Jednostka 5: Trwałość danych

Zapewnij swoim aplikacjom działanie podczas wszelkich zakłóceń w podstawowych sieciach lub procesach, aby zapewnić płynne i spójne środowisko użytkownika.

Spis treści

Wprowadzenie do SQL, Room i Flow

Poznaj podstawy odczytywania i manipulowania danymi za pomocą SQL oraz tworzenia i używania relacyjnych baz danych w aplikacji na Androida z biblioteką Room.

Podstawy SQL

1. Zanim zaczniesz

Wcześniej dowiedziałeś się, jak włączać dane sieciowe do swojej aplikacji, a także o używaniu współprogramów do obsługi współbieżnych zadań. Na tej ścieżce poznasz kolejną podstawową umiejętność programowania na Androida, która pozwoli Ci tworzyć wysokiej jakości aplikacje: *trwałość* . Nawet jeśli wcześniej nie słyszałeś tego terminu, prawdopodobnie spotkałeś się z uporem podczas korzystania z aplikacji. Od pisania listy zakupów, przez przewijanie zdjęć sprzed kilku lat w aplikacji do zdjęć, po wstrzymywanie i wznawianie gry, aplikacje wykorzystują trwałość, aby zapewnić bezproblemową obsługę. Chociaż użytkownikom łatwo jest przyjąć te funkcje za pewnik, utrwalanie danych jest podstawową umiejętnością programisty do tworzenia wysokiej jakości aplikacji.

W dalszej części tego rozdziału dowiesz się więcej o trwałości w systemie Android oraz o bibliotece o nazwie Room, która umożliwia aplikacjom odczytywanie i zapisywanie z bazy danych. Jednak zanim zagłębisz się w pracę z trwałością w systemie Android, ważne jest, aby zapoznać się z podstawami relacyjnych baz danych oraz jak czytać i manipulować danymi za pomocą czegoś, co nazywa się SQL (skrót od Structured Query Language). Jeśli znasz już te koncepcje, potraktuj tę lekcję jako powtórkę, aby upewnić się, że te koncepcje są świeże, gdy poznasz pokój. Jeśli nie, to w porządku! W tym momencie nie oczekujemy, że będziesz wiedział coś o bazach danych. Pod koniec tego ćwiczenia z kodowania będziesz mieć wszystkie podstawy potrzebne do rozpoczęcia nauki pracy z bazami danych w aplikacji na Androida.

Warunki wstępne

- Poruszaj się po projekcie w Android Studio.

Czego się nauczysz

- Struktura relacyjnej bazy danych: tabele, kolumny i wiersze
- `SELECT`oświadczenia zawierające `WHERE`, `ORDER BY`, `GROUP BY`i `LIMIT`klauzule
- Jak wstawiać, aktualizować i usuwać wiersze za pomocą SQL

Co będziesz potrzebował

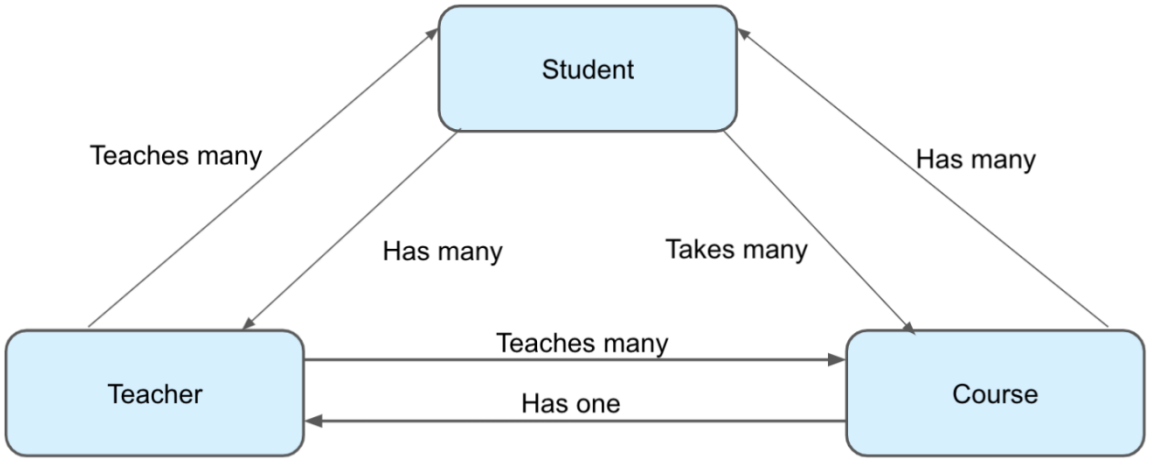
- Komputer z zainstalowanym Android Studio.

2. Przegląd relacyjnej bazy danych

Co to jest relacyjna baza danych?

W informatyce baza danych jest po prostu uporządkowanym zbiorem danych, do których można uzyskać elektroniczny dostęp i do których można zapisywać. Bazy danych mogą przechowywać te same informacje, które możesz reprezentować w aplikacji za pomocą Kotlina. Na urządzeniach mobilnych bazy danych są powszechnie używane do zapisywania danych z uruchomionej aplikacji, aby można było uzyskać do nich dostęp przy następnym otwarciu aplikacji, bez pobierania danych z innego źródła, na przykład z Internetu. Nazywa się to trwałością danych.

Mówiąc o trwałości danych, często słyszysz termin „relacyjna baza danych”. Relacyjna baza danych to popularny typ bazy danych, który organizuje dane w tabele, kolumny i wiersze. Pisząc kod Kotlin, tworzysz klasy, które reprezentują obiekty. Tabela w relacyjnej bazie *danych* działa w ten sam sposób. Oprócz reprezentowania danych, tabele mogą również odwoływać się do innych tabel, dzięki czemu można mieć między nimi relacje. Klasycznym przykładem może być tabela „studentów”, „nauczycieli” i „kursu”. Kurs miałby jednego nauczyciela, ale uczeń może mieć wiele kursów. Baza danych może reprezentować relacje między tymi tabelami, dlatego często słyszysz termin *relacyjna baza danych* .



Relacje w świecie rzeczywistym można przedstawić za pomocą relacji między tabelami.

Tabele, kolumny i wiersze

Zdefiniowanie tabel lub danych, które reprezentujesz, to tylko pierwszy krok w tworzeniu relacyjnej bazy danych. Musisz także pomyśleć o tym, jakie konkretnie informacje są przechowywane w każdej tabeli. Specyficzne właściwości są reprezentowane przez *kolumny* . Kolumna składa się z nazwy i typu danych. Znasz już właściwości z pracy z klasami w Kotlinie. Możesz myśleć o tabelach SQL w ten sam sposób. Tabela jest jak definicja klasy, opisująca typ „rzeczy”, którą chcesz reprezentować. Kolumna jest specyficzną właściwością „rzeczy” tworzonej z każdym wpisem w tabeli.

Zakład	
ID	LICZBA CAŁKOWITA
gatunek	Ciąg tekstowy)
Nazwa	Ciąg tekstowy)
kolor	Ciąg tekstowy)
Ogród	
ID	LICZBA CAŁKOWITA
Nazwa	Ciąg tekstowy)
długość	LICZBA CAŁKOWITA
szerokość	LICZBA CAŁKOWITA

Poszczególne wpisy w tabeli nazywane są *wierszami* . To jest jak instancja klasy w Kotlinie. Każdy wiersz zawiera dane odpowiadające każdej kolumnie. Tabela zawiera szablon, ale wiersze definiują rzeczywiste dane przechowywane w tabeli.

ID	gatunek	Nazwa	kolor
1	Camellia Sinensis	Herbata	Zielony
2	Echinacea Purpurea	Jeżówka purpurowa	fioletowy
3	Ferula Foetida	Asafetyda	Zielony

Główny klucz

W powyższym przykładzie zwróć uwagę na kolumnę dla właściwości id. Chociaż istnieje prawdopodobieństwo, że gatunki roślin występujące w naturze lub cokolwiek reprezentujesz w swojej bazie danych, prawdopodobnie nie mają wygodnie ponumerowanego identyfikatora, ważne jest, aby wiersze w tabeli danych miały jakiś unikalny identyfikator. Jest to powszechnie znane jako *klucz podstawowy* i jest unikalny dla każdego wiersza w tabeli. Jest to przydatne, jeśli musisz odwoływać się do wierszy w jednej tabeli danych z innej tabeli. Załóżmy na przykład, że istniała inna tabela o nazwie „ogród”, w której chcesz powiązać ogród ze wszystkimi zawartymi w nim gatunkami roślin. Możesz użyć klucza podstawowego w tabeli roślin, aby odwołać się do rośliny z wpisu w tabeli ogrodów lub dowolnej innej tabeli w bazie danych.

Klucze podstawowe umożliwiają utrzymywanie relacji w relacyjnej bazie danych. Chociaż w tym kursie nie będziesz używać baz danych z więcej niż jedną tabelą, posiadanie unikalnego identyfikatora pomaga w zapytaniach, aktualizowaniu i usuwaniu istniejących elementów w tabeli.

Typy danych

Podobnie jak w przypadku definiowania właściwości klas Kotlin, kolumny w bazie danych mogą być jednym z wielu możliwych typów danych. Kolumna może reprezentować znak, ciąg, liczbę (z liczbą dziesiętną lub bez) lub dane binarne. Inne dane, takie jak daty i godziny, mogą być reprezentowane numerycznie lub jako ciąg, w zależności od przypadku użycia. Podczas pracy z Roomem będziesz głównie pracował z typami Kotlin, ale za kulisami są one mapowane na typy SQL.

Uwaga: Podobnie jak w Kotlinie, kolumny w bazie danych mogą mieć różne typy danych. To nie to samo, co typy danych w Kotlinie. Jeśli chcesz dowiedzieć się więcej, zapoznaj się z tym zasobem , aby zapoznać się z omówieniem podstawowych typów danych w SQL. Jednak podczas pracy z bazą danych w Room nie będziesz pracować bezpośrednio z tymi typami danych, ale raczej definiujesz swoje tabele danych w kodzie przy użyciu równoważnych typów danych Kotlin.

SQL

Podczas uzyskiwania dostępu do relacyjnej bazy danych, niezależnie od tego, czy jest to samodzielne, czy przy użyciu biblioteki, takiej jak Room, będziesz potrzebować czegoś, co nazywa się *SQL.*

Co to jest SQL? SQL (czasami wymawiane jako „sequel") oznacza *Structured Query Language* i umożliwia odczytywanie i manipulowanie danymi w relacyjnej bazie danych. Nie martw się jednak — nie będziesz musiał uczyć się zupełnie nowego języka programowania tylko po to, aby zaimplementować trwałość w swojej aplikacji. W przeciwieństwie do języka programowania, takiego jak Kotlin, SQL składa się tylko z kilku typów instrukcji do odczytu i zapisu z bazy danych. Gdy nauczysz się podstawowego formatu każdego z nich, wystarczy wypełnić puste pola na konkretne informacje, które czytasz lub piszesz z bazy danych.

Poniżej znajdują się najpopularniejsze instrukcje SQL i te, z którymi będziesz pracować.

WYBIERZ	Pobiera określone informacje z tabeli danych, a wyniki można filtrować i sortować na różne sposoby.
WSTAWIĆ	Dodaje nowy wiersz do tabeli.
AKTUALIZACJA	Aktualizuje istniejący wiersz (lub wiersze) w tabeli.
KASOWAĆ	Usuwa istniejący wiersz (lub wiersze) z tabeli.

Teraz, zanim będziesz mógł zrobić cokolwiek w SQL, będziesz potrzebować bazy danych. Na następnym ekranie skonfigurujesz przykładowy projekt, który zawiera bazę danych do ćwiczenia zapytań SQL.

3. Kod startowy - Baza parków

Kod startowy, który pobierzesz, jest nieco inny niż w przypadku poprzednich ćwiczeń z programowania. Zamiast budować na istniejącym projekcie, udostępnimy prosty projekt Android Studio, który utworzy bazę danych, której możesz użyć do ćwiczenia zapytań SQL. Po jednokrotnym uruchomieniu aplikacji będziesz mógł uzyskać dostęp do bazy danych za pomocą narzędzia Android Studio o nazwie Database Inspector.

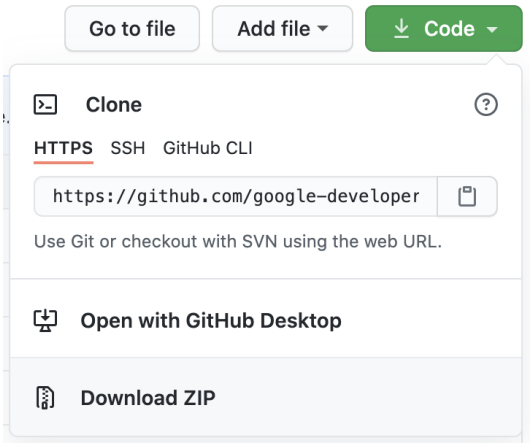
URL kodu startowego: https://github.com/google-developer-training/android-basics-kotlin-sql-basics-app

Oddział:main

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

Pobierz kod

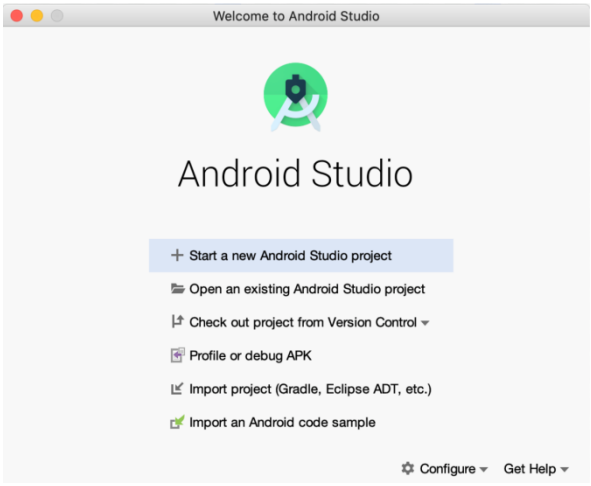
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



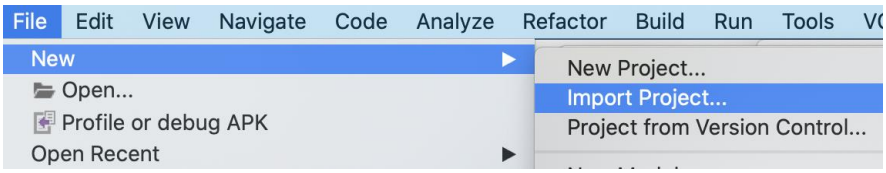
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

Otwórz projekt w Android Studio

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



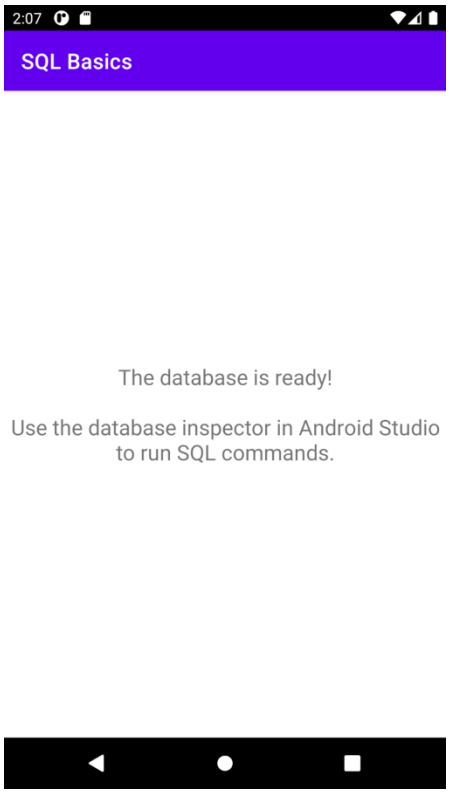
Uwaga: jeśli Android Studio jest już otwarte, wybierz opcję menu **Plik** > **Nowy** > **Importuj projekt** .



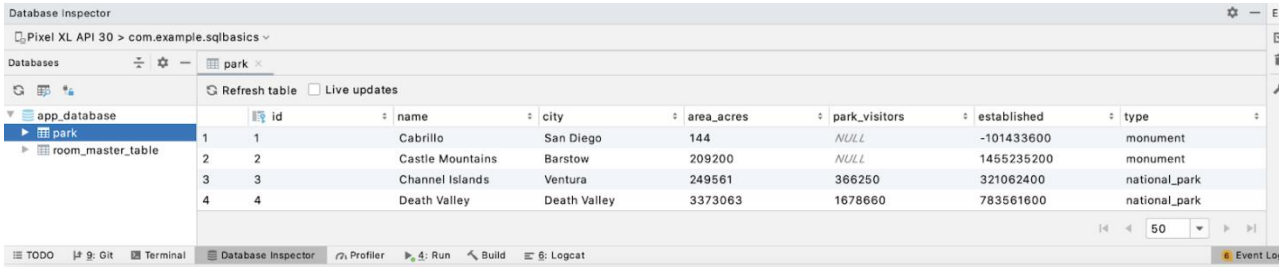
3. W oknie dialogowym **Importuj projekt** przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.
6. Kliknij przycisk **Uruchom** , aby skompilować i uruchomić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

Przed przejściem do następnej sekcji wykonaj następujące kroki, aby upewnić się, że projekt początkowy został skonfigurowany.

1. Uruchom aplikację. Aplikacja powinna pokazywać pojedynczy ekran, który wygląda tak.



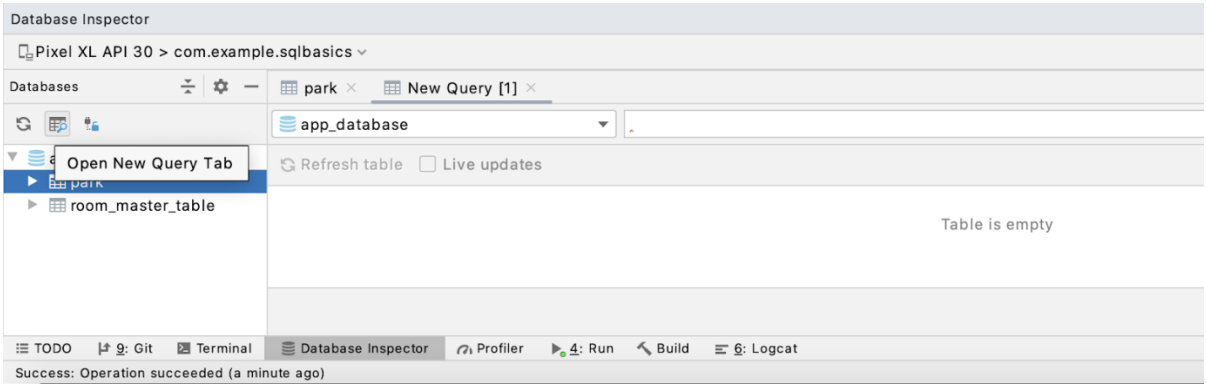
3. W Android Studio otwórz Inspektora baz danych za pomocą opcji **Widok** > **Okna narzędzi** > **Inspektor bazy danych** .
4. Na dole powinna pojawić się nowa karta oznaczona jako „Inspektor bazy danych”. Załadowanie może potrwać kilka sekund, ale po lewej stronie powinna pojawić się lista z tabelami danych, które można wybrać do uruchamiania zapytań.



Uwaga : jeśli używasz Android Studio 2020.3.1 Arctic Fox, dostęp do Inspektora bazy danych można uzyskać, wybierając Widok > Narzędzie > Kontrola aplikacji. Zakładka na dole nadal będzie nazywać się Inspektorem Bazy Danych.

4. Podstawowe instrukcje SELECT

W poniższych ćwiczeniach będziesz uruchamiać zapytania w Inspektorze bazy danych. Upewnij się, że wybrałeś właściwą tabelę w lewym okienku (park), kliknij przycisk **Otwórz nową kartę zapytania** i powinieneś zobaczyć pole tekstowe, w którym możesz wpisywać polecenia SQL.



Instrukcja SQL to polecenie, coś w rodzaju wiersza kodu, które uzyskuje dostęp (odczyt lub zapis) do bazy danych. Najbardziej podstawową rzeczą, jaką możesz zrobić w SQL, jest po prostu zebranie wszystkich danych w tabeli. Aby to zrobić, zaczynasz od słowa `SELECT`, co oznacza, że chcesz czytać dane. Następnie dodajesz gwiazdkę (`*`). W tym miejscu możesz określić kolumny, które chcesz wybrać, a użycie gwiazdki jest skrótem do zaznaczania wszystkich kolumn. Następnie użyj `FROM`słowa kluczowego, po którym następuje nazwa tabeli danych, `park`. Uruchom następujące polecenie w Inspektorze bazy danych i obserwuj całą tabelę ze wszystkimi wierszami i kolumnami.

```
SELECT * FROM park
```

Jeśli chcesz wybrać tylko określoną kolumnę zamiast wszystkich kolumn w tabeli danych, możesz określić nazwę kolumny.

```
SELECT city FROM park
```

Możesz także wybrać wiele konkretnych kolumn, oddzielonych przecinkami.

```
SELECT name, established, city FROM park
```

Czasami wybranie wszystkich wierszy w bazie danych nie jest całkowicie konieczne. Możesz dodać klauzule — część instrukcji SQL — w celu dalszego zawężenia wyników.

Jedna klauzula to `LIMIT`, która pozwala ustawić limit liczby zwracanych wierszy. Więc zamiast zwracać wszystkie 23 wyniki, poniższe zapytanie zwraca tylko pierwsze pięć.

```
SELECT name FROM park
LIMIT 5
```

Jedną z najczęstszych i najbardziej użytecznych klauzul jest `WHERE`klauzula. Klauzula `WHERE`umożliwia filtrowanie wyników na podstawie co najmniej jednej kolumny.

```
SELECT name FROM park
WHERE type = "national_park"
```

Uwaga: W przeciwieństwie do Kotlina, gdzie operator = jest używany do przypisania, a == jest używany do porównania, w SQL do porównania dwóch wartości używa się tylko jednego znaku równości.

`!=`Istnieje również operator „nie równa się” ( ). Następujące zapytanie zawiera listę wszystkich parków o powierzchni ponad 100 000 akrów, które nie są `recreation_area`. Z `WHERE`klauzulami można również używać operatorów logicznych, takich jak `AND`lub `OR`, aby dodać więcej niż jeden warunek.

```
SELECT name FROM park
WHERE type != "recreation_area"
AND area_acres > 100000
```

Ćwiczyć

Zapytania SQL mogą być przydatne do odpowiadania na różne pytania dotyczące danych, a najlepszym sposobem na ćwiczenie jest pisanie własnych zapytań. W kolejnych kilku krokach będziesz mieć możliwość napisania zapytania, aby odpowiedzieć na określone pytanie. Pamiętaj, aby przetestować go w Inspektorze baz danych przed przejściem dalej.

Wszystkie ćwiczenia będą opierać się na skumulowanej wiedzy ze wszystkich poprzednich sekcji, a na końcu ćwiczenia z programowania będą dostępne instrukcje sprawdzające Twoje odpowiedzi.

Problem 1:

Napisz zapytanie SQL, aby uzyskać nazwy wszystkich parków z mniej niż 1 000 000 odwiedzających.

5. Wspólne funkcje SQL

Pierwsze zapytanie, które napisałeś, zwróciło po prostu każdy wiersz w bazie danych.

```
SELECT * FROM park
```

Być może jednak nie chcesz zwracać długiej listy wyników. SQL oferuje również funkcje agregujące, które mogą pomóc zredukować dane do pojedynczej znaczącej wartości. Załóżmy na przykład, że chcesz poznać liczbę wierszy w `park`tabeli. Zamiast `SELECT * ...`, użyj `COUNT()`funkcji i przekaż `*`(dla wszystkich wierszy) lub nazwę kolumny, a zapytanie zwróci liczbę wszystkich wierszy.

```
SELECT COUNT(*) FROM park
```

Inną przydatną funkcją agregującą jest `SUM()`funkcja do sumowania wartości w kolumnie. To zapytanie filtruje tylko parki narodowe (ponieważ są to jedyne wpisy z `park_visitors`kolumną, która nie jest pusta) i sumuje całkowitą liczbę odwiedzających dla każdego parku.

```
SELECT SUM(park_visitors) FROM park
WHERE type = "national_park"
```

Warto zauważyć, że nadal możesz używać `SUM()`wartości null, ale wartość będzie po prostu traktowana jako zero. Następujące zapytanie zwróci to samo, co powyższe. Jednak nadal dobrze jest być jak najbardziej konkretnym, aby uniknąć błędów, gdy zaczniesz używać SQL w swoich aplikacjach.

```
SELECT SUM(park_visitors) FROM park
```

Oprócz agregowania wartości istnieją inne przydatne funkcje, takie jak `MAX()`i `MIN()`pobieranie odpowiednio największej lub najmniejszej wartości.

```
SELECT MAX(area_acres) FROM park
WHERE type = 'national_park'
```

Uzyskiwanie WYRÓŻNIANYCH wartości

Możesz zauważyć, że w przypadku niektórych wierszy kolumna ma taką samą wartość jak inne wiersze. Na przykład kolumna typu ma tylko skończoną liczbę możliwych wartości. Zduplikowane wartości można wyeliminować z wyników zapytania za pomocą `DISTINCT`słowa kluczowego. Na przykład, aby uzyskać wszystkie unikatowe wartości dla kolumny typu, możesz użyć następującego zapytania.

```
SELECT DISTINCT type FROM park
```

Możesz również użyć `DISTINCT`w funkcji agregującej, więc zamiast wymieniać unikalne `type`adresy i liczyć je samodzielnie, możesz po prostu zwrócić liczbę.

```
SELECT COUNT(DISTINCT type) FROM park
```

Ćwiczyć

Poświęć trochę czasu na zastosowanie tego, czego się nauczyłeś i sprawdź, czy możesz napisać następujące pytania. Pamiętaj, aby użyć Inspektora bazy danych, aby sprawdzić, czy kod działa.

Problem 2:

Napisz zapytanie SQL, aby uzyskać liczbę różnych miast w `park`tabeli

Problem 3:

Napisz zapytanie SQL, aby uzyskać całkowitą liczbę odwiedzających parki w San Francisco.

6. Porządkowanie i grupowanie wyników zapytania

W poprzednich przykładach znalezienie konkretnego wpisu mogło być trudne. Na szczęście możesz także sortować wyniki `SELECT`instrukcji za pomocą `ORDER BY`klauzuli. Dodaj `ORDER BY`klauzulę na końcu zapytania po `WHERE`klauzuli (jeśli istnieje) i po prostu określ nazwę kolumny, według której chcesz sortować. Poniższy przykład pobiera nazwę każdego parku w bazie danych, ale sortuje wyniki w kolejności alfabetycznej.

```
SELECT name FROM park
ORDER BY name
```

Domyślnie wyniki są sortowane w kolejności rosnącej, ale możesz dodać słowo kluczowe `ASC`lub `DESC`do klauzuli order by, aby sortować w kolejności rosnącej lub malejącej. Na początku nie musisz określać `ASC`jako pierwszej listy wyników zapytania w kolejności rosnącej, ale jeśli chcesz uzyskać wyniki w kolejności malejącej, dodaj `DESC`słowo kluczowe na końcu `ORDER BY`klauzuli.

```
SELECT name FROM park
ORDER BY name DESC
```

Aby wyniki były bardziej czytelne, masz również możliwość grupowania ich według kolumn. Przed `ORDER BY`klauzulą (jeśli istnieje) możesz opcjonalnie określić `GROUP BY`klauzulę i kolumnę. Powoduje to podzielenie wyników na podzbiór właściwy dla kolumny w `GROUP BY`, a dla każdej kolumny wyniki zostaną przefiltrowane i uporządkowane zgodnie z resztą zapytania.

```
SELECT type, name FROM park
GROUP BY type
ORDER BY name
```

Najlepiej to zrozumieć na przykładzie. Zamiast zliczać wszystkie parki w bazie danych, możesz zobaczyć, ile jest obecnych parków każdego typu i uzyskać osobną liczbę dla każdego z nich.

```
SELECT type, COUNT(*) FROM park
GROUP BY type
ORDER BY type
```

Ćwiczyć

Poświęć trochę czasu na zastosowanie tego, czego się nauczyłeś i sprawdź, czy możesz napisać następujące zapytanie. Pamiętaj, aby użyć Inspektora bazy danych, aby sprawdzić, czy kod działa.

Problem 4 : Napisz zapytanie SQL do 5 najpopularniejszych nazw parków wraz z ich liczbą odwiedzających, którzy mieli najwięcej odwiedzających, w porządku malejącym.

7. Wstawianie i usuwanie wierszy

Musisz mieć możliwość zapisywania danych, aby w pełni korzystać z utrwalania danych w Androidzie z Room. Oprócz wykonywania zapytań do bazy danych dostępne są również instrukcje SQL do wstawiania, aktualizowania i usuwania wierszy. Będziesz potrzebować podstawowej wiedzy na ten temat, gdy nauczysz się zapisywać dane w pokoju później w ścieżce 2.

WSTAW oświadczenie

Aby dodać nowy wiersz, użyj `INSERT`instrukcji. Po `INSERT`oświadczeniu następuje `INTO`słowo kluczowe i nazwa tabeli, w której chcesz dodać wiersz. Po słowie kluczowym WARTOŚCI podajesz wartość dla każdej kolumny (w kolejności) w nawiasach, oddzielając każdą z nich przecinkami. Format `INSERT`oświadczenia jest następujący.

```
INSERT INTO table_name
VALUES (column1, column2, ...)
```

Aby dodać wiersz do `park`tabeli, `INSERT`instrukcja wyglądałaby mniej więcej tak. Wartości odpowiadają kolejności, w jakiej kolumny są zdefiniowane dla `park`tabeli. Zauważ, że niektóre dane nie są określone. Na razie jest to w porządku, ponieważ zawsze możesz zaktualizować wiersz po jego wstawieniu.

```
INSERT INTO park
VALUES (null, 'Googleplex', 'Mountain View', 12, null, 0, '')
```

Zauważ też, że przekazujesz `null`dowód tożsamości. Chociaż możesz podać konkretny numer, nie jest to zbyt wygodne, ponieważ Twoja aplikacja musiałaby śledzić najnowszy identyfikator, aby upewnić się, że nie ma duplikatów. Możesz jednak skonfigurować bazę danych tak, aby klucz podstawowy był automatycznie zwiększany, co zostało zrobione tutaj. W ten sposób możesz przekazać `null`, a następny identyfikator zostanie wybrany automatycznie.

Sprawdź, czy wpis został utworzony, używając `WHERE`klauzuli określającej park o nazwie `"Googleplex"`.

```
SELECT * FROM park
WHERE name = 'Googleplex'
```

Oświadczenie UPDATE

Po utworzeniu wiersza możesz w dowolnym momencie zmienić jego zawartość. Możesz to zrobić za pomocą `UPDATE`instrukcji. Podobnie jak w przypadku wszystkich innych instrukcji SQL, które widziałeś, najpierw musisz określić nazwę tabeli. W `SET`klauzuli po prostu ustaw każdą kolumnę, którą chcesz zmienić, na nową wartość.

```
UPDATE table_name
SET column1 = ...,
column2 = ...,
...
```

```
WHERE column_name = ...
...
```

W przypadku wpisu Googleplex jedna istniejąca właściwość jest aktualizowana, a niektóre inne pola są wypełniane (wcześniej te pola miały wartość, ale był to ciąg pusty, `""`). Możesz zaktualizować wiele (lub wszystkie) pól jednocześnie za pomocą `UPDATE`instrukcji.

```
UPDATE park
SET area_acres = 46,
established = 1088640000,
type = 'office'
WHERE name = 'Googleplex'
```

Zobacz aktualizacje odzwierciedlone w wynikach zapytania

```
SELECT * FROM park
WHERE name = 'Googleplex'
```

DELETE oświadczenie

Na koniec możesz również użyć polecenia SQL, aby usunąć wiersze z bazy danych. Ponownie określ nazwę tabeli i tak jak w przypadku `SELECT`instrukcji, użyj `WHERE`klauzuli, aby podać kryteria dla wierszy, które chcesz usunąć. Ponieważ `WHERE`klauzula może pasować do wielu wierszy, możesz usunąć wiele wierszy za pomocą jednego polecenia.

```
DELETE FROM table_name
WHERE <column_name> = ...
```

Ponieważ Googleplex nie jest parkiem narodowym, spróbuj użyć `DELETE`instrukcji, aby usunąć ten wpis z bazy danych.

```
DELETE FROM park
WHERE name = 'Googleplex'
```

Sprawdź, czy wiersz został usunięty za pomocą `SELECT`instrukcji. Zapytanie nie powinno zwracać żadnych wyników, co oznacza, że wszystkie wiersze o nazwie „Googleplex” zostały pomyślnie usunięte.

```
SELECT * FROM park
WHERE name = 'Googleplex'
```

To wszystko, jeśli chodzi o wstawianie, aktualizowanie i usuwanie danych. Wszystko, co musisz wiedzieć, to format polecenia SQL, które chcesz wykonać, i określić wartości zgodne z kolumnami w bazie danych. Kiedy wprowadzimy Room w następnym laboratorium programowania, skupisz się przede wszystkim na czytaniu z bazy danych. Wstawianie, aktualizowanie i usuwanie danych zostanie szczegółowo omówione w ścieżce 2.

8. Rozwiązania ćwiczeń

Mamy nadzieję, że ćwiczenia praktyczne były pomocne w ugruntowaniu Twojego zrozumienia pojęć SQL. Jeśli utknąłeś w którymś z nich lub chcesz sprawdzić swoje odpowiedzi, zapoznaj się z naszymi odpowiedziami poniżej

Problem 1: Napisz zapytanie SQL, aby uzyskać nazwy wszystkich parków z mniej niż 1 000 000 odwiedzających.

Problem ten wymaga podania nazw parków (pojedyncza kolumna) z wymogiem, aby liczba odwiedzających była mniejsza niż 1 000 000, co można określić w `WHERE`klauzuli.

```
SELECT name FROM park
WHERE park_visitors < 1000000
```

Problem 2: Napisz zapytanie SQL, aby uzyskać liczbę różnych miast w `park`tabeli

Całkowitą liczbę kolumn można obliczyć za pomocą `COUNT()`funkcji, ale ponieważ potrzebujesz tylko odrębnych miast (ponieważ niektóre miasta mają wiele parków), możesz użyć `DISTINCT`słowa kluczowego przed nazwą kolumny w `COUNT()`funkcji.

```
SELECT COUNT(DISTINCT city) FROM park
```

Problem 3: Napisz zapytanie SQL, aby uzyskać całkowitą liczbę odwiedzających parki w San Francisco.

Całkowitą liczbę odwiedzających można obliczyć za pomocą `SUM()`funkcji. Dodatkowo potrzebna jest `WHERE`klauzula określająca tylko parki znajdujące się w San Francisco.

```
SELECT SUM(park_visitors) FROM park
WHERE city = "San Francisco"
```

Problem 4: Napisz zapytanie SQL do 5 najlepszych parków (tylko nazwy) wraz z liczbą odwiedzających, które miały najwięcej odwiedzających, w porządku malejącym.

Zapytanie musi pobrać zarówno kolumny name, jak i park_visitors. Wyniki są sortowane według `park_visitors`kolumny w kolejności malejącej przy użyciu `ORDER BY`klauzuli. Ponieważ nie chcesz grupować wyników w innej kolumnie *i sortować w tych grupach* , `GROUP BY`klauzula nie jest konieczna.

```
SELECT name, park_visitors FROM park
ORDER BY park_visitors DESC
LIMIT 5
```

9. Gratulacje

W podsumowaniu:

- Relacyjne bazy danych umożliwiają przechowywanie danych zorganizowanych w tabele, kolumny i wiersze.
- Możesz pobrać dane z bazy danych za pomocą instrukcji SQL `SELECT`.
- Możesz użyć różnych klauzul w `SELECT`wyrażeniu, w tym `WHERE`, `GROUP BY`, `ORDER BY`i , `LIMIT`aby uszczegółowić zapytania.
- Możesz użyć funkcji agregujących, aby połączyć dane z wielu wierszy w jedną kolumnę.
- Możesz dodawać, aktualizować i usuwać wiersze w bazie danych, używając odpowiednio instrukcji SQL `INSERT`, `UPDATE`i `DELETE`.

Ucz się więcej

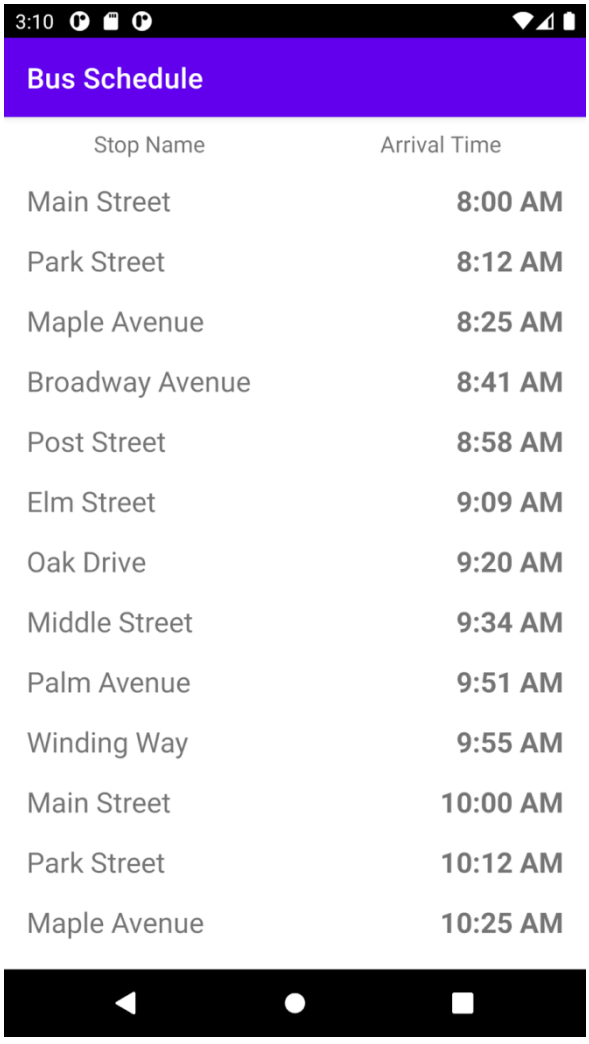
- Typy danych SQL
- Funkcje agregujące
- Łączy

Wprowadzenie do pomieszczenia i przepływu

1. Zanim zaczniesz

W poprzednim ćwiczeniu z programowania poznałeś podstawy relacyjnych baz danych oraz sposób odczytywania i zapisywania danych za pomocą poleceń SQL: SELECT, INSERT, UPDATE i DELETE. Nauka pracy z relacyjnymi bazami danych to podstawowa umiejętność, którą zabierzesz ze sobą podczas całej swojej przygody z programowaniem. Wiedza o tym, jak działają relacyjne bazy danych, jest również niezbędna do implementacji trwałości danych w aplikacji na Androida, którą zaczniesz robić w tej lekcji.

Prostym sposobem korzystania z bazy danych w aplikacji na Androida jest biblioteka o nazwie Room. Room to tak zwana biblioteka ORM (Object Relational Mapping), która, jak sama nazwa wskazuje, mapuje tabele w relacyjnej bazie danych na obiekty używane w kodzie Kotlin. W tej lekcji skupisz się tylko na czytaniu danych. Korzystając z wstępnie wypełnionej bazy danych, załadujesz dane z tabeli czasów przyjazdu autobusów i przedstawisz je w formacie `RecyclerView`.



W trakcie tego procesu poznasz podstawy korzystania z Room, w tym klasę bazy danych, obiekt DAO, encje i modele widoków. Zostaniesz również wprowadzony do `ListAdapter`klasy, innego sposobu prezentacji danych w `RecyclerView`, i przepływu, funkcji języka Kotlin podobnej do `LiveData`tej, która pozwoli Twojemu interfejsowi użytkownika reagować na zmiany w bazie danych.

Warunki wstępne

- Znajomość programowania obiektowego oraz używania klas, obiektów i dziedziczenia w Kotlinie.
- Podstawowa wiedza na temat relacyjnych baz danych i języka SQL wykładana na zajęciach z programowania podstaw języka SQL .
- Doświadczenie w stosowaniu współprogramów Kotlin.

Czego się nauczysz

Pod koniec tej lekcji powinieneś być w stanie

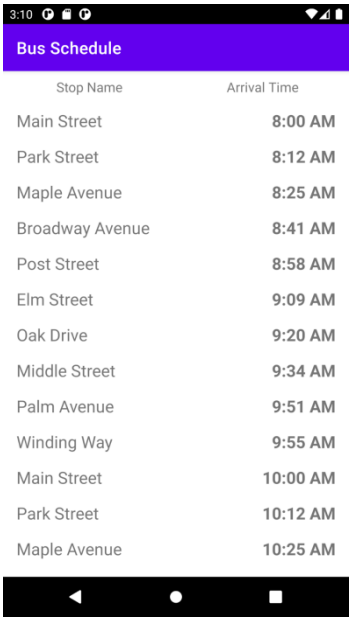
- Reprezentuj tabele bazy danych jako obiekty Kotlin (jednostki).
- Zdefiniuj klasę bazy danych, która ma używać Room w aplikacji, i wstępnie wypełnij bazę danych z pliku.
- Zdefiniuj klasę DAO i użyj zapytań SQL, aby uzyskać dostęp do bazy danych z kodu Kotlin.
- Zdefiniuj model widoku, aby umożliwić interakcję interfejsu użytkownika z DAO.
- Jak używać ListAdapter z widokiem recyklera.
- Podstawy przepływu Kotlin i jak go używać, aby interfejs użytkownika reagował na zmiany w danych bazowych.

