Jednostka 4: Połącz się z Internetem

Napisz współprogramy dla złożonego kodu i dowiedz się o HTTP i REST, aby uzyskać dane z Internetu.

Spis treści

Współprogramy

Napisz kod dla bardziej zaawansowanych i złożonych aplikacji na Androida.

Wprowadzenie do współprogramów

1. Zanim zaczniesz

Posiadanie responsywnego interfejsu użytkownika jest niezbędnym elementem świetnej aplikacji. Chociaż mogłeś przyjąć to za pewnik w aplikacjach, które do tej pory zbudowałeś, gdy zaczynasz dodawać bardziej zaawansowane funkcje, takie jak funkcje sieciowe lub bazy danych, napisanie kodu, który jest zarówno funkcjonalny, jak i wydajny, może być coraz trudniejsze. Poniższy przykład ilustruje, co może się stać, jeśli długotrwałe zadania, takie jak pobieranie obrazów z Internetu, nie są obsługiwane prawidłowo. Podczas gdy funkcjonalność obrazu działa, przewijanie jest niestabilne, przez co interfejs użytkownika wygląda na nie reagujący (i nieprofesjonalny!).

Aby uniknąć problemów z powyższą aplikacją, musisz dowiedzieć się trochę o czymś, co nazywa się wątkami. Wątek jest trochę abstrakcyjnym pojęciem, ale możesz o nim myśleć jako o pojedynczej ścieżce wykonywania kodu w Twojej aplikacji. Każdy wiersz kodu, który piszesz, jest instrukcją, która ma zostać wykonana w tej samej kolejności w tym samym wątku.

Pracowałeś już z wątkami w Androidzie. Każda aplikacja na Androida ma domyślny „główny” wątek. To jest (zazwyczaj) wątek interfejsu użytkownika. Cały kod, który do tej pory napisałeś, znajduje się w głównym wątku. Każda instrukcja (tj. linia kodu) czeka na zakończenie poprzedniej przed wykonaniem następnej linii.

Jednak w uruchomionej aplikacji jest więcej wątków oprócz głównego wątku. Za kulisami procesor w rzeczywistości nie działa z oddzielnymi wątkami, ale raczej przełącza się między różnymi seriami instrukcji, aby uzyskać wrażenie wielozadaniowości. Wątek to abstrakcja, której można użyć podczas pisania kodu, aby określić, którą ścieżkę wykonania powinna przejść każda instrukcja. Praca z wątkami innymi niż wątek główny umożliwia aplikacji wykonywanie złożonych zadań, takich jak pobieranie obrazów, w tle, podczas gdy interfejs użytkownika aplikacji pozostaje responsywny. Nazywa się to kodem współbieżnym lub po prostu współbieżnością.

W tym laboratorium dowiesz się o wątkach i jak używać funkcji Kotlin zwanej współprogramami do pisania jasnego, nieblokującego współbieżnego kodu.

Warunki wstępne

- Znajomość podstawowych pojęć programowania Kotlin, w tym pętli i funkcji, nauczanych w ścieżce 1: Wprowadzenie do Kotlina
- Jak używać funkcji lambda w Kotlinie, nauczonych w Ścieżce 3: Kolekcje w Kotlinie

Czego się nauczysz

- Czym jest współbieżność i dlaczego jest ważna
- Jak używać współprogramów i wątków do pisania nieblokującego współbieżnego kodu?
- Jak uzyskać dostęp do głównego wątku, aby bezpiecznie wykonywać aktualizacje interfejsu użytkownika podczas wykonywania zadań w tle?
- Jak i kiedy używać różnych wzorców współbieżności (zakres/dyspozytorzy/odroczone)
- Jak napisać kod, który współdziała z zasobami sieciowymi

Co zbudujesz

- W tym ćwiczeniu z kodowania napiszesz kilka małych programów do zbadania pracy z wątkami i współprogramami w Kotlin

Czego potrzebujesz

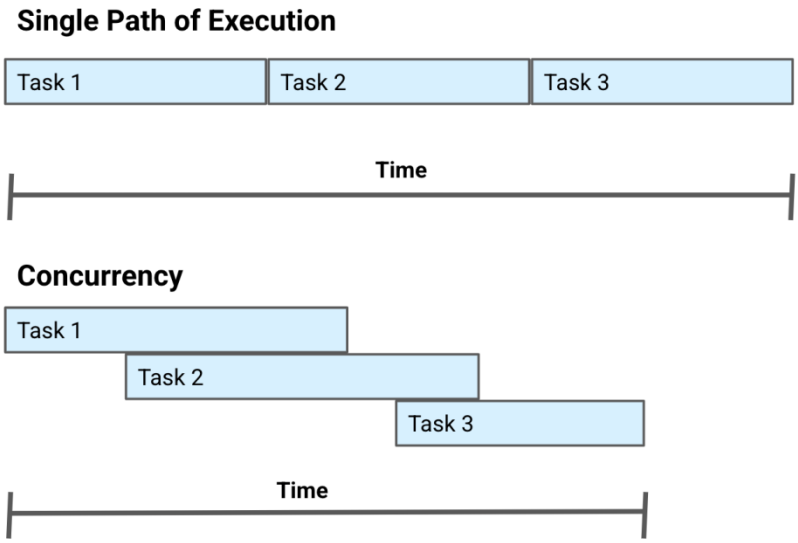
- Komputer z nowoczesną przeglądarką internetową, np. najnowszą wersją Chrome
- Dostęp do Internetu na Twoim komputerze

2. Wstęp

Wielowątkowość i współbieżność

Do tej pory traktowaliśmy aplikację na Androida jako program z jedną ścieżką wykonania. Dzięki tej pojedynczej ścieżce wykonania możesz wiele zrobić, ale wraz z rozwojem aplikacji musisz pomyśleć o współbieżności.

Współbieżność umożliwia wykonywanie wielu jednostek kodu poza kolejnością lub pozornie równolegle, co pozwala na bardziej efektywne wykorzystanie zasobów. System operacyjny może wykorzystywać charakterystykę systemu, język programowania i jednostkę współbieżności do zarządzania wielozadaniowością.



Dlaczego musisz używać współbieżności? Ponieważ Twoja aplikacja staje się coraz bardziej złożona, ważne jest, aby kod nie blokował. Oznacza to, że wykonanie długotrwałego zadania, takiego jak żądanie sieciowe, nie zatrzyma wykonywania innych czynności w Twojej aplikacji. Nieprawidłowe zaimplementowanie współbieżności może sprawić, że Twoja aplikacja nie będzie odpowiadać użytkownikom.

Przyjrzysz się kilku przykładom demonstrującym programowanie współbieżne w Kotlinie. Wszystkie przykłady można uruchomić w Kotlin Playground:

https://developer.android.com/training/kotlinplayground

Wątek to najmniejsza jednostka kodu, którą można zaplanować i uruchomić w ramach programu. Oto mały przykład, w którym możemy uruchomić współbieżny kod.

Możesz stworzyć prosty wątek, podając lambdę. Wypróbuj następujące rozwiązania na placu zabaw.

```
fun main() {
    val thread = Thread {
        println("${Thread.currentThread()} has run.")
    }
    thread.start()
}
```

Wątek nie jest wykonywany, dopóki funkcja nie osiągnie `start()`wywołania funkcji. Wynik powinien wyglądać mniej więcej tak.

```
Thread[Thread-0,5,main] has run.
```

Zauważ, że `currentThread()`zwraca `Thread`instancję, która jest konwertowana na jej reprezentację w postaci ciągu, która zwraca nazwę wątku, priorytet i grupę wątków. Powyższe dane wyjściowe mogą być nieco inne.

Tworzenie i uruchamianie wielu wątków

Aby zademonstrować prostą współbieżność, utwórzmy kilka wątków do wykonania. Kod utworzy 3 wątki drukujące linię informacyjną z poprzedniego przykładu.

```
fun main() {
  val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
  repeat(3) {
     Thread {
        println("${Thread.currentThread()} has started")
        for (i in states) {
           println("${Thread.currentThread()} - $i")
           Thread.sleep(50)
        }
     }.start()
  }
}
```

Wyjście na placu zabaw:

Thread[Thread-2,5,main] has started Thread[Thread-2,5,main] - Starting Thread[Thread-0,5,main] - Doing Task 1 Thread[Thread-1,5,main] - Doing Task 1 Thread[Thread-2,5,main] - Doing Task 1 Thread[Thread-0,5,main] - Doing Task 2 Thread[Thread-1,5,main] - Doing Task 2 Thread[Thread-2,5,main] - Doing Task 2 Thread[Thread-0,5,main] - Ending Thread[Thread-2,5,main] - Ending Thread[Thread-1,5,main] - Ending
Thread[Thread-0,5,main] has started
Thread[Thread-0,5,main] - Starting
Thread[Thread-1,5,main] has started
Thread[Thread-1,5,main] - Starting

Wyjście w AS (konsola):

Thread[Thread-0,5,main] has started
Thread[Thread-1,5,main] has started
Thread[Thread-2,5,main] has started
Thread[Thread-1,5,main] - Starting
Thread[Thread-0,5,main] - Starting
Thread[Thread-2,5,main] - Starting
Thread[Thread-1,5,main] - Doing Task 1
Thread[Thread-0,5,main] - Doing Task 1
Thread[Thread-2,5,main] - Doing Task 1
Thread[Thread-0,5,main] - Doing Task 2
Thread[Thread-1,5,main] - Doing Task 2
Thread[Thread-2,5,main] - Doing Task 2
Thread[Thread-0,5,main] - Ending
Thread[Thread-2,5,main] - Ending
Thread[Thread-1,5,main] - Ending

Uruchom kod kilka razy. Zobaczysz zróżnicowane wyniki. Czasami wątki wydają się działać w sekwencji, a innym razem treść będzie przeplatana.

Uwaga: ta niezmienność jest spowodowana sposobem wykonywania wątków. Harmonogram przydziela wycinek czasu do każdego wątku i albo kończy się w określonym przedziale czasu, albo jest zawieszony do czasu otrzymania kolejnego wycinka czasu.

3. Wyzwania z wątkami

Korzystanie z wątków to prosty sposób na rozpoczęcie pracy z wieloma zadaniami i współbieżnością, ale nie jest bezproblemowy. Gdy używasz `Thread`bezpośrednio w kodzie, może pojawić się wiele problemów.

Wątki wymagają dużo zasobów

Tworzenie, przełączanie i zarządzanie wątkami zajmuje zasoby systemowe i czas, ograniczając ilość wątków, którymi można zarządzać jednocześnie. Koszty stworzenia naprawdę mogą się sumować.

Podczas gdy uruchomiona aplikacja będzie miała wiele wątków, każda aplikacja będzie miała jeden dedykowany wątek, w szczególności odpowiedzialny za interfejs użytkownika aplikacji. Ten wątek jest często nazywany wątkiem głównym lub wątkiem interfejsu użytkownika.

Uwaga: W niektórych przypadkach wątek interfejsu użytkownika i wątek główny mogą się różnić.

Ponieważ ten wątek jest odpowiedzialny za uruchamianie interfejsu użytkownika aplikacji, ważne jest, aby główny wątek działał wydajnie, aby aplikacja działała płynnie. Wszelkie długotrwałe zadania zablokują go do czasu zakończenia i spowodują, że aplikacja przestanie odpowiadać.

System operacyjny robi wiele, aby zachować responsywność dla użytkownika. Obecne telefony próbują aktualizować interfejs użytkownika od 60 do 120 razy na sekundę (co najmniej 60). Na przygotowanie i narysowanie interfejsu użytkownika jest krótki, skończony czas (przy 60 klatkach na sekundę, każda aktualizacja ekranu powinna zająć 16 ms lub mniej). Android porzuci ramki lub przerwie próbę ukończenia pojedynczego cyklu aktualizacji, aby spróbować nadrobić zaległości. Niektóre ramki spadają, a fluktuacje są normalne, ale zbyt wiele sprawi, że Twoja aplikacja przestanie odpowiadać.

Warunki rasowe i nieprzewidywalne zachowanie

Jak wspomniano, wątek jest abstrakcją tego, jak procesor wydaje się obsługiwać wiele zadań jednocześnie. Gdy procesor przełącza się między zestawami instrukcji w różnych wątkach, dokładny czas wykonania wątku i wstrzymania wątku jest poza twoją kontrolą. Podczas bezpośredniej pracy z wątkami nie zawsze można oczekiwać przewidywalnych wyników.

Na przykład poniższy kod używa prostej pętli do zliczania od 1 do 50, ale w tym przypadku nowy wątek jest tworzony dla każdego zwiększenia liczby. Zastanów się, jak będzie wyglądał wynik, a następnie uruchom kod kilka razy.

```
fun main() {
  var count = 0
  for (i in 1..50) {
     Thread {
        count += 1
```

```
            println("Thread: $i count: $count")
        }.start()
    }
}
```

Czy wynik był taki, jakiego się spodziewałeś? Czy za każdym razem było tak samo? Oto przykładowe dane wyjściowe, które otrzymaliśmy.

```
Thread: 50 count: 49 Thread: 43 count: 50 Thread: 1 count: 1
Thread: 2 count: 2
Thread: 3 count: 3
Thread: 4 count: 4
Thread: 5 count: 5
Thread: 6 count: 6
Thread: 7 count: 7
Thread: 8 count: 8
Thread: 9 count: 9
Thread: 10 count: 10
Thread: 11 count: 11
Thread: 12 count: 12
Thread: 13 count: 13
Thread: 14 count: 14
Thread: 15 count: 15
Thread: 16 count: 16
Thread: 17 count: 17
Thread: 18 count: 18
Thread: 19 count: 19
Thread: 20 count: 20
Thread: 21 count: 21
Thread: 23 count: 22
Thread: 22 count: 23
Thread: 24 count: 24
Thread: 25 count: 25
Thread: 26 count: 26
Thread: 27 count: 27
Thread: 30 count: 28
Thread: 28 count: 29
Thread: 29 count: 41
Thread: 40 count: 41
Thread: 39 count: 41
Thread: 41 count: 41
Thread: 38 count: 41
Thread: 37 count: 41
Thread: 35 count: 41
Thread: 33 count: 41
Thread: 36 count: 41
```

```
Thread: 34 count: 41
Thread: 31 count: 41
Thread: 32 count: 41
Thread: 44 count: 42
Thread: 46 count: 43
Thread: 45 count: 44
Thread: 47 count: 45
Thread: 48 count: 46
Thread: 42 count: 47
Thread: 49 count: 48
```

W przeciwieństwie do tego, co mówi kod, wygląda na to, że ostatni wątek został wykonany jako pierwszy, a niektóre inne wątki zostały wykonane niewłaściwie. Jeśli spojrzysz na „liczbę" niektórych iteracji, zauważysz, że pozostaje niezmieniona po wielu wątkach. Co jeszcze dziwniejsze, liczba osiąga 50 w wątku 43, mimo że dane wyjściowe sugerują, że jest to tylko drugi wątek do wykonania. Sądząc po samych wynikach, nie można wiedzieć, jaka jest ostateczna wartość `count`.

To tylko jeden ze sposobów, w jaki wątki mogą prowadzić do nieprzewidywalnego zachowania. Podczas pracy z wieloma wątkami możesz również napotkać coś, co nazywa się sytuacją wyścigu. Dzieje się tak, gdy wiele wątków próbuje jednocześnie uzyskać dostęp do tej samej wartości w pamięci. Warunki wyścigu mogą skutkować trudnymi do odtworzenia, losowo wyglądającymi błędami, które mogą spowodować awarię aplikacji, często w nieprzewidywalny sposób.

Problemy z wydajnością, warunki wyścigu i trudne do odtworzenia błędy to tylko niektóre z powodów, dla których nie zalecamy bezpośredniej pracy z wątkami. Zamiast tego dowiesz się o funkcji w Kotlinie zwanej Coroutines, która pomoże ci pisać współbieżny kod.

4. Współprogramy w Kotlinie

Tworzenie i używanie wątków do zadań w tle ma swoje miejsce bezpośrednio w Androidzie, ale Kotlin oferuje również Coroutines, które zapewniają bardziej elastyczny i łatwiejszy sposób zarządzania współbieżnością.

Współprogramy umożliwiają wielozadaniowość, ale zapewniają inny poziom abstrakcji niż zwykła praca z wątkami. Jedną z kluczowych cech współprogramów jest możliwość przechowywania stanu, dzięki czemu można je zatrzymać i wznowić. Współprogram może zostać wykonany lub nie.

Stan, reprezentowany przez *kontynuacje* , umożliwia częściom kodu sygnalizowanie, kiedy muszą przekazać kontrolę lub czekać, aż inny współprogram zakończy swoją pracę przed wznowieniem. Ten przepływ nazywa się kooperacyjną wielozadaniowością. Implementacja współprogramów w Kotlin dodaje szereg funkcji wspomagających wielozadaniowość. Oprócz kontynuacji, tworzenie współprogramu obejmuje pracę w `Job`, anulowalną jednostkę pracy z cyklem życia, wewnątrz `CoroutineScope`. A `CoroutineScope`to kontekst, który wymusza rekurencyjnie anulowanie i inne reguły swoim dzieciom i ich dzieciom. A `Dispatcher`zarządza wątkiem pomocniczym, którego współprogram będzie używał do jego wykonania, zdejmując odpowiedzialność za to, kiedy i gdzie użyć nowego wątku od programisty.

Stanowisko	Jednostka pracy, którą można anulować, na przykład utworzona za pomocą launch()funkcji.
Współprogram Zakres	Funkcje używane do tworzenia nowych współprogramów, takie jak launch()i async()extend CoroutineScop
Dyspozytor	Określa wątek, którego będzie używać współprogramu. Dyspozytor Mainzawsze będzie uruchamiał współprogramy w głównym wątku, podczas gdy dyspozytorzy, jak Default, IOlub Unconfinedbędą używa innych wątków.

Dowiesz się o nich więcej później, ale Dispatchersjest to jeden ze sposobów, w jaki współprogramy mogą być tak skuteczne. Pozwala to uniknąć kosztu wydajności inicjowania nowych wątków.

Dostosujmy nasze wcześniejsze przykłady do użycia współprogramów.

```
import kotlinx.coroutines.*

fun main() {
   repeat(3) {
      GlobalScope.launch {
         println("Hi from ${Thread.currentThread()}")
      }
   }
}
```

```
Hi from Thread[DefaultDispatcher-worker-2@coroutine#2,5,main]
Hi from Thread[DefaultDispatcher-worker-1@coroutine#1,5,main]
Hi from Thread[DefaultDispatcher-worker-1@coroutine#3`,5,main]
```

Powyższy fragment kodu tworzy trzy współprogramy w zakresie globalnym przy użyciu domyślnego dyspozytora. Pozwala na GlobalScopedziałanie wszystkich zawartych w nim współprogramów tak długo, jak działa aplikacja. Z powodów, o których mówiliśmy, dotyczących głównego wątku, nie jest to zalecane poza przykładowym kodem. Kiedy używasz współprogramów w swoich aplikacjach, będziemy używać innych zakresów.

Funkcja launch()tworzy współprogram z załączonego kodu opakowanego w obiekt Job, który można anulować. launch()jest używany, gdy wartość zwracana nie jest potrzebna poza granicami współprogramu.

Przyjrzyjmy się pełnej sygnaturze, launch()aby zrozumieć kolejną ważną koncepcję we współprogramach.

```
fun CoroutineScope.launch() {
   context: CoroutineContext = EmptyCoroutineContext,
   start: CoroutineStart = CoroutineStart.DEFAULT,
   block: suspend CoroutineScope.() -> Unit
}
```

Za kulisami blok kodu, który przekazałeś do uruchomienia, jest oznaczony suspendsłowem kluczowym. Zawieszenie sygnalizuje, że blok kodu lub funkcji można wstrzymać lub wznowić.

Słowo o blokowaniu run

Następne przykłady będą używać `runBlocking()`, co jak sama nazwa wskazuje, rozpoczyna nowy współprogram i blokuje bieżący wątek aż do zakończenia. Służy głównie do łączenia kodu blokującego i nieblokującego w głównych funkcjach i testach. Nie będziesz go często używać w typowym kodzie Androida.

```
import kotlinx.coroutines.*
import java.time.LocalDateTime
import java.time.format.DateTimeFormatter

val formatter = DateTimeFormatter.ISO_LOCAL_TIME
val time = { formatter.format(LocalDateTime.now()) }

suspend fun getValue(): Double {
    println("entering getValue() at ${time()}")
    delay(3000)
    println("leaving getValue() at ${time()}")
    return Math.random()
}

fun main() {
    runBlocking {
        val num1 = getValue()
        val num2 = getValue()
        println("result of num1 + num2 is ${num1 + num2}")
    }
}
```

`getValue()`zwraca losową liczbę po określonym czasie opóźnienia. Używa `DateTimeFormatter`. Aby zilustrować odpowiednie czasy wejścia i wyjścia. Funkcja main wywołuje `getValue()`dwa razy i zwraca sumę.

```
entering getValue() at 17:44:52.311
leaving getValue() at 17:44:55.319
entering getValue() at 17:44:55.32
leaving getValue() at 17:44:58.32
result of num1 + num2 is 1.4320332550421415
```

Aby zobaczyć to w akcji, zastąp `main()`funkcję (zachowując cały pozostały kod) następującą.

```
fun main() {
    runBlocking {
        val num1 = async { getValue() }
        val num2 = async { getValue() }
        println("result of num1 + num2 is ${num1.await() + num2.await()}")
    }
```

```
}
```

Te dwa wywołania `getValue()` są niezależne i niekoniecznie wymagają współprogramu do zawieszenia. Kotlin ma funkcję asynchroniczną, która jest podobna do uruchamiania. Funkcja `async()` jest zdefiniowana w następujący sposób.

```
fun CoroutineScope.async() {
   context: CoroutineContext = EmptyCoroutineContext,
   start: CoroutineStart = CoroutineStart.DEFAULT,
   block: suspend CoroutineScope.() -> T
}: Deferred<T>
```

Funkcja `async()` zwraca wartość typu `Deferred`. A `Deferred` jest anulowalnym `Job`, który może zawierać odniesienie do przyszłej wartości. Używając a `Deferred`, nadal możesz wywołać funkcję tak, jakby natychmiast zwracała wartość — a `Deferred` służy po prostu jako symbol zastępczy, ponieważ nie możesz być pewien, kiedy asynchroniczne zadanie zwróci. A `Deferred` (nazywane również Promise lub Future w innych językach) gwarantuje, że wartość zostanie zwrócona do tego obiektu w późniejszym czasie. Z drugiej strony zadanie asynchroniczne domyślnie nie będzie blokować ani czekać na wykonanie. Aby zainicjować, że bieżący wiersz kodu musi czekać na wyjście a `Deferred`, możesz go wywołać `await()`. Zwróci surową wartość.

```
entering getValue() at 22:52:25.025
entering getValue() at 22:52:25.03
leaving getValue() at 22:52:28.03
leaving getValue() at 22:52:28.032
result of num1 + num2 is 0.8416379026501276
```

Kiedy oznaczyć funkcje jako zawieszone?

W poprzednim przykładzie mogłeś zauważyć, że `getValue()` funkcja jest również zdefiniowana za pomocą `suspend` słowa kluczowego. Powodem jest to, że wywołuje `delay()`, co jest również `suspend` funkcją. Ilekroć funkcja wywołuje inną `suspend` funkcję, powinna to być również `suspend` funkcja.

Jeśli tak jest, to dlaczego `main()` funkcja w naszym przykładzie nie miałaby być oznaczona `suspend`? `getValue()` W końcu woła .

Niekoniecznie. Funkcja `getValue()` jest faktycznie wywoływana w lambdzie przekazanej do `runBlocking()`. Lambda jest `suspend` funkcją podobną do funkcji przekazywanych do `launch()` i `async()`. Jednak `runBlocking()` samo w sobie nie jest `suspend` funkcją. Funkcja `getValue()` nie jest wywoływana sama w `main()` sobie, ani nie jest `runBlocking()` funkcją `suspend`, więc `main()` nie jest oznaczona `suspend`. Jeśli funkcja nie wywołuje `suspend` funkcji, to nie musi być `suspend` samą funkcją.

5. Ćwicz na własną rękę

Na początku tego ćwiczenia z programowania widziałeś następujący przykład, w którym użyto wielu wątków. Mając wiedzę na temat współprogramów, przepisz kod tak, aby używał współprogramów zamiast `Thread`.

Uwaga: nie musisz edytować `println()`instrukcji, nawet jeśli odwołują się do `Thread`.

```
fun main() {
  val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
  repeat(3) {
     Thread {
        println("${Thread.currentThread()} has started")
        for (i in states) {
           println("${Thread.currentThread()} - $i")
           Thread.sleep(50)
        }
     }.start()
  }
}
```

6. Rozwiązanie do ćwiczeń praktycznych

```
import kotlinx.coroutines.*

fun main() {
  val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
  repeat(3) {
     GlobalScope.launch {
        println("${Thread.currentThread()} has started")
        for (i in states) {
           println("${Thread.currentThread()} - $i")
        }
     }
  }
}
```

7. Podsumowanie

Nauczyłeś się

- Dlaczego współbieżność jest potrzebna
- Czym jest wątek i dlaczego wątki są ważne dla współbieżności
- Jak pisać współbieżny kod w Kotlinie za pomocą współprogramów

- Kiedy i kiedy nie oznaczać funkcji jako „zawieszenia”
- Role CoroutineScope, Job i Dispatcher
- Różnica między odroczonym a oczekiwaniem

8. Dowiedz się więcej

- Wątek
- Współprogramy
- Współprogramy i wydajność aplikacji

Uzyskaj dane z internetu

Pobieraj i wyświetlaj obrazy przez Internet za pomocą HTTP i REST.

Uzyskaj dane z Internetu

1. Zanim zaczniesz

Większość dostępnych na rynku aplikacji na Androida łączy się z Internetem w celu wykonywania niektórych operacji sieciowych. Takie jak pobieranie wiadomości e-mail, wiadomości lub podobnych informacji z serwera zaplecza. Gmail, YouTube i Zdjęcia Google to kilka przykładowych aplikacji, które łączą się z internetem w celu wyświetlania danych użytkownika.

W tym ćwiczeniu z kodowania używasz opracowanych bibliotek open source do budowania warstwy sieciowej i pobierania danych z serwera zaplecza. To znacznie upraszcza pobieranie danych, a także pomaga aplikacji dostosować się do najlepszych praktyk Androida, takich jak wykonywanie operacji na wątku w tle. Zaktualizujesz również interfejs użytkownika aplikacji, jeśli internet będzie wolny lub niedostępny; dzięki temu użytkownik będzie informowany o wszelkich problemach z łącznością sieciową.

Co powinieneś już wiedzieć

- Jak tworzyć i wykorzystywać fragmenty.
- Jak korzystać ze składników architektury Androida `ViewModel`i `LiveData`.
- Jak dodać zależności w pliku Gradle.

