

# **LAPORAN PRAKTIKUM**

## **PEMROGRAMAN BERBASIS MOBILE**

### **PERTEMUAN KE-12**



**Disusun Oleh :**

**NAMA** : Raden Isnawan Argi Aryasatya  
**NIM** : 195410257  
**JURUSAN** : Informatika  
**JENJANG** : S1  
**KELAS** : 5

**Laboratorium Terpadu**  
**Sekolah Tinggi Manajemen Informatika Komputer**  
**AKAKOM**  
**YOGYAKARTA**

**2021**

## **PERTEMUAN KE-12** **(MENGAMBIL DATA DARI INTERNET)**

### **TUJUAN**

Mahasiswa mampu membuat aplikasi yang dapat mengakses data dari internet menggunakan Retrofit.

### **DASAR TEORI**

#### **1. Pengantar**

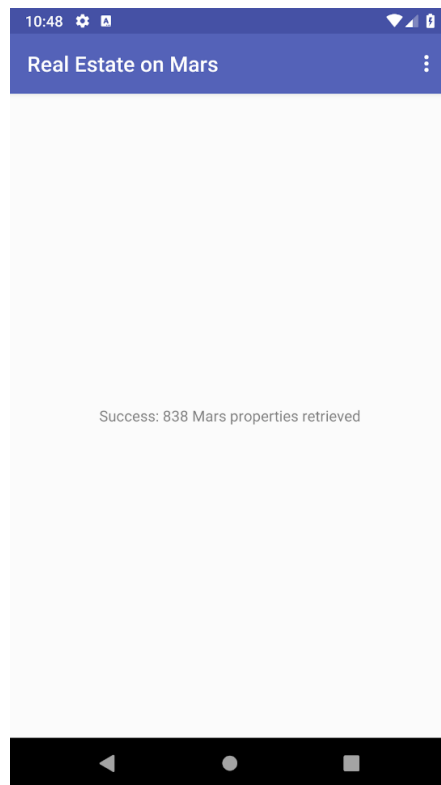
Hampir semua aplikasi Android yang Anda buat harus terhubung ke internet di beberapa titik. Dalam modul ini, Anda membuat aplikasi yang terhubung ke layanan web untuk mengambil dan menampilkan data. Anda juga mengembangkan apa yang Anda pelajari di codelab sebelumnya tentang ViewModel, LiveData, dan RecyclerView. Dalam modul ini, Anda menggunakan pustaka yang dikembangkan komunitas untuk membangun lapisan jaringan. Ini sangat menyederhanakan pengambilan data dan gambar, dan juga membantu aplikasi menyesuaikan dengan beberapa praktik terbaik Android, seperti memuat gambar pada thread latar belakang dan menyimpan gambar yang dimuat ke dalam cache. Untuk bagian asinkron atau non-pemblokiran dalam kode, seperti berbicara dengan lapisan layanan web (web service), Anda akan memodifikasi aplikasi untuk menggunakan coroutine Kotlin. Anda juga akan memperbarui antarmuka pengguna aplikasi jika internet lambat atau tidak tersedia untuk memberi tahu pengguna apa yang sedang terjadi.

#### **2. Ikhtisar aplikasi**

Dalam modul ini, Anda bekerja dengan aplikasi pemula yang disebut MarsRealEstate, yang menampilkan properti untuk dijual di Mars. Aplikasi ini terhubung ke layanan web (web service) untuk mengambil dan menampilkan data properti, termasuk detail seperti harga dan apakah properti tersedia untuk dijual atau disewakan. Gambar yang mewakili setiap properti adalah foto kehidupan nyata dari Mars yang diambil dari penjelajah Mars NASA.



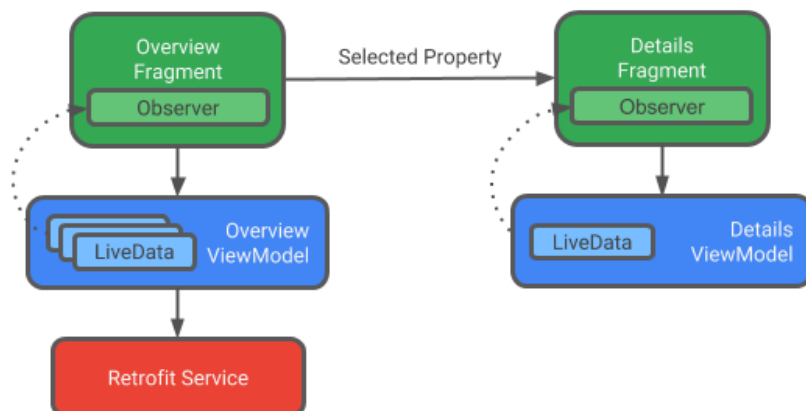
Versi aplikasi yang Anda buat dalam modul ini tidak akan memiliki banyak flash visual: ini berfokus pada bagian lapisan jaringan aplikasi untuk terhubung ke internet dan mengunduh data properti mentah menggunakan layanan web. Untuk memastikan bahwa data diambil dan diurai dengan benar, Anda hanya akan mencetak jumlah properti di Mars dalam tampilan teks:



### 3. Tugas: Jelajahi aplikasi starter MarsRealEstate

Arsitektur untuk aplikasi MarsRealEstate memiliki dua modul utama:

- Fragmen overview, yang berisi kisi gambar properti thumbnail, dibuat dengan RecyclerView.
- Fragmen detail tampilan, berisi informasi tentang setiap properti.



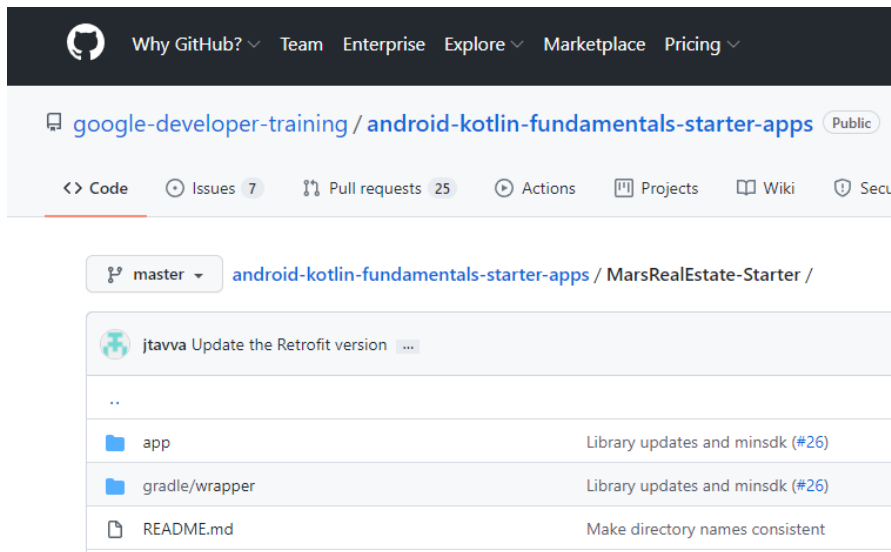
Aplikasi memiliki ViewModel untuk setiap fragmen. Untuk codelab ini, Anda membuat lapisan untuk layanan jaringan, dan ViewModel berkomunikasi langsung dengan lapisan jaringan tersebut. Ini mirip dengan apa yang Anda lakukan di codelab sebelumnya saat ViewModel berkomunikasi dengan database Room. Gambaran umum ViewModel bertanggung jawab melakukan panggilan jaringan untuk mendapatkan informasi real estate Mars. Detail ViewModel menyimpan detail untuk satu bagian real estat Mars yang ditampilkan di fragmen detail. Untuk setiap ViewModel, Anda menggunakan LiveData dengan pengikatan data berbasis siklus proses untuk memperbarui UI aplikasi saat datanya berubah. Anda menggunakan komponen Navigation untuk menavigasi di antara dua fragmen, dan meneruskan properti yang dipilih sebagai argumen.