Co zbudujesz

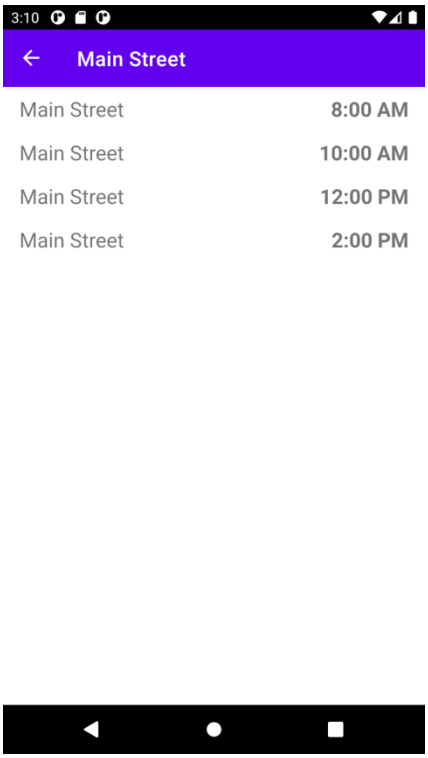
- Odczytuj dane z wstępnie wypełnionej bazy danych za pomocą Room i prezentuj je w widoku recyklera w prostej aplikacji rozkładu jazdy autobusów.

2. Rozpocznij

Aplikacja, z którą będziesz pracować podczas tego ćwiczenia programowania, nazywa się Harmonogram autobusów. Aplikacja prezentuje listę przystanków autobusowych i godziny przyjazdu od najwcześniejszych do najpóźniejszych.



Stuknięcie w wiersz na pierwszym ekranie prowadzi do nowego ekranu pokazującego tylko nadchodzące czasy przyjazdu na wybrany przystanek autobusowy.



Dane dotyczące przystanków autobusowych pochodzą z bazy danych, w której znajduje się aplikacja. Jednak w obecnym stanie nic nie zostanie wyświetlone, gdy aplikacja zostanie uruchomiona po raz pierwszy. Twoim zadaniem jest zintegrowanie pokoju, aby aplikacja wyświetlała wstępnie wypełnioną bazę danych o czasach przybycia.

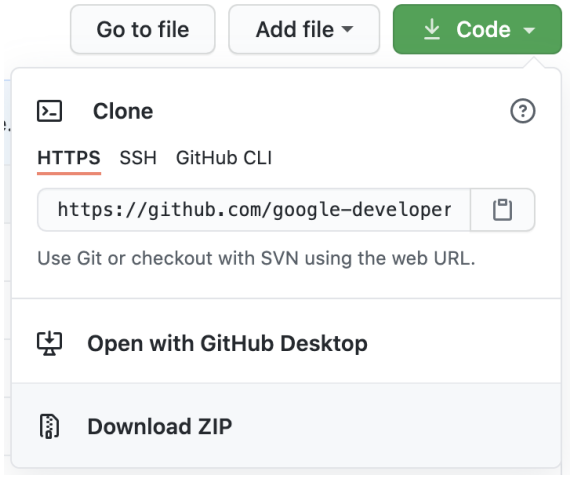
URL kodu startowego: https://github.com/google-developer-training/android-basics-kotlin-bus-schedule-app/tree/starter

Oddział:starter

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

Pobierz kod

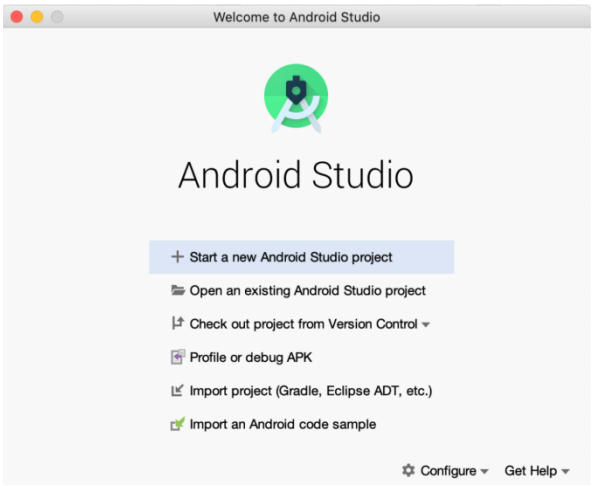
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



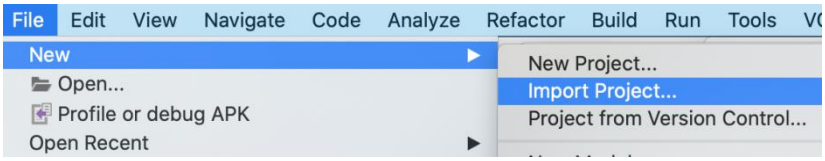
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

3. Dodaj zależność pokoju

Podobnie jak w przypadku każdej innej biblioteki, musisz najpierw dodać niezbędne zależności, aby móc korzystać z Room w aplikacji Bus Schedule. Będzie to wymagało tylko dwóch małych zmian, po jednej w każdym pliku Gradle.

1. `build.gradle`W pliku na poziomie projektu zdefiniuj `room_version`w bloku ext.

```
ext {
  kotlin_version = "1.3.72"
  nav_version = "2.3.1"
  room_version = '2.3.0'
}
```

2. W pliku na poziomie `build.gradle`aplikacji na końcu listy zależności dodaj następujące zależności.

```
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"

// optional - Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:$room_version"
```

3. Zsynchronizuj zmiany i skompiluj projekt, aby sprawdzić, czy zależności zostały dodane poprawnie.

Na następnych kilku stronach zapoznasz się z komponentami potrzebnymi do zintegrowania Room z aplikacją: modelami, DAO, modelami widoku i klasą bazy danych.

4. Utwórz podmiot

Kiedy poznałeś relacyjne bazy danych w poprzednim laboratorium programowania, zobaczyłeś, jak dane są zorganizowane w tabele składające się z wielu kolumn, z których każda reprezentuje określoną właściwość określonego typu danych. Podobnie jak klasy w Kotlinie zapewniają szablon dla każdego obiektu, tabela w bazie danych zapewnia szablon dla każdego elementu lub wiersza w tej tabeli. Nie powinno więc dziwić, że do reprezentowania każdej tabeli w bazie danych można użyć klasy Kotlin.

Podczas pracy z programem Room każda tabela jest reprezentowana przez klasę. W bibliotece ORM (Object Relational Mapping), takiej jak Room, są one często nazywane *klasami modeli* lub *jednostkami* .

Baza danych aplikacji Bus Schedule składa się tylko z jednej tabeli, harmonogramu, która zawiera podstawowe informacje o przyjeździe autobusu.

- `id`: liczba całkowita zapewniająca unikalny identyfikator, który służy jako klucz podstawowy
- `stop_name`: sznurek
- `arrival_time`: Liczba całkowita

Zauważ, że typy SQL używane w bazie danych to w rzeczywistości `INTEGER`for `Int`i `TEXT`for `String`. Jednak podczas pracy z Roomem podczas definiowania klas modeli powinieneś zajmować się tylko typami Kotlin. Mapowanie typów danych w klasie modelu na te używane w bazie danych jest obsługiwane automatycznie.

Gdy projekt zawiera wiele plików, należy rozważyć zorganizowanie plików w różnych pakietach, aby zapewnić lepszą kontrolę dostępu dla każdej klasy i ułatwić lokalizowanie powiązanych klas. Aby utworzyć jednostkę dla tabeli „schedule”, w pakiecie **com.example.busschedule** dodaj nowy pakiet o nazwie **database** . W ramach tego pakietu dodaj nowy pakiet o nazwie **harmonogram** dla swojej jednostki. Następnie w pakiecie **database.schedule** utwórz nowy plik o nazwie **Schedule.kt** i zdefiniuj klasę danych o nazwie `Schedule`.

```
data class Schedule(
)
```

Jak omówiono w lekcji Podstawy SQL, tabele danych powinny mieć klucz podstawowy, aby jednoznacznie identyfikować każdy wiersz. Pierwsza właściwość, którą dodasz do `Schedule`klasy, to liczba całkowita reprezentująca unikalny identyfikator. Dodaj nową właściwość i oznacz ją `@PrimaryKey`adnotacją. Dzięki temu Room ma traktować tę właściwość jako klucz podstawowy podczas wstawiania nowych wierszy.

```
@PrimaryKey val id: Int
```

Dodaj kolumnę z nazwą przystanku autobusowego. Kolumna powinna być typu `String`. W przypadku nowych kolumn musisz dodać `@ColumnInfo`adnotację, aby określić nazwę kolumny. Zazwyczaj nazwy kolumn SQL mają słowa oddzielone podkreśleniem, w przeciwieństwie do lowerCamelCase używanego we właściwościach Kotlin. W przypadku tej kolumny również nie chcemy, aby wartość była null , więc należy ją oznaczyć `@NonNull`adnotacją.

```
@NonNull @ColumnInfo(name = "stop_name") val stopName: String,
```

Uwaga: W SQL kolumny mogą domyślnie mieć wartości null i muszą być wyraźnie oznaczone jako nie null, jeśli chcesz inaczej. Jest to przeciwieństwo działania w Kotlinie, gdzie wartości nie mogą być domyślnie puste.

Czasy przybycia są reprezentowane w bazie danych za pomocą liczb całkowitych. Jest to uniksowy znacznik czasu , który można przekonwertować na użyteczną datę. Chociaż różne wersje SQL oferują sposoby konwertowania dat, do swoich celów będziesz trzymać się funkcji formatowania daty Kotlin. Dodaj następującą `@NonNull`kolumnę do klasy modelu.

```
@NonNull @ColumnInfo(name = "arrival_time") val arrivalTime: Int
```

Wreszcie, aby Room rozpoznał tę klasę jako coś, czego można użyć do zdefiniowania tabel bazy danych, musisz dodać adnotację do samej klasy. Dodaj `@Entity`w osobnym wierszu przed nazwą klasy.

Domyślnie Room używa nazwy klasy jako nazwy tabeli bazy danych. Tak więc nazwa tabeli zdefiniowana przez klasę w tej chwili to Schedule. Opcjonalnie można również określić @Entity(tableName="schedule"), ale ponieważ w zapytaniach o pokój nie jest rozróżniana wielkość liter, można pominąć jawne definiowanie nazwy tabeli małymi literami w tym miejscu.

Klasa encji harmonogramu powinna teraz wyglądać następująco.

```
@Entity
data class Schedule(
  @PrimaryKey val id: Int,
  @NonNull @ColumnInfo(name = "stop_name") val stopName: String,
  @NonNull @ColumnInfo(name = "arrival_time") val arrivalTime: Int
)
```

5. Zdefiniuj DAO

Kolejną klasą, którą musisz dodać, aby zintegrować Room, jest DAO. DAO to skrót od Data Access Object i jest klasą Kotlin, która zapewnia dostęp do danych. W szczególności DAO to miejsce, w którym można uwzględnić funkcje do czytania i manipulowania danymi. Wywołanie funkcji na DAO jest równoznaczne z wykonaniem polecenia SQL w bazie danych. W rzeczywistości funkcje DAO, takie jak te, które zdefiniujesz w tej aplikacji, często określają polecenie SQL, dzięki czemu możesz dokładnie określić, co ma robić funkcja. Twoja wiedza na temat SQL z poprzedniego ćwiczenia z programowania przyda się podczas definiowania DAO.

1. Dodaj klasę DAO dla encji Schedule. W pakiecie **database.schedule** utwórz nowy plik o nazwie `ScheduleDao.kt`i zdefiniuj interfejs o nazwie `ScheduleDao`. Podobnie jak w przypadku `Schedule`klasy, musisz dodać adnotację, tym razem `@Dao`, aby interfejs mógł być używany z Room.

   ```
   @Dao
   interface ScheduleDao {
   }
   ```

   Uwaga: Chociaż DAO jest akronimem, konwencje nazewnictwa dla kodu Kotlin liczą się tylko do pierwszej litery w akronimach, a więc nazwy, `ScheduleDao`a nie `ScheduleDAO`.

2. W aplikacji są dwa ekrany i każdy będzie wymagał innego zapytania. Pierwszy ekran pokazuje wszystkie przystanki autobusowe w kolejności rosnącej według czasu przybycia. W takim przypadku zapytanie musi tylko pobrać wszystkie kolumny i zawierać odpowiednią `ORDER BY`klauzulę. Zapytanie jest określone jako ciąg znaków przekazany do `@Query`adnotacji. Zdefiniuj funkcję `getAll()`, która zwraca listę `Schedule`obiektów wraz z `@Query`adnotacją, jak pokazano.

   ```
   @Query("SELECT * FROM schedule ORDER BY arrival_time ASC")
   fun getAll(): List<Schedule>
   ```

3. W przypadku drugiego zapytania chcesz również wybrać wszystkie kolumny z tabeli zestawieniowej. Jednak potrzebujesz tylko wyników, które pasują do wybranej nazwy przystanku, więc musisz dodać `WHERE`klauzulę. Możesz odwoływać się do wartości Kotlin z zapytania,

poprzedzając je dwukropkiem (:) (np :stopName. z parametru funkcji). Podobnie jak poprzednio, wyniki są uporządkowane rosnąco według czasu przybycia. Zdefiniuj getByStopName()funkcję, która pobiera Stringwywoływany parametr stopNamei zwraca obiekt z Listadnotacją , jak pokazano.Schedule@Query

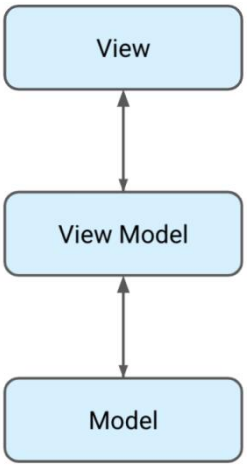
```
@Query("SELECT * FROM schedule WHERE stop_name = :stopName ORDER BY arrival_time ASC")
fun getByStopName(stopName: String): List<Schedule>
```

6. Zdefiniuj ViewModel

Teraz, gdy już skonfigurowałeś DAO, technicznie masz wszystko, czego potrzebujesz, aby rozpocząć dostęp do bazy danych ze swoich fragmentów. Jednak, chociaż działa to teoretycznie, generalnie nie jest uważane za najlepszą praktykę. Powodem jest to, że w bardziej złożonych aplikacjach prawdopodobnie masz wiele ekranów, które mają dostęp tylko do określonej części danych. Chociaż ScheduleDaojest to stosunkowo proste, łatwo zauważyć, jak może to wymknąć się spod kontroli podczas pracy z dwoma lub więcej różnymi ekranami. Na przykład DAO może wyglądać mniej więcej tak:

```
@Dao
interface ScheduleDao {

    @Query(...)
    getForScreenOne() ...

    @Query(...)
    getForScreenTwo() ...

    @Query(...)
    getForScreenThree()

}
```

Chociaż kod dla ekranu 1 może uzyskać dostęp do getForScreenOne(), nie ma dobrego powodu, aby uzyskać dostęp do innych metod. Zamiast tego uważa się, że najlepszą praktyką jest oddzielenie części DAO, którą uwidaczniasz w widoku, do oddzielnej klasy zwanej *modelem widoku* . Jest to powszechny wzorzec architektoniczny w aplikacjach mobilnych. Korzystanie z modelu widoku pomaga wymusić wyraźne oddzielenie kodu interfejsu użytkownika aplikacji od jej modelu danych. Pomaga również w niezależnym testowaniu każdej części kodu, co zgłębisz dalej, kontynuując swoją przygodę z programowaniem Androida.



Korzystając z modelu widoku, możesz skorzystać z `ViewModel`klasy. Klasa `ViewModel`służy do przechowywania danych związanych z interfejsem użytkownika aplikacji, a także jest świadoma cyklu życia, co oznacza, że reaguje na zdarzenia cyklu życia podobnie jak działanie lub fragment. Jeśli zdarzenia cyklu życia, takie jak obracanie ekranu, spowodują zniszczenie i odtworzenie działania lub fragmentu, skojarzone `ViewModel`nie będzie musiało zostać ponownie utworzone. Nie jest to możliwe przy bezpośrednim dostępie do klasy DAO, więc najlepiej jest użyć `ViewModel`podklasy, aby oddzielić odpowiedzialność za ładowanie danych od aktywności lub fragmentu.

Uwaga: Rozkład jazdy to stosunkowo prosta aplikacja i zawiera tylko dwa ekrany o w większości identycznej treści. Do celów dydaktycznych utworzymy jedną klasę modelu widoku, która może być używana przez oba ekrany, ale w większej aplikacji możesz chcieć użyć oddzielnego modelu widoku dla każdego fragmentu.

1. Aby utworzyć klasę modelu widoku, utwórz nowy plik o nazwie **BusScheduleViewModel.kt** w nowym pakiecie o nazwie **viewmodels** . Zdefiniuj klasę dla modelu widoku. Powinien przyjąć jeden parametr typu `ScheduleDao`.

   ```
   class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel() {
   ```

2. Ponieważ ten model widoku będzie używany z obydwoma ekranami, musisz dodać metodę, aby uzyskać pełny harmonogram, a także harmonogram przefiltrowany według nazwy przystanku. Możesz to zrobić, wywołując odpowiednie metody z `ScheduleDao`.

   ```
   fun fullSchedule(): List<Schedule> = scheduleDao.getAll()


   fun scheduleForStopName(name: String): List<Schedule> = scheduleDao.getByStopName(name)
   ```

   Chociaż zakończyłeś definiowanie modelu widoku, nie możesz po prostu `BusScheduleViewModel`bezpośrednio utworzyć instancji i oczekiwać, że wszystko będzie działać. Ponieważ klasa ViewModel `BusScheduleViewModel`musi być świadoma cyklu życia, powinna być tworzona przez obiekt, który może reagować na zdarzenia cyklu życia. Jeśli utworzysz jego wystąpienie bezpośrednio w jednym z twoich fragmentów, twój obiekt fragmentu będzie musiał obsłużyć całe zarządzanie pamięcią, co wykracza poza zakres tego, co powinien robić kod twojej aplikacji. Zamiast tego możesz utworzyć klasę, zwaną fabryką, która będzie tworzyć instancje obiektów widoku modelu.

1. Aby utworzyć fabrykę, poniżej klasy modelu widoku utwórz nową klasę `BusScheduleViewModelFactory`, która dziedziczy po `ViewModelProvider.Factory`.

```
class BusScheduleViewModelFactory(
  private val scheduleDao: ScheduleDao
) : ViewModelProvider.Factory {
}
```

2. Potrzebujesz tylko trochę kodu wzorcowego, aby poprawnie utworzyć instancję modelu widoku. Zamiast inicjowania klasy bezpośrednio, nadpiszesz wywołaną metodę `create()`, która zwraca a `BusScheduleViewModelFactory`z pewnym sprawdzaniem błędów. Zaimplementuj `create()`wnętrze `BusScheduleViewModelFactory`klasy w następujący sposób.

```
override fun <T : ViewModel> create(modelClass: Class<T>): T {
    if (modelClass.isAssignableFrom(BusScheduleViewModel::class.java)) {
      @Suppress("UNCHECKED_CAST")
      return BusScheduleViewModel(scheduleDao) as T
    }
    throw IllegalArgumentException("Unknown ViewModel class")
  }
```

Możesz teraz utworzyć instancję `BusScheduleViewModelFactory`obiektu za pomocą `BusScheduleViewModelFactory.create()`, dzięki czemu Twój model widoku może być świadomy cyklu życia bez konieczności obsługi tego fragmentu bezpośrednio.

7. Utwórz klasę bazy danych i wstępnie wypełnij bazę danych

Teraz, gdy już zdefiniowałeś modele, DAO i model widoku dla fragmentów, aby uzyskać dostęp do DAO, nadal musisz powiedzieć Roomowi, co ma zrobić ze wszystkimi tymi klasami. W tym `AppDatabase`miejscu pojawia się klasa. Aplikacja na Androida korzystająca z Room, taka jak Twoja, tworzy podklasy dla `RoomDatabase`klasy i ma kilka kluczowych obowiązków. W Twojej aplikacji `AppDatabase`należy :

1. Określ, które jednostki są zdefiniowane w bazie danych.
2. Zapewnij dostęp do pojedynczego wystąpienia każdej klasy DAO.
3. Wykonaj dowolną dodatkową konfigurację, taką jak wstępne wypełnienie bazy danych.

Chociaż możesz się zastanawiać, dlaczego Room nie może po prostu znaleźć dla Ciebie wszystkich encji i obiektów DAO, całkiem możliwe, że Twoja aplikacja może mieć wiele baz danych lub dowolną liczbę scenariuszy, w których biblioteka nie może przyjąć Twoich intencji, deweloper. Klasa `AppDatabase`zapewnia pełną kontrolę nad modelami, klasami DAO i dowolną konfiguracją bazy danych, którą chcesz wykonać.

1. Aby dodać `AppDatabase`klasę, w pakiecie **bazy danych** utwórz nowy plik o nazwie `AppDatabase.kt`i zdefiniuj nową klasę abstrakcyjną, `AppDatabase`która dziedziczy po `RoomDatabase`.

```
abstract class AppDatabase: RoomDatabase() {
}
```

2. Klasa bazy danych umożliwia innym klasom łatwy dostęp do klas DAO. Dodaj funkcję abstrakcyjną, która zwraca `ScheduleDao`.

   ```
   abstract fun scheduleDao(): ScheduleDao
   ```

3. Korzystając z `AppDatabase`klasy, chcesz mieć pewność, że istnieje tylko jedna instancja bazy danych, aby zapobiec wyścigom lub innym potencjalnym problemom. Instancja jest przechowywana w obiekcie towarzyszącym i będziesz potrzebować również metody, która albo zwróci istniejącą instancję, albo po raz pierwszy utworzy bazę danych. Jest to zdefiniowane w obiekcie towarzyszącym. Dodaj poniższe `companion object`tuż pod `scheduleDao()`funkcją.

   ```
   companion object {
   }
   ```

   W `companion object`, dodaj właściwość o nazwie `INSTANCE`type `AppDatabase`. Ta wartość jest początkowo ustawiona na `null`, więc typ jest oznaczony `?`. Jest to również oznaczone `@Volatile`adnotacją. Chociaż szczegóły dotyczące tego, kiedy użyć właściwości volatile, są nieco zaawansowane w tej lekcji, warto użyć jej w swojej `AppDatabase`instancji, aby uniknąć potencjalnych błędów.

   ```
   @Volatile
   private var INSTANCE: AppDatabase? = null
   ```

   Poniżej `INSTANCE`właściwości zdefiniuj funkcję zwracającą `AppDatabase`instancję

   ```
   fun getDatabase(context: Context): AppDatabase {
      return INSTANCE ?: synchronized(this) {
         val instance = Room.databaseBuilder(
            context,
            AppDatabase::class.java,
            "app_database")
            .createFromAsset("database/bus_schedule.db")
            .build()
         INSTANCE = instance

         instance
      }
   }
   ```

   W implementacji dla `getDatabase()`, używasz operatora Elvis, aby albo zwrócić istniejącą instancję bazy danych (jeśli już istnieje) albo utworzyć bazę danych po raz pierwszy, jeśli to konieczne. W tej aplikacji, ponieważ dane są wstępnie wypełnione. Wołasz również, `createFromAsset()`aby załadować istniejące dane. Plik `bus_schedule.db`znajdziesz w `assets.database`pakiecie w swoim projekcie.

4. Podobnie jak klasy modeli i DAO, klasa bazy danych wymaga adnotacji zawierającej określone informacje. Wszystkie typy jednostek (dostęp do samego typu za pomocą `ClassName::class`) są wymienione w tablicy. Baza danych otrzymuje również numer wersji, który ustawisz na 1. Dodaj `@Database`adnotację w następujący sposób.

```
@Database(entities = arrayOf(Schedule::class), version = 1)
```

Uwaga: Numer wersji jest zwiększany za każdym razem, gdy dokonujesz zmiany schematu. Aplikacja porównuje tę wersję z wersją w bazie danych, aby określić, czy i jak należy przeprowadzić migrację.

Teraz, po utworzeniu `AppDatabase`klasy, pozostał tylko jeden krok, aby można było z niej korzystać. Musisz podać niestandardową podklasę `Application`klasy i utworzyć `lazy`właściwość, która będzie przechowywać wynik `getDatabase()`.

5. W pakiecie **com.example.busschedule** dodaj nowy plik o nazwie `BusScheduleApplication.kt`i utwórz `BusScheduleApplication`klasę, która dziedziczy po `Application`.

```
class BusScheduleApplication : Application() {
}
```

6. Dodaj właściwość bazy danych typu `AppDatabase`. Właściwość powinna być leniwa i zwracać wynik wywołania `getDatabase()`twojej `AppDatabase`klasy.

```
class BusScheduleApplication : Application() {
  val database: AppDatabase by lazy { AppDatabase.getDatabase(this) }
```

7. Na koniec, aby upewnić się, że `BusScheduleApplication`klasa jest używana (zamiast domyślnej klasy bazowej `Application`), musisz dokonać niewielkiej zmiany w manifeście. W `AndroidMainifest.xml`programie ustaw `android:name`właściwość na `com.example.busschedule.BusScheduleApplication`.

```
<application
   android:name="com.example.busschedule.BusScheduleApplication"
   ...
```

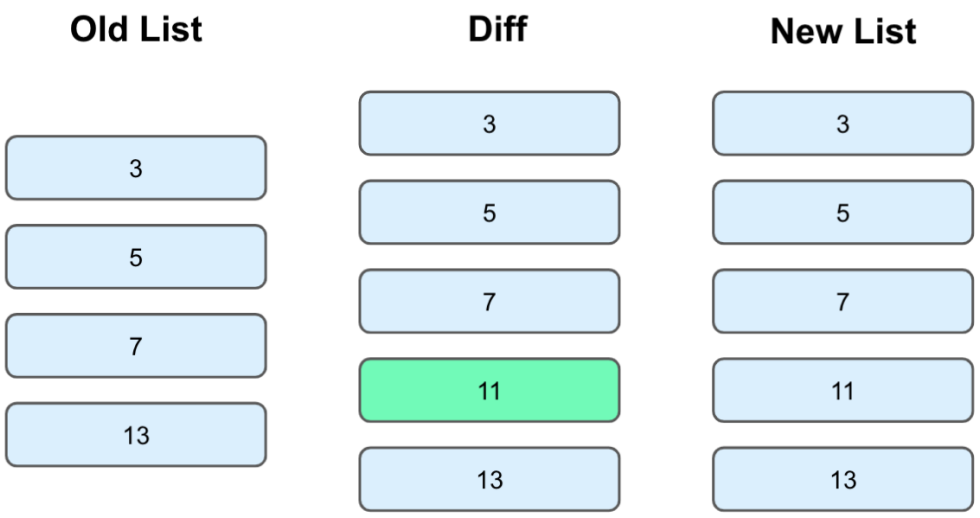
To tyle, jeśli chodzi o konfigurację modelu Twojej aplikacji. Wszystko gotowe do korzystania z danych z pokoju w interfejsie użytkownika. Na następnych kilku stronach utworzysz `ListAdapter`dla swojej aplikacji, `RecyclerView`aby prezentować dane rozkładu jazdy autobusów i dynamicznie reagować na zmiany danych.

8. Utwórz ListAdapter

Czas wziąć całą tę ciężką pracę i podłączyć model do widoku. Wcześniej, używając `RecyclerView`, używałeś `RecyclerView`. `Adapter`do przedstawienia statycznej listy danych. Chociaż z pewnością zadziała to w przypadku aplikacji takiej jak Bus Schedule, typowym scenariuszem podczas pracy z bazami danych jest obsługa zmian danych w czasie rzeczywistym. Nawet jeśli zmieni się zawartość tylko jednego elementu, cały widok recyklera zostanie odświeżony. To nie wystarczy w przypadku większości aplikacji korzystających z trwałości.

Alternatywą dla dynamicznie zmieniającej się listy jest nazwa `ListAdapter`. `ListAdapter`używa AsyncListDiffer do określenia różnic między starą listą danych a nową listą danych. Następnie widok recyklera jest aktualizowany tylko na podstawie różnic

między tymi dwiema listami. W rezultacie widok recyklera jest bardziej wydajny podczas obsługi często aktualizowanych danych, co często zdarza się w przypadku aplikacji bazodanowych.



Ponieważ interfejs użytkownika jest identyczny dla obu ekranów, wystarczy utworzyć jeden `ListAdapter`, którego można używać z obydwoma ekranami.

1. Utwórz nowy plik `BusStopAdapter.kt`i `BusStopAdapter`klasę, jak pokazano. Klasa jest rozszerzeniem generycznego `ListAdapter`, który pobiera listę `Schedule`obiektów i `BusStopViewHolder`klasę interfejsu użytkownika. W przypadku `BusStopViewHolder`, podajesz również `DiffCallback`typ, który wkrótce zdefiniujesz. Sama `BusStopAdapter`klasa również przyjmuje parametr `onItemClicked()`. Ta funkcja będzie używana do obsługi nawigacji, gdy element jest zaznaczony na pierwszym ekranie, ale na drugim ekranie po prostu przekażesz pustą funkcję.

```
class BusStopAdapter(private val onItemClicked: (Schedule) -> Unit) : ListAdapter<Schedule,
BusStopAdapter.BusStopViewHolder>(DiffCallback) {
}
```

2. Podobnie jak w przypadku adaptera widoku recyklera, potrzebujesz posiadacza widoku, aby uzyskać dostęp do widoków utworzonych z pliku układu w kodzie. Układ komórek jest już utworzony. Po prostu utwórz `BusStopViewHolder`klasę, jak pokazano, i zaimplementuj `bind()`funkcję, aby ustawić `stopNameTextView`tekst na nazwę zatrzymania `arrivalTimeTextView`, a tekst na sformatowaną datę.

```
class BusStopViewHolder(private var binding: BusStopItemBinding):
RecyclerView.ViewHolder(binding.root) {
   @SuppressLint("SimpleDateFormat")
   fun bind(schedule: Schedule) {
      binding.stopNameTextView.text = schedule.stopName
      binding.arrivalTimeTextView.text = SimpleDateFormat(
         "h:mm a").format(Date(schedule.arrivalTime.toLong() * 1000)
      )
   }
}
```

3. Zastąp i zaimplementuj `onCreateViewHolder()`i napełnij układ oraz ustaw `onClickListener()`wywołanie `onItemClicked()`elementu w bieżącej pozycji.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): BusStopViewHolder {
  val viewHolder = BusStopViewHolder(
    BusStopItemBinding.inflate(
      LayoutInflater.from( parent.context),
      parent,
      false
    )
  )
  viewHolder.itemView.setOnClickListener {
    val position = viewHolder.adapterPosition
    onItemClicked(getItem(position))
  }
  return viewHolder
}
```

4. Zastąp i zaimplementuj `onBindViewHolder()`oraz powiąż widok w określonej pozycji.

```
override fun onBindViewHolder(holder: BusStopViewHolder, position: Int) {
  holder.bind(getItem(position))
}
```

5. Pamiętasz tę `DiffCallback`klasę, którą określiłeś dla `ListAdapter`? Jest to tylko obiekt, który pomaga `ListAdapter`określić, które pozycje na nowej i starej liście różnią się podczas aktualizacji listy. Istnieją dwie metody: `areItemsTheSame()`sprawdzenie, czy obiekt (lub wiersz w bazie danych w twoim przypadku) jest taki sam, sprawdzając tylko identyfikator. `areContentsTheSame()`sprawdza, czy wszystkie właściwości, a nie tylko identyfikator, są takie same. Metody te pozwalają `ListAdapter`określić, które elementy zostały wstawione, zaktualizowane i usunięte, aby można było odpowiednio zaktualizować interfejs użytkownika.

   Dodaj obiekt towarzyszący i zaimplementuj `DiffCallback`, jak pokazano.

```
companion object {
  private val DiffCallback = object : DiffUtil.ItemCallback<Schedule>() {
    override fun areItemsTheSame(oldItem: Schedule, newItem: Schedule): Boolean {
      return oldItem.id == newItem.id
    }

    override fun areContentsTheSame(oldItem: Schedule, newItem: Schedule): Boolean {
      return oldItem == newItem
    }
  }
}
```

   To wszystko, co trzeba zrobić, aby skonfigurować adapter. Będziesz go używać na obu ekranach aplikacji.

1. Najpierw w `FullScheduleFragment.kt`programie musisz uzyskać odniesienie do modelu widoku.

```
private val viewModel: BusScheduleViewModel by activityViewModels {
  BusScheduleViewModelFactory(
    (activity?.application as BusScheduleApplication).database.scheduleDao()
  )
}
```

2. Następnie w `onViewCreated()`programie dodaj następujący kod, aby skonfigurować widok recyklera i przypisać jego menedżera układu.

```
recyclerView = binding.recyclerView
recyclerView.layoutManager = LinearLayoutManager(requireContext())
```

3. Następnie przypisz właściwość adaptera. Przekazana akcja użyje `stopName`do nawigacji na wybranym następnym ekranie, aby można było filtrować listę przystanków autobusowych.

```
val busStopAdapter = BusStopAdapter({
  val action = FullScheduleFragmentDirections.actionFullScheduleFragmentToStopScheduleFragment(
    stopName = it.stopName
  )
  view.findNavController().navigate(action)
})
recyclerView.adapter = busStopAdapter
```

4. Na koniec, aby zaktualizować widok listy, zadzwoń `submitList()`, przekazując listę przystanków autobusowych z modelu widoku.

```
// submitList() is a call that accesses the database. To prevent the
// call from potentially locking the UI, you should use a
// coroutine scope to launch the function. Using GlobalScope is not
// best practice, and in the next step we'll see how to improve this.
GlobalScope.launch(Dispatchers.IO) {
  busStopAdapter.submitList(viewModel.fullSchedule())
}
```

5. Zrób to samo w `StopScheduleFragment`. Najpierw zdobądź odniesienie do modelu widoku.

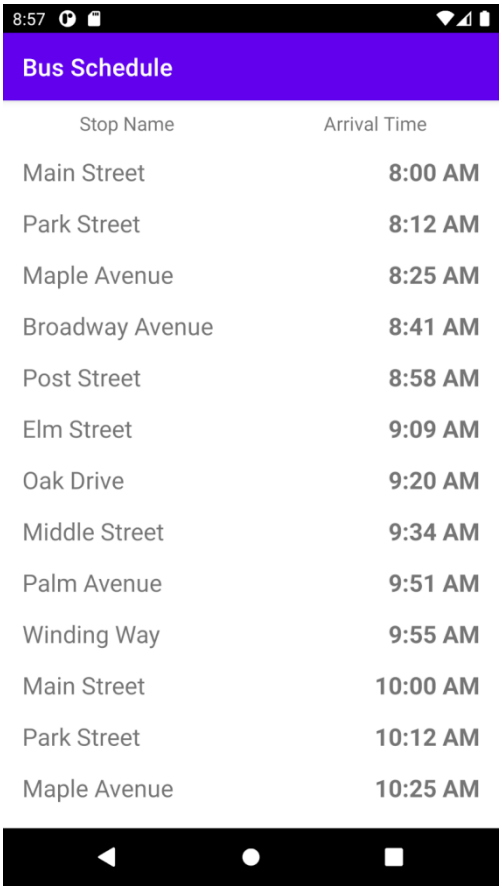
```
private val viewModel: BusScheduleViewModel by activityViewModels {
  BusScheduleViewModelFactory(
    (activity?.application as BusScheduleApplication).database.scheduleDao()
  )
}
```

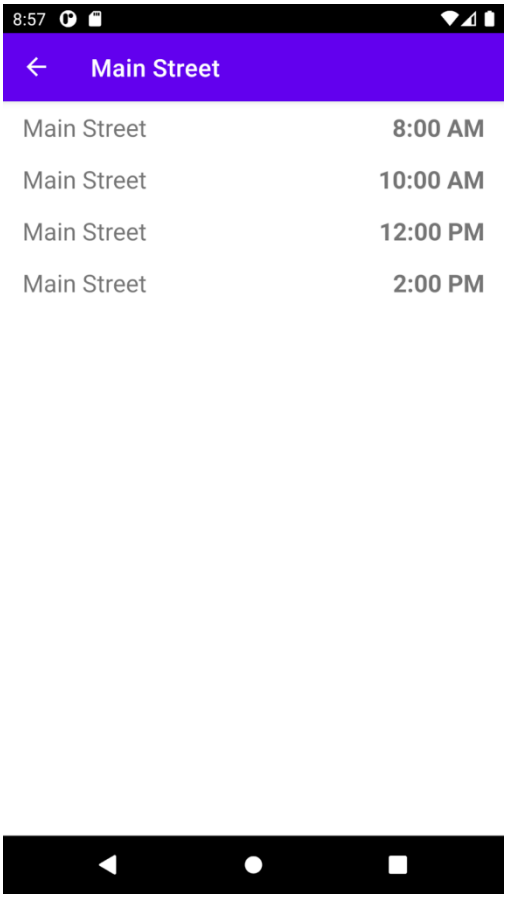
6. Następnie skonfiguruj widok recyklingu w `onViewCreated()`. Tym razem wystarczy przekazać pusty blok (funkcję) za pomocą `{}`. Właściwie nie chcesz, aby cokolwiek się działo po dotknięciu wierszy na tym ekranie.

```
recyclerView = binding.recyclerView
recyclerView.layoutManager = LinearLayoutManager(requireContext())
```

```
val busStopAdapter = BusStopAdapter({})
recyclerView.adapter = busStopAdapter
// submitList() is a call that accesses the database. To prevent the
// call from potentially locking the UI, you should use a
// coroutine scope to launch the function. Using GlobalScope is not
// best practice, and in the next step we'll see how to improve this.
GlobalScope.launch(Dispatchers.IO) {
  busStopAdapter.submitList(viewModel.scheduleForStopName(stopName))
}
```

7. Teraz, po skonfigurowaniu adaptera, integracja Room z aplikacją Bus Schedule jest zakończona. Poświęć chwilę na uruchomienie aplikacji i powinieneś zobaczyć listę czasów przyjazdu. Dotknięcie wiersza powinno przejść do ekranu szczegółów.





9. Reaguj na zmiany danych za pomocą Flow

Chociaż widok listy jest skonfigurowany tak, aby skutecznie obsługiwać zmiany danych przy każdym `submitList()`wywołaniu, Twoja aplikacja nie będzie jeszcze w stanie obsługiwać aktualizacji dynamicznych. Aby zobaczyć na własne oczy, spróbuj otworzyć Inspektora bazy danych i uruchomić następującą kwerendę, aby wstawić nowy element do tabeli harmonogramu.

```
INSERT INTO schedule
VALUES (null, 'Winding Way', 1617202500)
```

Zauważysz jednak, że w emulatorze nic się nie dzieje. Użytkownik zakłada, że dane są niezmienione. Aby zobaczyć zmiany, musisz ponownie uruchomić aplikację.

Problem polega na tym, że `List<Schedule>`z każdej funkcji DAO zwracany jest tylko raz. Nawet jeśli dane bazowe zostaną zaktualizowane, `submitList()`nie będą wywoływane w celu aktualizacji interfejsu użytkownika, az perspektywy użytkownika będzie wyglądać, jakby nic się nie zmieniło.

Aby to naprawić, możesz skorzystać z funkcji Kotlina zwanej *przepływem asynchronicznym* (często nazywanej po prostu *flow*), która pozwoli DAO na ciągłe emitowanie danych z bazy danych. Jeśli element zostanie wstawiony, zaktualizowany lub usunięty, wynik zostanie odesłany z powrotem do fragmentu. Korzystając z funkcji o nazwie `collect()`,, możesz wywołać `submitList()`przy użyciu nowej wartości emitowanej z przepływu, dzięki czemu `ListAdapter`możesz zaktualizować interfejs użytkownika na podstawie nowych danych.

1. Aby użyć przepływu w harmonogramie autobusów, otwórz `ScheduleDao.kt`. Aby przekonwertować funkcje DAO na zwracające a `Flow`, po prostu zmień typ zwracany `getAll()`funkcji na `Flow<List<Schedule>>`.

   ```
   fun getAll(): Flow<List<Schedule>>
   ```

2. Podobnie zaktualizuj wartość zwracaną przez `getByStopName()`funkcję.

   ```
   fun getByStopName(stopName: String): Flow<List<Schedule>>
   ```

3. Należy również zaktualizować funkcje w modelu widoku, które uzyskują dostęp do DAO. Zaktualizuj zwracane wartości do `Flow<List<Schedule>>`obu `fullSchedule()`i `scheduleForStopName()`.

   ```
   class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel() {

     fun fullSchedule(): Flow<List<Schedule>> = scheduleDao.getAll()

     fun scheduleForStopName(name: String): Flow<List<Schedule>> =
   scheduleDao.getByStopName(name)
   }
   ```

4. Wreszcie, w `FullScheduleFragment.kt`, `busStopAdapter`powinien zostać zaktualizowany po wywołaniu `collect()`wyników zapytania. Ponieważ `fullSchedule()`jest funkcją zawieszenia, należy ją wywołać ze współprogramu. Wymień linię.

   ```
   busStopAdapter.submitList(viewModel.fullSchedule())
   ```

   Z tym kodem, który używa przepływu zwróconego z `fullSchedule()`.

   ```
   lifecycle.coroutineScope.launch {
     viewModel.fullSchedule().collect() {
       busStopAdapter.submitList(it)
     }
   ```

```
}
```

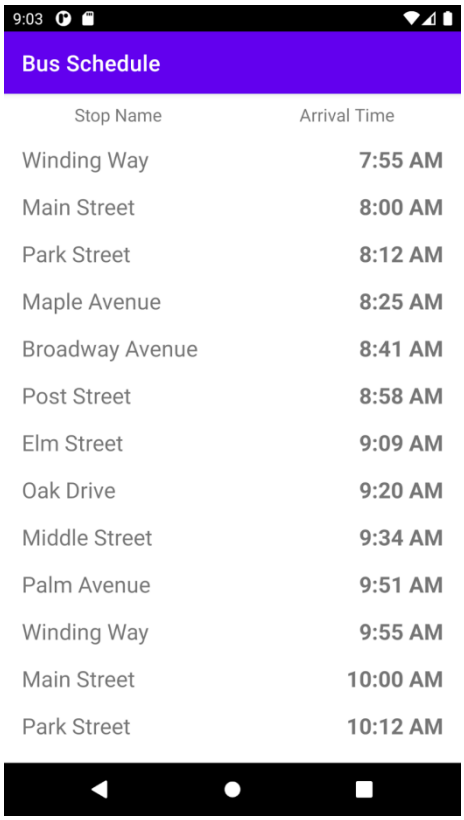
5. Zrób to samo w `StopScheduleFragment`, ale zastąp wywołanie `scheduleForStopName()`, następującym.

