1. Wprowadzenie

Jednym z głównych priorytetów tworzenia aplikacji bezbłędnie dla użytkownika jest upewnienie się, że interfejs użytkownika zawsze reaguje i działa płynnie. Jednym ze sposobów na poprawę wydajności interfejsu użytkownika jest przeniesienie długotrwałych zadań, takich jak operacje bazy danych, w tło.

W tym ćwiczeniu zaimplementujesz część aplikacji TrackMySleepQuality, używając coroutines Kotlin do wykonywania operacji bazy danych z dala od głównego wątku.

Co powinieneś już wiedzieć

Powinieneś znać:

- Budowanie podstawowego interfejsu użytkownika (UI) przy użyciu działania, fragmentów, widoków i programów obsługi kliknięć.
- Nawigacja między fragmentami i używanie SafeArgs do przekazywania prostych danych między fragmentami.
- View models, view model factories, transformations, i `LiveData`.
- Jak utworzyć bazę danych `Room` database, create a DAO, utworzyć DAO i zdefiniować entities.

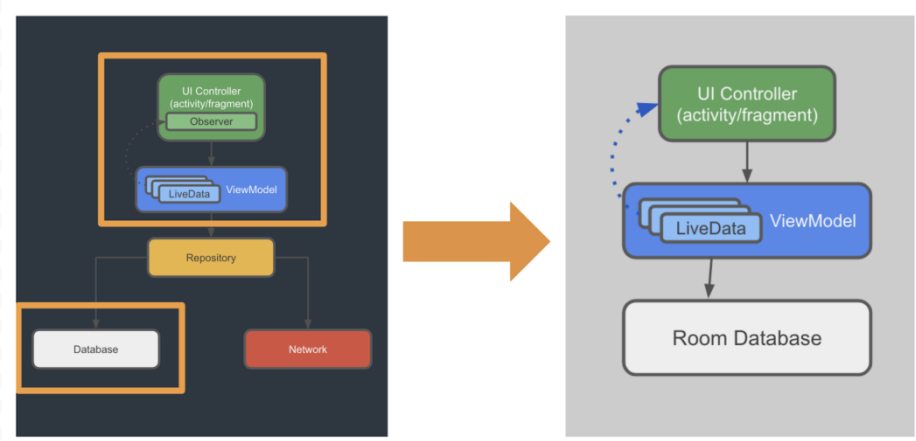
Czego się nauczysz

- Jak działają wątki w Androidzie.
- Jak korzystać z Kotlin coroutines , aby przenieść operacje bazy danych z głównego wątku.
- Jak wyświetlić sformatowane dane w `TextView`.

Co będziesz robić

- Rozszerz aplikację TrackMySleepQuality o zbieranie, przechowywanie i wyświetlanie danych w bazie danych iz bazy danych.
- Używaj coroutines do uruchamiania długotrwałych operacji na bazach danych w tle.
- Użyj LiveData, aby uruchomić nawigację i wyświetlanie paska snackbar.
- Użyj LiveData, aby włączyć lub wyłączyć przyciski.

2. Przegląd aplikacji



3. Zadanie: Sprawdź kod startowy

Step 1: Download and run the starter app

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

Step 2: Sprawdź kod

1. Otwórz plik **res/layout/activity_main.xml.** Ten układ zawiera fragment `nav_host_fragment` Zwróć także uwagę na tag `<merge>`.

   Znacznika `merge` można użyć do wyeliminowania zbędnych układów podczas dołączania układów, dlatego warto go używać. Przykładem redundantnego układu może być ConstraintLayout > LinearLayout > TextView, , w którym system może być w stanie wyeliminować LinearLayout. Ten rodzaj optymalizacji może uprościć hierarchię widoków i poprawić wydajność aplikacji.
2. W folderze **navigation** otwórz plik **navigation.xml**. Możesz zobaczyć dwa fragmenty i łączące je działania nawigacyjne.
3. W folderze **layout** kliknij dwukrotnie fragment sleep tracker fragment , aby zobaczyć jego XML layout. Zwróć uwagę na następujące:

- Dane układu są zawinięte w element `<layout>` aby umożliwić powiązanie danych.
- `ConstraintLayout` i inne widoki są umieszczone wewnątrz elementu `<layout>`.
- Plik ma placeholder `<data>`.

