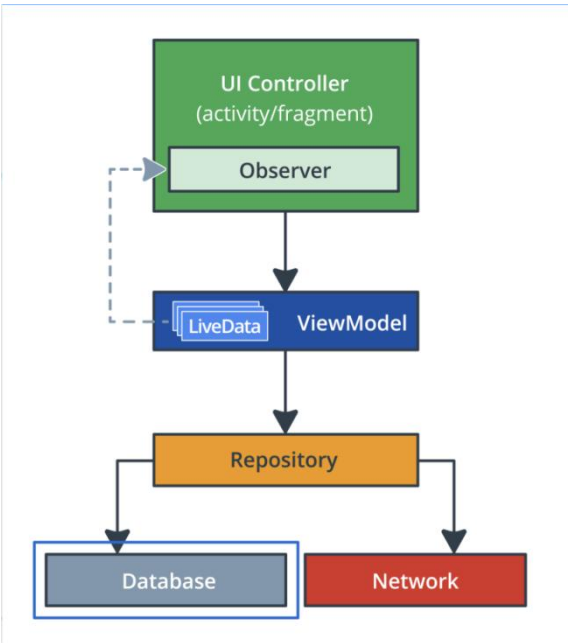
1. Wprowadzenie

Większość aplikacji zawiera dane, które należy przechowywać, nawet po zamknięciu aplikacji przez użytkownika. Na przykład aplikacja może przechowywać listę odtwarzania, spis elementów gry, zapisy wydatków i dochodów, katalog konstelacji lub dane dotyczące snu w czasie. Zwykle baza danych służy do przechowywania trwałych danych.

`Room` to biblioteka bazy danych, która jest częścią Androida Jetpack. `Room` zajmuje się wieloma obowiązkami związanymi z konfigurowaniem i konfigurowaniem bazy danych i umożliwia aplikacji interakcję z bazą danych za pomocą zwykłych wywołań funkcji. Pod maską Room to warstwa abstrakcji na bazie danych SQLite. Terminologia `Room`'s i składnia zapytań dla bardziej złożonych zapytań są zgodne z modelem SQLite..

Poniższy obraz pokazuje, jak baza danych `Room` pasuje do ogólnej zalecanej architektury.



Co powinieneś już wiedzieć

Po ostatnich ćwiczeniach powinieneś znać:
• Budowanie podstawowego interfejsu użytkownika (UI) dla aplikacji na Androida
• Korzystanie z działań, fragmentów i widoków.
• Nawigacja między fragmentami i używanie Bezpiecznych Argumentów (wtyczki Gradle) do przesyłania danych między fragmentami.
•Korzystanie z wzorców View model, view-model factory oraz LiveData i jej obserwatorów.

Czego się nauczysz

- Jak utworzyć bazę danych `Room` i współpracować z nią, aby utrwalić dane.
- Jak utworzyć klasę danych, która definiuje tabelę w bazie danych.
- Jak używać obiektu dostępu do danych (DAO) do mapowania funkcji Kotlin na zapytania SQL.

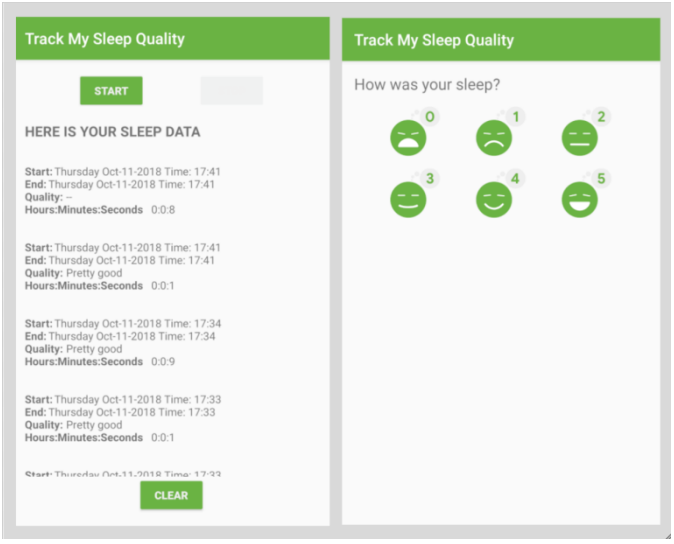
- Jak sprawdzić, czy baza danych działa.

