AKADEMIA GÓRNICZO-HUTNICZA
im. S. Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

“Aplikacja Pogodowa - tutorial”

Kierunek:
Informatyka:
Przedmiot:
Wprowadzenie do technologii mobilnych
Prowadzący:
dr. inż. Paweł Skrzyński
Autorzy:
Krzysztof Pala
Szymon Czaplak
Data utworzenia:
18.06.2020, Semestr letni 2019/2020

Spis Treści

1. Opis aplikacji

1.1 Funkcje aplikacji

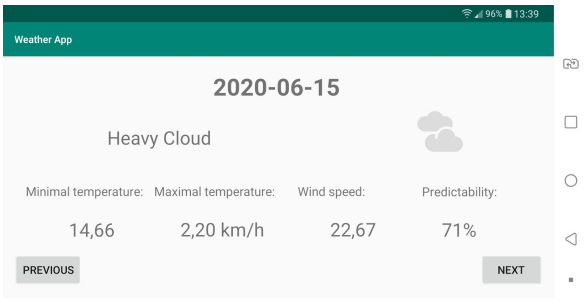
Tworzona aplikacja ma za zadanie wyświetlać wybrane dane dotyczące pogody dla określonej lokalizacji na najbliższe 5 dni. Do pobrania niezbędnych danych wykorzystuje ***metaweather API*** dostępne na stronie: https://www.metaweather.com/api/.

Podstawowy widok zawiera następujące dane:

- Dane pobrane z ***meta weather API***:
- datę opisywanego dnia
- krótki opis charakteru pogody w danym dniu
- ikonę przedstawiającą charakter pogody
- minimalną i maksymalną temperaturze w danym dniu
- prędkość wiatru
- przewidywalność pomiaru
- Pozostałe:
- Nazwa aplikacji
- Przyciski previous i next pozwalające na zmianę dnia wyświetlanej pogody

Przy naciśnięciu przycisku previous oraz next wyświetla się pogoda dla poprzedniego, lub następnego dnia.

Aplikacja obsługuje także widok horyzontalny.



A także wykrywa włączony tryb samolotowy na urządzeniu. (dokładny opis tej funkcjonalności znajduje się w sekcji jej poświęconej).

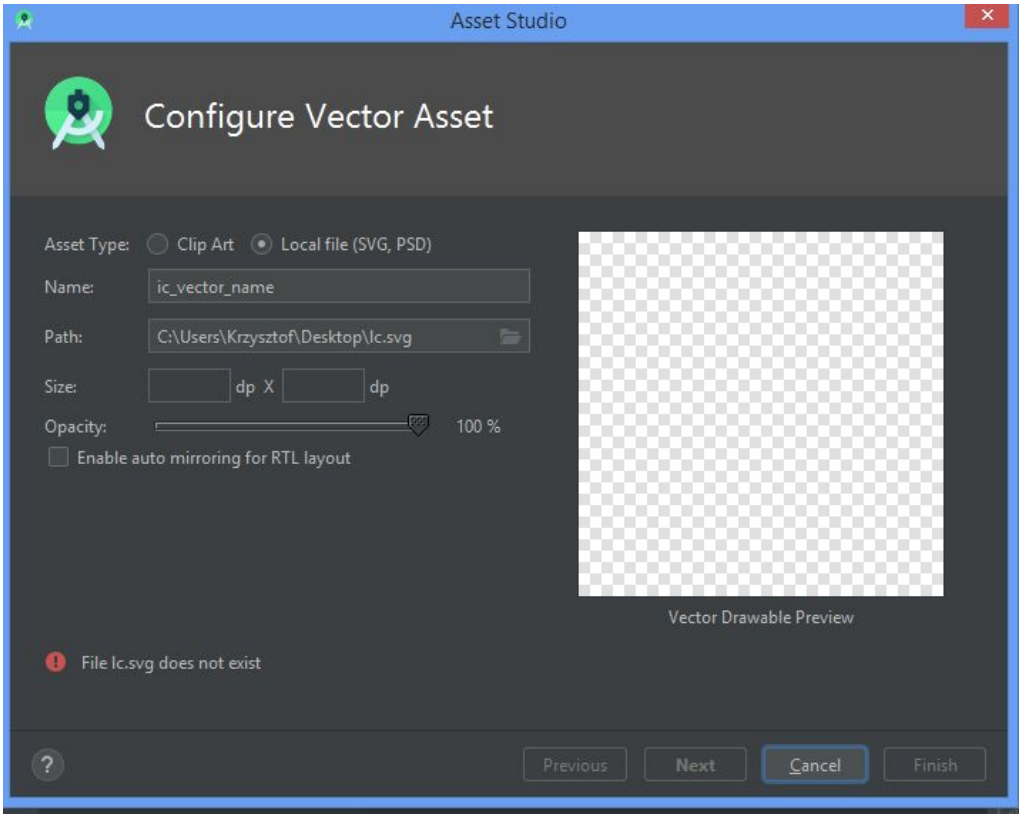
1.2 Wykorzystane technologie/biblioteki

- **Android Studio v3.6**
- minSDKVersion: 15
- targetSDKVersion: 29
- android.net - pobieranie zasobów z meta weather API
- org.JSON - konwersja zasobów pobranych z API
- android.database - obsługa bazy danych SQL Lite
- standardowe biblioteki androida: os, util, view, widget etc.
- standardowe bibliteki javy: util, io etc.

2.Tutorial

2.1 Inicjalizacja projektu

Inicjalizacja projektu w Android Studio nie wymaga żadnych niestandardowych działań. Należy kliknąć file -> new -> newProject a następnie stworzyć projekt z wykorzystaniem EmptyActivity template. Ważną rzeczą jest jednak pobranie i zapisanie w projekcie ikon pogody, które będą potrzebne na dalszym etapie prac nad aplikacją. Aby to zrobić należy pobrać ikony w rozszerzeniu svg ze strony https://www.metaweather.com/api/. Następnie należy je dodać do folderu app/res/drawables klikając prawym przyciskiem mysze na ten folder new -> vector asset a następnie wskazując lokalizację zapisanych plików.



Należy dodać wszystkie 10 ikonek zapisując ich nazwy jako **ic_ORYGINALNA_NAZWA.** Jest to pewne uproszczenie, które będziemy dalej wykorzystywali.

