>

    <style name="Theme.Day.Land" parent="Theme.CoolWeatherApp">
        <item name="android:windowBackground">@drawable/sunny_bg_land</item>
    </style>

    <style name="Theme.Night" parent="Theme.CoolWeatherApp">
        <item name="android:windowBackground">@drawable/night_bg</item>
    </style>

    <style name="Theme.Night.Land" parent="Theme.CoolWeatherApp">
        <item name="android:windowBackground">@drawable/night_bg_land</item>
    </style>
</resources>
```

Listing 1: Creating new Themes.

```
  override fun onCreate(savedInstanceState: Bundle?) {
         ...
        when (resources.configuration.orientation) {
            Configuration.ORIENTATION_PORTRAIT ->
               if(day) setTheme(R.style.Theme_Day)
               else setTheme(R.style.Theme_Night)
            Configuration.ORIENTATION_LANDSCAPE ->
               if(day) setTheme(R.style.Theme_Day_Land)
               else setTheme(R.style.Theme_Night_Land)
        }
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
    }
```

Listing 2: onCreate

3. Construct the layout using an <ImageView>, <Button>, <EditText> and <TextView> elements. Dont forget to assign a differnet "id" for each one. Latter, we will manipuate the values of theses element sin Kotlin and for that reason we will need the id. For some elements it is necessary to put text. Make new strings in the file "strings.xml" with the content and use them in the respective attribute. Also, define the <EditText> as **android:inputType="numberDecimal|numberSigned"**.

4. Execute the program and check if the UI is close to what you expect. Small changes in the layout of the elements are expected.

5. Navigate to your "strings.xml" file and access the Strings Editor located in the top right corner. Add two languages (Portuguese and English), and assign the appropriate text for each string and language. Once this is complete, modify the language settings on your device and run the application. Verify that all strings change correctly according to the selected language.

6. Modify the boolean variable "day" and ensure that the Theme changes correctly in response.

7. Find out how to change the application launcher icon and change it.

2.4 Application Code

In this section, we will incorporate the business logic of our application directly into the MainActivity class. Execute the following:

1. To get started, create a new Kotlin class file named 'WeatherData.kt' and add the code shown in listings 3. Please note that the class arrangement provided is for demonstration purposes only and should be customized to match the data retrieved from the Weather API. The classes **WeatherData**, **CurrentWeather**, and **Hourly** will store the information obtained from the Weather API. The enumeration **WMO_weatherCode** contains integer codes and drawable icons representing various weather conditions. You can utilize a MAP to retrieve the name of the drawable associated with the weather code obtained from the API and use it to depict the weather in the layout of the Cool Weather Application.

```
data class WeatherData(
    var latitude: String,
    var longitude: String,
    var timezone:  String,
    var current_weather: CurrentWeather,
    var hourly: Hourly

)
data class CurrentWeather (
    var temperature : Float,
    var windspeed : Float,
    var winddirection : Int,
    var weathercode : Int,
    var time : String
)
data class Hourly (
    var time: ArrayList<String>,
    var temperature_2m: ArrayList<Float>,
    var weathercode: ArrayList<Int>,
    var pressure_msl:ArrayList<Double>
)

enum class WMO_WeatherCode(var code: Int, var image: String) {
    CLEAR_SKY(0,"clear_"),
    MAINLY_CLEAR(1, "mostly_clear_"),
    PARTLY_CLOUDY(2,"partly_cloudy_"),
    OVERCAST(3,"cloudy"),
    FOG(45,"fog"),
    DEPOSITING_RIME_FOG(48,"fog"),
    DRIZZLE_LIGHT(51, "drizzle"),
    DRIZZLE_MODERATE(53, "drizzle"),
    DRIZZLE_DENSE(55,"drizzle"),
    FREEZING_DRIZZLE_LIGHT(56,"freezing_drizzle"),
    FREEZING_DRIZZLE_DENSE(57,"freezing_drizzle"),
    RAIN_SLIGHT(61,"rain_light"),
    RAIN_MODERATE(63,"rain"),
    RAIN_HEAVY(65,"rain_heavy"),
    FREEZING_RAIN_LIGHT(66,"freezing_rain_light"),
    FREEZING_RAIN_HEAVY(67,"freezing_rain_heavy"),
    SNOW_FALL_SLIGHT(71,"snow_light"),
    SNOW_FALL_MODERATE(73,"snow"),
    SNOW_FALL_HEAVY(75,"snow_heavy"),
    SNOW_GRAINS(77, "snow"),
    RAIN_SHOWERS_SLIGHT(80, "rain_light"),
    RAIN_SHOWERS_MODERATE(81, "rain"),
    RAIN_SHOWERS_VIOLENT(82, "rain_heavy"),
    SNOW_SHOWERS_SLIGHT(85, "snow_light"),
    SNOW_SHOWERS_HEAVY(86, "snow_heavy"),
    THUNDERSTORM_SLIGHT_MODERATE(95, "tstorm"),
    THUNDERSTORM_HAIL_SLIGHT(96,"tstorm"),
    THUNDERSTORM_HAIL_HEAVY(99,"tstorm")
}
fun getWeatherCodeMap() : Map<Int,WMO_WeatherCode> {
    var weatherMap = HashMap<Int,WMO_WeatherCode>()
    WMO_WeatherCode.values().forEach {
        weatherMap.put(it.code,it)
    }
    return weatherMap
}
```

Listing 3: WeatherData.kt

2. To send an API request to **open-meto.com**, utilize the code presented in listings 4. To use this code you need to add the following line to your "AndroidManifest.xml" file:

```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

add also the following dependency to you "build.gradle" file (Module:app):

```
implementation 'com.google.code.gson:gson:2.8.9
```

and also all of the necessary imports to your "MainActivity.kt" file.

```
private fun WeatherAPI_Call(lat: Float, long: Float) : WeatherData {
  val reqString = buildString {
      append("https://api.open-meteo.com/v1/forecast?"
      append("latitude=${lat}&longitude=${long}&")
      append("current_weather=true&")
      append("hourly=temperature_2m,weathercode,pressure_msl,windspeed_10m")
  }
  val str = reqString.toString()
  val url = URL(reqString.toString());
  url.openStream().use {
    val request = Gson().fromJson(InputStreamReader(it,"UTF-8"),WeatherData::class.java)
    return request
  }
}
```

Listing 4: WeatherAPI_Call

3. In Android, the UI thread is responsible for drawing and updating the user interface. If any long-running task is performed on this thread, it can cause the UI to freeze, leading to an unresponsive and poor user experience. Therefore, it is recommended to run any time-consuming or blocking operations on a separate thread, usually through the use of **AsyncTask** or Thread. However, when it is necessary to update the UI from a background thread, **runOnUiThread** can be used to execute code on the UI thread from the background thread. By using this method, we ensure that any UI updates happen on the UI thread and not on the background thread, thus avoiding any potential UI freezing issues. Listings 5 shows how you can call the weather API and update the UI.

   At the start of the application and whenever the user presses the update button, you must invoke the **fetchWeatherData** method. Updating the UI involves utilizing the latitude and longitude coordinates to retrieve weather data and using that data to modify the weather variables, weather image, and background based on the local hour and sunrise/sunset information for that location.

```kotlin
private fun fetchWeatherData(lat: Float, long: Float) : Thread {
  return Thread {
    val weather = WeatherAPI_Call(lat,long)
    updateUI(weather)
  }
}


private fun updateUI(request: WeatherData)
{
  runOnUiThread {

    val weatherImage : ImageView = findViewById(R.id.weatherImage)
    val pressure : TextView = findViewById(R.id.pressureValue)

    //TODO ...

    pressure.text = request.hourly.pressure_msl.get(12).toString() + "  hPa"

    //TODO...

    val mapt = getWeatherCodeMap();
    val wCode = mapt.get(request.current_weather.weathercode)
    val wImage = when(wCode) {
      WMO_WeatherCode.CLEAR_SKY,
      WMO_WeatherCode.MAINLY_CLEAR,
      WMO_WeatherCode.PARTLY_CLOUDY->if(day) wCode?.image+"day" else wCode?.image+"night"
      else -> wCode?.image
    }
    val res = getResources()
    weatherImage.setImageResource(R.drawable.fog)
    val resID = res.getIdentifier(wImage, "drawable",getPackageName());
    val drawable = this.getDrawable(resID);
    weatherImage.setImageDrawable(drawable);

    //TODO...

  }
}
```

Listing 5: Fetch Weather Data and Update UI

4. To test your application, modify the latitude and longitude coordinates and observe how the weather image, background, and weather parameters update accordingly. To choose suitable latitude and longitude coordinates, refer to Figures 3 and 4, which demonstrates how the altitude and longitude coordinate system operates.