Co będziesz robić

- Utwórz bazę danych `Room` z interfejsem dla danych dotyczących snu.
- Przetestuj bazę danych za pomocą dostarczonych testów.

2. App overview

Aplikacja ma dwa ekrany reprezentowane przez fragmenty, jak pokazano na poniższym rysunku.



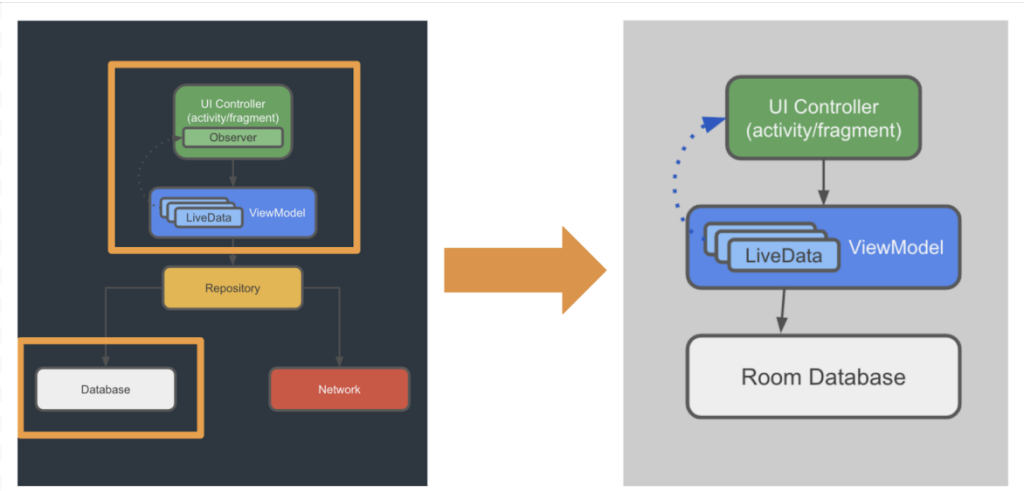
Pierwszy ekran, pokazany po lewej stronie, zawiera przyciski do uruchamiania i zatrzymywania śledzenia. Ekran pokazuje wszystkie dane dotyczące snu użytkownika. Przycisk Wyczyść trwale usuwa wszystkie dane zebrane przez użytkownika dla aplikacji. Drugi ekran, pokazany po prawej, służy do wyboru oceny jakości snu. W aplikacji ocena jest reprezentowana numerycznie. Dla celów programistycznych aplikacja wyświetla zarówno ikony twarzy, jak i ich numeryczne odpowiedniki.

“user's flow” jest następujący:

- Użytkownik otwiera aplikację i wyświetla się ekran śledzenia snu.
- Użytkownik naciska przycisk Start. Rejestruje czas rozpoczęcia i wyświetla go. Przycisk Start jest wyłączony, a przycisk Stop jest włączony.
- Użytkownik naciska przycisk Stop. Rejestruje czas zakończenia i otwiera ekran jakości snu.
- Użytkownik wybiera ikonę jakości snu. Ekran zamyka się, a ekran śledzenia wyświetla czas zakończenia i jakość snu. Przycisk Stop jest wyłączony, a przycisk Start jest włączony. Aplikacja jest gotowa na kolejną noc.
- Przycisk Wyczyść jest włączony, ilekroć w bazie danych znajdują się dane. Gdy użytkownik stuknie przycisk Wyczyść, wszystkie jego dane zostaną usunięte bez konieczności odwoływania się - nie ma pytania „Jesteś pewien?” wiadomość.

Aplikacja korzysta z uproszczonej architektury, jak pokazano poniżej w kontekście pełnej architektury. Aplikacja wykorzystuje tylko następujące komponenty:

- UI controller
- View model i `LiveData`
- Baza danych Room



3. Zadanie: Pobierz i sprawdź aplikację startową

Step 1: Pobierz i uruchom aplikację startową

1. Pobierz aplikację.
2. Zbuduj i uruchom aplikację. Aplikacja wyświetla interfejs użytkownika dla fragmentu SleepTrackerFragment, ale nie zawiera danych. Przyciski nie reagują na stuknięcia.