```
lifecycle.coroutineScope.launch {
  viewModel.scheduleForStopName(stopName).collect() {
     busStopAdapter.submitList(it)
  }
}
```

6. Po wprowadzeniu powyższych zmian możesz ponownie uruchomić aplikację, aby sprawdzić, czy zmiany danych są teraz obsługiwane w czasie rzeczywistym. Po uruchomieniu aplikacji wróć do Inspektora bazy danych i wyślij następujące zapytanie, aby wstawić nowy czas przybycia przed godziną 8:00.

```
INSERT INTO schedule
VALUES (null, 'Winding Way', 1617202500)
```

Nowa pozycja powinna pojawić się na górze listy.



To tyle, jeśli chodzi o aplikację Bus Schedule. Świetna robota, która dotarła tak daleko. Powinieneś mieć teraz solidne podstawy do pracy z Room. W następnej ścieżce zagłębisz się w Room z nową przykładową aplikacją i dowiesz się, jak zapisywać dane utworzone przez użytkownika na urządzeniu.

10. Kod rozwiązania

Kod rozwiązania dla tego ćwiczenia programowania znajduje się w projekcie i module pokazanym poniżej.

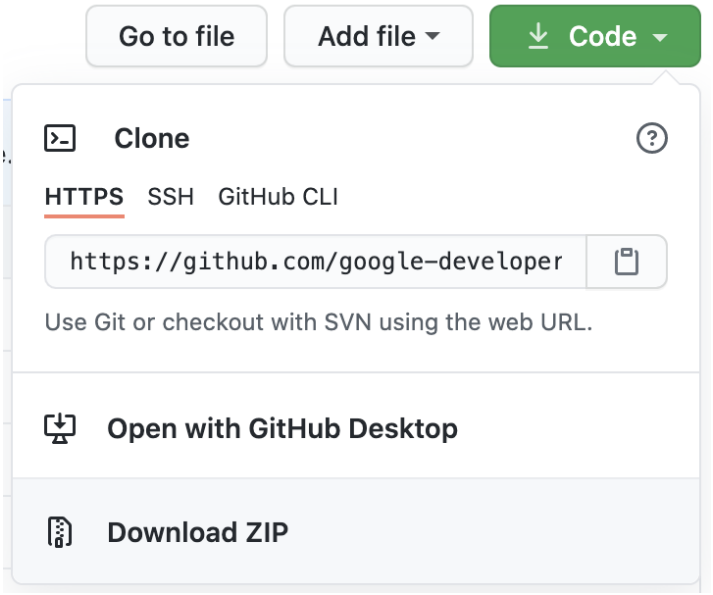
Adres URL kodu rozwiązania: https://github.com/google-developer-training/android-basics-kotlin-bus-schedule-app

Nazwa filii:main

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

Pobierz kod

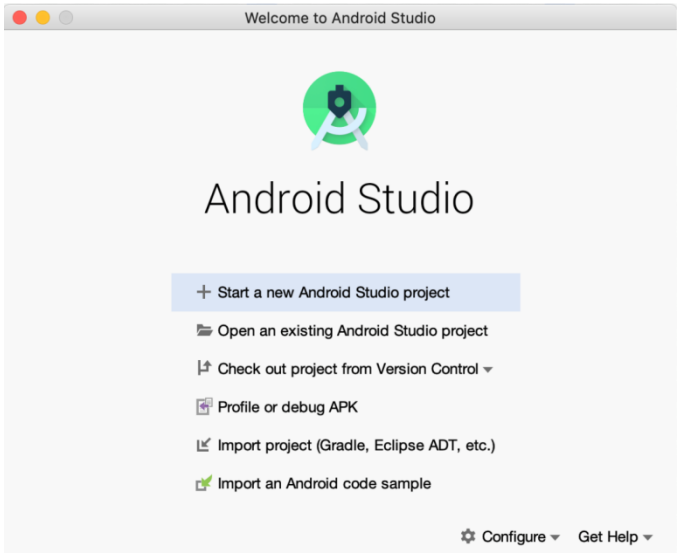
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



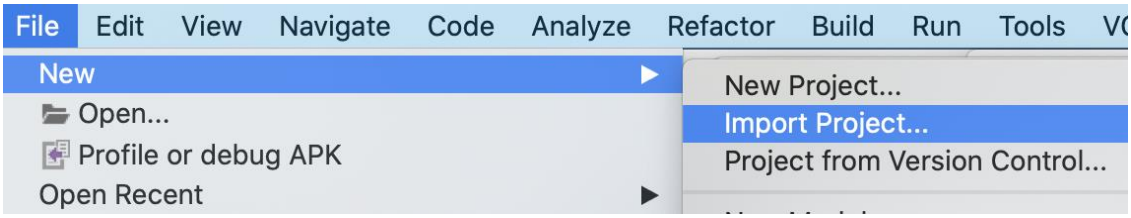
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

11. Gratulacje

W podsumowaniu:

- Tabele w bazie danych SQL są reprezentowane w Room przez klasy Kotlina zwane encjami.
- DAO udostępnia metody odpowiadające poleceniom SQL, które współdziałają z bazą danych.
- `ViewModel`to komponent uwzględniający cykl życia, używany do oddzielania danych aplikacji od jej widoku.
- Klasa `AppDatabase`informuje Room, których encji użyć, zapewnia dostęp do DAO i wykonuje dowolną konfigurację podczas tworzenia bazy danych.

- `ListAdapter`to adapter używany z `RecyclerView`, który jest idealny do obsługi dynamicznie aktualizowanych list.
- Flow to funkcja Kotlin do zwracania strumienia danych i może być używana z Room, aby zapewnić synchronizację interfejsu użytkownika i bazy danych.

Ucz się więcej

- ZobaczModel
- ViewModelProvider.Factory
- Baza danych sal
- @Adnotacja lotna
- ListAdapter
- AsyncListDiffer

Użyj pokoju do utrwalania danych

Użyj biblioteki pokoi, aby umożliwić aplikacjom odczytywanie i zapisywanie z bazy danych.

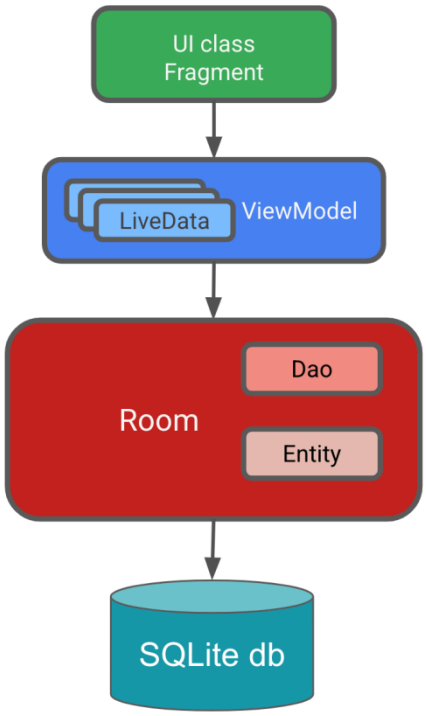
Utrwalaj dane z pokojem

1. Zanim zaczniesz

Większość aplikacji o jakości produkcyjnej zawiera dane, które należy zapisać, nawet po zamknięciu aplikacji przez użytkownika. Na przykład aplikacja może przechowywać listę odtwarzania utworów, pozycje na liście rzeczy do zrobienia, rejestr wydatków i dochodów, katalog konstelacji lub historię danych osobowych. W większości przypadków do przechowywania tych trwałych danych używana jest baza danych.

Room to biblioteka trwałości, która jest częścią Android Jetpack . Room to warstwa abstrakcji na bazie bazy danych SQLite . SQLite używa specjalistycznego języka (SQL) do wykonywania operacji na bazie danych. Zamiast bezpośrednio korzystać z SQLite, Room upraszcza prace związane z konfiguracją, konfiguracją i interakcją z bazą danych. Room zapewnia również sprawdzanie instrukcji SQLite w czasie kompilacji.

Poniższy obrazek pokazuje, jak Room wpisuje się w ogólną architekturę zalecaną w tym kursie.



Warunki wstępne

- Wiesz, jak zbudować podstawowy interfejs użytkownika (UI) dla aplikacji na Androida.

- Wiesz, jak korzystać z działań, fragmentów i widoków.
- Wiesz, jak poruszać się między fragmentami, używając bezpiecznych argumentów do przekazywania danych między fragmentami.
- Znasz składniki architektury systemu Android `ViewModel`, `LiveData`i `Flow`i wiesz, jak używać `ViewModelProvider.Factory`do tworzenia wystąpienia ViewModels.
- Znasz podstawy współbieżności.
- Wiesz, jak używać współprogramów do długotrwałych zadań.
- Masz podstawową wiedzę na temat baz danych SQL i języka SQLite.

Czego się nauczysz

- Jak tworzyć i współdziałać z bazą danych SQLite przy użyciu biblioteki Room.
- Jak utworzyć klasę encji, DAO i bazy danych.
- Jak używać obiektu dostępu do danych (DAO) do mapowania funkcji Kotlina na zapytania SQL.

Co zbudujesz

- Zbudujesz aplikację Inventory, która zapisuje elementy zapasów w bazie danych SQLite.

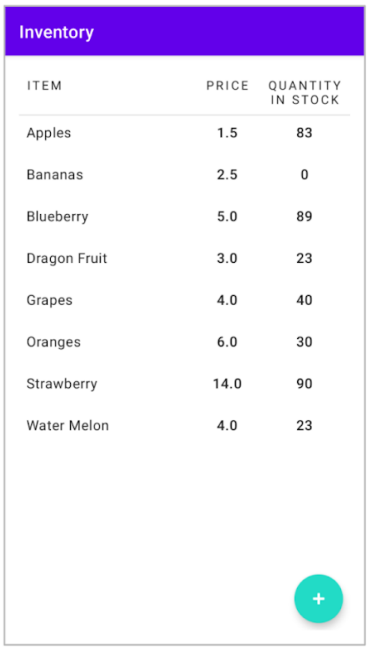
Czego potrzebujesz

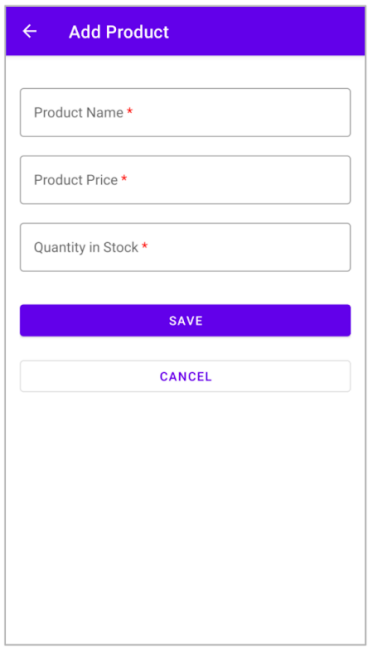
- Kod startowy aplikacji **Inventory** .
- Komputer z zainstalowanym Android Studio.

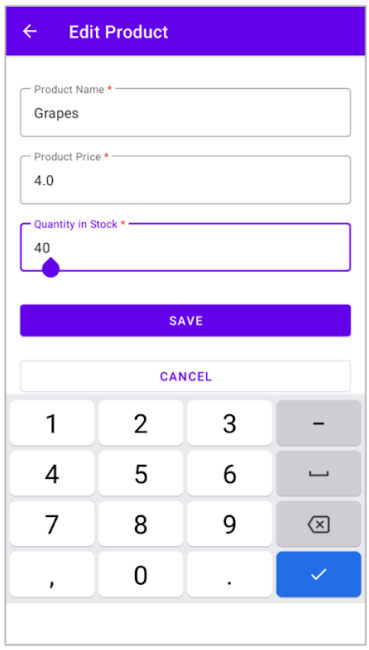
2. Przegląd aplikacji

W tym ćwiczeniu z programowania będziesz pracować z aplikacją startową o nazwie Inventory app i dodasz do niej warstwę bazy danych za pomocą biblioteki Room. Ostateczna wersja aplikacji wyświetla listę elementów z bazy danych spisu za pomocą `RecyclerView`. Użytkownik będzie miał możliwość dodania nowego elementu, zaktualizowania istniejącego elementu i usunięcia elementu z bazy danych inwentarza (funkcjonalność aplikacji uzupełnisz w następnym ćwiczeniu z programowania).

Poniżej zrzuty ekranu z ostatecznej wersji aplikacji.







Uwaga : powyższe zrzuty ekranu pochodzą z ostatecznej wersji aplikacji na końcu ścieżki, a nie na końcu tego ćwiczenia z programowania. Te zrzuty ekranu znajdują się tutaj, aby dać wyobrażenie o ostatecznej wersji aplikacji.

3. Przegląd aplikacji startowej

Pobierz kod startowy do tego ćwiczenia z programowania

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 znany z poprzednich ćwiczeń z programowania, a także kod, który jest Ci nieznany, o którym dowiesz się w późniejszych ćwiczeniach z programowania.

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

Adres URL kodu startowego:

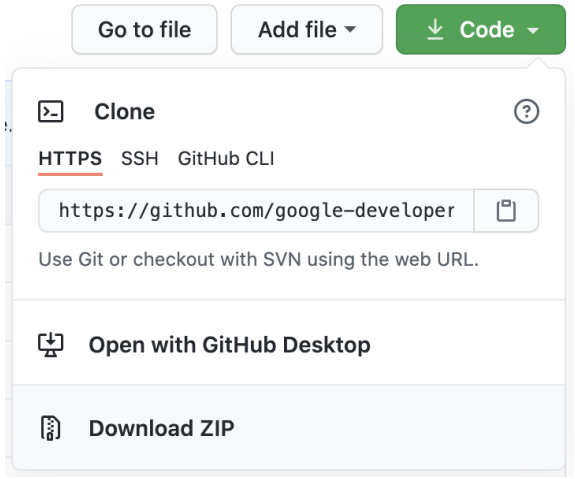
https://github.com/google-developer-training/android-basics-kotlin-inventory-app/tree/starter

Nazwa oddziału z kodem startowym: `starter`

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

Pobierz kod

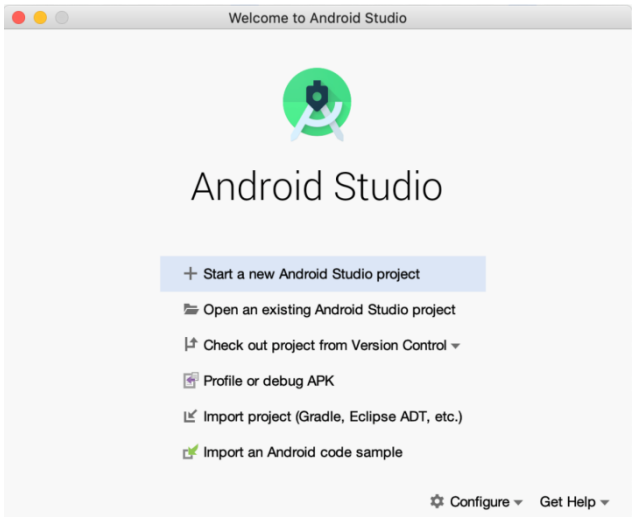
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



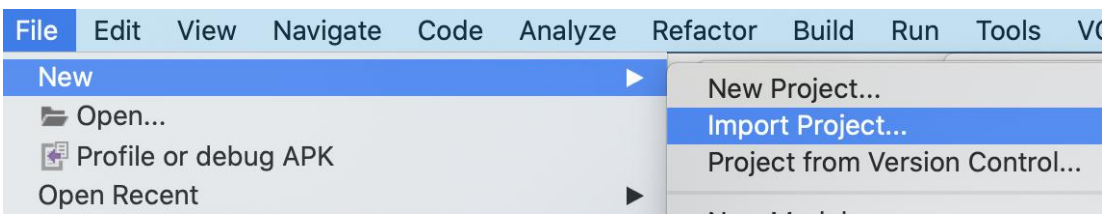
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

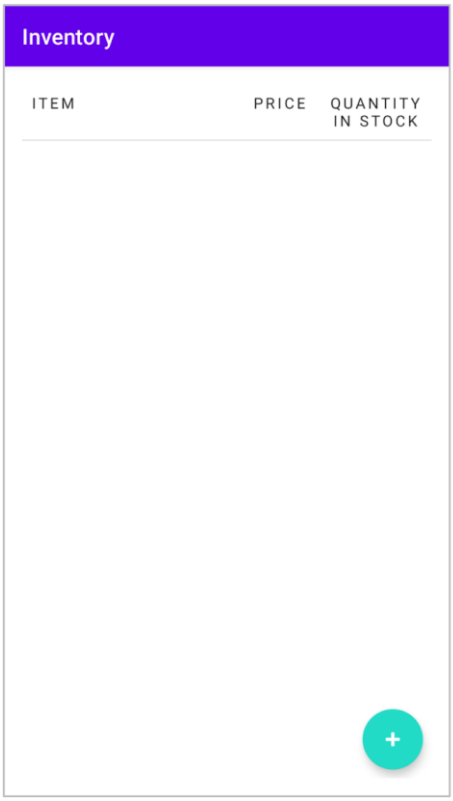


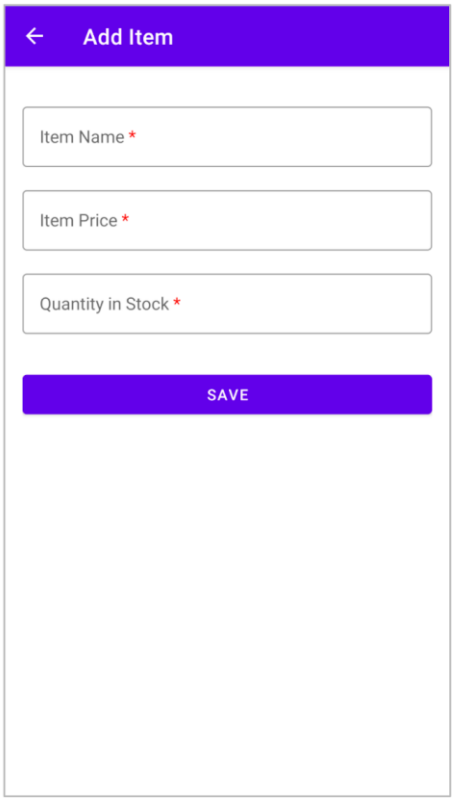
3. W oknie dialogowym **Importuj projekt** przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).

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.

Przegląd kodu startowego

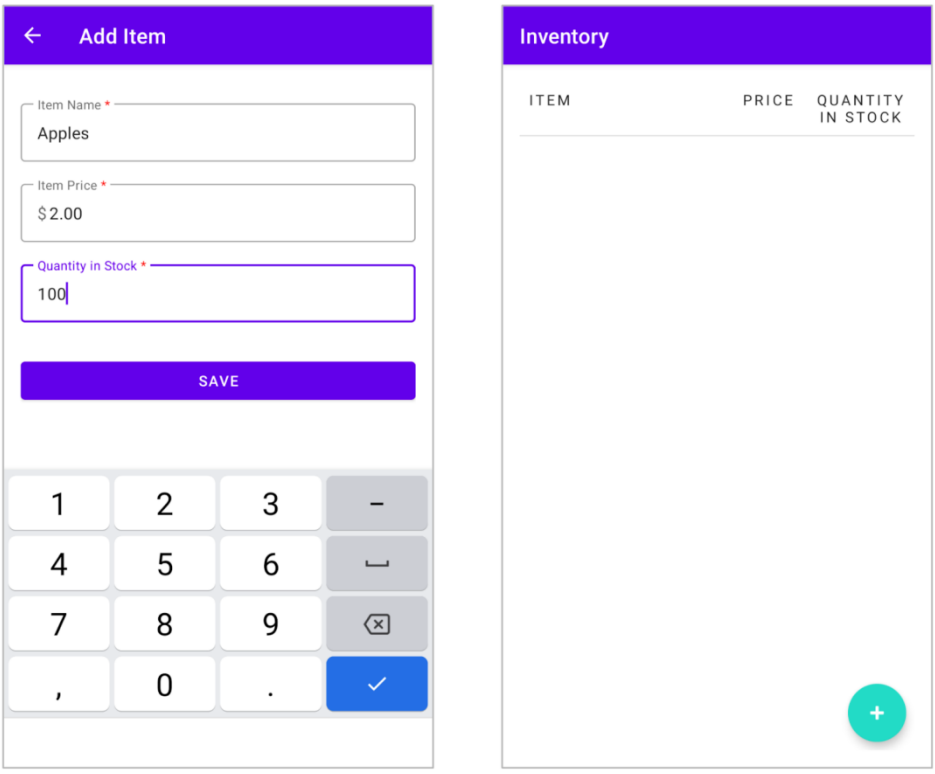
1. Otwórz projekt za pomocą kodu startowego w Android Studio.
2. Uruchom aplikację na urządzeniu z systemem Android lub w emulatorze. Upewnij się, że emulator lub podłączone urządzenie działa na poziomie API 26 lub wyższym. Database Inspector działa najlepiej na emulatorze/urządzeniach z interfejsem API 26.
3. Aplikacja nie pokazuje danych inwentaryzacyjnych. Zwróć uwagę, że FAB dodaje nowe pozycje do bazy danych.
4. Kliknij na FAB. Aplikacja przechodzi do nowego ekranu, na którym możesz wprowadzić szczegóły nowego elementu.





Problemy z kodem startowym

1. Na ekranie **Dodaj element** wprowadź szczegóły elementu. Kliknij **Zapisz** . Fragment dodawania elementu nie jest zamknięty. Przejdź wstecz za pomocą systemowego klawisza wstecz. Nowy przedmiot nie jest zapisywany i nie jest wyświetlany na ekranie ekwipunku. Zauważ, że aplikacja jest niekompletna, a funkcja przycisku **Zapisz** nie jest zaimplementowana.



W tym ćwiczeniu z programowania dodasz część bazy danych aplikacji, która zapisuje szczegóły inwentarza w bazie danych SQLite. Będziesz używać biblioteki trwałości pokoju do interakcji z bazą danych SQLite.

Opis kodu

Pobrany kod startowy ma wstępnie zaprojektowane układy ekranu. W tej ścieżce skupisz się na implementacji logiki bazy danych. Oto krótki opis niektórych plików na początek.

główna_aktywność.xml

Główna aktywność, która obsługuje wszystkie pozostałe fragmenty w aplikacji. Metoda `onCreate()`pobiera `NavController`z `NavHostFragment`i konfiguruje pasek akcji do użytku z `NavController`.

item_list_fragment.xml

Pierwszy ekran pokazany w aplikacji. Zawiera głównie RecyclerView i FAB. RecyclerView zostanie zaimplementowany w dalszej części ścieżki.

fragment_add_item.xml

Ten układ zawiera pola tekstowe do wprowadzania szczegółów nowej pozycji magazynowej, która ma zostać dodana.

ElementListFragment.kt

Ten fragment zawiera głównie standardowy kod. W `onViewCreated()`metodzie click listener jest ustawiony na FAB, aby przejść do fragmentu dodawania elementu.

AddItemFragment.kt

Ten fragment służy do dodawania nowych pozycji do bazy danych. Funkcja `onCreateView()`inicjuje zmienną wiążącą i `onDestroyView()`ukrywa klawiaturę przed zniszczeniem fragmentu.

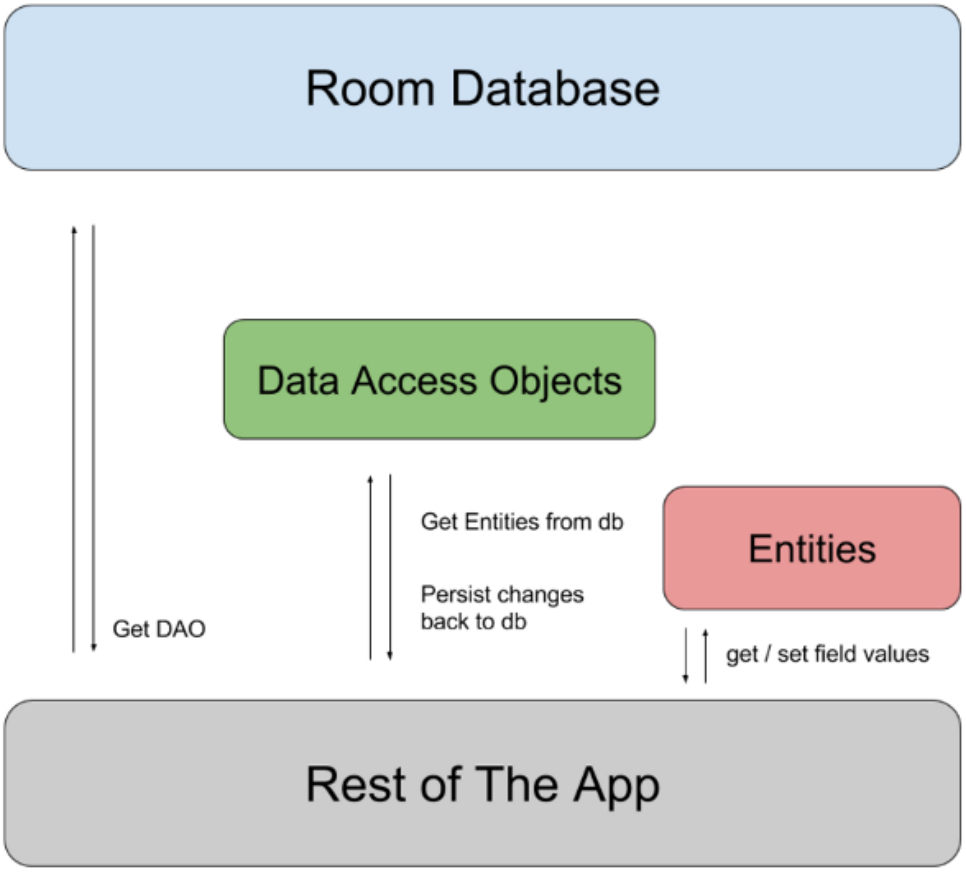
4. Główne elementy pokoju

Kotlin zapewnia łatwy sposób radzenia sobie z danymi poprzez wprowadzenie klas danych. Dostęp do tych danych i ewentualnie ich modyfikacja odbywa się za pomocą wywołań funkcji. Jednak w świecie baz danych potrzebne *są tabele* i *zapytania* , aby uzyskać dostęp do danych i je modyfikować. Poniższe składniki programu Room sprawiają, że te procesy robocze są bezproblemowe.

Pokój składa się z trzech głównych elementów:

- Jednostki danych reprezentują tabele w bazie danych Twojej aplikacji. Służą do aktualizacji danych przechowywanych w wierszach w tabelach oraz do tworzenia nowych wierszy do wstawienia.
- Obiekty dostępu do danych (DAO) zapewniają metody, których aplikacja używa do pobierania, aktualizowania, wstawiania i usuwania danych w bazie danych.
- Klasa bazy danych przechowuje bazę danych i jest głównym punktem dostępu dla bazowego połączenia z bazą danych aplikacji. Klasa bazy danych udostępnia aplikacji instancje obiektów DAO powiązanych z tą bazą danych.

Zaimplementujesz i dowiesz się więcej o tych komponentach w dalszej części ćwiczenia z programowania. Poniższy diagram pokazuje, w jaki sposób komponenty Pokoju współpracują ze sobą w celu interakcji z bazą danych.



Dodaj biblioteki pokoi

W tym zadaniu dodasz wymagane biblioteki komponentów pomieszczenia do swoich plików Gradle.

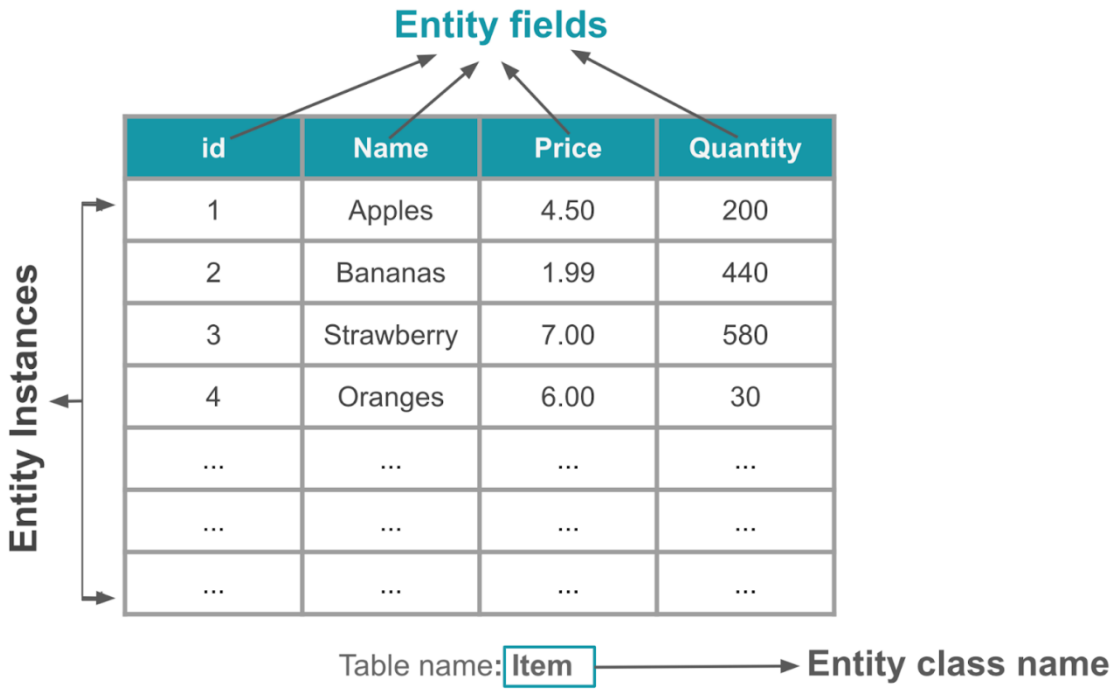
1. Otwórz plik gradientu na poziomie modułu. `build.gradle (Module: InventoryApp.app)`W `dependencies`bloku dodaj następujące zależności dla biblioteki Room.

```
// Room
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
implementation "androidx.room:room-ktx:$room_version"
```

Uwaga : w przypadku zależności bibliotecznych w pliku Gradle zawsze używaj najnowszych numerów wersji stabilnych ze strony wydań AndroidX .

5. Utwórz element Jednostka

Klasa Entity definiuje tabelę, a każde wystąpienie tej klasy reprezentuje wiersz w tabeli bazy danych. Klasa jednostki ma mapowania, aby poinformować Room, w jaki sposób zamierza prezentować i wchodzić w interakcje z informacjami w bazie danych. W Twojej aplikacji encja będzie przechowywać informacje o pozycjach magazynowych, takie jak nazwa pozycji, cena pozycji i dostępne zapasy.

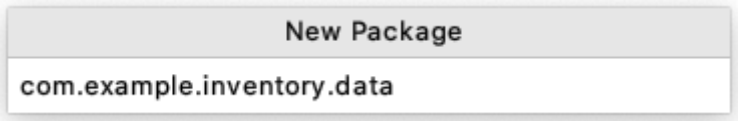


`@Entity`adnotacja oznacza klasę jako klasę Entity bazy danych. Dla każdej klasy Entity tworzona jest tabela bazy danych do przechowywania elementów. Każde pole Entity jest reprezentowane jako kolumna w bazie danych, chyba że zaznaczono inaczej (szczegóły w dokumentacji Entity). Każde wystąpienie jednostki przechowywane w bazie danych musi mieć klucz podstawowy. Klucz podstawowy jest używany do jednoznacznej identyfikacji każdego rekordu/wpisu w tabelach bazy danych. Raz przypisany klucz podstawowy nie może być modyfikowany, reprezentuje obiekt jednostki, o ile istnieje w bazie danych.

W tym zadaniu utworzysz klasę Entity. Zdefiniuj pola do przechowywania następujących informacji o zapasach dla każdego towaru.

- An `Int`do przechowywania klucza podstawowego.
- A `String`do przechowywania nazwy przedmiotu.
- A `double`do przechowywania ceny przedmiotu.
- `Int`Przechowywanie ilości w magazynie .

1. Otwórz kod startowy w Android Studio.
2. Utwórz pakiet o nazwie `data`pod `com.example.inventory`pakietem podstawowym.



3. Wewnątrz `data`pakietu utwórz klasę Kotlin o nazwie `Item`. Ta klasa będzie reprezentować jednostkę bazy danych w Twojej aplikacji. W kolejnym kroku dodasz odpowiednie pola do przechowywania informacji o stanie magazynowym.
4. Zaktualizuj `Item`definicję klasy za pomocą następującego kodu. Zadeklaruj `id`type `Int`, `itemName`type `String`, `itemPrice`type `Double`i `quantityInStock`type `Int` jako parametry dla konstruktora podstawowego. Przypisz domyślną wartość `0`do `id`. Będzie to klucz podstawowy, identyfikator, który jednoznacznie identyfikuje każdy rekord/wpis w Twojej `Item`tabeli.

```
class Item(
  val id: Int = 0,
  val itemName: String,
  val itemPrice: Double,
  val quantityInStock: Int
)
```

Odświeżanie na głównym konstruktorze : Podstawowy konstruktor jest częścią nagłówka klasy w klasie Kotlin: idzie po nazwie klasy (i opcjonalnych parametrach typu).

Klasy danych

Klasy danych służą przede wszystkim do przechowywania danych w Kotlinie. Są one oznaczone słowem kluczowym `data`. Obiekty klasy danych Kotlin mają dodatkowe zalety, kompilator automatycznie generuje narzędzia do porównywania, drukowania i kopiowania, takie jak `toString()`, `copy()`i `equals()`.

Przykład:

```
// Example data class with 2 properties.
data class User(val first_name: String, val last_name: String){
}
```

Aby zapewnić spójność i sensowne zachowanie generowanego kodu, klasy danych muszą spełniać następujące wymagania:

- Konstruktor podstawowy musi mieć co najmniej jeden parametr.
- Wszystkie podstawowe parametry konstruktora muszą być oznaczone jako `val`lub `var`.
- Klasy danych nie mogą być `abstract`, `open`ani .`sealedinner`

Ostrzeżenie : kompilator używa tylko właściwości zdefiniowanych w konstruktorze głównym dla funkcji generowanych automatycznie. Właściwości zadeklarowane w treści klasy są wykluczone z generowanych implementacji.

Aby dowiedzieć się więcej o klasach danych, zapoznaj się z dokumentacją .

5. Konwertuj `Item`klasę na klasę danych, poprzedzając jej definicję klasy `data`słowem kluczowym.

```
data class Item(
  val id: Int = 0,
  val itemName: String,
  val itemPrice: Double,
  val quantityInStock: Int
)
```

6. Nad `Item`deklaracją klasy dodaj do klasy danych adnotację `@Entity`. Użyj `tableName`argumentu, aby podać `item`jako nazwę tabeli SQLite.

```
@Entity(tableName = "item")
data class Item(
  ...
```

)

Ważne : po wyświetleniu monitu przez Android Studio zaimportuj Entityi wszystkie inne adnotacje do pokoju (których będziesz używać później w ćwiczeniu z programowania) z androidxbiblioteki. Na przykład,androidx.room.Entity.

Uwaga : @Entityadnotacja ma kilka możliwych argumentów. Domyślnie (bez argumentów do @Entity) nazwa tabeli będzie taka sama jak klasa. Argument tableNamepozwala podać inną lub bardziej pomocną nazwę tabeli. Ten argument tableNamejest opcjonalny, ale wysoce zalecany. Dla uproszczenia podasz taką samą nazwę jak nazwa klasy, czyli item. Istnieje kilka innych argumentów @Entity, które możesz zbadać w dokumentacji .

7. Aby zidentyfikować idklucz jako klucz podstawowy, dodaj do idwłaściwości adnotację @PrimaryKey. Ustaw parametr autoGeneratena truetak, aby Roomgenerował identyfikator dla każdej jednostki. Gwarantuje to, że identyfikator dla każdego przedmiotu jest unikalny.

```
@Entity(tableName = "item")
data class Item(
  @PrimaryKey(autoGenerate = true)
  val id: Int = 0,
  ...
)
```

8. Opisz pozostałe właściwości za pomocą @ColumnInfo. Adnotacja ColumnInfosłuży do dostosowania kolumny powiązanej z konkretnym polem. Na przykład, używając nameargumentu, możesz określić inną nazwę kolumny dla pola zamiast nazwy zmiennej. Dostosuj nazwy właściwości za pomocą parametrów, jak pokazano poniżej. To podejście jest podobne do używania tableNamew celu określenia innej nazwy bazy danych.

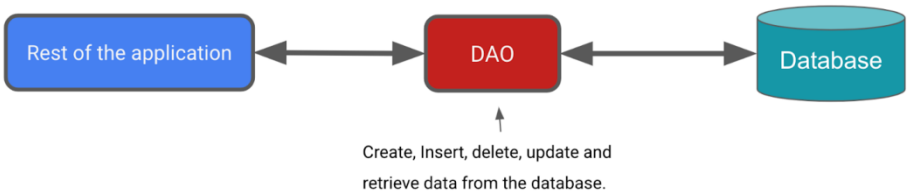
```
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity
data class Item(
  @PrimaryKey(autoGenerate = true)
  val id: Int = 0,
  @ColumnInfo(name = "name")
  val itemName: String,
  @ColumnInfo(name = "price")
  val itemPrice: Double,
  @ColumnInfo(name = "quantity")
  val quantityInStock: Int
)
```

6. Utwórz element DAO

Obiekt dostępu do danych (DAO)

Obiekt dostępu do danych (DAO) to wzorzec używany do oddzielenia warstwy trwałości od reszty aplikacji poprzez zapewnienie abstrakcyjnego interfejsu. Ta izolacja jest zgodna z zasadą pojedynczej odpowiedzialności , którą widzieliście w poprzednich ćwiczeniach z programowania.

Funkcjonalność DAO polega na ukryciu przed resztą aplikacji wszystkich złożoności związanych z wykonywaniem operacji na bazie danych w podstawowej warstwie trwałości. Pozwala to na zmianę warstwy dostępu do danych niezależnie od kodu, który wykorzystuje dane.



W tym zadaniu definiujesz obiekt dostępu do danych (DAO) dla pokoju. Obiekty dostępu do danych są głównymi komponentami Room, które są odpowiedzialne za zdefiniowanie interfejsu, który uzyskuje dostęp do bazy danych.

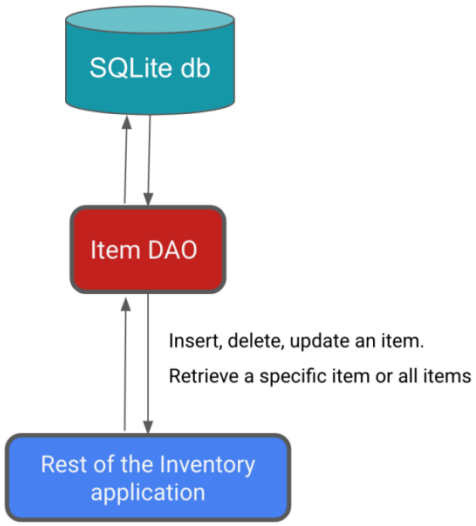
Utworzone DAO będzie niestandardowym interfejsem zapewniającym wygodne metody wykonywania zapytań/pobierania, wstawiania, usuwania i aktualizowania bazy danych. Room wygeneruje implementację tej klasy w czasie kompilacji.

W przypadku typowych operacji na bazach danych `Room`biblioteka zapewnia wygodne adnotacje, takie jak `@Insert`, `@Delete`i `@Update`. Do wszystkiego innego jest `@Query`adnotacja. Możesz napisać dowolne zapytanie obsługiwane przez SQLite.

Jako dodatkowy bonus, gdy piszesz zapytania w Android Studio, kompilator sprawdza twoje zapytania SQL pod kątem błędów składniowych.

W przypadku aplikacji do inwentaryzacji musisz mieć możliwość wykonania następujących czynności:

- **Wstaw** lub dodaj nowy element.
- **Zaktualizuj** istniejący przedmiot, aby zaktualizować nazwę, cenę i ilość.
- **Uzyskaj** określony przedmiot na podstawie jego klucza podstawowego, `id`.
- **Zdobądź wszystkie przedmioty** , aby móc je wyświetlić.
- **Usuń** wpis w bazie danych.



Teraz zaimplementuj element DAO w swojej aplikacji:

1. W `data`pakiecie utwórz klasę Kotlin `ItemDao.kt`.
2. Zmień definicję klasy na `interface`i opisz za pomocą `@Dao`.

