Praktikum 5 – Datenhaltung in der App

In diesem Praktikum wird die Handhabung von Daten in der *DriverHelp* App behandelt. Daten werden auf einer lokalen SQLite Datenbank gespeichert, auf die per Room- Middleware zugegriffen wird. Die wesentlichen Grundlagen hierzu wurden bereits in der Vorlesung anhand des Projektes **AndRoomMileage** beschrieben. Das Projekt befindet sich im Lernraum.

Schritt 1: Projekt sichern

Sichern Sie zunächst den aktuellen Zustand des Projektes.

Schritt 2: Konfiguration der Gradle Einstellungen

Im Rahmen der Erweiterung des Projektes werden einige zusätzlich Features benutzt. Diese müssen in der *Gradle-Datei des Moduls* eingefügt werden.

```
// Lifecycle Komponenten
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.5.1"
implementation "androidx.lifecycle:lifecycle-common-java8:2.5.1"

// … zusaetzliche Kotlin-Komponenten
api "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.2"
api "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.2"

// Room Komponenten
implementation "androidx.room:room-ktx:2.4.0"
kapt "androidx.room:room-compiler:2.4.0"
androidTestImplementation "androidx.room:room-testing:2.4.0"
```

Wir müssen auch noch die Einstellung **kotlinOptions** in der Gradle-Datei anpassen.

```
kotlinOptions {
    jvmTarget = '1.8'
    freeCompilerArgs += [
            "-Xjvm-default=all",
    ]
}
```

Es bietet sich an, an dieser Stelle einen *sync* vorzunehmen.

Schritt 3: Klassen für Datenhaltung anlegen

Nach diesen Vorbereitungen können die zur Datenhaltung relevanten Klassen angelegt werden.

1. Legen Sie zunächst ein Package **de.hsos.driverhelp.data** an.
2. In diesem Package werden die Klassen zur Verwaltung der Daten abgelegt:
    a. **Mileage.kt**: Beschreibung der Entität **mileage_table**. Diese enthält Spalten zur Speicherung der zurückgelegten Distanz (**distance**), des Benzinverbrauchs (**fuel**) und ein Feld (**info**), das aus Gründen der Datensparsamkeit und bessern Darstellbarkeit Informationen zu Datum, aktuellem Kilometerstand und Kosten zusammenfasst. Bsp.: *14.09.2022 | 225257 km | 79.3 €*

Ggf. kann für weitergehende Auswertungen diese Zeichenkette zerlegt werden.

b. **`MileageDAO.kt`**: Im DAO (*Data Access Object*) Interface geschieht das Mapping von CRUD-Operationen auf SQL-Queries. Bei komplexen Datentypen (z.B. *Date*) müssen ggf. *Typ-Konverter* bereit gestellt werden. In dem Bsp. hier werden keine solche Datentypen eingesetzt und insofern keine Konverter benötigt.
Betrachtet man die Methode:

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun insertMileage (mileage: Mileage)
```

So bedeutet `'suspend'`: die Funktion läuft in separatem Thread / Coroutine und kann unterbrochen werde. Nimmt man das Attribut weg, kommt es zu einer Exception (siehe Logcat) und die Ausführung wird unterbunden. Der Grund dafür ist, dass bei aufwändigen DB-Operationen ansonsten der Main-/UI-Thread blockieren kann und damit das UI hängt. Das ist unbedingt zu vermeiden.

c. **`MileageRepository.kt`**: Die Daten (DAO) werden in einem Repository verwaltet. gestellt. Hier werden also die Interface-Methoden des DAO aufgerufen. Bsp.:

```
@Suppress("RedundantSuspendModifier")
@WorkerThread
suspend fun insert(mileage: Mileage) {
    mileageDao.insertMileage(mileage)
}
```

Aufgrund der asynchronen Ausführung aller Queries in einem jeweils separaten Thread, kann ein *Observer-Observable-Pattern* mit eines **`Flow`**-Objektes (Attribut des Repositorys) umgesetzt werden. Der beobachtete Flow informiert alle registrierten Observer, wenn sich die Daten im Repository geändert haben.

```
val allMileages: Flow<List<Mileage>> = …
```

d. **`MileageRoomDatabase.kt`**: Die eigentliche Datenbank wird in einem *Singleton-Objekt* (in Kotlin: **`companion object`**) verwaltet, d.h. es gibt nur eine Instanz dieser Klasse. Ändert sich das Schema der Datenbank, so könnte hier ggf. auch eine automatische Migrationsprozedur hinterlegt werden. Das ist hier nicht notwendig.

e. **`MileageViewModel.kt`**: Die Daten werden über ein ‚*Shared ViewModel*' bereit gestellt. Das Repository und das UI sind über das ViewModel komplett getrennt.

Dazu nutzen wir sog. *LiveData*-Objekte, die Caching unterstützen. Auf Basis dieser Lösung können Observer hinzugefügt werden (z.B. View-Komponenten im UI), die auf Änderungen des Datenmodells reagieren können.