Step 2: Sprawdź aplikację startową

Wskazówka: Zapoznanie się z aplikacją startową ułatwi identyfikację i naprawę problemów, jeśli na nie natrafisz.

1. Spójrz na pliki Gradle:

- **Plik Gradle projektu**
  W pliku build.gradle na poziomie projektu zwróć uwagę na zmienne określające wersje bibliotek. Wersje używane w aplikacji startowej działają dobrze razem i działają dobrze z tą aplikacją. Zanim skończysz ćwiczenie, Android Studio może poprosić o aktualizację niektórych wersji. Od Ciebie zależy, czy chcesz aktualizować, czy pozostać przy wersjach zawartych w aplikacji.
- Plik modułu Gradle. Zwróć uwagę na podane zależności dla wszystkich bibliotek Android Jetpack, w tym `Room`, oraz zależności dla coroutines.

2. Spójrz na pakiety i interfejs użytkownika**.** Aplikacja ma funkcjonalną strukturę. Pakiet zawiera pliki “placeholder” , w których dodasz kod w całej serii zadań.

- **The `database` package**, Pakiet bazy danych dla całego kodu związanego z bazą danych `Room`.
- **Pakiety `sleepquality` i `sleeptracker`** zawierają fragment, view model, i view model factory dla każdego ekranu.

3. Spójrz na plik **`Util.kt` file**, , który zawiera funkcje pomocne w wyświetlaniu danych o jakości snu. Część kodu jest komentowana, ponieważ odwołuje się do view model, który utworzysz później
4. Spójrz na folder **androidTest (`SleepDatabaseTest.kt`).** Ten test posłuży do sprawdzenia, czy baza danych działa zgodnie z przeznaczeniem.

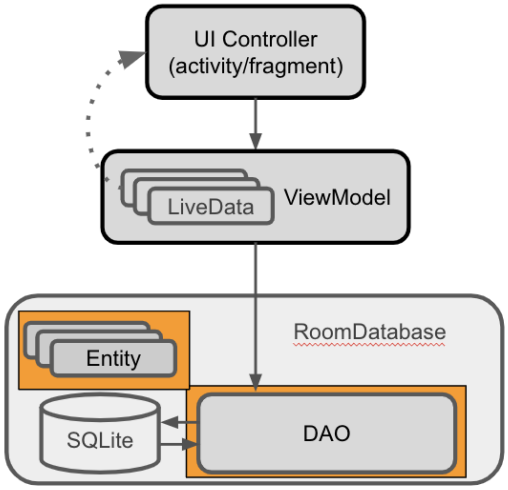
4. Zadanie: Utwórz obiekt SleepNight

W Androidzie dane są reprezentowane w klasach danych, a dane są dostępne i modyfikowane za pomocą wywołań funkcji. Jednak w świecie baz danych potrzebujesz encji i zapytań (*entities* i *queries*).

- *entity* reprezentuje obiekt lub koncepcję i jej właściwości do przechowywania w bazie danych. Klasa encji definiuje tabelę, a każda instancja tej klasy reprezentuje wiersz w tabeli. Każda właściwość definiuje kolumnę. W Twojej aplikacji jednostka będzie przechowywać informacje o nocy snu.
- *query* Zapytanie to żądanie danych lub informacji z tabeli bazy danych lub kombinacji tabel lub żądanie wykonania działania na danych. Typowe zapytania dotyczą pobierania, wstawiania i aktualizowania jednostek. Na przykład możesz zapytać o wszystkie zarejestrowane noce snu posortowane według godziny rozpoczęcia.

`Room` wykonuje całą ciężką pracę, aby uzyskać od klas Kotlin danych do encji, które mogą być przechowywane w tabelach SQLite, od deklaracji funkcji do zapytań SQL..

Musisz zdefiniować każdą jednostkę jako adnotowaną klasę danych, a interakcje jako adnotowany interfejs, obiekt dostępu do danych *data access object (DAO)*. `Room` używa tych klas z adnotacjami do tworzenia tabel w bazie danych i zapytań, które działają na bazie danych.



Step 1: Utwórz encję SleepNight (SleepNight entity)

W tym zadaniu definiujesz jedną noc snu jako klasę danych z adnotacjami.