```
@Dao
interface ItemDao {
}
```

3. W treści interfejsu dodaj `@Insert`adnotację. Poniżej `@Insert`dodaj funkcję, która jako argument `insert()`przyjmuje instancję `Entity`klasy . `item`Wykonywanie operacji na bazie danych może zająć dużo czasu, dlatego powinny być wykonywane w osobnym wątku. Ustaw funkcję jako funkcję zawieszenia, aby ta funkcja mogła być wywołana ze współprogramu.

```
@Insert
suspend fun insert(item: Item)
```

4. Dodaj argument `OnConflict`i przypisz mu wartość `OnConflictStrategy.IGNORE`. Argument `OnConflict`mówi Pokoju, co ma zrobić w przypadku konfliktu. Strategia `OnConflictStrategy.IGNORE`ignoruje nowy element, jeśli jego klucz podstawowy znajduje się już w bazie danych. Aby dowiedzieć się więcej o dostępnych strategiach konfliktów, zapoznaj się z dokumentacją .

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun insert(item: Item)
```

Teraz `Room`wygeneruje cały niezbędny kod do wstawienia `item`do bazy danych. Kiedy dzwonisz `insert()`z kodu Kotlin, `Room`wykonuje zapytanie SQL, aby wstawić jednostkę do bazy danych. (Uwaga: funkcja może mieć dowolną nazwę; nie musi być wywoływana `insert()`.)

5. Dodaj `@Update`adnotację z `update()`funkcją dla jednego `item`. Zaktualizowana jednostka ma ten sam klucz, co przekazana jednostka. Możesz zaktualizować niektóre lub wszystkie inne właściwości jednostki. Podobnie jak w przypadku `insert()`metody, wykonaj następującą `update()`metodę `suspend`.

```
@Update
suspend fun update(item: Item)
```

6. Dodaj @Deleteadnotację z delete()funkcją usuwania elementów. Uczyń z tego metodę zawieszenia. Adnotacja @Deleteusuwa jeden element lub listę elementów. (Uwaga: musisz przekazać encje do usunięcia, jeśli nie masz encji, być może będziesz musiał ją pobrać przed wywołaniem delete()funkcji.)

   ```
   @Delete
   suspend fun delete(item: Item)
   ```

   Nie ma adnotacji dla wygody pozostałych funkcji, więc musisz użyć @Queryadnotacji i dostarczyć zapytania SQLite.

7. Napisz zapytanie SQLite, aby pobrać określony element z tabeli elementów na podstawie podanego id. Następnie dodasz adnotację do pokoju i użyjesz zmodyfikowanej wersji następującego zapytania w późniejszych krokach. W kolejnych krokach zmienisz to również w metodę DAO za pomocą Room.
8. Wybierz wszystkie kolumny zitem
9. WHEREpasuje do idokreślonej wartości.

   Przykład:

   ```
   SELECT * from item WHERE id = 1
   ```

8. Zmień powyższe zapytanie SQL, aby było używane z adnotacją Room i argumentem. Dodaj @Queryadnotację, podaj zapytanie jako parametr ciągu do @Queryadnotacji. Dodaj Stringparametr do @Querytego zapytania SQLite, aby pobrać element z tabeli elementów.
9. Wybierz wszystkie kolumny zitem
10. WHEREidpasuje do argumentu id:. Zwróć uwagę na :id. Używasz notacji z dwukropkiem w zapytaniu, aby odwoływać się do argumentów w funkcji.

    ```
    @Query("SELECT * from item WHERE id = :id")
    ```

9. Poniżej @Queryadnotacji dodaj getItem()funkcję, która pobiera Intargument i zwraca Flow<Item>.

   ```
   @Query("SELECT * from item WHERE id = :id")
   fun getItem(id: Int): Flow<Item>
   ```

   Użycie Flowlub LiveDatajako zwracanego typu zapewni, że będziesz otrzymywać powiadomienia o każdej zmianie danych w bazie danych. Zaleca się stosować Floww warstwie utrwalającej. Aktualizuje Roomto Flowza Ciebie, co oznacza, że wystarczy jednorazowo pobrać dane tylko raz. Jest to pomocne przy aktualizacji spisu inwentarza, który zaimplementujesz podczas następnego ćwiczenia z programowania. Ze względu na Flowtyp zwracany Room uruchamia również zapytanie w wątku w tle. Nie musisz jawnie tworzyć z niej suspendfunkcji i wywoływać wewnątrz wspólnego zakresu.

   Może być konieczne zaimportowanie Flowz kotlinx.coroutines.flow.Flow.

10. Dodaj a @Queryz getItems()funkcją:
11. Niech zapytanie SQLite zwróci wszystkie kolumny z itemtabeli w kolejności rosnącej.
12. Zwróć getItems()listę Itempodmiotów jako Flow. Roomaktualizuje to Flowza Ciebie, co oznacza, że wystarczy jednorazowo uzyskać dane tylko raz.

```
@Query("SELECT * from item ORDER BY name ASC")
fun getItems(): Flow<List<Item>>
```

11. Chociaż nie zobaczysz żadnych widocznych zmian, uruchom aplikację, aby upewnić się, że nie zawiera błędów.

7. Utwórz instancję bazy danych

W tym zadaniu tworzysz obiekt RoomDatabase, który korzysta z Entityobiektów i obiektów DAO utworzonych w poprzednim zadaniu. Klasa bazy danych definiuje listę encji i obiektów dostępu do danych. Jest to również główny punkt dostępu dla podstawowego połączenia.
Klasa Databaseudostępnia aplikacji instancje zdefiniowanych obiektów DAO. Z kolei aplikacja może używać obiektów DAO do pobierania danych z bazy danych jako wystąpień skojarzonych obiektów encji danych. Aplikacja może również używać zdefiniowanych jednostek danych do aktualizowania wierszy z odpowiednich tabel lub do tworzenia nowych wierszy do wstawienia.

Musisz utworzyć RoomDatabaseklasę abstrakcyjną z adnotacją @Database. Ta klasa ma jedną metodę, która albo tworzy wystąpienie, RoomDatabasejeśli nie istnieje, albo zwraca istniejące wystąpienie RoomDatabase.

Oto ogólny proces pobierania RoomDatabaseinstancji:

- Utwórz public abstractklasę, która rozszerza RoomDatabase. Nowa zdefiniowana klasa abstrakcyjna działa jako posiadacz bazy danych. Zdefiniowana klasa jest abstrakcyjna, ponieważ Roomtworzy implementację za Ciebie.
- Dodaj do klasy adnotację @Database. W argumentach wymień jednostki dla bazy danych i ustaw numer wersji.
- Zdefiniuj abstrakcyjną metodę lub właściwość, która zwraca ItemDaoInstancję, a Roomwygeneruje dla Ciebie implementację.
- Potrzebujesz tylko jednej instancji RoomDatabasedla całej aplikacji, więc stwórz RoomDatabasesingletona.
- Użyj Room's, Room.databaseBuilderaby utworzyć item_databasebazę danych ( ) tylko wtedy, gdy ona nie istnieje. W przeciwnym razie zwróć istniejącą bazę danych.

Wskazówka: Poniższy kod może służyć jako szablon dla przyszłych projektów. Sposób tworzenia RoomDatabaseinstancji jest podobny do procesu zdefiniowanego powyżej. Może być konieczne zastąpienie encji i danych Dao specyficznych dla Twojej aplikacji.

Utwórz bazę danych

1. W datapakiecie utwórz klasę Kotlin ItemRoomDatabase.kt.
2. W ItemRoomDatabase.ktpliku stwórz ItemRoomDatabaseklasę jako abstractklasę rozszerzającą RoomDatabase. Dodaj do klasy adnotację @Database. W następnym kroku naprawisz błąd brakujących parametrów.

```
@Database
abstract class ItemRoomDatabase : RoomDatabase() {}
```

3. Adnotacja @Databasewymaga kilku argumentów, dzięki czemu Roommożna zbudować bazę danych.

- Określ Itemjako jedyną klasę z listą entities.
- Ustaw versionjako 1**.** Za każdym razem, gdy zmienisz schemat tabeli bazy danych, będziesz musiał zwiększyć numer wersji.
- Ustaw exportSchemana false, aby nie przechowywać kopii zapasowych historii wersji schematu.

  @Database(entities = [Item::class], version = 1, exportSchema = false)

4. Baza danych musi wiedzieć o DAO. Wewnątrz ciała klasy zadeklaruj abstrakcyjną funkcję, która zwraca ItemDao. Możesz mieć wiele DAO.

   abstract fun itemDao(): ItemDao

5. Poniżej funkcji abstrakcyjnej zdefiniuj companionobiekt. Obiekt towarzyszący umożliwia dostęp do metod tworzenia lub pobierania bazy danych przy użyciu nazwy klasy jako kwalifikatora.

   companion object {}

6. Wewnątrz companionobiektu zadeklaruj prywatną zmienną dopuszczającą wartość null INSTANCEdla bazy danych i zainicjuj ją do null. Zmienna INSTANCEzachowa odniesienie do bazy danych, gdy została utworzona. Pomaga to w utrzymaniu pojedynczej instancji bazy danych otwartej w określonym czasie, co jest kosztownym zasobem w tworzeniu i utrzymaniu.

   Dodaj komentarz INSTANCEza pomocą @Volatile. Wartość zmiennej ulotnej nigdy nie będzie buforowana, a wszystkie zapisy i odczyty będą wykonywane do iz pamięci głównej. Pomaga to upewnić się, że wartość INSTANCEjest zawsze aktualna i taka sama dla wszystkich wątków wykonania. Oznacza to, że zmiany wprowadzone przez jeden wątek INSTANCEsą natychmiast widoczne dla wszystkich innych wątków.

   @Volatile
   private var INSTANCE: ItemRoomDatabase? = null

7. Poniżej INSTANCE, pozostając wewnątrz companionobiektu, zdefiniuj getDatabase()metodę z Contextparametrem, którego będzie potrzebował konstruktor bazy danych. Zwróć typ ItemRoomDatabase. Zobaczysz błąd, ponieważ getDatabase()jeszcze niczego nie zwraca.

   fun getDatabase(context: Context): ItemRoomDatabase {}

8. Wiele wątków może potencjalnie wpaść w sytuację wyścigu i jednocześnie poprosić o instancję bazy danych, co spowoduje powstanie dwóch baz danych zamiast jednej. Zawijanie kodu w celu pobrania bazy danych do synchronizedbloku oznacza, że tylko jeden wątek wykonania na raz może wprowadzić ten blok kodu, co zapewnia, że baza danych zostanie zainicjowana tylko raz.

   Inside getDatabase(), zwróć INSTANCEzmienną lub jeśli INSTANCEjest null, zainicjuj ją wewnątrz synchronized{}bloku. Użyj do tego operatora elvis(?:). Przekaż thisobiekt towarzyszący, który chcesz zablokować wewnątrz bloku funkcyjnego. Naprawisz błąd w późniejszych krokach.

   return INSTANCE ?: synchronized(this) { }

9. Wewnątrz zsynchronizowanego bloku utwórz valzmienną instancji i użyj konstruktora bazy danych, aby uzyskać bazę danych. Nadal będziesz mieć błędy, które naprawisz w kolejnych krokach.

```
val instance = Room.databaseBuilder()
```

10. Na końcu `synchronized`bloku wróć `instance`.

```
return instance
```

11. Wewnątrz `synchronized`bloku zainicjuj `instance`zmienną i użyj konstruktora bazy danych, aby uzyskać bazę danych. Przekaż kontekst aplikacji, klasę bazy danych i nazwę bazy danych `item_database`do `Room.databaseBuilder()`.

```
val instance = Room.databaseBuilder(
  context.applicationContext,
  ItemRoomDatabase::class.java,
  "item_database"
)
```

Android Studio wygeneruje błąd niezgodności typów. Aby usunąć ten błąd, musisz dodać strategię migracji oraz `build()`w poniższych krokach.

12. Dodaj wymaganą strategię migracji do konstruktora. Użyj `.fallbackToDestructiveMigration()`.

Zwykle należałoby dostarczyć obiekt migracji ze strategią migracji na wypadek zmiany schematu. Obiekt *migracji* to obiekt, który definiuje sposób pobierania wszystkich wierszy ze starego schematu i konwertowania ich na wiersze w nowym schemacie, dzięki czemu żadne dane nie zostaną utracone. Migracja wykracza poza zakres tego ćwiczenia z programowania. Prostym rozwiązaniem jest zniszczenie i odbudowa bazy danych, co oznacza utratę danych.

```
.fallbackToDestructiveMigration()
```

13. Aby utworzyć instancję bazy danych, wywołaj `.build()`. Powinno to usunąć błędy Android Studio.

```
.build()
```

14. Wewnątrz `synchronized`bloku przypisz `INSTANCE = instance`.

```
INSTANCE = instance
```

15. Na końcu `synchronized`bloku wróć `instance`. Twój ostateczny kod powinien wyglądać tak:

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Item::class], version = 1, exportSchema = false)
abstract class ItemRoomDatabase : RoomDatabase() {

  abstract fun itemDao(): ItemDao

  companion object {
```

```
        @Volatile
        private var INSTANCE: ItemRoomDatabase? = null
        fun getDatabase(context: Context): ItemRoomDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    ItemRoomDatabase::class.java,
                    "item_database"
                )
                    .fallbackToDestructiveMigration()
                    .build()
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

16. Zbuduj swój kod, aby upewnić się, że nie ma błędów.

Implementuj klasę aplikacji

W tym zadaniu utworzysz instancję bazy danych w klasie Application.

1. Otwórz `InventoryApplication.kt`, utwórz `val`wywołany `database`typ `ItemRoomDatabase`. Utwórz `database`wystąpienie, wywołując przekazywanie `getDatabase()`w `ItemRoomDatabase`kontekście. Użyj `lazy` delegata, aby wystąpienie `database`było leniwie tworzone, gdy po raz pierwszy potrzebujesz/uzyskasz dostęp do odwołania (a nie podczas uruchamiania aplikacji). Spowoduje to utworzenie bazy danych (fizycznej bazy danych na dysku) przy pierwszym dostępie.

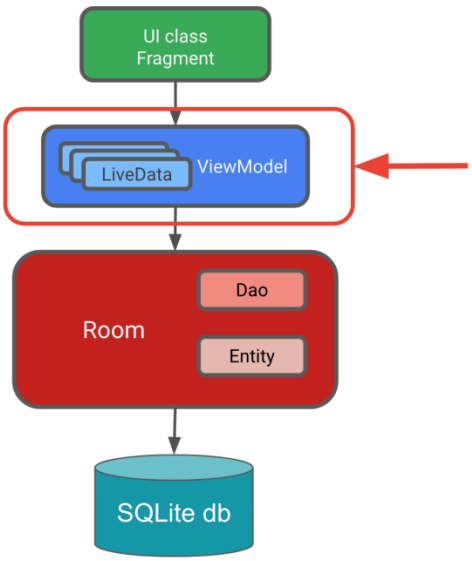
```
import android.app.Application
import com.example.inventory.data.ItemRoomDatabase

class InventoryApplication : Application(){
    val database: ItemRoomDatabase by lazy { ItemRoomDatabase.getDatabase(this) }
}
```

Użyjesz tego `database`wystąpienia później w ćwiczeniu programowania podczas tworzenia wystąpienia ViewModel.

Masz teraz wszystkie klocki do pracy ze swoim pokojem. Ten kod kompiluje się i działa, ale nie masz możliwości sprawdzenia, czy rzeczywiście działa. To dobry moment, aby dodać nowy przedmiot do bazy danych zapasów, aby przetestować bazę danych. Aby to osiągnąć, musisz `ViewModel`rozmawiać z bazą danych.

8. Dodaj ViewModel

Do tej pory stworzyłeś bazę danych, a klasy interfejsu użytkownika były częścią kodu startowego. Aby zapisać tymczasowe dane aplikacji, a także uzyskać dostęp do bazy danych, potrzebujesz ViewModel. Twój model widoku Inventory będzie komunikował się z bazą danych za pośrednictwem DAO i dostarczy dane do interfejsu użytkownika. Wszystkie operacje na bazie danych będą musiały zostać usunięte z głównego wątku interfejsu użytkownika, zrobisz to za pomocą współprogramów i viewModelScope.



Utwórz model widoku zapasów

1. W com.example.inventorypakiecie utwórz plik klasy Kotlin InventoryViewModel.kt.
2. Rozszerz InventoryViewModelklasę z ViewModelklasy. Przekaż ItemDaoobiekt jako parametr do domyślnego konstruktora.

```
class InventoryViewModel(private val itemDao: ItemDao) : ViewModel() {}
```

3. Na końcu InventoryViewModel.ktpliku poza klasą dodaj InventoryViewModelFactoryklasę, aby utworzyć InventoryViewModelinstancję. Przekaż ten sam parametr konstruktora, co InventoryViewModelinstancja ItemDao. Rozszerz klasę z ViewModelProvider.Factoryklasy. W następnym kroku naprawisz błąd dotyczący niewdrożonych metod.

```
class InventoryViewModelFactory(private val itemDao: ItemDao) : ViewModelProvider.Factory {
}
```

4. Kliknij czerwoną żarówkę i wybierz **Zaimplementuj elementy członkowskie** lub możesz zastąpić create()metodę wewnątrz ViewModelProvider.Factoryklasy w następujący sposób, która przyjmuje dowolny typ klasy jako argument i zwraca ViewModelobiekt.

```
override fun <T : ViewModel?> create(modelClass: Class<T>): T {
  TODO("Not yet implemented")
}
```

5. Implementuj create()metodę. Sprawdź, czy modelClassjest taka sama jak InventoryViewModelklasa i zwróć jej instancję. W przeciwnym razie zrzuć wyjątek.

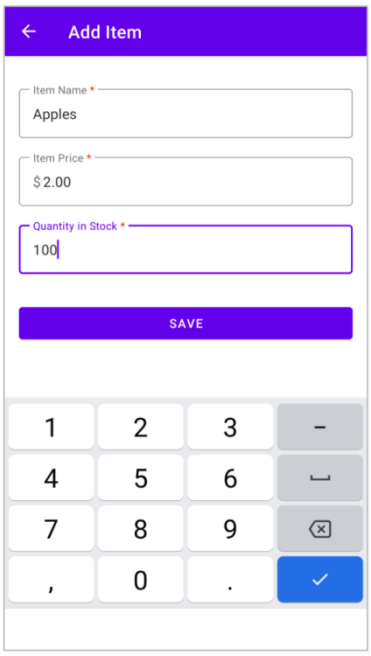
```
if (modelClass.isAssignableFrom(InventoryViewModel::class.java)) {
  @Suppress("UNCHECKED_CAST")
  return InventoryViewModel(itemDao) as T
}
throw IllegalArgumentException("Unknown ViewModel class")
```

Wskazówka: Utworzenie fabryki ViewModel to głównie standardowy kod, więc możesz ponownie użyć tego kodu w przyszłych fabrykach ViewModel.

Wypełnij ViewModel

W tym zadaniu wypełnisz `InventoryViewModel`klasę, aby dodać dane inwentaryzacyjne do bazy danych. Obserwuj `Item`encję i ekran **Dodaj pozycję** w aplikacji Inventory.

```
@Entity
data class Item(
  @PrimaryKey(autoGenerate = true)
  val id: Int = 0,
  @ColumnInfo(name = "name")
  val itemName: String,
  @ColumnInfo(name = "price")
  val itemPrice: Double,
  @ColumnInfo(name = "quantity")
  val quantityInStock: Int
)
```



Potrzebujesz nazwy, ceny i zapasów dla tej konkretnej pozycji, aby dodać podmiot do bazy danych. W dalszej części ćwiczenia z programowania użyjesz ekranu **Dodaj element** , aby uzyskać te szczegóły od użytkownika. W bieżącym zadaniu używasz trzech ciągów jako danych

wejściowych do ViewModel, konwertujesz je na Iteminstancję encji i zapisujesz w bazie danych przy użyciu ItemDaoinstancji. Czas na wdrożenie.

1. W InventoryViewModelklasie dodaj privatefunkcję o nazwie insertItem(), która pobiera Itemobiekt i dodaje dane do bazy danych w sposób nieblokujący.

```
private fun insertItem(item: Item) {
}
```

2. Aby wchodzić w interakcję z bazą danych poza głównym wątkiem, uruchom współprogram i wywołaj w nim metodę DAO. Wewnątrz insertItem()metody użyj , viewModelScope.launchaby rozpocząć współprogram w ViewModelScope. Wewnątrz funkcji uruchamiania wywołaj funkcję suspend insert()przy itemDaoprzekazywaniu item. Jest ViewModelScope to właściwość rozszerzenia ViewModelklasy, która automatycznie anuluje współprogramy podrzędne po ViewModelzniszczeniu.

```
private fun insertItem(item: Item) {
  viewModelScope.launch {
     itemDao.insert(item)
  }
}
```

Importkotlinx.coroutines.launch, androidx.lifecycle.*viewModelScope*

com.example.inventory.data.Item, jeśli nie są importowane automatycznie.

Uwaga : podczas importowania ćwiczeń z programowania com.example.inventory.data.Itemdla Itemencji na żądanie Android Studio.

3. W InventoryViewModelklasie dodaj kolejną prywatną funkcję, która pobiera trzy ciągi i zwraca Iteminstancję.

```
private fun getNewItemEntry(itemName: String, itemPrice: String, itemCount: String): Item {
  return Item(
     itemName = itemName,
     itemPrice = itemPrice.toDouble(),
     quantityInStock = itemCount.toInt()
  )
}
```

4. Nadal wewnątrz InventoryViewModelklasy dodaj wywołaną funkcję publiczną, addNewItem()która pobiera trzy ciągi znaków dla szczegółów elementu. Przekaż szczegółowe ciągi elementów do getNewItemEntry()funkcji i przypisz zwróconą wartość do zmiennej o nazwie newItem. Zadzwoń do insertItem()przekazania, newItemaby dodać nową jednostkę do bazy danych. Zostanie to wywołane z fragmentu interfejsu użytkownika, aby dodać szczegóły pozycji do bazy danych.

```
fun addNewItem(itemName: String, itemPrice: String, itemCount: String) {
  val newItem = getNewItemEntry(itemName, itemPrice, itemCount)
  insertItem(newItem)
}
```

Zauważ, że nie użyłeś `viewModelScope.launch`for `addNewItem()`, ale jest to potrzebne powyżej w `insertItem()`przypadku wywoływania metody DAO. Powodem jest to, że *funkcje suspend mogą być wywoływane tylko ze współprogramu lub innej funkcji zawieszenia* . Ta funkcja `itemDao.insert(item)`jest funkcją zawieszenia.

Dodałeś wszystkie wymagane funkcje, aby dodać encje do bazy danych. W następnym zadaniu zaktualizujesz fragment **Dodaj element** , aby korzystać z powyższych funkcji.

9. Zaktualizuj AddItemFragment

1. W `AddItemFragment.kt`, na początku `AddItemFragment`klasy tworzymy `private val`wywołanie `viewModel`typu `InventoryViewModel`. Użyj `by activityViewModels()`delegata właściwości Kotlin, aby udostępnić `ViewModel`wszystkie fragmenty. Naprawisz błąd w następnym kroku.

```
private val viewModel: InventoryViewModel by activityViewModels {
}
```

2. Wewnątrz lambdy wywołaj `InventoryViewModelFactory()`konstruktor i przekaż `ItemDao`instancję. Użyj `database`instancji utworzonej w jednym z poprzednich zadań, aby wywołać `itemDao`konstruktora.

```
private val viewModel: InventoryViewModel by activityViewModels {
  InventoryViewModelFactory(
    (activity?.application as InventoryApplication).database
      .itemDao()
  )
}
```

Wskazówka: jest to w większości standardowy kod, więc możesz ponownie użyć kodu w przyszłości, aby utworzyć instancję ViewModel przy użyciu fabryki ViewModel.

3. Poniżej `viewModel`definicji utwórz `lateinit var`wywołanie `item`typu `Item`.

```
lateinit var item: Item
```

4. Ekran **Dodaj element** zawiera trzy pola tekstowe, w których użytkownik otrzymuje szczegółowe informacje o elemencie. W tym kroku dodasz funkcję sprawdzającą, czy tekst w polach TextFields nie jest pusty. Użyjesz tej funkcji, aby zweryfikować dane wprowadzone przez użytkownika przed dodaniem lub aktualizacją jednostki w bazie danych. Tę weryfikację należy przeprowadzić we fragmencie, `ViewModel`a nie we fragmencie. W `InventoryViewModel`klasie dodaj następującą `public`funkcję o nazwie `isEntryValid()`.

```
fun isEntryValid(itemName: String, itemPrice: String, itemCount: String): Boolean {
  if (itemName.isBlank() || itemPrice.isBlank() || itemCount.isBlank()) {
    return false
  }
  return true
}
```

5. W AddItemFragment.kt, poniżej onCreateView()funkcji utwórz privatefunkcję o nazwie isEntryValid(), która zwraca Boolean. W następnym kroku naprawisz błąd brakującej wartości zwracanej.

```
private fun isEntryValid(): Boolean {
}
```

6. W AddItemFragmentklasie zaimplementuj isEntryValid()funkcję. Wywołaj isEntryValid()funkcję na viewModelinstancji, przekazując tekst z widoków tekstowych. Zwróć wartość viewModel.isEntryValid()funkcji.

```
private fun isEntryValid(): Boolean {
  return viewModel.isEntryValid(
      binding.itemName.text.toString(),
      binding.itemPrice.text.toString(),
      binding.itemCount.text.toString()
  )
}
```

7. W AddItemFragmentklasie poniżej isEntryValid()funkcji dodaj kolejną privatefunkcję wywoływaną addNewItem()bez parametrów i nic nie zwracaj. isEntryValid()Wewnątrz funkcji wywołaj ifwarunek.

```
private fun addNewItem() {
  if (isEntryValid()) {
  }
}
```

8. Wewnątrz ifbloku wywołaj addNewItem()metodę w viewModelinstancji. Podaj szczegóły pozycji wprowadzone przez użytkownika, użyj bindinginstancji do ich odczytania.

```
if (isEntryValid()) {
  viewModel.addNewItem(
  binding.itemName.text.toString(),
  binding.itemPrice.text.toString(),
  binding.itemCount.text.toString(),
  )
}
```

9. Poniżej ifbloku utwórz a, val actionaby przejść z powrotem do ItemListFragment. Zadzwoń , przejeżdżając .*findNavController*().navigate()action

```
val action = AddItemFragmentDirections.actionAddItemFragmentToItemListFragment()
findNavController().navigate(action)
```

Importandroidx.navigation.fragment.findNavController.

10. Kompletna metoda powinna wyglądać następująco.

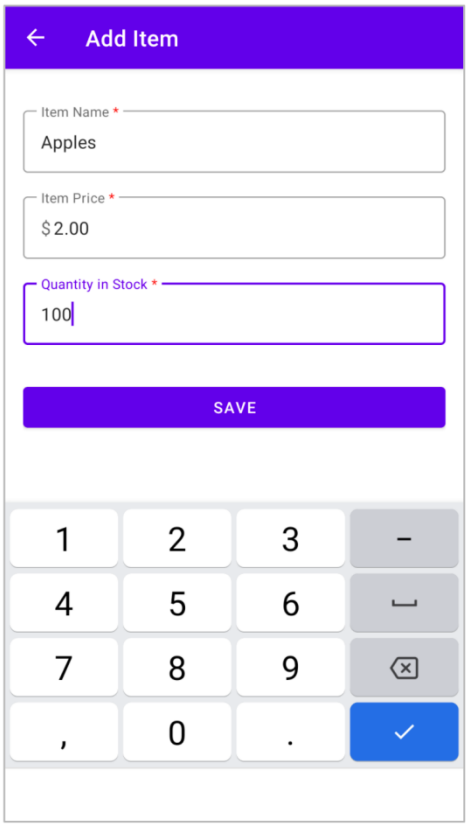
```
private fun addNewItem() {
    if (isEntryValid()) {
```

```
            viewModel.addNewItem(
                binding.itemName.text.toString(),
                binding.itemPrice.text.toString(),
                binding.itemCount.text.toString(),
            )
            val action = AddItemFragmentDirections.actionAddItemFragmentToItemListFragment()
            findNavController().navigate(action)
        }
    }
```

11. Aby powiązać wszystko razem, dodaj obsługę kliknięć do przycisku **Zapisz** . W `AddItemFragment`klasie, nad `onDestroyView()`funkcją, nadpisz `onViewCreated()`funkcję.
12. Wewnątrz `onViewCreated()`funkcji dodaj obsługę kliknięcia do przycisku zapisywania i wywołaj `addNewItem()`z niego.

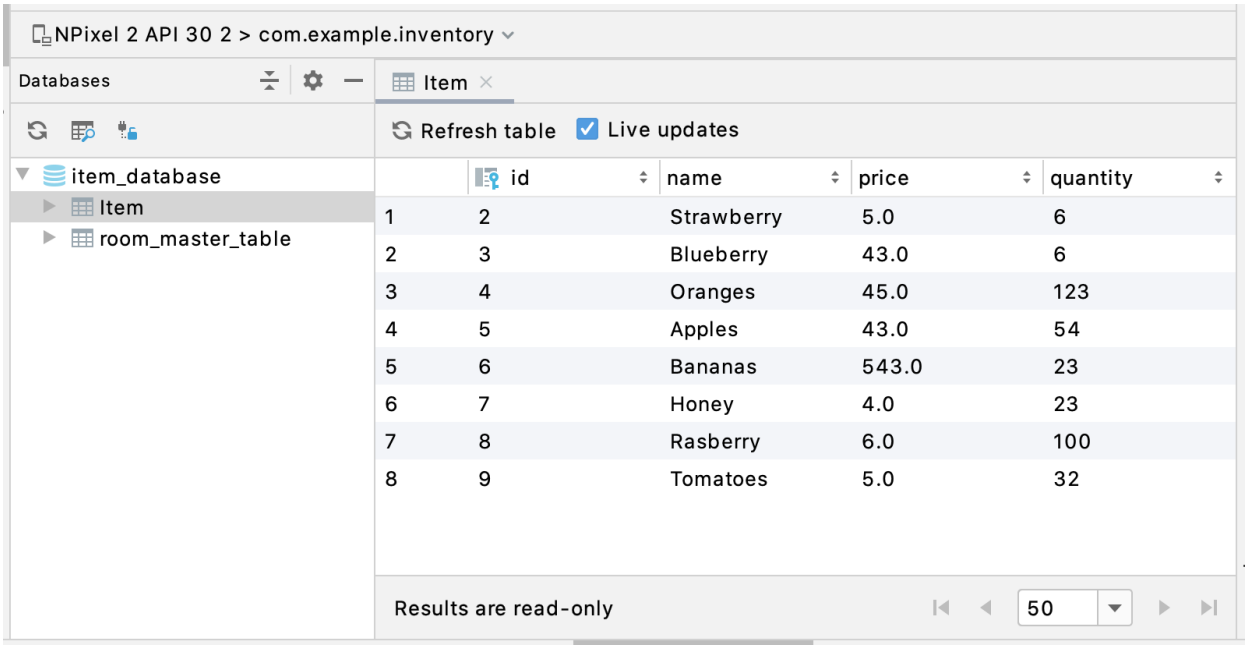
```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  binding.saveAction.setOnClickListener {
     addNewItem()
  }
}
```

13. Zbuduj i uruchom swoją aplikację. Dotknij **+** Fab. Na ekranie **Dodaj element** dodaj szczegóły elementu i dotknij **Zapisz** . Ta akcja zapisuje dane, ale nie widzisz jeszcze niczego w aplikacji. W następnym zadaniu użyjesz Inspektora baz danych , aby wyświetlić zapisane dane.



Wyświetl bazę danych za pomocą Database Inspector

1. Uruchom aplikację na emulatorze lub podłączonym urządzeniu z interfejsem API 26 lub wyższym, jeśli jeszcze tego nie zrobiłeś. Database Inspector działa najlepiej na emulatorze/urządzeniach z interfejsem API 26.
2. W Android studio wybierz **Widok** > **Okna narzędziowe** > **Inspektor bazy danych** z paska menu.
3. W okienku Database Inspector wybierz `com.example.inventory`z menu rozwijanego.
4. Baza **danych item_database** w aplikacji Inventory pojawi się w okienku **Databases .** Rozwiń węzeł dla bazy danych **item_database** i wybierz opcję **Element** do sprawdzenia. Jeśli okienko **Bazy danych** jest puste, użyj emulatora, aby dodać niektóre elementy do bazy danych za pomocą ekranu **Dodaj element .**
5. Zaznacz pole wyboru **Aktualizacje na żywo** w Inspektorze bazy danych, aby automatycznie aktualizować dane, które prezentuje podczas interakcji z uruchomioną aplikacją w emulatorze lub urządzeniu.



Gratulacje! Utworzyłeś aplikację, która może utrwalać dane za pomocą usługi Room . W następnym laboratorium programowania dodasz a `RecyclerView`do swojej aplikacji, aby wyświetlić elementy w bazie danych i dodać nowe funkcje do aplikacji, takie jak usuwanie i aktualizowanie encji. Do zobaczenia tam!

10. Kod rozwiązania

Kod rozwiązania dla tego ćwiczenia programowania znajduje się w repozytorium GitHub i gałęzi pokazanej poniżej.

Adres URL kodu rozwiązania:

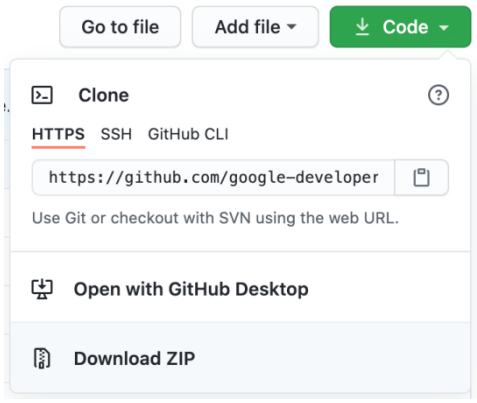
https://github.com/google-developer-training/android-basics-kotlin-inventory-app/tree/room

Oddział: pokój

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

Pobierz kod

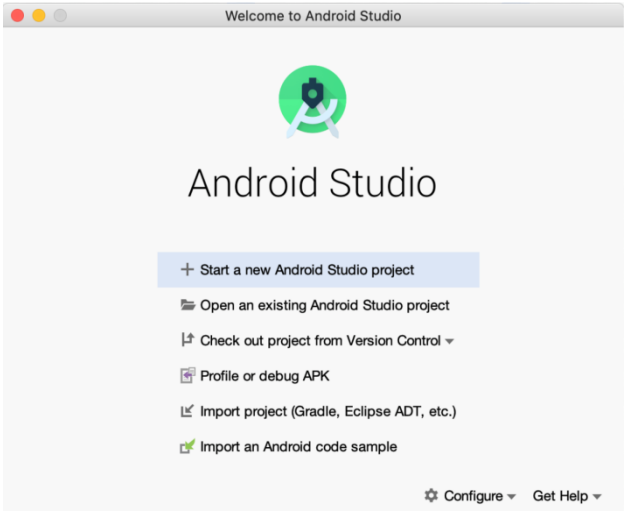
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



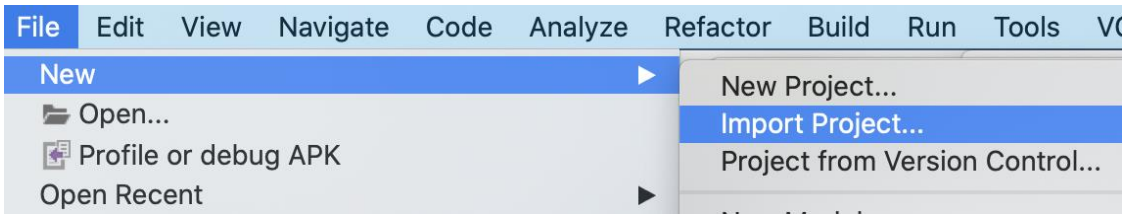
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

11. Podsumowanie

- Zdefiniuj swoje tabele jako klasy danych z adnotacją `@Entity`. Zdefiniuj właściwości opisane `@ColumnInfo`jako kolumny w tabelach.
- Zdefiniuj obiekt dostępu do danych (DAO) jako interfejs z adnotacją `@Dao`. DAO mapuje funkcje Kotlina na zapytania do bazy danych.
- Użyj adnotacji, aby zdefiniować funkcje `@Insert`, `@Delete`i `@Update`.
- Użyj `@Query`adnotacji z ciągiem zapytania SQLite jako parametru dla innych zapytań.
- Użyj Database Inspector , aby wyświetlić dane zapisane w bazie danych Android SQLite.

12. Dowiedz się więcej

Dokumentacja programisty Androida

- Zapisz dane w lokalnej bazie danych za pomocą Room
- androidx.pokój
- Debuguj swoją bazę danych za pomocą Inspektora baz danych

Posty na blogu

- 7 porad profesjonalistów dotyczących pokoju
- Jedyny przedmiot. Słownictwo Kotlina

Filmy

- Kotlin: Korzystanie z API Room Kotlin
- Inspektor baz danych

Inna dokumentacja i artykuły

- Jednotonowy wzór
- Obiekty towarzyszące
- Samouczek SQLite — łatwy sposób na szybkie opanowanie SQLite

Czytaj i aktualizuj dane za pomocą Room

1. Zanim zaczniesz

W poprzednich ćwiczeniach z programowania nauczyłeś się korzystać z biblioteki trwałości pokoju, warstwy abstrakcji na bazie bazy danych SQLite do przechowywania danych aplikacji. Podczas tego ćwiczenia z programowania dodasz więcej funkcji do aplikacji Inventory i nauczysz się odczytywać, wyświetlać, aktualizować i usuwać dane z bazy danych SQLite za pomocą Room. Użyjesz a, `RecyclerView`aby wyświetlić dane z bazy danych i automatycznie zaktualizować dane, gdy zmienią się podstawowe dane w bazie danych.

Warunki wstępne

- Wiesz, jak tworzyć i współdziałać z bazą danych SQLite za pomocą biblioteki Room.
- Wiesz, jak tworzyć klasy encji, DAO i bazy danych.
- Wiesz, jak używać obiektu dostępu do danych (DAO) do mapowania funkcji Kotlina na zapytania SQL.
- Wiesz, jak wyświetlić elementy listy w `RecyclerView`.
- Wykonałeś poprzednie ćwiczenia z programowania w tej jednostce, Trwałe dane z pokojem

Czego się nauczysz

- Jak czytać i wyświetlać encje z bazy danych SQLite.
- Jak aktualizować i usuwać encje z bazy danych SQLite za pomocą biblioteki Room.

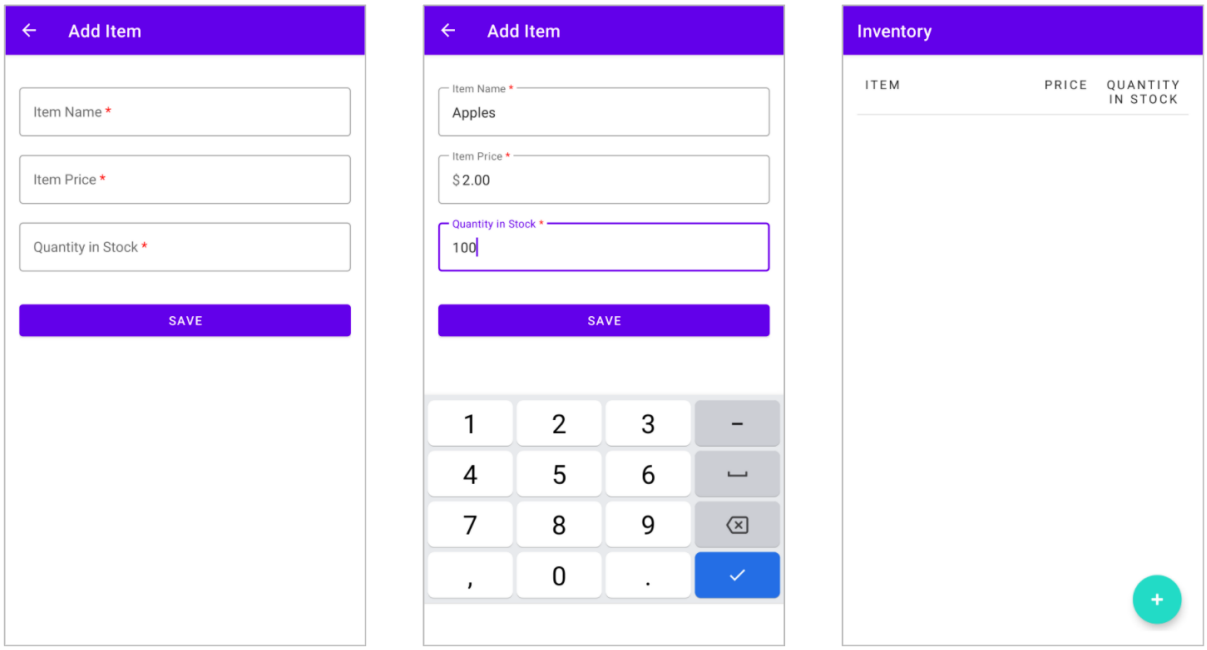
Co zbudujesz

- Zbudujesz aplikację Inventory, która wyświetli listę pozycji magazynowych. Aplikacja może aktualizować, edytować i usuwać elementy z bazy danych aplikacji za pomocą pokoju.