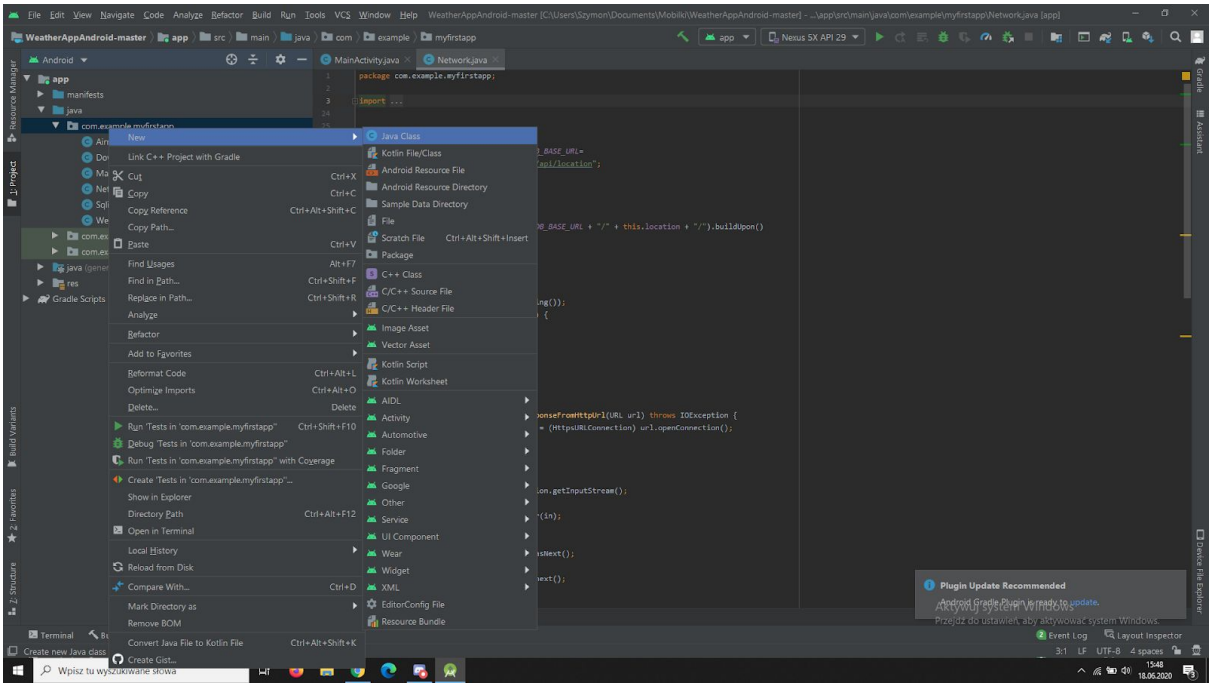
2.2 Komunikacja z meta weather API

Aby uzyskać informacje o pogodzie trzeba wybrać API, które będziemy odpytywać i pobierać stosowne informacje. Zalecamy użycie "www.metaweather.com/api" ze względu na wygode użytkowania i brak konieczności zakładania konta i dzielenia się swoimi danymi osobowymi. Minusem tego API jest to, że nie ma tam wszystkich miast.

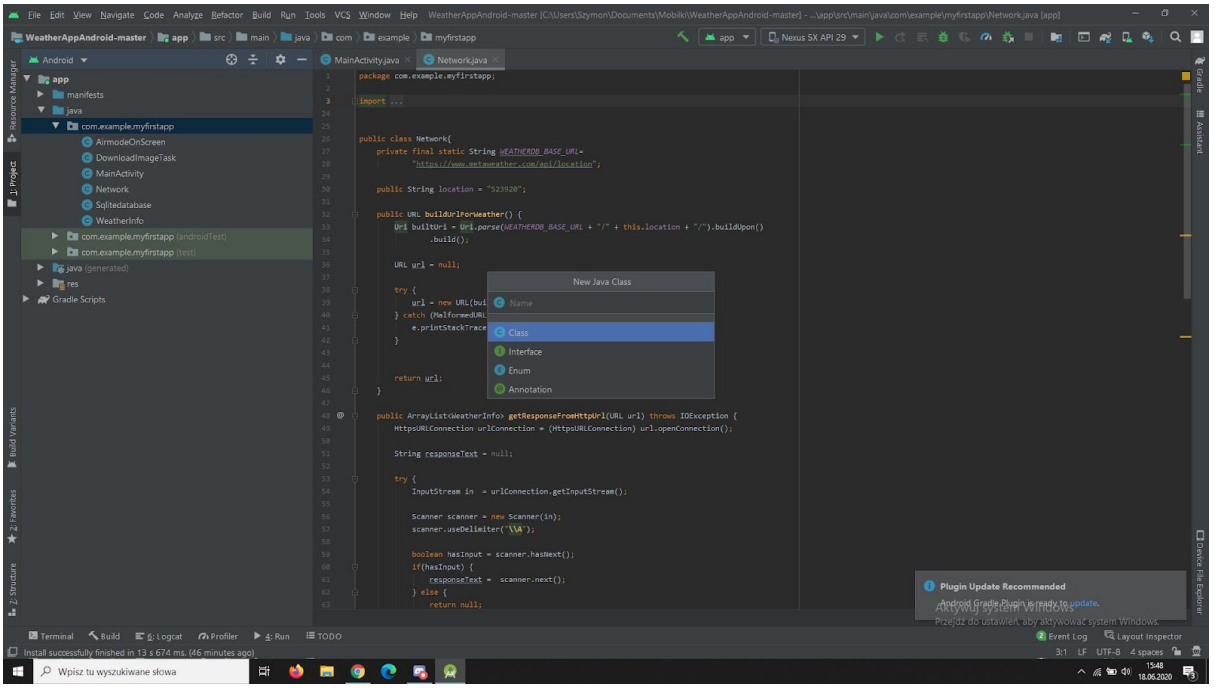
W celu komunikacji z API strony metaweather wystarczy wykonać odpowiedniego requesta na odpowiedni adres (do znalezienia na stronie API) i dostaniemy dane w formacie json. Zalecamy użycie pojedynczego adresu, np:
https://www.metaweather.com/api/location/523920
gdzie, zgodnie z instrukcją na stronie www.metaweather.com/api , numer 523920 jest kodem lokalizacji Warszawy. Aby znaleźć taki kod dla dowolnego miasta wpisujemy w przeglądarkę:
www.metaweather.com/api/location/search/?query=**MOJE_MIASTO**
gdzie MOJE_MIASTO oznacza stringa (nawet niekompletnego) z nazwą miasta.
Po wpisaniu https://www.metaweather.com/api/location/search/?query=War:
otrzymujemy w przeglądarce json:
[{"title":"Warsaw","location_type":"City","woeid":523920,"latt_long":"52.235352,21.009390"},{"title":"Newark","location_type":"City","woeid":2459269,"latt_long":"40.731972,-74.174179"}]
skąd wyłuskujemy ze "woeid" to 523920. I tego będziemy dalej używać w aplikacji. Ten endpoint na API byłby przydatny gdybyśmy zaimplementowali wybór miast przez użytkownika. Wtedy należałoby po prostu odpytywać API i podstawiać stringa który wprowadzi użytkownik.

Aby kod był czytelny zalecamy enkapsulacje całej logiki tworzenia requestów do API w jednej klasie, dla przykładu "Network". Aby utworzyć nową klasę klikamy prawym przyciskiem na folder "com.example.myfirstapp" i wybieramy:

New -> Java Class



Następnie wpisujemy “Network w okienku:



W skonstruowanej w ten sposób klasie tworzymy statyczne pole WEATHERDB_BASE_URL w którym przechowujemy adres do api, którego będziemy używać. Przydatnym będzie wydzielenie WOEID do osobnej zmiennej statycznej, co ułatwi przejście na inne miasto, bądź późniejsze rozbudowanie aplikacji do obsługi większej ilości miast i ponowny reuse kodu.

