1. Lab 8

a) Pobieranie danych z Internetu
b) Ładowanie i wyświetlanie zdjęć z Internetu
c) Filtrowanie i widok szczegółów z danymi internetowymi

Wprowadzenie

Prawie każda aplikacja na Androida w pewnym momencie połączy się z Internetem. W ćwiczeniu, zbudujesz aplikację, która łączy się z usługą internetową w celu pobierania i wyświetlania danych Opierasz się również na tym, czego nauczyłeś się w poprzednich kodelabach na temat ViewModel, LiveData a dodatkowo `RecyclerView`.

Będziesz używać bibliotek opracowanych przez społeczność do budowania warstwy sieci. To znacznie upraszcza pobieranie danych i obrazów, a także pomaga aplikacji dostosować się do niektórych najlepszych praktyk Androida, takich jak ładowanie obrazów w tle i buforowanie załadowanych obrazów. W przypadku asynchronicznych lub nieblokujących sekcji w kodzie, takich jak rozmowa z warstwą usług internetowych, zmodyfikujesz aplikację tak, aby korzystać z coroutines Kotlina. Zaktualizujesz również interfejs użytkownika aplikacji, jeśli Internet jest wolny lub niedostępny, aby użytkownik wiedział, co się dzieje.

Czego się nauczysz

- Co to jest usługa internetowa REST.
- Korzystanie z biblioteki Retrofit w celu połączenia z usługą REST w Internecie i uzyskania odpowiedzi.
- Użycie biblioteki Moshi do parsowania odpowiedzi JSON na obiekt danych.

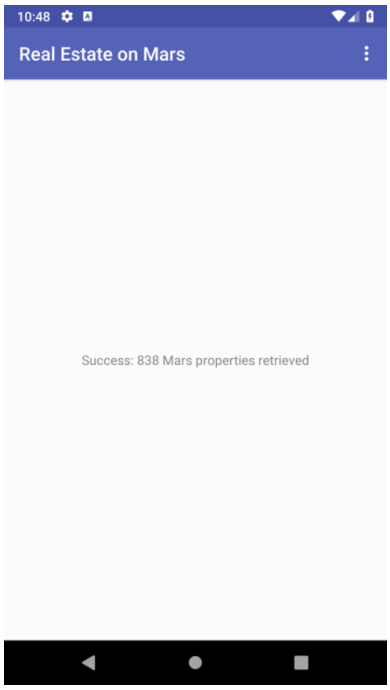
Co będziesz robić

- Zmodyfikuj aplikację startową, aby utworzyć żądanie interfejsu API usługi sieci Web i obsłużyć odpowiedź.
- Zaimplementuj warstwę sieciową dla swojej aplikacji za pomocą biblioteki Retrofit.
- Analizuj odpowiedź JSON z usługi sieciowej w danych na żywo aplikacji za pomocą biblioteki Moshi.
- Użyj wsparcia Retrofit dla coroutines, aby uprościć kod.

2. Przegląd aplikacji

W tym ćwiczeniu pracujesz z aplikacją startową o nazwie MarsRealEstate, która pokazuje działki na sprzedaż na Marsie. Ta aplikacja łączy się z usługą internetową w celu pobrania i wyświetlenia danych nieruchomości, w tym takich szczegółów, jak cena i czy nieruchomość jest dostępna do sprzedaży lub do wynajęcia. Obrazy reprezentujące każdą nieruchomość to prawdziwe zdjęcia z Marsa zrobione z łazików marsjańskich NASA.

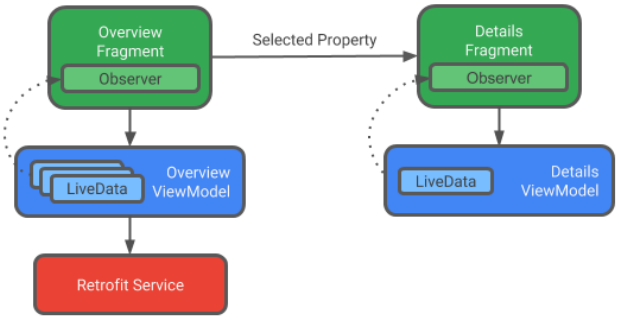
Wersja aplikacji, którą zbudujesz pierwszej częsci nie będzie miała dużo efektów wizualnych, Skoncentrujesz się na części aplikacji w warstwie sieciowej, aby połączyć się z Internetem i pobrać surowe dane działek za pomocą usługi internetowej. Aby upewnić się, że dane są poprawnie pobierane i analizowane, wystarczy wydrukować liczbę działek na Marsie w widoku tekstowym:



3. Zadanie: poznaj aplikację startową MarsRealEstate

Architektura aplikacji MarsRealEstate składa się z dwóch głównych modułów:

- Fragment poglądowy, który zawiera siatkę obrazów właściwości miniatur, zbudowany za pomocą. `RecyclerView`.
- Fragment widoku szczegółowego, zawierający informacje o każdej nieruchomości.



