

JĘZYKI PROGRAMOWANIA WYSOKIEGO POZIOMU

temat projektu: aplikacja pogodowa na androida

Sebastian Kulig, Karol Wnęk

April 2020

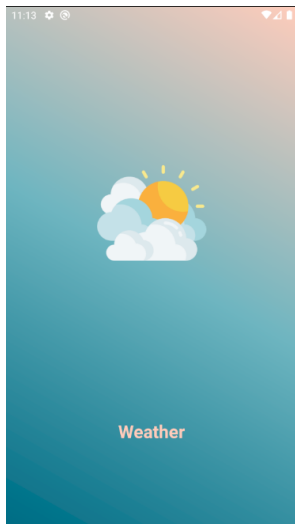
Plan prezentacji

- Krótkie omówienie naszej aplikacji
- API
- JSON
- Wykorzystanie biblioteki retrofit do interakcji z API
- Fragmenty

Krótkie omówienie naszej aplikacji

Naszym projektem było stowrzenie programu pobierającego dane z API. Zrealizowaliśmy to w formie aplikacji pogodowej na Androida, działającej na urządzeniach od wersji 7.0 Nougat. Prognozę pogody uzyskaliśmy z Open Weather API za pomocą biblioteki Retrofit. Do wyświetlenia otrzymanych danych wykorzystaliśmy fragmenty.

Krótkie omówienie naszej aplikacji

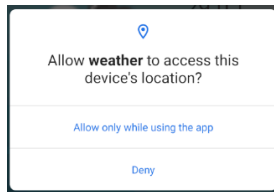


Po uruchomieniu aplikacji użytkownik witany jest przez *splash screen*, w trakcie którego wczytywane są dane zapisane w pamięci urządzenia.

Krótkie omówienie naszej aplikacji



Następnie użytkownik przenoszony jest do ekranu głównego. Jeżeli jest to pierwsze uruchomienie aplikacji zostanie poproszony o zgodę na użycie lokalizacji.



Krótkie omówienie naszej aplikacji

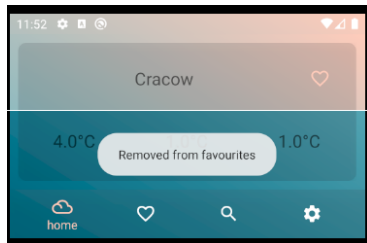
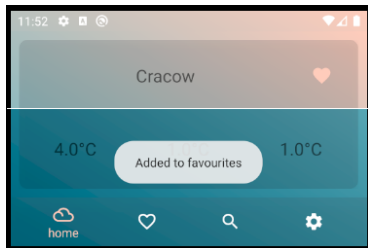
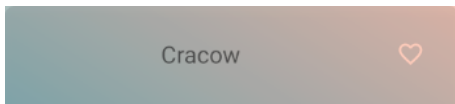


Do poruszania się po aplikacji służy dolny pasek nawigacyjny i tak są to kolejno:

- ekran domowy
- ulubione
- wyszukiwanie lokalizacji
- about

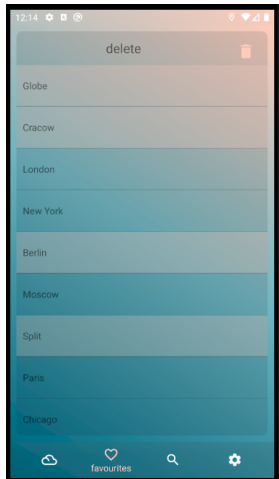
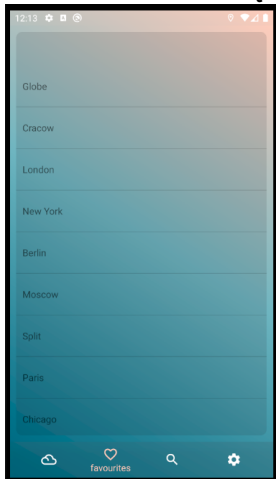
Krótkie omówienie naszej aplikacji

Ikona serduszka służy do dodawania i usuwania z ulubionych, każdej takiej operacji towarzyszy stosowny komunikat.