2. Przegląd aplikacji startowej

To laboratorium programowania używa kodu rozwiązania aplikacji Inventory z poprzedniego ćwiczenia z programowania jako kodu startowego. Aplikacja startowa już zapisuje dane przy użyciu biblioteki trwałości pomieszczenia . Użytkownik może dodawać dane do bazy danych aplikacji za pomocą ekranu **Dodaj element .**

Uwaga: aktualna wersja aplikacji startowej nie wyświetla daty zapisanej w bazie danych.



W tym ćwiczeniu z programowania rozszerzysz aplikację o odczytywanie i wyświetlanie danych, aktualizowanie i usuwanie encji w bazie danych za pomocą biblioteki pokoi.

Pobierz kod startowy do tego ćwiczenia z programowania

Ten kod startowy jest taki sam jak kod rozwiązania z poprzedniego ćwiczenia z programowania.

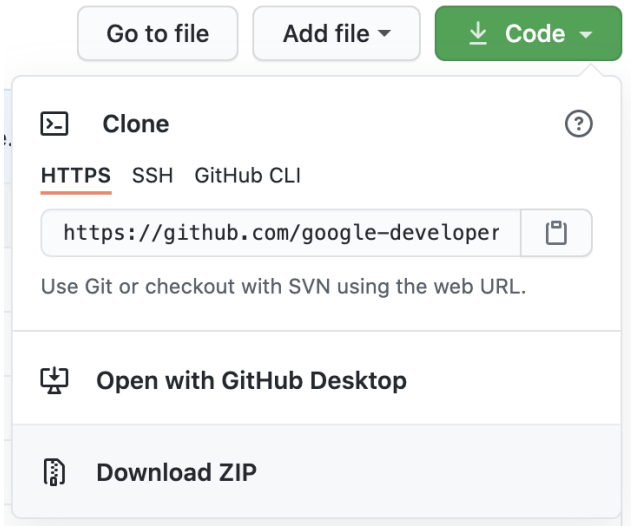
URL kodu startowego: https://github.com/google-developer-training/android-basics-kotlin-inventory-app/tree/room

Nazwa filii:room

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

Pobierz kod

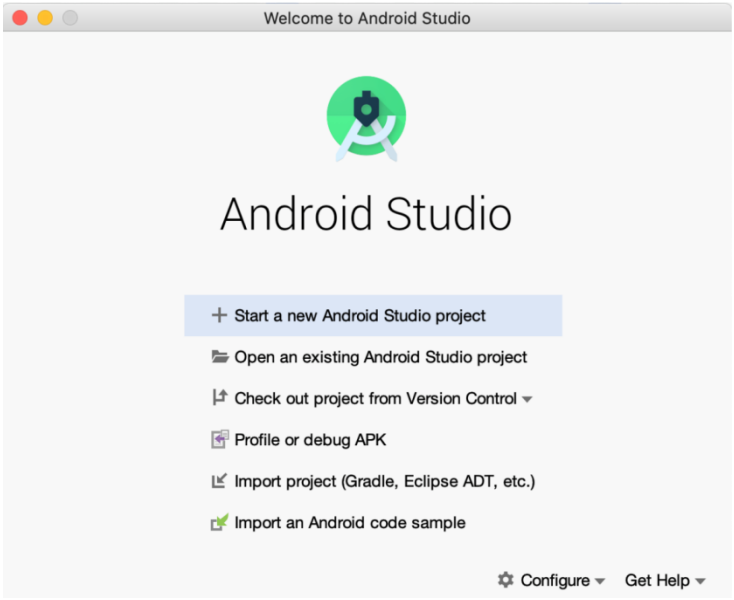
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



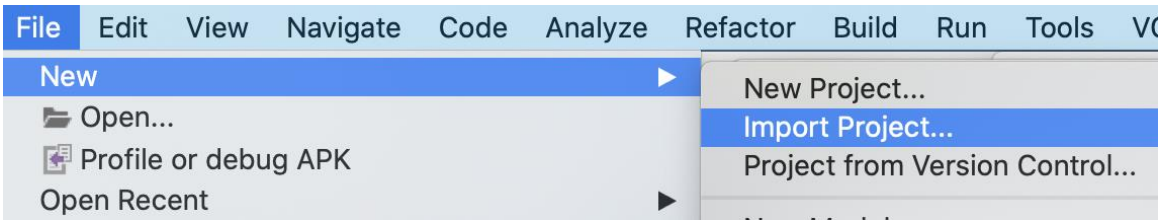
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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



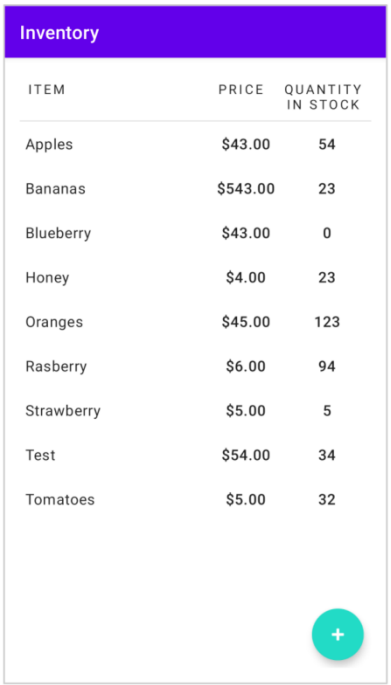
3. W oknie dialogowym **Importuj projekt** przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.
6. Kliknij przycisk **Uruchom** , aby skompilować i uruchomić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

3. Dodaj widok Recycler

W tym zadaniu dodasz a `RecyclerView` do aplikacji, aby wyświetlić dane przechowywane w bazie danych.

Dodaj funkcję pomocniczą do sformatowania ceny

Poniżej znajduje się zrzut ekranu z finalnej aplikacji.



Zauważ, że cena jest wyświetlana w formacie waluty. Aby przekonwertować podwójną wartość na żądany format waluty, dodasz do `Item`klasy funkcję rozszerzenia.

Funkcje rozszerzeń

Kotlin daje możliwość rozszerzenia klasy o nową funkcjonalność bez konieczności dziedziczenia po klasie lub modyfikowania istniejącej definicji klasy. Oznacza to, że możesz dodawać funkcje do istniejącej klasy bez konieczności uzyskiwania dostępu do jej kodu źródłowego. Odbywa się to poprzez specjalne deklaracje zwane *rozszerzeniami* .

Na przykład możesz napisać nowe funkcje dla klasy z biblioteki innej firmy, której nie możesz modyfikować. Takie funkcje są dostępne do wywołania w zwykły sposób, tak jakby były metodami oryginalnej klasy. Funkcje te nazywane są *funkcjami rozszerzającymi* . (Istnieją również *właściwości rozszerzenia,* które umożliwiają definiowanie nowych właściwości dla istniejących klas, ale są one poza zakresem tego ćwiczenia z programowania).

Funkcje rozszerzające w rzeczywistości nie modyfikują klasy, ale pozwalają na użycie notacji kropkowej podczas wywoływania funkcji na obiektach tej klasy.

Na przykład w poniższym fragmencie kodu masz klasę o nazwie `Square`. Ta klasa ma właściwość boku i funkcję obliczania pola kwadratu. Zwróć uwagę na `Square.perimeter()`funkcję rozszerzenia, nazwa funkcji jest poprzedzona klasą, na której działa. Wewnątrz funkcji możesz odwoływać się do publicznych właściwości `Square`klasy.

Obserwuj użycie funkcji rozszerzenia w `main()`funkcji. Utworzona funkcja rozszerzenia, `perimeter()`, jest wywoływana jako zwykła funkcja wewnątrz tej `Square`klasy.

Przykład:

```
class Square(val side: Double){
      fun area(): Double{
      return side * side;
   }
}

// Extension function to calculate the perimeter of the square
fun Square.perimeter(): Double{
      return 4 * side;
}

// Usage
fun main(args: Array<String>){
    val square = Square(5.5);
    val perimeterValue = square.perimeter()
    println("Perimeter: $perimeterValue")
    val areaValue = square.area()
    println("Area: $areaValue")
}
```

W tym kroku sformatujesz cenę towaru jako ciąg formatu waluty. Ogólnie rzecz biorąc, nie chcesz zmieniać klasy encji, która reprezentuje dane, tylko w celu sformatowania danych (patrz zasada pojedynczej odpowiedzialności), więc zamiast tego dodasz funkcję rozszerzenia.

1. W Item.kt, poniżej definicji klasy, dodaj funkcję rozszerzenia o nazwie Item.getFormattedPrice(), która nie przyjmuje parametrów i zwraca String. Zwróć uwagę na nazwę klasy i notację kropkową w nazwie funkcji.

   ```
   fun Item.getFormattedPrice(): String =
     NumberFormat.getCurrencyInstance().format(itemPrice)
   ```

   Importuj java.text.NumberFormatpo wyświetleniu monitu przez Android Studio.

Dodaj ListAdapter

W tym kroku dodasz adapter listy do RecyclerView. Ponieważ jesteś zaznajomiony z implementacją adaptera z poprzednich ćwiczeń z programowania, instrukcje zostały podsumowane poniżej. Ukończony ItemListAdapterplik znajduje się na końcu tego kroku dla Twojej wygody i aby pomóc w lepszym zrozumieniu koncepcji pomieszczenia w ćwiczeniu z programowania.

1. W com.example.inventorypakiecie dodaj klasę Kotlin o nazwie ItemListAdapter. Przekaż funkcję wywoływaną onItemClicked()jako parametr konstruktora, która przyjmuje Itemobiekt jako parametr.
2. Zmień ItemListAdapterpodpis klasy na rozszerzenie ListAdapter. Przekaż Itemi ItemListAdapter.ItemViewHolderjako parametry.
3. Dodaj parametr konstruktora DiffCallback; ListAdapterużyją tego, aby dowiedzieć się, co zmieniło się na liście .
4. Zastąp wymagane metody onCreateViewHolder()i onBindViewHolder().
5. Metoda onCreateViewHolder()zwraca nowy ViewHolder, gdy RecyclerView go potrzebuje.
6. Wewnątrz onCreateViewHolder()metody utwórz nowy View, napełnij go z item_list_item.xmlpliku układu za pomocą automatycznie generowanej klasy wiążącej, ItemListItemBinding.
7. Implementuj onBindViewHolder()metodę. Pobierz aktualną pozycję za pomocą metody getItem(), przekazując pozycję.
8. Ustaw odbiornik kliknięcia na itemView, wywołaj funkcję onItemClicked()wewnątrz odbiornika.
9. Zdefiniuj ItemViewHolderklasę, rozszerz ją z RecyclerView.ViewHolder.Override the bind()function, przekaż Itemobiekt.
10. Zdefiniuj obiekt towarzyszący. Wewnątrz obiektu towarzyszącego zdefiniuj a valtypu DiffUtil.ItemCallback<Item>()o nazwie DiffCallback. Zastąp wymagane metody areItemsTheSame()i areContentsTheSame()i zdefiniuj je.

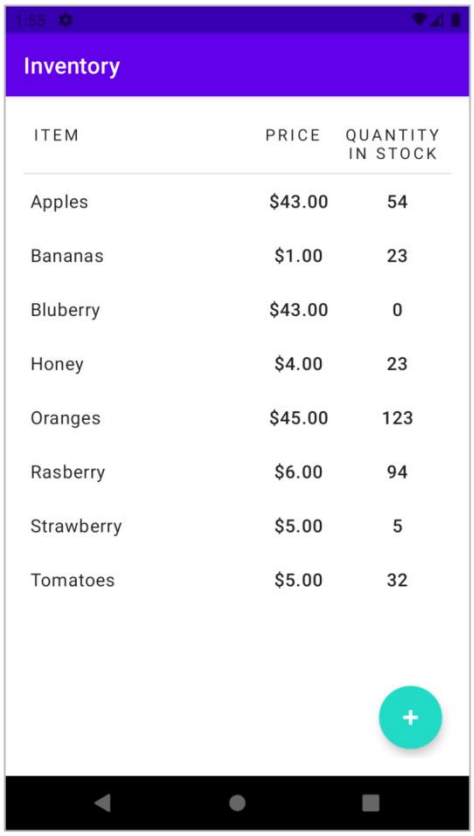
    Gotowa klasa powinna wyglądać następująco:

    ```
    package com.example.inventory

    import android.view.LayoutInflater
    import android.view.ViewGroup
    import androidx.recyclerview.widget.DiffUtil
    import androidx.recyclerview.widget.ListAdapter
    import androidx.recyclerview.widget.RecyclerView
    import com.example.inventory.data.Item
    import com.example.inventory.data.getFormattedPrice
    import com.example.inventory.databinding.ItemListItemBinding
    ```

```
/**
* [ListAdapter] implementation for the recyclerview.
*/

class ItemListAdapter(private val onItemClicked: (Item) -> Unit) :
  ListAdapter<Item, ItemListAdapter.ItemViewHolder>(DiffCallback) {

  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {
    return ItemViewHolder(
      ItemListItemBinding.inflate(
        LayoutInflater.from(
          parent.context
        )
      )
    )
  }

  override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    val current = getItem(position)
    holder.itemView.setOnClickListener {
      onItemClicked(current)
    }
    holder.bind(current)
  }

  class ItemViewHolder(private var binding: ItemListItemBinding) :
    RecyclerView.ViewHolder(binding.root) {

    fun bind(item: Item) {

    }
  }

  companion object {
    private val DiffCallback = object : DiffUtil.ItemCallback<Item>() {
      override fun areItemsTheSame(oldItem: Item, newItem: Item): Boolean {
        return oldItem === newItem
      }

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

```
}
```

Obserwuj ekran spisu inwentarza z gotowej aplikacji (aplikacja rozwiązania z końca tego ćwiczenia z programowania). Zwróć uwagę, że każdy element listy wyświetla nazwę pozycji magazynowej, cenę w formacie waluty i aktualny stan magazynowy. W poprzednich krokach użyłeś `item_list_item.xml`pliku układu z trzema TextViews do tworzenia wierszy. W następnym kroku powiążesz szczegóły encji z tymi widokami TextView.



11. W `ItemListAdapter.kt`programie zaimplementuj `bind()`funkcję w `ItemViewHolder`klasie. Powiąż `itemName`TextView z `item.itemName`. Uzyskaj cenę w formacie waluty za pomocą `getFormattedPrice()`funkcji rozszerzenia i powiąż ją z `itemPrice`TextView. Przekonwertuj `quantityInStock`wartość na `String`i powiąż ją z `itemQuantity`TextView. Gotowa metoda powinna wyglądać tak:

```
fun bind(item: Item) {
  binding.apply {
     itemName.text = item.itemName
     itemPrice.text = item.getFormattedPrice()
     itemQuantity.text = item.quantityInStock.toString()
  }
}
```

Po wyświetleniu monitu przez Android Studio, importuj `com.example.inventory.data.getFormattedPrice`.

Użyj ListAdapter

W tym zadaniu zaktualizujesz `InventoryViewModel`i , `ItemListFragment`aby wyświetlić szczegóły pozycji na ekranie za pomocą adaptera listy utworzonego w poprzednim kroku.

1. Na początku klasy `InventoryViewModel`utwórz `val`nazwę `allItems`typu `LiveData<List<Item>>`dla elementów z bazy danych. Nie przejmuj się błędem, wkrótce go naprawisz.

   ```
   val allItems: LiveData<List<Item>>
   ```

   Importuj `androidx.lifecycle.LiveData`po wyświetleniu monitu przez Android Studio.

2. Zadzwoń i przypisz go `getItems()`do . Funkcja zwraca . Aby wykorzystać dane jako wartość, użyj funkcji. Gotowa definicja powinna wyglądać tak:`itemDao``allItems``getItems()``Flow``LiveData``asLiveData()`

   ```
   val allItems: LiveData<List<Item>> = itemDao.getItems().asLiveData()
   ```

   Importuj `androidx.lifecycle.asLiveData`po wyświetleniu monitu przez Android Studio.

3. W `ItemListFragment`, na początku klasy zadeklaruj `private`niezmienną właściwość o nazwie `viewModel`typu `InventoryViewModel`. Użyj `by`delegata, aby przekazać inicjalizację właściwości do `activityViewModels`klasy. Podaj `InventoryViewModelFactory`konstruktora.

   ```
   private val viewModel: InventoryViewModel by activityViewModels {
     InventoryViewModelFactory(
       (activity?.application as InventoryApplication).database.itemDao()
     )
   }
   ```

   Importuj `androidx.fragment.app.activityViewModels`na żądanie Android Studio.

4. Pozostając w obrębie `ItemListFragment`, przewiń do funkcji `onViewCreated()`. Poniżej wywołania `super.onViewCreated()`, zadeklaruj `val`nazwany `adapter`. Zainicjuj nową `adapter`właściwość przy użyciu domyślnego konstruktora, `ItemListAdapter{}`nie przekazując nic.
5. Powiąż nowo utworzony `adapter` z `recyclerView`następującym:

   ```
   val adapter = ItemListAdapter {
   }
   binding.recyclerView.adapter = adapter
   ```

6. Nadal w środku `onViewCreated()`, po ustawieniu adaptera. Dołącz obserwatora `allItems`do nasłuchiwania zmian danych.
7. Wewnątrz obserwatora zadzwoń `submitList()`na `adapter`i przekaż nową listę. Spowoduje to zaktualizowanie RecyclerView o nowe pozycje na liście.

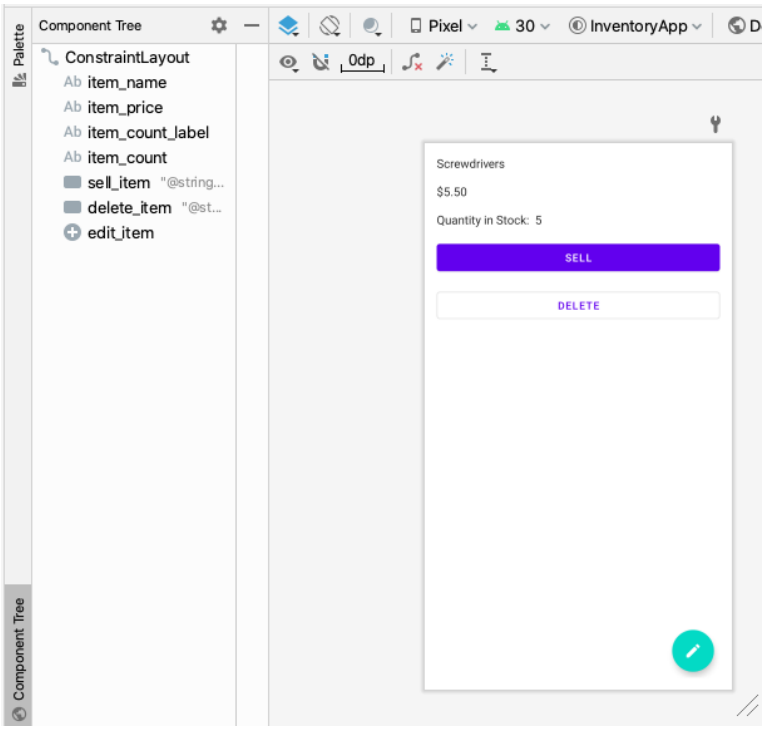
   ```
   viewModel.allItems.observe(this.viewLifecycleOwner) { items ->
     items.let {
       adapter.submitList(it)
     }
   }
   ```

8. Sprawdź, czy ukończona `onViewCreated()`metoda wygląda jak poniżej. Uruchom aplikację. Zwróć uwagę, że lista zapasów jest wyświetlana, jeśli elementy zapisane w bazie danych aplikacji. Dodaj niektóre pozycje zapasów do bazy danych aplikacji, jeśli lista jest pusta.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)

  val adapter = ItemListAdapter {
    }
  binding.recyclerView.adapter = adapter
  viewModel.allItems.observe(this.viewLifecycleOwner) { items ->
    items.let {
      adapter.submitList(it)
    }
  }
  binding.recyclerView.layoutManager = LinearLayoutManager(this.context)
  binding.floatingActionButton.setOnClickListener {
    val action = ItemListFragmentDirections.actionItemListFragmentToAddItemFragment(
      getString(R.string.add_fragment_title)
    )
    this.findNavController().navigate(action)
  }
}
```

4. Wyświetl szczegóły przedmiotu

W tym zadaniu przeczytasz i wyświetlisz szczegóły encji na ekranie **Szczegóły pozycji** . Klucz podstawowy (pozycja `id`) zostanie użyty do odczytania szczegółów, takich jak nazwa, cena i ilość, z bazy danych aplikacji magazynowej i wyświetlenia ich na ekranie **Szczegóły pozycji** przy użyciu `fragment_item_detail.xml`pliku układu. Plik układu `fragment_item_detail.xml`jest wstępnie zaprojektowany dla Ciebie i zawiera trzy TextViews, które wyświetlają szczegóły elementu.



W tym zadaniu zrealizujesz następujące kroki:

- Dodaj procedurę obsługi kliknięć do RecyclerView, aby przejść po aplikacji do ekranu **Szczegóły elementu .**
- We `ItemListFragment`fragmencie pobierz dane z bazy danych i wyświetl.
- Powiąż TextViews z danymi ViewModel.

Dodaj moduł obsługi kliknięć

1. W `ItemListFragment`programie przewiń do `onViewCreated()`funkcji, aby zaktualizować definicję adaptera.
2. Dodaj lambda jako parametr konstruktora do `ItemListAdapter{}`.

```
val adapter = ItemListAdapter {
}
```

3. Wewnątrz lambdy utwórz `val`nazwę `action`. Wkrótce naprawisz błąd inicjalizacji.

```
val adapter = ItemListAdapter {
    val action
}
```

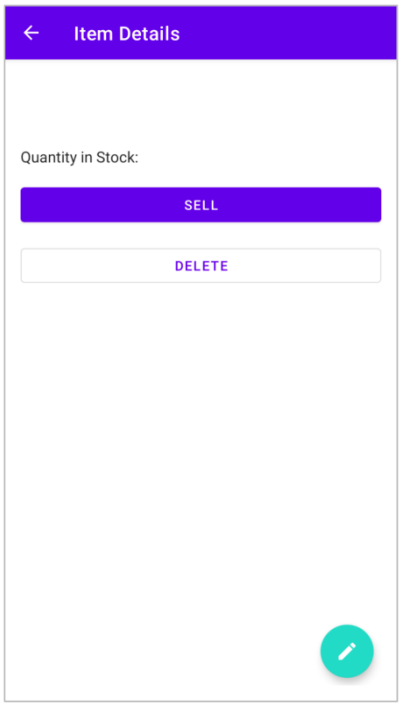
4. Wywołaj `actionItemListFragmentToItemDetailFragment()`metodę na `ItemListFragmentDirections`przejściu w elemencie `id`. Przypisz zwrócony `NavDirections`obiekt do `action`.

```
val adapter = ItemListAdapter {
  val action =    ItemListFragmentDirections.actionItemListFragmentToItemDetailFragment(it.id)
}
```

5. Poniżej `action`definicji pobierz `NavController`instancję za pomocą i wywołaj ją, przekazując w . Definicja adaptera powinna wyglądać tak:`this.findNavController()navigate()action`

```
val adapter = ItemListAdapter {
  val action =   ItemListFragmentDirections.actionItemListFragmentToItemDetailFragment(it.id)
  this.findNavController().navigate(action)
}
```

6. Uruchom aplikację. Kliknij element w `RecyclerView`. Aplikacja przejdzie do ekranu **Szczegóły pozycji .** Zwróć uwagę, że szczegóły są puste. Dotknij przycisków, nic się nie dzieje.



W kolejnych krokach wyświetlisz szczegóły encji na ekranie **Szczegóły pozycji** i dodasz funkcje do przycisków sprzedaży i usuwania.

Pobierz szczegóły przedmiotu

W tym kroku dodasz nową funkcję do `InventoryViewModel`, aby pobrać szczegóły pozycji z bazy danych na podstawie pozycji `id`. W następnym kroku użyjesz tej funkcji do wyświetlenia szczegółów jednostki na ekranie **Szczegóły pozycji** .

1. W programie `InventoryViewModel`dodaj funkcję o nazwie `retrieveItem()`, która przyjmuje jako `Int`identyfikator elementu i zwraca `LiveData<Item>`. Wkrótce naprawisz błąd wyrażenia zwrotnego.

```
fun retrieveItem(id: Int): LiveData<Item> {
}
```

2. Wewnątrz nowej funkcji wywołaj `getItem()`, `itemDao`przekazując parametr `id`. Funkcja `getItem()`zwraca `Flow`. Aby wykorzystać `Flow`wartość jako funkcję `LiveData`wywołania `asLiveData()`i użyć jej jako zwrotu `retrieveItem()`funkcji. Ukończona funkcja powinna wyglądać następująco:

```
fun retrieveItem(id: Int): LiveData<Item> {
  return itemDao.getItem(id).asLiveData()
}
```

Powiąż dane z TextViews

W tym kroku utworzysz instancję ViewModel w `ItemDetailFragment`i powiążesz dane ViewModel z TextViews na ekranie **Item Details** . Dołączysz również obserwatora do danych w ViewModel, aby aktualizować listę inwentaryzacyjną na ekranie, jeśli podstawowe dane w bazie danych ulegną zmianie.

1. W programie `ItemDetailFragment`dodaj zmienną właściwość o nazwie jednostki `item`typu `Item`. Użyjesz tej właściwości do przechowywania informacji o pojedynczej jednostce. Ta właściwość zostanie zainicjowana później, więc poprzedź ją przedrostkiem `lateinit`.

```
lateinit var item: Item
```

Importuj `com.example.inventory.data.Item`po wyświetleniu monitu przez Android Studio.

2. Na początku klasy `ItemDetailFragment`zadeklaruj `private`niezmienną właściwość o nazwie `viewModel`typu `InventoryViewModel`. Użyj `by`delegata, aby przekazać inicjalizację właściwości do `activityViewModels`klasy. Podaj `InventoryViewModelFactory`konstruktora.

```
private val viewModel: InventoryViewModel by activityViewModels {
  InventoryViewModelFactory(
     (activity?.application as InventoryApplication).database.itemDao()
  )
}
```

Importuj `androidx.fragment.app.activityViewModels`, jeśli pojawi się monit Android Studio.

3. Nadal w programie `ItemDetailFragment`utwórz `private`funkcję o nazwie `bind()`, która przyjmuje instancję `Item`jednostki jako parametr i nic nie zwraca.

```
private fun bind(item: Item) {
}
```

4. Zaimplementuj `bind()`funkcję, jest to podobne do tego, co zrobiłeś w `ItemListAdapter`. Ustaw `text`właściwość `itemName`TextView na `item.itemName`. Wywołaj właściwość , aby sformatować wartość ceny, i ustaw ją na właściwość TextView. Przekonwertuj na i ustaw go na właściwość TextView.*getFormattedPrice*()itemtextitemPricequantityInStockStringtextitemQuantity

```
private fun bind(item: Item) {
  binding.itemName.text = item.itemName
  binding.itemPrice.text = item.getFormattedPrice()
  binding.itemCount.text = item.quantityInStock.toString()
}
```

5. Zaktualizuj `bind()`funkcję, aby używać `apply{}`funkcji zakresu do bloku kodu, jak pokazano poniżej.

```
private fun bind(item: Item) {
  binding.apply {
    itemName.text = item.itemName
    itemPrice.text = item.getFormattedPrice()
    itemCount.text = item.quantityInStock.toString()
  }
}
```

6. Nadal w `ItemDetailFragment`, zastąp `onViewCreated()`.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
}
```

7. W jednym z poprzednich kroków przekazano identyfikator elementu jako argument nawigacyjny `ItemDetailFragment`z elementu `ItemListFragment`. Wewnątrz `onViewCreated()`, poniżej wywołania funkcji superklasy, utwórz niezmienną zmienną o nazwie `id`. Pobierz i przypisz argument nawigacyjny do tej nowej zmiennej.

```
val id = navigationArgs.itemId
```

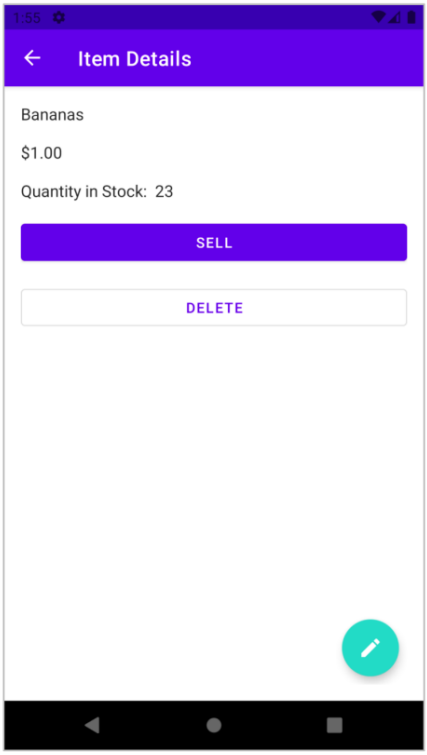
8. Teraz użyjesz tej `id`zmiennej, aby pobrać szczegóły pozycji. Jeszcze w środku `onViewCreated()`wywołaj `retrieveItem()`funkcję na `viewModel`przejściu w `id`. Dołącz obserwatora do zwróconej wartości przechodzącej w `viewLifecycleOwner`i lambda.

```
viewModel.retrieveItem(id).observe(this.viewLifecycleOwner) {
  }
```

9. Wewnątrz lambdy przekaż `selectedItem`jako parametr zawierający `Item`encję pobraną z bazy danych. W treści funkcji lambda przypisz `selectedItem`wartość do `item`. Wywołaj `bind()`funkcję przekazującą w `item`. Ukończona funkcja powinna wyglądać następująco.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  val id = navigationArgs.itemId
  viewModel.retrieveItem(id).observe(this.viewLifecycleOwner) { selectedItem ->
    item = selectedItem
    bind(item)
  }
}
```

10. Uruchom aplikację. Kliknij dowolny element listy na ekranie **ekwipunku** , zostanie wyświetlony ekran **Szczegóły przedmiotu .** Zauważ, że teraz ekran nie jest już pusty, wyświetla szczegóły encji pobrane z bazy danych inwentarzowych.



11. Dotknij przycisków **Sprzedaj** , **Usuń** i FAB. Nic się nie dzieje! W kolejnych zadaniach zaimplementujesz funkcjonalność tych przycisków.

5. Wdrożenie sprzedaj przedmiot

W tym zadaniu rozszerzysz funkcje aplikacji, wdrożysz funkcjonalność sprzedaży. Oto ogólne podsumowanie instrukcji dla tego kroku.

- Dodaj funkcję w ViewModel, aby zaktualizować encję
- Utwórz nową metodę, aby zmniejszyć ilość i zaktualizuj jednostkę w bazie danych aplikacji.
- Dołącz detektor kliknięć do przycisku **Sprzedaj**
- Wyłącz przycisk **Sprzedaj** , jeśli ilość wynosi zero.

Załóżmy kod:

1. W programie `InventoryViewModel`dodaj prywatną funkcję o nazwie `updateItem()`, która pobiera instancję klasy encji `Item`i nic nie zwraca.

```
private fun updateItem(item: Item) {
}
```

2. Zaimplementuj nową metodę, `updateItem()`. Aby wywołać `update()`metodę suspend z `ItemDao`klasy, uruchom współprogram za pomocą `viewModelScope`. Wewnątrz bloku uruchamiania

wywołaj `update()`funkcję po `itemDao`przekazaniu `item`. Twoja ukończona metoda powinna wyglądać następująco.

```
private fun updateItem(item: Item) {
  viewModelScope.launch {
     itemDao.update(item)
  }
}
```

3. Nadal wewnątrz `InventoryViewModel`, dodaj inną wywołaną metodę `sellItem()`, która pobiera instancję `Item`klasy encji i nic nie zwraca.

```
fun sellItem(item: Item) {
}
```

4. Wewnątrz `sellItem()`funkcji dodaj `if`warunek, aby sprawdzić, czy `item.quantityInStock`jest większe niż `0`.

```
fun sellItem(item: Item) {
  if (item.quantityInStock > 0) {
  }
}
```

Wewnątrz `if`bloku użyjesz `copy()`funkcji dla klasy Data do aktualizacji encji.

Klasa danych: kopia()

Funkcja `copy()`jest domyślnie udostępniana wszystkim instancjom klas danych. Ta funkcja służy do kopiowania obiektu w celu zmiany niektórych jego właściwości, ale pozostałe właściwości pozostają niezmienione.

Rozważmy na przykład `User`klasę i jej instancję `jack`, jak pokazano poniżej. Jeśli chcesz utworzyć nową instancję, tylko aktualizując `age`właściwość, jej implementacja wyglądałaby następująco:

Przykład

```
// Data class
data class User(val name: String = "", val age: Int = 0)

// Data class instance
val jack = User(name = "Jack", age = 1)

// A new instance is created with its age property changed, rest of the properties unchanged.
val olderJack = jack.copy(age = 2)
```

5. Wróćmy do `sellItem()`funkcji w `InventoryViewModel`. Wewnątrz `if`bloku utwórz nową niezmienną właściwość o nazwie `newItem`. Wywołanie `copy()`funkcji na `item`instancji przechodzącej w zaktualizowany `quantityInStock`, czyli zmniejszenie stanu magazynowego o `1`.

```
val newItem = item.copy(quantityInStock = item.quantityInStock - 1)
```

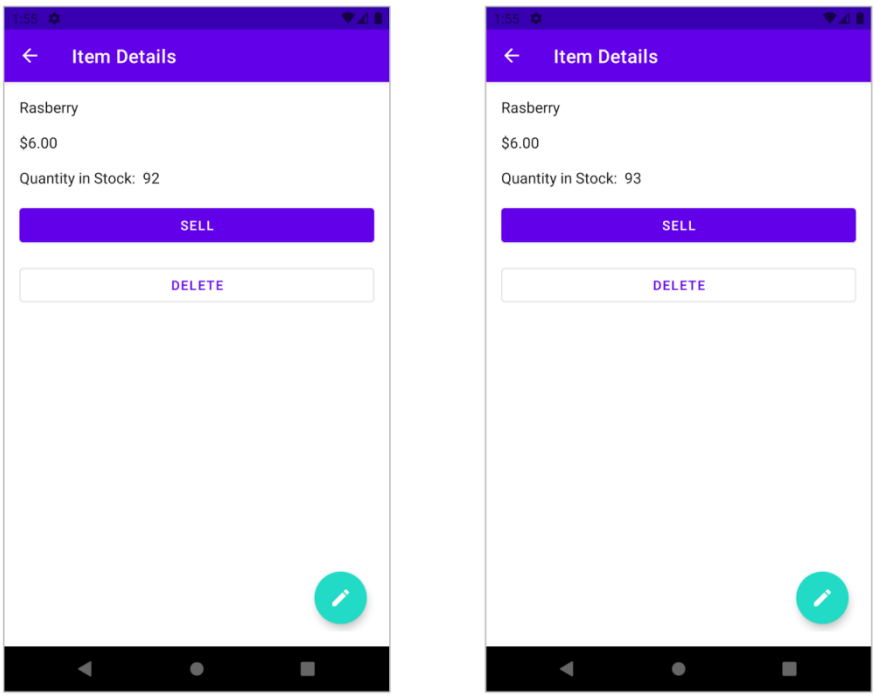
6. Poniżej definicji `newItem`, wykonaj wywołanie `updateItem()`funkcji przekazującej w nowej zaktualizowanej encji, czyli `newItem`. Ukończona metoda powinna wyglądać następująco.

```
fun sellItem(item: Item) {
  if (item.quantityInStock > 0) {
     // Decrease the quantity by 1
     val newItem = item.copy(quantityInStock = item.quantityInStock - 1)
     updateItem(newItem)
  }
}
```

7. Aby dodać funkcję sprzedaży zapasów, przejdź do `ItemDetailFragment`. Przewiń do końca `bind()`funkcji. Wewnątrz `apply`bloku ustaw detektor kliknięć na przycisk **Sprzedaj** i wywołaj `sellItem()`funkcję na `viewModel`.

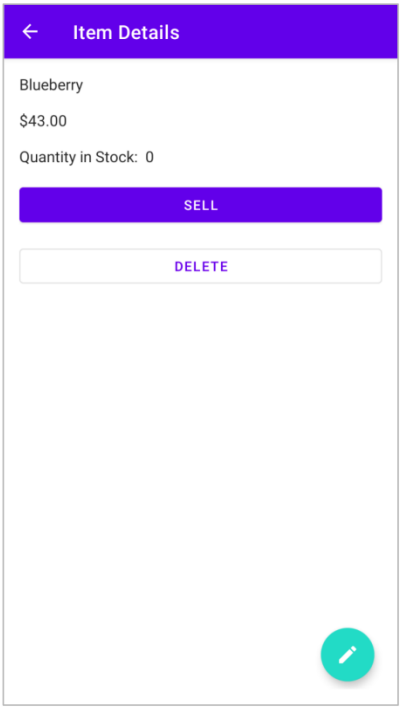
```
private fun bind(item: Item) {
binding.apply {

...
   sellItem.setOnClickListener { viewModel.sellItem(item) }
   }
}
```

8. Uruchom aplikację. Na ekranie **Inwentarz** kliknij element listy z ilością większą niż zero. Zostanie wyświetlony ekran **Szczegóły pozycji .** Dotknij przycisku **Sprzedaj** , zauważ, że wartość ilości została zmniejszona o jeden.



9. Na ekranie **Szczegóły przedmiotu ustaw** ilość na 0, stale dotykając przycisku **Sprzedaj** . (Wskazówka: wybierz jednostkę z mniejszą ilością zapasów lub utwórz nową z mniejszą ilością). Gdy ilość wynosi zero, naciśnij przycisk **Sprzedaj** . Nie będzie zmiany

wizualnej. Dzieje się tak, ponieważ Twoja funkcja `sellItem()`sprawdza, czy ilość jest większa od zera, przed aktualizacją ilości.



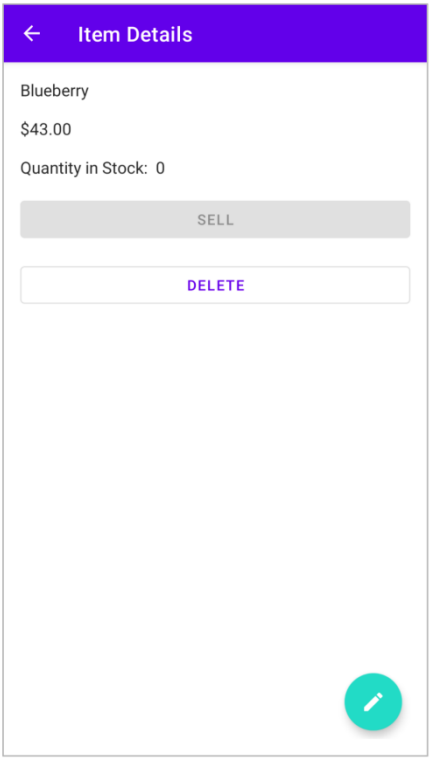
10. Aby przekazać użytkownikom lepsze opinie, możesz wyłączyć przycisk **Sprzedaj** , gdy nie ma przedmiotu do sprzedania. W `InventoryViewModel`programie dodaj funkcję sprawdzającą, czy ilość jest większa niż `0`. Nazwij funkcję `isStockAvailable()`, która pobiera `Item`instancję i zwraca `Boolean`.

```
fun isStockAvailable(item: Item): Boolean {
  return (item.quantityInStock > 0)
}
```

11. Przejdź do `ItemDetailFragment`, przewiń do `bind()`funkcji. Wewnątrz bloku Apply wywołaj `isStockAvailable()`funkcję przy `viewModel`przekazywaniu `item`. Ustaw wartość zwracaną na `isEnabled`właściwość przycisku **Sprzedaj** . Twój kod powinien wyglądać mniej więcej tak.

```
private fun bind(item: Item) {
  binding.apply {
    ...
    sellItem.isEnabled = viewModel.isStockAvailable(item)
    sellItem.setOnClickListener { viewModel.sellItem(item) }
  }
}
```

12. Uruchom aplikację i zauważ, że przycisk **Sprzedaj** jest nieaktywny, gdy ilość w magazynie wynosi zero. Gratulujemy wdrożenia funkcji sprzedaży przedmiotu w Twojej aplikacji.



Usuń element elementu

Podobnie jak w poprzednim zadaniu, rozszerzysz funkcje swojej aplikacji, wdrażając funkcję usuwania. Oto ogólne instrukcje dotyczące tego kroku, jest to znacznie łatwiejsze niż wdrożenie funkcji sprzedaży.

- Dodaj funkcję w ViewModel, aby usunąć jednostkę z bazy danych
- Dodaj nową metodę w `ItemDetailFragment`celu wywołania nowej funkcji usuwania i obsługi nawigacji.
- Dołącz detektor kliknięć do przycisku **Usuń** .