Aplikacja ma ViewModel dla każdego fragmentu. Dla tego kodu tworzysz warstwę dla usługi sieciowej, a ViewModel komunikuje się bezpośrednio z tą warstwą sieci. Jest to podobne do tego, co robiłeś w poprzednich kodelabach, gdy ViewModel komunikował się z bazą danych Room.

Przegląd ViewModel jest odpowiedzialny za nawiązywanie połączeń sieciowych w celu uzyskania informacji o nieruchomościach na Marsie. Szczegół ViewModel przechowuje szczegóły dotyczące pojedynczej nieruchomości Marsa, która jest wyświetlana we fragmencie szczegółów. Dla każdego ViewModel używasz LiveData z powiązaniem danych uwzględniającym cykl życia, aby aktualizować interfejs użytkownika aplikacji, gdy dane się zmieniają.

Komponent Nawigacja służy do nawigowania między dwoma fragmentami i przekazywania wybranej właściwości jako argumentu.

W tym zadaniu pobierasz i uruchamiasz aplikację startową dla MarsRealEstate i zapoznajesz się ze strukturą projektu.

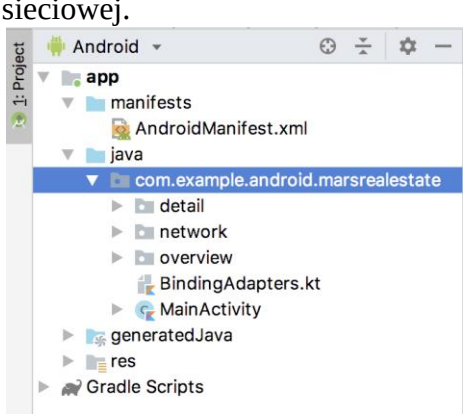
Step 1: fragmenty i nawigacja

1. Pobierz aplikację startową MarsRealEstate i otwórz ją w Android Studio.
2. Sprawdź `app/java/MainActivity.kt`. Aplikacja używa fragmentów dla obu ekranów, więc jedynym zadaniem dla activity jest załadowanie layoutu activity.

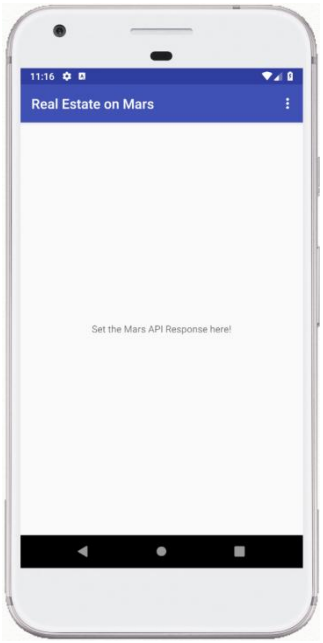
3. Sprawdź `app/res/layout/activity_main.xml`. jest hostem dla dwóch fragmentów zdefiniowanych w pliku nawigacyjnym. Ten układ tworzy instancję `NavHostFragment` i związanego z nią kontrolera nawigacji z zasobem `nav_graph`.
4. Otwórz `app/res/navigation/nav_graph.xml`. Tutaj możesz zobaczyć relację nawigacyjną między dwoma fragmentami. Wykres nawigacyjny `StartDestination` wskazuje na `overviewFragment`, więc fragment overview jest tworzony przy uruchamianiu aplikacji.

Step 2: Poznaj pliki źródłowe Kotlin i powiązanie danych

1. W panelu Projekt rozwiń **app** > **java**. Zauważ, że aplikacja MarsRealEstate ma trzy foldery pakietów: `detail`, `network`, i `overview`. Odpowiadają one trzem głównym komponentom aplikacji: overview fragment, detail fragment, oraz kodowi warstwy sieciowej.



2. Otwórz `app/java/overview/OverviewFragment.kt`. `OverviewFragment` "leniwie" inicjuje `OverviewViewModel`, , co oznacza, że `OverviewViewModel` jest tworzony przy pierwszym użyciu..
3. Sprawdź metodę `onCreateView()` method Ta metoda napełnia układ fragmentu `fragment_overview` za pomocą powiązania danych, ustawia właściciela cyklu życia powiązania na siebie (`this`), i ustawia dla niego zmienną `viewModel` w obiekcie powiązania. Ponieważ ustawiliśmy właściciela cyklu życia, wszelkie `LiveData` użyte w powiązaniu danych będą automatycznie obserwowane dla wszelkich zmian, a interfejs użytkownika zostanie odpowiednio zaktualizowany.
4. Otwórz `app/java/overview/OverviewViewModel`. Ponieważ odpowiedzią jest `LiveData` i ustawiliśmy cykl życia zmiennej powiązania, wszelkie zmiany w niej zaktualizują interfejs aplikacji.
5. Sprawdź blok `init` Podczas tworzenia `ViewModel` wywołuje metodę `getMarsRealEstateProperties()`.
6. Sprawdź metodę `getMarsRealEstateProperties()` method. W tej aplikacji startowej ta metoda zawiera (placeholder response). Celem tego kodu jest aktualizacja odpowiedzi LiveData w ramach ViewModel przy użyciu rzeczywistych danych uzyskanych z Internetu.
7. Otwórz `app/res/layout/fragment_overview.xml`. Jest to układ fragmentu z którym pracujesz w tej części, i obejmuje on powiązanie danych dla modelu widoku. Importuje OverviewViewModel, a następnie wiąże odpowiedź z ViewModel z TextView. W późniejszych częściach zamieniasz widok tekstu na siatkę obrazów w RecyclerView `RecyclerView`.
8. Skompiluj i uruchom aplikację. W obecnej wersji tej aplikacji widzisz tylko odpowiedź startera: —"Set the Mars API Response here!"