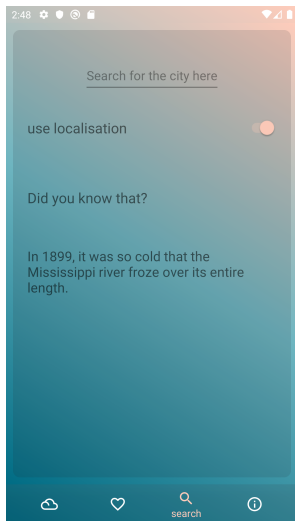


Krótkie omówienie tego co udało się nam zrobić

Naciśnięcie na dany element listy przenosi użytkownika do ekranu głównego i wyświetla pogodę dla wybranej lokalizacji, dłuższe przytrzymanie pozwala na zaznaczenie w celu usunięcia z ulubionych.



Krótkie omówienie tego co udało się nam zrobić

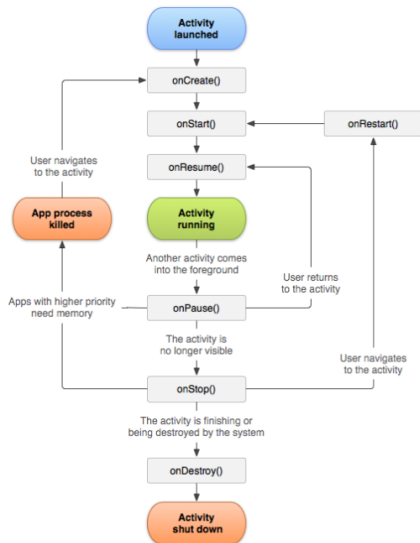


Tutaj możemy wyszukać lokalizację ręcznie oraz włączyć i wyłączyć geolokalizację. Ciekawostki są losowane z puli zapisanej na urządzeniu.

Jest to podstawowy komponent aplikacji. Jest niejako odpowiednikiem metody `main()`. Różnica polega na tym, że system nie wykonuje jej całej, a jedynie poszczególne fragmenty w odpowiednich momentach jej życia.

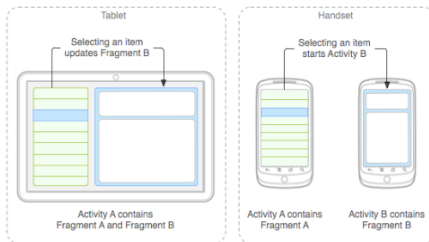
```
public class MainActivity extends Activity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
}
```

Aktywność, cykl życia

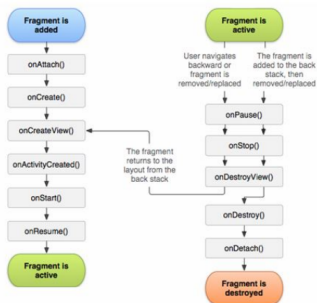


Fragmenty - czym są, jak z nich korzystać

Fragment to, najprościej ujmując, część interfejsu aplikacji posiadająca własne zachowania. Każdy fragment musi być zagnieżdżony w jakiejś aktywności, przy czym jedna taka aktywność może posiadać wiele fragmentów, które są od siebie wzajemnie niezależne. Dzięki fragmentom możemy tworzyć modularne interfejsy, które będą poprawnie działały, niezależnie od urządzenia, na którym jest uruchomiona aplikacja. Zostały wprowadzone w Androidzie 3.0 Honeycomb, API level 11 (24.01.2011).



Cykl życia fragmentu



Fragment posiada własny cykl życia, którym podąża dopóki istnieje aktywność, do której jest dowiązany.