---

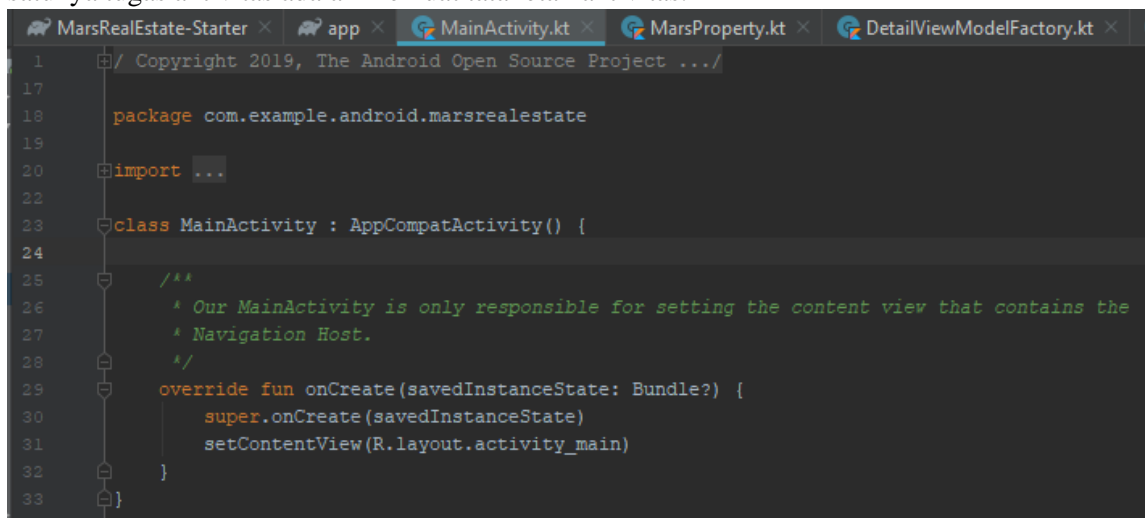
## PRAKTIK

### Langkah 1: Jelajahi fragmen dan navigasi

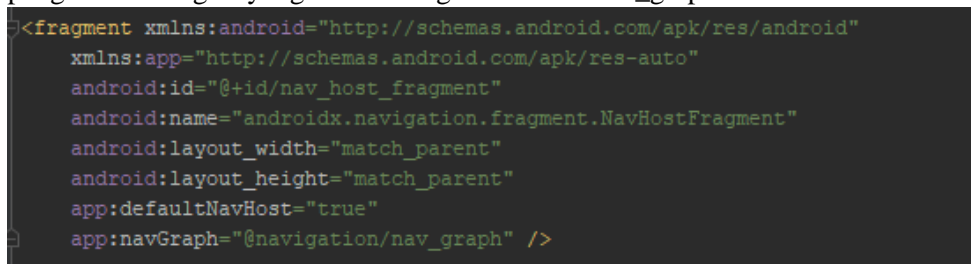
1. Unduh aplikasi starter MarsRealEstate (MarsRealEstate-Starter.zip juga disertakan di ELA) dan buka di Android Studio.



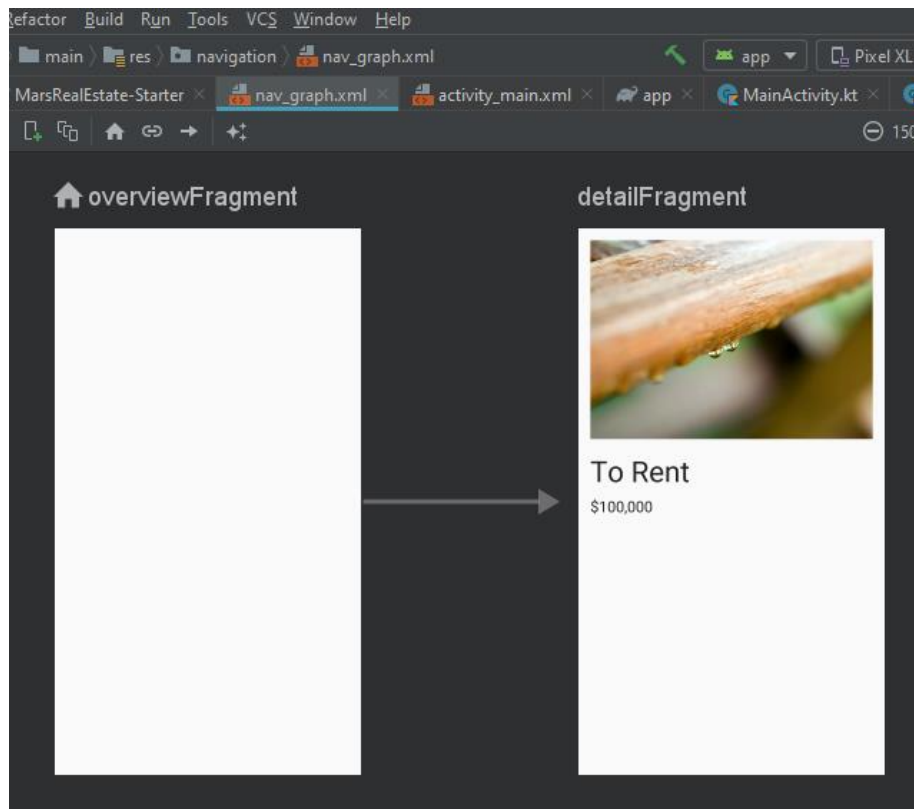
2. Periksa app/java/MainActivity.kt. Aplikasi menggunakan fragmen untuk kedua layar, jadi satu-satunya tugas aktivitas adalah memuat tata letak aktivitas.



3. Periksa app/res/layout/activity\_main.xml. Tata letak aktivitas adalah tuan rumah (host) untuk dua fragmen, yang ditentukan dalam file navigasi. Tata letak ini membuat instance NavHostFragment dan pengontrol navigasi yang terkait dengan resource nav\_graph.

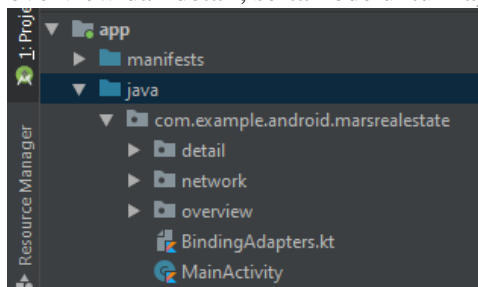


4. Buka app/res/navigation/nav\_graph.xml. Di sini Anda dapat melihat hubungan navigasi antara dua fragmen. Grafik navigasi StartDestination menunjuk ke overviewFragment, sehingga fragmen ikhtisar dibuat saat aplikasi diluncurkan.