4. Zadanie: Połącz się z usługą internetową za pomocą Retrofit

Dane nieruchomości Mars są przechowywane na serwerze internetowym jako usługa internetowa REST. Usługi sieciowe wykorzystujące architekturę REST są budowane przy użyciu standardowych komponentów i protokołów internetowych.

Zgłaszasz żądanie do usługi internetowej w znormalizowany sposób za pomocą identyfikatorów URI. Znany internetowy adres URL jest w rzeczywistości rodzajem identyfikatora URI i oba są używane zamiennie w trakcie tego kursu. Na przykład w aplikacji dla tej lekcji pobierasz wszystkie dane z następującego serwera:

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

Jeśli wpiszesz następujący adres URL w przeglądarce, otrzymasz listę wszystkich dostępnych nieruchomości na Marsie!

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

Odpowiedź z usługi internetowej jest zwykle sformatowana w JSON, formacie wymiany służącym do reprezentowania danych strukturalnych. Dowiesz się więcej o JSON w następnym zadaniu, ale krótkie wyjaśnienie jest takie, że obiekt JSON to zbiór par klucz-wartość, czasami nazywany słownikiem, mapą skrótu lub tablicą asocjacyjną. Zbiór obiektów JSON jest tablicą JSON i jest to tablica, którą otrzymujesz w odpowiedzi z usługi internetowej.

Aby pobrać te dane do aplikacji, aplikacja musi ustanowić połączenie sieciowe i komunikować się z tym serwerem, a następnie odbierać i analizować dane odpowiedzi w formacie, którego może używać aplikacja. W tym ćwiczeniu używasz biblioteki klienta REST o nazwie Retrofit, aby nawiązać to połączenie..

Step 1: Dodaj zależności modernizacji do Gradle

1. Otwórz **build.gradle (Module: app)**.
2. W sekcji `dependencies` sectiondodaj następujące wiersze dla bibliotek Retrofit:

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

Zauważ, że numery wersji są zdefiniowane osobno w pliku Gradle projektu. Pierwsza zależność dotyczy samej biblioteki Retrofit 2, a druga zależy od konwertera skalarnego Retrofit. Ten konwerter umożliwia Retrofit zwrócić wynik JSON jako ciąg znaków. Dwie biblioteki współpracują ze sobą.

3. Kliknij opcję Synchronizuj teraz, aby odbudować projekt z nowymi zależnościami.

Step 2: Zaimplementuj MarsApiService

Retrofit tworzy sieciowy interfejs API dla aplikacji na podstawie zawartości usługi internetowej. Pobiera dane z usługi internetowej i kieruje je przez osobną bibliotekę konwertera, która wie, jak dekodować dane i zwrócić je w postaci użytecznych obiektów. Retrofit obejmuje wbudowaną obsługę popularnych formatów danych internetowych, takich jak XML i JSON. Retrofit ostatecznie tworzy dla ciebie większość warstwy sieci, w tym kluczowe szczegóły, takie jak uruchamianie żądań w wątkach w tle.

Klasa `MarsApiService` przechowuje warstwę sieciową dla aplikacji; to jest interfejs API, którego Twój `ViewModel` będzie używał do komunikacji z usługą internetową. Jest to klasa, w której zaimplementujesz interfejs API usługi Retrofit.

1. Otwórz `app/java/network/MarsApiService.kt`. W tej chwili plik zawiera tylko jedną rzecz: stałą dla podstawowego adresu URL usługi internetowej.

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

2. Tuż poniżej tej stałej użyj konstruktora Retrofit, aby utworzyć obiekt Retrofit. Na żądanie importuj

   `retrofit2.Retrofit` i
   `retrofit2.converter.scalars.ScalarsConverterFactory`.

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

wymaga co najmniej dwóch dostępnych elementów do zbudowania interfejsu API usług sieciowych: podstawowego identyfikatora URI dla usługi internetowej i fabryki konwerterów. Konwerter mówi Retrofit, co zrobić z danymi, które odzyskuje z serwisu internetowego. W takim przypadku chcesz, aby Modernizacja pobierała odpowiedź JSON z usługi sieci Web i

zwracała ją jako `String`. Retrofit ma `ScalarsConverter` , który obsługuje ciągi znaków i inne prymitywne typy, więc wywołujesz `addConverterFactory()` w kreatorze z instancją `ScalarsConverterFactory`. Na koniec wywołujesz `build()` aby utworzyć obiekt Retrofit.