Podstawowe metody, które powinniśmy implementować:

- `onCreate();`
- `onCreateView();`
- `onPause();`

onCreate(), onCreateView(), onPause()

- onCreate() - system wywołuje tą metodę podczas tworzenia fragmentu, tutaj powinniśmy inicjalizować elementy które chcemy zachować kiedy fragment zostaje zapauzowany lub zatrzymany.
- onCreateView() - niejako najważniejsza metoda, bowiem odpowiada ona za narysowanie fragmentu.
- onPause() - wywoływana przez system przy pierwszym podejrzeniu, że użytkownik opuszcza fragment.

hello fragment! - onCreateView()

Najprostsza implementacja metody onCreateView() może wyglądać następująco:

```
12 public class HelloFragmentA extends Fragment {  
13  
14     @Nullable  
15     @Override  
16     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {  
17         View view = inflater.inflate(R.layout.hello_fragment_a, container, attachToRoot: false);  
18         return view;  
19     }  
20 }  
21
```

- *LayoutInflater inflater* to obiekt, który umożliwi utworzenie wyglądu naszego fragmentu.
- *ViewGroup container* jest to widok rodzica, do którego będzie dołączony nasz fragment.
- *Bundle savedInstanceState* pozwala na odtworzenie poprzedniego stanu fragmentu, np. wartości zmiennych przez niego używanych

hello fragment! - onCreateView()

```
12 public class HelloFragmentA extends Fragment {  
13  
14     @Nullable  
15     @Override  
16     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {  
17         View view = inflater.inflate(R.layout.hello_fragment_a, container, attachToRoot false);  
18         return view;  
19     }  
20 }  
21
```

Pod wyrażeniem `R.layout.hello_fragment_a` kryje się plik xml opisujący docelowy wygląd tworzonego fragmentu.

Parametr `attachToRoot: false` mówi czy widok, który tworzymy ma być dołączony do rodzica - container. Jako iż podpinamy go do naszego fragmentu przekazujemy *false*.

hello fragment! - layout.xml

Jak już wcześniej było wspomniane, wygląd naszego fragmentu opisuje plik *hello_fragment_a.xml*

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:background="#6EB5C0">
7
8      <TextView
9          android:layout_width="wrap_content"
10         android:layout_height="wrap_content"
11         android:layout_centerInParent="true"
12         android:textSize="25sp"
13         android:text="hello fragment A!">
14     </TextView>
15 </RelativeLayout>
```



hello fragment A!

hello fragment! - MainActivity

Jako iż nasz prosty fragment jest gotowy, przyszedł czas na aktywność, która powoła go do życia.

```
21
22 public class MainActivity extends AppCompatActivity {
23
24     private HelloFragmentA fragmentA;
25
26     @Override
27     protected void onCreate(Bundle savedInstanceState) {
28         super.onCreate(savedInstanceState);
29         setContentView(R.layout.activity_main);
30
31         if (fragmentA == null) {
32             fragmentA = new HelloFragmentA();
33         }
34
35         FragmentManager fragmentManager = getSupportFragmentManager();
36         FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
37         fragmentTransaction.replace(R.id.fragment_container, fragmentA).commit();
38     }
39 }
```

hello fragment! - MainActivity

- wywołanie `setContentView(R.layout.activity_main)`; powoduje, że nasza aktywność będzie miała taki układ jak ten zdefiniowany w pliku `activity_main.xml`
- `getSupportFragmentManager()` zwraca obiekt typu `FragmentManager` służący do interakcji z fragmentami.
- `.beginTransaction()` otwiera nam drogę do modyfikowania menadżera fragmentów.
- `.replace()` zamienia obecny fragment podpięty do aktywności w miejscu wskazanym przez `R.id.fragment_container` na fragmentA
- `.commit()` wprowadza zmiany w życie.

komunikacja między fragmentami

Jako że fragmenty powinny być niezależne od siebie, nie zaleca się bezpośredniej komunikacji ze sobą. Mogą to jednak robić za pomocą pośrednika tj. aktywności. Podstawą takiej wymiany danych jest interfejs, najczęściej deklarowany wewnątrz fragmentu (nested interface).

```
14
15 public interface HelloFragmentAListener{
16     void sendData(String text);
17 }
18
```

Nasz fragment musi również posiadać instancję tego interfejsu:

```
19 private HelloFragmentAListener helloFragmentAListener;
```

komunikacja między fragmentami

Kolejnym krokiem jest upewnienie się, że ktoś implementuje interfejs *HelloFragmentAListener*. W tym celu należy skorzystać z metody `onAttach()`.