Na jedną noc snu musisz zapisać czas rozpoczęcia, czas zakończenia i ocenę jakości.

Potrzebujesz identyfikatora ID, aby jednoznacznie zidentyfikować noc.

1. W pakiecie `database` znajdź i otwórz plik `SleepNight.kt`.
2. Utwórz klasę danych `SleepNight` z parametrami identyfikatora, czasem rozpoczęcia (w milisekundach), czasem zakończenia (w milisekundach) i liczbową oceną jakości snu.

- Musisz zainicjować funkcję `sleepQuality`, więc ustaw ją na -1, wskazując, że nie zebrano żadnych danych dotyczących jakości.
- Musisz także zainicjować czas zakończenia. Ustaw czas rozpoczęcia, aby zasygnalizować, że nie zarejestrowano jeszcze czasu zakończenia.

```
data class SleepNight(
        var nightId: Long = 0L,
        val startTimeMilli: Long = System.currentTimeMillis(),
        var endTimeMilli: Long = startTimeMilli,
        var sleepQuality: Int = -1
)
```

3. Przed deklaracją klasy oznacz adnotację klasy danych za pomocą `@Entity`. Nazwij tabelę `daily_sleep_quality_table`. Argument dla `tableName` jest opcjonalny, ale zalecany. Możesz wyszukać inne argumenty w dokumentacji.

   Jeśli pojawi się monit, zaimportuj `Entity` i wszystkie inne adnotacje z biblioteki `androidx`.

```
@Entity(tableName = "daily_sleep_quality_table")
data class SleepNight(...)
```

4. Aby zidentyfikować `nightId` jako klucz podstawowy, oznacz właściwość `nightId` za pomocą `@PrimaryKey`. Ustaw parametr `autoGenerate` na `true` aby `Room` generował identyfikator dla każdej jednostki. Gwarantuje to, że identyfikator każdej nocy jest unikalny.

```
@PrimaryKey(autoGenerate = true)
var nightId: Long = 0L,...
```

5. Opisz pozostałe właściwości za pomocą `@ColumnInfo`. Dostosuj nazwy właściwości za pomocą parametrów, jak pokazano poniżej.

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

@Entity(tableName = "daily_sleep_quality_table")
data class SleepNight(
       @PrimaryKey(autoGenerate = true)
       var nightId: Long = 0L,

       @ColumnInfo(name = "start_time_milli")
       val startTimeMilli: Long = System.currentTimeMillis(),

       @ColumnInfo(name = "end_time_milli")
       var endTimeMilli: Long = startTimeMilli,

       @ColumnInfo(name = "quality_rating")
       var sleepQuality: Int = -1
)
```

6. Zbuduj i uruchom kod, aby upewnić się, że nie zawiera błędów.

5. Zadanie: Utwórz DAO

W tym zadaniu definiujesz obiekt dostępu do danych (DAO). W systemie Android DAO zapewnia wygodne metody wstawiania, usuwania i aktualizacji bazy danych.

Korzystając z bazy danych `Room` , przeszukujesz bazę danych, definiując i wywołując funkcje Kotlin w kodzie. Te funkcje Kotlin są mapowane na zapytania SQL. Te odwzorowania definiujesz w DAO za pomocą adnotacji, a `Room` tworzy niezbędny kod.

Pomyśl o DAO jako o zdefiniowanym niestandardowym interfejsie dostępu do bazy danych.

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

Jako dodatkowy bonus podczas tworzenia zapytań w Android Studio kompilator sprawdza zapytania SQL pod kątem błędów składniowych.

W przypadku naszej bazy danych musisz wykonać następujące czynności::

- **Wstaw nowe noce (Insert** new nights).

- **Zaktualizuj istniejącą noc, aby zaktualizować godzinę zakończenia i ocenę jakości**.
- **Uzyskaj konkretną noc na podstawie jej klucza**.
- **Zbierz wszystkie noce, abyś mógł je wyświetlić**.
- **Uzyskaj najnowszą noc.**
- **Usuń wszystkie wpisy z bazy danych**.

Step 1: Utwórz DAO SleepDatabase

1. W pakiecie `database` otwórz `SleepDatabaseDao.kt`.
2. Zauważ, `interface SleepDatabaseDao` est opatrzony adnotacją `@Dao`. Wszystkie DAO muszą być opatrzone adnotacjami ze słowem kluczowym `@Dao`.