3. Tuż pod odwołaniem do konstruktora Retrofit zdefiniuj interfejs, który definiuje sposób, w jaki Retrofit komunikuje się z serwerem WWW za pomocą żądań HTTP. Zaimportuj `retrofit2.http.GET` i `retrofit2.Call`.

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

W tej chwili celem jest uzyskanie ciągu odpowiedzi JSON z usługi sieci web i potrzebujesz tylko jednej metody: `getProperties()`.Aby powiedzieć Retrofit, co powinna zrobić ta metoda, użyj adnotacji `@GET` i określ ścieżkę lub punkt końcowy dla tej metody usługi sieci Web. W tym przypadku punkt końcowy nazywa się `realestate`. Po wywołaniu metody `getProperties()` Retrofit dołącza `realestate` punktu końcowego do podstawowego adresu URL (zdefiniowanego w kreatorze Retrofit) i tworzy obiekt `Call` Ten obiekt `Call` służy do uruchomienia żądania.

4. Poniżej interfejsu `MarsApiService` zdefiniuj obiekt publiczny o nazwie `MarsApi` aby zainicjować usługę.

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

Metoda Retrofit `create()` tworzy samą usługę Retrofit z interfejsem `MarsApiService` Ponieważ to połączenie jest drogie, a aplikacja potrzebuje tylko jednej instancji usługi Retrofit , możesz udostępnić usługę pozostałej części aplikacji za pomocą obiektu publicznego o nazwie `MarsApi`, i leniwie zainicjować (lazily) tam usługę modernizacji. Teraz, gdy cała konfiguracja jest zakończona, za każdym razem, gdy aplikacja wywoła `MarsApi.retrofitService`, otrzyma pojedynczy obiekt Retrofit, który implementuje `MarsApiService`.

Step 3: Połącz sie do usługi internetowej w OverviewViewModel

1. Otwórz `app/java/overview/OverviewViewModel.kt`. Przewiń w dół do metody`getMarsRealEstateProperties()`.

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

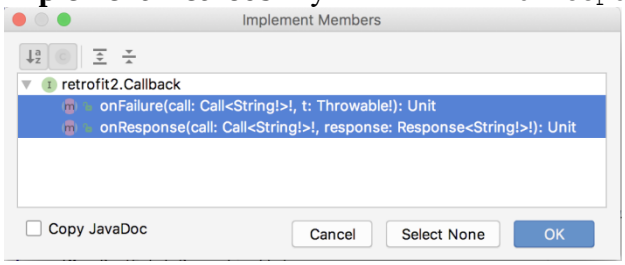
Jest to metoda, w której wywołasz usługę Retrofit i obsłużysz zwrócony ciąg JSON. W tej chwili jest tylko łańcuch zastępczy (placeholder string) dla odpowiedzi.

2. Usuń wiersz placeholder , który ustawia odpowiedź na "Set the Mars API Response here!"
3. Wewnątrz `getMarsRealEstateProperties()`,dodaj kod pokazany poniżej. Na żądanie importuj `retrofit2.Callback` i `com.example.android.marsrealestate.network.MarsApi`.

   Metoda `MarsApi.retrofitService.getProperties()` zwraca obiekt `Call` Następnie możesz wywołać `enqueue()` na tym obiekcie, aby uruchomić żądanie sieciowe w wątku w tle.

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

4. Kliknij tekst `object`, który jest podkreślony na czerwono. Wybierz **Code > Implement methods** Wybierz zarówno `onResponse()` jak i `onFailure()` z listy.



   Android Studio dodaje kod do wykonania TODO w każdej metodzie:

```
override fun onFailure(call: Call<String>, t: Throwable) {
        TODO("not implemented")
}

override fun onResponse(call: Call<String>,
    response: Response<String>) {
        TODO("not implemented")
}
```

5. W `onFailure()`,usuń TODO i ustaw odpowiedź `_response` na komunikat o błędzie, jak pokazano poniżej. `_response` jest łańcuchem `LiveData` , który określa, co jest wyświetlane w widoku tekstowym. Każdy stan musi zaktualizować `_response` `LiveData`.

   Wywołanie zwrotne `onFailure()` jest wywoływane, gdy odpowiedź usługi internetowej nie powiedzie się. Dla tej odpowiedzi ustaw wartość `_response` na `"Failure: "` połączoną z komunikatem z argumentu `Throwable`.