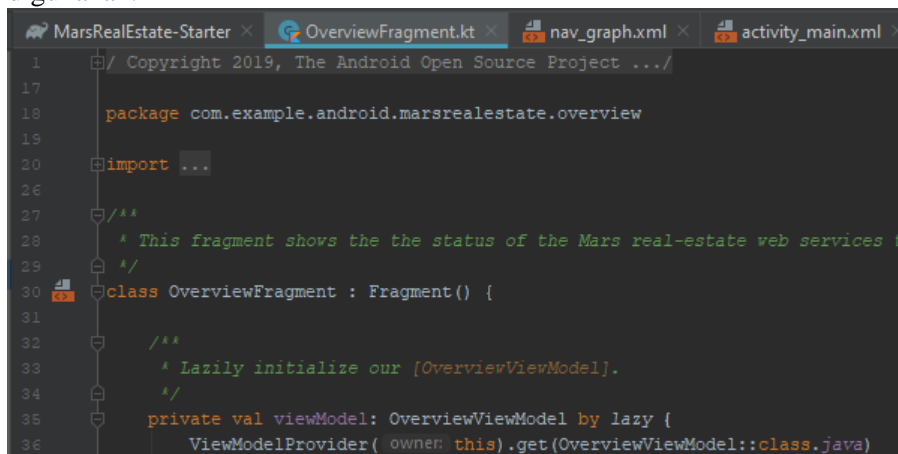


## Langkah 2: Jelajahi file sumber Kotlin dan data binding

1. Di panel Project, luaskan app>java. Perhatikan bahwa aplikasi MarsRealEstate memiliki tiga folder paket: detail, network, dan overview. Ini sesuai dengan tiga komponen utama aplikasi Anda: fragmen overview dan detail, serta kode untuk lapisan jaringan.



2. Buka app/java/overview/OverviewFragment.kt. OverviewFragment dengan secara lazy menginisialisasi OverviewViewModel, yang berarti OverviewViewModel dibuat saat pertama kali digunakan.



3. Periksa metode `onCreateView()`. Metode ini mengembangkan (inflate) tata letak `fragment_overview` menggunakan data binding, menyetel pemilik siklus proses binding ke dirinya sendiri (`this`), dan menyetel variabel `viewModel` di objek binding ke sana. Karena kita telah menyetel pemilik siklus proses, semua `LiveData` yang digunakan dalam pengikatan data akan secara otomatis diamati untuk setiap perubahan, dan UI akan diperbarui sesuai dengan itu.

```
/**
 * Inflates the layout with Data Binding, sets its lifecycle owner to the OverviewFragment
 * to enable Data Binding to observe LiveData, and sets up the RecyclerView with an adapter.
 */
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?): View? {
    val binding = FragmentOverviewBinding.inflate(inflater)

    // Allows Data Binding to Observe LiveData with the lifecycle of this Fragment
    binding.lifecycleOwner = this

    // Giving the binding access to the OverviewViewModel
    binding.viewModel = viewModel

    setHasOptionsMenu(true)
    return binding.root
}
```

4. Buka `app/java/overview/OverviewViewModel`. Karena responsnya adalah `LiveData` dan kita telah menyetel siklus proses untuk variabel binding, setiap perubahan padanya akan memperbarui UI aplikasi.

```
ate- Starter  x  OverviewFragment.kt  x  OverviewViewModel.kt  x  nav_graph.xml  x  activity_main
1  // Copyright 2019, The Android Open Source Project .../
17
18  package com.example.android.marsrealestate.overview
19
20  import ...
23
24  /**
25   * The [ViewModel] that is attached to the [OverviewFragment].
26   */
27  class OverviewViewModel : ViewModel() {
28
29      // The internal MutableLiveData String that stores the most recent response
30      private val _response = MutableLiveData<String>()
31
32      // The external immutable LiveData for the response String
33      val response: LiveData<String>
34      get() = _response
35  }
```

5. Periksa blok `init`. Saat `ViewModel` dibuat, ia memanggil metode `getMarsRealEstateProperties()`.

```
init {
    getMarsRealEstateProperties()
}
```

6. Periksa metode `getMarsRealEstateProperties()`. Dalam aplikasi starter ini, metode ini berisi respons placeholder. Tujuan dari codelab ini adalah untuk memperbarui respons `LiveData` dalam `ViewModel` menggunakan data nyata yang Anda dapatkan dari internet.

```
private fun getMarsRealEstateProperties() {
    _response.value = "Set the Mars API Response here!"
}
```

7. Buka `app/res/layout/fragment_overview.xml`. Ini adalah layout untuk fragmen `overview` yang Anda gunakan dalam codelab ini, dan ini menyertakan data binding untuk model tampilan. Ini mengimpor `OverviewViewModel` dan kemudian mengikat respons dari `ViewModel` ke `TextView`. Di codelab

selanjutnya, Anda mengganti tampilan teks dengan kisi gambar di RecyclerView.

rsRealEstate-Starter] - ...\\app\\src\\main\\res\\layout\\fragment\_overview.xml [app] - Android Studio

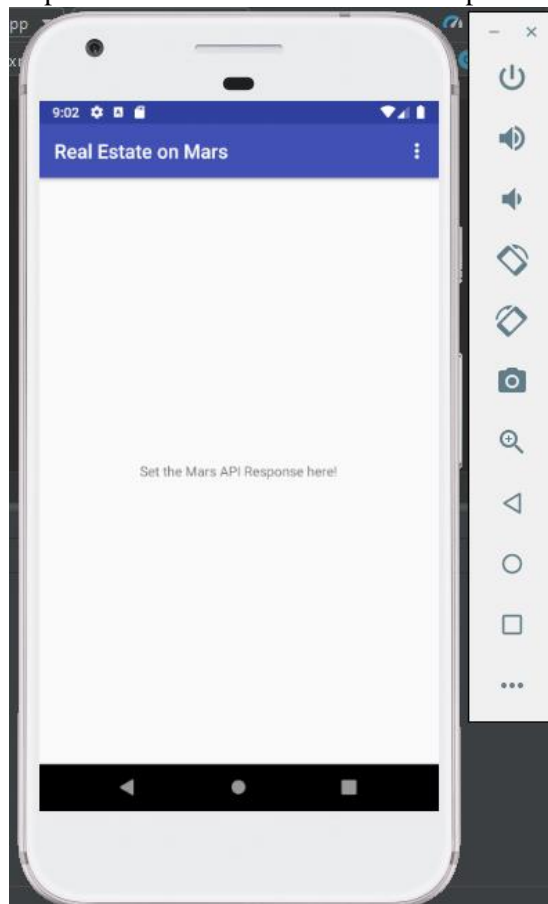
```
Tools VCS Window Help
layout > fragment_overview.xml
fragment_overview.xml OverviewViewModel.kt nav_graph.xml activity

<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <data>
        <variable
            name="viewModel"
            type="com.example.android.marsrealestate.overview.OverviewViewModel" />
        </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="com.example.android.marsrealestate.MainActivity">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{viewModel.response}"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

8. Kompilasi dan jalankan aplikasi. Semua yang Anda lihat di versi saat ini dari aplikasi ini adalah respons awal— "Set the Mars API Response here!"



## 4. Tugas: Menghubungkan ke layanan web dengan Retrofit

Data real estat Mars disimpan di server web, sebagai layanan web (web service) REST. Layanan web menggunakan arsitektur REST yang dibangun menggunakan komponen dan protokol web standar.

Anda membuat permintaan ke layanan web dengan cara standar melalui URI. URL web yang sudah dikenal sebenarnya adalah jenis URI, dan keduanya digunakan secara bergantian selama kursus ini. Misalnya, dalam aplikasi untuk pelajaran ini, Anda mengambil semua data dari server berikut:

<https://android-kotlin-fun-mars-server.appspot.com>

Jika Anda mengetik URL berikut di browser Anda, Anda akan mendapatkan daftar semua properti real estate yang tersedia di Mars!

<https://android-kotlin-fun-mars-server.appspot.com/realestate>

Respons dari layanan web biasanya diformat dalam JSON, format pertukaran untuk merepresentasikan data terstruktur. Anda mempelajari lebih lanjut tentang JSON di tugas berikutnya, tetapi penjelasan singkatnya adalah bahwa objek JSON adalah kumpulan pasangan kunci-nilai, terkadang disebut kamus, peta hash, atau array asosiatif. Kumpulan objek JSON adalah larik JSON, dan itu larik yang Anda dapatkan kembali sebagai respons dari layanan web.

Untuk memasukkan data ini ke dalam aplikasi, aplikasi Anda perlu membuat koneksi jaringan dan berkomunikasi dengan server tersebut, lalu menerima dan mengurai data respons ke dalam format yang dapat digunakan aplikasi. Dalam codelab ini, Anda menggunakan pustaka klien REST yang disebut Retrofit untuk membuat koneksi ini.

### Langkah 1: Tambahkan dependensi Retrofit ke Gradle

1. Buka build.gradle (Module: app).
2. Di bagian dependencies, tambahkan baris berikut untuk pustaka Retrofit:

```
implementation "com.squareup.retrofit2:retrofit:$version_retrofit"  
implementation "com.squareup.retrofit2:converter-scalars:$version_retrofit"
```

Perhatikan bahwa nomor versi ditentukan secara terpisah dalam file Gradle proyek. Dependensi pertama adalah untuk pustaka Retrofit 2 itu sendiri, dan dependensi kedua untuk konverter skalar Retrofit. Konverter ini mengaktifkan Retrofit untuk mengembalikan hasil JSON sebagai String. Kedua pustaka bekerja sama.

3. Klik Sync Now untuk membangun kembali proyek dengan dependensi baru.



### Langkah 2: Tambahkan dukungan untuk fitur bahasa Java 8

Banyak pustaka pihak ketiga termasuk Retrofit2 menggunakan fitur bahasa Java 8. Plugin Android Gradle menyediakan dukungan bawaan untuk menggunakan fitur bahasa Java 8 tertentu. Untuk menggunakan fitur bawaan ini, perbarui file modul build.gradle, seperti yang ditunjukkan di bawah ini:

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}  
kotlinOptions {  
    jvmTarget = JavaVersion.VERSION_1_8.toString()  
}
```

### Langkah 3: Terapkan MarsApiService



Retrofit membuat API jaringan untuk aplikasi berdasarkan konten dari layanan web. Ini mengambil data dari layanan web dan merutekannya melalui pustaka konverter terpisah yang mengetahui cara mendekode data dan mengembalikannya dalam bentuk objek yang berguna. Retrofit menyertakan dukungan bawaan untuk format data web populer seperti XML dan JSON. Retrofit pada akhirnya menciptakan sebagian besar lapisan jaringan untuk Anda, termasuk detail penting seperti menjalankan permintaan pada utas (thread) latar belakang.

Kelas MarsApiService memegang lapisan jaringan untuk aplikasi; yaitu, ini adalah API yang akan digunakan ViewModel Anda untuk berkomunikasi dengan layanan web. Ini adalah kelas tempat Anda akan menerapkan API layanan Retrofit.

1. Buka app/java/network/MarsApiService.kt. Saat ini file tersebut hanya berisi satu hal: konstanta untuk URL dasar untuk layanan web.

```
private const val BASE_URL = "https://android-kotlin-fun-mars-server.appspot.com/"
```

2. Tepat di bawah konstanta itu, gunakan Retrofit builder untuk membuat objek Retrofit. Import retrofit2.Retrofit dan retrofit2.converter.scalars.ScalarsConverterFactory jika diminta.

```
private val retrofit = Retrofit.Builder()
    .addConverterFactory(ScalarsConverterFactory.create())
    .baseUrl(BASE_URL)
    .build()
```

Retrofit memerlukan setidaknya dua hal yang tersedia untuk membangun API layanan web: URI dasar untuk layanan web, dan converter factory. Konverter memberi tahu Retrofit apa yang dilakukan dengan data yang didapatnya kembali dari layanan web. Dalam hal ini, Anda ingin Retrofit mengambil respons JSON dari layanan web, dan mengembalikannya sebagai String. Retrofit memiliki ScalarsConverter yang mendukung string dan jenis primitif lainnya, jadi Anda memanggil addConverterFactory() pada builder dengan instance ScalarsConverterFactory. Terakhir, Anda memanggil build() untuk membuat objek Retrofit.

3. Tepat di bawah panggilan ke Retrofit builder, tentukan antarmuka yang menentukan bagaimana Retrofit berbicara ke server web menggunakan permintaan HTTP. Impor retrofit2.http.GET dan retrofit2.Call saat diminta.

```
interface MarsApiService {
    @GET("realstate")
    fun getProperties():
        Call<String>
}
```

Saat ini tujuannya adalah untuk mendapatkan string respons JSON dari layanan web, dan Anda hanya perlu satu metode untuk melakukannya: getProperties(). Untuk memberi tahu Retrofit apa yang harus dilakukan metode ini, gunakan anotasi @GET dan tentukan jalur, atau titik akhir, untuk metode layanan web tersebut. Dalam hal ini titik akhir disebut realestate. Ketika metode getProperties() dipanggil, Retrofit menambahkan titik akhir realestate ke URL dasar (yang Anda tentukan di Retrofit builder), dan membuat objek Call. Objek Call itu digunakan untuk memulai permintaan.

Di bawah interface MarsApiService, tentukan objek publik yang disebut MarsApi untuk menginisialisasi layanan Retrofit.

```
object MarsApi {
    val retrofitService : MarsApiService by lazy {
        retrofit.create(MarsApiService::class.java) }
}
```

Metode Retrofit create() membuat layanan Retrofit itu sendiri dengan interface MarsApiService. Karena panggilan ini mahal, dan aplikasi hanya memerlukan satu instance layanan Retrofit, Anda mengekspos layanan ke aplikasi lainnya menggunakan objek publik yang disebut MarsApi, dan dengan malas menginisialisasi (lazily initialize) layanan Retrofit di sana. Sekarang setelah semua

penyiapan selesai, setiap kali aplikasi Anda memanggil MarsApi.retrofitService, aplikasi Anda akan mendapatkan objek Retrofit tunggal yang mengimplementasikan MarsApiService.