```
@Dao
interface SleepDatabaseDao {}
```

3. W treści interfejsu dodaj adnotację `@Insert` Pod `@Insert`, dodaj funkcję `insert()`, która przyjmuje za argument instancję `Entity` class `SleepNight`.

   to jest to! `Room` wygeneruje cały kod niezbędny do wstawienia `SleepNight` do bazy danych. Po wywołaniu funkcji `insert()` z kodu Kotlin, `Room` wykonuje zapytanie SQL w celu wstawienia encji do bazy danych. (Uwaga: możesz wywołać tę funkcję, jak chcesz.)

```
@Insert
fun insert(night: SleepNight)
```

4. Dodaj adnotację `@Update` z funkcją `update()` dla jednego `SleepNight`. Encja, która jest aktualizowana, to encja, która ma taki sam klucz jak ten, który został przekazany. Możesz zaktualizować niektóre lub wszystkie inne właściwości encji.

```
@Update
fun update(night: SleepNight)
```

Brak pozostałej adnotacji dla pozostałej funkcjonalności, więc musisz użyć adnotacji `@Query` i podać zapytania SQLite..

5. Dodaj adnotację `@Query` za pomocą funkcji `get()`, która pobiera argument `Long key` i zwraca parametr `SleepNight`. Zobaczysz błąd dotyczący brakującego parametru.

```
@Query
fun get(key: Long): SleepNight?
```

6. Kwerenda jest dostarczana jako parametr ciągu do adnotacji. Dodaj parametr do `@Query`. Niech to będzie ciąg znaków, który jest zapytaniem SQLite.

- Wybierz wszystkie kolumny z tabeli `daily_sleep_quality_table`
- `WHERE nightId` pasuje do argumentu:`key`.

  Zwróć uwagę na `:key`. W zapytaniu używasz notacji dwukropka, aby odwoływać się do argumentów w funkcji.

```
("SELECT * from daily_sleep_quality_table WHERE nightId = :key")
```

7. Dodaj kolejny `@Query` z a `clear()` z funkcją `DELETE` i kwerendą SQLite, aby usunąć wszystko z tabeli `daily_sleep_quality_table`. To zapytanie nie usuwa samej tabeli.

   Adnotacja `@Delete` usuwa jeden element. Możesz użyć `@Delete` i podać listę nocy do usunięcia. Wadą jest to, że musisz pobrać lub wiedzieć, co jest w tabeli. Adnotacja `@Delete` doskonale nadaje się do usuwania określonych pozycji, ale nie jest skuteczna do usuwania wszystkich pozycji z tabeli.

```
@Query("DELETE FROM daily_sleep_quality_table")
fun clear()
```

8. Dodaj `@Query` za pomocą funkcji `getTonight()` Ustaw wartość `SleepNight` returned zwróconą przez `getTonight()` nullable, aby funkcja mogła obsłużyć przypadek, w którym tabela jest pusta. (Tabela jest pusta na początku i po wyczyszczeniu danych.)

   Aby uzyskać "tonight" z bazy danych, napisz zapytanie SQLite, które zwraca pierwszy element listy wyników uporządkowanych według `nightId` w kolejności malejącej. Użyj `LIMIT 1` , aby zwrócić tylko jeden element.

```
@Query("SELECT * FROM daily_sleep_quality_table ORDER BY nightId DESC LIMIT
1")
fun getTonight(): SleepNight?
```

9. Dodaj `@Query` z funkcją `getAllNights()`:

- Poproś, aby zapytanie SQLite zwróciło wszystkie kolumny z tabeli `daily_sleep_quality_table`, , uporządkowane w kolejności malejącej.
- Niech `getAllNights()` zwraca listę encji `SleepNight` jako `LiveData`. `Room` aktualizuje dla Ciebie `LiveData` , co oznacza, że potrzebujesz tylko jednokrotnie uzyskać dane.
- Może być konieczne zaimportowanie `LiveData` z `androidx.lifecycle.LiveData`.