Czego się nauczysz

- Czym jest usługa sieciowa REST .
- Korzystanie z biblioteki Retrofit , aby połączyć się z usługą sieciową REST w Internecie i uzyskać odpowiedź.
- Użycie biblioteki Moshi do przetworzenia odpowiedzi JSON na obiekt danych.

Co zrobisz

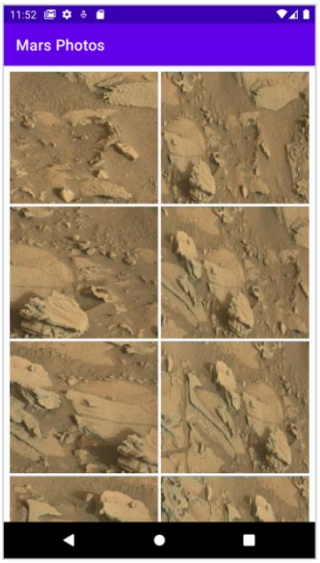
- Zmodyfikuj aplikację startową, aby utworzyć żądanie interfejsu API usługi sieciowej i obsłużyć odpowiedź.
- Zaimplementuj warstwę sieciową dla swojej aplikacji za pomocą biblioteki Retrofit.
- Przeanalizuj odpowiedź JSON z usługi internetowej do obiektów aplikacji za `LiveData`pomocą biblioteki Moshi.
- Skorzystaj z obsługi programu Retrofit dla współprogramów, aby uprościć kod.

Czego potrzebujesz

- Komputer z zainstalowanym Android Studio.
- Kod startowy aplikacji MarsPhotos.

2. Przegląd aplikacji

W tej ścieżce pracujesz z aplikacją startową o nazwie **MarsPhotos** , która wyświetla obrazy powierzchni Marsa. Ta aplikacja łączy się z usługą internetową, aby pobrać i wyświetlić zdjęcia Marsa. Obrazy są prawdziwymi zdjęciami z Marsa uchwyconymi przez łaziki marsjańskie NASA. Poniżej znajduje się zrzut ekranu ostatniej aplikacji, która zawiera siatkę obrazów właściwości miniatur, zbudowaną za pomocą `RecyclerView`.



Uwaga : powyższy zrzut ekranu to zrzut ekranu ostatniej aplikacji, którą zbudujesz na końcu ścieżki, która znajduje się na końcu następnego ćwiczenia z programowania. Ten zrzut ekranu jest pokazany w tym ćwiczeniu z kodowania, aby dać ci lepsze wyobrażenie o ogólnym funkcjonowaniu aplikacji.

Wersja aplikacji, którą tworzysz w tym laboratorium, nie będzie miała dużo wizualnego flashowania: skupia się na warstwie sieciowej aplikacji, aby połączyć się z Internetem i pobrać nieprzetworzone dane właściwości za pomocą usługi internetowej. Aby upewnić się, że dane są prawidłowo pobierane i analizowane, wystarczy wydrukować liczbę zdjęć otrzymanych z serwera zaplecza w widoku tekstowym:

3. Poznaj aplikację startową MarsPhotos

Pobierz kod startowy

To ćwiczenie z programowania zapewnia kod startowy, który można rozszerzyć o funkcje nauczane w tym ćwiczeniu z programowania. Kod startowy może zawierać kod, który jest Ci zarówno znany, jak i nieznany z poprzednich ćwiczeń z programowania. Więcej o nieznanym kodzie dowiesz się w późniejszych ćwiczeniach z programowania.

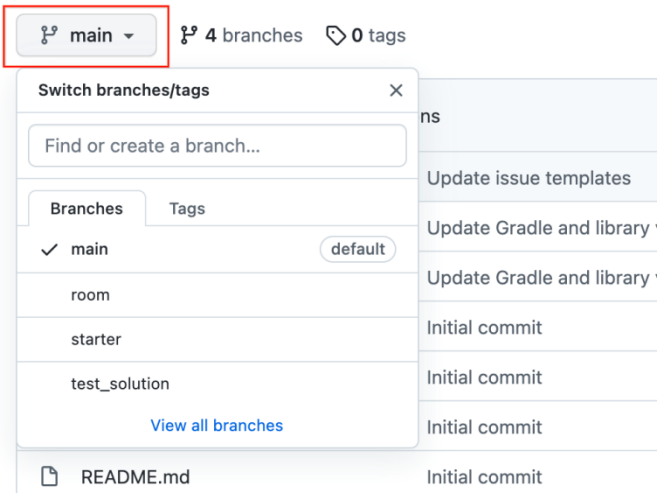
Jeśli używasz kodu startowego z GitHub, pamiętaj, że nazwa folderu to `android-basics-kotlin-mars-photos-app`. Wybierz ten folder podczas otwierania projektu w Android Studio.

Adres URL kodu startowego:

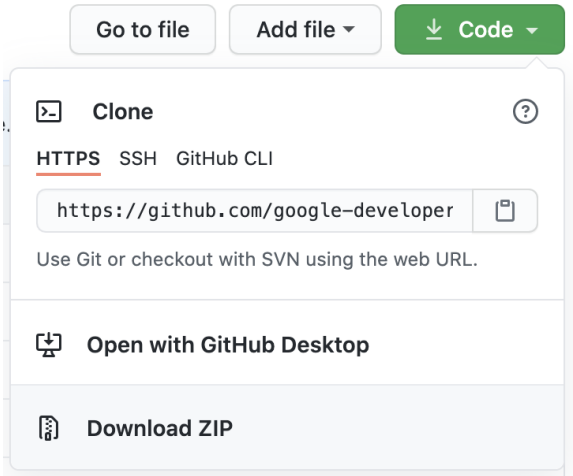
https://github.com/google-developer-training/android-basics-kotlin-mars-photos-app/tree/starter

Nazwa oddziału: starter

1. Przejdź do dostarczonej strony repozytorium GitHub dla projektu.
2. Sprawdź, czy nazwa oddziału jest zgodna z nazwą oddziału określoną w ćwiczeniach z programowania. Na przykład na poniższym zrzucie ekranu nazwa gałęzi to **main** .



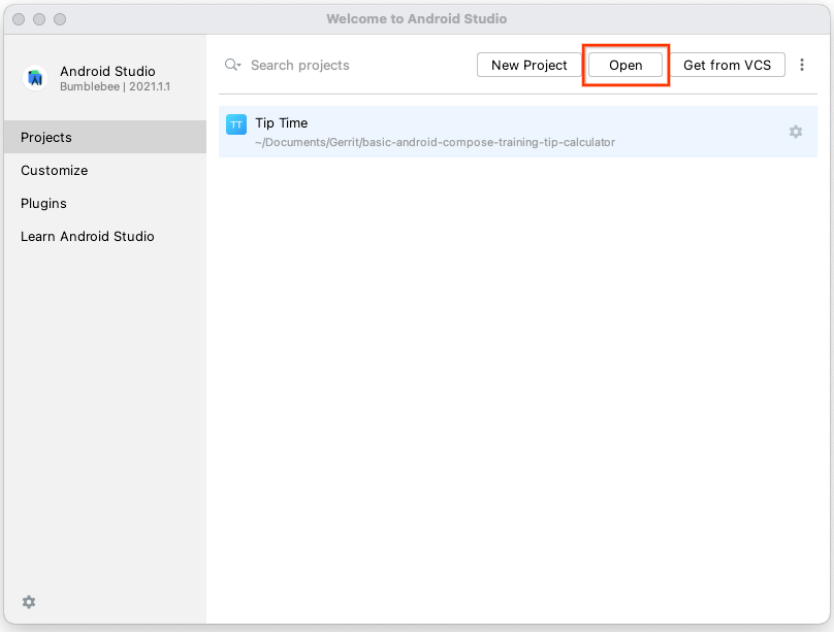
3. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli wyskakujące okienko.



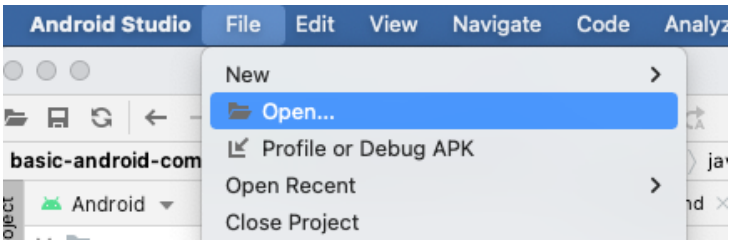
4. W wyskakującym okienku kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na komputerze. Poczekaj na zakończenie pobierania.
5. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
6. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego pliki projektu.

Otwórz projekt w Android Studio

1. Uruchom Android Studio.
2. W oknie **Witamy w Android Studio** kliknij **Otwórz** .



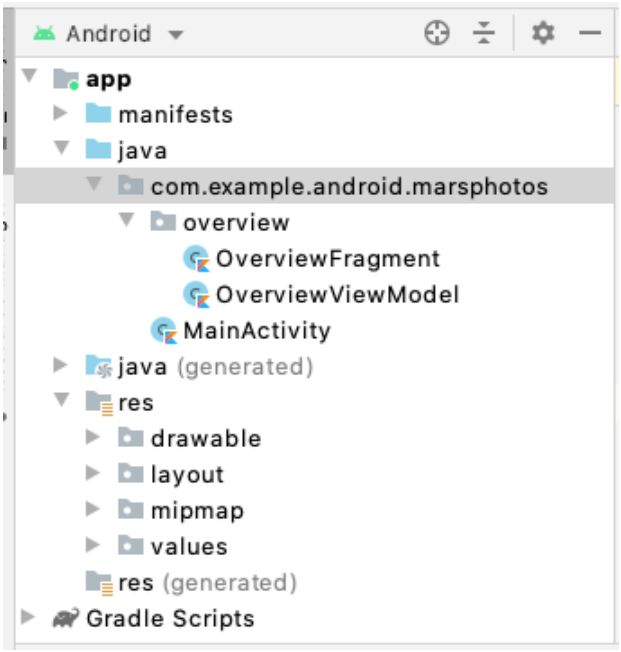
Uwaga: jeśli Android Studio jest już otwarte, wybierz opcję menu **Plik** > **Otwórz** .



3. W przeglądarce plików przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.
6. Kliknij przycisk **Uruchom** , aby skompilować i uruchomić aplikację. Upewnij się, że kompiluje się zgodnie z oczekiwaniami.

Uruchom kod startowy

1. Otwórz pobrany projekt w Android Studio. Nazwa folderu projektu to `android-basics-kotlin-mars-photos-app`. Struktura folderu kodu startowego powinna wyglądać jak poniżej.
2. W okienku **Androida** rozwiń **app** > **java** . Zwróć uwagę, że aplikacja ma folder pakietów o nazwie `overview`. To jest warstwa interfejsu użytkownika aplikacji.



3. Uruchom aplikację. Podczas kompilowania i uruchamiania aplikacji powinieneś zobaczyć następujący ekran z tekstem zastępczym w środku. Pod koniec tego ćwiczenia z programowania zaktualizujesz ten tekst zastępczy o liczbę pobranych zdjęć.

4. Przeglądaj pliki, aby zrozumieć kod startowy. W przypadku plików układu możesz użyć opcji **Podziel** w prawym górnym rogu, aby jednocześnie wyświetlić podgląd układu i XML.

Opis kodu statera

W tym zadaniu zapoznasz się ze strukturą projektu. Oto przewodnik po ważnych plikach i folderach w projekcie.

`OverviewFragment:`

- To jest fragment wyświetlany w `MainActivity`. Tekst zastępczy, który widziałeś w poprzednim kroku, jest wyświetlany w tym fragmencie.
- W kolejnym ćwiczeniu z kodowania ten fragment wyświetli dane otrzymane z serwera zaplecza zdjęć Marsa.
- Ta klasa zawiera odniesienie do `OverviewViewModel`obiektu.
- Ma `OverviewFragment`funkcję `onCreateView()`, która rozszerza `fragment_overview`układ przy użyciu powiązania danych, ustawia właściciela cyklu życia powiązania dla siebie i ustawia `viewModel`zmienną w obiekcie powiązania.
- Ponieważ właściciel cyklu życia jest przypisany, wszelkie `LiveData`zmiany używane w powiązaniu danych będą automatycznie obserwowane pod kątem wszelkich zmian, a interfejs użytkownika zostanie odpowiednio zaktualizowany.

`OverviewViewModel:`

- To jest odpowiedni model widoku dla `OverviewFragment`.
- Ta klasa zawiera `MutableLiveData`właściwość o nazwie `_status`wraz z jej właściwością zapasową. Aktualizacja wartości tej właściwości aktualizuje tekst zastępczy wyświetlany na ekranie.
- Metoda `getMarsPhotos()`aktualizuje odpowiedź zastępczą. W dalszej części ćwiczenia z programowania użyjesz tego do wyświetlenia danych pobranych z serwera. Celem tego ćwiczenia z kodowania jest aktualizacja przy `status` `LiveData`użyciu `ViewModel`rzeczywistych danych, które otrzymujesz z Internetu.

`res/layout/fragment_overview.xml`:

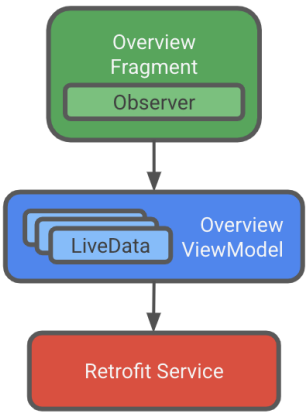
- Ten układ jest skonfigurowany do korzystania z powiązania danych i składa się z jednego `TextView`.
- Deklaruje `OverviewViewModel`zmienną, a następnie wiąże `status`z `ViewModel`do `TextView`.

`MainActivity.kt`: jedynym zadaniem tej czynności jest wczytanie układu czynności, *`activity_main`*.

*`layout/activity_main.xml:`*jest to główny układ aktywności z pojedynczym `FragmentContainerView`wskazaniem na `fragment_overview`, fragment przeglądu zostanie utworzony po uruchomieniu aplikacji.

4. Przegląd aplikacji

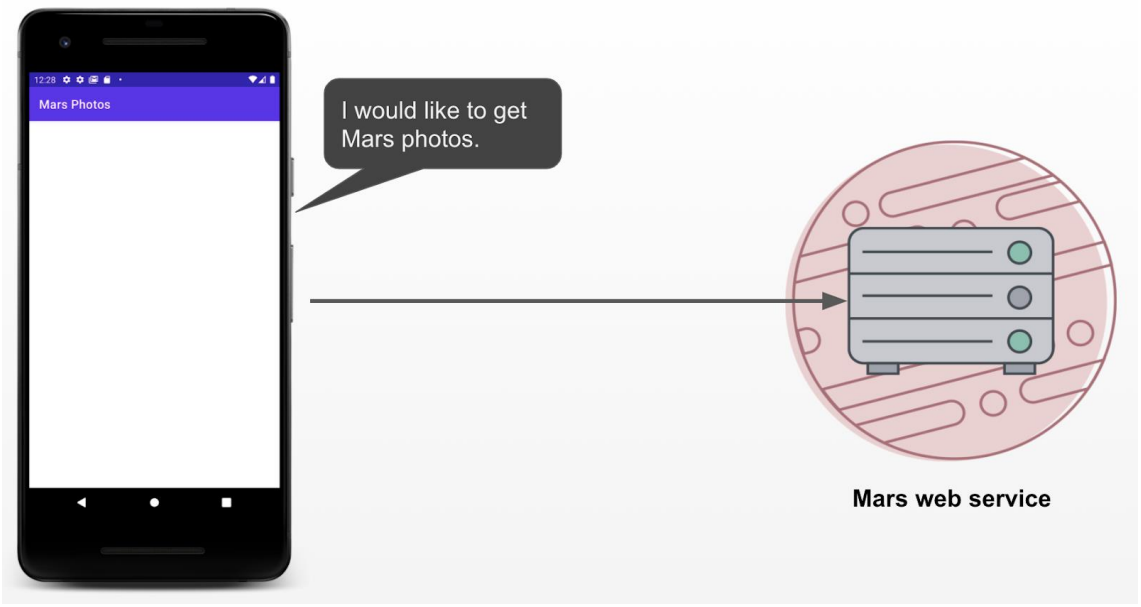
W tym laboratorium programowania tworzysz warstwę dla usługi sieciowej, która komunikuje się z serwerem zaplecza i pobiera wymagane dane. Aby to zaimplementować, użyjesz biblioteki innej firmy, zwanej Retrofit. Więcej na ten temat dowiesz się później. Komunikuje się `ViewModel`bezpośrednio z tą warstwą sieciową, reszta aplikacji jest przezroczysta dla tej implementacji.

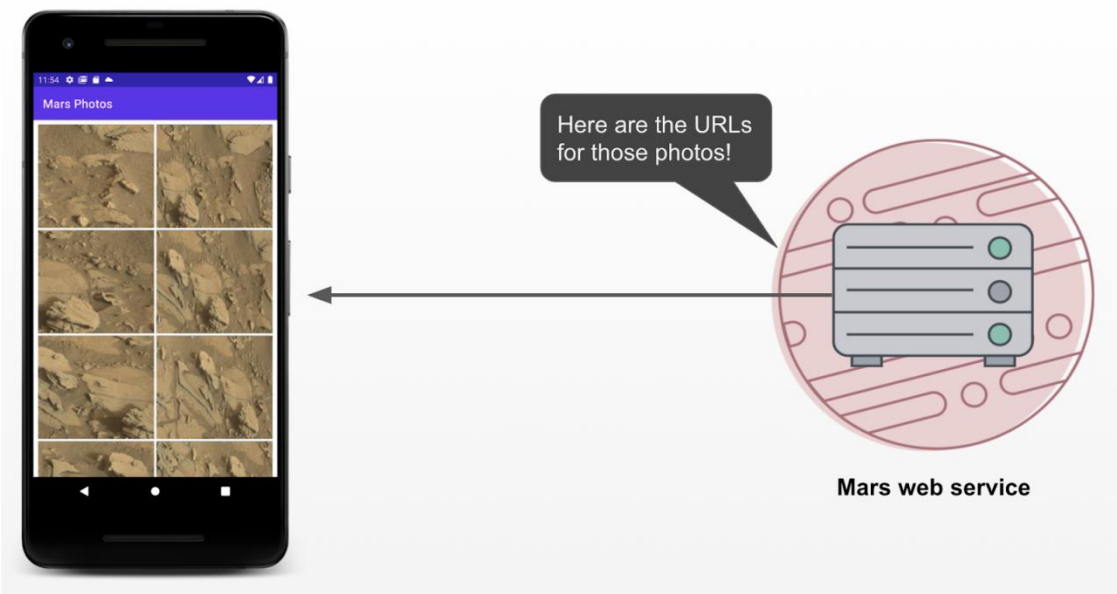


Jest `OverviewViewModel`odpowiedzialny za wykonanie połączenia sieciowego w celu uzyskania danych o zdjęciach Marsa. W programie `ViewModel`możesz użyć `LiveData`powiązania danych uwzględniającego cykl życia, aby zaktualizować interfejs użytkownika aplikacji, gdy dane ulegną zmianie.

5. Usługi internetowe i modernizacja

Dane zdjęć Marsa są przechowywane na serwerze WWW. Aby wprowadzić te dane do swojej aplikacji, musisz nawiązać połączenie i komunikować się z serwerem w Internecie.





Większość dzisiejszych serwerów internetowych obsługuje usługi sieciowe przy użyciu wspólnej bezstanowej architektury sieciowej znanej jako **REST** , co oznacza **REpresentational** State **T** ransfer . Usługi sieci Web, które oferują tę architekturę, są nazywane usługami RESTful.
Żądania są wysyłane do usług sieciowych RESTful w ustandaryzowany sposób za pośrednictwem identyfikatorów URI . Identyfikator URI (Uniform Resource Identifier) identyfikuje zasób na serwerze według nazwy, nie sugerując jego lokalizacji ani sposobu uzyskania do niego dostępu. Na przykład w aplikacji z tej lekcji pobierasz adresy URL obrazów przy użyciu następującego identyfikatora URI serwera (serwer ten obsługuje zarówno zdjęcia nieruchomości Mars, jak i zdjęcia Mars):
android-kotlin-fun-mars-server.appspot.com

Uniform Resource Locator (URL) to identyfikator URI, który określa sposób działania lub uzyskiwania reprezentacji zasobu, tj. określa zarówno jego główny mechanizm dostępu, jak i lokalizację sieciową.

Na przykład:

Poniższy adres URL zawiera listę wszystkich dostępnych nieruchomości na Marsie!

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

Poniższy adres URL zawiera listę zdjęć Marsa:

https://android-kotlin-fun-mars-server.appspot.com/photos
Te adresy URL odnoszą się do zidentyfikowanego zasobu, takiego jak /realestate lub /photos , który można uzyskać za pośrednictwem protokołu Hypertext Transfer Protocol (*http:*) z sieci. W tym laboratorium będziesz używać punktu końcowego /photos .

Uwaga : znany internetowy adres URL jest w rzeczywistości rodzajem identyfikatora URI. Zarówno adres URL, jak i URI są używane zamiennie w tym kursie, w zależności od wywołanego interfejsu API.

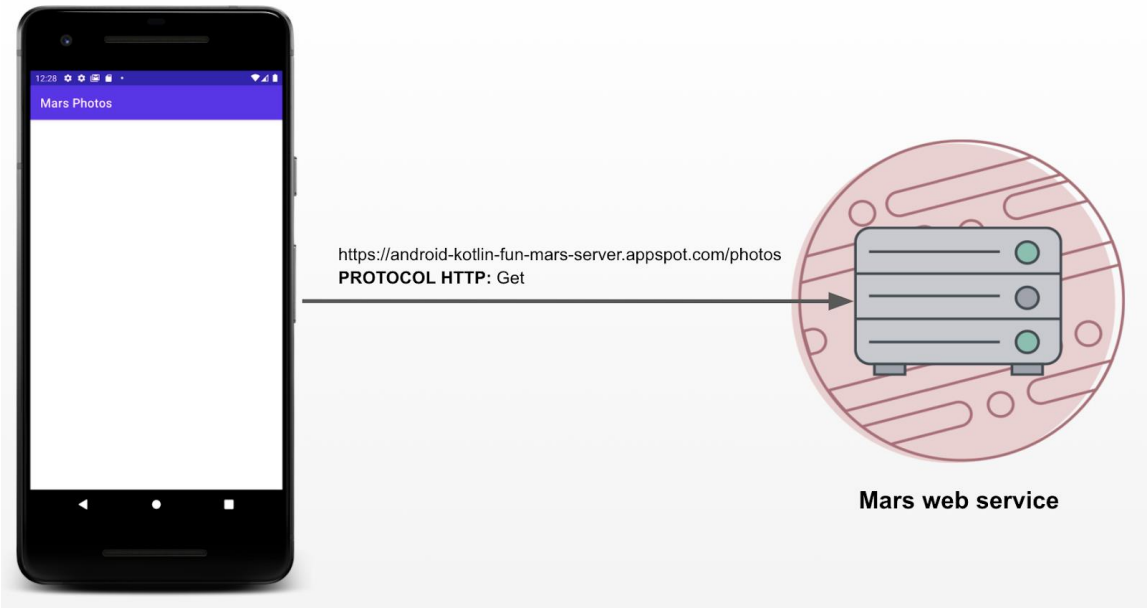
Żądanie **usługi internetowej**

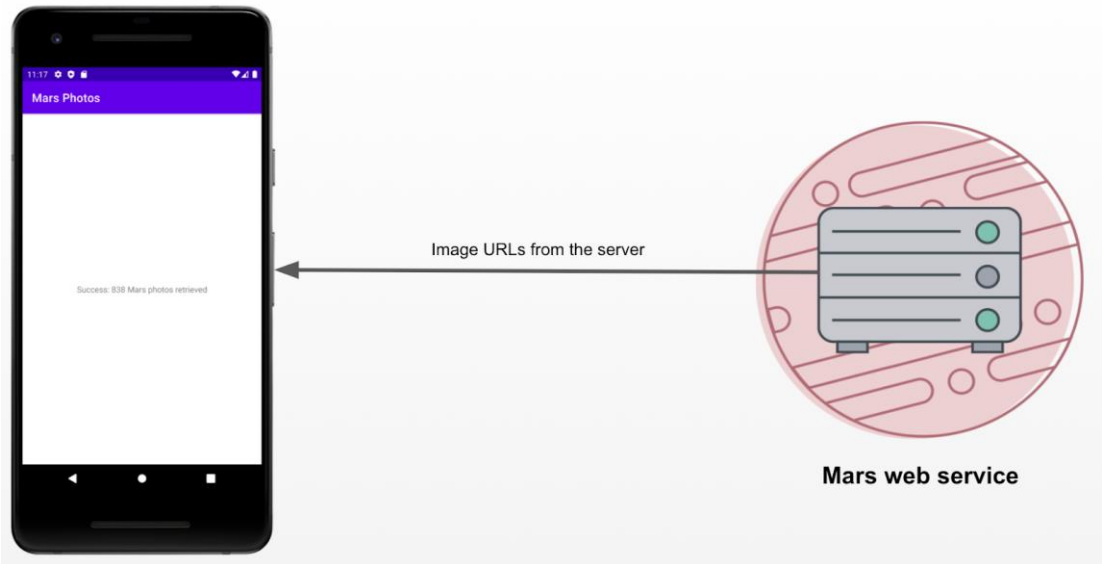
Każde żądanie usługi internetowej zawiera identyfikator URI i jest przesyłane na serwer przy użyciu tego samego protokołu HTTP, którego używają przeglądarki internetowe, takie jak Chrome. Żądania HTTP zawierają operację informującą serwer, co ma zrobić.

Typowe operacje HTTP obejmują:

- GET do pobierania danych z serwera
- POST lub PUT, aby dodać/utworzyć/zaktualizować serwer o nowe dane
- DELETE do usunięcia danych z serwera

Twoja aplikacja wyśle żądanie HTTP GET do serwera w celu uzyskania informacji o zdjęciach Marsa, a następnie serwer zwróci odpowiedź do naszej aplikacji, w tym adresy URL obrazów.





Odpowiedź z usługi internetowej jest często formatowana w jednym z popularnych formatów internetowych, takich jak XML lub JSON — formaty do przedstawiania uporządkowanych danych w parach klucz-wartość. Więcej o JSON dowiesz się w późniejszym zadaniu.

W tym zadaniu nawiązujesz połączenie sieciowe z serwerem, komunikujesz się z serwerem i otrzymujesz odpowiedź JSON. Będziesz korzystać z serwera zaplecza, który jest już dla Ciebie napisany. W tym ćwiczeniu z kodowania użyjesz biblioteki Retrofit, biblioteki innej firmy do komunikacji z serwerem zaplecza.