Secara lengkap MarsApiService.kt adalah seperti berikut:

```
package com.example.android.marsrealestate.network

import retrofit2.Call
import retrofit2.Retrofit
import retrofit2.converter.scalars.ScalarsConverterFactory
import retrofit2.http.GET

private const val BASE_URL = "https://android-kotlin-fun-mars-server.appspot.com/"

private val retrofit = Retrofit.Builder()
    .addConverterFactory(ScalarsConverterFactory.create())
    .baseUrl(BASE_URL)
    .build()

interface MarsApiService {
    @GET(value: "realestate")
    fun getProperties():
        Call<String>
}

object MarsApi {
    val retrofitService : MarsApiService by lazy {
        retrofit.create(MarsApiService::class.java) }
}
```

#### Langkah 4: Panggil layanan web di OverviewViewModel

1. Buka app/java/overview/OverviewViewModel.kt. Gulir ke bawah ke metode getMarsRealEstateProperties().

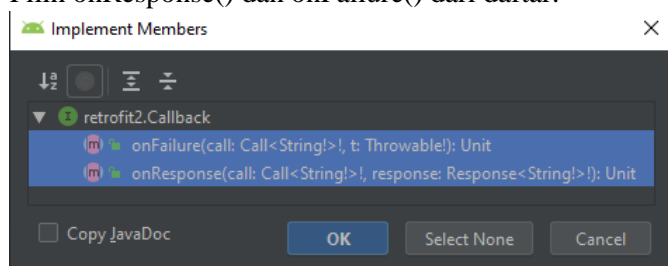
```
private fun getMarsRealEstateProperties() {
    _response.value = "Set the Mars API Response here!"
}
```

Ini adalah metode di mana Anda akan memanggil layanan Retrofit dan menangani string JSON yang dikembalikan. Saat ini hanya ada string placeholder untuk respons.

2. Hapus baris placeholder yang menyetel tanggapan ke "Set the Mars API Response here!"
3. Di dalam getMarsRealEstateProperties(), tambahkan kode yang ditunjukkan di bawah ini. Import retrofit2.Callback dan com.example.android.marsrealestate.network.MarsApi jika diminta. Metode MarsApi.retrofitService.getProperties() mengembalikan objek Call. Kemudian Anda bisa memanggil enqueue() pada objek itu untuk memulai permintaan jaringan pada thread latar belakang.

```
private fun getMarsRealEstateProperties() {
    MarsApi.retrofitService.getProperties().enqueue(
        object: Callback<String> {
        })
}
```

4. Klik pada kata object, yang digarisbawahi dengan warna merah. Pilih Code > Implement methods. Pilih onResponse() dan onFailure() dari daftar.



Android Studio menambahkan kode dengan TODO di setiap metode:

```
override fun onFailure(call: Call<String>, t: Throwable) {
    TODO( reason: "Not yet implemented")
}
override fun onResponse(call: Call<String>, response: Response<String>) {
    TODO( reason: "Not yet implemented")
}
}
```

5. Di `onFailure()`, hapus TODO dan setel `_response` ke pesan kegagalan, seperti yang ditunjukkan di bawah ini. `_response` adalah string LiveData yang menentukan apa yang ditampilkan dalam tampilan teks. Setiap status perlu memperbarui `_response` LiveData.

Callback `onFailure()` dipanggil saat respons layanan web gagal. Untuk respons ini, setel status `_response` ke "Failure: " yang digabungkan dengan pesan dari argumen Throwable.

```
override fun onFailure(call: Call<String>, t: Throwable) {
    _response.value = "Failure: " + t.message
}
```

6. Di `onResponse()`, hapus TODO dan setel `_response` ke isi respons. Callback `onResponse()` dipanggil saat permintaan berhasil dan layanan web mengembalikan respons.

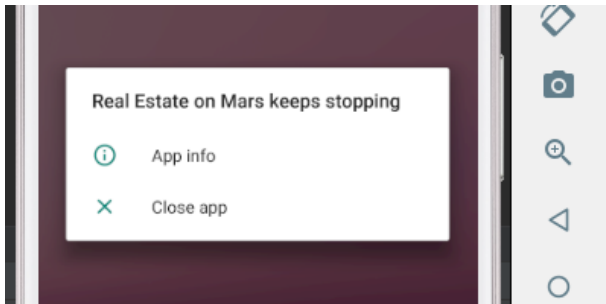
```
override fun onResponse(call: Call<String>, response: Response<String>) {
    _response.value = response.body()
}
```

Secara lengkap `OverviewViewModel.kt` adalah seperti berikut:

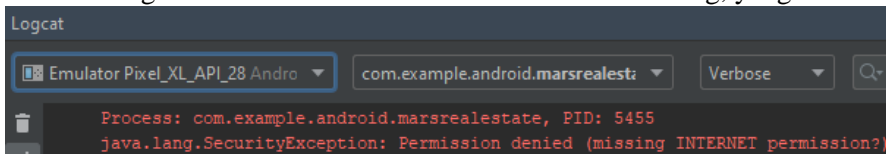
```
18 package com.example.android.marsrealestate.overview
19
20 import androidx.lifecycle.LiveData
21 import androidx.lifecycle.MutableLiveData
22 import androidx.lifecycle.ViewModel
23 import com.example.android.marsrealestate.network.MarsApi
24 import retrofit2.Call
25 import retrofit2.Response
26 import javax.security.auth.callback.Callback
27
28 /**
29  * The [ViewModel] that is attached to the [OverviewFragment].
30  */
31 class OverviewViewModel : ViewModel() {
32
33     // The internal MutableLiveData String that stores the most recent response
34     private val _response = MutableLiveData<String>()
35
36     // The external immutable LiveData for the response String
37     val response: LiveData<String>
38         get() = _response
39
40     /**
41     * Call getMarsRealEstateProperties() on init so we can display status immediately.
42     */
43     init {
44         getMarsRealEstateProperties()
45     }
46
47     /**
48     * Sets the value of the status LiveData to the Mars API status.
49     */
50     private fun getMarsRealEstateProperties() {
51         MarsApi.retrofitService.getProperties().enqueue(
52             object: retrofit2.Callback<String> {
53                 override fun onFailure(call: Call<String>, t: Throwable) {
54                     _response.value = "Failure: " + t.message
55                 }
56                 override fun onResponse(call: Call<String>, response: Response<String>) {
57                     _response.value = response.body()
58                 }
59             })
60     }
61 }
```