Pierwszą metodą, którą zalecamy zrobić jest metoda buildUrlForWeather w której będziemy w odpowiedni sposób konkatenować stringi, żeby wyszedł nam adres URL. Tutaj też jest robiona walidacja URL i obsługa wyjątku. Zwracaną wartością jest typ URL.

```
private final static String WEATHERDB_BASE_URL=
        "https://www.metaweather.com/api/location";

public String location = "523920";

public URL buildUrlForWeather() {
    Uri builtUri = Uri.parse(WEATHERDB_BASE_URL + "/" + this.location + "/").buildUpon()
            .build();

    URL url = null;

    try {
        url = new URL(builtUri.toString());
    } catch (MalformedURLException e) {
        e.printStackTrace();
    }


    return url;
}
```

Kolejną i ostatnią funkcją będzie funkcja getResponseFromHttpUrl, która przyjmuje zrobiony wcześniej adres URL (typ URL).
Pierwszą rzeczą która ta funkcja powinna robić jest połączenie się z API oraz uzyskanie stringa jsonowego z odpowiedniego adresu. W tym celu otwieramy połączenie z konkretnym url i później za pomocą klasy Scanner pobieramy odpowiedź serwera.

```
public ArrayList<WeatherInfo> getResponseFromHttpUrl(URL url) throws IOException {
    HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();

    String responseText = null;

    try {
        InputStream in  = urlConnection.getInputStream();

        Scanner scanner = new Scanner(in);
        scanner.useDelimiter("\\A");

        boolean hasInput = scanner.hasNext();
        if(hasInput) {
            responseText =  scanner.next();
        } else {
            return null;
        }
    } finally {
        urlConnection.disconnect();
    }
```

Następnie tak uzyskaną odpowiedź musimy sparsować do odpowiedniej struktury danych. W tym celu musimy wpierw przyjrzeć się odpowiedzi API:


```
{
  "consolidated_weather": [
    {
      "id": 6619281288593408,
      "weather_state_name": "Heavy Rain",
      "weather_state_abbr": "hr",
      "wind_direction_compass": "ESE",
      "created": "2020-06-18T13:22:58.743570Z",
      "applicable_date": "2020-06-18",
      "min_temp": 19.34,
      "max_temp": 27.07,
      "the_temp": 25.535,
      "wind_speed": 4.870855955977094,
      "wind_direction": 113.83199043358044,
      "air_pressure": 1008,
      "humidity": 70,
      "visibility": 11.114777201145312,
      "predictability": 77
    },
    {
      "id": 6687771253014528,
      "weather_state_name": "Heavy Rain",
      "weather_state_abbr": "hr",
      "wind_direction_compass": "ESE",
      "created": "2020-06-18T13:23:01.751524Z",
      "applicable_date": "2020-06-19",
      "min_temp": 18.175,
      "max_temp": 27.985,
      "the_temp": 25.825,
      "wind_speed": 5.6270111019724816,
      "wind_direction": 119.16356081330208,
      "air_pressure": 1008,
      "humidity": 63,
      "visibility": 10.845412789310426,
      "predictability": 77
    },
    {
      "id": 5277413246238720,
      "weather_state_name": "Heavy Rain",
      "weather_state_abbr": "hr",
      "wind_direction_compass": "W",
      "created": "2020-06-18T13:23:04.756712Z",
```

```
  "applicable_date": "2020-06-20",
  "min_temp": 18.369999999999997,
  "max_temp": 23.75,
  "the_temp": 22.24,
  "wind_speed": 4.788224701325593,
  "wind_direction": 280.48171627644206,
  "air_pressure": 1010.5,
  "humidity": 79,
  "visibility": 10.537834049152947,
  "predictability": 77
},
{
  "id": 5461719386161152,
  "weather_state_name": "Light Rain",
  "weather_state_abbr": "lr",
  "wind_direction_compass": "NW",
  "created": "2020-06-18T13:23:07.742544Z",
  "applicable_date": "2020-06-21",
  "min_temp": 17.465,
  "max_temp": 24.235,
  "the_temp": 22.675,
  "wind_speed": 5.888659997351468,
  "wind_direction": 310.5,
  "air_pressure": 1012,
  "humidity": 74,
  "visibility": 12.753022349479043,
  "predictability": 75
},
{
  "id": 5056854327558144,
  "weather_state_name": "Light Rain",
  "weather_state_abbr": "lr",
  "wind_direction_compass": "NNW",
  "created": "2020-06-18T13:23:10.868559Z",
  "applicable_date": "2020-06-22",
  "min_temp": 17.835,
  "max_temp": 24.445,
  "the_temp": 22.33,
  "wind_speed": 7.431780355916118,
  "wind_direction": 332.84926866798514,
  "air_pressure": 1015.5,
  "humidity": 75,
  "visibility": 9.626282510140777,
  "predictability": 75
},
```

```
  {
    "id": 6696291260170240,
    "weather_state_name": "Showers",
    "weather_state_abbr": "s",
    "wind_direction_compass": "NNW",
    "created": "2020-06-18T13:23:14.049708Z",
    "applicable_date": "2020-06-23",
    "min_temp": 16.545,
    "max_temp": 25.765,
    "the_temp": 23.42,
    "wind_speed": 6.882353256979241,
    "wind_direction": 348.5,
    "air_pressure": 1019,
    "humidity": 65,
    "visibility": 9.999726596675416,
    "predictability": 73
  }
],
"time": "2020-06-18T16:06:00.400040+02:00",
"sun_rise": "2020-06-18T04:14:00.084403+02:00",
"sun_set": "2020-06-18T21:00:30.420973+02:00",
"timezone_name": "LMT",
"parent": {
  "title": "Poland",
  "location_type": "Country",
  "woeid": 23424923,
  "latt_long": "51.918919,19.134300"
},
"sources": [
  {
    "title": "BBC",
    "slug": "bbc",
    "url": "http://www.bbc.co.uk/weather/",
    "crawl_rate": 360
  },
  {
    "title": "Forecast.io",
    "slug": "forecast-io",
    "url": "http://forecast.io/",
    "crawl_rate": 480
  },
  {
    "title": "HAMweather",
    "slug": "hamweather",
    "url": "http://www.hamweather.com/",
```

```
    "crawl_rate": 360
   },
   {
    "title": "Met Office",
    "slug": "met-office",
    "url": "http://www.metoffice.gov.uk/",
    "crawl_rate": 180
   },
   {
    "title": "OpenWeatherMap",
    "slug": "openweathermap",
    "url": "http://openweathermap.org/",
    "crawl_rate": 360
   },
   {
    "title": "World Weather Online",
    "slug": "world-weather-online",
    "url": "http://www.worldweatheronline.com/",
    "crawl_rate": 360
   }
  ],
  "title": "Warsaw",
  "location_type": "City",
  "woeid": 523920,
  "latt_long": "52.235352,21.009390",
  "timezone": "Europe/Warsaw"
}
```


Odpowiedź jest dosyć długa, ale po głębszej anali zie nie jest ona zbyt skomplikoawna. Dostajey duży słownik, w którym znajduje się pole “consolidated_weather” w którym jest array słowników o postaci:


```
 {
    "id": 6687771253014528,
    "weather_state_name": "Heavy Rain",
    "weather_state_abbr": "hr",
    "wind_direction_compass": "ESE",
    "created": "2020-06-18T13:23:01.751524Z",
    "applicable_date": "2020-06-19",
    "min_temp": 18.175,
    "max_temp": 27.985,
    "the_temp": 25.825,
    "wind_speed": 5.6270111019724816,
    "wind_direction": 119.16356081330208,
    "air_pressure": 1008,
    "humidity": 63,
    "visibility": 10.845412789310426,
    "predictability": 77  }
```

Zatem najpierw musimy całą odpowiedź przeparsować do klasy JSONObject, z którego wyciągniemy pole “consolidated_weather”. Następnie array kryjący się pod tym kluczem parsujemy za pomocą klasy JSONArray i każdy jego element parsujemy do JSONObject.

```
ArrayList<JSONObject> list = new ArrayList<~>();
try {

    JSONObject full_resp = new JSONObject(responseText);

    JSONArray jsonArray = new JSONArray(full_resp.get("consolidated_weather").toString());

    if (jsonArray.length() != 0) {
        int len = jsonArray.length();
        for (int i=0;i<len;i++){
            list.add(new JSONObject(jsonArray.get(i).toString()));
        }
    }

} catch (JSONException e) {
    Log.d( tag: "DATA", responseText);
    Log.d( tag: "FAIL",  msg: "FAILED TO PARSE RESPONSE");
    e.printStackTrace();
}
```

W ten sposób w obiekcie list mamy array obiektów o typie JSONObject, które reprezentują słownik określający pogodę w konkretnym dniu. Czyli dokładnie tego czego nam potrzeba!

