FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Labo Gebruikersinterfaces
Reeks 8: Using room database

13 mei 2022

In the next sessions we will create an app that shows a list of notifications on the traffic in and around Ghent. The list will display each notification's type and date, as shown in Figure 1. The details of a traffic notification are shown when the user touches the corresponding list item. At first we will use a static json file "verkeersmeldingen.json" in the assets folder with sample notifications. Later, we will fetch live data from the Open Data Tank of the Ghent city council.

In this session we will focus on building a view showing a single item and the back-end.

You will learn the following concepts:

- Using API and Training documentation on https://developer.android.com
- Building layouts in XML: GridLayout
- Updating selected item through clicks
- Room: Building the Entity, Dao, database and repository

Android Studio provides sample code for many types of application flows (File -> New -> Activity -> Gallery...). This lab exercise can be accomplished with the 'Master/Detail Flow' sample code. While this sample code works and performs well, the auto-generated code is outdated and overcomplicated for the main goal of this session. To complete this exercise we will instead start from scratch so each task is clear and easy to understand.

1 Using Room back-end

In the first part of this session, we will define a layout with a GridLayout and use the skills from previous exercises to implement data binding and model view. At first we will use dummy traffic notifications.

1.1 Project set-up

- The start code project can be downloaded from Ufora.
- To help you, the gradle file is already configured

1.2 Notification item screen with GridLayout using ViewModel

We will design the master activity according to the MVVM model, but the focus of this exercise is on the "View" and "ViewModel" parts of this architecture. The start code provides the `MasterActivity` with an XML layout and it is set as the launcher activity.

1.2.1 Layout

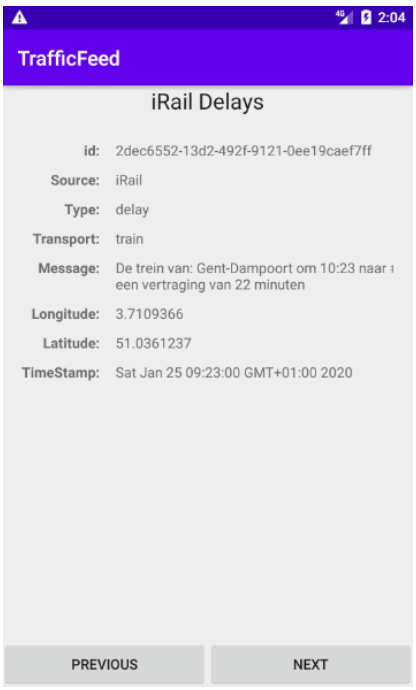
For this lab session, the layout file is given. All the properties of a traffic notification are displayed using a grid layout. Make sure you understand the different components of the layout.

1.2.2 The ViewModel

We will now create a first version of the `ViewModel`. A `ViewModel` contains all UI-related data. For instance, it keeps track of which fields are visible, and which model data should be shown to the user. Also, it translates commands from the UI controller (an `Activity` or `Fragment`) to method calls on the model which contains the business logic. The Android system manages the lifecycle of the `ViewModel`.

The `ViewModel` of our activity is very simple. This view model is responsible of holding the current selected item. We can use the item's properties directly with databinding.

- Create your own class that extends `AndroidViewModel`.
- Provide a field for the current selected item. You can set this property using the Kotlin object `DummyTrafficNotification`.
- Provide a method to select the next and previous item. For now these methods can stay empty.



Figuur 1: Example layout of the Traffic Notifications Application.

1.2.3 Binding the ViewModel

We will now bind the `text` attribute of our `TextViews` to the correct property of the selected item and handle the button presses in the view model.

- The layout XML file is already converted to the data binding layout and an instance of `ActivityMainBinding` is retrieved in `MainActivity`.
- Create a binding in the XML, using the @ syntax between the `text` attribute of our `TextView` and the `onClick` of the `Buttons`.
- Do not forget to create a `<data>` property for the `ViewModel` in your XML.
- In the `onCreate()` method of the activity, obtain a `ViewModel` component and assign the view model property in the binding class. For more information, see https://developer.android.com/topic/libraries/data-binding/architecture.

1.3 Room: Centralized data storage

1. Modify the `TrafficNotification` data class in the "db.model"package to a Room `Entity` . Use the id as the primary key for this database table.
2. Create a `Dao` interface for this `Entity` class in the “db” package. Provide the following annotated methods:
   - insertAll() which accepts an array of notifications.
   - deleteAll() to delete all notifications.
   - getAll() to get all traffic notifications.

   Warning: the methods of an interface need no implementation, be sure not to write curly braces ({}) behind the method declaration.
3. Create a `RoomDatabase` abstract class (named `TrafficNotificationDatabase`) in the “db” package which extends `RoomDatabase`. Annotate this class with the current database version and the Entities supported by the application.
4. Make a singleton of your `TrafficNotificationDatabase` class. Do this by adding a companion object with a method for retrieving a `TrafficNotificationDatabase` object. Make use of the `Room.databaseBuilder`
5. To repopulate the database every time the app is started, add a `RoomDatabase.Callback` and override `onCreate()`. (Experiment with the difference between overriding `onCreate()` and `onOpen()`.) Use a `Worker` and the `WorkManager` to populate the database on a background thread.
6. `InitializeDatabase` is a `Worker` class for populating the database from a JSON file. The code for reading the JSON file and converting it to a list of `TrafficNotifications` is given. Complete the method `doWork` in `InitializeDatabase` so that all `TrafficNotifications` in `list` are added to the database.
7. Create a repository class inside the “db” package. The repository will be simple for this lab session since we only have one data source (the room database).
8. The repository has one property that will hold all the notifications wrapped in a `LiveData` object. Get an instance of the database and assign the property with the `getAll` method from the DAO class. We can do this since the `getAll` method returns a list of `LiveData`

objects. The Room database executes all queries on a separate thread, observed `LiveData` will notify the observer on the main thread when the data has changed.

9. Create a repository in the `ViewModel` and get a list of the notifications from it.
10. Use the method `Transformations.map` to get the number of notifications. Every time the list of notifications changes this property will be updated.
    - **Hint**: In Kotlin, there is a convention: if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses, as can be seen in this code snippet:

      ```
      val user: LiveData<User> = ...
      val userName: LiveData<String> = Transformations.map(user)
      {"${it.name} ${it.lastName}"}}
      ```

    - **Hint**: inside the lambda expression, `it` refers to the argument of the lambda expression which is in this case `user`.
11. Set the first notification as the selected notification. Why should you do this inside the above `map` method?

    Solution: The notifications are loaded on a different thread than the UI. It is possible that the notifications are not loaded when the `ViewModel` is created.

    Keep the dummy notification as default.
12. The code inside the `map` method is only executed when one observes the result (the number of notifications: `notificationCount`). Since the layout does not use this, you should add an observer manually. You can do this by placing underlying code in the `onCreate` method in `MainActivity`.

    ```
    vm.notificationCount.observe(this, Observer{ })
    ```

    Make sure you import `androidx.lifecycle.Observer` and not `java.util.Observer`.
13. Navigate through the list when the user presses the previous or next button.

2 Opkuistip

Heb je al veel geoefend en begint de emulator al aardig vol te staan met (al dan niet afgewerkte) apps? Het kan nuttig zijn om eens grote kuis te houden. Open daarvoor de Device Manager, en kies voor de juiste emulator dan `Wipe Data`.