```
21      @Override
22      public void onAttach(@NonNull Context context) {
23          super.onAttach(context);
24          if (context instanceof HelloFragmentAListener) {
25              helloFragmentAListener = (HelloFragmentAListener) context;
26          } else {
27              throw new RuntimeException(context.toString()
28                  + " must implement HelloFragmentAListener");
29          }
30      }
```

Context context to nic innego jak nasza aktywność, która implementuje interfejs.

komunikacja między fragmentami

Warto również zadbać o to aby na końcu życia fragmentu zwolnić *helloFragmentAListener* - oszczędność zasobów. W tym celu możemy użyć metody `onDetach()`.

```
46      @Override
47      public void onDetach() {
48          super.onDetach();
49          helloFragmentAListener = null;
50      }
51  }
52
```

komunikacja między fragmentami

Teraz czas na najważniejsze, czyli wykorzystanie naszego *helloFragmentAListener*

```
38
39      @Override
40      public void onPause() {
41          super.onPause();
42          helloFragmentAListener.sendData( text: "new text");
43      }
44
```

Oczywiście można to zrobić dla dowolnego zdarzenia np. naciśnięcie przycisku, obrót urządzenia, czy zamknięciu aplikacji.

MainActivity:

```
62      @Override
63      public void sendData(String text) {
64          fragmentB.updateData(text);
65      }
66  }
```


komunikacja między fragmentami

Ostatnim krokiem jest *fragmentB*:

```
12
13 public class HelloFragmentB extends Fragment {
14
15     private TextView textView;
16     private String string;
17
18     @Nullable
19     @Override
20     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
21         View view = inflater.inflate(R.layout.hello_fragment_b, container, attachToRoot: false);
22         textView = view.findViewById(R.id.textB);
23
24         if (string != null && !string.isEmpty()) {
25             textView.setText(string);
26         }
27         return view;
28     }
29
30     void updateData(String text) {
31         if (textView != null){
32             string = text;
33         }
34     }
35 }
~~
```

API(Application Programming Interface) jest to zbiór reguł i rozwiązań, dzięki którym aplikacja może się komunikować z inną aplikacją, bazą danych, siecią czy systemem operacyjnym. Jednym z typów API są Web APIs, które udostępniają pewne funkcjonalności jako zasób w sieci. Używają one protokołu http do wysyłania tzw. Request messages. Odpowiedź (response message) jest dostarczana w formacie XML lub JSON (które omówimy w następnym punkcie).

Wykształcił się pewien styl architektury zwany REST (Representational State Transfer) zaprezentowany przez Roy Fieldinga w 2000 roku. Aby dane API można było nazwać RESTful musi spełniać następujące założenia:

- Client-server architecture - odseparowanie interfejsu użytkownika od operacji na serwerze
- Statelessness – pomiędzy zapytaniami (requests) na serwerze nie mogą być przechowywane żadne dane użytkownika (informacje o sesji przechowuje klient, więc każde zapytanie musi zawierać komplet informacji)
- Cacheability - odpowiedzi mogą być przechowywane.

- Layered system - warstwowa struktura sprawia, że klient nie musi wiedzieć czy komunikuje się z jakąś warstwą pośrednią
- Code on demand (optional) - udostępnianie dodatkowych funkcjonalności np. kiedy wiemy że użytkownicy będą wykonywać konkretne operacje możemy im dostarczyć gotowe rozwiązanie.
- Uniform Interface (jednolity interfejs)
 - 1 Każdy zasób jest dostępnym pod unikalnym adresem, jednak sposób jego reprezentacji jest niezależny od zasobu (np od schematu bazy danych na której jest przechowywany)
 - 2 Manipulacja zasobami jest realizowana poprzez jego reprezentację, która musi zawierać wystarczającą ilość informacji do jej modyfikacji
 - 3 Komunikaty serwera do klienta są samoopisujące, dostarczają informacji jak je przetworzyć
 - 4 Hypermedia as the Engine of Application State – klient przesyła w żądaniu swój stan, parametry, nagłówki oraz URI zasobu. Serwer dostarcza treść, kod i nagłówki odpowiedzi. Ta forma komunikacji jest określana jako „hypermedia”. Oprócz tego HATEOAS oznacza również, że linki zawarte są w treści odpowiedzi lub nagłówkach.

Przykład API

W naszym projekcie użyliśmy OpenWeatherMap API, które udostępnia prognozę pogody dla wybranego miejsca na świecie. Dla przykładu, aby otrzymać prognozę dla danego miasta, wystarczy do takiego API wysłać następujące żądanie (request) HTTP:

Examples