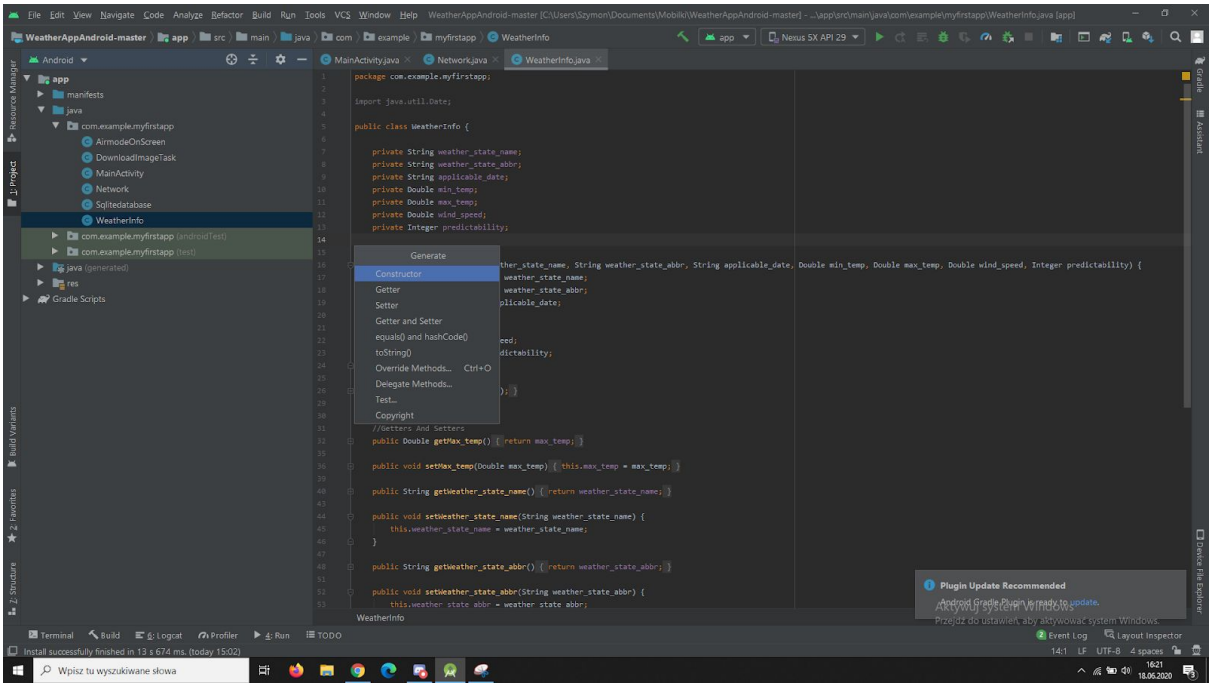
Przechowywanie obiektów w typie JSONObject nie jest zbyt wygodne, więc zróbmy nową klasę, która ułatwi nam dużo operacji. Nazwijmy ją WeatherInfo.
Będzie to prosta klasa przechowującą informacje z konkretnych pól ze słownika (który aktualnie mamy w JSONOBject).
W nowym pliku o nazwie WeatherInfo tworzymy następującą klasę:

```
public class WeatherInfo {

    private String weather_state_name;
    private String weather_state_abbr;
    private String applicable_date;
    private Double min_temp;
    private Double max_temp;
    private Double wind_speed;
    private Integer predictability;
```

Zauważ prosze zbieżność pól z kluczami słownika z odpowiedzi z API.

następnie klikamy ALT + Insert



i generujemy domyślne gettery, settery oraz konstruktor.
Tyle nam będzie potrzebne do dalszych działań. Wracając do klasy network i do metody getResponseFromHttpUrl:

Dla wygody tworzymy zmienną all_info, w której będą dokładnie te same informacje co w list, ale w typie WeatherInfo a nie JSONObject. Iterując po obiektach z list parsujemy i rzutujemy odpowiednie klucze z obiektów JSONObject i używając odpowiedniego konstruktora dodajemy do zmiennej all_info informacje o pogodzie. Tak przygotowane dane zwracamy.

```
ArrayList<WeatherInfo> all_info = new ArrayList<>();

for (JSONObject record : list){
    try {
        String weather_state_name = record.getString( name: "weather_state_name");
        String weather_state_abbr = record.getString( name: "weather_state_abbr");
        String applicable_date = record.getString( name: "applicable_date");
        Double min_temp = record.getDouble( name: "min_temp");
        Double max_temp = record.getDouble( name: "max_temp");
        Double wind_speed = record.getDouble( name: "wind_speed");
        Integer predictability = record.getInt( name: "predictability");

        all_info.add(new WeatherInfo(weather_state_name, weather_state_abbr, applicable_date, min_temp, max_temp, wind_speed, predictability));
    } catch (JSONException e) {
    Log.d( tag: "Json object", record.toString());
    Log.d( tag: "FAIL",  msg: "FAILED TO GET VALUE");
    e.printStackTrace();
}
}

Log.d( tag: "JsonObject",
        list.get(0).toString());



return all_info;
}
```

W głównym wątku aplikacji musimy stworzyć nowy wątek pobierający dane z API, następnie czekamy aż wątek skończy pracę za pomocą metody join. To rozwiązanie nie jest idealnie - podczas pracy wątku przy np słabym połączeniu interfejs aplikacji się zawiesi do momentu skończenia pracy wątku. Alternatywnym wyjściem byłoby zastosowanie AsyncTasków.

```
Thread api_info = new Thread((Runnable) () -> {
        // a potentially time consuming task
        Network net = new Network();

        try {
            ArrayList<WeatherInfo> all_infos = net.getResponseFromHttpUrl(net.buildUrlForWeather());
            Log.d( tag: "APIOUT", all_infos.get(0).getMax_temp().toString());
            Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
            db.insertData(all_infos);
            Log.d( tag: "DB",  msg: "Inserted to database");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
);
api_info.start();
try {
    api_info.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

2.3 Zapis danych pogodowych do bazy danych SQL Lite

W celu przechowywania i późniejszego dostępu do danych które pobraliśmy z API musimy stworzyć bazę danych do której będziemy się odwoływać później. Skorzystamy z wbudowanej funkcjonalności androida jaka jest SQLLite. W tym celu, stosując się do praktyki enkapsulacji kodu, tworzymy kolejna klase Sqllitedatabase dziedziczącą po SQLiteOpenHelper, w ten sam sposób co poprzednio.

Tworzymy pola prywatne plus konstruktor:

```
public class Sqlitedatabase extends SQLiteOpenHelper {
    private String DATABASE_MARKSTABLE = "Weather";
    private String w_name = "weather_state_name";
    private String w_abbr = "weather_state_abbr";
    private String w_ap_date ="applicable_date";
    private String w_min_temp ="min_temp";
    private String w_max_temp ="max_temp";
    private String w_wind_speed ="wind_speed";
    private String w_pred ="predictability";

    public Sqlitedatabase(Context context) { super(context, name: "WeatherDatabase.db", factory: null, version: 1); }
```

Prywatne stringi to nazwy kolumn w DB, odnoszące się do danychz WeatherInfo.

Musimy pamietac o nadpisaniu odpowiednich metod, tzn onCreate i onUpgrade. W metodzie onCreate tworzymy naszą baze danych komendą zgodną ze składnią języka SQL. nazwy kolumn będą odpowiadać zmiennym z klasy WeatherInfo .