Biblioteki zewnętrzne

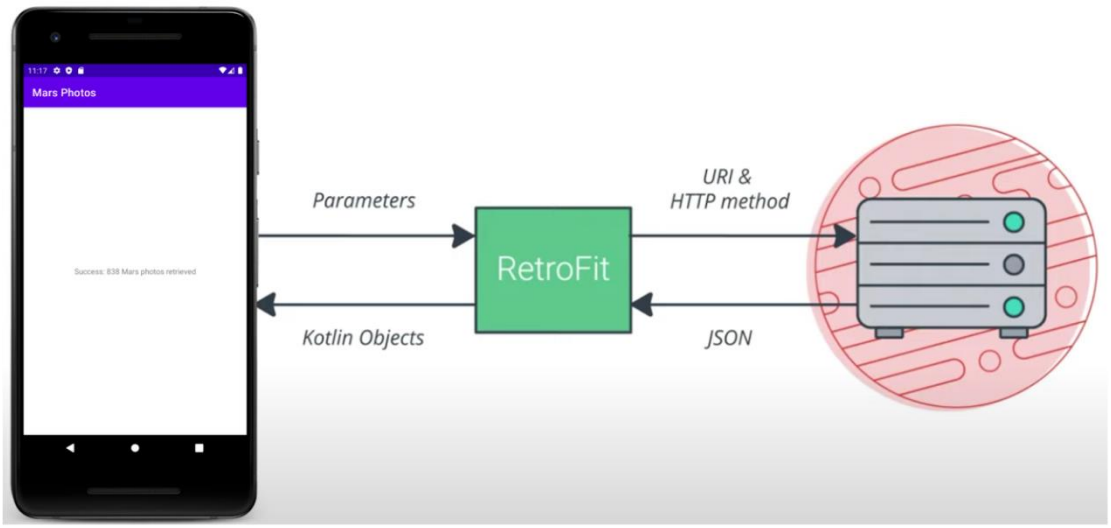
Biblioteki zewnętrzne lub biblioteki innych firm są jak rozszerzenia podstawowych interfejsów API systemu Android. Są to głównie oprogramowanie typu open source, opracowane przez społeczność i utrzymywane przez zbiorowy wkład ogromnej społeczności androidów na całym świecie. Dzięki temu programiści Androida, tacy jak Ty, mogą tworzyć lepsze aplikacje.

Ostrzeżenie : korzystanie z bibliotek stworzonych i utrzymywanych przez społeczność może być ogromną oszczędnością czasu, ale ważne jest, aby wybrać te biblioteki mądrze, ponieważ Twoja aplikacja jest ostatecznie odpowiedzialna za to, co robi kod w tych bibliotekach.

Biblioteka modernizacji

Biblioteka Retrofit, której będziesz używać w tym laboratorium do komunikacji z usługą sieciową RESTful Mars, jest dobrym przykładem dobrze obsługiwanej i utrzymywanej biblioteki. Możesz to powiedzieć, patrząc na jego stronę GitHub , sprawdzając otwarte problemy (niektóre z nich to prośby o funkcje) i zamknięte problemy. Jeśli programiści regularnie rozwiązują problemy i odpowiadają na żądania funkcji, oznacza to, że ta biblioteka jest dobrze utrzymana i jest dobrym kandydatem do użycia w aplikacji. Mają też stronę z dokumentacją modernizacji .

Biblioteka Retrofit będzie komunikować się z backendem. Tworzy identyfikatory URI dla usługi sieciowej na podstawie parametrów, które do niej przekazujemy. Więcej na ten temat dowiesz się w dalszych sekcjach.



Dodaj zależności modernizacji

Android Gradle umożliwia dodawanie zewnętrznych bibliotek do projektu. Oprócz zależności od biblioteki należy również uwzględnić repozytorium, w którym znajduje się biblioteka. Biblioteki Google, takie jak **ViewModel** i **LiveData** z biblioteki Jetpack, są hostowane w repozytorium

Google. Większość bibliotek społecznościowych, takich jak Retrofit, znajduje się w repozytoriach Google i MavenCentral.

1. build.gradle(Project: MarsPhotos)Otwórz plik najwyższego poziomu projektu . Zwróć uwagę na repozytoria wymienione pod repositoriesblokiem. Powinieneś zobaczyć dwa repozytoria, google(), mavenCentral().

```
repositories {
  google()
  mavenCentral()
}
```

2. Otwórz plik oceny na poziomie modułu, build.gradle (Module: MarsPhots.app).
3. W dependenciessekcji dodaj te wiersze dla bibliotek Retrofit:

```
// Retrofit
implementation "com.squareup.retrofit2:retrofit:2.9.0"
// Retrofit with Moshi Converter
implementation "com.squareup.retrofit2:converter-scalars:2.9.0"
```

Pierwsza zależność dotyczy samej biblioteki Retrofit2, a druga zależność dotyczy konwertera skalarnego Retrofit. Ten konwerter umożliwia Retrofit zwrócenie wyniku JSON jako String. Obie biblioteki współpracują ze sobą.

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

Dodaj obsługę funkcji językowych Java 8

Wiele bibliotek innych firm, w tym Retrofit2, korzysta z funkcji języka Java 8. Wtyczka Android Gradle zapewnia wbudowaną obsługę niektórych funkcji języka Java 8.

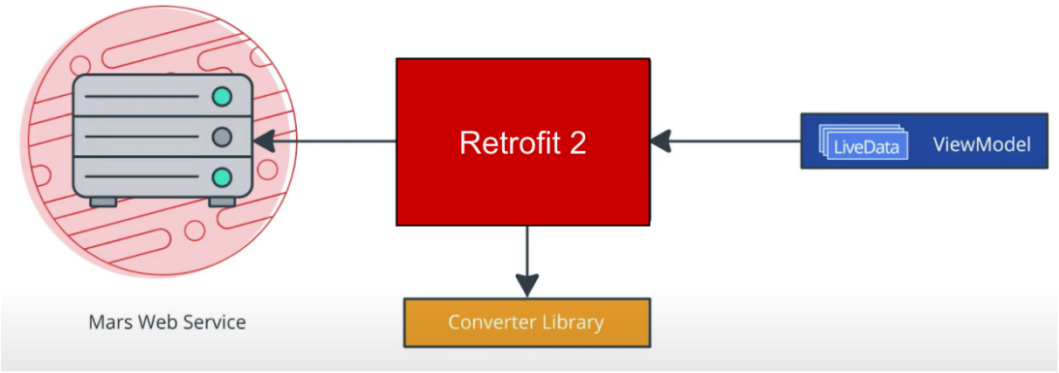
1. Aby korzystać z wbudowanych funkcji, potrzebujesz następującego kodu w build.gradlepliku twojego modułu. Ten krok jest już zrobiony za Ciebie, upewnij się, że następujący kod jest obecny w Twoimbuild.gradle(Module: MarsPhotos.app).

```
android {
  ...

 compileOptions {
   sourceCompatibility JavaVersion.VERSION_1_8
   targetCompatibility JavaVersion.VERSION_1_8
 }

 kotlinOptions {
   jvmTarget = '1.8'
 }
}
```

6. Łączenie się z Internetem

Użyjesz biblioteki Retrofit, aby porozmawiać z usługą internetową Mars i wyświetlić nieprzetworzoną odpowiedź JSON jako `String`. Symbol zastępczy `TextView`wyświetli zwrócony ciąg odpowiedzi JSON lub komunikat wskazujący na błąd połączenia.

Retrofit tworzy sieciowy interfejs API dla aplikacji na podstawie zawartości z usługi sieciowej. Pobiera dane z usługi sieciowej i kieruje je przez oddzielną bibliotekę konwertera, która wie, jak zdekodować dane i zwrócić je w postaci obiektów, takich jak `String`. Retrofit obejmuje wbudowaną obsługę popularnych formatów danych, takich jak XML i JSON. Retrofit ostatecznie tworzy kod do wywołania i wykorzystania tej usługi za Ciebie, w tym krytyczne szczegóły, takie jak uruchamianie żądań w wątkach w tle.



W tym zadaniu dodasz do projektu **MarsPhotos** warstwę sieciową, której `ViewModel`będziesz używać do komunikacji z usługą internetową. Wdrożysz API usługi Retrofit, wykonując następujące kroki.

- Utwórz warstwę sieciową, `MarsApiService`klasę.
- Utwórz obiekt Retrofit z podstawowym adresem URL i fabryką konwerterów.
- Utwórz interfejs wyjaśniający, jak Retrofit komunikuje się z naszym serwerem internetowym.
- Utwórz usługę Retrofit i udostępnij wystąpienie w usłudze API w pozostałej części aplikacji.

Wykonaj powyższe kroki:

1. Utwórz nowy pakiet o nazwie sieć. W panelu projektu Android kliknij prawym przyciskiem myszy pakiet `com.example.android.marsphotos`. Wybierz **Nowy > Pakiet** . W wyskakującym okienku dołącz **sieć** na końcu sugerowanej nazwy pakietu.
2. Utwórz nowy plik Kotlin w nowej **sieci** pakietów . Nazwij to`MarsApiService`.
3. Otwórz `network/MarsApiService.kt`. Dodaj następującą stałą dla podstawowego adresu URL usługi sieci Web.

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

4. Tuż poniżej tej stałej dodaj konstruktora Retrofit, aby zbudować i stworzyć obiekt Retrofit.

```
private val retrofit = Retrofit.Builder()
```

Importuj `retrofit2.Retrofit`po wyświetleniu monitu.

5. Retrofit potrzebuje podstawowego identyfikatora URI dla usługi sieci Web i fabryki konwerterów, aby zbudować interfejs API usług sieci Web. Konwerter informuje Retrofit, co ma zrobić z danymi, które otrzymuje z powrotem z usługi sieciowej. W takim przypadku chcesz, aby Retrofit pobrał odpowiedź JSON z usługi sieci Web i zwrócił ją jako `String`. Retrofit ma funkcję `ScalarsConverter`, która obsługuje ciągi i inne proste typy, więc wywołujesz `addConverterFactory()`konstruktora z instancją `ScalarsConverterFactory`.

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

   Importuj `retrofit2.converter.scalars.ScalarsConverterFactory`po wyświetleniu monitu.

6. Dodaj podstawowy identyfikator URI usługi sieci Web przy użyciu `baseUrl()`metody. Na koniec wywołaj, `build()`aby utworzyć obiekt Retrofit.

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

7. Poniżej wywołania konstruktora Retrofit zdefiniuj interfejs o nazwie `MarsApiService`, który określa, w jaki sposób Retrofit komunikuje się z serwerem WWW za pomocą żądań HTTP.

   ```
   interface MarsApiService {
   }
   ```

8. Wewnątrz `MarsApiService`interfejsu dodaj funkcję wywoływaną `getPhotos()`, aby uzyskać ciąg odpowiedzi z usługi sieciowej.

   ```
   interface MarsApiService {
      fun getPhotos()
   }
   ```

9. Użyj `@GET`adnotacji, aby poinformować Retrofit, że jest to żądanie GET, i określ punkt końcowy dla tej metody usługi sieci Web. W tym przypadku punkt końcowy nazywa się `photos`. Jak wspomniano w poprzednim zadaniu, w tym laboratorium będziesz używać punktu końcowego /photos .

   ```
   interface MarsApiService {
      @GET("photos")
      fun getPhotos()
   }
   ```

   Importuj `retrofit2.http.GET`na żądanie.

10. Po `getPhotos()`wywołaniu metody Retrofit dołącza punkt końcowy `photos`do podstawowego adresu URL (zdefiniowanego w konstruktorze Retrofit) używanego do uruchomienia żądania. Dodaj zwracany typ funkcji do `String`.

    ```
    interface MarsApiService {
       @GET("photos")
       fun getPhotos(): String
    ```

```
}
```

Deklaracje obiektu

W kotlin deklaracje obiektów służą do deklarowania obiektów singleton. Wzorzec Singleton zapewnia, że jedna i tylko jedna instancja obiektu jest tworzona, ma jeden globalny punkt dostępu do tego obiektu. Inicjalizacja deklaracji obiektu jest bezpieczna wątkowo i wykonywana przy pierwszym dostępie.

Kotlin ułatwia deklarowanie singletonów. Poniżej znajduje się przykład deklaracji obiektu i jego dostępu. Deklaracja obiektu zawsze ma nazwę następującą po `object`słowie kluczowym.

Przykład:

```
// Object declaration
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}

// To refer to the object, use its name directly.
DataProviderManager.registerDataProvider(...)
```

Wywołanie funkcji create() na obiekcie Retrofit jest drogie, a aplikacja potrzebuje *tylko jednej* instancji usługi Retrofit API. Tak więc udostępniasz usługę reszcie aplikacji za pomocą *deklaracji obiektu* .

1. Poza `MarsApiService`deklaracją interfejsu zdefiniuj obiekt publiczny wywoływany `MarsApi`w celu zainicjowania usługi Retrofit. Jest to publiczny obiekt singleton, do którego można uzyskać dostęp z pozostałej części aplikacji.

   ```
   object MarsApi {


   }
   ```

2. Wewnątrz `MarsApi`deklaracji obiektu dodaj leniwie zainicjowaną właściwość obiektu modernizacji o nazwie `retrofitService`type `MarsApiService`. Wykonujesz tę leniwą inicjalizację, aby upewnić się, że zostanie zainicjowana przy pierwszym użyciu. Naprawisz błąd w kolejnych krokach.

   ```
   object MarsApi {
       val retrofitService : MarsApiService by lazy {
       }
   }
   ```

3. Zainicjuj `retrofitService`zmienną za pomocą `retrofit.create()`metody z `MarsApiService`interfejsem.

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

Konfiguracja modernizacji zakończona! Za każdym razem, gdy aplikacja wywołuje `MarsApi.retrofitService`, obiekt wywołujący uzyska dostęp do tego samego pojedynczego obiektu Retrofit, który implementuje, `MarsApiService`a który jest tworzony przy pierwszym dostępie. W kolejnym zadaniu użyjesz zaimplementowanego obiektu Retrofit.

Uwaga : Pamiętaj, że „leniwe tworzenie instancji" ma miejsce, gdy tworzenie obiektu jest celowo opóźniane do momentu, w którym obiekt jest rzeczywiście potrzebny, aby uniknąć niepotrzebnych obliczeń lub użycia innych zasobów obliczeniowych. Kotlin ma pierwszorzędne wsparcie dla leniwych instancji.

Wywołaj usługę sieciową w OverviewViewModel

W tym kroku zaimplementujesz `getMarsPhotos()`metodę, która wywołuje usługę retrofit, a następnie obsługuje zwrócony ciąg JSON.

ZobaczModelZakres

A ViewModelScopeto wbudowany zakres współprogramu zdefiniowany dla każdego `ViewModel`w Twojej aplikacji. Wszelkie współprogramy uruchomione w tym zakresie są automatycznie anulowane, jeśli `ViewModel`zostaną wyczyszczone.

Wykorzystasz `ViewModelScope`do uruchomienia współprogramu i wykonania połączenia sieciowego Retrofit w tle.

4. W `MarsApiService`programie utwórz `getPhotos()`funkcję zawieszenia. Aby można było wywołać tę metodę z poziomu współprogramu.

```
@GET("photos")
suspend fun getPhotos(): String
```

5. Otwórz `overview/OverviewViewModel`. Przewiń w dół do `getMarsPhotos()`metody. Usuń wiersz, który ustawia odpowiedź statusu na `"Set the Mars API Response here!"`.Metoda `getMarsPhotos()`powinna być teraz pusta.

```
private fun getMarsPhotos() {

}
```

6. Wewnątrz `getMarsPhotos()`uruchom współprogram za pomocą `viewModelScope.launch`.

```
private fun getMarsPhotos() {
    viewModelScope.launch {
    }
}
```

Importuj `androidx.lifecycle.viewModelScope`i `kotlinx.coroutines.launch`po wyświetleniu monitu.

7. Wewnątrz `viewModelScope`użyj obiektu singleton `MarsApi`, aby wywołać `getPhotos()`metodę z `retrofitService`interfejsu. Zapisz zwróconą odpowiedź w `val`nazwie `listResult`.

```
viewModelScope.launch {
    val listResult = MarsApi.retrofitService.getPhotos()
}
```

Importuj `com.example.android.marsphotos.network.MarsApi`po wyświetleniu monitu.

8. Przypisz wynik, który właśnie otrzymaliśmy z serwera zaplecza do`_status.value.`

```
 val listResult = MarsApi.retrofitService.getPhotos()
 _status.value = listResult
```

9. Uruchom aplikację, zauważ, że aplikacja zamyka się natychmiast, może, ale nie musi, wyświetlić wyskakujące okienko błędu.
10. Kliknij kartę **Logcat** w Android Studio i zanotuj błąd w dzienniku, który zaczyna się od wiersza podobnego do tego „ `------- beginning of crash`”

```
    --------- beginning of crash
22803-22865/com.example.android.marsphotos E/AndroidRuntime: FATAL EXCEPTION: OkHttp
Dispatcher
    Process: com.example.android.marsphotos, PID: 22803
    java.lang.SecurityException: Permission denied (missing INTERNET permission?)
...
```

Ten komunikat o błędzie wskazuje, że aplikacji może brakować `INTERNET`uprawnień. Rozwiążesz ten problem, dodając uprawnienia internetowe do aplikacji w następnym zadaniu.

7. Dodaj uprawnienia do Internetu i obsługę wyjątków

Uprawnienia Androida

Celem uprawnień w Androidzie jest ochrona prywatności użytkownika Androida. Aplikacje na Androida muszą deklarować lub prosić o uprawnienia dostępu do poufnych danych użytkownika, takich jak kontakty, rejestry połączeń i niektóre funkcje systemu, takie jak aparat lub internet.

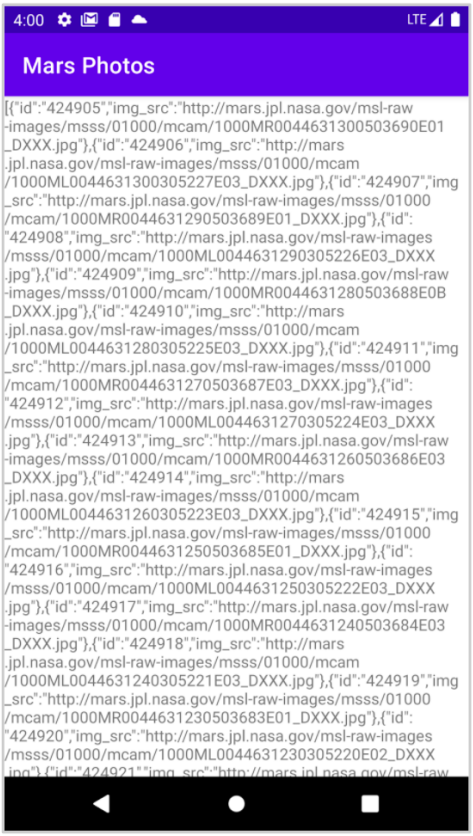
Aby Twoja aplikacja mogła uzyskać dostęp do Internetu, potrzebuje `INTERNET`pozwolenia. Łączenie się z Internetem stwarza obawy dotyczące bezpieczeństwa, dlatego aplikacje domyślnie nie mają połączenia z Internetem. Musisz wyraźnie zadeklarować, że aplikacja potrzebuje dostępu do internetu. Jest to uważane za normalne pozwolenie. Aby dowiedzieć się więcej o uprawnieniach systemu Android i ich typach, zapoznaj się z dokumentacją .

Na tym etapie aplikacja deklaruje wymagane uprawnienia, umieszczając w pliku `<uses-permission>`tagi .`AndroidManifest`

1. Otwórz `manifests/AndroidManifest.xml`. Dodaj tę linię tuż przed `<application>`tagiem:

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

2. Skompiluj i ponownie uruchom aplikację. Jeśli masz działające połączenie internetowe, powinieneś zobaczyć tekst JSON zawierający dane związane ze zdjęciami Marsa. Więcej o formacie JSON dowiesz się w dalszej części ćwiczenia z programowania.



3. Stuknij przycisk **Wstecz** na urządzeniu lub emulatorze, aby zamknąć aplikację.
4. Przełącz urządzenie lub emulator w tryb samolotowy, aby zasymulować błąd połączenia sieciowego. Ponownie otwórz aplikację z menu Ostatnie lub uruchom ponownie aplikację z Android Studio.
5. Kliknij kartę **Logcat** w Android Studio i zanotuj krytyczny wyjątek w dzienniku, który wygląda tak:

```
3302-3302/com.example.android.marsphotos E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.example.android.marsphotos, PID: 3302
    java.net.SocketTimeoutException: timeout
...
```

Ten komunikat o błędzie wskazuje, że aplikacja próbowała nawiązać połączenie i upłynął limit czasu. Takie wyjątki są bardzo częste w czasie rzeczywistym. W następnym kroku dowiesz się, jak radzić sobie z takimi wyjątkami.

Obsługa wyjątków

Wyjątki to błędy, które mogą wystąpić w czasie wykonywania (nie w czasie kompilacji) i nagle zakończyć działanie aplikacji bez powiadamiania użytkownika. Może to skutkować kiepskimi doświadczeniami użytkownika. *Obsługa wyjątków* to mechanizm, za pomocą którego zapobiegasz nagłemu zamknięciu aplikacji i obsługujesz ją w sposób przyjazny dla użytkownika.

Powód wyjątków może być tak prosty, jak dzielenie przez zero lub błąd w sieci. Te wyjątki są podobne do tych `NumberFormatException`, których nauczyłeś się podczas poprzedniego ćwiczenia z programowania.

Przykłady potencjalnych problemów podczas łączenia się z serwerem:

- Adres URL lub identyfikator URI używany w interfejsie API jest nieprawidłowy.
- Serwer jest niedostępny i aplikacja nie może się z nim połączyć.
- Problem z opóźnieniem sieci.
- Słabe lub brak połączenia internetowego na urządzeniu.

Tych wyjątków nie można przechwycić w czasie kompilacji. Możesz użyć `try-catch`bloku do obsługi wyjątku w czasie wykonywania. Aby uzyskać więcej informacji, zapoznaj się z dokumentacją .

Przykładowa składnia bloku try-catch

```
try {
    // some code that can cause an exception.
}
catch (e: SomeException) {
    // handle the exception to avoid abrupt termination.
}
```

Wewnątrz `try`bloku wykonujesz kod, w którym przewidujesz wyjątek, w Twojej aplikacji byłoby to połączenie sieciowe. W `catch`bloku zaimplementujesz kod, który zapobiega nagłemu zamknięciu aplikacji. Jeśli wystąpi wyjątek, `catch`blok zostanie wykonany w celu naprawienia błędu zamiast nagłego zakończenia aplikacji.

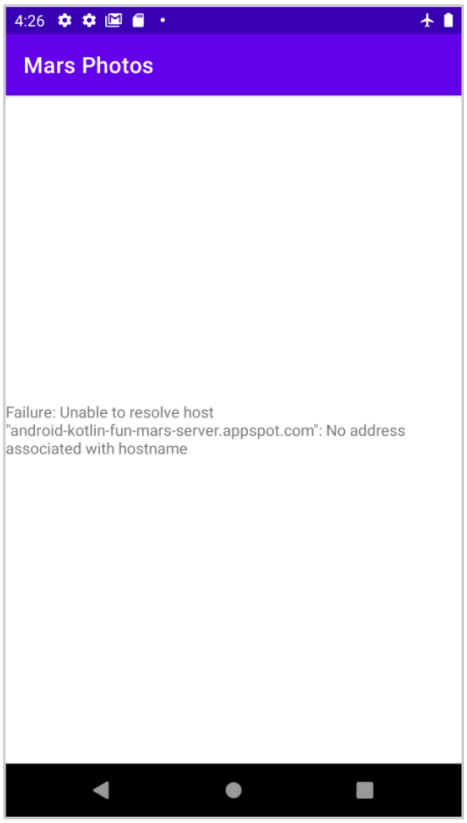
1. Otwórz `overview/OverviewViewModel.kt`. Przewiń w dół do `getMarsPhotos()`metody. Wewnątrz bloku uruchamiania dodaj `try`blok wokół `MarsApi`wywołania, aby obsługiwać wyjątki. Dodaj `catch`blok po `try`bloku:

```
viewModelScope.launch {
  try {
     val listResult = MarsApi.retrofitService.getPhotos()
     _status.value = listResult
  } catch (e: Exception) {

  }
}
```

2. Wewnątrz `catch {}`bloku obsłuż odpowiedź na awarię. Wyświetl komunikat o błędzie użytkownikowi, ustawiając `e.message`na `_status.value`.

```
catch (e: Exception) {
  _status.value = "Failure: ${e.message}"
}
```

3. Uruchom aplikację ponownie z włączonym trybem samolotowym. Tym razem aplikacja nie zamyka się nagle, ale zamiast tego wyświetla komunikat o błędzie.



4. Wyłącz tryb samolotowy w telefonie lub emulatorze. Uruchom i przetestuj swoją aplikację, upewnij się, że wszystko działa poprawnie i możesz zobaczyć ciąg JSON.

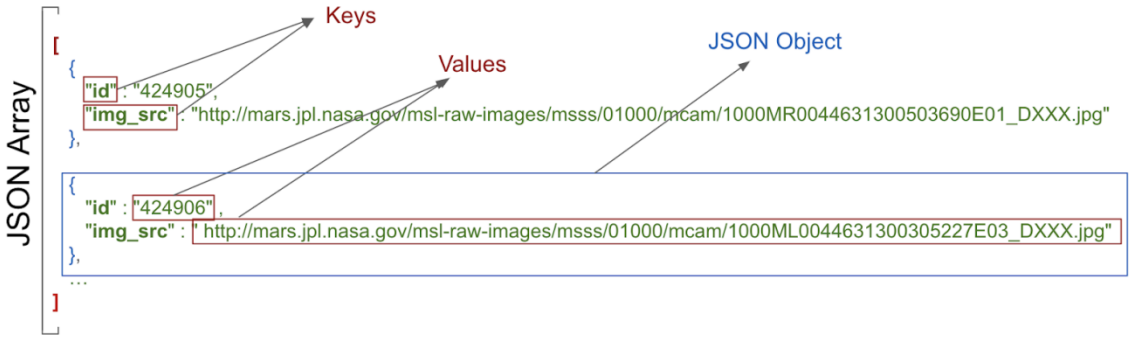
8. Przeanalizuj odpowiedź JSON za pomocą Moshi

JSON

Żądane dane są zwykle sformatowane w jednym z popularnych formatów danych, takich jak XML lub JSON. Każde wywołanie zwraca uporządkowane dane, a Twoja aplikacja musi wiedzieć, jaka jest ta struktura, aby odczytać dane z odpowiedzi.

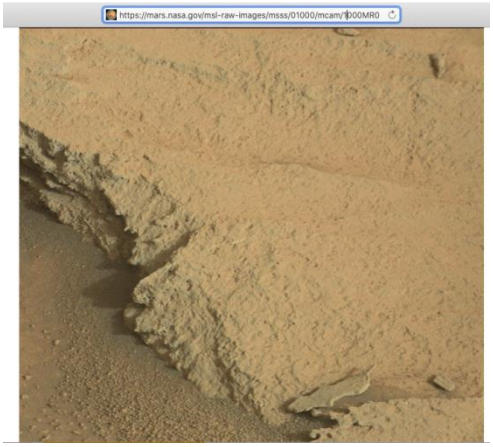
Na przykład w tej aplikacji będziesz pobierać dane z tego serwera: https:// android-kotlin-fun-mars-server.appspot.com/photos . Jeśli wpiszesz ten adres URL w przeglądarce, zobaczysz listę identyfikatorów i adresów URL obrazów powierzchni Marsa w formacie JSON!