Aplikacja o razu zapewnia również wymiary, kolory i styl dla interfejsu użytkownika. Aplikacja zawiera bazę danych Room, DAO i obiekt `SleepNight` entity.

4. Zadanie: dodaj ViewModel

Teraz, gdy masz bazę danych i interfejs użytkownika, musisz zbierać dane, dodawać dane do bazy danych i wyświetlać dane. Cała ta praca jest wykonywana w view model. Twój sleep-tracker view model będzie obsługiwał kliknięcia przycisków, wchodził w interakcje z bazą

danych za pośrednictwem DAO i dostarczał dane do interfejsu użytkownika za pośrednictwem `LiveData`. Wszystkie operacje na bazach danych będą musiały zostać usunięte z głównego wątku interfejsu użytkownika, a zrobisz to przy użyciu coroutines..

Step 1: Dodaj SleepTrackerViewModel

1. W pakiecie **sleeptracker** otwórz **SleepTrackerViewModel.kt**.
2. Sprawdź klasę `SleepTrackerViewModel` , która jest dostępna w aplikacji i jest również pokazana poniżej. Zauważ, że klasa rozszerza `AndroidViewModel()`.Ta klasa jest taka sama jak `ViewModel`, , ale przyjmuje kontekst aplikacji jako parametr i udostępnia ją jako właściwość. Będziesz tego potrzebować później.

```
class SleepTrackerViewModel(
       val database: SleepDatabaseDao,
       application: Application) : AndroidViewModel(application) {
}
```

Step 2: Dodaj SleepTrackerViewModelFactory

1. W pakiecie **sleeptracker** otwórz **SleepTrackerViewModelFactory.kt.**
2. Sprawdź kod, który został dostarczony dla modelu fabryki, pokazany poniżej:

```
class SleepTrackerViewModelFactory(
       private val dataSource: SleepDatabaseDao,
       private val application: Application) : ViewModelProvider.Factory {
   @Suppress("unchecked_cast")
   override fun <T : ViewModel?> create(modelClass: Class<T>): T {
       if (modelClass.isAssignableFrom(SleepTrackerViewModel::class.java))
{
           return SleepTrackerViewModel(dataSource, application) as T
       }
       throw IllegalArgumentException("Unknown ViewModel class")
   }
}
```

Zwróć uwagę na następujące kwestie:

- Dostarczony `SleepTrackerViewModelFactory` przyjmuje ten sam argument co `ViewModel` rozszerza `ViewModelProvider.Factory`.
- W fabryce kod zastępujący funkcję `create()`,, która przyjmuje dowolny typ klasy jako argument i zwraca `ViewModel`.
- W treści `create()`,kod sprawdza, czy dostępna jest klasa `SleepTrackerViewModel` a jeśli tak, zwraca jej instancję. W przeciwnym razie kod zgłasza wyjątek.

Tip: Jest to kod podstawowy „boilerplate”, więc można go ponownie wykorzystać w przyszłych fabrykach `ViewModel`.

Step 3: Zaktualizuj SleepTrackerFragment

1. W `SleepTrackerFragment`, uzyskaj odwołanie do kontekstu aplikacji. Umieść odwołanie w `onCreateView()`,, poniżej `binding`. Potrzebujesz odniesienia do

aplikacji, do której dołączony jest ten fragment, aby przejść do „view-model factory provider”.