## Langkah 5: Tentukan internet permission

1. Kompilasi dan jalankan aplikasi MarsRealEstate. Perhatikan bahwa aplikasi segera menutup dengan kesalahan.

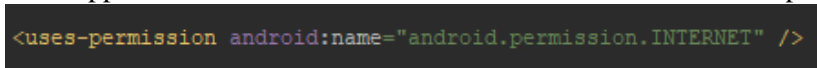


2. Klik tab Logcat di Android Studio dan catat kesalahan di log, yang dimulai dengan baris seperti ini:

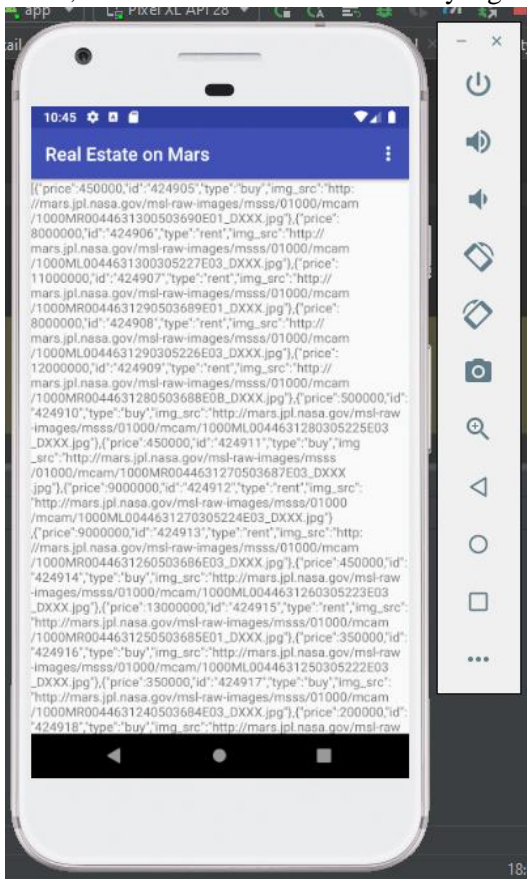


Pesan kesalahan memberi tahu Anda bahwa aplikasi Anda mungkin kehilangan izin INTERNET. Menghubungkan ke internet menimbulkan masalah keamanan, itulah sebabnya aplikasi tidak memiliki konektivitas internet secara default. Anda perlu memberi tahu Android secara eksplisit bahwa aplikasi memerlukan akses ke internet.

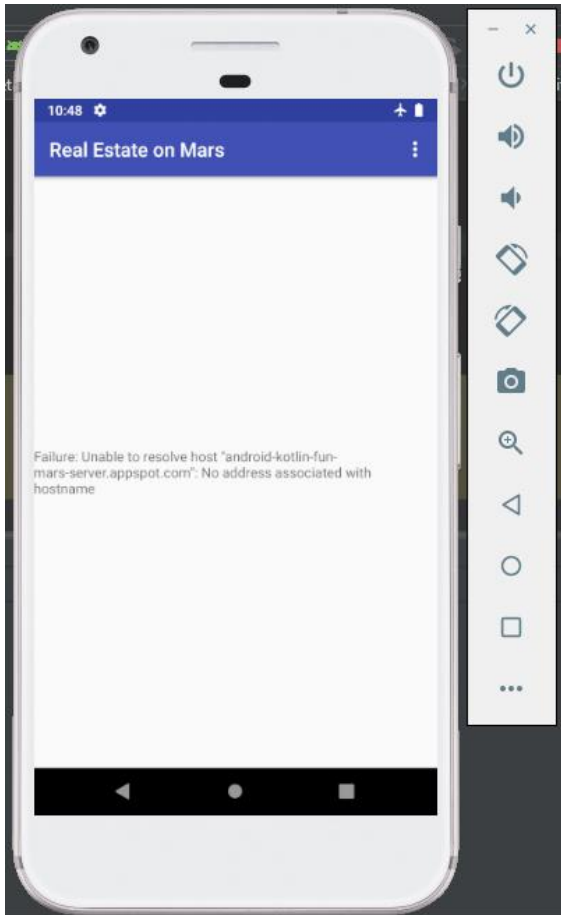
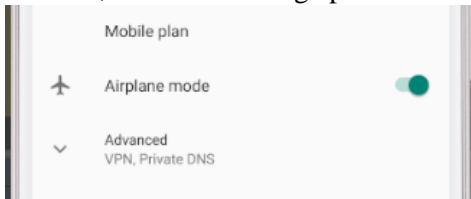
3. Buka app/manifests/AndroidManifest.xml. Tambahkan baris ini tepat sebelum tag <application>:



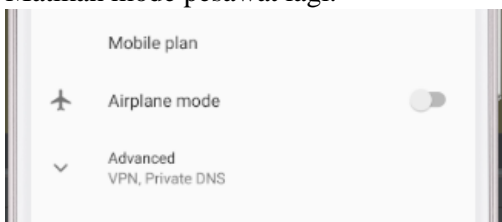
4. Kompilasi dan jalankan aplikasi lagi. Jika semuanya bekerja dengan benar dengan koneksi internet Anda, Anda akan melihat teks JSON yang berisi data Mars Property.



5. Ketuk tombol Back di perangkat atau emulator Anda untuk menutup aplikasi.
6. Alihkan perangkat atau emulator Anda ke mode pesawat, lalu buka kembali aplikasi dari menu Terbaru, atau mulai ulang aplikasi dari Android Studio.



7. Matikan mode pesawat lagi.



---

## 5. Tugas: Mengurai respons JSON dengan Moshi

Sekarang Anda mendapatkan respons JSON dari layanan web Mars, yang merupakan awal yang baik. Tapi yang benar-benar Anda butuhkan adalah objek Kotlin, bukan string JSON yang besar. Ada pustaka bernama Moshi, yang merupakan parser Android JSON yang mengubah string JSON menjadi objek Kotlin. Retrofit memiliki konverter yang berfungsi dengan Moshi, jadi ini adalah pustaka yang bagus untuk tujuan Anda di sini. Dalam tugas ini, Anda menggunakan pustaka Moshi dengan Retrofit untuk mengurai respons JSON dari layanan web menjadi objek Mars Property Kotlin yang berguna. Anda mengubah aplikasi sehingga sebagai

ganti menampilkan JSON mentah, aplikasi menampilkan jumlah Properti Mars yang dikembalikan.

### Langkah 1: Tambahkan dependensi pustaka Moshi

1. Buka build.gradle (Module: app).
2. Di bagian dependensi, tambahkan kode yang ditunjukkan di bawah ini untuk menyertakan dependensi Moshi. Seperti pada Retrofit, \$version\_moshi ditentukan secara terpisah dalam file Gradle tingkat proyek. Dependensi ini menambahkan dukungan untuk pustaka Moshi JSON dengan dukungan Kotlin.

```
implementation "com.squareup.moshi:moshi-kotlin:$version_moshi"
```

3. Temukan baris untuk converter Retrofit skalar di blok dependensi:

```
implementation "com.squareup.retrofit2:retrofit:$version_retrofit"  
implementation "com.squareup.retrofit2:converter-scalars:$version_retrofit"
```

4. Ubah baris ini untuk menggunakan converter-moshi:

```
implementation "com.squareup.retrofit2:converter-moshi:$version_retrofit"
```

5. Klik Sync Now untuk membangun kembali proyek dengan dependensi baru.

### Langkah 2: Terapkan kelas data MarsProperty

Contoh entri respons JSON yang Anda dapatkan dari layanan web terlihat seperti ini:

```
[{"price":450000,  
  "id":"424906",  
  "type":"rent",  
  "img_src":"http://mars.jpl.nasa.gov/msl-raw-  
images/msss/01000/mcam/1000ML0044631300305227E03_DXXX.jpg"},  
...]
```

Respons JSON yang ditunjukkan di atas adalah larik, yang ditunjukkan dengan tanda kurung siku. Larik berisi objek JSON, yang dikelilingi oleh tanda kurung kurawal. Setiap objek berisi satu set pasangan nama-nilai, dipisahkan oleh titik dua. Nama-nama diapit tanda kutip. Nilai bisa berupa angka atau string, dan string juga dikelilingi oleh tanda kutip. Misalnya, harga properti ini adalah \$450.000 dan img\_src adalah URL, yang merupakan lokasi file gambar di server.

Pada contoh di atas, perhatikan bahwa setiap entri properti Mars memiliki pasangan kunci dan nilai JSON berikut:

- price: harga properti Mars, sebagai angka.
- id: ID properti, sebagai string.
- type: "rent" atau "by".
- img\_src: URL gambar sebagai string.

