ANDROID ALARM APP JACK INGHAM

**APIs used**

**ConstraintLayout + FloatingActionbutton**

ConstraintLayout allows greater freedom for designing the app. Elements are placed relative to one another and the parent. The floating action button is an advanced element that ‘floats’ above the other elements and is always present. In my app it is used for adding a new alarm. The constraintlayout allows this to gravitiate to the bottom right of the screen; in the common alternative LinearLayout, this would move up and down as it is attached to the recyclerview which had varying height depending on how many entries are contained.

**TimePickerDialog**

The time picker dialog is a predesigned user interface that allows a user to pick a time on a clock face, and returns a Calendar object that is then processed in the callback. This is obviously the most intuitive way for a user to specify a time for the alarm

**SimpleDateFormat**

SimpleDateFormat is an in-built object type in Java/Kotlin. It has many very useful included methods to parse dates in different formats. The primary purpose for this in the app is to make the alarms, which are all stored as milliseconds/epoch, human readable into clock format, and in some cases display the day of the week.

**PendingIntent**

A pendingIntent is an object that contains an Intent to be executed at a later point. In my app the pendingIntent then broadcasts this to the receiver. This is needed for the alarms to be handled once they go off.

**BroadcastReceiver**

This is needed to handle the pendingintents broadcasts that the alarms send. This is set up to call a handle alarm function depending on the intent’s action. This listener is registered with a filter to allow only certain broadcast to be received in the activity.

**AlarmManager**

AlarmManager is a system service that allows events to be run at a specific epoch time (milliseconds since 1970/1/1). In this case, the manager is passed a pendingIntent for code to be run upon the alarm firing. It can also be used to cancel alarms by identiofying them with the same pendingintent. The app uses this to set all the alarms specified by the user.

**Room/SQLite**

SQLite is a client-side database that does not require a full DBMS. There exists a SQLite API, but my app uses the Room persistence library for more concise code. The database consists of 3 main objects; the data entity, the database access object, and the database object. In the app, this is used to store the alarms in a database, and to query the database and delete/change alarms.

**Viewmodel/LiveData**

The viewmodel is an object that stores application data. This works with LiveData objects to update the data when a change is made, by setting up the viewmodel to observe any changes in the LIveData. The app uses this to display all the alarms in the database to the user, and stores them in a recyclerview.

**RecyclerView+Adapter**

A recyclerview is a memory-efficient list that consists of a series of ViewHolders controller by an adapter. There are only as many viewHolders as items visible, and these are re-used when the list is scrolled. In my app, the adapter sets each viewholder to have 2 textviews to display alarm information to user. Each viewholder also has an onClickListener set up to handle presses on list items, in this case to delete the alarm.

**Coroutines**

A coroutine is a function that runs in a different thread to the main program. This is useful when 2 operations may want to run at the same time and one of them may take longer than the other. For example, when the Room DAO runs queries, they must be run in a separate thread to prevent the UI from hanging if the operation takes longer than expected. The coroutines are run from the inbuilt lifecyclescope in the main activity, and when a coroutine is needed elsewhere (e.g. the recyclerview adapter) then a custom scope is created.

**NotificationBuilder/Manager**

The notificationBuilder is used to construct the components of a notification, like the title, text, and icon. This can then be passed to the NotificationManager system service to send the notification. This is used to send a notification to the user once the broadcast has been sent from an alarm.

**Discussion**

A repeated theme throughout the project is handing of the times of alarms. The user inputs their alarm time as a human readable 24h time, but alarmManager and the like all handle time as epoch, or the number of milliseconds (or even seconds, adding further confusion) since 1st January 1970. Whenever the time needed to be presented to the user, SimpleDateFormat was used:

val sdf = SimpleDateFormat("HH:mm EEEE")  
myViewHolder.tvAlarmTime.*text* = sdf.format(alarms[index].milliseconds)

HH:mm EEEE means the date will be formatted as ”Hour:Minute Day”. SDF allows the date to be formatted in many ways, for example using EEE instead of EEEE would return the 3-letter day (i.e. mon, tue, wed, etc.) for more compact use.