```
api.openweathermap.org/data/2.5/weather?q={cityName}&appid={yourApiKey}
```

gdzie:

- api.openweathermap – to bazowy adres URL
- /data/2.5/weather – to tzw. „endpoint” wskazuje do jakiego zasobu chcemy się odwołać
- ? – wskazuje na początek query string (łańcucha zapytań). Po nim umieszczamy wszystkie parametry, w tym przypadku nazwę miasta oraz klucz dostępu do usług OpenWeatherMap. Parametry oddzielamy znakiem &.

JSON (JavaScript Object Notation) jest prostym i popularnym formatem wymiany danych. Wywodzi się on z języka JavaScript, ale jest całkowicie niezależny od języków programowania. W tym właśnie formacie zwracać będzie odpowiedź wykorzystywane przez nas API.

Wspierane typy danych:

- obiekt - para nazwa/wartość. Elementy tego typu składają się z nawiasów { } i umieszczonych w nich par "nazwa atrybutu":<wartość atrybutu>. Kolejne pary rozdzielone są przecinkami.

Examples

```
{  
  "id": 800,  
  "main": "Clear",  
  "description": "clear sky",  
  "icon": "01n"  
}
```

- Uporządkowana lista implementowana np jako tablica, lista lub wektor składająca się z nawiasów [] oraz umieszczonych w nich wartości rozdzielonych przecinkami. Wartości umieszczone w liście mogą mieć różne typy

Examples

```
{  
  "cars": [ "Ford", "BMW", "Fiat" ]  
}
```

Jak widać struktury te można zagnieżdżać.

- łańcuch znaków - (np. "lorem ipsum dolor sit amet") \ (backslash) pozwala na wprowadzenie znaków specjalnych
- liczby (dopuszczalna jest notacja naukowa oraz znak minusa -, nie używa się formatów ósemkowych i szesnastkowych)
- wartości: true/false/null
- białe znaki są ignorowane

Mimo iż specyfikacja formatu JSON nie narzuca w żaden sposób typu elementu głównego (w którym znajdują się wszystkie pozostałe), to jednak w tej roli najczęściej występuje obiekt.

A tak wygląda odpowiedź którą zwróciłoby przedstawione wcześniej zapytanie do OpenWeatherMap API, gdybyśmy jako miasto wybrali Londyn.

```
{
  "coord": {
    "lon": -0.13,
    "lat": 51.51
  },
  "weather": [
    {
      "id": 300,
      "main": "Drizzle",
      "description": "light intensity drizzle",
      "icon": "09d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 280.32,
    "pressure": 1012,
    "humidity": 81,
    "temp_min": 279.15,
    "temp_max": 281.15
  },
  "visibility": 10000,
  "wind": {
    "speed": 4.1,
    "deg": 80
  },
  "clouds": {
    "all": 90
  },
  "dt": 1485789600,
  "sys": {
    "type": 1,
    "id": 5091,
    "message": 0.0103,
    "country": "GB",
    "sunrise": 1485762037,
    "sunset": 1485794875
  },
  "id": 2643743,
  "name": "London",
  "cod": 200
}
```

Wykorzystanie retrofit do interakcji z API

Retrofit jest klientem HTTP, który zamienia HTTP API na interfejs Javy. Najprościej ujmując jest to biblioteka pozwalająca na łatwiejsze pobieranie i przesyłanie danych za pośrednictwem serwisów internetowych opartych na REST z wykorzystanie biblioteki OkHttp.

Zanim rozpoczniemy pracę z biblioteką retrofit i spróbujemy pobrać dane z API, będziemy musieli:

- dodać pozwolenie na korzystanie z sieci w pliku AndroidManifest.xml

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

- dodać potrzebne biblioteki uzupełniając dependencies w pliku build.gradle