Moshi mengurai data JSON ini dan mengubahnya menjadi objek Kotlin. Untuk melakukan ini, perlu memiliki kelas data Kotlin untuk menyimpan hasil parsing, jadi langkah selanjutnya adalah membuat kelas itu.

1. Buka app/java/network/MarsProperty.kt.
2. Ganti definisi kelas MarsProperty yang ada dengan kode berikut:

```
@data class MarsProperty(  
    val id: String, val img_src: String,  
    val type: String,  
    val price: Double
```

Perhatikan bahwa setiap variabel di kelas MarsProperty sesuai dengan nama kunci di objek JSON.

Untuk mencocokkan tipe di JSON, Anda menggunakan objek String untuk semua nilai kecuali price, yang merupakan Double. Double dapat digunakan untuk mewakili bilangan pada JSON.

Saat Moshi mem-parsing JSON, ia mencocokkan kunci berdasar nama dan mengisi objek data dengan nilai yang sesuai.

3. Ganti baris untuk kunci `img_src` dengan baris yang ditunjukkan di bawah ini. Impor `com.squareup.moshi.Json` jika diminta.

```
@Json(name = "img_src") val imgSrcUrl: String,
```

Terkadang nama kunci dalam respons JSON dapat membuat properti Kotlin yang membingungkan, atau mungkin tidak cocok dengan gaya pengkodean Anda — misalnya, dalam file JSON kunci `img_src` menggunakan garis bawah, sedangkan properti Kotlin biasanya menggunakan huruf besar dan kecil ("camel case").

Untuk menggunakan nama variabel di kelas data Anda yang berbeda dari nama kunci dalam respons JSON, gunakan anotasi `@Json`. Dalam contoh ini, nama variabel di kelas data adalah `imgSrcUrl`. Variabel dipetakan ke atribut JSON `img_src` menggunakan `@Json (name = "img_src")`.

Secara lengkap `MarsProperty.kt` berisi seperti berikut:

```
package com.example.android.marsrealestate.network

import com.squareup.moshi.Json

data class MarsProperty(
    val id: String, @Json(name = "img_src") val imgSrcUrl: String,
    val type: String,
    val price: Double
)
```

### Langkah 3: Perbarui `MarsApiService` dan `OverviewViewModel`

Dengan kelas data `MarsProperty` di tempat, Anda sekarang dapat memperbarui API jaringan dan `ViewModel` untuk menyertakan data Moshi.

1. Buka `network/MarsApiService.kt`. Anda mungkin melihat kesalahan kelas yang hilang untuk `ScalarsConverterFactory`. Ini karena perubahan dependensi Retrofit yang Anda buat pada Langkah 1. Anda segera memperbaiki kesalahan tersebut.

```
.addConverterFactory(ScalarsConverterFactory.create())
```

2. Di bagian atas file, tepat sebelum Retrofit builder, tambahkan kode berikut untuk membuat instance Moshi.

```
private val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()
```

Impor `com.squareup.moshi.Moshi` dan `com.squareup.moshi.kotlin.reflect.KotlinJsonAdapterFactory` jika diminta.

Mirip dengan apa yang Anda lakukan dengan Retrofit, di sini Anda membuat objek moshi menggunakan Moshi builder. Agar anotasi Moshi berfungsi dengan baik dengan Kotlin, tambahkan `KotlinJsonAdapterFactory`, lalu panggil `build()`.

3. Ubah Retrofit builder untuk menggunakan `MoshiConverterFactory` sebagai ganti `ScalarConverterFactory`, dan teruskan instance moshi yang baru saja Anda buat.

Impor `retrofit2.converter.moshi.MoshiConverterFactory` jika diminta.



```
private val retrofit = Retrofit.Builder()
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .baseUrl(BASE_URL)
    .build()
```

4. Hapus juga impor untuk ScalarConverterFactory.

Kode untuk dihapus:

```
import retrofit2.converter.scalars.ScalarsConverterFactory
```

5. Perbarui interface MarsApiService agar Retrofit menampilkan daftar objek MarsProperty, sebagai pengganti menampilkan Call<String>.

```
interface MarsApiService {
    @GET("realstate")
    fun getProperties():
        Call<List<MarsProperty>>
}
```

6. Buka OverviewViewModel.kt. Gulir ke bawah ke panggilan ke getProperties().enqueue() dalam metode getMarsRealEstateProperties().

7. Ubah argumen pada enqueue() dari Callback<String> menjadi Callback<List<MarsProperty>>. Impor com.example.android.marsrealstate.network.MarsProperty jika diminta.

```
MarsApi.retrofitService.getProperties().enqueue(
    object: Callback<List<MarsProperty>> {
```

8. Di onFailure(), ubah argumen dari Call<String> ke Call<List<MarsProperty>>:

```
override fun onFailure(call: Call<List<MarsProperty>>, t: Throwable) {
```

9. Lakukan perubahan yang sama pada kedua argumen ke onResponse():

```
override fun onResponse(call: Call<List<MarsProperty>>,
    response: Response<List<MarsProperty>>) {
```

10. Di badan onResponse(), ganti tugas yang ada ke \_response.value dengan tugas yang ditampilkan di bawah. Karena response.body() sekarang menjadi daftar objek MarsProperty, ukuran daftar tersebut adalah jumlah properti yang diurai. Pesan respon ini mencetak sejumlah properti:

```
_response.value = "Success: ${response.body()?.size} Mars properties retrieved"
```

11. Secara lengkap OverviewViewModel.kt menjadi seperti berikut:

```
OverviewViewModel.kt x MarsProperty.kt x AndroidManifest.xml x MarsApiService.kt x fragment_detail
1  // Copyright 2019, The Android Open Source Project .../
17
18  package com.example.android.marsrealstate.overview
19
20  import androidx.lifecycle.LiveData
21  import androidx.lifecycle.MutableLiveData
22  import androidx.lifecycle.ViewModel
23  import com.example.android.marsrealstate.network.MarsApi
24  import com.example.android.marsrealstate.network.MarsProperty
25  import retrofit2.Call
26  import retrofit2.Callback
27  import retrofit2.Response
28
29  /**
30   * The [ViewModel] that is attached to the [OverviewFragment].
31   */
32  class OverviewViewModel : ViewModel() {
33
34      // The internal MutableLiveData String that stores the most recent response
35      private val _response = MutableLiveData<String>()
36
```