Struktura przykładowej odpowiedzi JSON:



- Odpowiedź JSON to tablica wskazana przez nawiasy kwadratowe. Tablica zawiera obiekty JSON.
- Obiekty JSON są otoczone nawiasami klamrowymi.
- Każdy obiekt JSON zawiera zestaw par nazwa-wartość oddzielonych przecinkiem.
- Nazwa i wartość w parze są oddzielone dwukropkiem.
- Nazwy są otoczone cudzysłowami.
- Wartości mogą być liczbami, ciągami, wartością logiczną, tablicą, obiektem (obiektem JSON) lub wartością null.

Na przykład the `img_src`to adres URL, który jest ciągiem. Jeśli wkleisz adres URL do przeglądarki internetowej, zobaczysz obraz powierzchni Marsa.



Teraz otrzymujesz odpowiedź JSON z usługi internetowej Mars, co jest świetnym początkiem. Ale to, czego naprawdę potrzebujesz, to obiekty Kotlin, a nie duży ciąg JSON. Istnieje zewnętrzna biblioteka o nazwie Moshi , która jest parserem JSON Androida, który konwertuje ciąg JSON na obiekty Kotlin. Retrofit ma konwerter, który współpracuje z Moshi, więc jest to świetna biblioteka do twoich celów.

W tym zadaniu używasz biblioteki Moshi z Retrofit, aby przeanalizować odpowiedź JSON z usługi sieciowej na przydatne obiekty Kotlin, które reprezentują zdjęcia Marsa. Zmienisz aplikację tak, aby zamiast wyświetlać nieprzetworzony JSON, aplikacja wyświetlała liczbę zwróconych zdjęć Marsa.

Dodaj zależności biblioteki Moshi

1. Otwórz **build.gradle (Moduł: aplikacja)** .

2. W sekcji zależności dodaj poniższy kod, aby uwzględnić zależność Moshi. Ta zależność dodaje obsługę biblioteki Moshi JSON z obsługą Kotlin.

```
// Moshi
implementation 'com.squareup.moshi:moshi-kotlin:1.13.0'
```

3. Znajdź wiersze dla konwertera skalarnego Retrofit w `dependencies`bloku i zmień te zależności, aby użyć `converter-moshi`:

Wymień to

```
// Retrofit
implementation "com.squareup.retrofit2:retrofit:2.9.0"
// Retrofit with Moshi Converter
implementation "com.squareup.retrofit2:converter-scalars:2.9.0"
```

z tym

```
// Retrofit with Moshi Converter
implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'
```

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

Uwaga: Projekt może pokazywać błędy kompilatora związane z usuniętą zależnością skalarną Retrofit. Naprawisz je w następnych krokach.

Zaimplementuj klasę danych Mars Photos

Przykładowy wpis odpowiedzi JSON, którą otrzymujesz z usługi sieciowej, wygląda mniej więcej tak, podobnie do tego, co widzieliście wcześniej:

```
[{
   "id":"424906",
   "img_src":"http://mars.jpl.nasa.gov/msl-raw-
images/msss/01000/mcam/1000ML004463130030522 7E03_DXXX.jpg"
},
...]
```

W powyższym przykładzie zauważ, że każdy wpis dotyczący zdjęcia Marsa ma następujące pary klucza JSON i wartości:

- `id`: identyfikator właściwości w postaci ciągu. Ponieważ jest w `" "`nią zawinięty, jest typu `String`nie `Integer`.
- `img_src`: URL obrazu jako ciąg.

Moshi analizuje te dane JSON i konwertuje je na obiekty Kotlin. Aby to zrobić, Moshi musi mieć klasę danych Kotlin do przechowywania przeanalizowanych wyników, więc w tym kroku utworzysz klasę danych, `MarsPhoto`.

1. Kliknij prawym przyciskiem myszy pakiet **sieciowy i wybierz Nowy > Plik/Klasa Kotlin.**

2. W wyskakującym okienku wybierz **Klasa** i wprowadź `MarsPhoto`jako nazwę klasy. Tworzy to nowy plik o nazwie `MarsPhoto.kt`w `network`pakiecie.
3. Utwórz `MarsPhoto`klasę danych, dodając `data`słowo kluczowe przed definicją klasy. Zmień nawiasy {} na () nawiasy. To pozostawia błąd, ponieważ klasy danych muszą mieć zdefiniowaną co najmniej jedną właściwość.

   ```
   data class MarsPhoto(
   )
   ```

4. Dodaj następujące właściwości do `MarsPhoto`definicji klasy.

   ```
   data class MarsPhoto(
     val id: String, val img_src: String
   )
   ```

   Zauważ, że każda ze zmiennych w `MarsPhoto`klasie odpowiada nazwie klucza w obiekcie JSON. Aby dopasować typy w naszej konkretnej odpowiedzi JSON, używasz `String`obiektów dla wszystkich wartości.

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

   ### @Json Adnotacja

   Czasami nazwy kluczy w odpowiedzi JSON mogą wprowadzać mylące właściwości Kotlin lub mogą nie pasować do zalecanego stylu kodowania — na przykład w pliku JSON `img_src`klucz używa podkreślenia, podczas gdy konwencja Kotlin dla właściwości używa wielkich i małych liter („wielbłąd”).

   Aby użyć w klasie danych nazw zmiennych, które różnią się od nazw kluczy w odpowiedzi JSON, użyj `@Json`adnotacji. W tym przykładzie nazwa zmiennej w klasie danych to `imgSrcUrl`. Zmienną można mapować do atrybutu JSON `img_src`przy użyciu `@Json(name = "img_src")`.

5. Zastąp linię `img_src`klucza linią pokazaną poniżej. Importuj `com.squareup.moshi.Json`na żądanie.

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

Zaktualizuj MarsApiService i przeglądViewModel

W tym zadaniu utworzysz obiekt Moshi za pomocą Moshi Builder, podobnego do narzędzia Retrofit builder.

Zastąpisz `ScalarsConverterFactory`na `KotlinJsonAdapterFactory`, aby Retrofit wiedział, że może użyć Moshi do konwersji odpowiedzi JSON na obiekty Kotlin. Następnie zaktualizujesz interfejs API sieci i `ViewModel`użyjesz obiektu Moshi.

1. Otwórz `network/MarsApiService.kt`. Zwróć uwagę na nierozwiązane błędy odniesienia dla `ScalarsConverterFactory`. Dzieje się tak z powodu zmiany zależności Retrofit wprowadzonej w poprzednim kroku. Usuń import dla `ScalarConverterFactory`. Wkrótce naprawisz drugi błąd.

   Usunąć:

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

2. Na górze pliku, tuż przed kreatorem Retrofit, dodaj następujący kod, aby utworzyć obiekt Moshi, podobny do obiektu Retrofit.

```
private val moshi = Moshi.Builder()
```

Importuj `com.squareup.moshi.Moshi` i `com.squareup.moshi.kotlin.reflect.KotlinJsonAdapterFactory` na żądanie.

3. Aby adnotacje Moshi działały poprawnie z Kotlinem, w kreatorze Moshi dodaj `KotlinJsonAdapterFactory`, a następnie wywołaj `build()`.

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

4. W *retrofit*deklaracji obiektu zmień konstruktora Retrofit, aby używał `MoshiConverterFactory` zamiast `ScalarConverterFactory`, i przekaż do `moshi` właśnie utworzonej instancji.

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

Importuj `retrofit2.converter.moshi.MoshiConverterFactory` na żądanie.

5. Teraz, gdy masz już to `MoshiConverterFactory` miejsce, możesz poprosić Retrofit o zwrócenie listy `MarsPhoto` obiektów z tablicy JSON zamiast zwracania ciągu JSON. Zaktualizuj `MarsApiService` interfejs, aby Retrofit zwracał listę `MarsPhoto` obiektów, zamiast zwracać `String`.

```
interface MarsApiService {
  @GET("photos")
  suspend fun getPhotos(): List<MarsPhoto>
}
```

6. Wykonaj podobne zmiany w `viewModel`, open `OverviewViewModel.kt`. Przewiń w dół do `getMarsPhotos()` metody.
7. W metodzie `getMarsPhotos()` już nie `listResult` jest . Rozmiar tej listy to liczba zdjęć, które zostały odebrane i przeanalizowane. Aby wydrukować liczbę pobranych zdjęć, zaktualizuj w następujący sposób. `List<MarsPhoto>String_status.value`

```
_status.value = "Success: ${listResult.size} Mars photos retrieved"
```

Importuj `com.example.android.marsphotos.network.MarsPhoto` po wyświetleniu monitu.

8. Upewnij się, że tryb samolotowy jest wyłączony w urządzeniu lub emulatorze. Skompiluj i uruchom aplikację. Tym razem komunikat powinien pokazywać liczbę właściwości zwróconych z usługi sieciowej, a nie duży ciąg JSON:

Uwaga: jeśli połączenie internetowe nie działa, upewnij się, że na urządzeniu lub emulatorze został wyłączony tryb samolotowy.

9. Kod rozwiązania

Uwaga : Upewnij się, że nazwa pakietu w poniższym kodzie źródłowym jest zgodna z nazwą pakietu w Twoim projekcie.

build.gradle(Module : MarsPhotos.app)

Oto nowe zależności, które należy uwzględnić.

```
dependencies {
    ...
    // Moshi
    implementation 'com.squareup.moshi:moshi-kotlin:1.13.0'

    // Retrofit with Moshi Converter
    implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'

    ...
}
```

Manifests/AndroidManifest.xml

Dodaj zezwolenie internetowe, <uses-permission..>kod z poniższego fragmentu kodu.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.marsphotos">
```

```
    <!-- In order for our app to access the Internet, we need to define this permission. -->
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        ...
    </application>

</manifest>
```

network/MarsPhoto.kt

```
package com.example.android.marsphotos.network

import com.squareup.moshi.Json

/**
* This data class defines a Mars photo which includes an ID, and the image URL.
* The property names of this data class are used by Moshi to match the names of values in JSON.
*/
data class MarsPhoto(
    val id: String,
    @Json(name = "img_src") val imgSrcUrl: String
)
```

network/MarsApiService.kt

```
package com.example.android.marsphotos.network

import com.squareup.moshi.Moshi
import com.squareup.moshi.kotlin.reflect.KotlinJsonAdapterFactory
import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory
import retrofit2.http.GET

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

/**
* Build the Moshi object with Kotlin adapter factory that Retrofit will be using.
*/
private val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()

/**
```

```
* The Retrofit object with the Moshi converter.
*/
private val retrofit = Retrofit.Builder()
  .addConverterFactory(MoshiConverterFactory.create(moshi))
  .baseUrl(BASE_URL)
  .build()

/**
* A public interface that exposes the [getPhotos] method
*/
interface MarsApiService {
  /**
   * Returns a [List] of [MarsPhoto] and this method can be called from a Coroutine.
   * The @GET annotation indicates that the "photos" endpoint will be requested with the GET
   * HTTP method
   */
  @GET("photos")
  suspend fun getPhotos() : List<MarsPhoto>
}

/**
* A public Api object that exposes the lazy-initialized Retrofit service
*/
object MarsApi {
  val retrofitService: MarsApiService by lazy { retrofit.create(MarsApiService::class.java) }
}
```

Overview/OverviewViewModel.kt

```
package com.example.android.marsphotos.overview

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.example.android.marsphotos.network.MarsApi
import kotlinx.coroutines.launch

/**
* The [ViewModel] that is attached to the [OverviewFragment].
*/
class OverviewViewModel : ViewModel() {

  // The internal MutableLiveData that stores the status of the most recent request
  private val _status = MutableLiveData<String>()
```

```
    // The external immutable LiveData for the request status
    val status: LiveData<String> = _status
    /**
     * Call getMarsPhotos() on init so we can display status immediately.
     */
    init {
        getMarsPhotos()
    }

    /**
     * Gets Mars photos information from the Mars API Retrofit service and updates the
     * [MarsPhoto] [List] [LiveData].
     */
    private fun getMarsPhotos() {
        viewModelScope.launch {
            try {
                val listResult = MarsApi.retrofitService.getPhotos()
                _status.value = "Success: ${listResult.size} Mars photos retrieved"
            } catch (e: Exception) {
                _status.value = "Failure: ${e.message}"
            }
        }
    }
}
```

10. Podsumowanie

Usługi internetowe REST

- Usługa *sieci Web* to funkcja oprogramowania oferowana przez Internet, która umożliwia aplikacji wysyłanie żądań i odzyskiwanie danych.
- Typowe usługi WWW korzystają z architektury REST . Usługi sieci Web oferujące architekturę REST są znane jako usługi *RESTful* . Usługi sieciowe RESTful są budowane przy użyciu standardowych komponentów internetowych i protokołów.
- Żądanie do usługi sieciowej REST składasz w sposób ustandaryzowany, za pośrednictwem identyfikatorów URI.
- Aby korzystać z usługi internetowej, aplikacja musi nawiązać połączenie sieciowe i komunikować się z usługą. Następnie aplikacja musi odebrać i przeanalizować dane odpowiedzi w formacie, którego może używać aplikacja.
- Biblioteka Retrofit to biblioteka klienta, która umożliwia aplikacji wysyłanie żądań do usługi sieci Web REST.
- Użyj konwerterów, aby poinformować Retrofit, co ma zrobić z danymi, które wysyła do usługi sieciowej i wraca z usługi sieciowej. Na przykład `ScalarsConverter`konwerter traktuje dane usługi sieci Web jako `String`lub inny prymityw.

- Aby umożliwić aplikacji nawiązywanie połączeń z internetem, dodaj `"android.permission.INTERNET"`uprawnienia w manifeście Androida.

Parsowanie JSON

- Odpowiedź z usługi internetowej jest często sformatowana w formacie JSON , popularnym formacie reprezentacji danych strukturalnych.
- Obiekt JSON to zbiór par klucz-wartość.
- Kolekcja obiektów JSON to tablica JSON. Otrzymasz tablicę JSON jako odpowiedź z usługi internetowej.
- Klucze w parze klucz-wartość są otoczone cudzysłowami. Wartości mogą być liczbami lub ciągami.
- Biblioteka Moshi to parser Androida JSON, który konwertuje ciąg znaków JSON na obiekty Kotlin. Retrofit posiada konwerter współpracujący z Moshi.
- Moshi dopasowuje klucze w odpowiedzi JSON do właściwości w obiekcie danych o tej samej nazwie.
- Aby użyć innej nazwy właściwości dla klucza, dodaj do tej właściwości `@Json`adnotację i nazwę klucza JSON.

11. Dowiedz się więcej

Dokumentacja dla programistów Androida:

- ZobaczPrzegląd Modelu
- Przegląd danych na żywo
- `MutableLiveData`
- `ViewModelScope`

Dokumentacja Kotlina:

- Wyjątki: spróbuj, złap, w końcu rzuć, nic
- Laboratorium współprogramów
- Współprogramy, oficjalna dokumentacja
- Kontekst współprogramowy i dyspozytorzy

Inny:

- Modernizacja
- Palić
- Moshi z modernizacją w Kotlin

Załaduj i wyświetl obrazy z Internetu

1. Witamy!

Wstęp

W poprzednim laboratorium nauczyłeś się, jak uzyskać dane z usługi sieciowej i przetworzyć odpowiedź na obiekt Kotlin. W tym ćwiczeniu z programowania wykorzystujesz tę wiedzę, aby ładować i wyświetlać zdjęcia z internetowego adresu URL. Powrócisz również do tego, jak zbudować `RecyclerView`i używać go do wyświetlania siatki obrazów na stronie przeglądu.

Warunki wstępne

- Jak tworzyć i wykorzystywać fragmenty.
- Jak pobrać JSON z usługi internetowej REST i przeanalizować te dane do obiektów Kotlin za pomocą bibliotek Retrofit i Moshi .
- Jak skonstruować układ siatki za pomocą `RecyclerView`.
- Jak `Adapter`, `ViewHolder`i `DiffUtil`praca.

Czego się nauczysz

- Jak używać biblioteki Coil do ładowania i wyświetlania obrazu z internetowego adresu URL.
- Jak używać `RecyclerView`adaptera a i siatki do wyświetlania siatki obrazów.
- Jak radzić sobie z potencjalnymi błędami podczas pobierania i wyświetlania obrazów.

Co **zbudujesz**

- Zmodyfikuj aplikację MarsPhotos, aby uzyskać adres URL obrazu z danych Marsa, a następnie użyj funkcji Coil, aby załadować i wyświetlić ten obraz.
- Dodaj animację ładowania i ikonę błędu do aplikacji.
- Użyj a, `RecyclerView`aby wyświetlić siatkę obrazów Marsa.
- Dodaj obsługę stanu i błędów do `RecyclerView`.

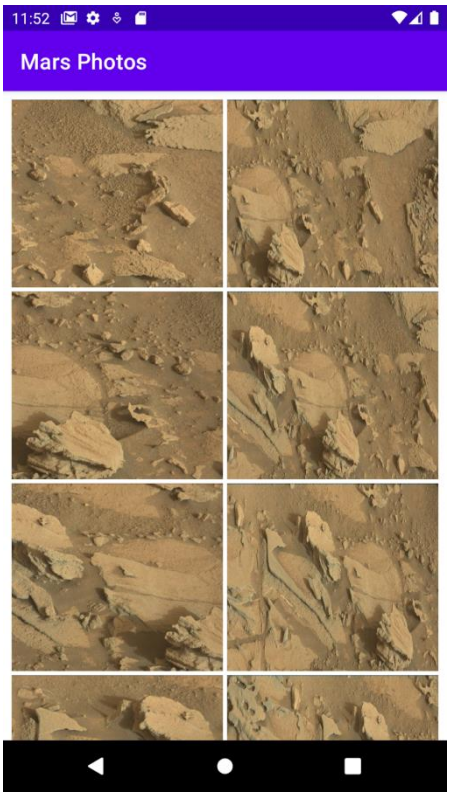
Czego potrzebujesz

- Komputer z nowoczesną przeglądarką internetową, np. najnowszą wersją Chrome .
- Dostęp do Internetu na Twoim komputerze.

2. Przegląd aplikacji

W tym laboratorium będziesz kontynuować pracę z aplikacją z poprzedniego laboratorium o nazwie MarsPhotos. Aplikacja MarsPhotos łączy się z usługą internetową, aby pobrać i wyświetlić liczbę obiektów Kotlin pobranych za pomocą Retrofit. Te obiekty Kotlin zawierają adresy URL rzeczywistych zdjęć powierzchni Marsa uchwyconych przez łaziki marsjańskie NASA.

Wersja aplikacji, którą zbudujesz w tym laboratorium, wypełni stronę przeglądu, która wyświetla zdjęcia Marsa w postaci siatki obrazów. Obrazy są częścią danych, które Twoja aplikacja pobrała z usługi internetowej Mars. Twoja aplikacja będzie używać biblioteki Coil do ładowania i wyświetlania obrazów oraz `RecyclerView`do tworzenia układu siatki dla obrazów. Twoja aplikacja z łatwością poradzi sobie również z błędami sieci.



3. Wyświetl obraz internetowy

Wyświetlanie zdjęcia z internetowego adresu URL może wydawać się proste, ale jest sporo inżynierii, aby to działało dobrze. Obraz musi zostać pobrany, zapisany wewnętrznie i zdekodowany ze skompresowanego formatu do obrazu, z którego może korzystać system Android. Obraz powinien być buforowany w pamięci podręcznej w pamięci, pamięci podręcznej opartej na magazynie lub obu. Wszystko to musi się dziać w wątkach w tle o niskim priorytecie, aby interfejs użytkownika pozostawał responsywny. Ponadto, aby uzyskać najlepszą wydajność sieci i procesora, możesz chcieć pobierać i dekodować więcej niż jeden obraz naraz.

Na szczęście możesz użyć biblioteki opracowanej przez społeczność o nazwie Coil do pobierania, buforowania, dekodowania i buforowania obrazów. Bez użycia cewki miałbyś znacznie więcej pracy do wykonania.

Cewka zasadniczo potrzebuje dwóch rzeczy:

- Adres URL obrazu, który chcesz załadować i wyświetlić.
- Obiekt `ImageView`do wyświetlania tego obrazu.

W tym zadaniu dowiesz się, jak używać Coil do wyświetlania pojedynczego obrazu z serwisu internetowego Mars. Wyświetlasz obraz pierwszego zdjęcia Marsa na liście zdjęć zwracanych przez serwis internetowy. Oto zrzuty ekranu przed i po:

Dodaj zależność cewki

1. Otwórz aplikację rozwiązania MarsPhotos z poprzedniego ćwiczenia z programowania.
2. Uruchom aplikację, aby zobaczyć, co robi. (Pokazuje całkowitą liczbę pobranych zdjęć Marsa).
3. Otwórz **build.gradle (Moduł: aplikacja)** .
4. W `dependencies`sekcji dodaj ten wiersz dla biblioteki Coil:

```
// Coil
implementation "io.coil-kt:coil:1.1.1"
```

Sprawdź i zaktualizuj najnowszą wersję biblioteki ze strony dokumentacji cewki .

5. Biblioteka Coil jest hostowana i dostępna w `mavenCentral()`repozytorium. W **build.gradle (Project: MarsPhotos)** dodaj `mavenCentral()`w górnym `repositories`bloku.

```
repositories {
  google()
  mavenCentral()
}
```

6. Kliknij opcję **Synchronizuj teraz** , aby odbudować projekt z nową zależnością.

Zaktualizuj ViewModel

W tym kroku dodasz do klasy `LiveData`właściwość `OverviewViewModel`do przechowywania otrzymanego obiektu Kotlin, MarsPhoto.

1. Otwórz `overview/OverviewViewModel.kt`. Tuż pod `_status`deklaracją właściwości dodaj nową zmienną właściwość o nazwie `_photos`typu `MutableLiveData`, która może przechowywać pojedynczy `MarsPhoto`obiekt.

```
private val _photos = MutableLiveData<MarsPhoto>()
```

Importuj na `com.example.android.marsphotos.network.MarsPhoto`żądanie.

2. Tuż pod `_photos`deklaracją dodaj publiczne pole zapasowe o nazwie `photos`typu `LiveData<MarsPhoto>`.

```
val photos: LiveData<MarsPhoto> = _photos
```

3. W `getMarsPhotos()`metodzie, wewnątrz `try{}`bloku, znajdź wiersz, który ustawia dane pobrane z usługi sieciowej na`listResult.`

```
try {
  val listResult = MarsApi.retrofitService.getPhotos()
  ...
}
```

4. Przypisz pierwsze pobrane zdjęcie Marsa do nowej zmiennej `_photos`. Zmień `listResult`na `_photos.value`. Przypisz adres URL pierwszych zdjęć w indeksie `0`. Spowoduje to wyświetlenie błędu, naprawisz go później.

```
try {
  _photos.value = MarsApi.retrofitService.getPhotos()[0]
  ...
}
```

5. W następnym wierszu zaktualizuj `status.value`do następującego. Użyj danych z nowej właściwości zamiast `listResult`. Wyświetl adres URL pierwszego obrazu z listy zdjęć.

```
try {
  ...
  _status.value = "   First Mars image URL : ${_photos.value!!.imgSrcUrl}"

}
```

6. Cały `try{}`blok wygląda teraz tak:

```
try {
  _photos.value = MarsApi.retrofitService.getPhotos()[0]
  _status.value = "   First Mars image URL : ${_photos.value!!.imgSrcUrl}"
}
```

7. Uruchom aplikację. `TextView`Teraz wyświetla adres URL pierwszego zdjęcia Marsa . Wszystko, co zrobiłeś do tej pory, to skonfigurowanie `ViewModel`i `LiveData`dla tego adresu URL.

Użyj adapterów do wiązania

Adaptery powiązań to metody z adnotacjami używane do tworzenia niestandardowych ustawień dla niestandardowych właściwości widoku.

Zwykle, gdy ustawiasz atrybut w swoim XML za pomocą kodu: `android:text="Sample Text"`. System Android automatycznie szuka settera o tej samej nazwie co `text`atrybut, który jest ustawiany przez `setText(String: text)`metodę. Metoda `setText(String: text)`jest metodą ustawiającą dla niektórych widoków dostarczanych przez platformę Android Framework. Podobne zachowanie można dostosować za pomocą adapterów powiązań; można podać niestandardowy atrybut i niestandardową logikę, które będą wywoływane przez bibliotekę powiązań danych.

Przykład:

Aby zrobić coś bardziej złożonego niż po prostu wywołanie settera w widoku obrazu, który ustawia obraz możliwy do narysowania. Rozważ wczytanie obrazów z wątku interfejsu użytkownika (wątku głównego) z Internetu. Najpierw wybierz niestandardowy atrybut, aby przypisać obraz do `ImageView`. W poniższym przykładzie jest to `imageUrl`.

```
<ImageView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   app:imageUrl="@{product.imageUrl}"/>
```

Jeśli nie dodasz żadnego dodatkowego kodu, system będzie szukał `setImageUrl(String)`metody `ImageView`i nie znajdzie jej, zgłaszając błąd, ponieważ jest to niestandardowy atrybut, który nie jest dostarczany przez framework. Musisz stworzyć sposób na

zaimplementowanie i ustawienie `app:imageUrl`atrybutu na `ImageView`. W tym celu użyjesz adapterów Binding (metody z adnotacjami).

Przykład adaptera do wiązania:

```
@BindingAdapter("imageUrl")
fun bindImage(imgView: ImageView, imgUrl: String?) {
   imgUrl?.let {
      // Load the image in the background using Coil.
      }
   }
}
```

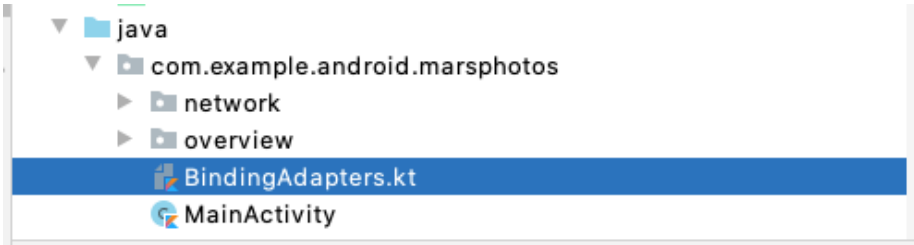
Adnotacja `@BindingAdapter`przyjmuje jako parametr nazwę atrybutu.

W `bindImage`metodzie pierwszy parametr metody to typ docelowego widoku, a drugi to wartość ustawiana w atrybucie.

Wewnątrz metody biblioteka Coil ładuje obraz z wątku interfejsu użytkownika i ustawia go w `ImageView`.

Utwórz adapter do wiązania i użyj Coil

1. W `com.example.android.marsphotos`pakiecie utwórz plik Kotlin o nazwie `BindingAdapters`. Ten plik będzie zawierał adaptery powiązania używane w całej aplikacji.



2. W `BindingAdapters.kt`celu utworzenia `bindImage()`funkcji jako funkcji najwyższego poziomu (nie wewnątrz klasy), która przyjmuje parametry `ImageView`i a `String`jako parametry.