```
@Override
public void onCreate(SQLiteDatabase db) {

    db.execSQL(" CREATE TABLE " + DATABASE_MARKSTABLE + " (" +
            w_ap_date + " TEXT PRIMARY KEY, " +
            w_name + " TEXT, " +
            w_abbr + " TEXT, " +
            w_min_temp + " REAL, " +
            w_max_temp + " REAL, " +
            w_wind_speed + " REAL, " +
            w_pred + " INTEGER);"
    );

}
```

Metodę onUpgrade zostawiamy pustą, ponieważ nie będziemy jej wykorzystywać (konieczne natomiast jest jej nadpisanie)

Dodatkowo przydatną metodą będzie insertData w której zapiszemy array obiektów klasy WeatherInfo do naszej bazy danych. W tym celu bierzemy obiekt typu SQLiteDatabase z metody udostępnianej z klasy nadrzędnej - this,getWritableDatabase(). Następnie w pętli for tworzymy obiekt ContentValues (odpowiadający każdemu wierszowi w tabelce sqlowej) w którym zapiszemy wszystkie parametry obiektu WeatherInfo. Za każdym razem wykonujemy metode db.insert(). Na koniec zamykamy bazę danych.

```
public void insertData(ArrayList<WeatherInfo> infos) {
    SQLiteDatabase db = this.getWritableDatabase();



    for (WeatherInfo record : infos){
        ContentValues values = new ContentValues();
        values.put(w_ap_date, record.getApplicable_date());
        values.put(w_name, record.getWeather_state_name());
        values.put(w_abbr, record.getWeather_state_abbr());
        values.put(w_min_temp, record.getMin_temp());
        values.put(w_max_temp, record.getMax_temp());
        values.put(w_wind_speed, record.getWind_speed());
        values.put(w_pred, record.getPredictability());

        // Inserting Row
        db.insert(DATABASE_MARKSTABLE,  nullColumnHack: null, values);
    }

    db.close(); // Closing database connection
}
```

Kolejną przydatną funkcją będzie getAllData(), której zadaniem będzie odczytanie z bazy wszystkich informacji i zwrócenie ich w postaci array'a obiektów Weather info. Aby zrobią tą funkcje trzeba wpierw stworzyć bardzo proste zapytanie SQLowe pozwalające na wyciągnięciw wszystkich rekordów (SELECT * from <nazwa_bazy>). Następnie iterując po wierszach rzutujemy dane do typu WeatherInfo i dodajemy kolejno do Array'a. Całość zwracamy.

```
public ArrayList<WeatherInfo> getAllData() {
    ArrayList<WeatherInfo> contactList = new ArrayList<~>();
    // Select All Query
    String selectQuery = "SELECT  * FROM " + DATABASE_MARKSTABLE;

    Log.d( tag: "DBQ", selectQuery);

    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery,  selectionArgs: null);

    // looping through all rows and adding to list
    if (cursor.moveToFirst()) {
        do {
            WeatherInfo info = new WeatherInfo();
            info.setApplicable_date(cursor.getString(cursor.getColumnIndex(w_ap_date)));
            info.setWeather_state_name(cursor.getString( i: 1));
            info.setWeather_state_abbr(cursor.getString( i: 2));
            info.setMin_temp(cursor.getDouble( i: 3));
            info.setMax_temp(cursor.getDouble( i: 4));
            info.setWind_speed(cursor.getDouble( i: 5));
            info.setPredictability(cursor.getInt( i: 6));
            // Adding record to list
            contactList.add(info);
        } while (cursor.moveToNext());
    }


    // return contact list
    return contactList;
}
```

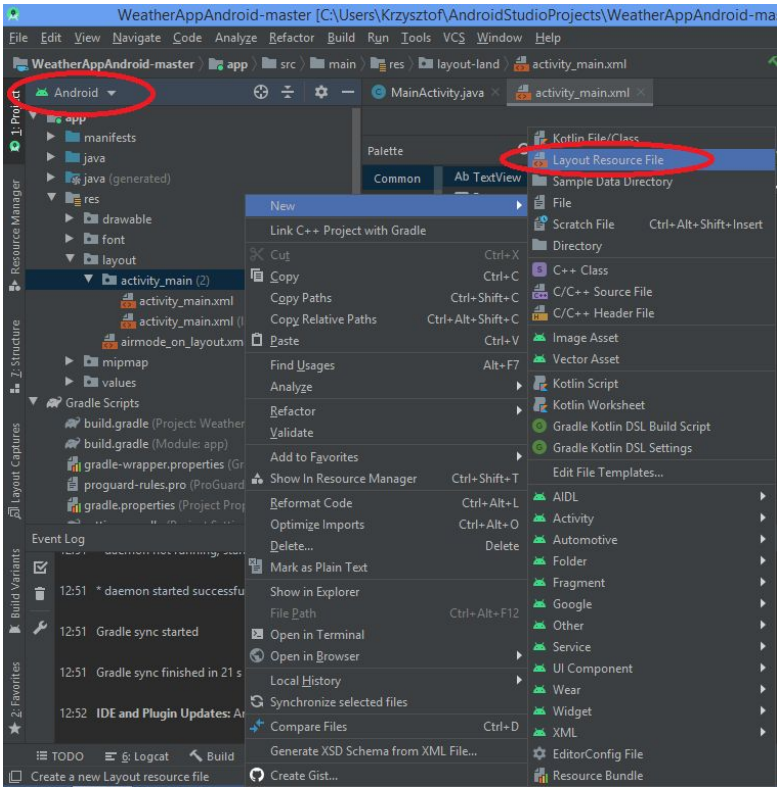
Zapis do bazy danych wykonujemy w wątku pobierającym dane z API, a odczytywanie danych - tam gdzie to konieczne.