```
@Query("SELECT * FROM daily_sleep_quality_table ORDER BY nightId DESC")
fun getAllNights(): LiveData<List<SleepNight>>
```

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

6. Zadanie: Utwórz i przetestuj bazę danych Room

W tym zadaniu tworzysz bazę danych `Room` która korzysta z encji i DAO utworzonych w poprzednim zadaniu.

Musisz utworzyć abstrakcyjną klasę posiadacza bazy danych, opatrzoną adnotacjami `@Database`. Ta klasa ma jedną metodę, która albo tworzy instancję bazy danych, jeśli baza danych nie istnieje, albo zwraca odwołanie do istniejącej bazy danych.

Uzyskiwanie bazy danych `Room` jest trochę skomplikowane, więc oto ogólny proces przed rozpoczęciem korzystania z kodu:

- Utwórz publiczną klasę abstrakcyjną, `public abstract` , która rozszerza `extends RoomDatabase`. Ta klasa ma działać jako właściciel bazy danych. Klasa jest abstrakcyjna, ponieważ `Room` tworzy dla Ciebie implementację.
- Adnotuj klasę za pomocą`@Database`. W argumentach zadeklaruj entities bazy danych i ustaw numer wersji
- W obiekcie `companion` zdefiniuj abstrakcyjną metodę lub właściwość, która zwraca `SleepDatabaseDao`. `Room` wygeneruje dla ciebie ciało.
- Potrzebujesz tylko jednego wystąpienia bazy danych `Room` dla całej aplikacji, więc zadeklaruj `RoomDatabase` jako singleton.
- Użyj konstruktora bazy danych `Room`'s , aby utworzyć bazę danych tylko wtedy, gdy baza danych nie istnieje. W przeciwnym razie zwróć istniejącą bazę danych.

Tip: Kod będzie taki sam dla każdej bazy danych `Room` , więc możesz użyć tego kodu jako szablonu.

Step 1: Utwórz bazę danych

1. W pakiecie `database` otwórz `SleepDatabase.kt`.
2. W pliku utwórz abstrakcyjną klasę `SleepDatabase` , która rozszerza `RoomDatabase`.

   Adnotuj klasę za pomocą `@Database`.

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

3. Zobaczysz błąd dotyczący brakujących elementów i parametrów wersji. Adnotacja `@Database` wymaga kilku argumentów, aby Room mógł zbudować bazę danych.

- Podaj `SleepNight` jako jedyny element z listą `entities`.
- Ustaw `version` jako 1. Ilekroć zmienisz schemat, będziesz musiał zwiększyć numer wersji.
- Ustaw wartość `exportSchema` na `false`, , aby nie przechowywać kopii wersji.

```
entities = [SleepNight::class], version = 1, exportSchema = false
```

4. Baza danych musi wiedzieć o DAO. W treści klasy zadeklaruj wartość abstrakcyjną, która zwraca `SleepDatabaseDao`. Możesz mieć wiele DAO.

```
abstract val sleepDatabaseDao: SleepDatabaseDao
```

5. Poniżej zdefiniuj obiekt `companion` object. companion umożliwia klientom dostęp do metod tworzenia lub pobierania bazy danych bez tworzenia instancji klasy. Ponieważ jedynym celem tej klasy jest zapewnienie bazy danych, nie ma powodu, aby kiedykolwiek ją tworzyć.

```
 companion object {}
```

6. Wewnątrz obiektu `companion` zadeklaruj prywatną zerowalną zmienną nullable variable `INSTANCE` dla bazy danych i zainicjuj ją na wartość `null`. Po utworzeniu zmienna `INSTANCE` zachowa odwołanie do bazy danych. Pomaga to uniknąć wielokrotnego otwierania połączeń z bazą danych, co jest kosztowne.

Adnotuj `INSTANCE` za pomocą `@Volatile`. Wartość zmiennej volatile nigdy nie będzie buforowana, a wszystkie zapisy i odczyty będą wykonywane do iz pamięci głównej. Pomaga to upewnić się, że wartość parametru `INSTANCE` jest zawsze aktualna i taka sama dla wszystkich wątków wykonania. Oznacza to, że zmiany wprowadzone przez jeden wątek w `INSTANCE` są natychmiast widoczne dla wszystkich innych wątków i nie występuje sytuacja, w której, powiedzmy, dwa wątki aktualizują ten sam byt w pamięci podręcznej, co spowodowałoby problem.

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

7. Poniżej `INSTANCE`, , wciąż wewnątrz obiektu `companion` , zdefiniuj metodę `getInstance()` z parametrem `Context` , którego będzie potrzebował konstruktor bazy danych. Zwracany typ to `SleepDatabase`. Zobaczysz błąd, ponieważ `getInstance()` eszcze niczego nie zwraca.