Funkcja `requireNotNull` Kotlin zgłasza wyjątek `IllegalArgumentException` , jeśli wartość jest równa `null`.

```
val application = requireNotNull(this.activity).application
```

2. Potrzebujesz odniesienia do źródła danych poprzez odniesienie do DAO. W `onCreateView()`, przed `return`, zdefiniuj `dataSource`. Aby uzyskać odniesienie do DAO bazy danych, użyj `SleepDatabase.getInstance(application).sleepDatabaseDao`.

```
val dataSource = SleepDatabase.getInstance(application).sleepDatabaseDao
```

3. W `onCreateView()`,przed `return`, , utwórz instancję `viewModelFactory`. Musisz przekazać `dataSource` i `application`.

```
val viewModelFactory = SleepTrackerViewModelFactory(dataSource,
application)
```

4. Teraz, gdy masz już fabrykę, uzyskaj odniesienie do `SleepTrackerViewModel`. Parametr `SleepTrackerViewModel::class.java` odnosi się do klasy Java środowiska wykonawczego tego obiektu.

```
val sleepTrackerViewModel =
       ViewModelProviders.of(
               this,
viewModelFactory).get(SleepTrackerViewModel::class.java)
```

5. Gotowy kod powinien wyglądać następująco:

```
// Create an instance of the ViewModel Factory.
val dataSource = SleepDatabase.getInstance(application).sleepDatabaseDao
val viewModelFactory = SleepTrackerViewModelFactory(dataSource,
application)

// Get a reference to the ViewModel associated with this fragment.
val sleepTrackerViewModel =
       ViewModelProviders.of(
               this,
viewModelFactory).get(SleepTrackerViewModel::class.java)
```

Metoda `onCreateView()`:

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {

        // Get a reference to the binding object and inflate the fragment
views.
        val binding: FragmentSleepTrackerBinding = DataBindingUtil.inflate(
                inflater, R.layout.fragment_sleep_tracker, container,
false)

        val application = requireNotNull(this.activity).application
```

```
        val dataSource =
SleepDatabase.getInstance(application).sleepDatabaseDao

        val viewModelFactory = SleepTrackerViewModelFactory(dataSource,
application)

        val sleepTrackerViewModel =
                ViewModelProviders.of(
                        this,
viewModelFactory).get(SleepTrackerViewModel::class.java)

        return binding.root
    }
```

Step 4: Dodaj powiązanie danych dla view model

Po zainstalowaniu podstawowego `ViewModel` i musisz zakończyć konfigurowanie powiązania danych (data Winding) w `SleepTrackerFragment` , aby połączyć `ViewModel` z interfejsem użytkownika.

W pliku układu `fragment_sleep_tracker.xml`:

1. Wewnątrz bloku `<data>` utwórz `<variable>` która odwołuje się do klasy `SleepTrackerViewModel`.

```
<data>
   <variable
       name="sleepTrackerViewModel"

type="com.example.android.trackmysleepquality.sleeptracker.SleepTrackerView
Model" />
</data>
```

W `SleepTrackerFragment`:

1. Ustaw bieżącą aktywność jako właściciela cyklu życia powiązania. Dodaj ten kod do metody `onCreateView()` przed instrukcją `return`:

```
binding.setLifecycleOwner(this)
```

2. Przypisz zmienną powiązania `sleepTrackerViewModel` do `sleepTrackerViewModel`. Umieść ten kod wewnątrz `onCreateView()`,poniżej kodu tworzącego `SleepTrackerViewModel`:

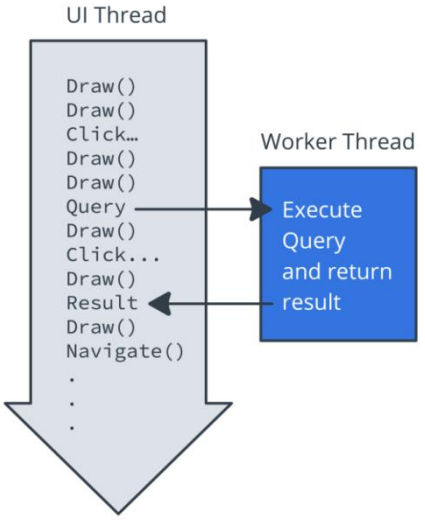
```
binding.sleepTrackerViewModel = sleepTrackerViewModel
```

3. Prawdopodobnie zobaczysz błąd, ponieważ musisz odtworzyć obiekt powiązania. Wyczyść i przebuduj projekt, aby pozbyć się błędu.
4. Na koniec, jak zawsze, upewnij się, że kod buduje się i działa bez błędów.

5. Pojęcie: Coroutines

Jednym z wzorców wykonywania długotrwałych zadań bez blokowania głównego wątku jest użycie wywołań zwrotnych callbacks. Zobacz więcej Multi-threading & callbacks primer

pozwalają konwertować kod oparty na wywołaniu zwrotnym na kod sekwencyjny. Kod pisany sekwencyjnie jest zazwyczaj łatwiejszy do odczytania i może nawet korzystać z funkcji językowych, takich jak wyjątki. Ostatecznie, coroutines i callbacks robią to samo: czekają, aż wynik będzie dostępny z długo działającego zadania i kontynuują wykonywanie.



Coroutines mają następujące właściwości:

- Coroutines są asynchroniczne i nie blokują.
- Coroutines używają funkcji *suspend* do sekwencjonowania kodu asynchronicznego.

Coroutines są asynchroniczne.

działa niezależnie od głównych kroków wykonania programu. Może to być równoległe lub na osobnym procesorze. Jednym z ważnych aspektów asynchronizacji jest to, że nie można oczekiwać, że wynik będzie dostępny, dopóki jawnie na niego nie poczekasz.

Załóżmy na przykład, że masz pytanie wymagające wiecej czasu czasu i poprosisz kolegę o znalezienie odpowiedzi. Odchodzi i pracuje nad tym, tak jakby wykonywał pracę „asynchronicznie" i „na osobnym wątku". Możesz kontynuować pracę, która nie zależy od odpowiedzi, dopóki twój kolega nie wróci i nie powie ci, jaka jest odpowiedź.

Coroutines nie są blokujące.

Non-blocking Nieblokowanie oznacza, że coroutine nie blokuje wątku głównego ani interfejsu użytkownika. Tak więc w przypadku coroutines użytkownicy mają zawsze możliwie płynne działanie, ponieważ interakcja interfejsu użytkownika zawsze ma priorytet..

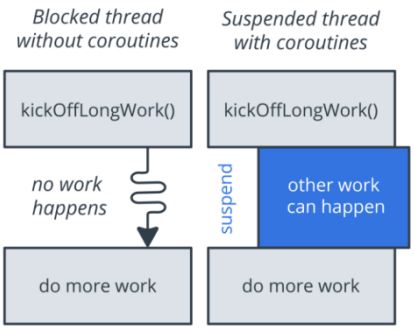
Coroutines używają funkcji suspend do sekwencjonowania kodu asynchronicznego.

Słowo kluczowe `suspend` to sposób Kotlina na oznaczenie funkcji lub typu funkcji jako dostępnej dla coroutines. Gdy coroutine wywołuje funkcję oznaczoną `suspend`, , zamiast blokować, aż funkcja powróci jak normalne wywołanie funkcji, coroutine zawiesza wykonywanie, dopóki wynik nie będzie gotowy. Następnie coroutine wraca do miejsca, w którym zostało przerwane, z wynikiem.

Podczas gdy coroutine jest zawieszona i czeka na wynik, odblokowuje wątek, na którym działa. W ten sposób mogą działać inne funkcje lub coroutines.

Słowo kluczowe `suspend` nie określa wątku, na którym działa kod. Funkcja zawieszenia może działać w wątku w tle lub w wątku głównym.