```
fun bindImage(imgView: ImageView, imgUrl: String?) {

}
```

Importuj `android.widget.ImageView`na żądanie.

3. Opisz funkcję za pomocą `@BindingAdapter`. Adnotacja `@BindingAdapter`informuje powiązanie danych o wykonaniu tego adaptera powiązania, gdy element View ma `imageUrl`atrybut.

```
@BindingAdapter("imageUrl")
fun bindImage(imgView: ImageView, imgUrl: String?) {

}
```

Importuj `androidx.databinding.BindingAdapter` na żądanie.

niech zakres funkcji
`let`[jest jedną z funkcji Scope](#) Kotlina, która pozwala wykonać blok kodu w kontekście obiektu. W Kotlinie istnieje pięć funkcji Scope, zapoznaj się z [dokumentacją](#) , aby dowiedzieć się więcej.

Stosowanie:

`let`służy do wywoływania jednej lub więcej funkcji na wynikach łańcuchów wywołań.

Funkcja `let`wraz z operatorem bezpiecznego wywołania (`?.`) służy do wykonania bezpiecznej operacji zerowej na obiekcie. W takim przypadku `let`blok kodu zostanie wykonany tylko wtedy, gdy obiekt nie jest pusty.

4. Wewnątrz `bindImage()`funkcji dodaj `let{}`blok do `imgUrl`argumentu, używając operatora bezpiecznego wywołania.

```
imgUrl?.let {
}
```

5. Wewnątrz `let{}`bloku dodaj następujący wiersz, aby przekonwertować ciąg adresu URL na `Uri`obiekt przy użyciu `toUri()`metody. Aby użyć schematu HTTPS, dołącz `buildUpon.scheme("https")`do `toUri`konstruktora. Zadzwoń `build()`do budowy obiektu.

```
val imgUri = imgUrl.toUri().buildUpon().scheme("https").build()
```

Importuj `androidx.core.net.toUri`na żądanie.

6. Wewnątrz `let{}`bloku, po `imgUri`deklaracji, użyj `load(){}`from [Coil](#) , aby załadować obraz z `imgUri`obiektu do `imgView`.

```
imgView.load(imgUri) {
}
```

Importuj `coil.load`na żądanie.

7. Twoja pełna metoda powinna wyglądać jak poniżej:

```
@BindingAdapter("imageUrl")
fun bindImage(imgView: ImageView, imgUrl: String?) {
    imgUrl?.let {
        val imgUri = imgUrl.toUri().buildUpon().scheme("https").build()
        imgView.load(imgUri)
    }
}
```

Zaktualizuj układ i fragmenty

W poprzedniej sekcji użyłeś biblioteki obrazów Coil do załadowania obrazu. Aby zobaczyć obraz na ekranie, następnym krokiem jest aktualizacja `ImageView`o nowy atrybut, aby wyświetlić *pojedynczy* obraz.

W dalszej części ćwiczenia z programowania użyjesz `res/layout/grid_view_item.xml`jako pliku zasobów układu każdego elementu siatki w `RecyclerView`. W tym zadaniu tymczasowo użyjesz

tego pliku, aby wyświetlić obraz przy użyciu adresu URL obrazu, który pobrałeś w poprzednim zadaniu. Tymczasowo będziesz używać tego pliku układu zamiast fragment_overview.xml.

Uwaga : W tym zadaniu będziesz używać grid_view_item.xmlukładu do tymczasowego wyświetlania pojedynczego obrazu. Odbywa się to w ten sposób, aby uniknąć tworzenia i usuwania tymczasowych plików układu. Wygenerowana nazwa klasy wiążącej będzie GridViewItemBinding, ponieważ nazwa klasy jest oparta na nazwie pliku układu, który jest grid_view_item.xml. Mimo że klasa wiążąca nazywa GridViewItemBindingsię , nie będziesz używać jej do wyświetlania obrazów siatki wewnątrz RecyclerVieww tym zadaniu. Odbywa się to w późniejszym zadaniu.

1. Otwórz res/layout/grid_view_item.xml.
2. Nad <ImageView>elementem dodaj <data>element dla powiązania danych i powiąż z OverviewViewModelklasą:

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

3. Dodaj app:imageUrlatrybut do ImageViewelementu, aby użyć nowego adaptera powiązania ładowania obrazu. Pamiętaj, że photoszawierają listę MarsPhotospobraną z serwera. Przypisz adres URL pierwszego wpisu do imageUrlatrybutu.

```
<ImageView
   android:id="@+id/mars_image"
   ...
   app:imageUrl="@{viewModel.photos.imgSrcUrl}"
   ... />
```

4. Otwórz overview/OverviewFragment.kt. W onCreateView()metodzie skomentuj wiersz, który wypełnia FragmentOverviewBindingklasę i przypisuje ją do zmiennej wiążącej. Zobaczysz błędy spowodowane usunięciem tej linii. To jest tylko tymczasowe; naprawisz je później.

```
//val binding = FragmentOverviewBinding.inflate(inflater)
```

5. Użyj grid_view_item.xmlzamiast fragment_overview.xml.Dodaj następujący wiersz, aby GridViewItemBindingzamiast tego nadmuchać klasę.

```
val binding = GridViewItemBinding.inflate(inflater)
```

Importuj com.example.android.marsphotos. databinding.GridViewItemBindingna żądanie.

Uwaga: ta zmiana może spowodować błędy wiązania danych w Android Studio. Aby rozwiązać te błędy, musisz wyczyścić i odbudować projekt. Wybierz **Kompiluj > Wyczyść projekt** , a następnie **Kompiluj > Przebuduj projekt**

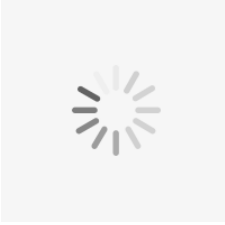
6. Uruchom aplikację. Teraz powinieneś zobaczyć pojedynczy obraz Marsa.

Dodaj obrazy ładowania i błędów

Używając Coil, możesz poprawić wrażenia użytkownika, wyświetlając obraz zastępczy podczas ładowania obrazu i obraz błędu, jeśli ładowanie się nie powiedzie, na przykład jeśli brakuje obrazu lub jest on uszkodzony. W tym kroku dodasz tę funkcję do adaptera powiązania.

1. Otwórz `res/drawable/ic_broken_image.xml` i kliknij kartę **Projekt** po prawej stronie. W przypadku obrazu błędu używasz ikony uszkodzonego obrazu, która jest dostępna we wbudowanej bibliotece ikon. Ten wektor do rysowania używa `android:tint` atrybutu do pokolorowania ikony na szaro.

2. Otwórz `res/drawable/loading_animation.xml`. Ten rysowalny jest animacją, która obraca rysowalny obraz `loading_img.xml` wokół punktu środkowego. (Nie widzisz animacji w podglądzie).



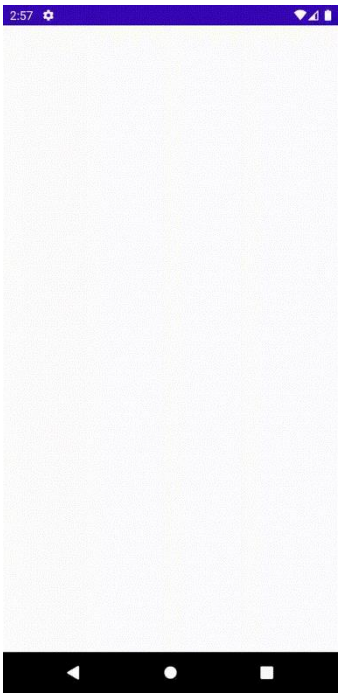
3. Wróć do BindingAdapters.ktakt. W bindImage()metodzie zaktualizuj wywołanie, aby dodać końcowe lambda w następujący sposób: Ten kod ustawia obraz ładowania zastępczego, który ma być używany podczas ładowania (do rysowania). Ten kod ustawia również obraz do użycia, jeśli ładowanie obrazu nie powiedzie się (do rysowania).imgView.*load*(imgUri)loading_animationbroken_image

```
imgView.load(imgUri) {
  placeholder(R.drawable.loading_animation)
  error(R.drawable.ic_broken_image)
}
```

4. Kompletna bindImage()metoda wygląda teraz tak:

```
@BindingAdapter("imageUrl")
fun bindImage(imgView: ImageView, imgUrl: String?) {
   imgUrl?.let {
      val imgUri = imgUrl.toUri().buildUpon().scheme("https").build()
      imgView.load(imgUri) {
         placeholder(R.drawable.loading_animation)
         error(R.drawable.ic_broken_image)
      }
   }
}
```

5. Uruchom aplikację. W zależności od szybkości połączenia sieciowego możesz przez chwilę zobaczyć obraz ładowania podczas pobierania cewki i wyświetlania obrazu właściwości. Ale nie zobaczysz jeszcze ikony uszkodzonego obrazu, nawet jeśli wyłączysz sieć — naprawisz to w ostatnim zadaniu ćwiczenia kodowania.



6. Cofnij tymczasowe zmiany wprowadzone w programie overview/OverviewFragment.kt. W metodzie onCreateview()odkomentuj linię, która się nadmuchuje FragmentOverviewBinding. Usuń lub skomentuj linię, która się nadmuchuje GridViewIteMBinding.

```
val binding = FragmentOverviewBinding.inflate(inflater)
 // val binding = GridViewItemBinding.inflate(inflater)
```

4. Wyświetl siatkę obrazów za pomocą RecyclerView

Twoja aplikacja ładuje teraz zdjęcie Marsa z internetu. Korzystając z danych z pierwszego MarsPhotoelementu listy, utworzyłeś LiveDatawłaściwość w ViewModeli użyłeś adresu URL obrazu z danych zdjęcia Marsa do wypełnienia ImageView. Ale celem jest, aby Twoja aplikacja wyświetlała siatkę obrazów, więc w tym zadaniu użyjesz RecyclerViewmenedżera układu siatki, aby wyświetlić siatkę obrazów.

Zaktualizuj model widoku

W poprzednim zadaniu, w OverviewViewModel, dodałeś LiveDataobiekt o nazwie _photos, który zawiera jeden MarsPhotoobiekt — pierwszy na liście odpowiedzi z usługi sieciowej. W tym kroku zmienisz to, LiveDataaby przechowywać całą listę MarsPhotoobiektów.

1. Otwórz overview/OverviewViewModel.kt.
2. Zmień _photostyp na listę MarsPhotoobiektów.

```
private val _photos = MutableLiveData<List<MarsPhoto>>()
```

3. photosZastąp również typ właściwości kopii zapasowej na List<MarsPhoto>typ:

```
 val photos: LiveData<List<MarsPhoto>> = _photos
```

4. Przewiń w dół do try {}bloku wewnątrz getMarsPhotos()metody. TheMarsApi.retrofitService.getPhotos()

   zwraca listę MarsPhotoobiektów, możesz po prostu przypisać ją do _photos.value.

```
_photos.value = MarsApi.retrofitService.getPhotos()
_status.value = "Success: Mars properties retrieved"
```

5. Cały try/catchblok wygląda teraz tak:

```
try {
    _photos.value = MarsApi.retrofitService.getPhotos()
    _status.value = "Success: Mars properties retrieved"
} catch (e: Exception) {
    _status.value = "Failure: ${e.message}"
}
```

Układ siatki

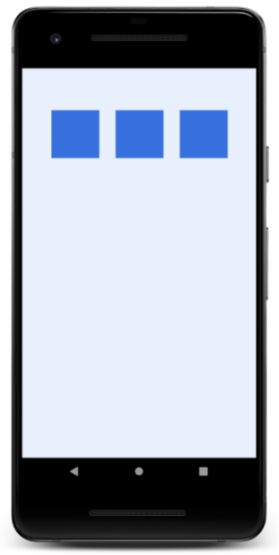
For GridLayoutManagerprzedstawia RecyclerViewdane w postaci przewijanej siatki, jak pokazano poniżej.

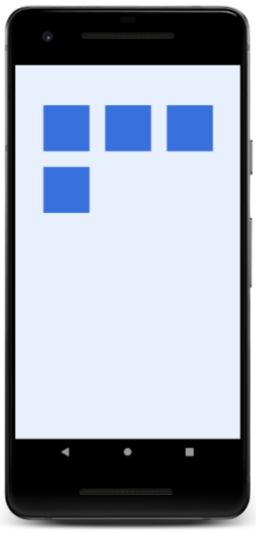
Z perspektywy projektowania układ siatki najlepiej sprawdza się w przypadku list, które mogą być reprezentowane jako ikony lub obrazy, na przykład listy w aplikacji do przeglądania zdjęć Mars.

Jak układ siatki rozmieszcza elementy

Układ siatki rozmieszcza elementy w siatce wierszy i kolumn. Zakładając przewijanie w pionie, domyślnie każdy element w rzędzie zajmuje jeden „rozpiętość”. Element może zajmować wiele przęseł. W poniższym przypadku jeden rozpiętość odpowiada szerokości jednego słupa, czyli 3.

W dwóch przykładach pokazanych poniżej każdy rząd składa się z trzech przęseł. Domyślnie GridLayoutManagerkażdy element jest rozmieszczany w jednym rozpiętości, aż do określonej liczby rozpiętości. Kiedy osiągnie liczbę rozpiętości, zawija się do następnej linii.





Dodaj opinię Recyklera

W tym kroku zmienisz układ aplikacji, aby używał widoku recyklingu z układem siatki zamiast widoku pojedynczego obrazu.

1. Otwórz `layout/grid_view_item.xml`. Usuń `viewModel`zmienną danych.
2. Wewnątrz `<data>`tagu dodaj następującą `photo`zmienną typu `MarsPhoto`.

```
<data>
  <variable
    name="photo"
    type="com.example.android.marsphotos.network.MarsPhoto" />
</data>
```

3. W `<ImageView>`, zmień `app:imageUrl`atrybut, aby odwoływał się do adresu URL obrazu w `MarsPhoto`obiekcie. Te zmiany cofają tymczasowe zmiany wprowadzone w poprzednim zadaniu.

```
app:imageUrl="@{photo.imgSrcUrl}"
```

4. Otwórz `layout/fragment_overview.xml`. Usuń cały `<TextView>`element.
5. Zamiast tego dodaj następujący `<RecyclerView>`element. Ustaw identyfikator na `photos_grid`, `width`a `height`atrybuty na `0dp`, aby wypełniał rodzica `ConstraintLayout`. Będziesz używać układu siatki, więc ustaw `layoutManager`atrybut na `androidx.recyclerview.widget.GridLayoutManager`. Ustaw na `spanCount`, `2`aby mieć dwie kolumny.

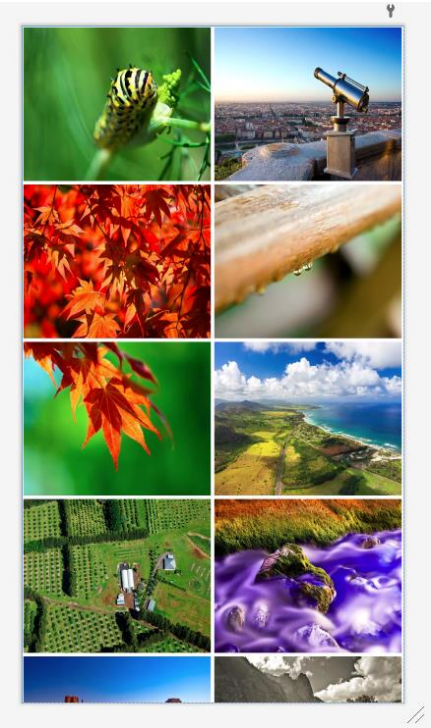
```
<androidx.recyclerview.widget.RecyclerView
   android:id="@+id/photos_grid"
   android:layout_width="0dp"
   android:layout_height="0dp"
   app:layoutManager=
     "androidx.recyclerview.widget.GridLayoutManager"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent"
   app:spanCount="2" />
```

6. Aby zobaczyć podgląd tego, jak powyższy kod wygląda w widoku **Projekt** , użyj , `tools:itemCount`aby ustawić liczbę elementów wyświetlanych w naszym układzie na `16`. Atrybut określa liczbę elementów , `itemCount`które edytor układu powinien renderować w oknie **podglądu** . Ustaw układ elementów listy na `grid_view_item`using `tools:listitem`.

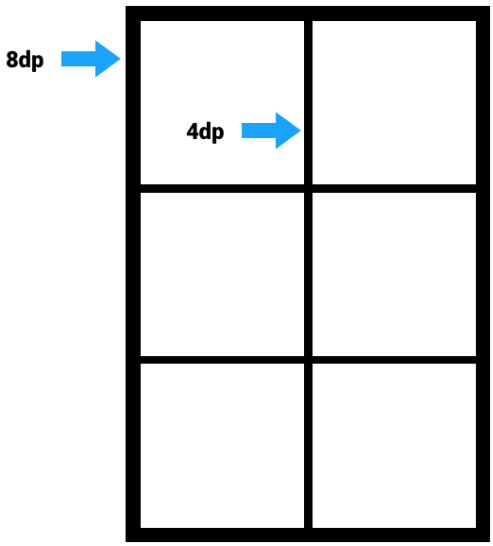
```
<androidx.recyclerview.widget.RecyclerView
        ...
        tools:itemCount="16"
        tools:listitem="@layout/grid_view_item" />
```

7. Przejdź do widoku **Projekt** , powinieneś zobaczyć podgląd podobny do poniższego zrzutu ekranu. To nie wygląda jak zdjęcia Marsa, ale pokaże Ci, jak będzie wyglądał układ Twojej siatki

recyklera. Podgląd używa dopełnienia i `grid_view_item`układu dla każdego elementu siatki `recyclerview`.



8. Zgodnie z [wytycznymi Material design](), powinieneś mieć `8dp`miejsce na górze, na dole i po bokach listy oraz `4dp`przestrzeń między elementami. Możesz to osiągnąć za pomocą kombinacji dopełnienia w `fragment_overview.xml`układzie i w `grid_view_item.xml`układzie.



9. Otwórz `layout/gridview_item.xml`. Zwróć uwagę na `padding`atrybut, który już masz `2dp`z dopełnieniem między zewnętrzną częścią elementu a zawartością. Dzięki temu uzyskamy `4dp`przestrzeń między zawartością przedmiotu i `2dp`wzdłuż zewnętrznych krawędzi, co oznacza, że będziemy potrzebować dodatkowej `6dp`wyściółki na zewnętrznych krawędziach, aby dopasować ją do wytycznych projektowych.
10. Wróć do `layout/fragment_overview.xml`. Dodaj `6dp`wyściółkę dla `RecyclerView`, dzięki czemu będziesz mieć `8dp`na zewnątrz i `4dp`wewnątrz jako wytyczne.

```
<androidx.recyclerview.widget.RecyclerView
        ...
        android:padding="6dp"
        ... />
```

11. Cały `<RecyclerView>`element powinien wyglądać następująco.

```
<androidx.recyclerview.widget.RecyclerView
   android:id="@+id/photos_grid"
   android:layout_width="0dp"
   android:layout_height="0dp"
   android:padding="6dp"
   app:layoutManager=
      "androidx.recyclerview.widget.GridLayoutManager"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent"
   app:spanCount="2"
   tools:itemCount="16"
   tools:listitem="@layout/grid_view_item"  />
```

Dodaj adapter siatki zdjęć

Teraz `fragment_overview`układ ma układ `RecyclerView`z siatką. W tym kroku powiążesz dane pobrane z serwera WWW z `RecyclerView`adapterem `RecyclerView`.

Uwaga: to może być dobry moment na przejrzenie `RecyclerView`ćwiczeń z programowania w tej ścieżce !

ListAdapter (odświeżanie)

`ListAdapter`jest podklasą `RecyclerView.Adapter`klasy do prezentowania danych List w a `RecyclerView`, w tym obliczania różnic między Listami w wątku w tle.

W tej aplikacji będziesz korzystać z `DiffUtil`implementacji w `ListAdapter`.Zaletą używania `DiffUtil`jest to, że za każdym razem, gdy jakiś element `RecyclerView`zostanie dodany, usunięty lub zmieniony, cała lista nie jest odświeżana. Odświeżane są tylko te pozycje, które zostały zmienione.

Dodaj `ListAdapter`do swojej aplikacji.

1. W `overview`pakiecie utwórz klasę Kotlin o nazwie `PhotoGridAdapter`.
2. Rozszerz `PhotoGridAdapter`klasę z `ListAdapter`o parametry konstruktora pokazane poniżej. Klasa `PhotoGridAdapter`extends `ListAdapter`, której konstruktor potrzebuje typu elementu listy, posiadacza widoku i `DiffUtil.ItemCallback`implementacji.

```
class PhotoGridAdapter : ListAdapter<MarsPhoto,
        PhotoGridAdapter.MarsPhotoViewHolder>(DiffCallback) {
}
```

Zaimportuj klasy androidx.recyclerview.widget.ListAdapteri na com.example.android.marsphoto.network.MarsPhotożądanie. W kolejnych krokach zaimplementujesz inne brakujące implementacje tego konstruktora, które powodują błędy.

3. Aby rozwiązać powyższe błędy, w tym kroku dodasz wymagane metody i zaimplementujesz je w dalszej części tego zadania. Kliknij PhotoGridAdapterklasę, kliknij czerwoną żarówkę, z menu wybierz **Zaimplementuj członków.** W wyświetlonym wyskakującym okienku wybierz ListAdaptermetody, onCreateViewHolder()oraz onBindViewHolder(). Android Studio nadal będzie wyświetlał błędy, które naprawisz do końca tego zadania.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
PhotoGridAdapter.MarsPhotoViewHolder {
  TODO("Not yet implemented")
}

override fun onBindViewHolder(holder: PhotoGridAdapter.MarsPhotoViewHolder, position: Int) {
  TODO("Not yet implemented")
}
```

Do wdrożenia onCreateViewHolderi onBindViewHoldermetod potrzebujesz MarsPhotoViewHolder, które dodasz w następnym kroku.

4. Wewnątrz PhotoGridAdapter, dodaj wewnętrzną definicję klasy dla MarsPhotoViewHolder, która rozszerza RecyclerView.ViewHolder. Potrzebujesz GridViewItemBindingzmiennej do powiązania z MarsPhotoukładem, więc przekaż zmienną do MarsPhotoViewHolder. Klasa bazowa ViewHolderwymaga widoku w swoim konstruktorze, przekazujesz go do wiążącego widoku głównego.

```
class MarsPhotoViewHolder(private var binding:
            GridViewItemBinding):
    RecyclerView.ViewHolder(binding.root) {
}
```

Importuj androidx.recyclerview.widget.RecyclerViewi com.example.android.marsrealestate.databinding.GridViewItemBindingna żądanie.

5. W MarsPhotoViewHolderprogramie utwórz bind()metodę, która przyjmuje MarsPhotoobiekt jako argument i ustawia binding.propertygo na ten obiekt. Wywołaj executePendingBindings()po ustawieniu właściwości, co powoduje natychmiastowe wykonanie aktualizacji.

```
fun bind(MarsPhoto: MarsPhoto) {
  binding.photo = MarsPhoto
  binding.executePendingBindings()
}
```

Uwaga: ta zmiana może spowodować błędy wiązania danych w Android Studio. Aby rozwiązać te błędy, może być konieczne wyczyszczenie i odbudowanie aplikacji.

6. Pozostając w PhotoGridAdapterklasie w onCreateViewHolder(), usuń TODO i dodaj wiersz pokazany poniżej. Metoda onCreateViewHolder()musi zwrócić new MarsPhotoViewHolder, utworzony przez wypełnienie GridViewItemBindingi użycie kontekstu LayoutInflaternadrzędnego .ViewGroup

```
return MarsPhotoViewHolder(GridViewItemBinding.inflate(
    LayoutInflater.from(parent.context)))
```

Importuj `android.view.LayoutInflater` na żądanie.

7. W `onBindViewHolder()` metodzie usuń TODO i dodaj linie pokazane poniżej. Tutaj wywołujesz, `getItem()` aby pobrać `MarsPhoto` obiekt skojarzony z bieżącą `RecyclerView` pozycją, a następnie przekazujesz tę właściwość do `bind()` metody w `MarsPhotoViewHolder`.

```
val marsPhoto = getItem(position)
holder.bind(marsPhoto)
```

8. Wewnątrz `PhotoGridAdapter` dodaj definicję obiektu towarzyszącego dla `DiffCallback`, jak pokazano poniżej.
   Obiekt `DiffCallback` rozszerza `DiffUtil.ItemCallback` się o ogólny typ obiektu, który chcesz porównać — `MarsPhoto`. Porównasz dwa obiekty fotograficzne Marsa w tej implementacji.

```
companion object DiffCallback : DiffUtil.ItemCallback<MarsPhoto>() {
}
```

Importuj `androidx.recyclerview.widget.DiffUtil` na żądanie.

9. Naciśnij czerwoną żarówkę, aby zaimplementować metody porównawcze dla `DiffCallback` obiektu, którymi są `areItemsTheSame()` i `areContentsTheSame()`.

```
override fun areItemsTheSame(oldItem: MarsPhoto, newItem: MarsPhoto): Boolean {
    TODO("Not yet implemented")
}

override fun areContentsTheSame(oldItem: MarsPhoto, newItem: MarsPhoto): Boolean {
    TODO("Not yet implemented") }
```

10. W `areItemsTheSame()` metodzie usuń `TODO`. Ta metoda jest wywoływana przez, `DiffUtil` aby zdecydować, czy dwa obiekty reprezentują ten sam Item. `DiffUtil` używa tej metody, aby dowiedzieć się, czy nowy `MarsPhoto` obiekt jest taki sam jak stary `MarsPhoto` obiekt. Identyfikator każdej pozycji (`MarsPhoto` obiektu) jest unikalny. Porównaj identyfikatory `oldItem` i `newItem` i zwróć wynik.

```
override fun areItemsTheSame(oldItem: MarsPhoto, newItem: MarsPhoto): Boolean {
    return oldItem.id == newItem.id
}
```

11. W `areContentsTheSame()` programie usuń `TODO`. Ta metoda jest wywoływana przez `DiffUtil` , gdy chce sprawdzić, czy dwa elementy mają te same dane. Ważnymi danymi w MarsPhoto jest adres URL obrazu. Porównaj adresy URL `oldItem` i `newItem` i zwróć wynik.

```
override fun areContentsTheSame(oldItem: MarsPhoto, newItem: MarsPhoto): Boolean {
    return oldItem.imgSrcUrl == newItem.imgSrcUrl
}
```

Upewnij się, że możesz skompilować i uruchomić aplikację bez żadnych błędów, ale emulator wyświetla pusty ekran. Masz gotowy recyclerview, ale nie są do niego przekazywane żadne dane, które zaimplementujesz w następnym kroku.

