GHENT UNIVERSITY

Lab 3: Using Room database

Design and Development of Mobile Applications
E. De Coninck, S. Leroux, P. Simoens
2019-2020

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.

For the first 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 While this sample code works and performs well, the auto-generated code is outdated and overcomplicated for the

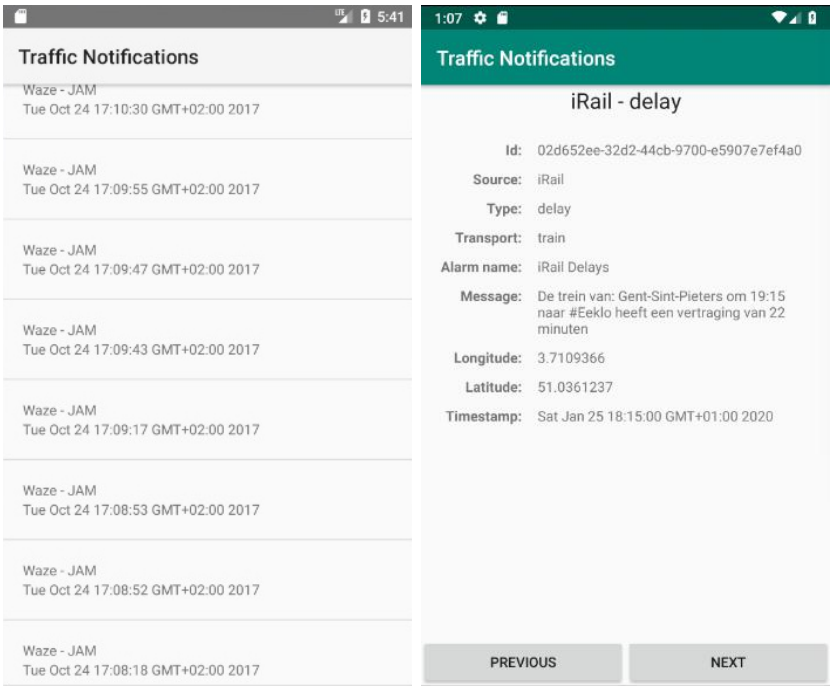

Figure 1: Example layout of the Traffic Notifications Application.

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. It includes `db.model.TrafficNotification.java`, `util.JsonUtils.java`, `assets/verkeersmeldingen.json` and `MainActivity.java` with a XML resource file.
- Configure your build.gradle file (app module):
  - Enable data binding
  - Add the dependencies for Lifecycle components. Pick the right options among the alternatives: we will use `LiveData`, `ViewModel` and `Room`.

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 model will be shared with the `RecyclerView` in the next lab session. The start code provides the `MasterActivity` with an XML layout and it is set as the launcher activity.

1.2.1 Layout declaration

- Modify the layout file so it contains a title and `ScrollView` with some properties of a traffic notification. Inside this `ScrollView` we can use a `GridLayout` to layout the properties with a label on the left side and the value on the right side.
- Add two buttons to cycle through the traffic notification items.

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 `ViewModel`.
  - Provide a field for the current selected item.
  - Provide a method to select the next and previous item.

1.2.3 Binding the ViewModel

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

- Bindings require getter methods for the members of your ViewModel. A convenient way is to have Android Studio generate these methods for you via Code > Generate > Getters.
- Convert your layout XML to data binding layout.
- 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, obtain an instance of the binding class and assign the view model property in the binding class. For more information, see https://developer.android.com/topic/libraries/data-binding/architecture.
  - Test this with dummy traffic notifications. You can use the provided getDummy() method in TrafficNotification to get a dummy notification.
- Add a log statement to the constructor of your ViewModel. Launch the app, and rotate your device. Check the log files and confirm that your Activity is re-created, but not your ViewModel upon rotating the device.

1.3 Room: Centralized data storage

The database package "db" of our application will consist of a few classes: model.TrafficNotification.java, TrafficNotificationDao.java, TrafficRepository.java and TrafficRoomDatabase.java. You are going to convert the announcements in the provided JSON file to a List of TrafficNotifications that is used to fill up the database when it's opened. For more information, see https://developer.android.com/training/data-storage/room/.

- The TrafficNotification class is a POJO class with a unique ID for each item and getters for all properties. The util.JsonUtils class has two methods: 1) to read a JSON file from the assets folder and return a JSONObject, 2) parse a JSONObject to a list of notification items.
  - Modify the TrafficNotification POJO class in the "db.model" package to a Room Entity. Use the id as the primary key for this database table.
- 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.
- Create a RoomDatabase abstract class in the "db" package which extends RoomDatabase. Annotate this class with the current database version and the Entities supported by the application.
  - Create and implement a singleton method getDatabase(Context context) which will instantiate the current class and create the database using the Room.databaseBuilder.
  - Currently we will insert items in the database on the main thread. To allow calling database queries on the main thread call allowMainThreadQueries() before building the Room database. Normally you would use an AsyncTask to fill up the database and LiveData to listen to database changes. This will be for later.

- Create a singleton repository class inside the "db" package. This will hold an instance of the database and provides methods which can be used by the view models. A Repository manages query threads and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in a local database.
  - The constructor will call the traffic notification dao class to insert items from the local json file. Normally you would decide to fetch data remotely or query the database. Paste the following code into your constructor to fill the database and save a cached version the list of notifications inside the repository:

```
TrafficRoomDatabase db = TrafficRoomDatabase.getDatabase(application);
TrafficNotificationDao dao = db.trafficNotificationDAO();
Log.d("repository", "before␣deleteAll()");
dao.deleteAll();
Log.d("repository", "after␣deleteAll()");
try {
  JSONObject json = JsonUtils.loadJSONFromAsset(application, "verkeersmeldingen.json");
  List<TrafficNotification> notifications = JsonUtils.parseJSON(json);
  Log.d("repository", "before␣insertAll()");
  dao.insertAll(notifications.toArray(new TrafficNotification[0]));
  Log.d("repository", "after␣insertAll()");
} catch (IOException e) {
  e.printStackTrace();
} catch (JSONException e) {
  e.printStackTrace();
}
allTrafficNotifications = trafficNotificationDao.getAllNotifications();
```

  - Provide a public method for the view models to get all traffic notifications. This will return the cached version in the repository so we do not have to query the database all the time.
- Update the `MainActivity` to create a new `Repository` and inject this repository in the `ViewModel`.
  - In `ViewModel` get a list of notifications when the repository is set.
  - Set the first item as the selected item.
  - Navigate through the list when the user presses the previous or next button.