Tip: Różnica między blokowaniem a zawieszaniem polega na tym, że jeśli wątek jest zablokowany, żadna inna praca nie jest wykonywana. Jeśli wątek zostanie zawieszony, będą wykonywane inne prace, dopóki wynik nie będzie dostępny.



Aby używać coroutines w Kotlin, potrzebujesz trzech rzeczy:

- Job (zadanie)
- Dispatcher (dyspozytor)
- Scope (zakres)

Job: Zasadniczo zadanie to wszystko, co można anulować. Każda coroutine ma zadanie i możesz użyć tego zadania do anulowania coroutine. Zadania można uporządkować w hierarchii rodzic-dziecko. Anulowanie pracy rodzica natychmiast anuluje wszystkie dzieci tej pracy, co jest o wiele wygodniejsze niż ręczne anulowanie każdej coroutine.

Dispatcher: Dyspozytor wysyła coroutines do działania w różnych wątkach. Na przykład, `Dispatcher.Main` uruchamia zadania w głównym wątku, a `Dispatcher.IO` zwalnia blokowanie zadań we / wy do wspólnej puli wątków.

Scope: Zakres coroutine określa kontekst, w którym działa coroutine. Scope łączy informacje o pracy i dyspozytorze coroutine. Scopes śledzą coroutines. Kiedy uruchamiasz coroutine, jest ona „w zakresie”, co oznacza, że wskazałeś, który zakres będzie śledził coroutine.

6. Zadanie: Zbierz i wyświetl dane

Chcesz, aby użytkownik mógł wchodzić w interakcje z danymi dotyczącymi snu na następujące sposoby:

- Gdy użytkownik stuknie przycisk **Start** , aplikacja tworzy nową noc snu i zapisuje noc snu w bazie danych.
- Gdy użytkownik dotknie przycisku **Stop** , aplikacja aktualizuje noc o godzinę zakończenia.
- Gdy użytkownik stuknie przycisk **Clear** aplikacja usuwa dane z bazy danych.

Te operacje na bazie danych mogą zająć dużo czasu, dlatego powinny działać w osobnym wątku.

Step 1: Skonfiguruj coroutines dla operacji bazy danych

Po stuknięciu przycisku **Start** w aplikacji Sleep Tracker chcesz wywołać funkcję w `SleepTrackerViewModel` , aby utworzyć nową instancję `SleepNight` i zapisać ją w bazie danych.

Stuknięcie dowolnego przycisku uruchamia operację bazy danych, taką jak utworzenie lub aktualizacja `SleepNight`. Z tego i innych powodów korzystasz z coroutines do implementacji programów obsługi kliknięć dla przycisków aplikacji.

1. Otwórz plik `build.gradle` na poziomie aplikacji i znajdź zależności dla coroutines. Aby korzystać z coroutines, potrzebujesz tych zależności, które zostały dla Ciebie dodane.

   `$coroutine_version` jest zdefiniowany w pliku `build.gradle` na poziomie projektu jako `coroutine_version = '1.0.0'.`

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

2. Otwórz plik `SleepTrackerViewModel`.
3. W treści klasy zdefiniuj `viewModelJob` i przypisz mu instancję`Job`. Ten `viewModelJob` pozwala anulować wszystkie coroutines rozpoczęte przez ten model widoku, gdy model widoku nie jest już używany i jest zniszczony. W ten sposób nie skończysz z coroutines , do których nie masz gdzie wrócić.

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

4. Na końcu treści klasy przesłoń `onCleared()` anuluj wszystkie coroutines. Po zniszczeniu `ViewModel` wywoływana jest funkcja `onCleared()`.

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