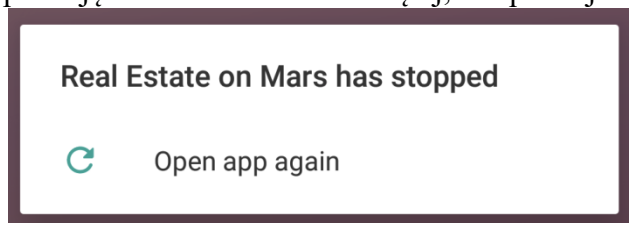
```
override fun onFailure(call: Call<String>, t: Throwable) {
    _response.value = "Failure: " + t.message
}
```

6. W `onResponse()`,usuń TODO i ustaw odpowiedź `_response`. Wywołanie zwrotne `onResponse()` jest wywoływane, gdy żądanie zakończy się powodzeniem, a usługa internetowa zwróci odpowiedź.

```
override fun onResponse(call: Call<String>,
```

```
    response: Response<String>) {
        _response.value = response.body()
}
```

Step 4: Zdefiniuj uprawnienia internetowe

1. Skompiluj i uruchom aplikację MarsRealEstate. Pamiętaj, że aplikacja zamyka się natychmiast z błędem. 

2. Kliknij kartę **Logcat** w Android Studio i znajdź błąd w dzienniku, który zaczyna się od następującej linii:

```
Process: com.example.android.marsrealestate, PID: 10646
java.lang.SecurityException: Permission denied (missing INTERNET
permission?)
```

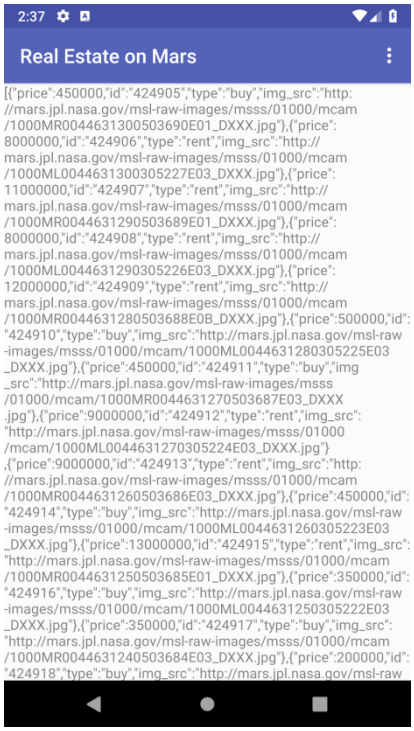
Komunikat o błędzie informuje, że w aplikacji może brakować uprawnienia (`INTERNET` permission). Łączenie się z Internetem budzi obawy związane z bezpieczeństwem, dlatego aplikacje domyślnie nie mają połączenia z Internetem. Musisz wyraźnie powiedzieć Androidowi, że aplikacja potrzebuje dostępu do Internetu.

3. Otwórz `app/manifests/AndroidManifest.xml`. Dodaj ten wiersz tuż przed znacznikiem `<application>`:

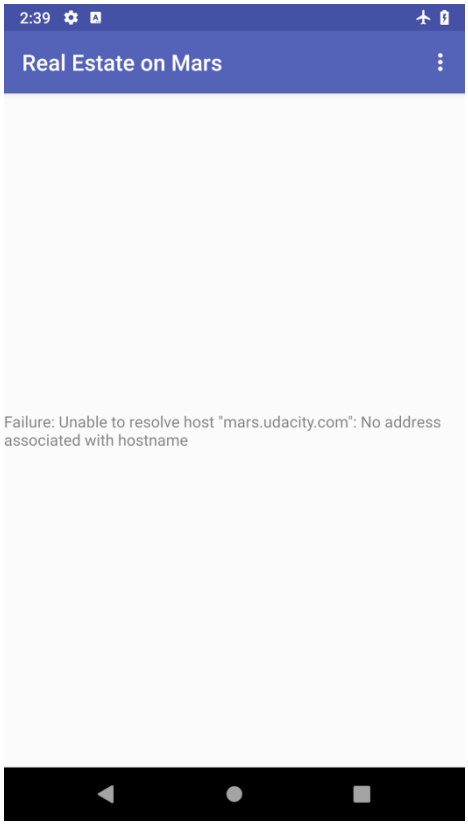
```
<uses-permission android:name="android.permission.INTERNET" />
```

4. Skompiluj i uruchom aplikację ponownie. Jeśli wszystko działa poprawnie z połączeniem internetowym, zobaczysz tekst JSON zawierający dane nieruchomości na

Marsie.



5. Stuknij przycisk Wstecz w urządzeniu lub emulatorze, aby zamknąć aplikację.
6. Przełącz urządzenie lub emulator w tryb samolotowy, a następnie ponownie otwórz aplikację z menu Ostatnie lub uruchom ponownie aplikację z Android Studio.



7. Ponownie wyłącz tryb samolotowy.

5. Zadanie: przeanalizuj odpowiedź JSON za pomocą Moshi

Teraz otrzymujesz odpowiedź JSON z serwisu internetowego Mars, co jest świetnym początkiem. Ale tak naprawdę potrzebujesz obiektów Kotlin, a nie dużego ciągu JSON. Istnieje biblioteka o nazwie Moshi, która jest parserem JSON dla Androida, który konwertuje ciąg JSON na obiekty Kotlin. Retrofit ma konwerter współpracujący z Moshi, więc jest to świetna biblioteka do twoich celów.

Step 1: Dodaj zależności biblioteki Moshi

1. Otwórz **build.gradle (Module: app)**.
2. W sekcji zależności dodaj poniższy kod, aby uwzględnić zależności Moshi. Podobnie jak w przypadku Retrofit, `$version_moshi` jest zdefiniowany osobno w pliku Gradle na poziomie projektu. Zależności te dodają obsługę podstawowej biblioteki Moshi JSON oraz obsługi Kotlin przez Moshi.