```
val allMileages: LiveData<List<Mileage>> =
      repository.allMileages.asLiveData()
```

3. Die entsprechenden Klassen können dem Beispiel aus der Vorlesung entnommen werden. Beim Kopieren sollten die Paket-Namen automatisch angepasst werden. Das ist aber zu kontrollieren.

Schritt 4: Klasse zur Editierung von Datenbank-Einträgen realisieren

Werden Daten in der *MainActivity* erfasst, so sollen diese in die Datenbank eingetragen werden. Dabei kann es aber natürlich zu fehlerhaften Eingaben kommen. Über eine Editier-Komponente sollen nach Auswahl eines Eintrages auch nachträglich Korrekturen an diesem bereits erfassten Datensatz vorgenommen werden können.

1. Legen Sie eine *Aktivität* **`EditMileageActivity`** an. Überlegen Sie sich, warum ein Fragment hier keinen Sinn macht.
2. Die Layout-Ressource sollte **`activity_edit_data.xml`** lauten.
3. Achten Sie darauf, dass die Aktivität im Manifest der App erscheint:

```
<activity android:name=".EditMileageActivity" />
```

Schritt 5: Applikation anlegen

Wie oben beschrieben, werden die Daten den Views über ein ‚*Shared ViewModel*' bereit gestellt. Das ViewModel soll applikationsweit zur Verfügung stehen.

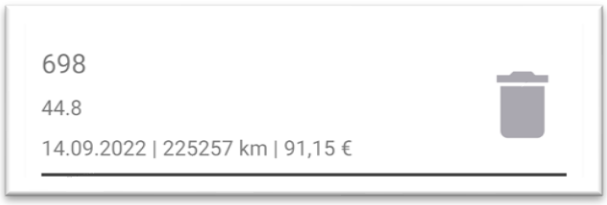
Legen Sie dazu im Paket **`de.hsos.driverhelp`** eine Klasse **`DriverHelpApplication`** an. In der Klasse müssen folgende Schritte ausgeführt werden:

```
// Anlegen eines Coroutinen-Scopes. Endet mit dem dem Prozess.
val applicationScope = CoroutineScope(SupervisorJob())

// 'by lazy' meint: Datenbank und Repository werden erst erzeugt,
// wenn sie benötigt werden. Nicht gleich beim Start der App.
val database by lazy { MileageRoomDatabase.getDatabase(this, applicationScope) }
val repository by lazy { MileageRepository(database.mileageDao()) }
```

Schritt 6: Fragment zur Anzeige der erfassten Daten

1. Legen Sie zunächst ein weiteres Fragment an. Das Vorgehen dazu ist bekannt.
2. Das Fragment sollte über einen Menü-Eintrag erreichbar sein. Per entsprechender Navigation soll das Fragment vom Hauptfragment erreichbar sein und eine Rückkehr dahin möglich sein.
3. Die erfassten Daten sollten in einem *RecyclerView* angezeigt werden. Einträge sollen in der folgenden Form angezeigt werden:



Das Einfügen von neuen Einträgen soll über die *MainActivity* erfolgen. Der Einfachheit halber sollte eine Eintragung immer dann vorgenommen werden, wenn ein neuer Verbrauch Ein Floating Button zum *Einfügen* ist damit *nicht notwendig*.
Des Weiteren sollen die folgenden Anforderungen umgesetzt werden.

a. Einträge sollen per Anklicken modifizierbar (via **`EditMileageActivity`**) sein.
b. Ein Löschen von Einträgen sollte ebenfalls möglich sein.

Optionale Aufgaben

- Es stellt sich die Frage, wofür man die aufgezeichneten Daten verwenden kann. Eine naheliegende Idee bestünde darin, den Durchschnittsverbrauch aus den erfassten Daten auszurechnen und den Wert an exponierter Stelle in der *MainActivity* darzustellen. Idealerweise wird das Datum in einem Shared ViewModel verwaltet.

- Die gespeicherten Einträge sollten noch im Hinblick auf die Konsistenz überprüft werden. Gibt es Lücken bei der Erfassung? Gibt es unplausible Eintragungen (z.B. ein zu viel zu hoher Verbrauch)? Wurden bzgl. der Datumsangaben oder Kilometerstände überlappende Eintragungen vorgenommen? …
  Idealerweise wird vor der Aufnahme eines Datensatzes in die Datenbank auf ein entsprechendes Problem aufmerksam gemacht.

Informationen und Hilfen

- Room Architecture: https://developer.android.com/topic/libraries/architecture/room.html
- Entwicklung einer Beispiel-Applikation mit Room (über ein Google *Codelab*): https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0
- Dynamic lists with RecyclerView: https://developer.android.com/develop/ui/views/layout/recyclerview
- Shared ViewModel: https://developer.android.com/codelabs/basic-android-kotlin-training-shared-viewmodel