```
}
```

5. Tuż pod definicją `viewModelJob`, zdefiniuj `uiScope` dla coroutines. Zakres określa, w jakim wątku będzie wykonywana coroutine, a zakres musi również wiedzieć o zadaniu. Aby uzyskać zakres, przypisz instancję `CoroutineScope`, i przekaż dyspozytora i zadanie.

Korzystanie z `Dispatchers.Main` oznacza, że coroutines uruchomione w `uiScope` będą działać na głównym wątku. Jest to sensowne w przypadku wielu coroutines uruchomionych przez `ViewModel`, ponieważ po ich przetworzeniu niektóre z nich powodują aktualizację interfejsu użytkownika.

```
private val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)
```

6. Poniżej definicji `uiScope`, zdefiniuj zmienną `tonight` aby zatrzymać aktualną noc. Stwórz zmienną `MutableLiveData`, , ponieważ musisz być w stanie obserwować dane i je zmieniać.

```
private var tonight = MutableLiveData<SleepNight?>()
```

7. Aby jak najszybciej zainicjalizować zmienną `tonight` utwórz blok `init` oniżej definicji `tonight` i wywołaj `initializeTonight()`.Zdefiniujesz `initializeTonight()` w następnym kroku.

```
init {
   initializeTonight()
}
```

8. Pod blokiem `init` zaimplementuj `initializeTonight()`.W `uiScope`, uruchom coroutine. Wewnątrz pobierz wartość dla `tonight` z bazy danych, wywołując metodę `getTonightFromDatabase()`,przypisz wartość do `tonight.value`. Zdefiniuj `getTonightFromDatabase()` w następnym kroku.

```
private fun initializeTonight() {
   uiScope.launch {
       tonight.value = getTonightFromDatabase()
   }
}
```

9. Zaimplementuj `getTonightFromDatabase()`.Zdefiniuj go jako prywatną funkcję `private suspend` która zwraca zerowalną wartość `SleepNight`, jeśli nie ma aktualnie uruchomionej funkcji `SleepNight`. To powoduje błąd, ponieważ funkcja musi coś zwrócić.

```
private suspend fun getTonightFromDatabase(): SleepNight? { }
```

10. W treści funkcji `getTonightFromDatabase()`,zwróć wynik z coroutine działającej w kontekście `Dispatchers.IO` Użyj programu rozsyłającego we / wy, ponieważ pobieranie danych z bazy danych jest operacją we / wy i nie ma nic wspólnego z interfejsem użytkownika.

```
  return withContext(Dispatchers.IO) {}
```

11. W bloku powrotu pozwól, aby coroutine dostał dziś wieczorem (najnowszą noc) z bazy danych. Jeśli godziny rozpoczęcia i zakończenia nie są takie same, co oznacza, że noc została już zakończona, zwróć wartość null. W przeciwnym razie zwróć noc.

```
var night = database.getTonight()
if (night?.endTimeMilli != night?.startTimeMilli) {
    night = null
}
night
```

Twoja kompletna funkcja `getTonightFromDatabase()` powinna wyglądać tak. Nie powinno być więcej błędów.

```
private suspend fun getTonightFromDatabase(): SleepNight? {
   return withContext(Dispatchers.IO) {
       var night = database.getTonight()
       if (night?.endTimeMilli != night?.startTimeMilli) {
           night = null
       }
       night
   }
}
```

Step 2: Dodaj moduł obsługi kliknięć dla przycisku Start

Teraz możesz zaimplementować `onStartTracking()`,moduł obsługi kliknięć dla przycisku Start. Musisz utworzyć nowy `SleepNight`, wstawić go do bazy danych i przypisać do. `tonight`. Struktura `onStartTracking()` będzie wyglądać bardzo podobnie do `initializeTonight()`.

1. Zacznij od definicji funkcji `onStartTracking()`.Możesz umieścić programy obsługi kliknięć powyżej `onCleared()` w pliku `SleepTrackerViewModel`.

```
fun onStartTracking() {}
```

2. W programie `onStartTracking()`,uruchom coroutine w `uiScope`, ponieważ potrzebujesz tego wyniku, aby kontynuować i zaktualizować interfejs użytkownika.

```
uiScope.launch {}
```

3. Wewnątrz uruchomienia coroutine, stwórz nowy `SleepNight`, który przechwytuje aktualny czas jako czas rozpoczęcia.

```
val newNight = SleepNight()
```

4. Wciąż w trakcie uruchamiania coroutine, wywołaj `insert()` aby wstawić `newNight` do bazy danych. Zobaczysz błąd, ponieważ nie zdefiniowałeś jeszcze tej funkcji `suspend insert()`. (To nie jest funkcja DAO o tej samej nazwie.)

```
insert(newNight)
```

5. Również w środku coroutine, zaktualizuj `tonight`.

```
        tonight.value = getTonightFromDatabase()
```

6. Poniżej `onStartTracking()`,zdefiniuj `insert()` jako prywatną funkcję `private suspend` , która przyjmuje parametr `SleepNight` jako argument.

```
private suspend fun insert(night: SleepNight) {}
```

7. W ciele `insert()`,uruchom coroutine w kontekście I / O i wstaw noc do bazy danych, wywołując `insert()` z DAO.

```
   withContext(Dispatchers.IO) {
       database.insert(night)
   }
```

8. W pliku `fragment_sleep_tracker.xml` l dodaj moduł obsługi kliknięcia dla funkcji `onStartTracking()` do przycisku `start_button` korzystając z magii powiązania danych, którą skonfigurowałeś wcześniej. Notacja funkcji `@{() ->` tworzy funkcję lambda, która nie przyjmuje żadnych argumentów i wywołuje moduł obsługi kliknięć w `sleepTrackerViewModel`.