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

3. Zlokalizuj wiersz dla Retrofit skalarnego konwertera w bloku `dependencies`:

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

4. Zmień tę linię, aby użyć `converter-moshi`:

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

5. Kliknij opcję Synchronizuj teraz, aby odbudować projekt z nowymi zależnościami.

Note: Projekt może wyświetlać błędy kompilatora związane z usuniętą zależnością skalarną Retrofit. Naprawiasz je w następnych krokach.

Step 2: Zaimplementuj klasę danych MarsProperty

Przykładowy wpis odpowiedzi JSON uzyskany z usługi sieci Web wygląda mniej więcej tak:

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

Pokazana powyżej odpowiedź JSON jest tablicą, która jest wskazana w nawiasach kwadratowych. Tablica zawiera obiekty JSON, które są otoczone nawiasami klamrowymi. Każdy obiekt zawiera zestaw par nazwa-wartość, oddzielonych dwukropkami. Nazwy są otoczone cudzysłowami. Wartości mogą być liczbami lub ciągami, a ciągi są również otoczone cudzysłowami. Na przykład cena tej właściwości wynosi 450 000 USD, a `img_src` to adres URL, który jest lokalizacją pliku obrazu na serwerze.

W powyższym przykładzie zauważ, że każdy wpis właściwości Marsa ma następujące pary kluczy JSON i wartości::

- `price`: cena nieruchomości Marsa jako liczba.
- `id`: identyfikator właściwości jako string.
- `type`: `"rent"` lub `"buy"`.
- `img_src`: adres URL obrazu jako ciąg.

Moshi analizuje te dane JSON i konwertuje je na obiekty Kotlin. Aby to zrobić, musi mieć klasę danych Kotlin do przechowywania przeanalizowanych wyników, więc następnym krokiem jest utworzenie tej klasy.

1. Otwórz `app/java/network/MarsProperty.kt`.
2. Zastąp istniejącą definicję klasy `MarsProperty` następującym kodem:

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

Zauważ, że każda ze zmiennych w klasie MarsProperty odpowiada nazwie klucza w obiekcie JSON. Aby dopasować typy w JSON, używasz obiektów String dla wszystkich wartości oprócz `price`, który jest `Double`. `Double` może być używany do reprezentowania dowolnych danych liczbowych z JSON..

Kiedy Moshi analizuje JSON, dopasowuje klucze według nazwy i wypełnia obiekty danych odpowiednimi wartościami.

3. Zastąp wiersz klucza `img_src` wierszem pokazanym poniżej. Na żądanie zaimportuj plik `com.squareup.moshi.Json`.

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

Czasami nazwy kluczy w odpowiedzi JSON mogą powodować mylące właściwości Kotlin lub mogą nie pasować do twojego stylu kodowania - na przykład w pliku JSON klucz `img_src` używa podkreślnika, podczas gdy właściwości Kotlin zwykle używają wielkich i małych liter ("camel case").

Aby użyć nazw zmiennych w klasie danych, które różnią się od nazw kluczy w odpowiedzi JSON, użyj adnotacji `@Json` W tym przykładzie nazwa zmiennej w klasie danych to `imgSrcUrl`. Zmienna jest odwzorowana na atrybut JSON `img_src` przy użyciu `@Json(name = "img_src")`.

Step 3: Update Zaktualizuj MarsApiService i OverviewViewModel

Mając klasę danych `MarsProperty` możesz teraz zaktualizować interfejs network API i `ViewModel`, aby uwzględnić dane Moshi.

1. Otwórz `network/MarsApiService.kt`. Mogą występować błędy braku klasy dla `ScalarsConverterFactory`. Jest to spowodowane zmianą zależności Retrofit wprowadzoną w kroku 1. Wkrótce naprawisz te błędy.
2. W górnej części pliku, tuż przed konstruktorem Retrofit, dodaj następujący kod, aby utworzyć instancję Moshi. Na żądanie zaimportuj `com.squareup.moshi.Moshi` i `com.squareup.moshi.kotlin.reflect.KotlinJsonAdapterFactory`.

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

Podobnie jak w przypadku Retrofit, tutaj tworzysz obiekt moshi za pomocą konstruktora Moshi. Aby adnotacje Moshi działały poprawnie z Kotlinem, dodaj `KotlinJsonAdapterFactory`, a następnie wywołaj `build()`.

3. Zmień konstruktora Retrofit, aby używał `MoshiConverterFactory` zamiast `ScalarConverterFactory`, i przekaż utworzoną instancję `moshi` Na żądanie importuj `retrofit2.converter.moshi.MoshiConverterFactory`.

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

4. Usuń również import `ScalarConverterFactory`.

Kod do usunięcia:

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

5. Zaktualizuj interfejs `MarsApiService` , aby Retrofit zwracała listę obiektów `MarsProperty` zamiast zwracać `Call<String>`.