Dodaj adapter do wiązania i połącz części

W tym kroku użyjesz a BindingAdapterdo zainicjowania PhotoGridAdapterlisty MarsPhotoobiektów. Użycie a BindingAdapterdo ustawienia RecyclerViewdanych powoduje, że powiązanie danych automatycznie obserwuje LiveDatalistę MarsPhotoobiektów. Następnie adapter powiązania jest wywoływany automatycznie po MarsPhotozmianie listy.

1. Otwórz BindingAdapters.kt.
2. Na końcu pliku dodaj metodę, która jako argumenty bindRecyclerView()przyjmuje RecyclerViewai listę obiektów. MarsPhotoOpisz tę metodę za @BindingAdapterpomocą listDataatrybutu with .

```
@BindingAdapter("listData")
fun bindRecyclerView(recyclerView: RecyclerView,
    data: List<MarsPhoto>?) {
}
```

Importuj androidx.recyclerview.widget.RecyclerViewi com.example.android.marsphotos.network.MarsPhotona żądanie.

3. Wewnątrz bindRecyclerView()funkcji rzutuj i przypisz recyclerView.adapterdo PhotoGridAdapternowej valwłaściwościadapter.

```
val adapter = recyclerView.adapter as PhotoGridAdapter
```

4. Na końcu bindRecyclerView()funkcji wywołaj adapter.submitList()dane z listy zdjęć Marsa. To informuje, RecyclerViewkiedy nowa lista jest dostępna.

```
adapter.submitList(data)
```

Importuj com.example.android.marsrealestate.overview.PhotoGridAdapterna żądanie.

5. Kompletny bindRecyclerViewadapter wiązania powinien wyglądać tak:

```
@BindingAdapter("listData")
fun bindRecyclerView(recyclerView: RecyclerView,
                     data: List<MarsPhoto>?) {
  val adapter = recyclerView.adapter as PhotoGridAdapter
  adapter.submitList(data)

}
```

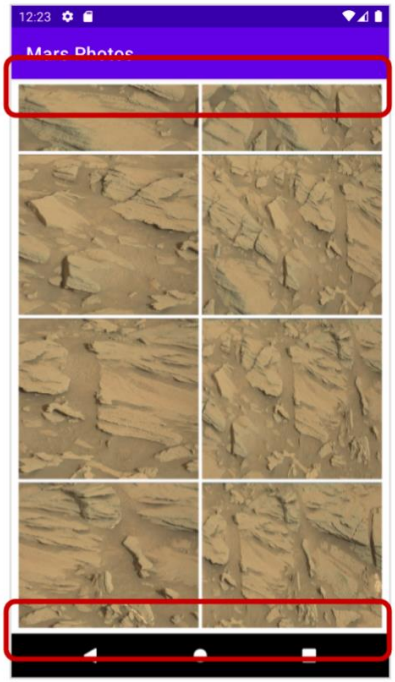
6. Aby wszystko połączyć, otwórz res/layout/fragment_overview.xml. Dodaj app:listDataatrybut do RecyclerViewelementu i ustaw viewmodel.photosprzy użyciu powiązania danych. Jest to podobne do tego, co zrobiłeś ImageVieww poprzednim zadaniu.

```
app:listData="@{viewModel.photos}"
```

7. Otwórz `overview/OverviewFragment.kt`. W `onCreateView()`tuż przed `return`instrukcją zainicjuj `RecyclerView`adapter w `binding.photosGrid`nowym `PhotoGridAdapter`obiekcie.

```
binding.photosGrid.adapter = PhotoGridAdapter()
```

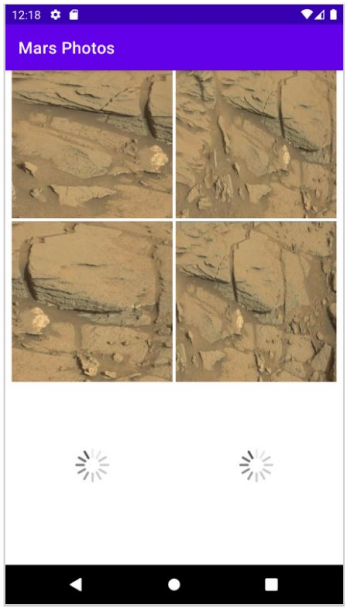
8. Uruchom aplikację. Powinieneś zobaczyć siatkę przewijających się obrazów Marsa. Podczas przewijania, aby zobaczyć nowe obrazy, wygląda to trochę dziwnie. Wypełnienie pozostaje na górze i na dole podczas `RecyclerView`przewijania, więc nigdy nie wygląda na to, że lista jest przewijana pod paskiem akcji.



9. Aby to naprawić, musisz powiedzieć programowi, `RecyclerView`aby nie przycinał wewnętrznej zawartości do wypełnienia za pomocą atrybutu android:clipToPadding. To sprawia, że rysuje widok przewijania w obszarze dopełnienia. Wróć do `layout/fragment_overview.xml`. Dodaj `android:clipToPadding`atrybut dla `RecyclerView`, ustaw go na `false`.

```
<androidx.recyclerview.widget.RecyclerView
        ...
        android:clipToPadding="false"
        ... />
```

10. Uruchom swoją aplikację. Zauważ, że aplikacja wyświetla również ikonę postępu ładowania przed wyświetleniem samego obrazu, zgodnie z oczekiwaniami. To jest zastępczy obraz ładujący, który został przekazany do biblioteki obrazów Coil.



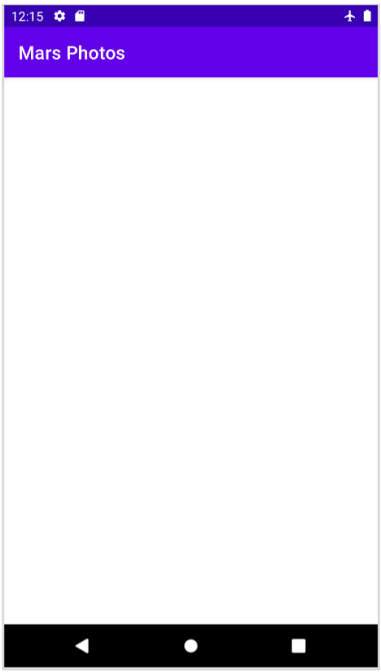
11. Gdy aplikacja jest uruchomiona, włącz tryb samolotowy. Przewiń obrazy w emulatorze. Obrazy, które nie zostały jeszcze załadowane, są wyświetlane jako ikony uszkodzonych obrazów. Jest to obraz, który można narysować, który został przekazany do biblioteki obrazów Coil w celu wyświetlenia na wypadek, gdyby jakikolwiek błąd sieci lub obraz nie mógł zostać pobrany.

Gratulacje, prawie skończyłeś. W następnym końcowym zadaniu jeszcze bardziej poprawisz wrażenia użytkownika, dodając więcej obsługi błędów do aplikacji.

5. Dodaj obsługę błędów w RecyclerView

Aplikacja MarsPhotos wyświetla ikonę uszkodzonego obrazu, gdy nie można pobrać obrazu. Ale gdy nie ma sieci, aplikacja wyświetla pusty ekran. W następnym kroku zweryfikujesz pusty ekran.

1. Włącz tryb samolotowy na swoim urządzeniu lub emulatorze. Uruchom aplikację z Android Studio. Zwróć uwagę na pusty ekran.



To nie jest wspaniałe doświadczenie użytkownika. W tym zadaniu dodasz podstawową obsługę błędów, aby dać użytkownikowi lepsze wyobrażenie o tym, co się dzieje. Jeśli internet nie jest dostępny, aplikacja wyświetli ikonę błędu połączenia, a podczas pobierania `MarsPhoto`listy aplikacja wyświetli animację ładowania.

Dodaj status do ViewModel

W tym zadaniu utworzysz właściwość w `OverviewViewModel`celu reprezentowania stanu żądania internetowego. Należy wziąć pod uwagę trzy stany — ładowanie, sukces i porażkę. Stan ładowania ma miejsce podczas oczekiwania na dane. Status sukcesu jest wtedy, gdy pomyślnie pobieramy dane z usługi sieciowej. Stan awarii wskazuje na wszelkie błędy sieci lub połączenia.

Zajęcia Enum w Kotlinie

Aby przedstawić te trzy stany w swojej aplikacji, użyjesz `enum`. `enum`jest skrótem od enumeration, co oznacza uporządkowaną listę wszystkich elementów w kolekcji. Każda `enum`stała jest obiektem `enum`klasy.

W Kotlinie an `enum`to typ danych, który może przechowywać zestaw stałych. Są one definiowane przez dodanie słowa kluczowego `enum`przed definicją klasy, jak pokazano poniżej. Stałe wyliczenia są oddzielone przecinkami.

Definicja:

```
enum class Direction {
   NORTH, SOUTH, WEST, EAST
}
```

Stosowanie:

```
var direction : Direction = Direction.NORTH
```

Jak pokazano powyżej, do `enum`obiektów można się odwoływać, używając nazwy klasy, po której następuje operator kropki (.) i nazwa stałej.

Uwaga : stałe wyliczenia można inicjować za pomocą wartości niestandardowych przy użyciu parametru konstruktora w definicji klasy. To wykracza poza zakres tego ćwiczenia z kodowania, zapoznaj się z dokumentacją Kotlina, aby dowiedzieć się więcej.

Dodaj definicję klasy wyliczenia z wartościami stanu w Viewmodel.

1. Otwórz `overview/OverviewViewModel.kt`. Na górze pliku (po imporcie, przed definicją klasy) dodaj an, `enum`aby reprezentować wszystkie dostępne statusy:

```
enum class MarsApiStatus { LOADING, ERROR, DONE }
```

2. Przewiń do definicji `_status`i `status`właściwości, zmień typy z `String`na `MarsApiStatus. MarsApiStatus`to klasa wyliczeniowa zdefiniowana w poprzednim kroku.

```
private val _status = MutableLiveData<MarsApiStatus>()


val status: LiveData<MarsApiStatus> = _status
```

3. W `getMarsPhotos()`metodzie zmień `"Success: ..."`ciąg na `MarsApiStatus.DONE`stan, a `"Failure..."`ciąg na `MarsApiStatus.ERROR`.

```
try {
   _photos.value = MarsApi.retrofitService.getPhotos()
   _status.value = MarsApiStatus.DONE
} catch (e: Exception) {
    _status.value = MarsApiStatus.ERROR
}
```

4. Ustaw status na `MarsApiStatus.LOADING`powyżej `try {}`bloku. To jest stan początkowy, gdy współprogram jest uruchomiony i czekasz na dane. Cały blok wygląda teraz tak:*`viewModelScope.launch {}`*

```
viewModelScope.launch {
        _status.value = MarsApiStatus.LOADING
        try {
           _photos.value = MarsApi.retrofitService.getPhotos()
           _status.value = MarsApiStatus.DONE
        } catch (e: Exception) {
           _status.value = MarsApiStatus.ERROR
        }
    }
```

5. Po stanie błędu w `catch {}`bloku ustaw `_photos`pustą listę. Spowoduje to wyczyszczenie widoku Recyklera.

```
} catch (e: Exception) {
  _status.value = MarsApiStatus.ERROR
  _photos.value = listOf()
}
```

6. Kompletna `getMarsPhotos()` metoda powinna wyglądać tak:

```
private fun getMarsPhotos() {
  viewModelScope.launch {
      _status.value = MarsApiStatus.LOADING
      try {
        _photos.value = MarsApi.retrofitService.getPhotos()
        _status.value = MarsApiStatus.DONE
      } catch (e: Exception) {
        _status.value = MarsApiStatus.ERROR
        _photos.value = listOf()
      }
   }
}
```

Zdefiniowałeś stany wyliczeniowe dla stanu i ustawiłeś stan ładowania na początku współprogramu, ustawiłeś jako gotowe, gdy aplikacja zakończy pobieranie danych z serwera WWW i ustawiłeś błąd, gdy występuje wyjątek. W następnym zadaniu użyjesz adaptera powiązania, aby wyświetlić odpowiednie ikony.

Dodaj adapter powiązania dla stanu ImageView

Skonfigurowałeś `MarsApiStatus` w `OverviewViewModel`, używając zestawu `enum` stanów. W tym kroku sprawisz, że pojawi się w aplikacji. Używasz adaptera powiązania dla `ImageView`, do wyświetlania ikon dla stanów ładowania i błędów. Gdy aplikacja jest w stanie ładowania lub w stanie błędu, `ImageView` powinien być widoczny symbol . Po zakończeniu ładowania aplikacji ikona `ImageView` powinna być niewidoczna.

1. Otwórz `BindingAdapters.kt`, przewiń do końca pliku, aby dodać kolejny adapter. Dodaj nowy adapter powiązania o nazwie `bindStatus()`, który przyjmuje wartość `ImageView` i `MarsApiStatus` jako argumenty. Opisz metodę z `@BindingAdapter` przekazaniem niestandardowego atrybutu `marsApiStatus` jako parametru.

```
@BindingAdapter("marsApiStatus")
fun bindStatus(statusImageView: ImageView,
        status: MarsApiStatus?) {
}
```

Importuj `com.example.android.marsrealestate.overview.MarsApiStatus` na żądanie.

2. Dodaj `when {}` blok wewnątrz `bindStatus()` metody, aby przełączać się między różnymi stanami.

```
when (status) {
```

```
}
```

3. Wewnątrz `when {}`dodaj przypadek dla stanu ładowania (`MarsApiStatus.LOADING`). W tym stanie ustaw jako `ImageView`widoczny i przypisz mu animację ładowania. Jest to ta sama animacja, którą można narysować w przypadku zwoju w poprzednim zadaniu.

```
when (status) {
  MarsApiStatus.LOADING -> {
    statusImageView.visibility = View.VISIBLE
    statusImageView.setImageResource(R.drawable.loading_animation)
  }
}
```

Importuj `android.view.View`na żądanie.

4. Dodaj wielkość liter dla stanu błędu, którym jest `MarsApiStatus.ERROR`. Podobnie do tego, co zrobiłeś dla `LOADING`stanu, ustaw status `ImageView`na widoczny i użyj rysowania błędu połączenia.

```
MarsApiStatus.ERROR -> {
  statusImageView.visibility = View.VISIBLE
  statusImageView.setImageResource(R.drawable.ic_connection_error)
}
```

5. Dodaj przypadek dla stanu gotowe, czyli `MarsApiStatus.DONE`. Tutaj masz pomyślną odpowiedź, więc ustaw widoczność statusu `ImageView`na , `View.GONE`aby go ukryć.

```
MarsApiStatus.DONE -> {
  statusImageView.visibility = View.GONE
}
```

Skonfigurowano adapter powiązania dla widoku obrazu stanu. W następnym kroku dodasz widok obrazu, który korzysta z nowego adaptera powiązania.

Dodaj status ImageView

W tym kroku dodasz widok obrazu, w `fragment_overview.xml`którym będzie widoczny wcześniej zdefiniowany status.

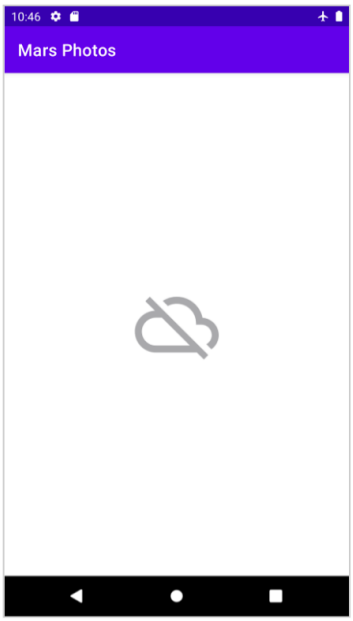
1. Otwórz `res/layout/fragment_overview.xml`. Wewnątrz `ConstraintLayout`, pod `RecyclerView`elementem, dodaj `ImageView`pokazany poniżej.

```
<ImageView
  android:id="@+id/status_image"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent"
```

app:marsApiStatus="@{viewModel.status}" />

Powyższe `ImageView`ma takie same ograniczenia jak `RecyclerView`. Jednak szerokość i wysokość służą `wrap_content`do wyśrodkowania obrazu, a nie do rozciągania obrazu w celu wypełnienia widoku. Zwróć także uwagę na `app:marsApiStatus`atrybut ustawiony na `viewModel.status`, który wywołuje twoją `BindingAdapter`właściwość status w `ViewModel`zmianach.

2. Aby przetestować powyższy kod, zasymuluj błąd połączenia sieciowego, włączając tryb samolotowy w emulatorze lub urządzeniu. Skompiluj i uruchom aplikację i zauważ, że pojawia się obraz błędu:



3. Dotknij przycisku Wstecz, aby zamknąć aplikację i wyłączyć tryb samolotowy. Użyj ekranu z ostatnimi, aby zwrócić aplikację. W zależności od szybkości połączenia sieciowego, gdy aplikacja wysyła zapytanie do usługi sieci Web, zanim obrazy zaczną się ładować, może pojawić się bardzo krótki wskaźnik ładowania.

   Gratulujemy ukończenia tego ćwiczenia z programowania i stworzenia aplikacji MarsPhotos! Nadszedł czas, aby pochwalić się swoją aplikacją za pomocą prawdziwych zdjęć Marsa z rodziną i przyjaciółmi.

6. Kod rozwiązania

Kod rozwiązania dla tego ćwiczenia programowania znajduje się w projekcie pokazanym poniżej. Użyj **głównej** gałęzi, aby pobrać lub pobrać kod.

Adres URL kodu rozwiązania:

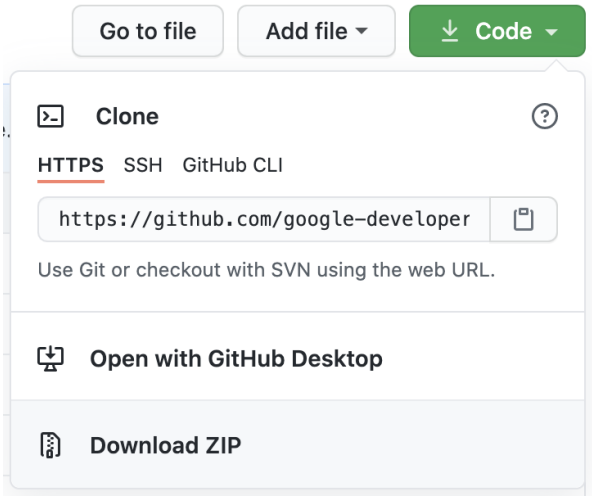
https://github.com/google-developer-training/android-basics-kotlin-mars-photos-app/tree/main

Nazwa oddziału: główna

Aby uzyskać kod tego ćwiczenia z programowania i otworzyć go w Android Studio, wykonaj następujące czynności.

Pobierz kod

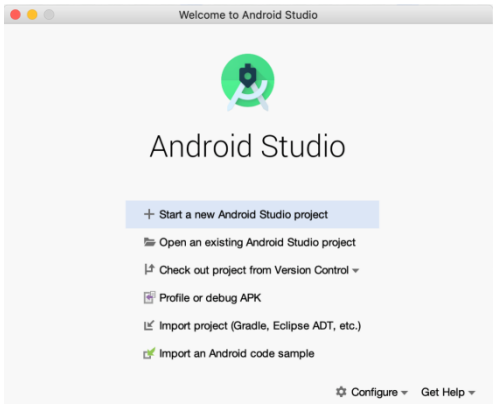
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



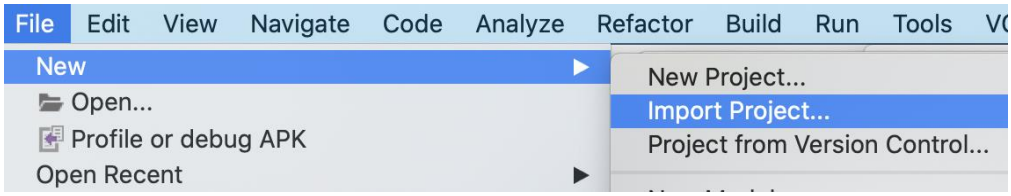
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego pliki projektu.

Otwórz projekt w Android Studio

1. Uruchom Android Studio.
2. W oknie **Witamy w Android Studio** kliknij **Otwórz istniejący projekt Android Studio.**



Uwaga: jeśli Android Studio jest już otwarte, wybierz opcję menu **Plik** > **Nowy** > **Importuj projekt** .



3. W oknie dialogowym **Importuj projekt** przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.
6. Kliknij przycisk **Uruchom**  , aby skompilować i uruchomić aplikację. Upewnij się, że kompiluje się zgodnie z oczekiwaniami.
7. Przeglądaj pliki projektu w oknie narzędzia **Projekt** , aby zobaczyć, jak skonfigurowana jest aplikacja.

7. Podsumowanie

- Biblioteka Coil upraszcza proces zarządzania obrazami, takimi jak pobieranie, buforowanie, dekodowanie i obrazy pamięci podręcznej w Twojej aplikacji.
- Karty powiązań to metody rozszerzające, które znajdują się między widokiem a powiązanymi danymi tego widoku. Adaptery powiązania zapewniają niestandardowe zachowanie w przypadku zmiany danych, na przykład wywołanie Coil w celu załadowania obrazu z adresu URL do pliku `ImageView`.
- Karty powiązań to metody rozszerzające opatrzone `@BindingAdapter`adnotacją.
- Aby wyświetlić siatkę obrazów, użyj a `RecyclerView`z `GridLayoutManager`.
- Aby zaktualizować listę właściwości po jej zmianie, użyj adaptera powiązania między `RecyclerView`układem a układem.

8. Dowiedz się więcej

Dokumentacja dla programistów Androida:

- ZobaczPrzegląd Modelu
- Przegląd danych na żywo
- Współprogramy, oficjalna dokumentacja
- Adaptery do wiązania

Inny:

- Cewka
- Wiązanie adapterów z Kotlin

Debuguj z punktami przerwania

1. Zanim zaczniesz

Do tego momentu większość początkujących programistów prawdopodobnie będzie świadoma debugowania za pomocą instrukcji Log. Jeśli ukończyłeś część 1, będziesz również wiedział, jak czytać ślady stosu i badać komunikaty o błędach. Chociaż oba są potężnymi narzędziami do debugowania, nowoczesne IDE zapewniają więcej funkcji, aby proces debugowania był bardziej wydajny.

W tej lekcji dowiesz się o zintegrowanym debugerze Android Studio, jak wstrzymywać wykonywanie aplikacji i jak wykonywać pojedyncze wiersze kodu na raz, aby wskazać dokładne źródło błędu. Dowiesz się również, jak korzystać z funkcji o nazwie Zegarki i śledzić określone zmienne zamiast dodawać określone instrukcje dziennika.

Warunki wstępne

- Wiesz, jak poruszać się po projekcie w Android Studio.
- Znajomość zasad logowania w Kotlinie.

Czego się nauczysz

- Jak dołączyć debuger do uruchomionej aplikacji.
- Użyj punktów przerwania, aby wstrzymać uruchomioną aplikację i sprawdzić kod po jednym wierszu na raz.
- Dodaj wyrażenia warunkowe do punktów przerwania, aby zaoszczędzić czas na debugowanie.
- Dodaj zmienne do okienka Zegarki, aby ułatwić debugowanie.

Co będziesz potrzebował

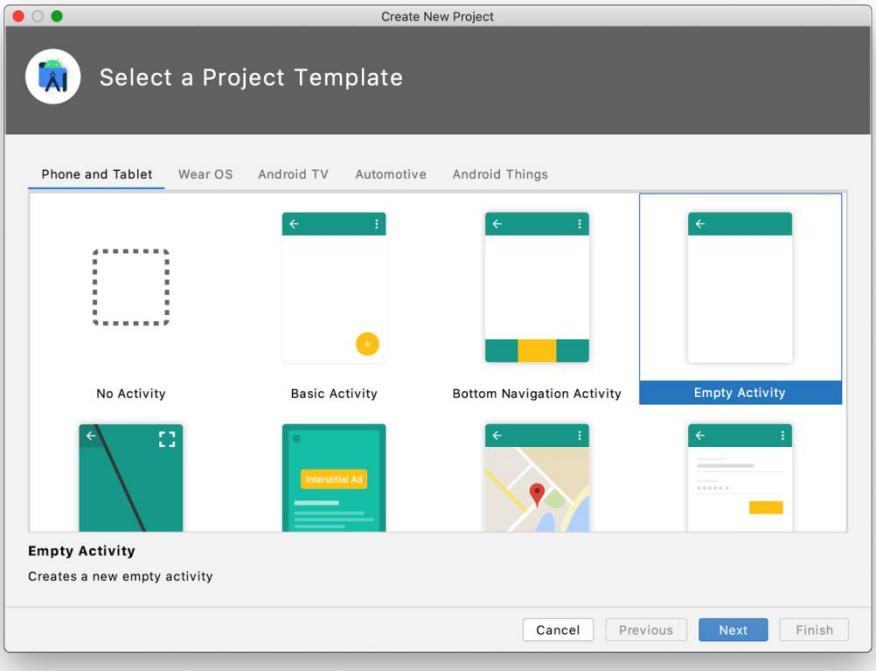
- Komputer z zainstalowanym Android Studio.

2. Utwórz nowy projekt

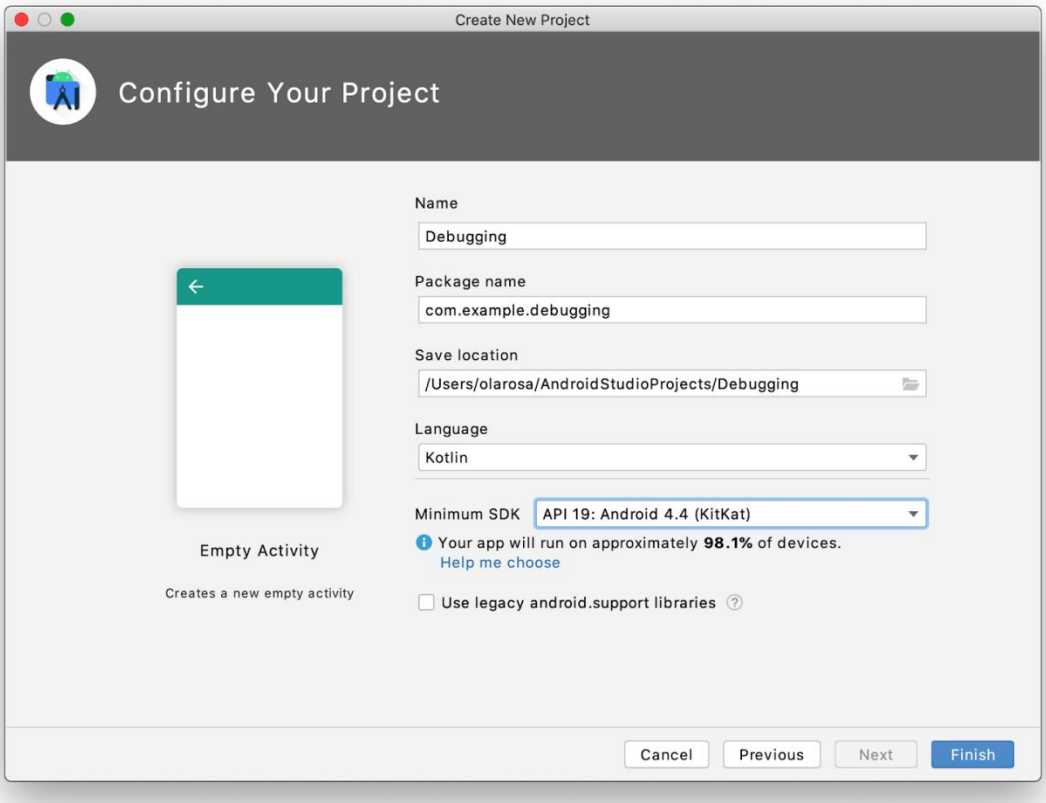
Zamiast debugować dużą i złożoną aplikację, zaczniemy od pustego projektu i wprowadzimy błędny kod, aby zademonstrować narzędzia do debugowania w Android Studio.