```
android:onClick="@{() -> sleepTrackerViewModel.onStartTracking()}"
```

9. Zbuduj i uruchom aplikację. Naciśnij przycisk Start. Ta akcja tworzy dane, ale nic jeszcze nie widzisz. Napraw to potem.

Ważne: teraz możesz zobaczyć wzór:

1. Uruchom coroutine działającą w wątku głównym lub interfejsie użytkownika, ponieważ wynik wpływa na interfejs użytkownika.
2. Wywołaj funkcję zawieszenia (suspend function) aby wykonać długotrwałą pracę, aby nie blokować wątku interfejsu użytkownika podczas oczekiwania na wynik.
3. Długotrwała praca nie ma nic wspólnego z interfejsem użytkownika. Przejdź do kontekstu we / wy, aby praca mogła przebiegać w puli wątków zoptymalizowanej i zarezerwowanej dla tego rodzaju operacji.
4. Następnie wywołaj funkcję bazy danych, aby wykonać pracę.

The pattern is shown below.

```
fun someWorkNeedsToBeDone {
   uiScope.launch {

         suspendFunction()

   }
}

suspend fun suspendFunction() {
   withContext(Dispatchers.IO) {
       longrunningWork()
   }
}
```

Step 3: Wyświetl dane

W `SleepTrackerViewModel`, zmienna `nights` odwołuje się do `LiveData` , ponieważ `getAllNights()` w DAO zwraca `LiveData`.

Jest to funkcja `Room`, w której za każdym razem, gdy dane w bazie danych ulegają zmianie, `LiveData nights` są aktualizowane w celu pokazania najnowszych danych. Nigdy nie musisz jawnie ustawiać ani aktualizować `LiveData`. `Room` aktualizuje dane w celu dopasowania do bazy danych.

Jeśli jednak wyświetlisz `nights` w widoku tekstowym, wyświetli się odniesienie do obiektu. Aby zobaczyć zawartość obiektu, przekształć dane w sformatowany ciąg. Użyj mapy `Transformation`, która jest wykonywana za każdym razem, `nights` otrzymuje nowe dane z bazy danych.

1. Otwórz plik `Util.kt` i usuń komentarz z kodu definicji formatu `formatNights()` i powiązanych instrukcji `import`.
2. Zwróć uwagę, że `formatNights()` zwraca typ `Spanned`, który jest ciągiem w formacie HTML.
3. Otwórz **strings.xml**. Zwróć uwagę na użycie `CDATA` do sformatowania zasobów ciągów do wyświetlania danych "sleep data".
4. Otwórz **SleepTrackerViewModel**. W klasie `SleepTrackerViewModel` poniżej definicji `uiScope`, zdefiniuj zmienną o nazwie `nights`. Pobierz wszystkie noce z bazy danych i przypisz je do zmiennej`nights`.

```
private val nights = database.getAllNights()
```

5. Tuż pod definicją `nights`, dodaj kod, aby przekształcić `nights` w `nightsString`. Użyj funkcji `formatNights()` z `Util.kt`.

   Przekaż `nights` do funkcji `map()` z klasy `Transformations`. Aby uzyskać dostęp do zasobów łańcuchowych, zdefiniuj funkcję mapowania jako wywołanie `formatNights()`. przekaż `nights` i `Resources`.

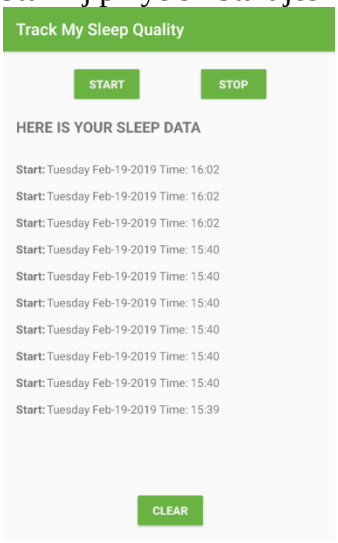
```
val nightsString = Transformations.map(nights) { nights ->
    formatNights(nights, application.resources)
}
```

6. Otwórz plik układu `fragment_sleep_tracker.xml`. W `TextView`, we właściwości `android:text` możesz teraz zastąpić ciąg zasobu odwołaniem do `nightsString`.