```
Thread api_info = new Thread((Runnable) () -> {
        // a potentially time consuming task
        Network net = new Network();

        try {
            ArrayList<WeatherInfo> all_infos = net.getResponseFromHttpUrl(net.buildUrlForWeather());
            Log.d( tag: "APIOUT", all_infos.get(0).getMax_temp().toString());
            Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
            db.insertData(all_infos);
            Log.d( tag: "DB",  msg: "Inserted to database");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
);
api_info.start();
try {
    api_info.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

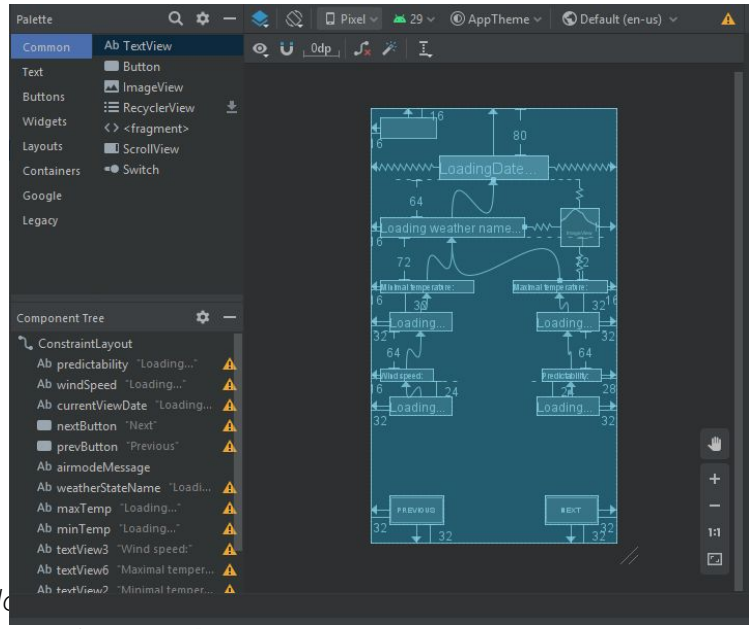
2.4 Tworzenie głównego layout'u

Po poprawnym utworzeniu logiki biznesowej naszej aplikacji możemy przejść do następnego elementu - stworzenie ładnego i funkcjonalnego GUI. Nasz główny layout znajdziemy w katalogu app/res/layout pod nazwą activity_main.xml. Jeżeli go tam nie ma, to utwórz nowy klikając prawym przyciskiem na katalog res -> new -> Layout resource file.

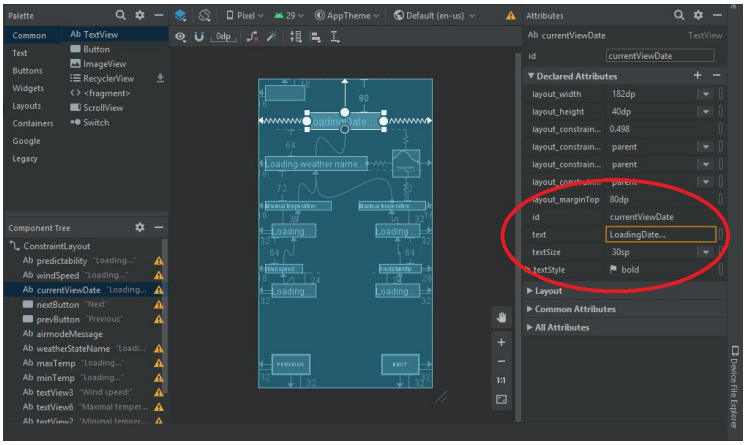


Następnie należy utworzyć (najłatwiej w widoku 'design') następujący, lub podobny układ naszej głównej aktywności. Prawie wszystkie elementy aktywności będą typu TextView. Są jednak trzy wyjątki:

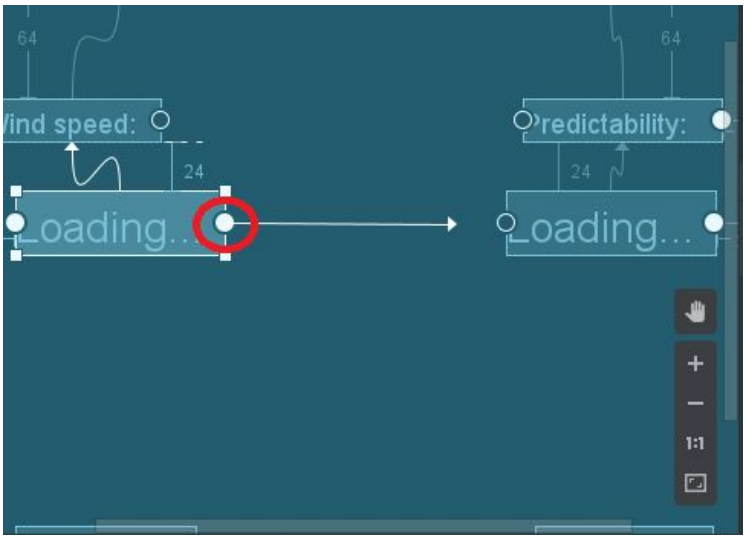
- miejsce przeznaczone na ikonkę pogody będzie typu ImageView
- Przyciski previous i next będą typu button



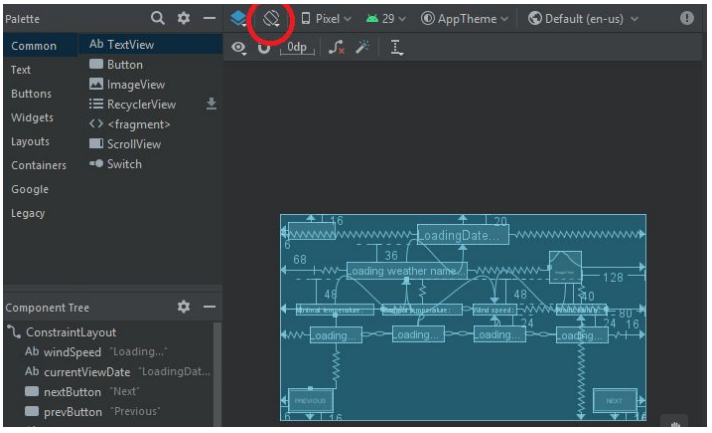
Przy kliknięciu na dowolny element aktywności po prawej stronie możemy zauważyć atrybuty danego elementu. Ustaw przejrzyste id dla elementów, których text będzie się zmieniać (miejsce danych na temat temperatury, daty, przewidywalności itp.). W tym samym menu możesz także ustawić domyślnie wyświetlany tekst.



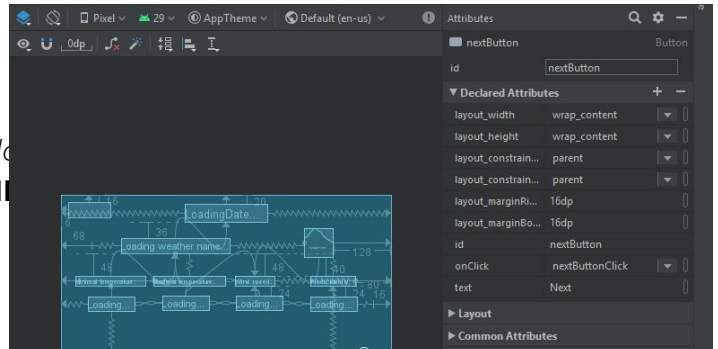
Każdemu z elementów ustaw ograniczenia (constrainst) z każdej strony. Służą one poprawnemu wyświetlaniu aktywności na różnych urządzeniach. Możesz posłużyć się wartościami zawartymi w poprzednich obrazkach, lub wykonać swój własny pomysł.



Aby utworzyć widok horyzontalny należy kliknąć zaznaczoną na obrazku ikonkę, a następnie wybrać “Create Horizontal Variation”. Następnie można ustalić nowy układ aktywności, lub wykorzystać zaprezentowany obok.



Zanim przejdziemy do następnego etapu, zaznacz jeszcze atrybut



Krzysztof Pal
AGH WEAIiI

onClick w przyciskach next oraz previous. Jako wartość ustaw metody MainActivity nextButtonClik oraz previousButtonClick, które będziemy implementować w następnej sekcji.