```
//retrofit2
implementation 'com.squareup.retrofit2:retrofit:2.7.2'
//Gson converter
implementation 'com.squareup.retrofit2:converter-gson:2.7.2'

//Picasso
implementation 'com.squareup.picasso:picasso:2.71828'
```

Do wykorzystania bilbiloteki będziemy potrzebowali kilku elementów:

- Model-POJO (Plain Old Java Object) - po ściągnięciu odpowiedzi z serwera w postaci JSON, będzie ona musiała być skonwertowana do POJO. Są to klasy modelu reprezentujące odpowiedź. Gson konwertuje takie obiekty do i z formatu JSON. Wartości z odpowiedzi zostają przypisane tylko do właściwości, które są zgodne z formatem odpowiedzi. Jeśli więc umieścimy w naszym modelu pola nadmiarowe lub zapomnimy o jakimś polu to w obu przypadkach zostaną one zignorowane. Do jego stworzenia można użyć różnych narzędzi, my wykorzystamy <http://www.jsonschema2pojo.org>. Po wklejeniu na stronę naszej odpowiedzi z serwera i wybraniu odpowiednich opcji otrzymamy wygenerowane przykładowe klasy naszego modelu.

```
public class WeatherResponse {  
    private String base;  
    private Main main;  
    private Integer dt;  
    private Integer id;  
    @SerializedName("name")  
    @Expose(serialize = false, deserialize = true)  
    private String fullName;  
    private Integer cod;  
  
    public String getBase() { return base; }  
    public void setBase(String base) { this.base = base; }  
    public Main getMain() { return main; }  
    public void setMain(Main main) { this.main = main; }  
    public Integer getDt() { return dt; }  
    public void setDt(Integer dt) { this.dt = dt; }  
    public Integer getId() { return id; }  
    public void setId(Integer id) { this.id = id; }  
    public String getName() { return fullName; }  
    public void setFullName(String name) { this.fullName = name; }  
    public Integer getCod() { return cod; }  
    public void setCod(Integer cod) { this.cod = cod; }  
}
```

Tak wygląda przykładowy model, dla przedstawionego wcześniej zapytania.

Jeśli chcemy aby nasze zmienne w programie nazywał się inaczej niż te wyspecyfikowane w obiekcie JSON możemy użyć:

`@SerializedName("name from JSON")`

Dzięki temu Gson będzie mógł dokonać prawidłowego mapowania.

Jeśli chcemy, aby tylko wybrane pola w naszym modelu były serializowane/deserializowane, możemy posłużyć się adnotacją:

`@Expose(serialize=T/F,deserialize=T/F)`

- Interfejs zapytań do API - zdefiniujemy w nim metody odwołujące się do zasobu sieciowego oraz endpoint. Retrofit generuje implementacje takiego interfejsu. Za pomocą adnotacji deklarujemy metody HTTP (mamy do dyspozycji następujące adnotacje/metody: GET, POST, PUT, PATCH, DELETE, OPTIONS i HEAD). W nawiasie zdefiniowaliśmy przedstawiony wcześniej endpoint. Zapytania mogą być parametryzowane przy użyciu @Path oraz @Query.

```
public interface WeatherAPIs {  
    @GET("/data/2.5/weather/")  
    Call<WeatherResult> getWeatherByCity(@Query("q") String city, @Query("units") String units, @Query("appid") String apiKey);  
}
```

- Kolejnym krokiem jest przygotowanie klasy z metodą statyczną, która zwraca instancję klasy Retrofit. Dokonamy tego używając Retrofit.builder oraz podając nasz bazowy URL oraz konwerter (który posłuży do deserializacji odpowiedzi, jest wiele opcji, ale my użyjemy gson).