Kontynuujmy kodowanie:

1. W programie `InventoryViewModel`dodaj nową funkcję o nazwie `deleteItem()`, która pobiera instancję `Item`wywoływanej klasy encji `item`i nic nie zwraca. Wewnątrz `deleteItem()`funkcji uruchom współprogram za pomocą `viewModelScope`. Wewnątrz `launch`bloku wywołaj `delete()`metodę przy `itemDao`przekazywaniu w `item`.

```
fun deleteItem(item: Item) {
  viewModelScope.launch {
     itemDao.delete(item)
  }
}
```

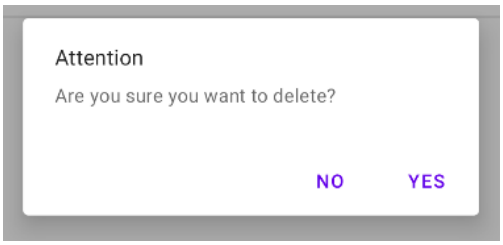
2. W `ItemDetailFragment`przewiń do początku `deleteItem()`funkcji. Zadzwoń `deleteItem()`, `viewModel`podaj `item`. Instancja `item`zawiera encję aktualnie wyświetlaną na ekranie **Szczegóły pozycji** . Twoja ukończona metoda powinna wyglądać tak.

```
private fun deleteItem() {
  viewModel.deleteItem(item)
  findNavController().navigateUp()
}
```

3. Pozostając w obrębie `ItemDetailFragment`, przewiń do `showConfirmationDialog()`funkcji. Ta funkcja jest podana jako część kodu startowego. Ta metoda wyświetla okno dialogowe alertu, aby uzyskać potwierdzenie użytkownika przed usunięciem elementu i wywołania `deleteItem()`funkcji po dotknięciu przycisku pozytywnego.

```
private fun showConfirmationDialog() {
      MaterialAlertDialogBuilder(requireContext())
         ...
         .setPositiveButton(getString(R.string.yes)) { _, _ ->
            deleteItem()
         }
         .show()
   }
```

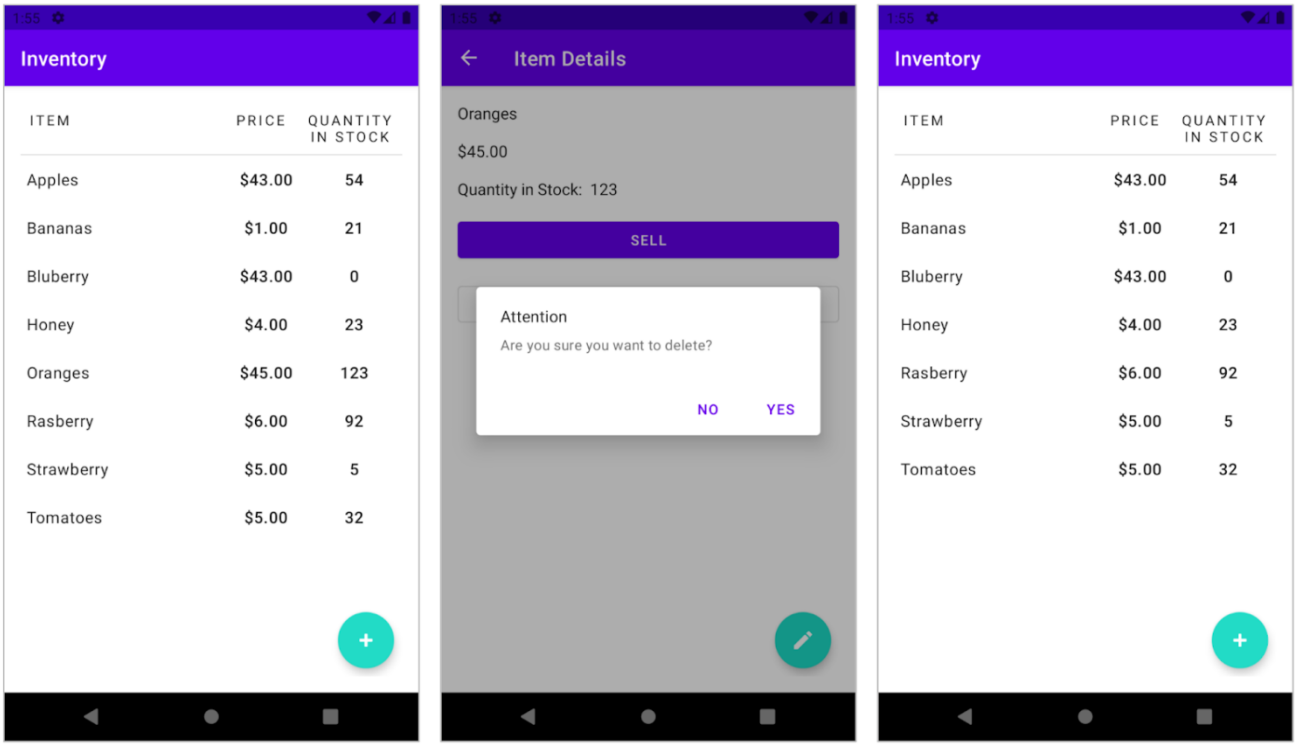
Funkcja `showConfirmationDialog()`wyświetla okno dialogowe alertu, które wygląda następująco:



4. W `ItemDetailFragment`, na końcu `bind()`funkcji, wewnątrz `apply`bloku, ustaw detektor kliknięć na przycisk usuwania. Wywołaj `showConfirmationDialog()`wewnątrz lambda nasłuchiwania kliknięć.

```
private fun bind(item: Item) {
  binding.apply {
     ...
     deleteItem.setOnClickListener { showConfirmationDialog() }
  }
}
```

5. Uruchom swoją aplikację! Wybierz element listy na ekranie Lista zapasów, na ekranie **Szczegóły pozycji** naciśnij przycisk **Usuń** . Stuknij Tak, aplikacja przejdzie z powrotem do ekranu ekwipunku. Zauważ, że usunięta encja nie znajduje się już w bazie danych aplikacji. Gratulujemy wdrożenia funkcji usuwania.



Edytuj element elementu

Podobnie jak w poprzednich zadaniach, w tym zadaniu dodasz kolejne ulepszenie funkcji do aplikacji. Zaimplementujesz encję elementu edycji.

Oto krótki przegląd kroków edycji encji w bazie danych aplikacji:

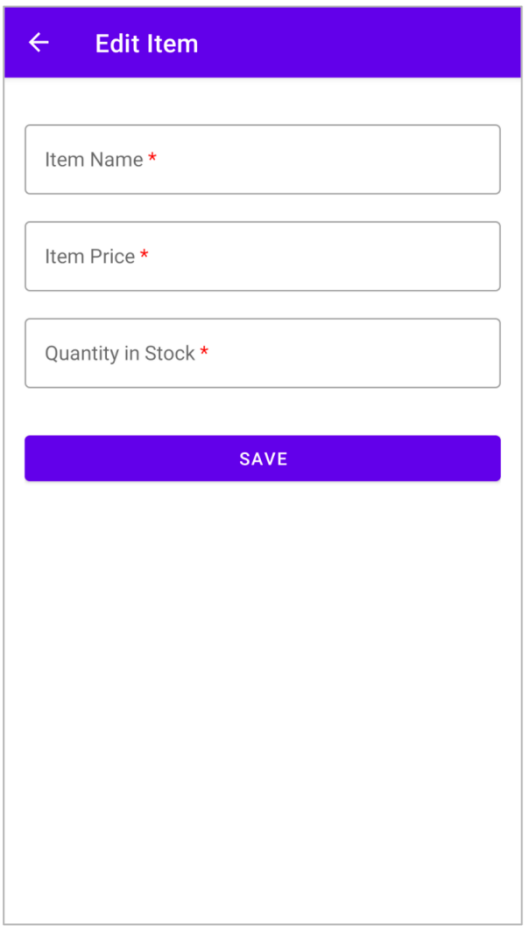
- Użyj ponownie ekranu **Dodaj element** , aktualizując tytuł fragmentu do Edytuj element.
- Dodaj odbiornik kliknięć do FAB, aby przejść do ekranu **Edytuj element .**
- Wypełnij TextViews szczegółami jednostki.
- Zaktualizuj jednostkę w bazie danych za pomocą Room.

Dodaj słuchacz kliknięć do FAB

1. W `ItemDetailFragment`programie dodaj nową `private`funkcję o nazwie `editItem()`, która nie przyjmuje parametrów i nic nie zwraca. W następnym kroku będziesz ponownie używać `fragment_add_item.xml`, aktualizując tytuł ekranu na **Edytuj element** . Aby to osiągnąć, w ramach akcji wyślesz fragment tytułu wraz z identyfikatorem przedmiotu.

```
private fun editItem() {
}
```

Po zaktualizowaniu tytułu fragmentu ekran **edycji elementu** powinien wyglądać następująco.



2. Wewnątrz `editItem()`funkcji utwórz niezmienną zmienną o nazwie `action`. Wywołaj `actionItemDetailFragmentToAddItemFragment()`po `ItemDetailFragmentDirections`przek azaniu ciągu tytułu `edit_fragment_title`i elementu `id`. Przypisz zwróconą wartość do `action`. Poniżej definicji `action`, wywołaj `this.findNavController().navigate()`przekazywanie w , `action`aby przejść do ekranu **Edytuj element .**

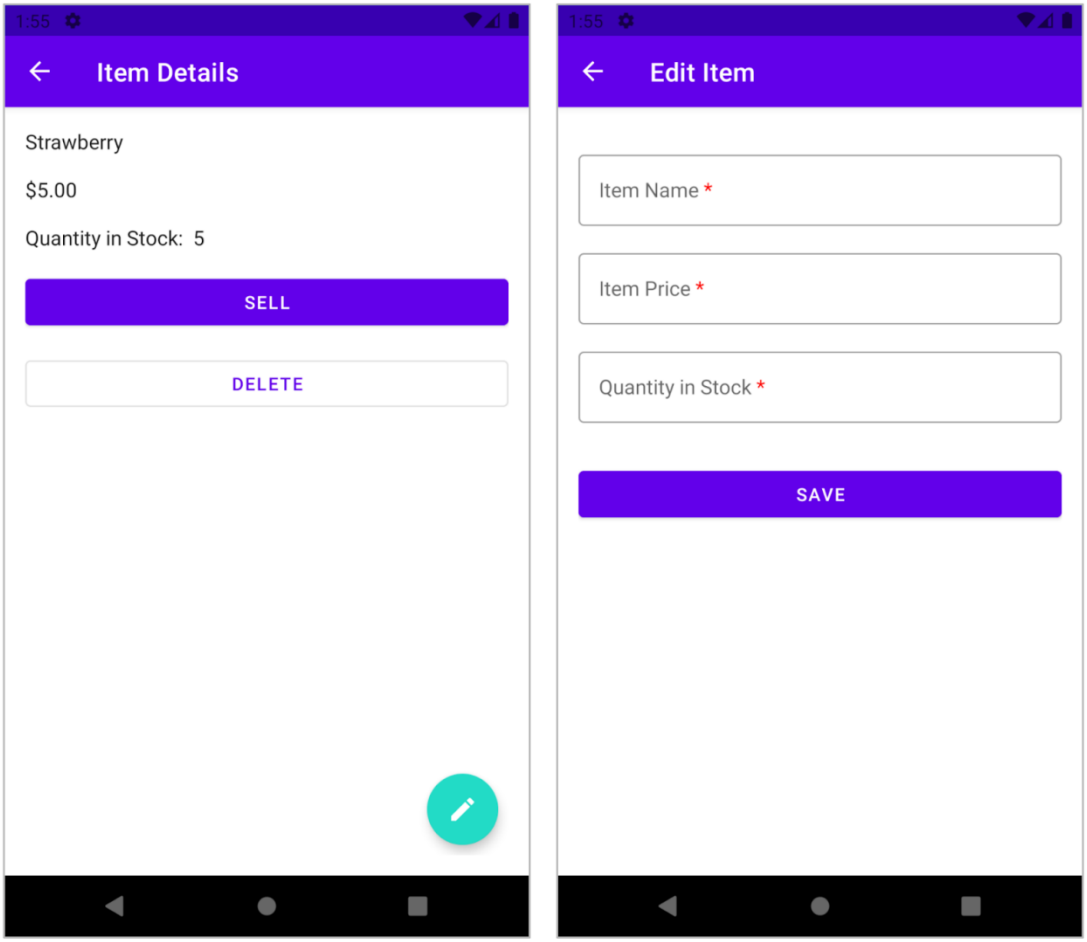
```
private fun editItem() {
  val action = ItemDetailFragmentDirections.actionItemDetailFragmentToAddItemFragment(
     getString(R.string.edit_fragment_title),
     item.id
  )
  this.findNavController().navigate(action)
}
```

3. Pozostając w obrębie `ItemDetailFragment`, przewiń do `bind()`funkcji. Wewnątrz `apply`bloku ustaw detektor kliknięć na FAB, wywołaj `editItem()`funkcję z lambda, aby przejść do ekranu **Edytuj element .**

```
private fun bind(item: Item) {
  binding.apply {
     ...
     editItem.setOnClickListener { editItem() }
  }
```

}

4. Uruchom aplikację. Przejdź do ekranu **Szczegóły pozycji .** Kliknij FAB. Zauważ, że tytuł ekranu został zaktualizowany do Edytuj element, ale wszystkie pola tekstowe są puste. W następnym kroku naprawisz to.



Wypełnij widoki tekstu

W tym kroku wypełnisz pola tekstowe na ekranie **Edytuj element** szczegółami encji. Ponieważ korzystamy z `Add Item`ekranu, dodamy nowe funkcje do pliku Kotlin, `AddItemFragment.kt`.

1. W programie `AddItemFragment`dodaj nową `private`funkcję, aby powiązać pola tekstowe ze szczegółami jednostki. Nazwij funkcję `bind()`, która przyjmuje instancję klasy encji Item i nic nie zwraca.

```
private fun bind(item: Item) {
}
```

2. Implementacja `bind()`funkcji jest bardzo podobna do tej, którą zrobiłeś wcześniej w programie `ItemDetailFragment`. Wewnątrz `bind()`funkcji zaokrąglij cenę do dwóch miejsc po przecinku za pomocą `format()`funkcji i przypisz ją do `val`nazwanej `price`, jak pokazano poniżej.

```
val price = "%.2f".format(item.itemPrice)
```

3. Poniżej `price`definicji użyj `apply`funkcji zakresu we `binding`właściwości, jak pokazano poniżej.

```
binding.apply {
}
```

4. Wewnątrz `apply`bloku kodu funkcji zakresu Ustaw `item.itemName`na właściwość text `itemName`. Użyj `setText()`funkcji i przekaż w `item.itemName`łańcuchu i `TextView.BufferType.SPANNABLE`jako `BufferType`.

```
binding.apply {
  itemName.setText(item.itemName, TextView.BufferType.SPANNABLE)
}
```

Importuj `android.widget.TextView`, jeśli pojawi się monit Android Studio.

5. Podobnie jak w powyższym kroku, ustaw właściwość tekstową ceny `EditText`, jak pokazano poniżej. W celu ustawienia `text`właściwości ilości EditText pamiętaj, aby przekonwertować `item.quantityInStock`na `String`. Twoja ukończona funkcja powinna wyglądać tak.

```
private fun bind(item: Item) {
  val price = "%.2f".format(item.itemPrice)
  binding.apply {
     itemName.setText(item.itemName, TextView.BufferType.SPANNABLE)
     itemPrice.setText(price, TextView.BufferType.SPANNABLE)
     itemCount.setText(item.quantityInStock.toString(), TextView.BufferType.SPANNABLE)
  }
}
```

6. Nadal wewnątrz `AddItemFragment`, przewiń do `onViewCreated()`funkcji. Po wywołaniu funkcji superklasy. Utwórz `val`wywołane `id`i pobierz `itemId`z argumentów nawigacji.

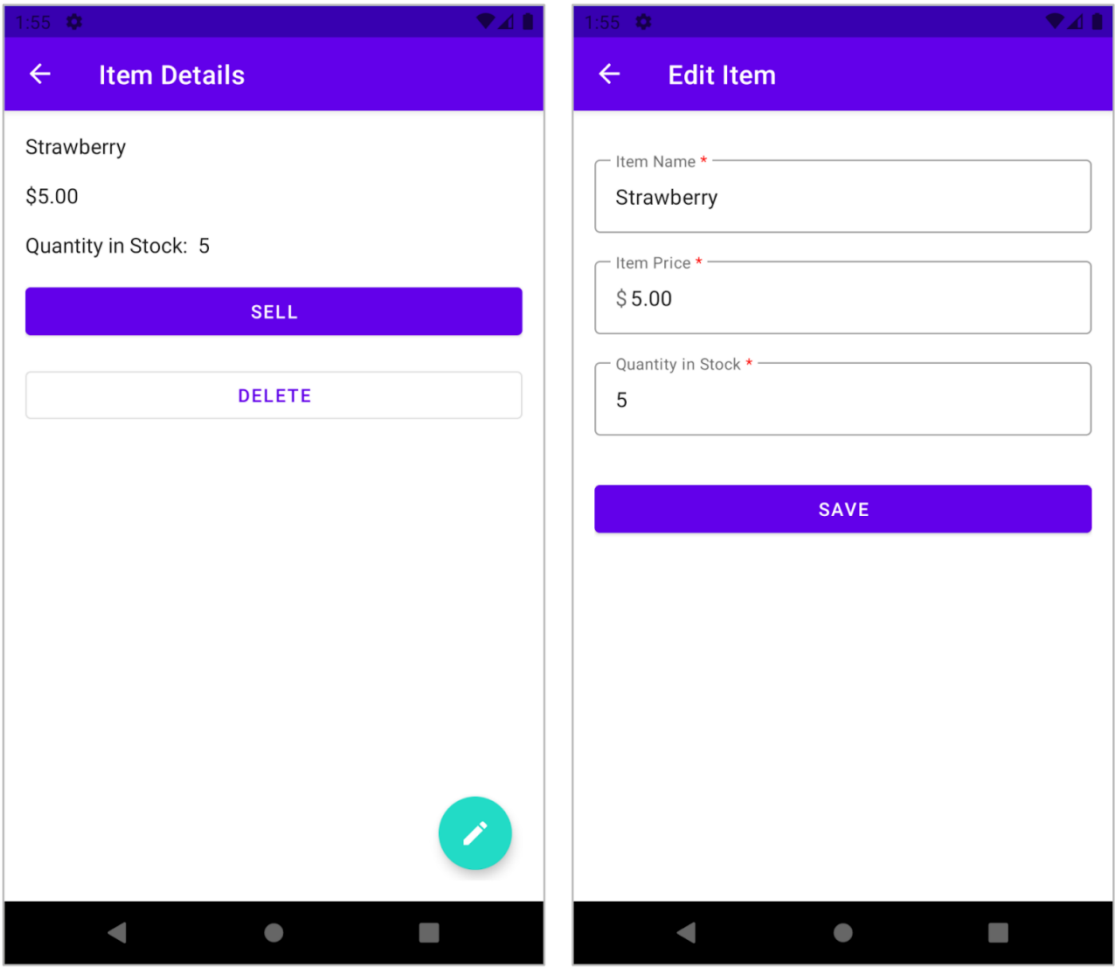
```
val id = navigationArgs.itemId
```

7. Dodaj `if-else`blok z warunkiem, aby sprawdzić, czy `id`jest większy od zera i przesuń przycisk **Zapisz** kliknij detektor do `else`bloku. Wewnątrz `if`bloku pobierz encję za pomocą `id`i dodaj do niej obserwatora. Wewnątrz obserwatora zaktualizuj `item`właściwość i wywołaj `bind()`przekazywanie w `item`. Pełna funkcja jest dostępna do kopiowania i wklejania. Jest prosty i łatwy do zrozumienia; musisz sam to rozszyfrować.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  val id = navigationArgs.itemId
  if (id > 0) {
     viewModel.retrieveItem(id).observe(this.viewLifecycleOwner) { selectedItem ->
        item = selectedItem
        bind(item)
     }
  } else {
     binding.saveAction.setOnClickListener {
        addNewItem()
     }
```

```
    }
}
```

8. Uruchom aplikację, przejdź do **szczegółów elementu** , dotknij **+** FAB. Zauważ, że pola są wypełnione szczegółami pozycji. Edytuj stan magazynowy lub dowolne inne pole i naciśnij przycisk Zapisz. Nic się nie dzieje! Dzieje się tak, ponieważ nie aktualizujesz jednostki w bazie danych aplikacji. Wkrótce to naprawisz.



Zaktualizuj jednostkę za pomocą Room

W tym ostatnim zadaniu dodaj ostatnie fragmenty kodu, aby zaimplementować funkcję aktualizacji. Niezbędne funkcje zdefiniujesz w ViewModel i użyjesz ich w `AddItemFragment`.

Znowu czas na kodowanie!

1. W programie `InventoryViewModel`dodaj `private`funkcję o nazwie `getUpdatedItemEntry()`, która przyjmuje ciągi , oraz `Int`trzy ciągi znaków dla szczegółów jednostki o nazwie `itemName`i . Zwróć instancję funkcji z funkcji. Kod podano w celach informacyjnych.`itemPriceitemCountItem`

```
private fun getUpdatedItemEntry(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
```

```
): Item {
}
```

2. Wewnątrz `getUpdatedItemEntry()`funkcji utwórz instancję Item za pomocą parametrów funkcji, jak pokazano poniżej. Zwróć `Item`instancję z funkcji.

```
private fun getUpdatedItemEntry(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
): Item {
    return Item(
        id = itemId,
        itemName = itemName,
        itemPrice = itemPrice.toDouble(),
        quantityInStock = itemCount.toInt()
    )
}
```

3. Nadal wewnątrz `InventoryViewModel`, dodaj kolejną funkcję o nazwie `updateItem()`. Ta funkcja pobiera również `Int`ciągi i trzy dla szczegółów jednostki i nic nie zwraca. Użyj nazw zmiennych z poniższego fragmentu kodu.

```
fun updateItem(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
) {
}
```

4. Wewnątrz `updateItem()`funkcji wykonaj wywołanie `getUpdatedItemEntry()`funkcji przekazującej informacje o jednostce, które są przekazywane jako parametry funkcji, jak pokazano poniżej. Przypisz zwróconą wartość do niezmiennej zmiennej o nazwie `updatedItem`.

```
val updatedItem = getUpdatedItemEntry(itemId, itemName, itemPrice, itemCount)
```

5. Tuż pod wywołaniem `getUpdatedItemEntry()`funkcji wykonaj wywołanie `updateItem()`funkcji przekazującej `updatedItem`. Wypełniona funkcja wygląda tak:

```
fun updateItem(
    itemId: Int,
    itemName: String,
    itemPrice: String,
    itemCount: String
) {
    val updatedItem = getUpdatedItemEntry(itemId, itemName, itemPrice, itemCount)
    updateItem(updatedItem)
```

```
}
```

6. Wróć do `AddItemFragment`, dodaj prywatną funkcję wywoływaną `updateItem()`bez parametrów i nic nie zwracaj. Wewnątrz funkcji dodaj `if`warunek, aby sprawdzić poprawność danych wejściowych użytkownika, wywołując funkcję `isEntryValid()`.

```
private fun updateItem() {
  if (isEntryValid()) {
  }
}
```

7. Wewnątrz `if`bloku wykonaj wywołanie, aby `viewModel.updateItem()`przekazać szczegóły encji. Użyj `itemId`z argumentów nawigacji i innych szczegółów encji, takich jak nazwa, cena i ilość z EditTexts, jak pokazano poniżej.

```
viewModel.updateItem(
    this.navigationArgs.itemId,
    this.binding.itemName.text.toString(),
    this.binding.itemPrice.text.toString(),
    this.binding.itemCount.text.toString()
)
```

8. Poniżej `updateItem()`wywołania funkcji zdefiniuj `val`wywołane `action`. Wywołaj `actionAddItemFragmentToItemListFragment()`i `AddItemFragmentDirections`przypisz zwróconą wartość do `action`. Przejdź do `ItemListFragment`, zadzwoń `findNavController().navigate()`przechodząc w `action`.

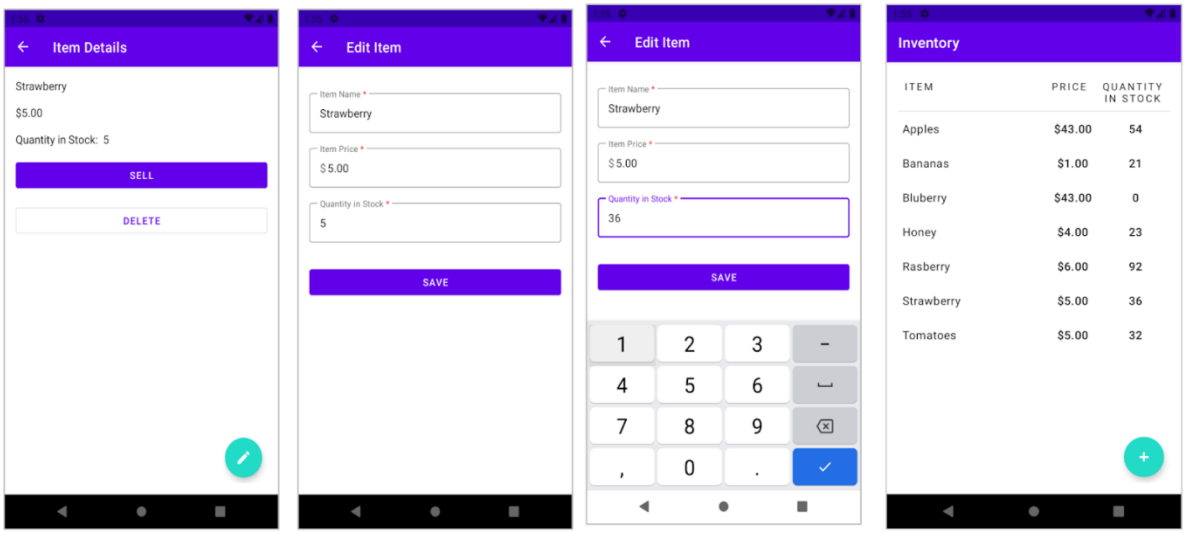
```
private fun updateItem() {
  if (isEntryValid()) {
     viewModel.updateItem(
        this.navigationArgs.itemId,
        this.binding.itemName.text.toString(),
        this.binding.itemPrice.text.toString(),
        this.binding.itemCount.text.toString()
     )
     val action = AddItemFragmentDirections.actionAddItemFragmentToItemListFragment()
     findNavController().navigate(action)
  }
}
```

9. Pozostając w obrębie `AddItemFragment`, przewiń do `bind()`funkcji. Wewnątrz `binding.`*`apply`*bloku funkcyjnego zakresu ustaw detektor kliknięcia dla przycisku **Zapisz** . Wykonaj wywołanie `updateItem()`funkcji wewnątrz lambdy, jak pokazano poniżej.