2.4 Komunikacja frontend-backend

Dzięki stworzeniu logiki aplikacji możemy pobrać dane pogodowe z meta weather API i zapisać je w bazie danych. Stworzenie głównej aktywności pozwoliło nam na wyodrębnienie miejsca służącego do prezentacji tych informacji. Przekazanie informacji z modelu do widoku będzie realizowane za pomocą klasy MainActivity. Znajdziemy ją w katalogu app/java/com/example/weatherapp. Jeżeli jej tam nie ma, utwórz ją klikając praawy przycisk myszy na katalogu i wybierając 'Java class'.

```
public class MainActivity extends AppCompatActivity {
    private Integer current_position = 0;
    private BroadcastReceiver receiver;
    private boolean airmode_status;

    @Override
    protected void onCreate(Bundle savedInstanceState)
```

Klasa typu AppCompatActivity musi zawierać metodę protected void onCreate. Jest to metoda, która wykonuje się jako pierwsza przy uruchomieniu aktywności. W naszej aplikacji jej zadaniem będzie: ustawienie stworzonego layout'u jako GUI naszej aktywności, pobranie danych z API, stworzenie bazy danych zawierające pobrane informacje oraz ustawienie zawartości TextView na wartości pobrane z bazy danych. Pierwszy punkt można zawrzeć w dwóch linijkach:

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
```

Wywołanie metody super.onCreate(savedInstanceState) powoduje wczytanie poprzedniego stanu aktywności w przypadku, gdy była już wcześniej włączana. Przyda się to w następnym podpunkcie - obsłudze trybu samolotowego - przy braku dostępu do internetu aplikacja będzie wyświetlać informacje pobrane przy poprzednim włączeniu aplikacji.

Następnie zajmiemy się przygotowaniem danych potrzebnych w aplikacji. Zrobimy to używając wcześniej stworzonych elementów programu: klas Network oraz klasy Sqlitedatabase. Instrukcje zawrzemy w metodzie run stworzonego specjalnie do tego celu wątku.

```
Thread api_info = new Thread(new Runnable() {
    public void run() {
        // a potentially time consuming task
        Network net = new Network();

        try {
            ArrayList<WeatherInfo> all_infos = net.getResponseFromHttpUrl(net.buildUrlForWeather());
```

Rozpoczniemy od utworzenia obiektu klasy network. Następnie w bloku try (ze względu na podatność kodu na wyjątki) stworzymy obiekt ArrayList<WeatherInfo> dzięki metodzie Network.getResponseFromHttpUrl(net.buildUrlForWeather). Następnie utworzony zostanie obiekt klasy Sqllitedatabase, która stworzy bazę danych i za pomocą metody Sqlitedatabase.insertData uzupełni ją o pobrane informacje z API.
Następnie za pomocą funkcji Thread.start oraz Thread.join uruchomimy wątek i sprawdzimy, czy miał pomyślny przebieg.

```
            Network net = new Network();

            try {
                ArrayList<WeatherInfo> all_infos = net.getResponseFromHttpUrl(net.buildUrlForWeather());
                Log.d( tag: "APIOUT", all_infos.get(0).getMax_temp().toString());
                Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
                db.insertData(all_infos);
                Log.d( tag: "DB",  msg: "Inserted to database");
            } catch (IOException e) {
                e.printStackTrace();
            }
        };
        }
);
api_info.start();
try {
    api_info.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Jeżeli wszystko pójdzie pomyślnie, będziemy mogli wykorzystać metodę Sqlitedatabase.getAllData i przekazać odpowiedni rekord do widoku naszej aplikacji. Zrobimy to jednak w osobnej funkcji - updateCurrentView, do której w tworzonej obecnie metodzie przekażemy pierwszy rekord zapisanych informacji - czyli dane dotyczące dzisiejszej pogody.

```
        // String temperature = net.doInBackground();

        Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
        ArrayList<WeatherInfo> recieved_infos = db.getAllData();
        Log.d( tag: "DB",  msg: "Read from database " + recieved_infos);

        this.updateCurrentView(recieved_infos.get(0));
```

Metoda upadteCurrentView będzie jednak prosta - będzie tworzyć obiekty TextView powiązane z polami w naszym layoucie posługując się nadanymi wcześniej identyfikatorami. Następnie za pomocą metody setText będzie ustawiać treść pól w widoku na pozyskane wcześniej dane. W przypadku ustawienia ikonki na odpowiedni obrazek, będziemy używać metody ImageView.setImageResource do której przekażemy id jednego z wcześniej zapisanych obrazków za pomocą lokalnej metody getResId, która wymaga od nas jedynie skrótu opisu pogody, który także pobraliśmy z bazy danych.

```
    void updateCurrentView(WeatherInfo currentWeather){
        try{
            TextView currentViewDate = findViewById(R.id.currentViewDate);
            currentViewDate.setText(currentWeather.getApplicable_date());

            TextView minTemp = findViewById(R.id.minTemp);
            minTemp.setText(String.format("%.2f", currentWeather.getMin_temp()));

            TextView maxTemp = findViewById(R.id.maxTemp);
            maxTemp.setText(String.format("%.2f", currentWeather.getMax_temp()));

            TextView windSpeed = findViewById(R.id.windSpeed);
            windSpeed.setText(String.format("%.2f", currentWeather.getWind_speed()) + " km/h");

            TextView predictability = findViewById(R.id.predictability);
            predictability.setText(String.valueOf(currentWeather.getPredictability() + "%"));

            TextView weatherState = findViewById(R.id.weatherStateName);
            weatherState.setText(currentWeather.getWeather_state_name());

            ImageView img= (ImageView) findViewById(R.id.icon);
            int resId = getResId(currentWeather.getWeather_state_abbr());

            // W przypadku braku opisu pogody/innemu opisowi ikonka pozostaje bez zmian.
            if(resId != -1){img.setImageResource(resId);};

        }catch (IndexOutOfBoundsException e){
            Log.e( tag: "ERR",  msg: "Could not find date in recieved infos");
        }
    }
```

Metoda getResId nie będzie robić nic innego, jak zwrócenie odpowiedniego id w zależności od przekazanego skrótu pogody. W przypadku, gdy skrót się nie zgadza lub ma wartość null powinna zwracać -1.