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

6. Otwórz `OverviewViewModel.kt`. Przewiń w dół do wywołania `getProperties().enqueue()` w metodzie `getMarsRealEstateProperties()`.
7. Zmień argument na `enqueue()` z `Callback<String>` na `Callback<List<MarsProperty>>`. Na żądanie zaimportuj `com.example.android.marsrealestate.network.MarsProperty`

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

8. W funkcji `onFailure()`,zmień argument z `Call<String>` na `Call<List<MarsProperty>>`:

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

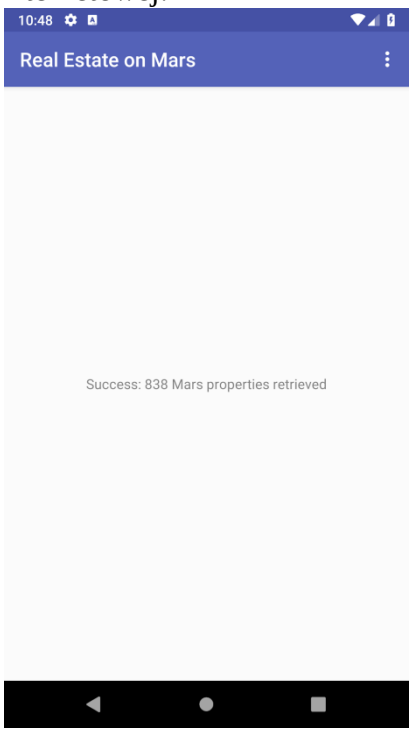
9. Wprowadź tę samą zmianę w obu argumentach funkcji `onResponse()`:

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

10. W ciele `onResponse()`,zamień istniejące przypisanie `_response.value` na przypisanie pokazane poniżej. Ponieważ `response.body()` jest teraz listą obiektów `MarsProperty` , rozmiar tej listy jest liczbą analizowanych nieruchomości. Ten komunikat odpowiedzi wyświetla tę liczbę właściwości:

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

11. Upewnij się, że tryb samolotowy jest wyłączony. Skompiluj i uruchom aplikację. Tym razem wiadomość powinna pokazywać liczbę właściwości zwróconych z usługi internetowej:



6. Zadanie: użyj coroutines z Retrofit

jest uruchomiona, ale używa wywołania zwrotnego z dwiema metodami wywołania zwrotnego, które trzeba zaimplementować. Jedna metoda obsługuje sukces, druga obsługuje awarię, a wynik awarii zgłasza wyjątki. Twój kod byłby bardziej wydajny i łatwiejszy do odczytania, gdybyś mógł używać coroutines z obsługą wyjątków, zamiast wywoływania zwrotnego. Dogodnie, Retrofit ma bibliotekę, która integruje coroutines.

W tym zadaniu przekształcasz usługę sieciową i moduł ViewModel, aby korzystały z coroutines.

Step 1: Dodaj zależności coroutine

1. Otwórz **build.gradle (Module: app)**.
2. W sekcji zależności dodaj obsługę podstawowych bibliotek coroutine Kotlin i biblioteki coroutine Retrofit:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
core:$version_kotlin_coroutines"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
android:$version_kotlin_coroutines"

implementation "com.jakewharton.retrofit:retrofit2-kotlin-coroutines-
adapter:$version_retrofit_coroutines_adapter"
```

3. Kliknij opcję Synchronizuj teraz, aby odbudować projekt z nowymi zależnościami.

Step 2: Zaktualizuj MarsApiService i OverviewViewModel

1. W `MarsApiService.kt`, zaktualizuj konstruktora Retrofit, aby używał `CoroutineCallAdapterFactory`. Pełny konstruktor wygląda teraz tak:

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

Adaptery połączeń dodają funkcję Retrofit do tworzenia interfejsów API, które zwracają coś innego niż domyślna klasa `Call` W tym przypadku `CoroutineCallAdapterFactory` pozwala nam zastąpić obiekt `Call` zwracany przez `getProperties()` obiektem `Deferred`.

2. W metodzie `getProperties()` zmień `Call<List<MarsProperty>>` na `Deferred<List<MarsProperty>>`. Zaimportuj `kotlinx.coroutines.Deferred` na żądanie. Pełna metoda `getProperties()` wygląda następująco:

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

Odroczony interfejs `Deferred` interface definiuje coroutine zadanie, które zwraca wartość wyniku (`Deferred` dziedziczy z `Job`). `Deferred` interfejs zawiera metodę o nazwie `await()`,, która powoduje, że kod czeka bez blokowania, aż wartość będzie gotowa, a następnie ta wartość zostanie zwrócona.

3. Otwórz `OverviewViewModel.kt`. Tuż przed blokiem `init` dodaj zadanie coroutine:

```
private var viewModelJob = Job()
```

4. Utwórz zakres coroutine dla tego nowego zadania za pomocą głównego dyspozytora (main dispatcher):