```
public class NetworkClient {  
    public static final String BASE_URL = "http://api.openweathermap.org";  
    public static Retrofit retrofit;  
    //This public static method will return Retrofit client anywhere in the application  
    public static Retrofit getRetrofitClient() {  
        //If condition to ensure we don't create multiple retrofit instances in a single application  
        if (retrofit == null) {  
            retrofit = new Retrofit.Builder()  
                .baseUrl(BASE_URL) //This is the only mandatory call on Builder object.  
                .addConverterFactory(GsonConverterFactory.create())  
                .build();  
        }  
        return retrofit;  
    }  
}
```

Retrofit - Zapytania

Żądanie sieciowe (HTTP Request) zwraca instancję typu Call. Może zostać ono wykonane synchronicznie (execute) lub asynchronicznie (enqueue).

```
Retrofit retrofit = NetworkClient.getRetrofitClient();
/*The main purpose of Retrofit is to create HTTP calls from the Java interface based on the annotation associated with each method.
This is achieved by just passing the interface class as parameter to the create method */
WeatherApi weatherAPIs = retrofit.create(WeatherApi.class);
/*Invoke the method corresponding to the HTTP request which will return a Call object.
This Call object will be used to send the actual network request with the specified parameters */
Call call = weatherAPIs.getWeatherByCity( city: "London,uk", apiKey: "235bef5a99d6bc6193525182c409602c");
/*This is the line which actually sends a network request.
Calling enqueue() executes a call asynchronously. It has two callback listeners which will be invoked on the main thread */
call.enqueue(new Callback() {
    @Override
    public void onResponse(Call call, Response response) {
        /* This is the success callback. Though the response type is JSON,
        with Retrofit we get the response in the form of WeatherResponse POJO class */
        Log.i( TAG, "Pogoda", response.body().toString());
        if (response.body() != null) {
            WeatherResponse wResponse = (WeatherResponse) response.body();
            currentWeather.setText("Temp: " + wResponse.getMain().getTemp());
        }
    }
    @Override
    public void onFailure(Call call, Throwable t) {
        Log.e(TAG, "api response failure");
    }
});
}
```


Bibliografia



<https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>



<https://www.sciencedirect.com/topics/computer-science/application-programming-interface>



<https://www.mulesoft.com/resources/api/what-is-an-api>



<https://devszczepaniak.pl/wstep-do-rest-api/>



<https://www.json.org/json-en.html>



<https://www.samouczekprogramisty.pl/format-json-w-jezyku-java/>





<https://openweathermap.org>

Bibliografia

 [*https://square.github.io/retrofit/*](https://square.github.io/retrofit/)


 [*http://www.jsonschema2pojo.org*](http://www.jsonschema2pojo.org)

 [*https://futurestud.io/tutorials/gson-model-annotations-how-to-ignore-fields-with-expose*](https://futurestud.io/tutorials/gson-model-annotations-how-to-ignore-fields-with-expose)

 [*https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html*](https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html)

 [*https://developer.android.com/guide/components/fragments*](https://developer.android.com/guide/components/fragments)

 [*https://developer.android.com/guide/components/activities/intro-activities*](https://developer.android.com/guide/components/activities/intro-activities)

 [*https://developer.android.com/guide/topics/ui/declaring-layout*](https://developer.android.com/guide/topics/ui/declaring-layout)

Bibliografia



https://www.vogella.com/tutorials/AndroidListView/article.html#adapterperformance_holder



<https://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html#syntax>



<https://codinginflow.com/tutorials/android/fragment-to-fragment-communication-with-interfaces>



<https://developer.android.com/guide/topics/ui/controls/checkbox>



<https://github.com/navasmdc/MaterialDesignLibrary/issues/199>



<https://stackoverflow.com/questions/3965484/custom-checkbox-image-android>



<https://javastart.pl/baza-wiedzy/android/listview-przyklad-widoku-listowego>

The End