```
fun getInstance(context: Context): SleepDatabase {}
```

8. Wewnątrz `getInstance()`,dodaj zsynchronizowany blok `synchronized{}` Przekaż `this` , aby uzyskać dostęp do kontekstu.

   Wiele wątków może potencjalnie poprosić o wystąpienie bazy danych jednocześnie, w wyniku czego powstają dwie bazy danych zamiast jednej. Ten problem prawdopodobnie nie wystąpi w tej przykładowej aplikacji, ale jest to możliwe w przypadku bardziej złożonej aplikacji. Zawijanie kodu w celu synchronizacji bazy danych oznacza, że tylko jeden wątek wykonania na raz może wejść do tego bloku kodu, co zapewnia, że baza danych zostanie zainicjowana tylko raz.

```
synchronized(this) {}
```

9. Wewnątrz zsynchronizowanego bloku skopiuj bieżącą wartość parametru `INSTANCE` do zmiennej lokalnej `instance`. Ma to na celu wykorzystanie smart cast, , która jest dostępna tylko dla zmiennych lokalnych.

```
var instance = INSTANCE
```

10. Wewnątrz bloku `synchronized` block, zwróć `return instance` na końcu bloku `synchronized` block. Zignoruj błąd niedopasowania typu zwracanego; nigdy nie zwrócisz null , gdy skończysz.

```
return instance
```

11. Nad instrukcją `return` dodaj instrukcję `if` , aby sprawdzić, czy `instance` ma wartość null, , czyli nie ma jeszcze bazy danych.

```
if (instance == null) {}
```

12. Jeśli `instance` ma wartość `null`, , użyj konstruktora bazy danych, aby uzyskać bazę danych. W treści instrukcji if wywołaj `Room.databaseBuilder` i podaj przekazany kontekst, klasę bazy danych oraz nazwę bazy danych,, `sleep_history_database`. Aby usunąć błąd, musisz dodać „migration strategy" i `build()` w poniższych krokach.

```
instance = Room.databaseBuilder(
                            context.applicationContext,
                            SleepDatabase::class.java,
                            "sleep_history_database")
```

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

    Zwykle trzeba dostarczyć obiekt migracji ze strategią migracji na wypadek zmiany schematu. *migration object* to obiekt, który określa sposób pobierania wszystkich wierszy ze starym schematem i konwertowania ich na wiersze w nowym schemacie, aby żadne dane nie zostały utracone. Migracja wykracza poza zakres tego kodu. Prostym rozwiązaniem jest zniszczenie i odbudowanie bazy danych, co oznacza utratę danych.

```
.fallbackToDestructiveMigration()
```

14. Na koniec wywołaj `.build()`.

```
.build()
```

15. Przypisz `INSTANCE = instance` jako ostatni krok w instrukcji if.

```
INSTANCE = instance
```

16. Twój końcowy kod powinien wyglądać następująco:

```
@Database(entities = [SleepNight::class], version = 1, exportSchema =
false)
abstract class SleepDatabase : RoomDatabase() {

   abstract val sleepDatabaseDao: SleepDatabaseDao

   companion object {

       @Volatile
       private var INSTANCE: SleepDatabase? = null

       fun getInstance(context: Context): SleepDatabase {
           synchronized(this) {
               var instance = INSTANCE

               if (instance == null) {
                   instance = Room.databaseBuilder(
                           context.applicationContext,
                           SleepDatabase::class.java,
                           "sleep_history_database"
                   )
                           .fallbackToDestructiveMigration()
                           .build()
```

```
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

17. Zbuduj i uruchom swój kod.

Masz teraz wszystkie elementy do pracy z bazą danych `Room` database. Ten kod kompiluje się i działa, ale nie można stwierdzić, czy rzeczywiście działa. To dobry moment na dodanie podstawowych testów.