```
private fun bind(item: Item) {
  ...
  binding.apply {
     ...
     saveAction.setOnClickListener { updateItem() }
  }
```

}

10. Uruchom aplikację! Spróbuj edytować elementy ekwipunku; powinieneś być w stanie edytować dowolny element w bazie danych aplikacji Inventory.



Gratulujemy utworzenia Twojej pierwszej aplikacji do korzystania z Pokoju do zarządzania bazą danych aplikacji!

6. Kod rozwiązania

Kod rozwiązania dla tego ćwiczenia programowania znajduje się w repozytorium GitHub i gałęzi pokazanej poniżej.

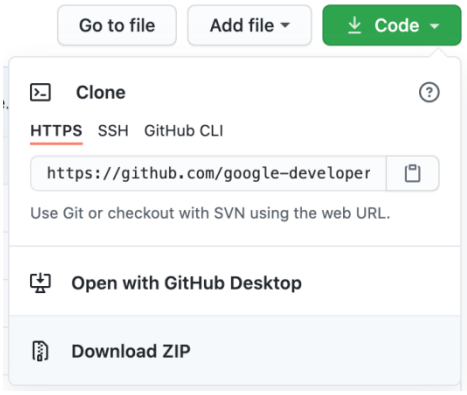
Adres URL kodu rozwiązania: https://github.com/google-developer-training/android-basics-kotlin-inventory-app

Nazwa filii:main

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

Pobierz kod

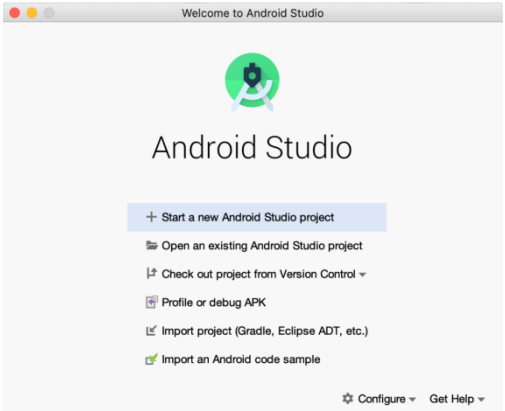
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



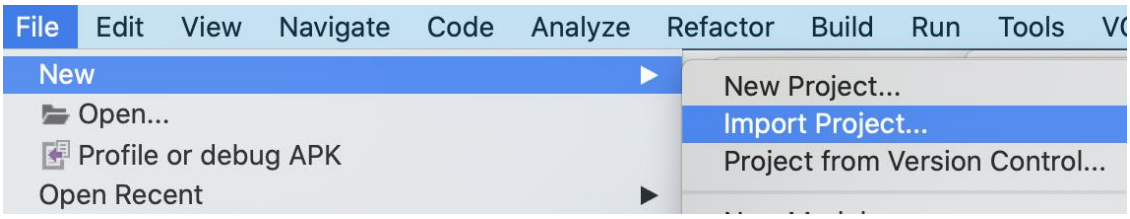
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego 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 działa zgodnie z oczekiwaniami.
7. Przeglądaj pliki projektu w oknie narzędzia **Projekt** , aby zobaczyć, jak aplikacja została zaimplementowana.

7. Podsumowanie

- Kotlin daje możliwość rozszerzenia klasy o nową funkcjonalność bez konieczności dziedziczenia po klasie lub modyfikowania istniejącej definicji klasy. Odbywa się to poprzez specjalne deklaracje zwane *rozszerzeniami* .
- Aby wykorzystać `Flow`dane jako `LiveData`wartość, użyj `asLiveData()`funkcji.
- Funkcja `copy()`jest domyślnie udostępniana wszystkim instancjom klas danych. Pozwala skopiować obiekt i zmienić niektóre jego właściwości, zachowując pozostałe właściwości bez zmian.

8. Dowiedz się więcej

Dokumentacja programisty Androida

- Przekazuj dane między miejscami docelowymi
- Ciąg Androida
- Formatowanie Androida
- Debuguj swoją bazę danych za pomocą Inspektora baz danych
- Zapisz dane w lokalnej bazie danych za pomocą Room

Referencje API

- `androidx.room`
- `asLiveData()`
- `TextView.BufferType`
- `AlertDialog.Builder`
- `ListAdapter`

Referencje Kotlina

- `Extensions`
- Funkcje zakresu

Wzorzec repozytorium

1. Zanim zaczniesz

Wstęp

W tym ćwiczeniu z programowania poprawisz środowisko użytkownika aplikacji, korzystając z buforowania w trybie offline. Wiele aplikacji korzysta z danych z sieci. Jeśli Twoja aplikacja pobiera dane z serwera przy każdym uruchomieniu, a użytkownik widzi ekran wczytywania, może to oznaczać złe wrażenia użytkownika, które prowadzą do odinstalowania aplikacji przez użytkowników.

Gdy użytkownicy uruchamiają aplikację, oczekują, że aplikacja szybko pokaże dane. Możesz osiągnąć ten cel, wdrażając buforowanie offline. Buforowanie offline oznacza, że Twoja aplikacja zapisuje dane pobrane z sieci w lokalnej pamięci urządzenia, co zapewnia szybszy dostęp.

Ponieważ aplikacja będzie mogła pobierać dane z sieci, a także przechowywać w trybie offline pamięć podręczną wcześniej pobranych wyników, będziesz potrzebować sposobu, w jaki Twoja aplikacja będzie mogła uporządkować te liczne źródła danych. Zrobisz to, implementując klasę repozytorium, która będzie służyć jako pojedyncze źródło prawdy dla danych aplikacji i wyabstrahować źródło danych (sieć, pamięć podręczna itp.) z modelu widoku.

Co powinieneś już wiedzieć

Powinieneś znać:

- Biblioteka trwałości danych, Pokój .
- Korzystanie z biblioteki sieciowej Retrofit .
- Podstawowe składniki architektury systemu Android , `ViewModel`, `ViewModelFactory`i `LiveData`.
- Transformacje dla klasy LiveData .
- Budowa i uruchomienie współprogramu .
- Powiązanie adapterów w powiązaniu danych.

Czego się nauczysz

- Jak zaimplementować repozytorium, aby oddzielić warstwę danych aplikacji od reszty aplikacji.
- Jak załadować dane z pamięci podręcznej za pomocą repozytorium.

Co zrobisz

- Użyj repozytorium, aby wyabstrahować warstwę danych i zintegruj klasę repozytorium z `ViewModel`.
- Wyświetl dane z pamięci podręcznej offline.

2. Kod startowy

Pobierz kod projektu

Zauważ, że nazwa folderu to `RepositoryPattern-Starter`. Wybierz ten folder podczas otwierania projektu w Android Studio.

Adres URL kodu startowego:

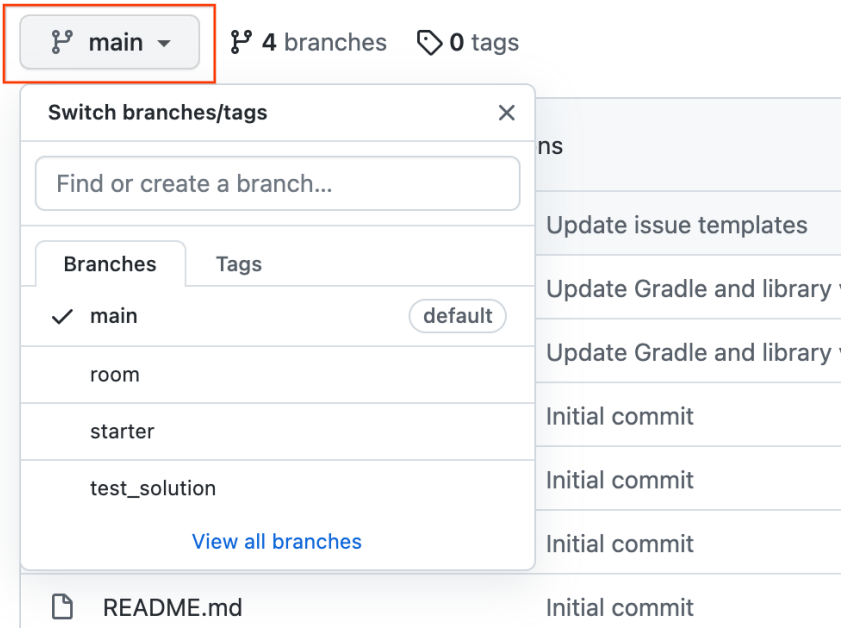
https://github.com/google-developer-training/android-kotlin-fundamentals-starter-apps/tree/master/RepositoryPattern-Starter

Nazwa oddziału z kodem startowym: `master`

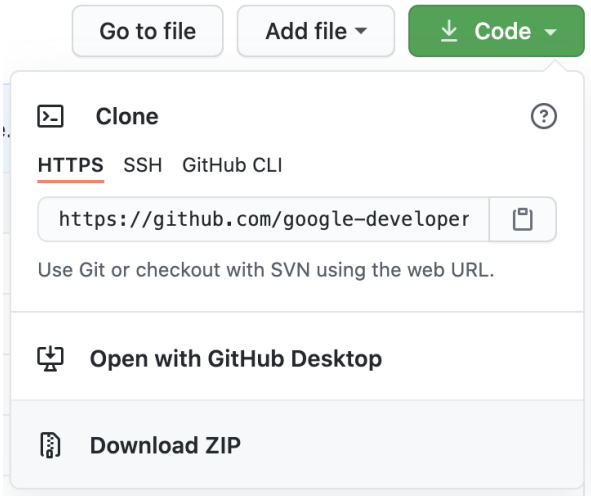
Aby uzyskać kod tego ćwiczenia z programowania i otworzyć go w Android Studio, wykonaj następujące czynności.

Pobierz kod

1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Sprawdź i potwierdź, że nazwa oddziału jest zgodna z nazwą oddziału określoną w ćwiczeniach z programowania. Na przykład na poniższym zrzucie ekranu nazwa gałęzi to **main** .



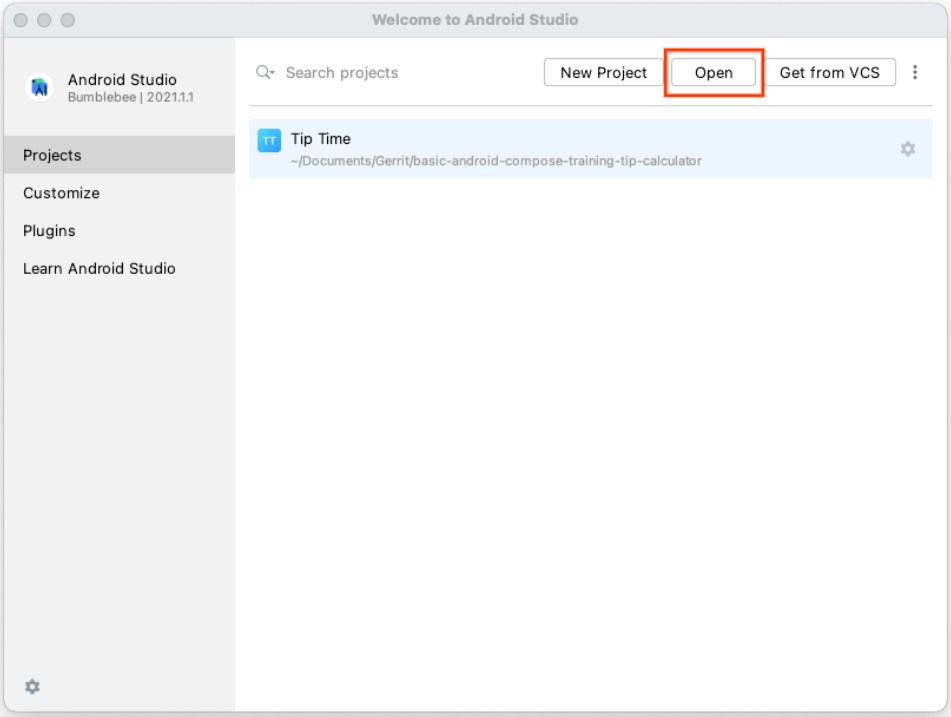
3. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli wyskakujące okienko.



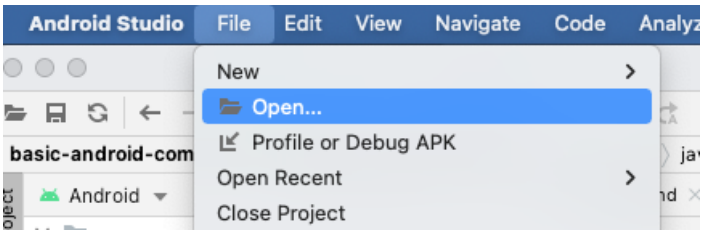
4. W wyskakującym okienku kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na komputerze. Poczekaj na zakończenie pobierania.
5. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
6. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

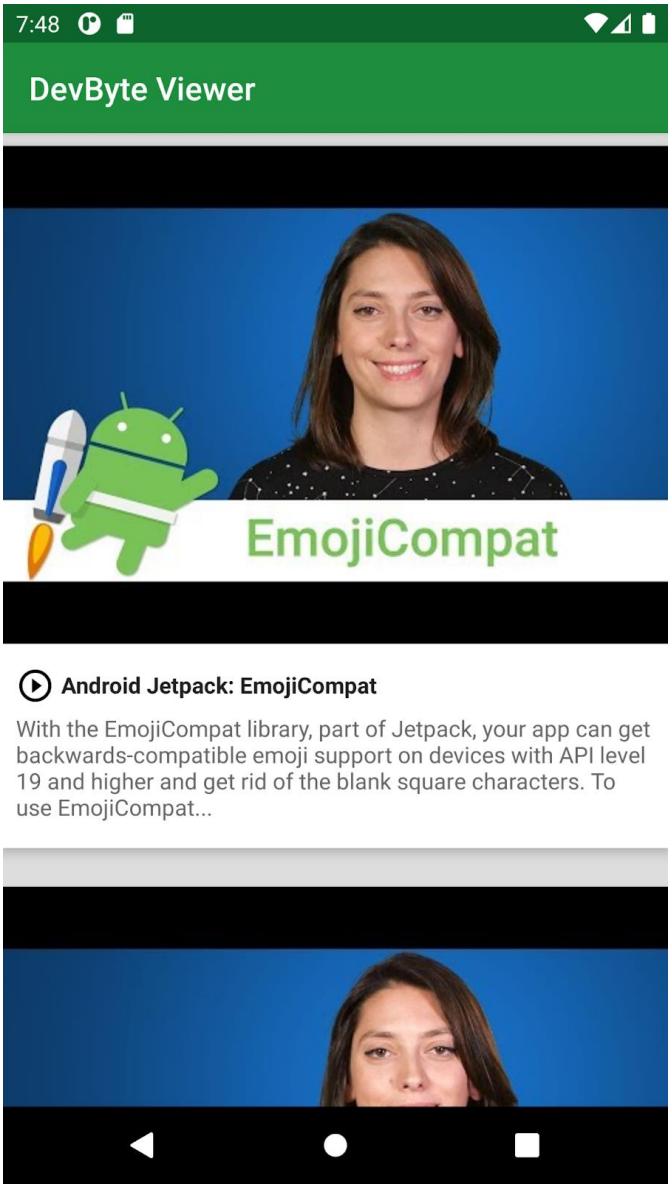


3. W przeglądarce plików przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.

6. Kliknij przycisk **Uruchom**  , aby skompilować i uruchomić aplikację. Upewnij się, że kompiluje się zgodnie z oczekiwaniami.

3. Przegląd aplikacji startowej

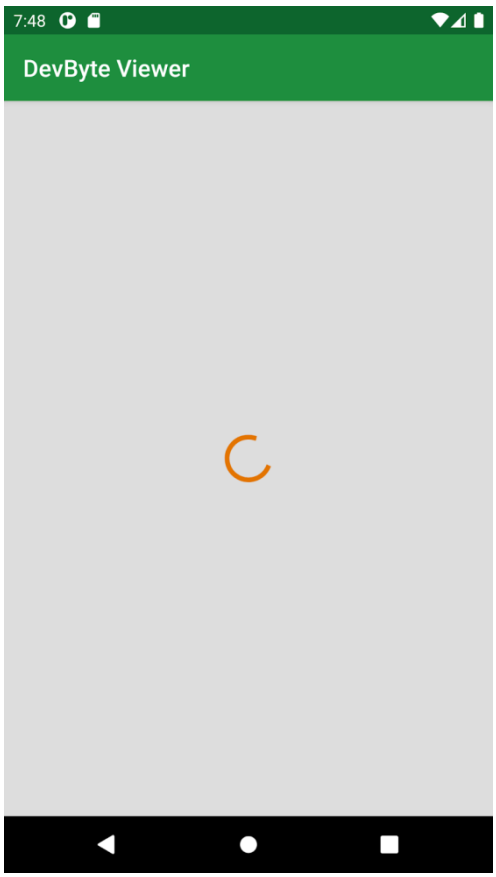
Aplikacja DevBytes przedstawia listę filmów DevBytes z kanału YouTube dla programistów Androida w widoku recyklera, w którym użytkownicy mogą kliknąć, aby otworzyć link do filmu.



Chociaż kod startowy działa w pełni, ma poważną wadę, która może negatywnie wpłynąć na wrażenia użytkownika. Jeśli użytkownik ma niestabilne połączenie lub w ogóle nie ma połączenia z Internetem, żaden z filmów nie zostanie wyświetlony. Dzieje się tak, nawet jeśli aplikacja została wcześniej otwarta. Jeśli użytkownik wyjdzie i ponownie uruchomi aplikację, tym razem bez Internetu, aplikacja spróbuje ponownie pobrać listę filmów i nie uda się jej pobrać.

Możesz to zobaczyć w akcji na emulatorze.

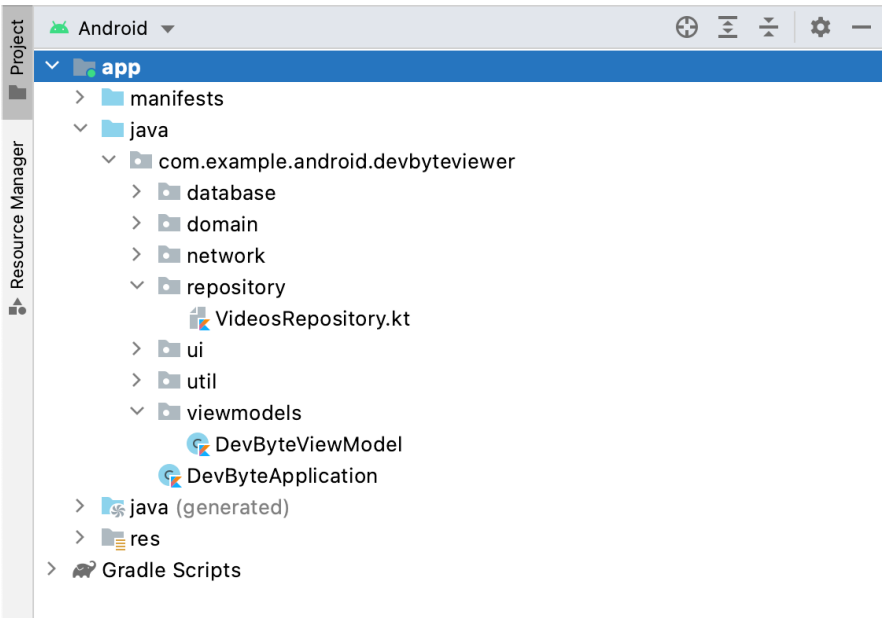
1. Tymczasowo włącz tryb samolotowy w emulatorze systemu Android (**aplikacja Ustawienia > Sieć i Internet > Tryb samolotowy**).
2. Uruchom aplikację DevBytes i obserwuj, czy ekran jest pusty.



3. Pamiętaj, aby wyłączyć tryb samolotowy przed kontynuowaniem pozostałych ćwiczeń z programowania.

   Dzieje się tak, ponieważ po pierwszym pobraniu danych przez aplikację DevBytes nic nie jest buforowane do późniejszego wykorzystania. Aplikacja zawiera obecnie bazę danych pokoi. Twoim zadaniem jest użycie go do zaimplementowania funkcji buforowania i zaktualizowania modelu widoku, aby korzystał z *repozytorium* , które albo pobierze nowe dane, albo pobierze je z bazy danych Room. Klasa repozytorium abstrahuje tę logikę z modelu widoku, zachowując porządek i rozdzielenie kodu.

   Projekt startowy jest podzielony na kilka pakietów.



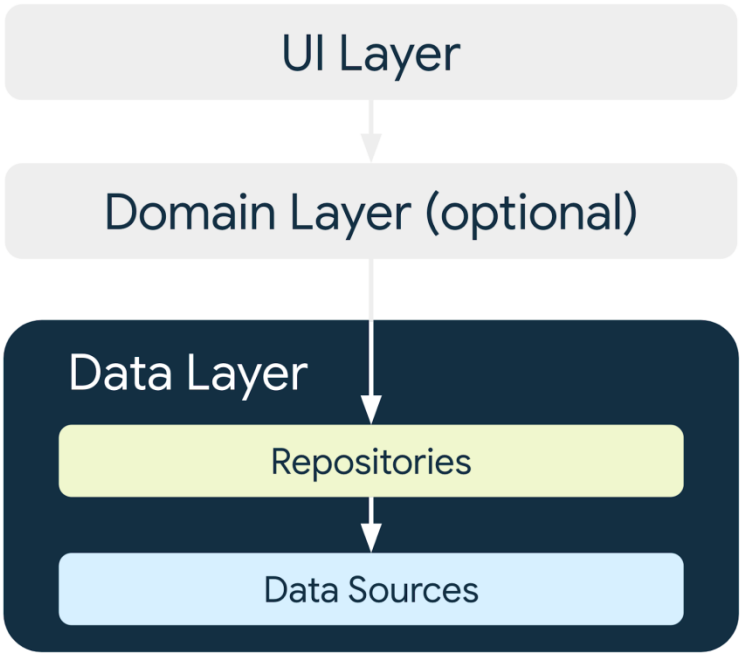
Chociaż jesteś mile widziany i zachęcany do zapoznania się z kodem, będziesz dotykał tylko dwóch plików: **repository/VideosRepository.kt** i **viewmodels/DevByteViewModel** . Najpierw utworzysz `VideosRepository`klasę, która implementuje wzorzec repozytorium do buforowania (więcej na ten temat dowiesz się na następnych kilku stronach), a następnie zaktualizujesz klasę, `DevByteViewModel`aby używała nowej `VideosRepository`klasy.

Zanim jednak przejdziesz do kodu, poświęćmy chwilę, aby dowiedzieć się więcej o buforowaniu i wzorcu repozytorium.

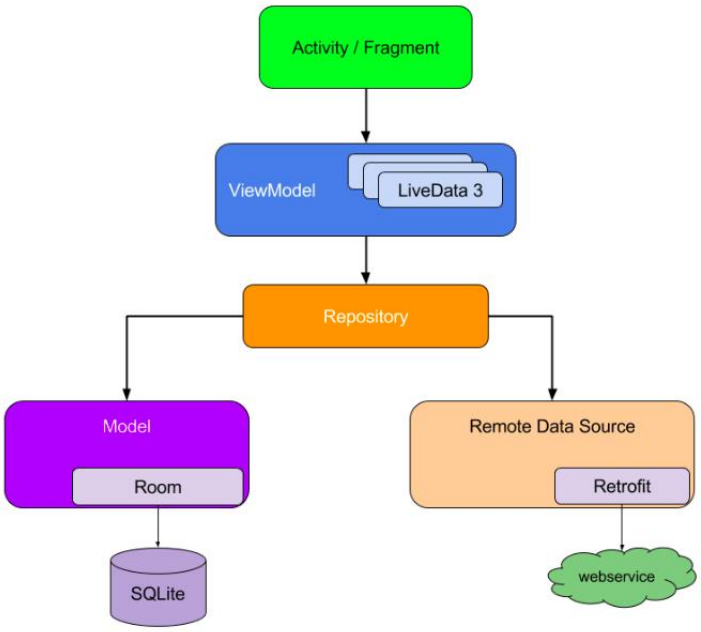
4. Buforowanie i wzorzec repozytorium

Repozytoria

Wzorzec repozytorium to wzorzec projektowy, który izoluje warstwę danych od reszty aplikacji. Warstwa danych odnosi się do części aplikacji, innej niż interfejs użytkownika, która obsługuje dane i logikę biznesową aplikacji, udostępniając spójne interfejsy API dla reszty aplikacji w celu uzyskania dostępu do tych danych. Podczas gdy interfejs użytkownika przedstawia informacje użytkownikowi, warstwa danych obejmuje takie elementy, jak kod sieciowy, bazy danych pomieszczeń, obsługa błędów i dowolny kod, który odczytuje lub manipuluje danymi.



Repozytorium może rozwiązywać konflikty między źródłami danych (takimi jak trwałe modele, usługi sieciowe i pamięci podręczne) oraz centralizować zmiany w tych danych. Poniższy diagram pokazuje, w jaki sposób składniki aplikacji, takie jak działania, mogą wchodzić w interakcje ze źródłami danych za pośrednictwem repozytorium.



Aby zaimplementować repozytorium, używasz oddzielnej klasy, takiej jak `VideosRepository`klasa, którą utworzysz w następnym zadaniu. Klasa repozytorium izoluje źródła danych od reszty aplikacji i zapewnia czysty interfejs API do dostępu do danych do reszty aplikacji. Użycie klasy repozytorium zapewnia, że ten kod jest oddzielony od `ViewModel`klasy i jest zalecaną najlepszą praktyką w zakresie separacji kodu i architektury.

Zalety korzystania z repozytorium

Moduł repozytorium obsługuje operacje na danych i umożliwia korzystanie z wielu backendów. W typowej aplikacji rzeczywistej repozytorium implementuje logikę podejmowania decyzji, czy pobrać dane z sieci, czy użyć wyników, które są buforowane w lokalnej bazie danych. Dzięki repozytorium można zamienić szczegóły implementacji, takie jak migracja do innej biblioteki trwałości, bez wpływu na kod wywołujący, taki jak modele widoku. Pomaga to również uczynić Twój kod modułowym i testowalnym. Możesz w prosty sposób zaprojektować repozytorium i przetestować resztę kodu.

Repozytorium powinno służyć jako pojedyncze źródło prawdy dla określonego fragmentu danych aplikacji. Podczas pracy z wieloma źródłami danych, takimi jak zasób sieciowy i pamięć podręczna w trybie offline, repozytorium zapewnia, że dane aplikacji są tak dokładne i aktualne, jak to tylko możliwe, zapewniając najlepsze możliwe wrażenia, nawet gdy aplikacja jest w trybie offline.

Buforowanie

Pamięć podręczna odnosi się do przechowywania danych używanych przez Twoją aplikację. Na przykład możesz chcieć tymczasowo zapisać dane z sieci na wypadek przerwania połączenia internetowego użytkownika. Mimo że sieć nie jest już dostępna, aplikacja może nadal korzystać z danych z pamięci podręcznej. Pamięć podręczna może być również przydatna do przechowywania tymczasowych danych dotyczących działań, które nie są już wyświetlane na ekranie, a nawet do utrwalania danych między uruchomieniami aplikacji.

Pamięć podręczna może przybierać różne formy, prostsze lub bardziej złożone, w zależności od konkretnego zadania. W poniższej tabeli przedstawiono kilka sposobów implementacji buforowania sieciowego w systemie Android.

Technika buforowania	**Zastosowania**
Retrofit to biblioteka sieciowa służąca do implementowania bezpiecznego typu klienta REST dla systemu Android. Możesz skonfigurować Retrofit tak, aby lokalnie przechowywał kopię każdego wyniku sieciowego.	Dobre rozwiązanie dla prostych żądań i odpowiedzi, rzadkich połączeń sieciowych lub małych zestawów danych.
Możesz używać DataStoredo przechowywania par klucz-wartość.	Dobre rozwiązanie dla małej liczby kluczy i prostych wartości, takich jak ustawienia aplikacji. Nie możesz używać tej techniki do przechowywania dużych ilości uporządkowanych danych.
Możesz uzyskać dostęp do katalogu pamięci wewnętrznej aplikacji i zapisywać w nim pliki danych. Nazwa pakietu aplikacji określa wewnętrzny katalog pamięci aplikacji, który znajduje się w specjalnej lokalizacji w systemie plików Androida. Ten katalog jest prywatny dla Twojej aplikacji i jest usuwany po odinstalowaniu aplikacji.	Dobre rozwiązanie, jeśli masz specyficzne potrzeby, które system plików może rozwiązać - na przykład, jeśli potrzebujesz zapisywać pliki multimedialne lub pliki danych i musisz samodzielnie zarządzać plikami. Nie możesz używać tej techniki do przechowywania złożonych i ustrukturyzowanych danych, których aplikacja wymaga zapytań.
Możesz buforować dane za pomocą Room , która jest biblioteką mapowania obiektów SQLite, która zapewnia warstwę abstrakcji w stosunku do SQLite.	Zalecane rozwiązanie w przypadku złożonych, ustrukturyzowanych danych, które można przeszukiwać, ponieważ najlepszym sposobem przechowywania ustrukturyzowanych danych w systemie plików urządzenia jest lokalna baza danych SQLite.

W tym ćwiczeniu z programowania używasz pokoju, ponieważ jest to zalecany sposób przechowywania uporządkowanych danych w systemie plików urządzenia. Aplikacja DevBytes jest już skonfigurowana do korzystania z pokoju. Twoim zadaniem jest zaimplementowanie buforowania offline przy użyciu wzorca repozytorium, aby oddzielić warstwę danych od kodu interfejsu użytkownika.

5. Zaimplementuj repozytorium wideo

Zadanie: Utwórz repozytorium

W tym zadaniu utworzysz repozytorium do zarządzania pamięcią podręczną offline, którą zaimplementowałeś w poprzednim zadaniu. Twoja baza danych pokoju nie ma logiki zarządzania pamięcią podręczną offline, ma tylko metody wstawiania, aktualizowania, usuwania i pobierania danych. Repozytorium będzie miało logikę do pobierania wyników sieci i aktualizowania bazy danych.

Krok 1: Dodaj repozytorium

1. W **repozytorium/VideosRepository.kt** utwórz `VideosRepository`klasę. Przekaż `VideosDatabase`obiekt jako parametr konstruktora klasy, aby uzyskać dostęp do metod DAO.

```
class VideosRepository(private val database: VideosDatabase) {
}
```

2. Wewnątrz `VideosRepository`klasy dodaj `suspend`metodę o nazwie `refreshVideos()`, która nie ma argumentów i nic nie zwraca. Ta metoda będzie interfejsem API używanym do odświeżania pamięci podręcznej offline.

```
suspend fun refreshVideos() {
}
```

Uwaga: Bazy danych w systemie Android są przechowywane w systemie plików lub na dysku i w celu zapisania muszą wykonać operację we/wy dysku. Dyskowe we/wy, czyli odczytywanie i zapisywanie na dysku, jest powolne i zawsze blokuje bieżący wątek do czasu zakończenia operacji. Z tego powodu musisz uruchomić dyskowe I/O w dyspozytorze I/O . Ten dyspozytor jest przeznaczony do odciążania blokowania zadań we/wy do współużytkowanej puli wątków przy użyciu .`withContext`(Dispatchers.IO) { ... }

3. Wewnątrz `refreshVideos()`metody przełącz kontekst współprogramu `Dispatchers.IO`na wykonywanie operacji sieciowych i bazodanowych.

```
suspend fun refreshVideos() {
  withContext(Dispatchers.IO) {
  }
}
```

4. Wewnątrz `withContext`bloku pobierz `DevByte`playlistę wideo z sieci za pomocą instancji usługi Retrofit `DevByteNetwork`.

```
val playlist = DevByteNetwork.devbytes.getPlaylist()
```

5. Wewnątrz `refreshVideos()`metody, po pobraniu listy odtwarzania z sieci, zapisz listę odtwarzania w bazie danych pokoju. Aby zapisać playlistę, użyj `VideosDatabase`klasy. Wywołaj `insertAll()`metodę DAO, przekazując listę odtwarzania pobraną z sieci. Użyj `asDatabaseModel()`funkcji rozszerzenia, aby zmapować listę odtwarzania na obiekt bazy danych.

```
database.videoDao.insertAll(playlist.asDatabaseModel())
```

6. Oto kompletna `refreshVideos()`metoda z instrukcją dziennika do śledzenia, gdy zostanie wywołana:

```
suspend fun refreshVideos() {
  withContext(Dispatchers.IO) {
     val playlist = DevByteNetwork.devbytes.getPlaylist()
     database.videoDao.insertAll(playlist.asDatabaseModel())
  }
}
```

Krok 2: Pobierz dane z bazy danych

W tym kroku tworzysz `LiveData`obiekt do odczytu listy odtwarzania wideo z bazy danych. Ten `LiveData`obiekt jest automatycznie aktualizowany podczas aktualizacji bazy danych. Załączony fragment, czyli aktywność, zostaje odświeżony o nowe wartości.

Uwaga:`LiveData` jest zachowana w tym przykładzie dla uproszczenia. Ogólnie zaleca się używanie Flowz repozytoriami, ponieważ jest to niezależne od cyklu życia.

1. W `VideosRepository`klasie zadeklaruj `LiveData`obiekt wywoływany `videos`do przechowywania listy `DevByteVideo`obiektów. Zainicjuj `videos`obiekt za pomocą `database.videoDao`. Wywołaj `getVideos()`metodę DAO. Ponieważ `getVideos()`metoda zwraca listę obiektów bazy danych, a nie listę `DevByteVideo`obiektów, Android Studio zgłasza błąd „niezgodności typu”.

```
val videos: LiveData<List<DevByteVideo>> = database.videoDao.getVideos()
```

2. Aby naprawić błąd, użyj `Transformations.map`funkcji konwersji listy obiektów bazy danych na listę obiektów domeny `asDomainModel()`.

```
val videos: LiveData<List<DevByteVideo>> = Transformations.map(database.videoDao.getVideos()) {
  it.asDomainModel()
}
```

Teraz zaimplementowałeś repozytorium dla swojej aplikacji. W następnym zadaniu użyjesz prostej strategii odświeżania, aby zapewnić aktualność lokalnej bazy danych.

Odświeżacz: Metoda Transformations.map()wykorzystuje funkcję konwersji do przekształcenia jednego `LiveData`obiektu w inny `LiveData`. Przekształcenia są obliczane tylko wtedy, gdy aktywna aktywność lub fragment obserwuje zwróconą `LiveData`właściwość.

6. Użyj VideosRepository w DevByteViewModel

Zadanie: Zintegruj repozytorium za pomocą strategii odświeżania

W tym zadaniu integrujesz swoje repozytorium z `ViewModel`wykorzystaniem prostej strategii odświeżania. Wyświetlasz listę odtwarzania wideo z bazy danych pokoi, a nie bezpośrednio z sieci.

Odświeżanie bazy danych to proces aktualizowania lub odświeżania lokalnej bazy danych w celu utrzymania jej synchronizacji z danymi z sieci. W przypadku tej przykładowej aplikacji użyjesz prostej strategii odświeżania, w której moduł żądający danych z repozytorium jest odpowiedzialny za odświeżanie danych lokalnych.

W rzeczywistej aplikacji Twoja strategia może być bardziej złożona. Na przykład Twój kod może automatycznie odświeżać dane w tle (biorąc pod uwagę przepustowość) lub buforować dane, których użytkownik najprawdopodobniej użyje w następnej kolejności.

1. W **viewmodels/DevByteViewModel.kt** , wewnątrz `DevByteViewModel`klasy, utwórz prywatną zmienną składową o nazwie `videosRepository`typu `VideosRepository`. Utwórz wystąpienie zmiennej, przekazując `VideosDatabase`obiekt singleton.

   ```
   private val videosRepository = VideosRepository(getDatabase(application))
   ```

2. W `DevByteViewModel`klasie zastąp `refreshDataFromNetwork()`metodę `refreshDataFromRepository()`metodą. Stara metoda, `refreshDataFromNetwork()`, pobierała playlistę wideo z sieci przy użyciu biblioteki Retrofit. Nowa metoda ładuje listę odtwarzania wideo z repozytorium. Repozytorium określa, z którego źródła (np. sieć, baza danych itp.) pobierana jest lista odtwarzania, zachowując szczegóły implementacji poza modelem widoku. Repozytorium sprawia również, że Twój kod jest łatwiejszy w utrzymaniu; jeśli miałbyś zmienić implementację do pobierania danych w przyszłości, nie musiałbyś modyfikować modelu widoku.

   ```
   private fun refreshDataFromRepository() {
     viewModelScope.launch {
        try {
           videosRepository.refreshVideos()
           _eventNetworkError.value = false
           _isNetworkErrorShown.value = false

        } catch (networkError: IOException) {
           // Show a Toast error message and hide the progress bar.
           if(playlist.value.isNullOrEmpty())
              _eventNetworkError.value = true
        }
     }
   }
   ```

3. W `DevByteViewModel`klasie, wewnątrz `init`bloku, zmień wywołanie funkcji z `refreshDataFromNetwork()`na `refreshDataFromRepository()`. Ten kod pobiera listę odtwarzania wideo z repozytorium, a nie bezpośrednio z sieci.

```
init {
  refreshDataFromRepository()
}
```

4. W `DevByteViewModel`klasie usuń `_playlist`właściwość i jej właściwość zapasową, `playlist`.

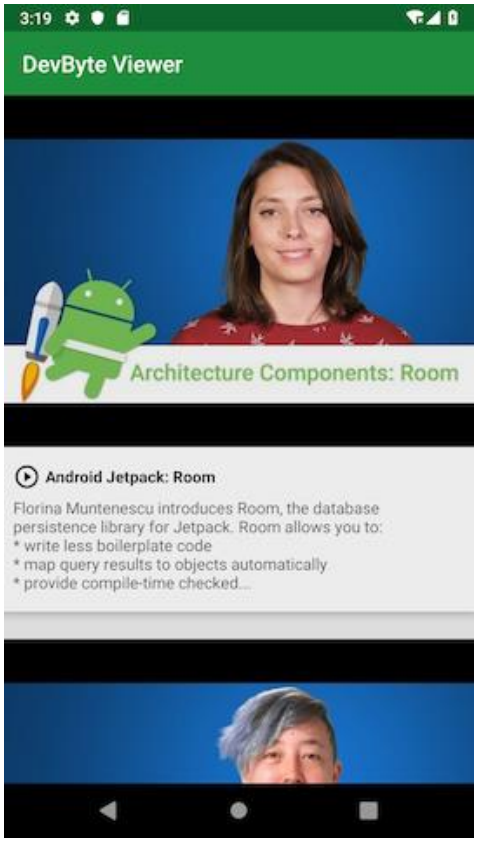
   Kod do usunięcia

   ```
   private val _playlist = MutableLiveData<List<Video>>()
   ...
   val playlist: LiveData<List<Video>>
     get() = _playlist
   ```

5. W `DevByteViewModel`klasie, po utworzeniu instancji `videosRepository`obiektu, dodaj nowy `val`wywołany `playlist`do przechowywania `LiveData`listy filmów z repozytorium.

   ```
   val playlist = videosRepository.videos
   ```

6. Uruchom swoją aplikację. Aplikacja działa jak poprzednio, ale teraz `DevBytes`playlista jest pobierana z sieci i zapisywana w bazie danych pokoju. Lista odtwarzania jest wyświetlana na ekranie z bazy danych Pokoi, a nie bezpośrednio z sieci.



7. Aby zauważyć różnicę, włącz tryb samolotowy w emulatorze lub urządzeniu.
8. Uruchom aplikację jeszcze raz. Zwróć uwagę, że komunikat toastowy „Błąd sieci” nie jest wyświetlany. Zamiast tego lista odtwarzania jest pobierana z pamięci podręcznej offline i wyświetlana.

9. Wyłącz tryb samolotowy w emulatorze lub urządzeniu.
10. Zamknij i ponownie otwórz aplikację. Aplikacja ładuje listę odtwarzania z pamięci podręcznej offline, podczas gdy żądanie sieciowe działa w tle.

    Jeśli z sieci nadejdą nowe dane, ekran zostanie automatycznie zaktualizowany, aby pokazać nowe dane. Jednak `DevBytes`serwer nie odświeża swojej zawartości, więc nie widzisz aktualizacji danych.

    Wskazówka: najłatwiejszym sposobem usunięcia pamięci podręcznej do testowania jest odinstalowanie aplikacji.

    Świetna robota! W tym ćwiczeniu z programowania zintegrowałeś pamięć podręczną w trybie offline z `ViewModel`wyświetlaniem listy odtwarzania z repozytorium zamiast pobierania listy odtwarzania z sieci.

7. Kod rozwiązania

Kod rozwiązania

Projekt Android Studio: RepositoryPattern

8. Gratulacje

Gratulacje! Na tej ścieżce nauczyłeś się:

- Buforowanie to proces przechowywania danych pobranych z sieci w pamięci urządzenia. Buforowanie umożliwia aplikacji dostęp do danych, gdy urządzenie jest w trybie offline lub gdy aplikacja musi ponownie uzyskać dostęp do tych samych danych.
- Najlepszym sposobem przechowywania przez Twoją aplikację danych strukturalnych w systemie plików urządzenia jest użycie lokalnej bazy danych SQLite. Room to biblioteka mapowania obiektów SQLite, co oznacza, że zapewnia warstwę abstrakcji w stosunku do SQLite. Korzystanie z pokoju to zalecana najlepsza praktyka wdrażania buforowania offline.
- Klasa repozytorium izoluje źródła danych, takie jak baza danych Room i usługi internetowe, od reszty aplikacji. Klasa repozytorium zapewnia czysty interfejs API do dostępu do danych do reszty aplikacji.
- Korzystanie z repozytoriów jest zalecaną najlepszą praktyką dotyczącą separacji kodu i architektury.
- Podczas projektowania pamięci podręcznej w trybie offline najlepszym rozwiązaniem jest oddzielenie obiektów sieci, domeny i bazy danych aplikacji. Ta strategia jest przykładem rozdzielenia obaw .

Ucz się więcej

- Architektura aplikacji
- Pomieszczenie
- Dane na żywo

Preferencje DataStore

1. Zanim zaczniesz

W poprzednich ćwiczeniach z programowania nauczyłeś się zapisywać dane w bazie danych SQLite za pomocą Room, warstwy abstrakcji bazy danych. To laboratorium programowania przedstawia Jetpack DataStore. Zbudowany w oparciu o współprogramy Kotlin i Flow, DataStore zapewnia dwie różne implementacje: Proto DataStore, która przechowuje wpisane obiekty, oraz Preferences DataStore, która przechowuje pary klucz-wartość.

W tym praktycznym laboratorium programowania nauczysz się korzystać z Preferences DataStore. Proto DataStore wykracza poza zakres tego ćwiczenia z programowania.

Warunki wstępne

- Znasz składniki architektury systemu Android `ViewModel`, `LiveData`i `Flow`i wiesz, jak używać `ViewModelProvider.Factory`do tworzenia wystąpienia `ViewModel`.
- Znasz podstawy współbieżności.
- Wiesz, jak używać współprogramów do długotrwałych zadań.

Czego się nauczysz

- Czym jest DataStore oraz dlaczego i kiedy warto z niego korzystać.
- Jak dodać Preference DataStore do swojej aplikacji.

Czego potrzebujesz

- Kod startowy aplikacji **Words** (jest taki sam jak kod rozwiązania aplikacji Words z poprzedniego ćwiczenia z programowania).
- Komputer z zainstalowanym Android Studio.

Pobierz kod startowy do tego ćwiczenia z programowania

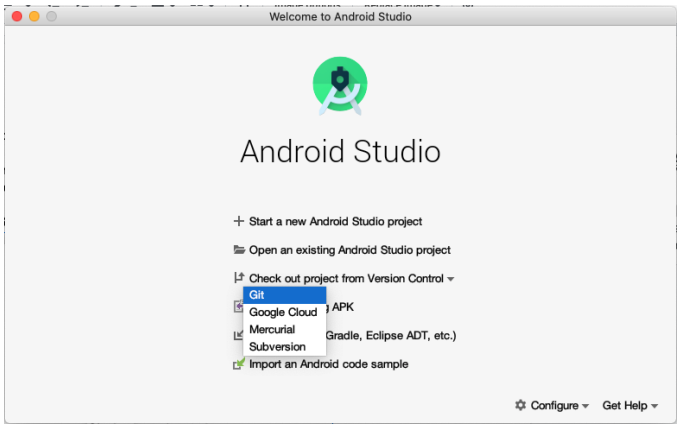
Podczas tego ćwiczenia z programowania rozszerzysz funkcje aplikacji Words z poprzedniego kodu rozwiązania. Kod startowy może zawierać kod, który jest Ci również znany z poprzednich ćwiczeń z programowania.

URL kodu startowego: https://github.com/google-developer-training/android-basics-kotlin-words-app/tree/main

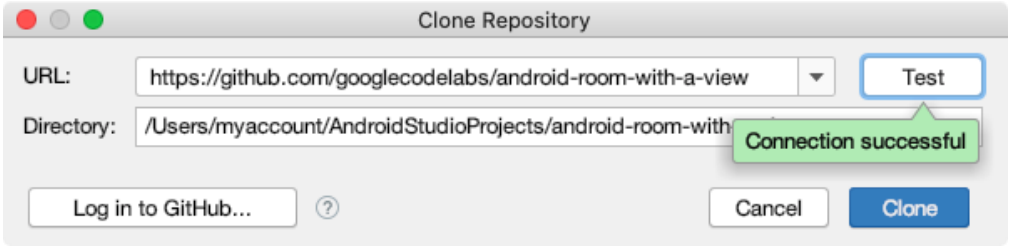
Oddział:`main`

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

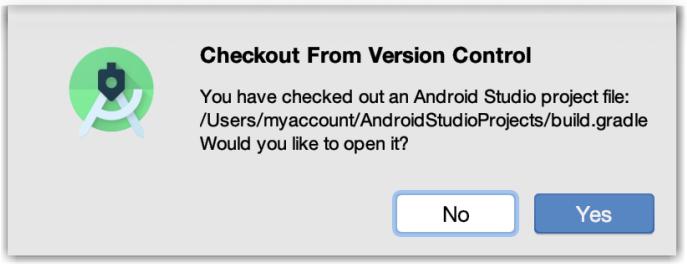
1. Uruchom Android Studio.
2. W oknie **Witamy w Android Studio** kliknij **Wyewidencjonuj projekt z kontroli wersji** .
3. Wybierz **Gita** .



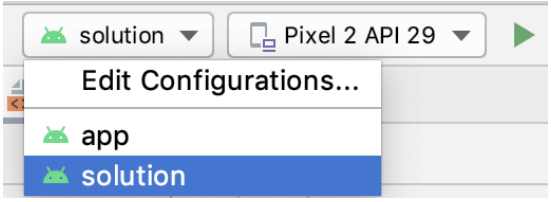
4. W oknie dialogowym **Clone Repository** wklej podany adres URL kodu w polu **adresu URL** .
5. Kliknij przycisk **Testuj** , poczekaj i upewnij się, że pojawia się zielony dymek z napisem **Połączenie udane** .
6. Opcjonalnie zmień **katalog** na inny niż sugerowany domyślny.



7. Kliknij **Klonuj** . Android Studio rozpocznie pobieranie Twojego kodu.
8. W wyskakującym okienku Pobierz **z kontroli wersji kliknij przycisk Tak** .



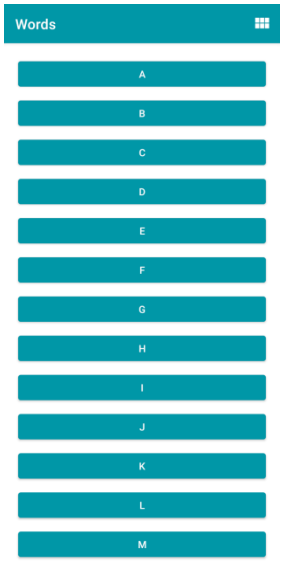
9. Poczekaj, aż otworzy się Android Studio.
10. Wybierz odpowiedni moduł dla kodu startowego lub rozwiązania codelab.

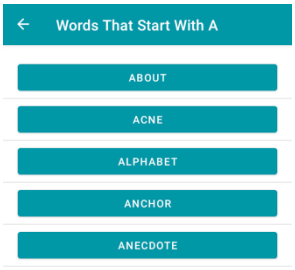


11. Kliknij przycisk **Uruchom**  , aby skompilować i uruchomić swój kod.

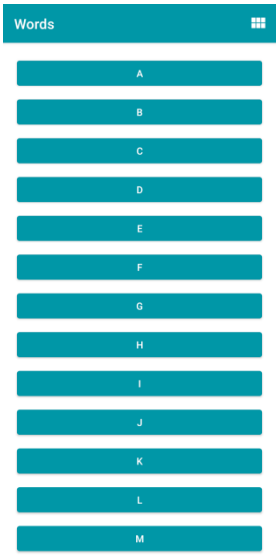
2. Przegląd aplikacji startowej

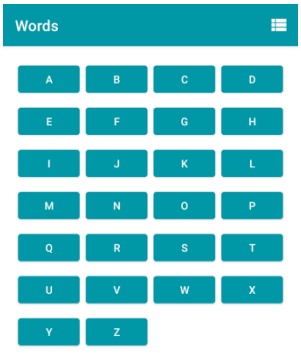
Aplikacja Words składa się z dwóch ekranów: Pierwszy ekran pokazuje litery, z których użytkownik może wybierać; drugi ekran wyświetla listę słów zaczynających się od wybranych liter.





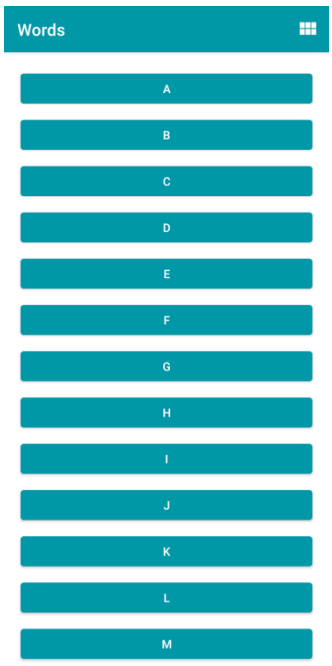
Ta aplikacja ma opcję menu, dzięki której użytkownik może przełączać się między układami listy i siatki dla liter.

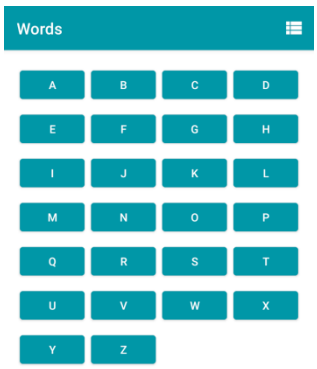


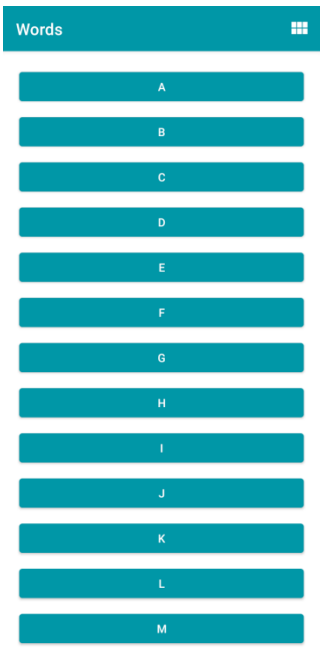


1. Pobierz kod startowy, otwórz go w Android Studio i uruchom aplikację. Litery są wyświetlane w układzie liniowym.
2. Stuknij opcję menu w prawym górnym rogu. Układ przełączy się na układ siatki.

3. Wyjdź i ponownie uruchom aplikację. Możesz to zrobić za pomocą opcji **Zatrzymaj „aplikację”** i **Uruchom „aplikację”** w Android Studio. Pamiętaj, że po ponownym uruchomieniu aplikacji litery są wyświetlane w układzie liniowym, a nie w siatce.







Zauważ, że wybór użytkownika nie jest zachowywany. W tym ćwiczeniu z programowania dowiesz się, jak rozwiązać ten problem.

Co zbudujesz

- W tym laboratorium programowania dowiesz się, jak używać Preferencji DataStore do utrwalania ustawienia układu w DataStore.

3. Wprowadzenie do Preferencji DataStore

Preferencje DataStore idealnie nadaje się do małych, prostych zestawów danych, takich jak przechowywanie danych logowania, ustawienie trybu ciemnego, rozmiar czcionki i tak dalej. DataStore nie jest odpowiedni dla złożonych zestawów danych, takich jak lista inwentaryzacyjna sklepu spożywczego online lub baza danych uczniów. Jeśli potrzebujesz przechowywać duże lub złożone zestawy danych, rozważ użycie funkcji Room zamiast DataStore.

Korzystając z biblioteki Jetpack DataStore możesz stworzyć proste, bezpieczne i asynchroniczne API do przechowywania danych. Zapewnia dwie różne implementacje: Preferences DataStore i

Proto DataStore. Chociaż zarówno Preferencje, jak i Proto DataStore umożliwiają zapisywanie danych, robią to na różne sposoby:

- **Preferencje DataStore** uzyskuje dostęp do danych i przechowuje je na podstawie kluczy, bez wcześniejszego definiowania schematu (modelu bazy danych).

- **Proto DataStore** definiuje schemat przy użyciu buforów protokołu . Korzystanie z buforów protokołów lub protobufów umożliwia **utrwalanie silnie wpisanych danych** . Protobufy są szybsze, mniejsze, prostsze i mniej niejednoznaczne niż XML i inne podobne formaty danych.

Pokój a Datastore: kiedy używać

Jeśli Twoja aplikacja musi przechowywać duże/złożone dane w formacie strukturalnym, takim jak SQL, rozważ użycie funkcji Room. Jeśli jednak potrzebujesz przechowywać tylko proste lub niewielkie ilości danych, które można zapisać w parach klucz-wartość, DataStore jest idealnym wyborem.

Proto a Preferencje DataStore: kiedy używać

Proto DataStore jest bezpieczny i wydajny, ale wymaga konfiguracji i konfiguracji. Jeśli dane Twojej aplikacji są na tyle proste, że można je zapisać w parach klucz-wartość, Preferencje DataStore są lepszym wyborem, ponieważ znacznie łatwiej je skonfigurować.

Dodaj Preferencje DataStore jako zależność

Pierwszym krokiem w integracji DataStore z Twoją aplikacją jest dodanie go jako zależności.

1. W `build.gradle(Module: Words.app)`Dodaj następującą zależność.

```
implementation "androidx.datastore:datastore-preferences:1.0.0"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.3.1"
```

4. Utwórz Preferencje DataStore

1. Dodaj pakiet o nazwie `data`i utwórz klasę Kotlin o nazwie `SettingsDataStore`wewnątrz niego.
2. Dodaj parametr konstruktora do `SettingsDataStore`klasy typu `Context`.

```
class SettingsDataStore(context: Context) {}
```

3. Poza `SettingsDataStore`klasą zadeklaruj `private const val`wywołane `LAYOUT_PREFERENCES_NAME`,i przypisz do niego wartość ciągu `layout_preferences`. Jest to nazwa magazynu danych preferencji, którego instancję utworzysz w następnym kroku.