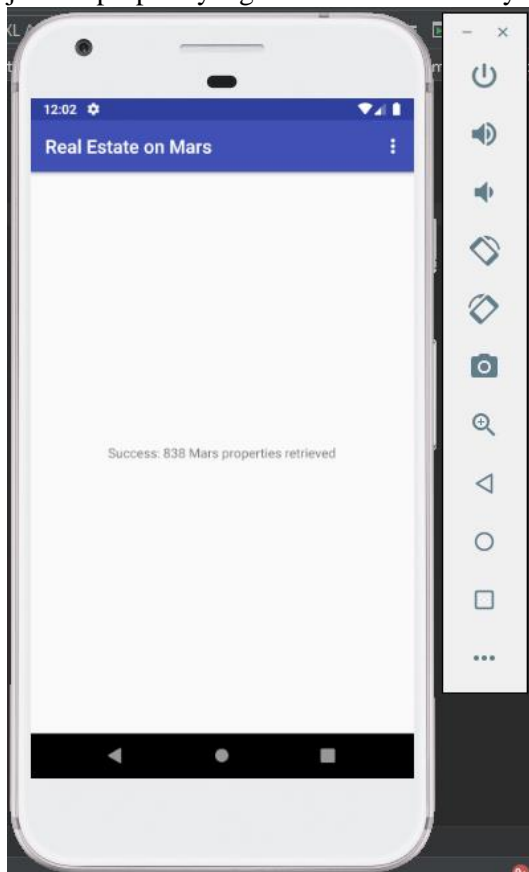


```

37 // The external immutable LiveData for the response String
38 val response: LiveData<String>
39     get() = _response
40
41 /**
42  * Call getMarsRealEstateProperties() on init so we can display status immediately.
43  */
44 init {
45     getMarsRealEstateProperties()
46 }
47
48 /**
49  * Sets the value of the status LiveData to the Mars API status.
50  */
51 private fun getMarsRealEstateProperties() {
52     MarsApi.retrofitService.getProperties().enqueue(
53         object: Callback<List<MarsProperty>> {
54             override fun onFailure(call: Call<List<MarsProperty>>, t: Throwable) {
55                 _response.value = "Failure: " + t.message
56             }
57             override fun onResponse(call: Call<List<MarsProperty>>,
58                                     response: Response<List<MarsProperty>>) {
59                 _response.value = "Success: ${response.body()?.size} Mars properties retrieved"
60             }
61         })
62 }
63 }

```

12. Pastikan mode pesawat dimatikan. Kompilasi dan jalankan aplikasi. Kali ini pesan harus menunjukkan jumlah properti yang dikembalikan dari layanan web:



## 6. Tugas: Menggunakan coroutine dengan Retrofit

Sekarang layanan Retrofit API sedang berjalan, tetapi menggunakan callback dengan dua metode callback yang harus Anda terapkan. Satu metode menangani keberhasilan dan yang lain menangani kegagalan, dan hasil kegagalan melaporkan eksepsi. Kode Anda akan lebih efisien dan lebih mudah dibaca jika Anda dapat menggunakan coroutine dengan penanganan eksepsi, daripada menggunakan callback. Dalam tugas ini, Anda

mengonversi layanan jaringan dan ViewModel untuk menggunakan coroutines.

### Langkah 1: Perbarui MarsApiService dan OverviewViewModel

1. Di MarsApiService, buat getProperties() sebagai fungsi penangguhan (suspend). Ubah Call<List<MarsProperty>> ke List<MarsProperty>. Metode getProperties() terlihat seperti ini:

```
@GET("realestate")
suspend fun getProperties(): List<MarsProperty>
```

2. Di file OverviewViewModel.kt, hapus semua kode di dalam getMarsRealEstateProperties(). Anda akan menggunakan coroutine di sini sebagai ganti panggilan ke enqueue() dan callback onFailure() dan onResponse().

3. Di dalam getMarsRealEstateProperties(), luncurkan coroutine menggunakan viewModelScope.

```
viewModelScope.launch { this: CoroutineScope
}
}
```

ViewModelScope adalah cakupan coroutine bawaan yang ditentukan untuk setiap ViewModel di aplikasi Anda. Coroutine apa pun yang diluncurkan dalam cakupan ini otomatis dibatalkan jika ViewModel dihapus.

4. Di dalam blok launch, tambahkan blok try/catch untuk menangani eksepsi:

```
try {
} catch (e: Exception) {
}
```

5. Di dalam blok try {}, panggil getProperties() pada objek retrofitService:

```
val listResult = MarsApi.retrofitService.getProperties()
```

Memanggil getProperties() dari layanan MarsApi akan membuat dan memulai panggilan jaringan pada thread latar belakang.

6. Juga di dalam blok try {}, perbarui pesan tanggapan untuk tanggapan yang berhasil:

```
_response.value = "Success: ${listResult.size} Mars properties retrieved"
```

7. Di dalam blok catch {}, tangani respons kegagalan:

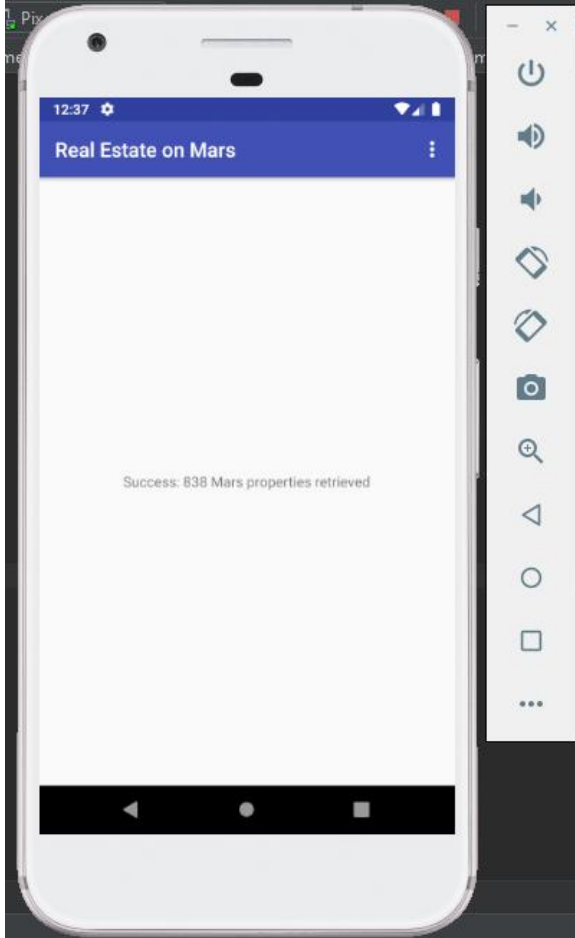
```
_response.value = "Failure: ${e.message}"
```

Metode getMarsRealEstateProperties() lengkap sekarang terlihat seperti ini:

```
private fun getMarsRealEstateProperties() {
    viewModelScope.launch { this: CoroutineScope
        try {
            val listResult = MarsApi.retrofitService.getProperties()
            _response.value = "Success: ${listResult.size} Mars properties retrieved"
        } catch (e: Exception) {
            _response.value = "Failure: ${e.message}"
        }
    }
}
```

8. Kompilasi dan jalankan aplikasi. Anda mendapatkan hasil yang sama seperti di tugas sebelumnya

(laporan jumlah properti), tetapi dengan kode yang lebih langsung dan penanganan error.



---

## KESIMPULAN

Di pertemuan ke-12 ini saya berhasil memenuhi tujuan dibuatnya modul ini yaitu mampu membuat aplikasi yang dapat mengakses data dari internet menggunakan Retrofit. Di praktikum ini, saya menggunakan aplikasi starter bernama MarsRealEstate. Selama proses pengerjaan laporan ke-12 ini, saya melakukan banyak hal baru seperti mengubah aplikasi starter untuk membuat permintaan API layanan web dan menangani respon, menerapkan lapisan jaringan untuk aplikasi dengan menggunakan pustaka Retrofit, parsing respon JSON dari layanan web ke dalam data aplikasi dengan library Moshi, dan menggunakan Retrofit support untuk coroutine untuk menyederhanakan kode.

**Terima Kasih**