```
private val coroutineScope = CoroutineScope(
```

```
    viewModelJob + Dispatchers.Main )
```

`Dispatchers.Main` Główny dyspozytor używa wątku interfejsu użytkownika do swojej pracy. Ponieważ Retrofit wykonuje całą pracę na wątku w tle, nie ma powodu, aby używać innego wątku dla zakresu. Pozwala to na łatwą aktualizację wartości `MutableLiveData` po otrzymaniu wyniku.

5. Usuń cały kod z `getMarsRealEstateProperties()`.Będziesz tutaj używał coroutines zamiast wywołania `enqueue()` oraz wywołań zwrotnych `onFailure()` i `onResponse()`.
6. Wewnątrz `getMarsRealEstateProperties()`,uruchom coroutine:

```
coroutineScope.launch {

}
```

Aby użyć odroczonego obiektu `Deferred` object , który Retrofit zwraca do zadania sieciowego, musisz znajdować się w coroutine, więc tutaj uruchom właśnie utworzoną coroutine. Nadal wykonujesz kod w głównym wątku, ale teraz pozwalasz coroutines zarządzać współbieżnością.

7. Wewnątrz bloku uruchamiania wywołaj `getProperties()` na obiekcie `retrofitService`:

```
var getPropertiesDeferred = MarsApi.retrofitService.getProperties()
```

Wywołanie `getProperties()` z usługi `MarsApi` tworzy i uruchamia połączenie sieciowe w wątku w tle, zwracając obiekt Odroczony dla tego zadania. `Deferred` object.

8. Również wewnątrz bloku uruchamiania dodaj blok `try`/`catch` aby obsłużyć wyjątki:

```
try {

} catch (e: Exception) {

}
```

9. W bloku `try {}` wywołaj funkcję `await()` na obiekcie `Deferred`:

```
var listResult = getPropertiesDeferred.await()
```

Wywołanie `await()` na obiekcie `Deferred` zwraca wynik połączenia sieciowego, gdy wartość jest gotowa. Metoda `await()` jest nieblokująca, więc usługa Mars API pobiera dane z sieci bez blokowania bieżącego wątku - co jest ważne, ponieważ jesteśmy w zakresie wątku interfejsu użytkownika. Po wykonaniu zadania kod będzie kontynuowany od momentu, w którym został przerwany. To jest w ramach `try {}` abyś mógł wychwycić wyjątki.

10. Również w bloku `try {}` po metodzie `await()` zaktualizuj komunikat odpowiedzi dla pomyślnej odpowiedzi:

```
_response.value =
```

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

11. Inside the `catch {}` block, handle the failure response:

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

Pełna metoda `getMarsRealEstateProperties()` wygląda teraz następująco:

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

12. Na dole klasy dodaj wywołanie zwrotne `onCleared()` z tym kodem:

```
override fun onCleared() {
    super.onCleared()
    viewModelJob.cancel()
}
```

Ładowanie danych powinno się zatrzymać, gdy `ViewModel` zostanie zniszczony, ponieważ `OverviewFragment` , który używa tego `ViewModel` zniknie. Aby zatrzymać ładowanie, gdy ViewModel zostanie zniszczony, przesłonisz funkcję `onCleared()` aby anulować zadanie.

13. Skompiluj i uruchom aplikację. Tym razem otrzymujesz taki sam wynik jak w poprzednim zadaniu (raport liczby właściwości), ale z prostszą obsługą kodu i błędów.

Podsumowanie

Usługi sieciowe REST

- A *web service* is a service on the internet that enables your app to make requests and get data back.
- Common web services use a REST architecture. Web services that offer REST architecture are known as *RESTful* services. RESTful web services are built using standard web components and protocols.
- You make a request to a REST web service in a standardized way, via URIs.
- To use a web service, an app must establish a network connection and communicate with the service. Then the app must receive and parse response data into a format the app can use.
- The Retrofit library is a client library that enables your app to make requests to a REST web service.

- Use converters to tell Retrofit what do with data it sends to the web service and gets back from the web service. For example, the `ScalarsConverter` converter treats the web service data as a `String` or other primitive.
- To enable your app to make connections to the internet, add the `"android.permission.INTERNET"` permission in the Android manifest.

JSON parsing

- The response from a web service is often formatted in JSON, a common interchange format for representing structured data.
- A JSON object is a collection of key-value pairs. This collection is sometimes called a *dictionary*, a *hash map*, or an *associative array*.
- A collection of JSON objects is a JSON array. You get a JSON array as a response from a web service.
- The keys in a key-value pair are surrounded by quotes. The values can be numbers or strings. Strings are also surrounded by quotes.
- The Moshi library is Android JSON parser that converts a JSON string into Kotlin objects. Retrofit has a converter that works with Moshi.
- Moshi matches the keys in a JSON response with properties in a data object that have the same name.
- To use a different property name for a key, annotate that property with the `@Json` annotation and the JSON key name.

Retrofit and coroutines

- Call adapters let Retrofit create APIs that return something other than the default `Call` class. Use the `CoroutineCallAdapterFactory` class to replace the `Call` with a coroutine `Deferred`.
- Use the `await()` method on the `Deferred` object to cause your coroutine code to wait without blocking until the value is ready, and then the value is returned.