Zacznij od stworzenia nowego projektu Android Studio.

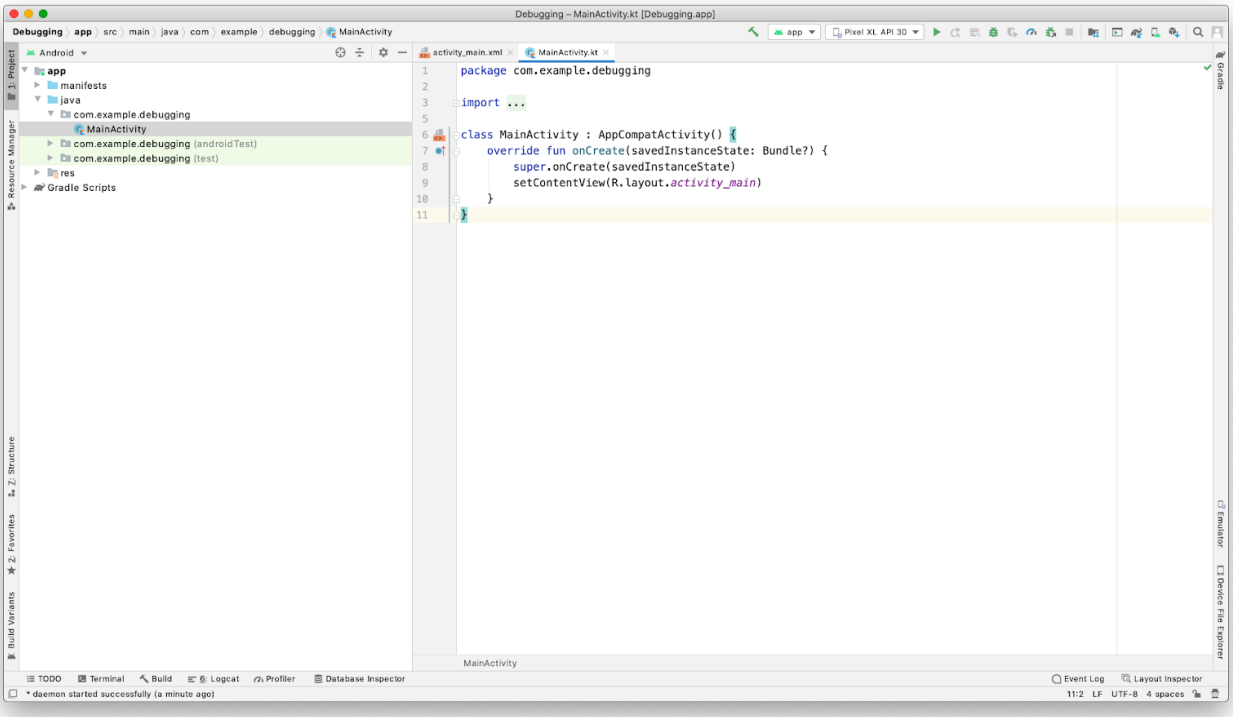
1. Na ekranie **Wybierz szablon projektu** wybierz **Puste działanie** .



2. Nazwij aplikację **Debugowanie** , upewnij się, że język jest ustawiony na Kotlin, a wszystko inne pozostaje niezmienione.



3. Zostaniesz przywitany nowym projektem Android Studio, pokazującym plik o nazwie MainActivity.kt.



Przedstaw błąd

Pamiętasz przykład dzielenia przez zero z lekcji debugowania w części 1? W ostatniej iteracji pętli, gdy aplikacja próbuje wykonać dzielenie przez zero, aplikacja ulega awarii z java.langArithmeticException, ponieważ dzielenie przez zero jest niemożliwe. Ten błąd został znaleziony i naprawiony przez zbadanie śladu stosu, a to założenie zostało zweryfikowane za pomocą instrukcji dziennika.

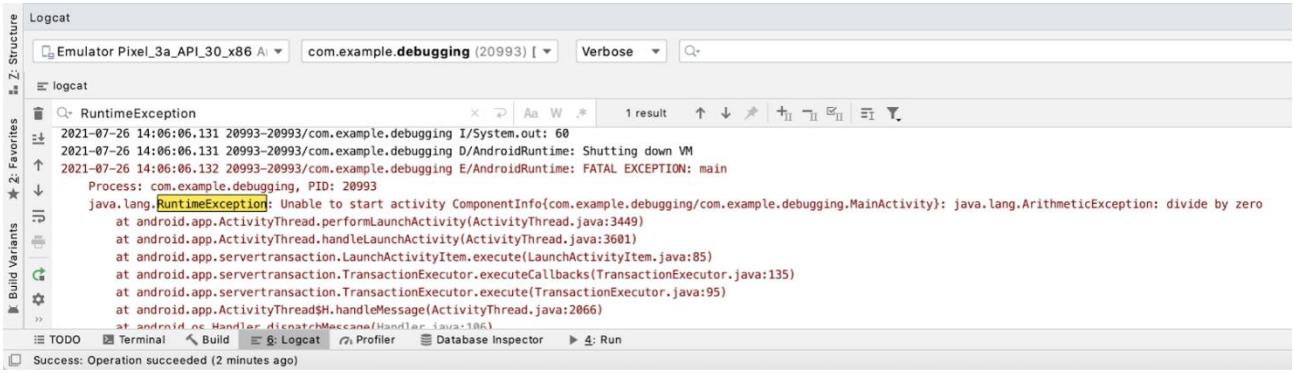
Jak już znasz ten przykład, zostanie on użyty do zademonstrowania, jak można użyć punktów przerwania. Punkty przerwania przechodzą przez kod po jednym wierszu naraz bez konieczności dodawania instrukcji dziennika i ponownego uruchamiania aplikacji.

1. Otwórz MainActivity.kti zastąp kod następującym:

```
package com.example.myapplication

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

public val TAG = "MainActivity"

class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
```

```
        division()
    }

    fun division() {
        val numerator = 60
        var denominator = 4
        repeat(5) {
            Log.v(TAG, "${numerator / denominator}")
            denominator--
        }
    }

}
```

2. Uruchom aplikację. B.Pamiętaj, że aplikacja ulega awarii zgodnie z oczekiwaniami.



3. Debuguj z punktami przerwania

Kiedy dowiedziałeś się o logowaniu, nauczyłeś się strategicznie umieszczać dzienniki, aby pomóc identyfikować błędy i sprawdzać, czy zostały one naprawione. Jednak w obliczu błędów, których nie wprowadziłeś, nie zawsze jest jasne, gdzie umieścić instrukcje dziennika lub które zmienne wydrukować. Często te informacje można znaleźć tylko w czasie wykonywania.

```
fun division() {
    val numerator = 60
    var denominator = 4
    repeat(5) {
        Log.v(TAG, "${numerator / denominator}")
        denominator--
    }
}
```

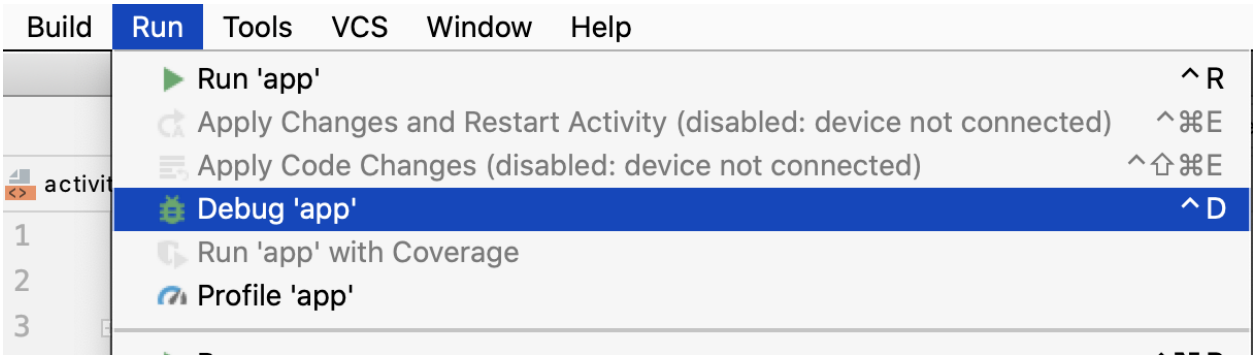
Tutaj pojawiają się punkty przerwania! Nawet jeśli masz mgliste pojęcie o tym, co powoduje błąd, na podstawie informacji ze śladu stosu, możesz dodać punkt przerwania służący jako znak stopu

dla określonej linii kodu. Po osiągnięciu punktu przerwania wstrzyma wykonywanie, umożliwiając korzystanie z innych narzędzi do debugowania w czasie wykonywania, aby z bliska przyjrzeć się temu, co się dzieje i co poszło nie tak.

Dołącz debuger

Za kulisami Android Studio używa narzędzia o nazwie **Android Debug Bridge** , znanego również w skrócie jako ADB. Jest to narzędzie wiersza polecenia, które jest zintegrowane z Android Studio i zapewnia możliwości debugowania, takie jak punkty przerwania, w uruchomionych aplikacjach. Narzędzie do debugowania jest często nazywane debuggerem.

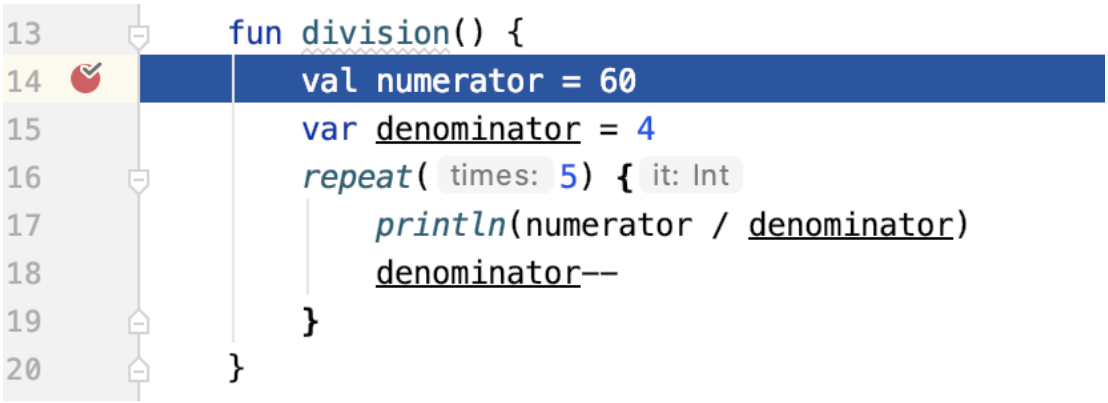
Aby użyć debugera lub dołączyć go do aplikacji, nie możesz po prostu uruchomić aplikacji za pomocą polecenia **Uruchom > Uruchom** , jak poprzednio. Zamiast tego uruchamiasz aplikację za pomocą polecenia **Uruchom > Debuguj „app”** .



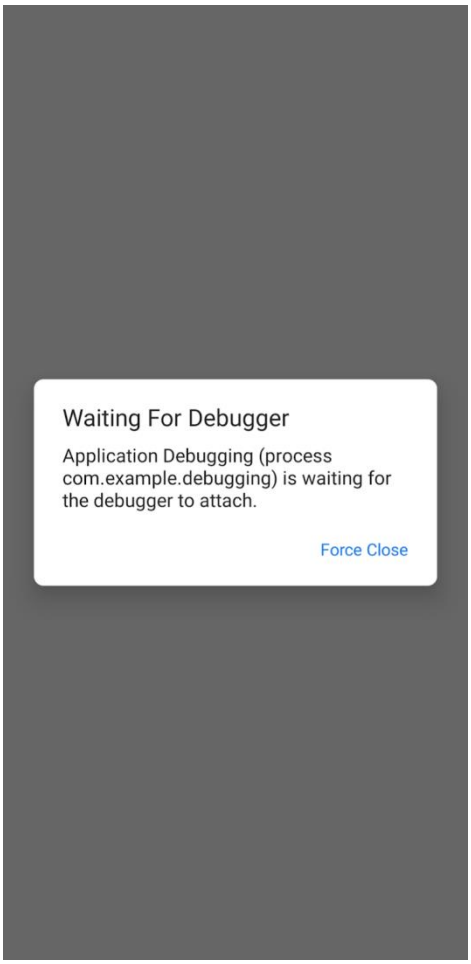
Dodaj punkty przerwania do swojego projektu

Wykonaj następujące kroki, aby zobaczyć punkty przerwania w akcji:

1. Dodaj punkt przerwania, klikając rynnę obok numeru wiersza, przy którym chcesz się zatrzymać. Obok numeru linii pojawi się kropka, a linia zostanie podświetlona.



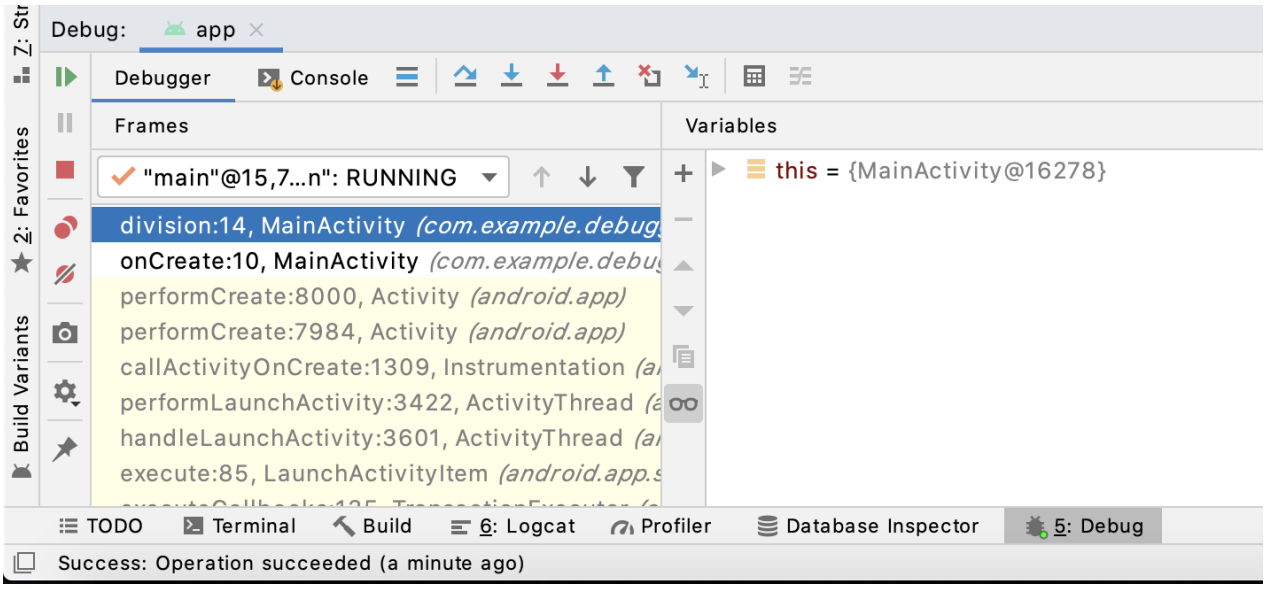
2. Uruchom aplikację z dołączonym debugerem, używając polecenia **Uruchom > Debuguj „aplikację”** lub ikony na pasku narzędzi. Po uruchomieniu aplikacji powinieneś zobaczyć taki ekran:



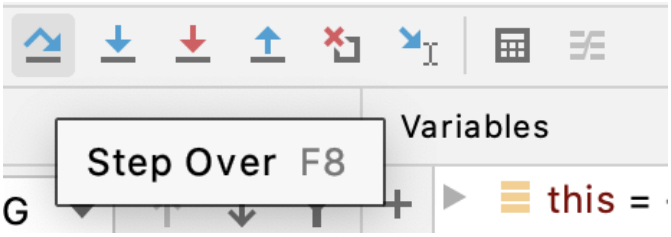
Po uruchomieniu aplikacji zobaczysz podświetlony punkt przerwania, gdy jest aktywowany.

```
13      fun division() {
14          val numerator = 60
15          var denominator = 4
16          repeat( times: 5) { it: Int
17              println(numerator / denominator)
18              denominator--
19          }
20      }
```

Na dole ekranu, w którym poprzednio oglądałeś okno Logcat, otworzyła się nowa zakładka **Debug .**

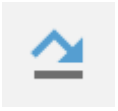


Po lewej stronie znajduje się lista funkcji, które są takie same jak lista, która pojawiła się w śladzie stosu. Po prawej stronie znajduje się okienko, w którym można sprawdzić wartości poszczególnych zmiennych w bieżącej funkcji (np `division()`.). Na górze znajdują się również przyciski, które pomagają poruszać się po programie, gdy jest wstrzymany. Przycisk, którego będziesz używać najczęściej, to **Step Over,** który wykonuje pojedynczą podświetloną linię kodu.

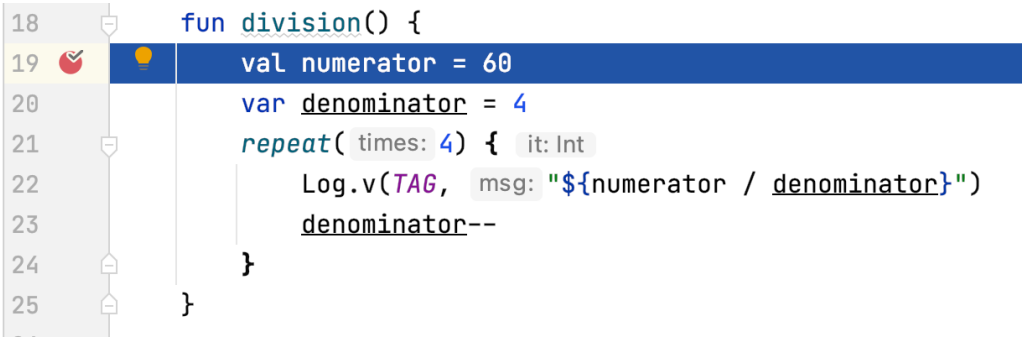


Wykonaj następujące kroki, aby debugować kod:

1. Po osiągnięciu punktu przerwania wiersz 19 (deklarujący `numerator`zmienną) jest teraz



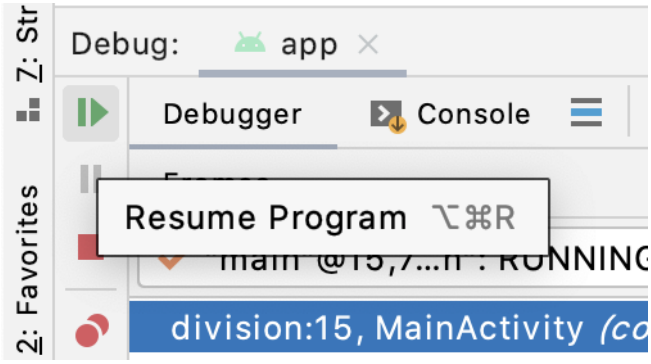
podświetlony, ale jeszcze nie został uruchomiony. Użyj przycisku **Step Over** , aby wykonać wiersz 19. Teraz wiersz 20 zostanie podświetlony.



2. Ustaw punkt przerwania w wierszu 22. To jest miejsce, w którym nastąpił podział i jest to wiersz, w którym ślad stosu zgłosił wyjątek.

```
18      fun division() {
19          val numerator = 60   numerator: 60
20          var denominator = 4
21          repeat( times: 4) {  it: Int
22              Log.v(TAG,  msg: "${numerator / denominator}")
23              denominator--
24          }
25      }
```

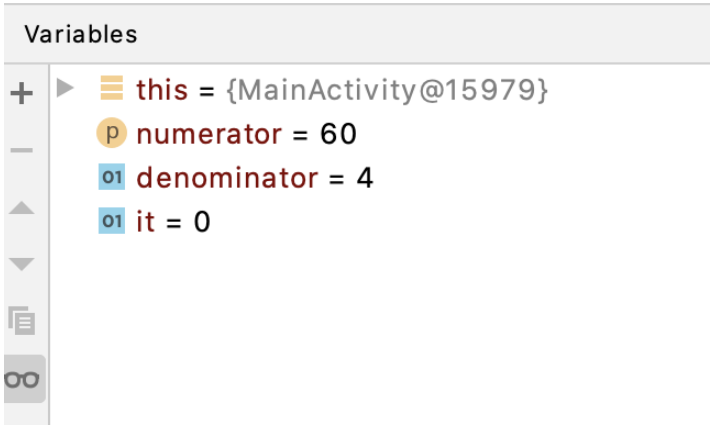
3. Użyj przycisku **Wznów program**  po lewej stronie okna **debugowania** , aby przejść do następnego punktu przerwania. Uruchom resztę `division()`funkcji.



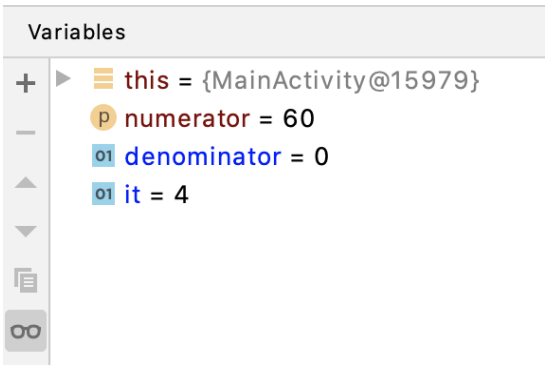
4. Zauważ, że wykonanie zatrzymuje się w linii 17 przed wykonaniem.

```
18      fun division() {
19          val numerator = 60   numerator: 60
20          var denominator = 4   denominator: 4
21          repeat( times: 4) {  it: Int
22              Log.v(TAG,  msg: "${numerator / deno
23              denominator--
24          }
25      }
```

5. Wartości każdej zmiennej `numerator`i `denominator`są wyświetlane obok ich deklaracji. Wartości zmiennych można zobaczyć w oknie debugowania w zakładce **Zmienne .**



6. Naciśnij przycisk **Wznów program** po lewej stronie okna debugowania jeszcze cztery razy. Za każdym razem pętla zatrzymuje się i obserwuje wartości `numerator`i `denominator`. W ostatniej iteracji `numerator`powinno być `60`i `denominator`powinno być `0`. I nie możesz podzielić 60 przez 0!



Teraz znasz dokładną linię kodu, która powoduje błąd, i znasz dokładną przyczynę. Tak jak poprzednio, możesz naprawić błąd, zmieniając liczbę powtórzeń kodu z `5`na `4`.

```
fun division() {
    val numerator = 60
    var denominator = 4
    repeat(4) {
        Log.v(TAG, "${numerator / denominator}")
        denominator--
    }
}
```

Wskazówka: aby usunąć punkt przerwania, kliknij kropkę, która pojawia się obok numeru wiersza.

4. Ustaw warunki dla punktów przerwania

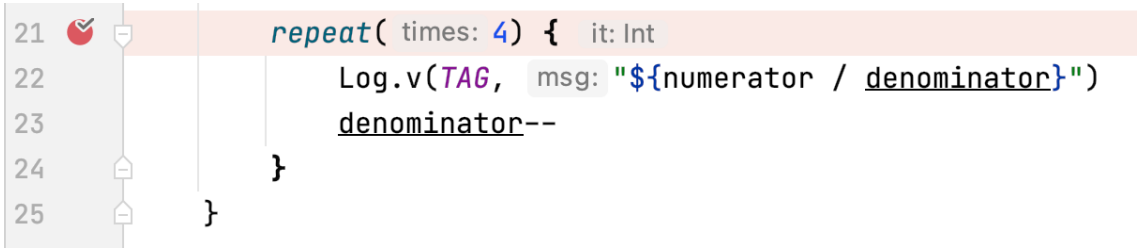
W poprzedniej sekcji trzeba było przejść przez każdą iterację pętli, aż mianownik wyniesie zero. W bardziej skomplikowanych aplikacjach może to być uciążliwe, gdy masz mniej informacji o błędzie. Jeśli jednak masz założenie, na przykład aplikacja ulega awarii tylko wtedy, gdy

mianownik wynosi zero, możesz zmodyfikować punkt przerwania, aby został osiągnięty dopiero po spełnieniu tego założenia, zamiast konieczności przechodzenia przez każdą iteracje pętli.

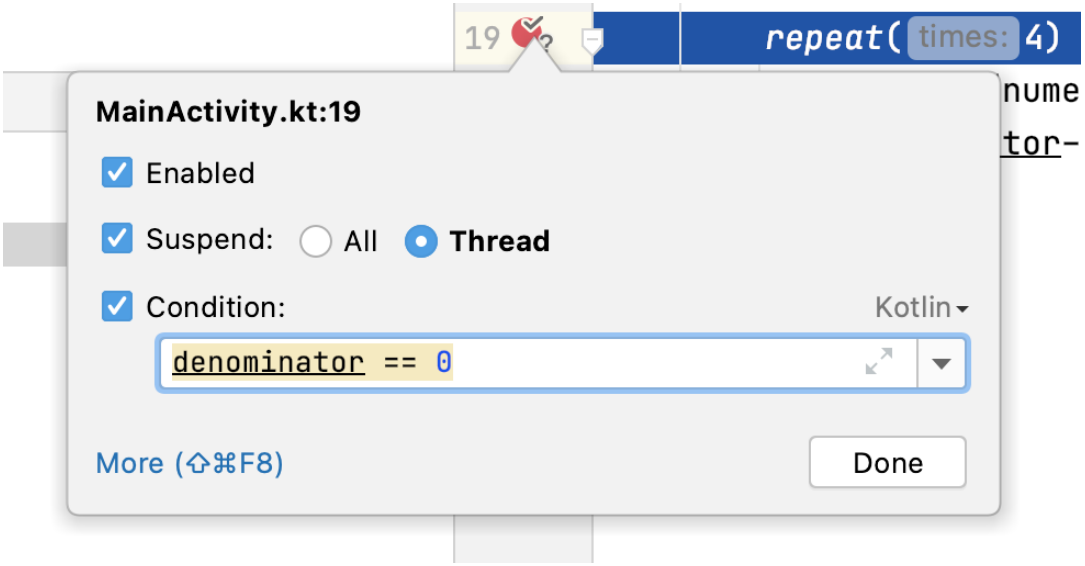
1. Jeśli to konieczne, ponownie wprowadź błąd, zmieniając `4` na `5` w pętli powtarzania.

```
repeat(4) {
    ...
}
```

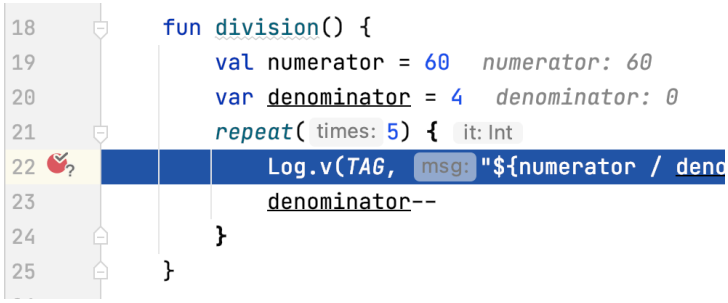
2. Umieść nowy punkt przerwania w wierszu z `repeat` instrukcją.