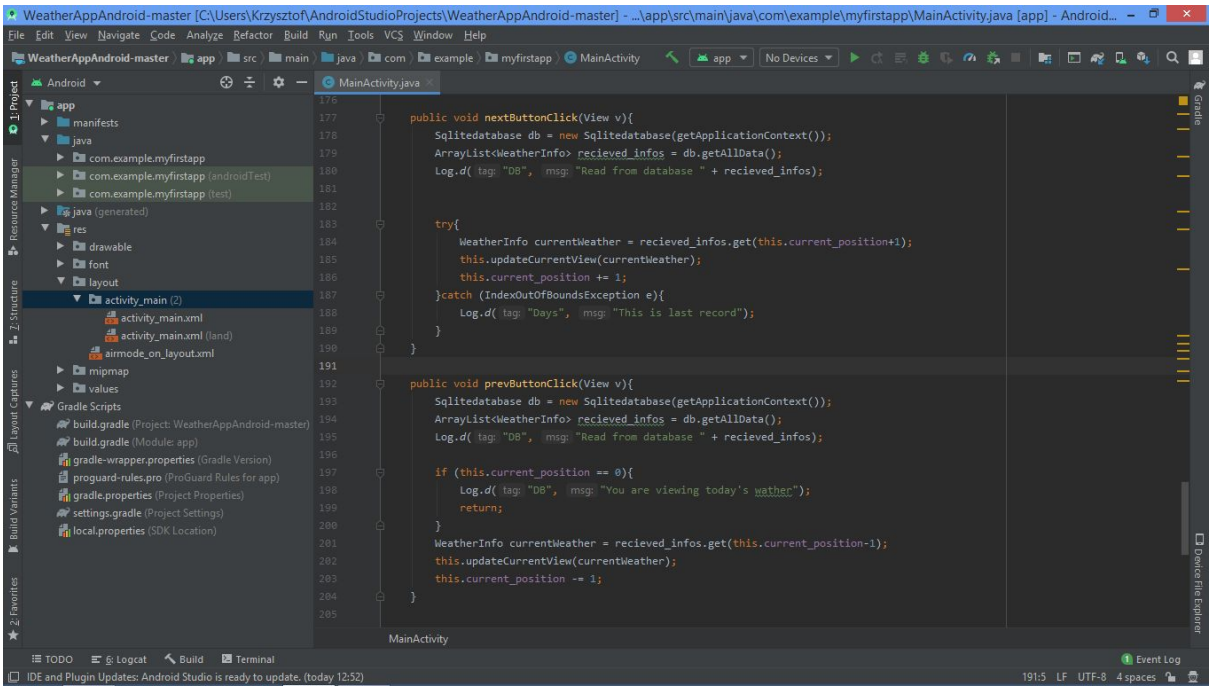
```
private int getResId(String weather_state_abbr) {
    if(weather_state_abbr.equals("s")){
        return R.drawable.ic_s;
    }
    if(weather_state_abbr.equals("c")){
        return R.drawable.ic_c;
    }
    if(weather_state_abbr.equals("h")){
        return R.drawable.ic_h;
    }
    if(weather_state_abbr.equals("hc")){
        return R.drawable.ic_hc;
    }
    if(weather_state_abbr.equals("hr")){
        return R.drawable.ic_hr;
    }
    if(weather_state_abbr.equals("lc")){
        return R.drawable.ic_lc;
    }
    if(weather_state_abbr.equals("lr")){
        return R.drawable.ic_lr;

    }if(weather_state_abbr.equals("sl")){
        return R.drawable.ic_sl;

    }if(weather_state_abbr.equals("sn")){
        return R.drawable.ic_sn;

    }if(weather_state_abbr.equals("t")){
```

Następnie musimy zaimplementować działania, które prezentuje aplikacja w przypadku naciśnięcia przycisku previous i next. Jak nietrudno się domyśleć będą one zmieniały zmienną currentposition o + lub - 1 i wywoływały metodę updateCurrentView dla odpowiedniego rekordu w bazie danych oznaczającego wybrany dzień.


WeatherAppAndroid-master [C:\Users\Krzysztof\AndroidStudioProjects\WeatherAppAndroid-master] - ...\app\src\main\java\com\example\myfirstapp\MainActivity.java [app] - Android...
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MainActivity.java
public void nextButtonClick(View v){
Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
ArrayList<WeatherInfo> recieved_infos = db.getAllData();
Log.d(tag: "DB", msg: "Read from database " + recieved_infos);
try{
WeatherInfo currentWeather = recieved_infos.get(this.current_position+1);
this.updateCurrentView(currentWeather);
this.current_position += 1;
}catch (IndexOutOfBoundsException e){
Log.d(tag: "Days", msg: "This is last record");
}
}
public void prevButtonClick(View v){
Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
ArrayList<WeatherInfo> recieved_infos = db.getAllData();
Log.d(tag: "DB", msg: "Read from database " + recieved_infos);
if (this.current_position == 0){
Log.d(tag: "DB", msg: "You are viewing today's wather");
return;
}
WeatherInfo currentWeather = recieved_infos.get(this.current_position-1);
this.updateCurrentView(currentWeather);
this.current_position -= 1;
}
MainActivity
TODO 6: Logcat Build Terminal
Event Log
IDE and Plugin Updates: Android Studio is ready to update. (today 12:52)
191:5 LF UTF-8 4 spaces

2.5 Obsługa trybu samolotowego

Aby zmienić tryb pracy aplikacji na korzystający tylko z już istniejącej bazy danych (bez komunikacji z internetem) musimy stworzyć tzw BroadcastReciever, który będzie wyczulony za pomocą odpowiedniego filtra sygnałów na ACTION_AIRPLANE_MODE_CHANGED. Tym sygnałem system notyfikuje naszą aplikację, że użytkownik włączył tryb samolotowy.

```
IntentFilter filter = new IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED);

BroadcastReceiver receiver = (context, intent) -> {
        if (intent.getAction().intern() == Intent.ACTION_AIRPLANE_MODE_CHANGED) {
            Log.d( tag: "AirplaneMode",  msg: "Service state changed");
            boolean airplane_active = isAirplaneModeOn(context);
            Log.d( tag: "AIRMODE",  msg: "Airplane mode is " + airplane_active);
        }
};

registerReceiver(receiver, filter);
```

Następnie w kodzie aplikacji musimy skorzystać z informacji którą otrzymaliśmy. Ważnym jest to że wiadomość systemowa informuje nas tylko że stan trybu samolotowego się zmienił, nie mówiąc z jakiego na jaki. Zatem pierw musimy sprawdzić czy tryb samolotowy jest włączony czy nie. Tworzymy zatem w MainActivity metodę sprawdzającą ten fakt.

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)
public static boolean isAirplaneModeOn(Context context) {
    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN_MR1) {
        return Settings.System.getInt(context.getContentResolver(),
                Settings.System.AIRPLANE_MODE_ON,  def: 0) != 0;
    } else {
        return Settings.Global.getInt(context.getContentResolver(),
                Settings.Global.AIRPLANE_MODE_ON,  def: 0) != 0;
    }
}
```

Następnie jak już wiemy czy tryb samolotowy jest włączony czy wyłączony to podejmujemy stosowną akcję.

```
if (this.airmode_status){
    Log.d( tag: "AIRMODE",  msg: "Airmode is on");

    TextView airmodeMessage = findViewById(R.id.airmodeMessage);
    airmodeMessage.setText("AirMode ON");
    try{
        Sqlitedatabase db = new Sqlitedatabase(getApplicationContext());
        ArrayList<WeatherInfo> recieved_infos = db.getAllData();
        Log.d( tag: "DB",  msg: "Read from database " + recieved_infos);

        this.updateCurrentView(recieved_infos.get(0));

    }catch (IndexOutOfBoundsException e){
        Log.e( tag: "DB",  msg: "Cant get info from DB - AIRMODE ON");
        Intent goToNextActivity = new Intent(getApplicationContext(), AirmodeOnScreen.class);
        startActivity(goToNextActivity);

    }
    return;
}
```

Nasz algorytm działania będzie wyglądał następująco:
jeśli jest włączony tryb samolotowy ustawiamy w widoku aplikacji tekst “AIR MODE ON” żeby powiadomić użytkownika o tym fakcie (dane mogą nie być aktualne). Jeśli tryb samolotowy jest wyłączony to wszystko działa bez modyfikacji. Naszym corner case jest sytuacja kiedy użytkownik pierwszy raz włącza aplikacje w trybie samolotowym, W tym przypadku nie mamy już istniejącej bazy danych z rekordami pogodowymi. Zatem przekierowywujemy użytkownika na nowy ekran w którym wyświetlamy stosowną wiadomość.