```
"@{sleepTrackerViewModel.nightsString}"
```

7. Przebuduj kod i uruchom aplikację. Wszystkie dane dotyczące snu z godzinami rozpoczęcia powinny zostać wyświetlone teraz.

8. Stuknij przycisk Start jeszcze kilka razy, a zobaczysz więcej danych.



W następnym kroku włączasz funkcję przycisku **Stop**.

Step 4: Dodaj moduł obsługi kliknięć dla przycisku Stop

Korzystając z tego samego wzoru, co w poprzednim kroku, zaimplementuj moduł obsługi kliknięć dla przycisku Stop w `SleepTrackerViewModel`.

1. Dodaj `onStopTracking()` do `ViewModel`. Uruchom coroutine w `uiScope`. Jeśli czas zakończenia nie został jeszcze ustawiony, ustaw `endTimeMilli` a bieżący czas systemowy i wywołaj `update()` z danymi nocnymi.

   W Kotlin składnia `return@label` (określa funkcję, z której zwraca instrukcję, spośród kilku zagnieżdżonych funkcji. syntax specifies the function from which this statement returns, among several nested functions?).

```
fun onStopTracking() {
    uiScope.launch {
        val oldNight = tonight.value ?: return@launch
        oldNight.endTimeMilli = System.currentTimeMillis()
        update(oldNight)
    }
}
```

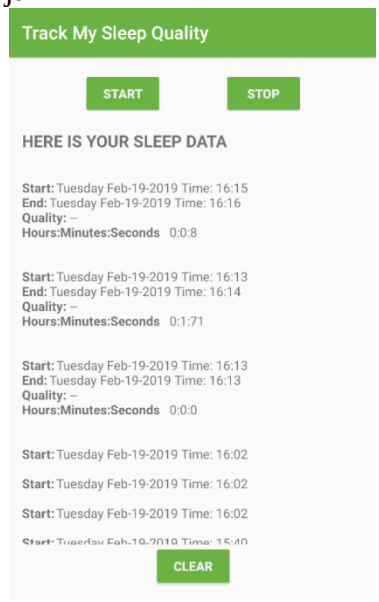
2. Zaimplementuj `update()` przy użyciu tego samego wzorca, co przy implementacji `insert()`.

```
private suspend fun update(night: SleepNight) {
    withContext(Dispatchers.IO) {
        database.update(night)
    }
}
```

3. Aby połączyć moduł obsługi kliknięć z interfejsem użytkownika, otwórz plik układu `fragment_sleep_tracker.xml` i dodaj moduł obsługi kliknięć do przycisku `stop_button`.

```
android:onClick="@{() -> sleepTrackerViewModel.onStopTracking()}"
```

4. Zbuduj i uruchom aplikację.
5. Wybierz **Start**, a następnie **Stop**. Zobaczysz czas rozpoczęcia, czas zakończenia, jakość snu bez wartości i czas snu.



Step 5: Dodaj moduł obsługi kliknięć dla przycisku Wyczyść

1. Podobnie, implementuj `onClear()` i `clear()`.

```
fun onClear() {
    uiScope.launch {
        clear()
        tonight.value = null
    }
}

suspend fun clear() {
    withContext(Dispatchers.IO) {
        database.clear()
    }
}
```

2. Aby połączyć moduł obsługi kliknięć z interfejsem użytkownika, otwórz `fragment_sleep_tracker.xml` i dodaj moduł obsługi kliknięć do przycisku `clear_button`.

```
android:onClick="@{() -> sleepTrackerViewModel.onClear()}"
```

3. Zbuduj i uruchom aplikację.
4. Stuknij Wyczyść, aby pozbyć się wszystkich danych. Następnie dotknij Start i Stop, aby utworzyć nowe dane.