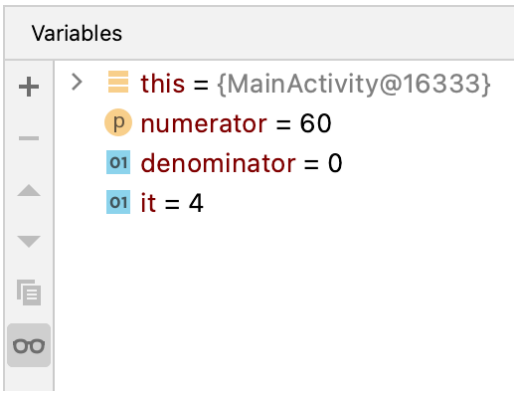
3. Kliknij prawym przyciskiem ikonę czerwonego punktu przerwania. Pojawi się menu z kilkoma opcjami, takimi jak czy punkt przerwania jest włączony. Wyłączony punkt przerwania nadal istnieje, ale nie zostanie wyzwolony w czasie wykonywania. Masz również możliwość dodania wyrażenia Kotlin, które jeśli zostanie ocenione jako prawda, zostanie wyzwolony punkt przerwania. Na przykład, jeśli użyjesz wyrażenia `denominator > 3`, punkt przerwania zostanie wyzwolony tylko podczas pierwszej iteracji pętli. Aby wyzwolić punkt przerwania tylko wtedy, gdy aplikacja potencjalnie ma zamiar podzielić przez zero, ustaw wyrażenie na `denominator == 0`. Opcje punktu przerwania powinny wyglądać następująco:



4. Uruchom aplikację za pomocą polecenia **Uruchom > Debuguj „aplikację”** i obserwuj, czy osiągnięto punkt przerwania.



Widać, że mianownik wynosi już 0. Punkt przerwania został wyzwolony tylko wtedy, gdy warunek został spełniony, co oszczędza czas i wysiłek, aby przejść przez kod.

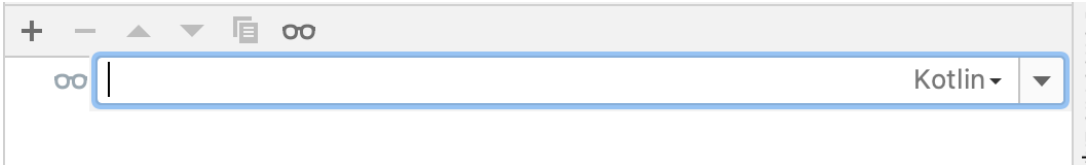


5. Jak poprzednio, widać, że błąd jest spowodowany przez pętlę wykonującą się o jeden raz za dużo, gdzie mianownik był ustawiony na 0.

Dodaj zegarki

Jeśli chcesz monitorować określoną wartość podczas debugowania, nie musisz przeszukiwać karty **Zmienne** , aby ją znaleźć. Możesz dodać coś o nazwie Zegarki, aby monitorować określone zmienne. Te zmienne będą widoczne w okienku debugowania. Gdy wykonywanie jest wstrzymane, a zmienna znajduje się w zakresie, będzie widoczna w okienku Zegarki. Dzięki temu debugowanie jest bardziej wydajne podczas pracy z większymi projektami. Będziesz mógł śledzić wszystkie istotne zmienne w jednym miejscu.

1. W widoku debugowania po prawej stronie panelu zmiennych powinien znajdować się kolejny pusty panel o nazwie **Watches** . Kliknij przycisk plusa  w lewym górnym rogu. Możesz zobaczyć opcję menu z napisem **New Watch** .



2. Wpisz nazwę zmiennej , `denominator` w odpowiednim polu i kliknij enter.
3. Uruchom ponownie swoją aplikację za pomocą polecenia **Uruchom > debuguj „app”** i zwróć uwagę, że po trafieniu punktu przerwania zobaczysz wartość mianownika w okienku zegarki.

5. Gratulacje

W podsumowaniu:

- Możesz ustawić punkty przerwania, aby wstrzymać wykonywanie aplikacji.
- Gdy wykonanie jest wstrzymane, możesz „przekroczyć”, aby wykonać tylko jeden wiersz kodu.
- Możesz ustawić instrukcje warunkowe, aby wyzwalały tylko punkty przerwania na podstawie wyrażenia Kotlin.
- Zegarki umożliwiają grupowanie interesujących zmiennych podczas debugowania.

Ucz się więcej

- Most debugowania Androida
- Debuguj swoją aplikację

Projekt: Aplikacja Płazy

1. Zanim zaczniesz

To laboratorium programowania przedstawia nową aplikację o nazwie Płazy, którą zbudujesz samodzielnie. To laboratorium kodowania przeprowadzi Cię przez kolejne kroki, aby ukończyć projekt aplikacji Amphibians, w tym konfigurację projektu i testowanie w Android Studio.

Warunki wstępne

- Ten projekt jest przeznaczony dla użytkowników, którzy ukończyli część 4 kursu Android Basics in Kotlin.

Co zbudujesz

- Zaimplementuj sieć w aplikacji za pomocą Retrofit i Moshi oraz odpowiednią obsługę błędów.

Co będziesz potrzebował

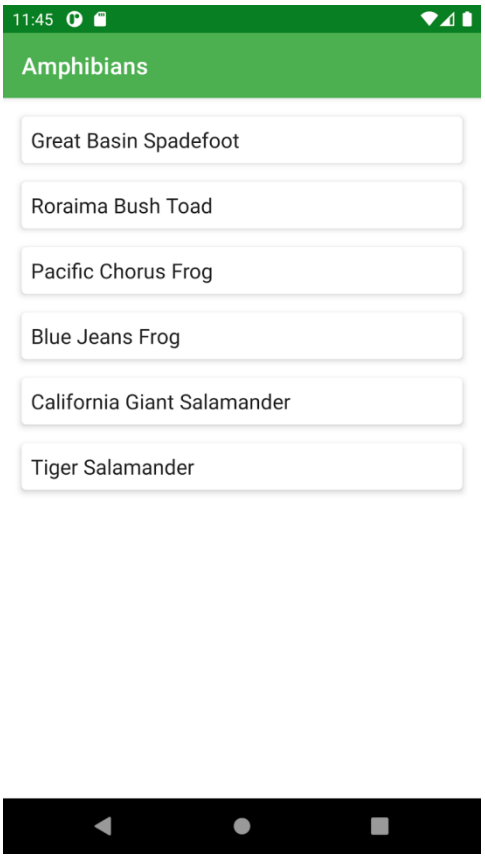
- Komputer z zainstalowanym Android Studio.

2. Zakończony przegląd aplikacji

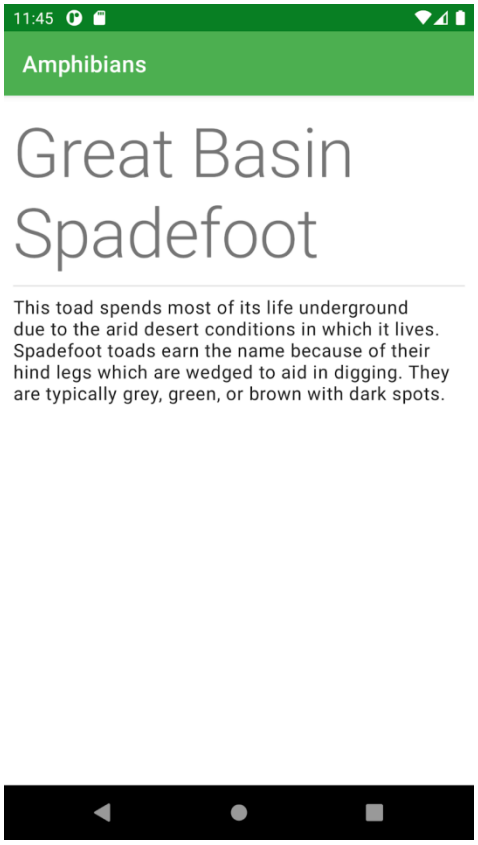
Witamy w Projekcie: Płazy!

Do tego momentu wszystkie utworzone aplikacje opierały się na danych przechowywanych lokalnie. Tym razem weźmiesz aplikację wyświetlającą informacje o różnych gatunkach płazów i wykorzystasz swoją wiedzę na temat sieci, parsowania JSON i przeglądania modeli, aby umożliwić aplikacji korzystanie z danych z sieci. Aplikacja pobierze swoje dane z niestandardowego interfejsu API dla tego projektu i wyświetli je w widoku listy.

W końcowej aplikacji pierwszy ekran, który zobaczy użytkownik, wyświetla nazwy każdego gatunku w widoku recyklera.



Dotknięcie elementu listy powoduje przejście do ekranu szczegółów, który wyświetla również nazwę gatunku oraz szczegółowy opis.



Chociaż część interfejsu użytkownika tej aplikacji jest już stworzona dla Ciebie, uruchomienie projektu startowego nie spowoduje wyświetlenia żadnych danych. Musisz zaimplementować część sieciową aplikacji, a następnie pokazać pobrane dane w układzie.

3. Rozpocznij

Pobierz kod projektu

Zauważ, że nazwa folderu to `android-basics-kotlin-amphibians-app`. Wybierz ten folder podczas otwierania projektu w Android Studio.

Adres URL kodu startowego:

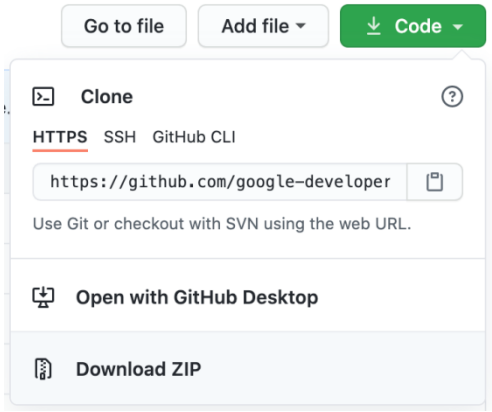
https://github.com/google-developer-training/android-basics-kotlin-amphibians-app/tree/main

Nazwa oddziału z kodem startowym: `main`

Aby uzyskać kod tego ćwiczenia z programowania i otworzyć go w Android Studio, wykonaj następujące czynności.

Pobierz kod

1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego pliki projektu.

Otwórz projekt w Android Studio

1. Uruchom Android Studio.
2. W oknie **Witamy w Android Studio** kliknij **Otwórz istniejący projekt Android Studio.**

Uwaga: jeśli Android Studio jest już otwarte, wybierz opcję menu **Plik** > **Nowy** > **Importuj projekt** .

3. W oknie dialogowym **Importuj projekt** przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.
6. Kliknij przycisk **Uruchom**  , aby skompilować i uruchomić aplikację. Upewnij się, że kompiluje się zgodnie z oczekiwaniami.
7. Przeglądaj pliki projektu w oknie narzędzia **Projekt** , aby zobaczyć, jak skonfigurowana jest aplikacja.

Implementuj usługę API

Podobnie jak w poprzednich projektach, większość aplikacji jest już dla Ciebie wdrożona. Musisz tylko zaimplementować część sieciową, korzystając z tego, czego nauczyłeś się w części 4. Zachęcamy do zapoznania się z kodem startowym. Większość pojęć powinna być już znana z jednostek od 1 do 3. Poniższe kroki przywołują określone części kodu, jeśli jest to konieczne do ukończenia każdego kroku.

Aplikacja wyświetla listę danych o płazach z sieci. Dane płazów pochodzą z obiektu JSON zwróconego przez API. Spójrz na `Amphibian.kt`plik w pakiecie **sieciowym .** Ta klasa modeluje pojedynczy obiekt ziemnowodny, którego lista zostanie zwrócona z interfejsu API. Każdy płaz ma trzy właściwości: nazwę, typ i opis.

```
data class Amphibian(
    val name: String,
    val type: String,
    val description: String
)
```

Backend dla tego API jest dość prosty. Istnieją dwie kluczowe informacje, których potrzebujesz, aby uzyskać dostęp do danych płazów: podstawowy adres URL i punkt końcowy, aby uzyskać listę płazów.

1. Podstawowy adres URL:`https://developer.android.com/courses/pathways/android-basics-kotlin-unit-4-pathway-2/`
2. Pobierz listę płazów:`android-basics-kotlin-unit-4-pathway-2-project-api.json`

Wskazówka: możesz wyświetlić surowy JSON w swojej przeglądarce.

Projekt ma już zależności Retrofit i Moshi. W pakiecie **sieciowym** znajdziesz `AmphibianApiService.kt`plik. Plik zawiera kilka `TODO`komentarzy. Wykonaj następujące pięć zadań, aby wdrożyć aplikację dla płazów:

1. Utwórz zmienną do przechowywania podstawowego adresu URL interfejsu API.
2. Zbuduj obiekt Moshi z fabryką adapterów Kotlin, której Retrofit będzie używał do parsowania JSON.
3. Zbuduj instancję aRetrofit za pomocą konwertera Moshi.
4. Zaimplementuj `AmphibianApiService`interfejs z `suspend`funkcją dla każdej metody API (w tej aplikacji jest tylko jedna metoda, aby POBIERZ listę płazów).
5. Utwórz `AmphibianApi`obiekt, aby odsłonić leniwą zainicjowaną usługę Retrofit, która korzysta z `AmphibianApiService`interfejsu.

Wskazówka: Aby uzyskać odświeżenie, zapoznaj się z treścią Łączenie z Internetem .

Zaimplementuj ViewModel

Po zaimplementowaniu interfejsu API wyślesz żądanie do interfejsu API płazów i zapiszesz wszystkie wartości, które muszą zostać wyświetlone. Zrobisz to w `AmphibianViewModel.kt`klasie znajdującej się w pakiecie **interfejsu użytkownika.**

Zauważysz, że nad deklaracją klasy znajduje się enum o nazwie `AmphibianApiStatus`.

```
enum class AmphibianApiStatus {LOADING, ERROR, DONE}
```

Trzy możliwe wartości, `LOADING`i `ERROR`, `DONE`są używane do pokazywania użytkownikowi statusu żądania.

W `AmphibianViewModel.kt`samej klasie będziesz musiał zaimplementować kilka `LiveData`zmiennych, funkcję do interakcji z API oraz funkcję do obsługi ustawienia płazów na ekranie szczegółów.

1. Dodaj `_status`prywatną zmienną MutableLiveData, która może zawierać `AmphibianApiStatus`wyliczenie i właściwość kopii zapasowej `status`dla stanu.
2. Dodaj `amphibians`zmienną i prywatną właściwość podkładu `_amphibians`dla listy płazów typu `List<Amphibian>`.
3. Dodaj zmienną _ `amphibian`typu `MutableLiveData<Amphibian>`i właściwość podkładu `amphibian`dla wybranego obiektu płazów typu `LiveData<Amphibian>`. Będzie on używany do przechowywania wybranego płazów pokazanego na ekranie szczegółów.
4. Zdefiniuj funkcję o nazwie `getAmphibianList()`. Uruchom współprogram za pomocą `ViewModelScope`, wewnątrz współprogramu wykonaj żądanie GET, aby pobrać dane płazów, wywołując `getAmphibians()`metodę usługi Retrofit. Będziesz musiał używać `try`i `catch`odpowiednio obsługiwać błędy. Przed wysłaniem żądania ustaw wartość `_status`na `AmphibianApiStatus.LOADING`. Pomyślne żądanie należy ustawić `_amphibians`na listę płazów z serwera i ustawić `_status`na `AmphibianApiStatus.DONE`. W przypadku błędu `_amphibians`należy ustawić na pustą listę i `_status`ustawić na `AmphibianApiStatus.ERROR`.
5. Zaimplementuj `onAmphibianClicked()`metodę, aby ustawić `_amphibian`zdefiniowaną właściwość w argumencie płazów przekazanym do funkcji. Ta metoda jest już wywoływana po wybraniu płazów, więc zostanie wyświetlona na ekranie szczegółowym.

Wskazówka: Podobnie jak w przypadku innych zdefiniowanych klas ViewModel, musisz odpowiednio używać `LiveData`i `MutableLiveData`typów.

Zaktualizuj interfejs użytkownika z ViewModel

Po zaimplementowaniu ViewModel wystarczy edytować klasy fragmentów i pliki układu, aby używać powiązań danych.

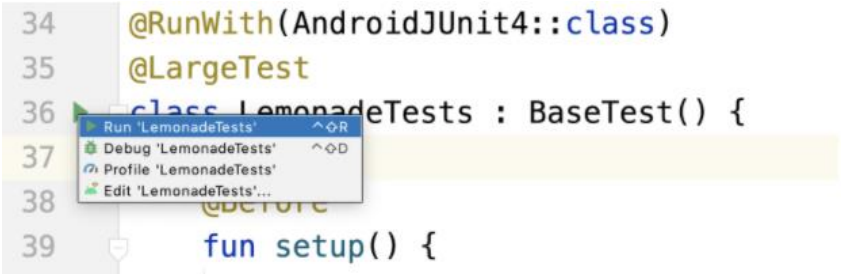
1. ViewModel jest już przywoływany w programie `AmphibianListFragment`. W `onCreateView()`metodzie, po rozdmuchaniu układu, po prostu wywołaj `getAmphibianList()`metodę z ViewModel.
2. W programie `fragment_amphibian_list.xml`znaczniki `<data>`zmiennych wiązania danych zostały już dodane do plików układu. Wystarczy zaimplementować TODO, aby interfejs użytkownika zaktualizował się na podstawie modelu widoku. Ustaw odpowiednie wiązania dla `listData`i `apiStatus`.
3. W `fragment_amphibian_detail.xml`programie zaimplementuj TODO, aby ustawić odpowiednie właściwości tekstowe dla nazwy i opisu płazów.

4. Instrukcje testowania

Przeprowadzanie testów

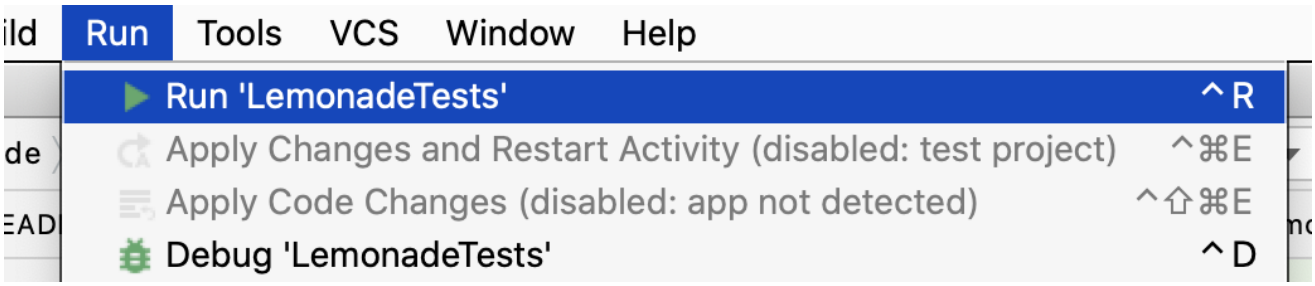
Aby uruchomić testy, możesz wykonać jedną z poniższych czynności.

W przypadku pojedynczego przypadku testowego otwórz klasę przypadku testowego i kliknij zieloną strzałkę po lewej stronie deklaracji klasy. Następnie możesz wybrać z menu opcję Uruchom. Spowoduje to uruchomienie wszystkich testów w przypadku testowym.

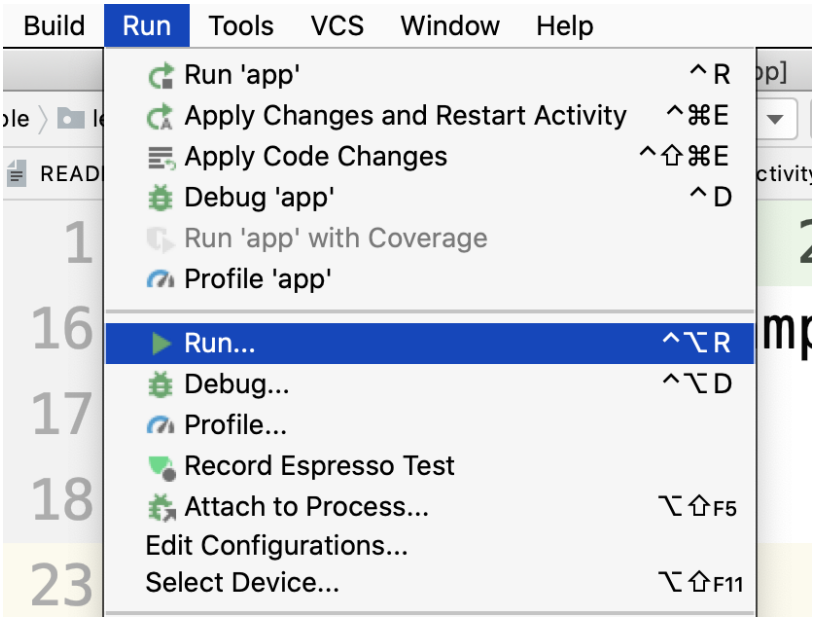


Często będziesz chciał uruchomić tylko jeden test, na przykład, jeśli tylko jeden test się nie powiedzie, a pozostałe testy zakończą się pomyślnie. Pojedynczy test można uruchomić tak samo, jak cały przypadek testowy. Użyj zielonej strzałki i wybierz opcję **Uruchom** .

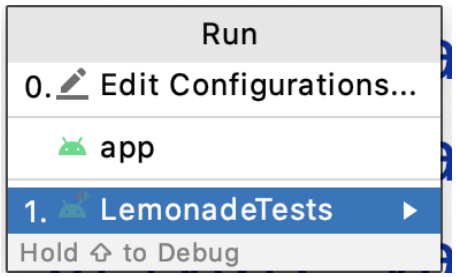
Jeśli masz wiele przypadków testowych, możesz również uruchomić cały zestaw testów. Podobnie jak w przypadku uruchamiania aplikacji, tę opcję znajdziesz w menu **Uruchom** .



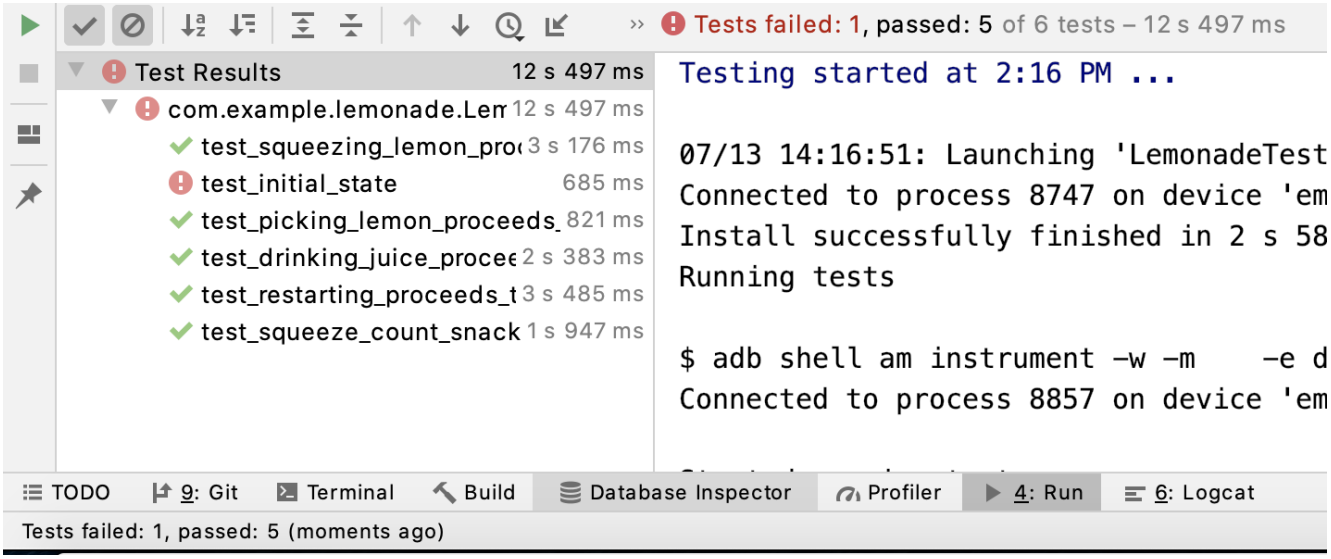
Zwróć uwagę, że Android Studio domyślnie dopasuje się do ostatniego uruchomionego celu (aplikacji, celów testowych itp.), więc jeśli w menu nadal pojawi się komunikat **Uruchom > Uruchom „aplikację”** , możesz uruchomić cel testu, wybierając **Uruchom > Uruchom** .



Następnie wybierz cel testowy z menu podręcznego.



Wyniki uruchomienia testów są pokazane w zakładce **Uruchom .** W okienku po lewej stronie zobaczysz listę testów zakończonych niepowodzeniem, jeśli takie istnieją. Testy zakończone niepowodzeniem są oznaczone czerwonym wykrzyknikiem obok nazwy funkcji. Zaliczone testy są oznaczone zielonym haczykiem.



Porada: Aby wyświetlić testy zaliczone, niezaliczone lub zarówno zaliczone, jak i niezaliczone w lewym okienku, możesz użyć dwóch przycisków w lewym górnym rogu. Zaznaczenie znacznika wyboru pokaże zaliczone testy, wybranie ikony okręgu z linią przez nią wyświetli listę nieudanych testów. Domyślnie wyświetlane są tylko testy zakończone niepowodzeniem.

Jeśli test się nie powiedzie, dane wyjściowe tekstowe zawierają informacje pomocne w rozwiązaniu problemu, który spowodował niepowodzenie testu.

Na przykład w powyższym komunikacie o błędzie test sprawdza, czy a `TextView` używa określonego zasobu ciągu. Jednak test kończy się niepowodzeniem. Tekst po słowach „Oczekiwany" i „Oczekiwany" nie jest zgodny, co oznacza, że wartość oczekiwana przez test nie jest zgodna z wartością z uruchomionej aplikacji. W tym przykładzie ciąg znaków użyty w `TextView` nie jest in-fact `squeeze_count`, zgodnie z oczekiwaniami testu.