```
private const val LAYOUT_PREFERENCES_NAME = "layout_preferences"
```

4. Nadal poza klasą utwórz `DataStore`instancję za pomocą `preferencesDataStore`delegata. Ponieważ korzystasz z Preferences Datastore, musisz przekazać `Preferences`jako typ datastore. Ponadto ustaw magazyn danych `name`na `LAYOUT_PREFERENCES_NAME`.

Uzupełniony kod:

```
private const val LAYOUT_PREFERENCES_NAME = "layout_preferences"

// Create a DataStore instance using the preferencesDataStore delegate, with the Context as
// receiver.
private val Context.dataStore : DataStore<Preferences> by preferencesDataStore(
  name = LAYOUT_PREFERENCES_NAME
```

)

Wskazówka: Utwórz raz instancję DataStore na najwyższym poziomie pliku Kotlin i uzyskaj do niej dostęp za pośrednictwem tej właściwości w pozostałej części aplikacji. Ułatwia to utrzymanie DataStore jako pojedynczego.

5. Implementuj klasę SettingsDataStore

Jak wspomniano, Preferences DataStore przechowuje dane w parach klucz-wartość. W tym kroku zdefiniujesz wymagane klawisze do przechowywania ustawień układu oraz zdefiniujesz funkcje do zapisu i odczytu w Preferencjach DataStore.

Funkcje typu klucza

Preferencje DataStore nie używa predefiniowanego schematu, takiego jak Room, używa odpowiednich funkcji typu klucza w celu zdefiniowania klucza dla każdej wartości przechowywanej w `DataStore<Preferences>`instancji. Na przykład, aby zdefiniować klucz dla `int`wartości, użyj `intPreferencesKey()`, a dla `string`wartości użyj `stringPreferencesKey()`. Ogólnie rzecz biorąc, te nazwy funkcji są poprzedzone typem danych, które chcesz przechowywać w kluczu.

Zaimplementuj w `data\SettingsDataStore`klasie:

1. Aby zaimplementować `SettingsDataStore`klasę, pierwszym krokiem jest utworzenie klucza przechowującego wartość logiczną określającą, czy ustawienie użytkownika jest układem liniowym. Utwórz `private`wywoływaną właściwość klasy `IS_LINEAR_LAYOUT_MANAGER`,i zainicjuj ją, *`booleanPreferencesKey()`*przekazując `is_linear_layout_manager`nazwę klucza jako parametr funkcji.

```
private val IS_LINEAR_LAYOUT_MANAGER = booleanPreferencesKey("is_linear_layout_manager")
```

Napisz do Preferencji DataStore

Teraz nadszedł czas, aby użyć klucza i zapisać ustawienie układu logicznego w `DataStore`. Preferencje DataStore udostępnia funkcję `edit()`zawieszenia, która transakcyjnie aktualizuje dane w programie `DataStore`. Parametr transformacji funkcji akceptuje blok kodu, w którym można aktualizować wartości zgodnie z potrzebami. Cały kod w bloku transformacji jest traktowany jako pojedyncza transakcja. Pod maską praca transakcyjna jest przeniesiona do `Dispacter.IO`, więc nie zapomnij utworzyć swojej funkcji `suspend`podczas jej wywoływania `edit()`.

1. Utwórz `suspend`funkcję o nazwie `saveLayoutToPreferencesStore()`, która przyjmuje dwa parametry: ustawienie układu Boolean i `Context`.

```
suspend fun saveLayoutToPreferencesStore(isLinearLayoutManager: Boolean, context: Context) {

}
```

2. Zaimplementuj powyższą funkcję, wywołaj i przekaż blok kodu, aby przechowywać nową wartość.*`dataStore`*`.edit(),`

```
suspend fun saveLayoutToPreferencesStore(isLinearLayoutManager: Boolean, context: Context) {
  context.dataStore.edit { preferences ->
```

```
        preferences[IS_LINEAR_LAYOUT_MANAGER] = isLinearLayoutManager
    }
}
```

Przeczytaj z Preferencji DataStore

Preferencje DataStore udostępnia dane przechowywane w a `Flow<Preferences>`, które są emitowane za każdym razem, gdy preferencja uległa zmianie. Nie chcesz eksponować całego `Preferences`obiektu, tylko `Boolean`wartość. Aby to zrobić, mapujemy `Flow<Preferences>`i uzyskujemy `Boolean`interesującą Cię wartość.

3. Odsłoń `preferenceFlow: Flow<UserPreferences>`, skonstruowany na podstawie `dataStore.data: Flow<Preferences>`, zmapuj go, aby pobrać `Boolean`preferencję. Ponieważ Datastore jest pusty przy pierwszym uruchomieniu, `true`domyślnie wróć.

```
val preferenceFlow: Flow<Boolean> = context.dataStore.data
    .map { preferences ->
        // On the first run of the app, we will use LinearLayoutManager by default
        preferences[IS_LINEAR_LAYOUT_MANAGER] ?: true
    }
```

4. Dodaj następujące importy, jeśli nie są importowane automatycznie.

```
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.catch
import kotlinx.coroutines.flow.map
```

Obsługa wyjątków

Ponieważ DataStore odczytuje i zapisuje dane z plików, `IOExceptions`może wystąpić podczas uzyskiwania dostępu do danych. Obsługujesz je za pomocą `catch()`operatora do przechwytywania wyjątków.

1. SharedPreference DataStore zgłasza `IOException`błąd podczas odczytywania danych. W `preferenceFlow`deklaracji, przed `map()`, użyj `catch()`operatora do przechwycenia `IOException`i emit `emptyPreferences()`. Aby uprościć sprawę, ponieważ nie oczekujemy tutaj żadnych innych typów wyjątków, jeśli zostanie zgłoszony inny typ wyjątku, zgłoś go ponownie.

```
val preferenceFlow: Flow<Boolean> = context.dataStore.data
    .catch {
        if (it is IOException) {
            it.printStackTrace()
            emit(emptyPreferences())
        } else {
            throw it
        }
    }
    .map { preferences ->
        // On the first run of the app, we will use LinearLayoutManager by default
        preferences[IS_LINEAR_LAYOUT_MANAGER] ?: true
```

```
}
```

Twoja `data\SettingsDataStore` klasa jest gotowa do użycia!

6. Użyj klasy SettingsDataStore

W następnym zadaniu będziesz używać `SettingsDataStore` w swojej `LetterListFragment` klasie. Dołączysz obserwatora do ustawień układu i odpowiednio zaktualizujesz interfejs.

Wykonaj następujące kroki w `LetterListFragment`

1. Zadeklaruj `private` zmienną klasy o nazwie `SettingsDataStore` type `SettingsDataStore`. Utwórz tę zmienną `lateinit`, ponieważ zainicjujesz ją później.

```
private lateinit var SettingsDataStore: SettingsDataStore
```

2. Na końcu `onViewCreated()` funkcji zainicjuj nową zmienną i przekaż ją `requireContext()` do `SettingsDataStore` konstruktora.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  ...
  // Initialize SettingsDataStore
  SettingsDataStore = SettingsDataStore(requireContext())
}
```

Przeczytaj i obserwuj dane

3. W `LetterListFragment`, wewnątrz `onViewCreated()` metody, poniżej `SettingsDataStore` inicjalizacji, przekonwertuj `preferenceFlow` na `Livedata` using . Dołącz obserwatora i przekaż jako właściciel. *`asLiveData()viewLifecycleOwner`*

```
SettingsDataStore.preferenceFlow.asLiveData().observe(viewLifecycleOwner, { })
```

4. Wewnątrz obserwatora przypisz nowe ustawienie układu do `isLinearLayoutManager` zmiennej. Wywołaj `chooseLayout()` funkcję, aby zaktualizować układ RecyclerView.

```
SettingsDataStore.preferenceFlow.asLiveData().observe(viewLifecycleOwner, { value ->
        isLinearLayoutManager = value
        chooseLayout()
  })
```

Ukończona `onViewCreated()` funkcja powinna wyglądać podobnie jak poniżej:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  recyclerView = binding.recyclerView
```

```
    // Initialize SettingsDataStore
    SettingsDataStore = SettingsDataStore(requireContext())
    SettingsDataStore.preferenceFlow.asLiveData().observe(viewLifecycleOwner, { value ->
        isLinearLayoutManager = value
        chooseLayout()
    })
}
```

Zapisz ustawienia układu w DataStore

Ostatnim krokiem jest zapisanie ustawienia układu w Preferencjach DataStore, gdy użytkownik dotknie opcji menu. Zapisywanie danych w magazynie danych preferencji powinno być wykonywane asynchronicznie wewnątrz współprogramu. Aby wykonać to wewnątrz fragmentu, użyj `CoroutineScope`wywołania `LifecycleScope`.

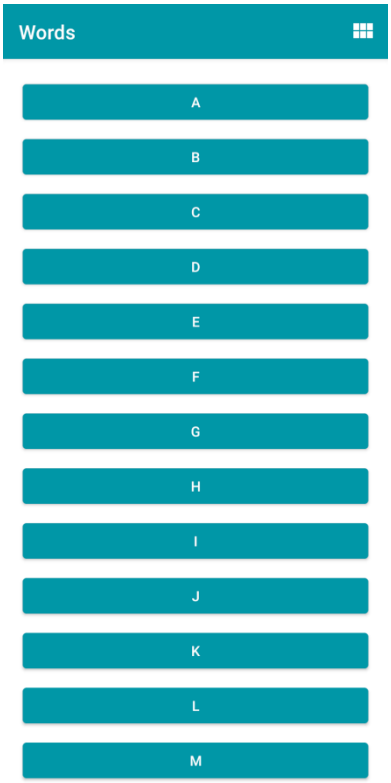
Zakres cyklu życia

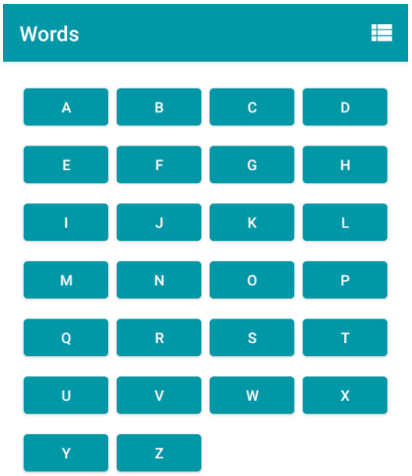
Składniki zgodne z cyklem życia, takie jak fragmenty, zapewniają pierwszorzędną współprogramową obsługę zakresów logicznych w Twojej aplikacji wraz z warstwą współdziałania z `LiveData`. A `LifecycleScope`jest zdefiniowane dla każdego `Lifecycle`obiektu. Wszelkie współprogramy uruchomione w tym zakresie są anulowane, gdy `Lifecycle`właściciel zostanie zniszczony.

1. W funkcji `LetterListFragment`wewnętrznej `onOptionsItemSelected()`, na końcu sprawy `R.id.action_switch_layout`uruchom współprogram za pomocą *lifecycleScope*. Wewnątrz `launch`bloku zadzwoń do `saveLayoutToPreferencesStore()`przechodniów w `isLinearLayoutManager`i `context`.

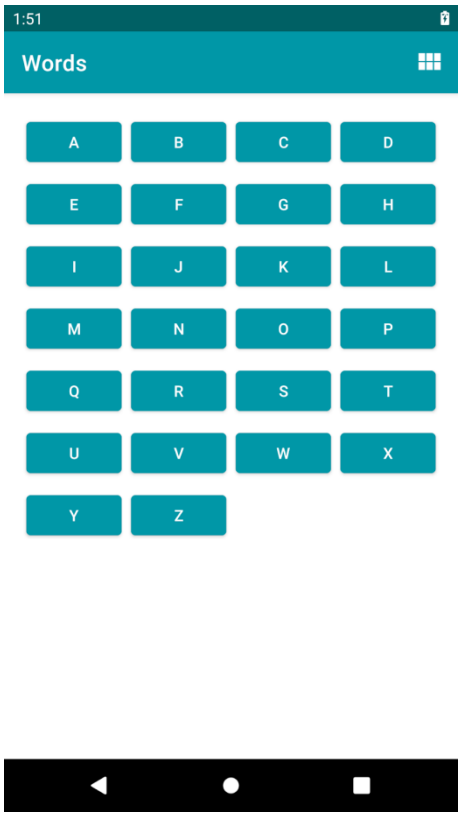
```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_switch_layout -> {
            ...
            // Launch a coroutine and write the layout setting in the preference Datastore
            lifecycleScope.launch
{               SettingsDataStore.saveLayoutToPreferencesStore(isLinearLayoutManager, requireContext())
            }
            ...

            return true
        }
```

2. Uruchom aplikację. Kliknij opcję menu, aby zmienić układ aplikacji.





3. Teraz przetestuj trwałość Preferencji DataStore. Zmień układ aplikacji na układ siatki. Wyjdź i uruchom ponownie aplikację (możesz to zrobić za pomocą opcji **Zatrzymaj „aplikację”**  i **Uruchom „aplikację”**  w Android Studio).



Po ponownym uruchomieniu aplikacji litery są teraz wyświetlane w układzie siatki, a nie w układzie liniowym. Twoja aplikacja pomyślnie zapisuje ustawienia układu wybrane przez użytkownika!

Zauważ, że chociaż litery są teraz wyświetlane w układzie siatki, ikona menu nie jest poprawnie aktualizowana. Następnie przyjrzymy się, jak rozwiązać ten problem.

7. Napraw błąd ikony menu

Powodem błędu ikony menu jest to, `onViewCreated()`że układ RecyclerView jest aktualizowany zgodnie z ustawieniem układu, ale nie ikona menu. Ten problem można rozwiązać, przerysowując menu wraz z aktualizacją układu RecyclerView.

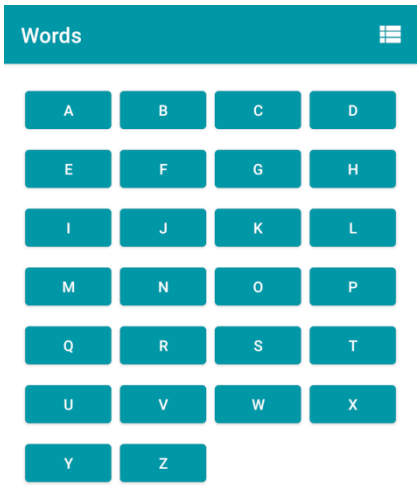
Przerysowanie menu opcji

Po utworzeniu menu nie jest ono przerysowywane w każdej klatce, ponieważ przerysowywanie tego samego menu w każdej klatce byłoby zbędne. Funkcja *invalidateOptionsMenu()*mówi Androidowi, aby przerysował menu opcji.
Możesz wywołać tę funkcję, gdy zmienisz coś w menu Opcje, na przykład dodasz element menu, usuniesz element lub zmienisz tekst lub ikonę menu. W tym przypadku zmieniono ikonę menu. Wywołanie tej metody deklaruje, że menu Options uległo zmianie i dlatego powinno zostać odtworzone. Metoda onCreateOptionsMenu(android.view.Menu) jest wywoływana przy następnym wyświetleniu.

1. W `LetterListFragment`, wewnątrz `onViewCreated()`, na końcu `preferenceFlow`obserwatora, poniżej wywołanie `chooseLayout()`. Przerysuj menu, wywołując *invalidateOptionsMenu()*. *activity*

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  ...
  SettingsDataStore.preferenceFlow.asLiveData().observe(viewLifecycleOwner, { value ->
        ...
        // Redraw the menu
        activity?.invalidateOptionsMenu()
  })
}
```

2. Uruchom aplikację ponownie i zmień układ.
3. Wyjdź i uruchom ponownie aplikację. Zauważ, że ikona menu jest teraz poprawnie zaktualizowana.



Gratulacje! Pomyślnie dodałeś Preferences DataStore do swojej aplikacji, aby zapisać wybór użytkownika.

8. Kod rozwiązania

Kod rozwiązania dla tego ćwiczenia programowania znajduje się w projekcie i module pokazanym poniżej.

Adres URL kodu rozwiązania: https://github.com/google-developer-training/android-basics-kotlin-words-app/tree/datastore

9. Podsumowanie

- DataStore posiada w pełni asynchroniczne API wykorzystujące współprogramy Kotlin i Flow, gwarantujące spójność danych.
- Jetpack DataStore to rozwiązanie do przechowywania danych, które umożliwia przechowywanie par klucz-wartość lub wpisanych obiektów za pomocą buforów protokołów .
- DataStore udostępnia dwie różne implementacje: Preferences DataStore i Proto DataStore.
- Preferencje DataStore nie używa wstępnie zdefiniowanego schematu.

- Preferencje DataStore używa odpowiedniej funkcji typu klucza w celu zdefiniowania klucza dla każdej wartości, którą należy przechowywać w `DataStore<Preferences>`instancji. Na przykład, aby zdefiniować klucz dla `int`wartości, użyj `intPreferencesKey()`.
- Preferencje DataStore udostępnia `edit()`funkcję, która transakcyjnie aktualizuje dane w pliku `DataStore`.

10. Dowiedz się więcej

- `DataStore` `guide`
- Odniesienie do DataStore
- Preferencje
- android.datastore.preferences.core

Blog
Preferuj przechowywanie danych za pomocą Jetpack DataStore

Projekt: Aplikacja paszy

1. Zanim zaczniesz

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

Warunki wstępne

- Ten projekt jest przeznaczony dla studentów, którzy ukończyli 5 część kursu Android Basics in Kotlin.

Co zbudujesz

- Dodaj trwałość z Room do istniejącej aplikacji, implementując encję, DAO, ViewModel i klasę bazy danych.

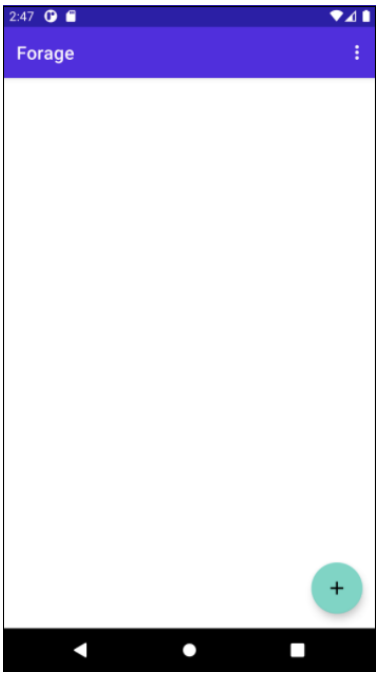
Co będziesz potrzebował

- Komputer z zainstalowanym Android Studio.

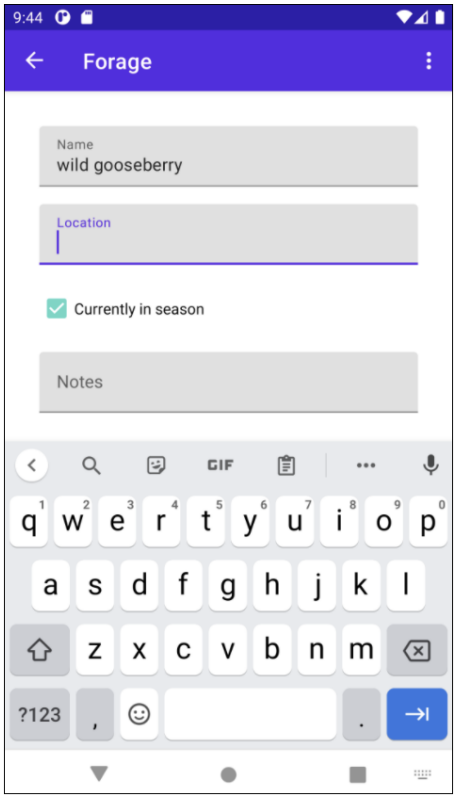
2. Zakończony przegląd aplikacji

Kompletna aplikacja Forage pozwala użytkownikom śledzić przedmioty, na przykład jedzenie, które szukali w naturze. Te dane są utrwalane między sesjami przy użyciu pokoju. Wykorzystasz swoją wiedzę o pokoju i wykonywaniu operacji odczytu, zapisu, aktualizacji i usuwania w bazie danych, aby zaimplementować trwałość w aplikacji Forage. Gotową aplikację i jej funkcjonalność opisano poniżej.

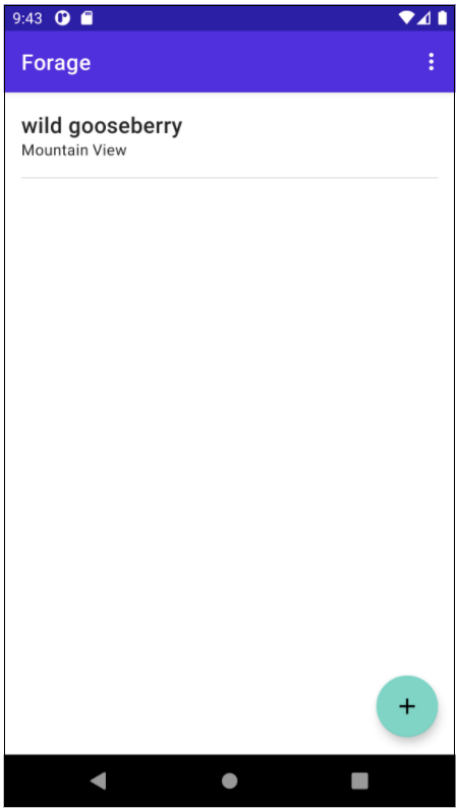
Gdy aplikacja jest uruchamiana po raz pierwszy, użytkownik jest prezentowany z pustym ekranem zawierającym widok recyklera, który wyświetla zebrane przedmioty, a także pływający przycisk w prawym dolnym rogu, aby dodać nowe przedmioty.



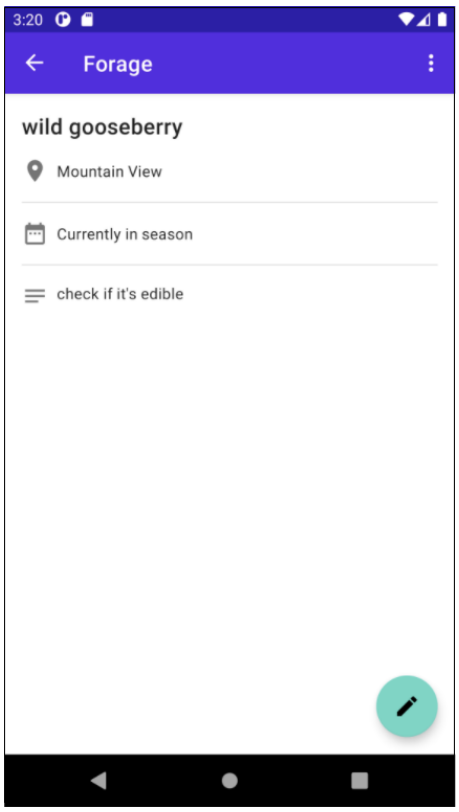
Dodając nową pozycję, użytkownik może podać nazwę, miejsce, w którym została odnaleziona, a także kilka dodatkowych uwag. Istnieje również pole wyboru, czy produkt spożywczy jest aktualnie w sezonie.



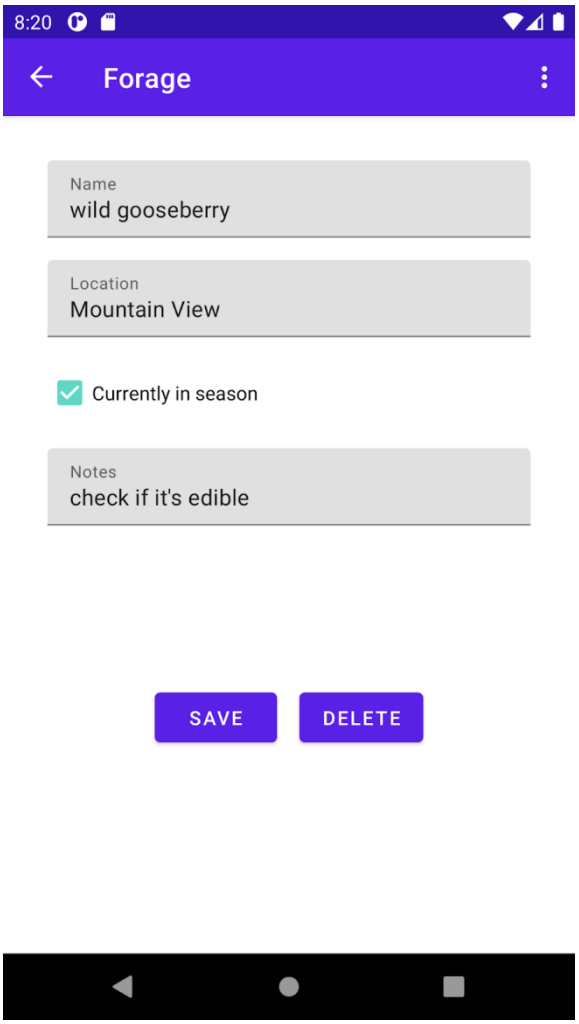
Po dodaniu elementu pojawi się on w widoku recyklera na pierwszym ekranie.



Dotknięcie elementu prowadzi do ekranu szczegółów, który pokazuje nazwę, lokalizację i notatki.



Pływający przycisk również zmieni się z symbolu plusa w ikonę edycji. Naciśnięcie tego przycisku prowadzi do ekranu, który umożliwia edycję nazwy, lokalizacji, notatek i pola wyboru „w sezonie”. Naciśnięcie przycisku usuń usunie element z bazy danych.



Chociaż część interfejsu użytkownika tej aplikacji została już zaimplementowana, Twoim zadaniem jest zaimplementowanie trwałości przy użyciu Twojej wiedzy o pokoju, tak aby aplikacja odczytywała, zapisywała, aktualizowała i usuwała elementy z bazy danych.

3. Rozpocznij

Pobierz kod projektu

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

Adres URL kodu startowego:

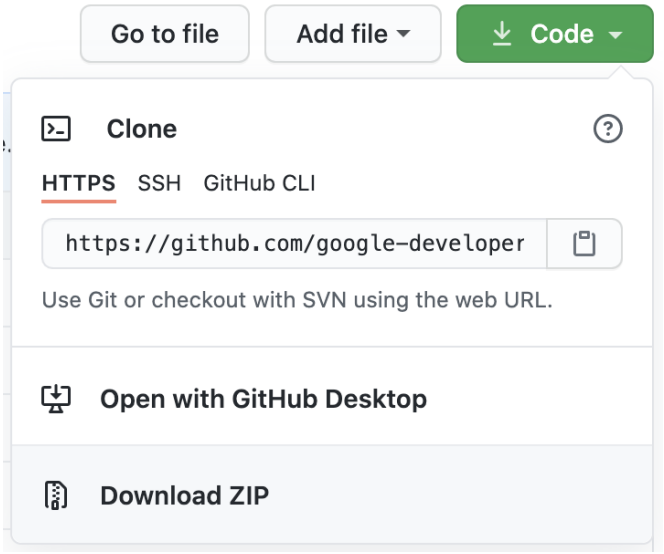
https://github.com/google-developer-training/android-basics-kotlin-forage-app/tree/main

Nazwa oddziału z kodem startowym: `main`

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

Pobierz kod

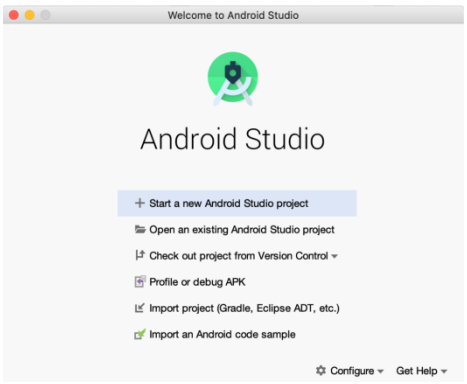
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



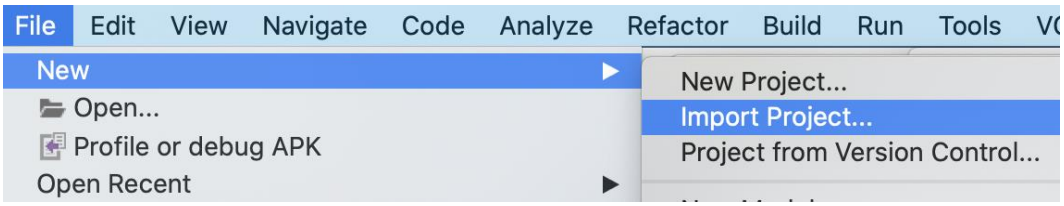
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego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

4. Skonfiguruj projekt do korzystania z pokoju

Zdefiniuj istotę zdatną do pożywienia

Projekt ma już `Forageable`klasę, która definiuje dane aplikacji (**model.Forageable.kt**). Ta klasa ma kilka właściwości: `id`, `name`, `address`, `inSeason`i `notes`.

```
data class Forageable(
  val id: Long = 0,
  val name: String,
  val address: String,
  val inSeason: Boolean,
  val notes: String?
)
```

Aby jednak użyć tej klasy do przechowywania trwałych danych, musisz przekonwertować ją na encję Room.

1. Opisz klasę za `@Entity`pomocą nazwy tabeli `"forageable_database"`.
2. Ustaw `id`właściwość jako klucz podstawowy. Klucz podstawowy powinien być generowany automatycznie.
3. Ustaw nazwę kolumny dla `inSeason`właściwości na `"in_season"`.

Wskazówka: aby się odświeżyć, wróć do instrukcji tworzenia elementów pokoju .

Implementuj DAO

`ForageableDao`(**data.ForageableDao.kt**), jak można się domyślić, to miejsce, w którym definiujesz metody odczytu i zapisu z bazy danych, do których będziesz mieć dostęp z modelu widoku. Ponieważ DAO jest tylko interfejsem, który definiujesz, w rzeczywistości nie będziesz

musiał pisać żadnego kodu, aby zaimplementować te metody. Zamiast tego należy użyć adnotacji do pomieszczeń, określając w razie potrzeby zapytanie SQL.

W `ForageableDao`interfejsie musisz dodać pięć metod.

1. Metoda `getForageables()`zwracająca a `Flow<List<Forageable>>`dla wszystkich wierszy w bazie danych.
2. Metoda `getForageable(id: Long)`, która zwraca a `Flow<Forageable>`, która pasuje do określonego `id`.
3. Metoda , `insert(forageable: Forageable)`która wstawia nowy `Forageable`do bazy danych.
4. Metoda , `update(forageable: Forageable)`która przyjmuje istniejący `Forageable`jako parametr i odpowiednio aktualizuje wiersz.
5. Metoda `delete(forageable: Forageable)`, która przyjmuje `Forageable`jako parametr i usuwa go z bazy danych.

Wskazówka: Jeśli potrzebujesz odświeżenia, zapoznaj się z instrukcjami definiowania elementu DAO w aplikacji Inventory.

Zaimplementuj model widoku

(`ForageableViewModel`ui.viewmodel.ForageableViewModel.kt **)** jest częściowo zaimplementowany, ale będziesz musiał dodać funkcjonalność, która uzyskuje dostęp do metod DAO, aby faktycznie mogła odczytywać i zapisywać dane. Wykonaj następujące kroki, aby zaimplementować `ForageableViewModel`.

1. Instancję `ForageableDao`należy przekazać jako parametr w konstruktorze klasy.
2. Utwórz zmienną typu `LiveData<List<Forageable>>`, która pobiera całą listę `Forageable`jednostek korzystających z DAO i konwertuje wynik na `LiveData`.
3. Utwórz metodę, która przyjmuje identyfikator (typu `Long`) jako parametr i zwraca a `LiveData<Forageable>`z wywołania `getForageable()`metody w DAO i przekonwertowania wyniku na `LiveData`.
4. W `addForageable()`metodzie uruchom współprogram za pomocą `viewModelScope`i użyj DAO, aby wstawić `Forageable`instancję do bazy danych.
5. W `updateForageable()`metodzie użyj DAO, aby zaktualizować `Forageable`jednostkę.
6. W `deleteForageable()`metodzie użyj DAO, aby zaktualizować `Forageable`jednostkę.
7. Utwórz, `ViewModelFactory`który może utworzyć instancję `ForageableViewModel`z `ForageableDao`parametrem konstruktora.

Zaimplementuj klasę Database

Klasa `ForageDatabase`(`data.ForageDatabase.kt`) jest tym, co faktycznie eksponuje twoje encje i DAO w Room. Zaimplementuj `ForageDatabase`klasę zgodnie z opisem.

1. Podmioty:`Forageable`
2. Wersja:`1`
3. exportSchemat:`false`
4. Wewnątrz `ForageDatabase`klasy dołącz abstrakcyjną funkcję, aby zwrócić a`ForageableDao`
5. Wewnątrz `ForageDatabase`klasy zdefiniuj obiekt towarzyszący z wywoływaną zmienną prywatną `INSTANCE`i `getDatabase()`funkcją zwracającą `ForageDatabase`singleton.

Wskazówka: Konfiguracja klasy bazy danych powinna przebiegać dokładnie tak samo, jak w dwóch poprzednich aplikacjach z jednostki 5. Jeśli potrzebujesz odświeżenia, zapoznaj się z instrukcjami tworzenia klasy bazy danych .

6. W `BaseApplication`klasie utwórz `database`właściwość, która zwraca `ForageDatabase`instancję przy użyciu inicjalizacji z opóźnieniem.

5. Utrzymuj i czytaj dane z fragmentów

Po skonfigurowaniu encji, obiektów DAO, modelu widoku i zdefiniowaniu klasy bazy danych w celu udostępnienia ich w Room, wystarczy tylko zmodyfikować fragmenty, aby uzyskać dostęp do modelu widoku. Musisz wprowadzić zmiany w trzech plikach, po jednym dla każdego ekranu w aplikacji.

Lista roślin pastewnych

Ekran listy zielonki wymaga tylko dwóch rzeczy: odniesienia do modelu widoku i dostępu do pełnej listy zielonki. Wykonaj następujące zadania w **ui.ForageableListFragment.kt** .

1. Klasa ma już `viewModel`właściwość. Nie jest to jednak użycie fabryki zdefiniowanej w poprzednim kroku. Musisz najpierw zmienić tę deklarację, aby użyć `ForageableViewModelFactory`.

```
private val viewModel: ForageableViewModel by activityViewModels {
  ForageableViewModelFactory(
    (activity?.application as BaseApplication).database.foragableDao()
  )
}
```

2. Następnie w `onViewCreated()`, obserwuj `allForageables`właściwość z `viewModel`i wywołaj `submitList()`adapter w razie potrzeby, aby wypełnić listę.

Ekran informacji o zbieraniu pokarmu

Zrobisz prawie to samo dla listy szczegółów w **ui/ForageableDetailFragment.kt** .

1. Przekonwertuj `viewModel`właściwość, aby poprawnie zainicjować `ForageableViewModelFactory`.
2. W programie `onViewCreated()`wywołaj `getForageable()`model widoku, przekazując `id`, aby uzyskać `Forageable`encję. Obserwuj dane na żywo i ustaw wynik na `forageable`właściwość, a następnie wywołaj `bindForageable()`aktualizację interfejsu użytkownika.

Ekran dodawania i edytowania roślin pastewnych

Na koniec musisz zrobić podobną rzecz w **ui.AddForageableFragment.kt** . Zauważ, że ten ekran jest również odpowiedzialny za aktualizowanie i usuwanie encji. Jednak te metody z modelu widoku są już wywoływane we właściwym miejscu. Będziesz musiał wprowadzić tylko dwie zmiany w tym pliku.

1. Ponownie dokonaj refaktoryzacji `viewModel`właściwości do użycia `ForageableViewModelFactory`.
2. W `onViewCreated()`bloku instrukcji if przed ustawieniem widoczności przycisku usuwania wywołaj `getForageable()`model widoku, przekazując `id`i ustawiając wynik na `forageable`właściwość.

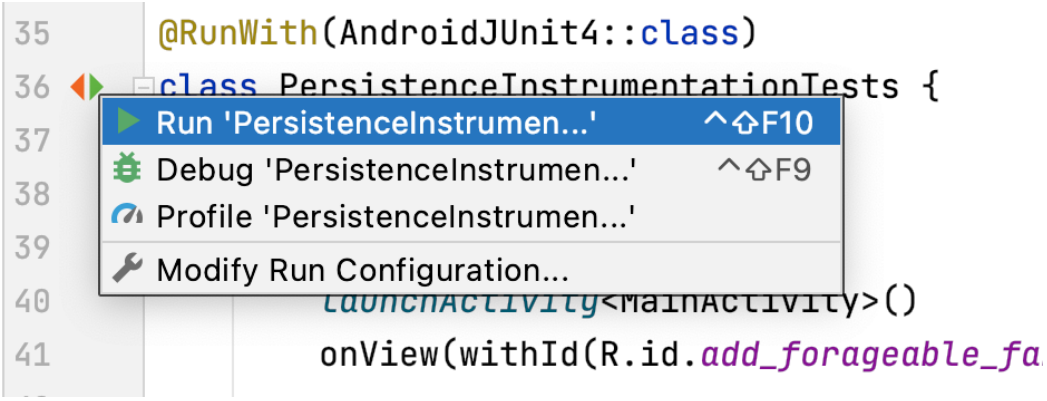
To wszystko, co musisz zrobić we fragmentach. Możesz teraz uruchomić swoją aplikację i powinieneś widzieć wszystkie funkcje utrwalania w akcji.

6. Instrukcje testowania

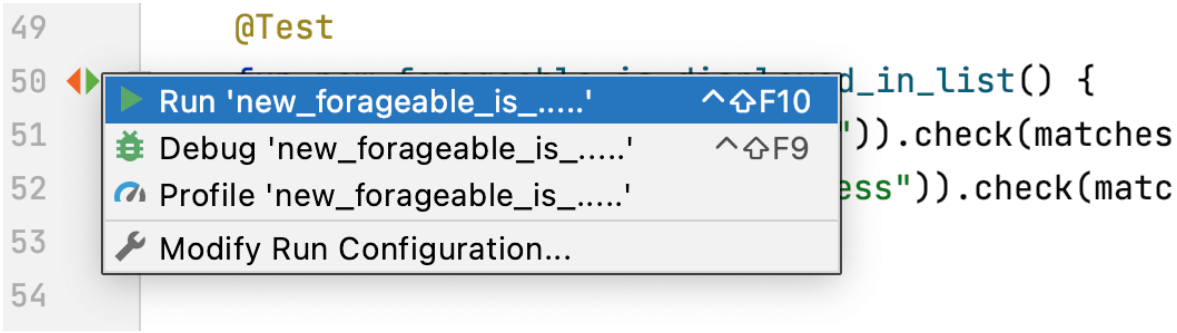
Przeprowadzanie testów

Aby uruchomić testy, możesz wykonać jedną z poniższych czynności.

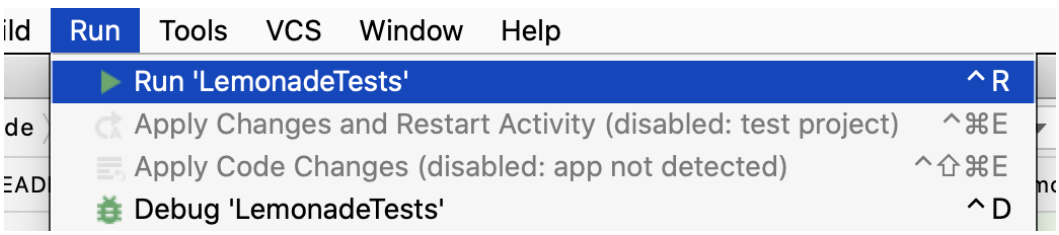
W przypadku pojedynczego przypadku testowego otwórz klasę przypadku testowego `PersistenceInstrumentationTests.kt` i kliknij zieloną strzałkę po lewej stronie deklaracji klasy. Następnie możesz wybrać z menu opcję Uruchom. Spowoduje to uruchomienie wszystkich testów w przypadku testowym.



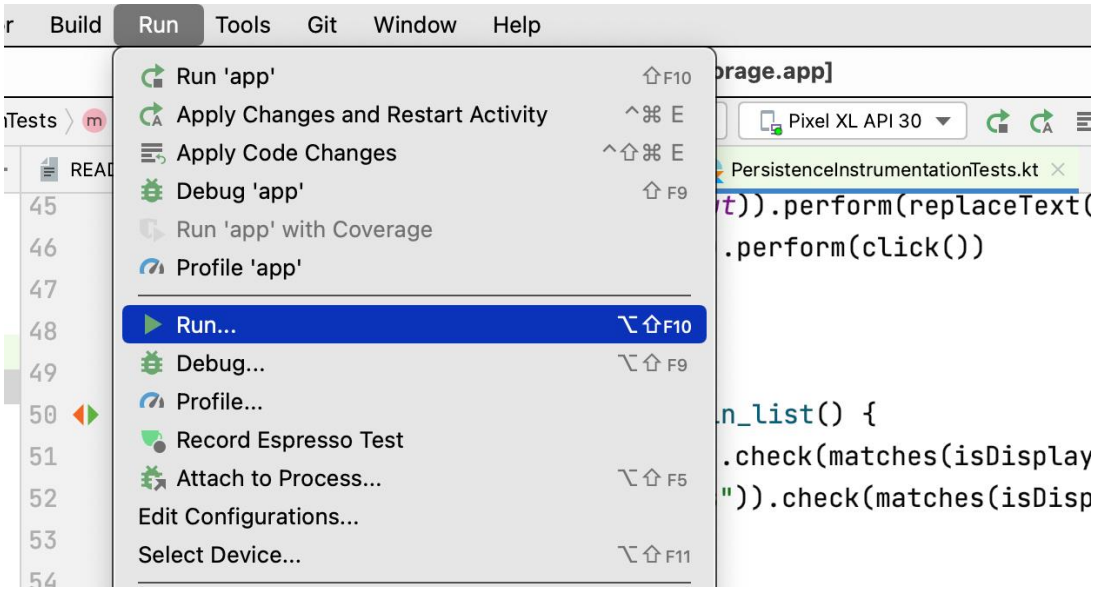
Często będziesz chciał uruchomić tylko jeden test, na przykład, jeśli tylko jeden test się nie powiedzie, a pozostałe testy zakończą się pomyślnie. Pojedynczy test można uruchomić tak samo, jak cały przypadek testowy. Użyj zielonej strzałki i wybierz opcję **Uruchom** .



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.