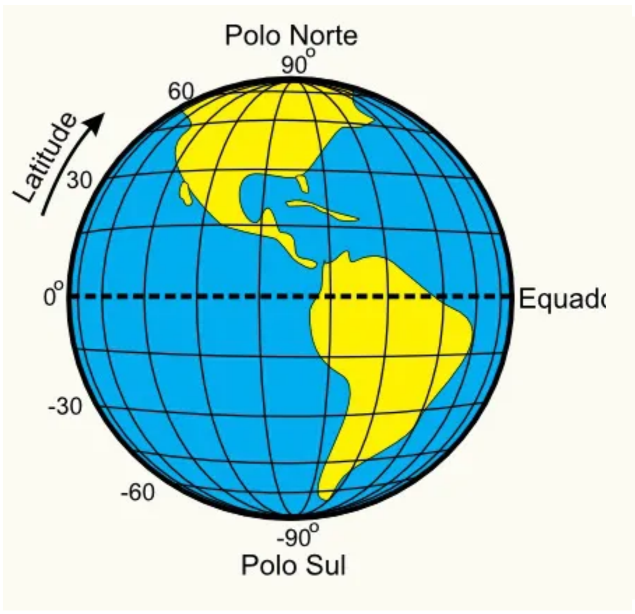


Figure 3: Latitude is a geographic coordinate that specifies a location's position north or south of the equator.

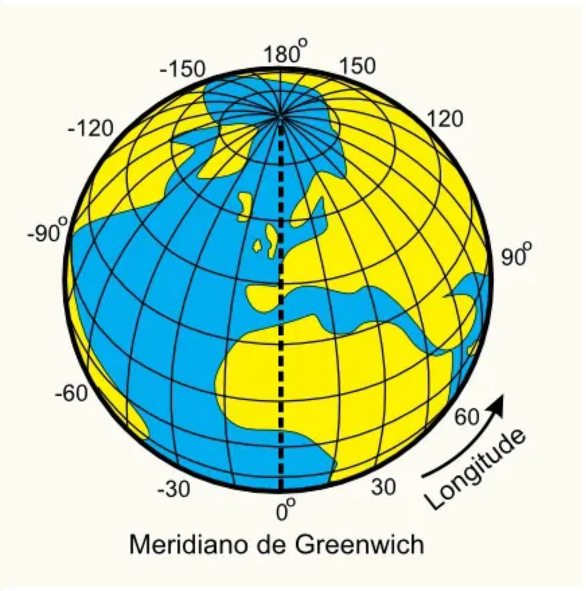


Figure 4: Longitude is a geographic coordinate that specifies a location's position east or west of the Prime Meridian.

3 Optional Work

In this section, you will find optional tasks that you can complete to earn up to 30% of the grade assigned to this tutorial.

3.1 Adding Functionality and Modifying Code in the App

- Upon launching the Cool Weather Application, display the latitude and longitude coordinates of the device, along with the corresponding weather information, instead of using the default Lisbon coordinates.
- Rather than using a hardcoded enumeration for WeatherCodes, modify the code to retrieve the relevant information (Weather Code, Description, and Image) from a resource XML file. Refer to the slides in "DAM 02 Android - Anatomy of Android Apps.pptx" for more information on using array resources.

3.2 Implementing the Model-View-ViewModel design pattern

As you delve deeper into Android development, it's important to become familiar with design patterns that can make your code more organized, scalable, and testable. One such pattern is the **MVVM (Model-View-ViewModel)** design pattern, which separates the business logic from the user interface and helps to keep your codebase organized. As part of your learning journey, you are required to modify the Cool Weather Application by incorporating the MVVM design pattern into it. Specifically, you will need to refactor the code in the **MainActivity** to comply with the MVVM pattern. Here are the steps you need to follow:

- Extract the business logic from the MainActivity and place it into a ViewModel class.
- Use data binding to bind the View to the **ViewModel**.
- Implement **LiveData** to allow the ViewModel to communicate with the View.
- Use the Observer pattern to observe changes in the ViewModel and update the View accordingly.
- Test your application to ensure that it works as expected and that the code is clean, efficient, and organized.

By completing this task, you will gain hands-on experience in implementing the MVVM design pattern in an Android application, which will help you to write better code and build more sophisticated applications in the future. Below are several helpful links that can aid you in completing this task:

- `https://medium.com/swlh/understanding-mvvm-architecture-in-android-aa66f7e1a70b`
- `https://www.youtube.com/watch?v=eUQebUJLnXI`
- `https://www.toptal.com/android/android-apps-mvvm-with-clean-architecture`
- `https://developer.android.com/codelabs/basic-android-kotlin-training-repository-pattern#0`
- `https://www.section.io/engineering-education/implementing-a-repository-kotlin/`
- `https://developer.android.com/topic/libraries/view-binding`
- `https://developer.android.com/topic/libraries/data-binding`
- `https://developer.android.com/topic/libraries/architecture/lifecycle`