Step 2: Przetestuj SleepDatabase

W tym kroku uruchamiane są dostarczone testy w celu sprawdzenia, czy baza danych działa. Pomaga to upewnić się, że baza danych działa. Podane testy są podstawowe. W przypadku aplikacji produkcyjnej wykonujesz wszystkie funkcje i zapytania we wszystkich DAO..

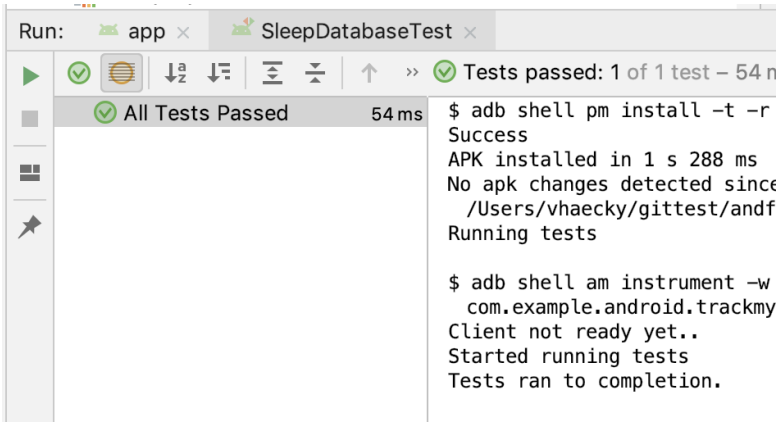
Aplikacja startowa zawiera folder **androidTest** Ten folder **androidTest** zawiera testy jednostkowe obejmujące oprzyrządowanie dla Androida więc musisz uruchomić testy na urządzeniu fizycznym lub wirtualnym. Oczywiście możesz także tworzyć i uruchamiać czyste testy jednostkowe, które nie wymagają systemu Android..

1. W Android Studio w folderze **androidTest** folder, otwórz plik **SleepDatabaseTest**.
2. Aby odkomentować kod, zaznacz cały skomentowany kod i naciśnij skrót klawiaturowy `Cmd+/`.
3. Spójrz na plik.

Oto krótki przegląd kodu testowego, ponieważ jest to kolejny fragment kodu, którego można użyć ponownie:

- `SleepDabaseTest` to klasa testowa**.**
- Adnotacja `@RunWith` identyfikuje program uruchamiający testy, czyli program, który konfiguruje i wykonuje testy.
- Podczas instalacji wykonywana jest funkcja z adnotacją`@Before` i tworzy ona `SleepDatabase` w pamięci za pomocą `SleepDatabaseDao`. „W pamięci” oznacza, że ta baza danych nie jest zapisywana w systemie plików i zostanie usunięta po uruchomieniu testów.
- Również podczas budowania bazy danych w pamięci kod wywołuje inną metodę specyficzną dla testu, `allowMainThreadQueries`. Domyślnie pojawia się błąd, gdy próbujesz uruchomić zapytania w głównym wątku. Ta metoda umożliwia uruchamianie testów w głównym wątku, co należy wykonywać tylko podczas testowania.
- W metodzie testowej opatrzonej adnotacją `@Test`, tworzysz, wstawiasz i pobierasz `SleepNight`, and assert that they are the same. oraz upewniasz się, że są takie same. Jeśli coś pójdzie nie tak, wyrzuca wyjątek. W prawdziwym teście miałbyś wiele metod `@Test`.
- Po zakończeniu testowania funkcja z adnotacją `@After` wykonuje zamkniecie bazy danych.

4. Kliknij prawym przyciskiem myszy plik testowy w panelu **Project** i wybierz opcję **Run 'SleepDatabaseTest'**.
5. Po uruchomieniu testów sprawdź w panelu **SleepDatabaseTest** , czy wszystkie testy przeszły pomyślnie.



Ponieważ wszystkie testy przeszły pomyślnie, wiesz teraz kilka rzeczy:

- Baza danych została poprawnie utworzona.
- Możesz wstawića `SleepNight` do bazy danych.
- Możesz odzyskać `SleepNight`.