When the user picks a time using the TimePickerDialogue, the times are all on the current day. This means it would be possible for a user to set an alarm for a past time. They may also do this unintentionally, for example if they set an alarm for the current time, and the time advances before the alarm. This alarm would never be triggered as the time has already passed. This is handled by adding 24h to any alarm set before the current time:

alarmTimeMillis = calendarObject.*timeInMillis*;  
if (alarmTimeMillis < System.currentTimeMillis()){  
 alarmTimeMillis += 86400000 // adds 24 hours for alarms at times already passed in current day  
}

Because the alarm is working in milliseconds, when an alarm is set, the seconds are also considered. This causes an issue as an alarm set 1 second before another minute advances will fire 59 seconds after the time the user intended. To fix this, when the main activity receives the epoch, it ignores the seconds by taking advantage of the properties of the Long type:

val roundedMs = (newAlarmMs/60000)\*60000//removes the seconds from the time

Another issue that took some time to solve was passing in the respective alarm id into the broadcast receiver. When a new alarm is added to the room database, the insert function returns the automatically-generated primary key. I then used this as the request code for pendingintent passed into the alarm manager, thinking that when the broadcastReceiver is activated, it could retrieve this code in a similar way to how it retrieves the action. However this is not possible to be accessed from within the broadcast – indeed, in previous versions the documentation stated that the request code was not used! In order to solve this, rather than using just a string for the intent of the pendingIntent, I made an intent first and used addExtras to put in the ID, before passing this in:

val alarmIntent = Intent()  
when (newAlarmFrequency){  
 "Once" -> alarmIntent.setAction("alarmTask")  
 "Daily" -> alarmIntent.setAction("alarmTaskDaily")  
 "Weekly" -> alarmIntent.setAction( "alarmTaskWeekly")  
 else -> null  
}  
var newAlarmID = 0L  
*lifecycleScope*.*launch* **{** *withContext*(Dispatchers.IO) **{** val alarm1 = Alarm(milliseconds = roundedMs, frequency = newAlarmFrequency)  
 newAlarmID = db.alarmDao().insert(alarm1)  
 Log.d("TAG","ID updated to $newAlarmID")  
  
 **}  
}**  
alarmIntent.putExtra("alarmID", newAlarmID)  
val pi = PendingIntent.getBroadcast(this, newAlarmID.toInt(), alarmIntent, PendingIntent.*FLAG\_UPDATE\_CURRENT*)

This ID could then be retrieved inside the broadcast and passed into the handleAlarm() function:

val receiver = object: BroadcastReceiver() {  
 override fun onReceive(context: Context?, intent: Intent?) {  
 val firedAlarmID = intent!!.getLongExtra("alarmID", -1)  
 when (intent.*action*) {  
 "alarmTask" -> handleAlarm(0, firedAlarmID)  
 "alarmTaskDaily" -> handleAlarm(1, firedAlarmID)  
 "alarmTaskWeekly" -> handleAlarm(2, firedAlarmID)  
  
 }  
 }  
}

Notice when it is retrieved we use the !! operator to force dangerous access to this, as there is a possibility of the ID being null.

The first parameter of the handleAlarm function is the typecode. This tells the function whether the alarm is a one-off alarm, or if it should be reset to repeat daily/weekly. It also sends through which action the new intent should have

if ((typeCode == 1) || (typeCode == 2)) { //resetting the alarm  
 var newAlarmMs = 0L  
 val newAlarmIntent = Intent()  
  
 if (typeCode ==1){  
 newAlarmMs =(System.currentTimeMillis()+86400000)  
 newAlarmIntent.setAction("alarmTaskDaily")  
 } else if (typeCode ==2){  
 newAlarmMs = (System.currentTimeMillis()+604800000)  
 newAlarmIntent.setAction("alarmTaskWeekly")  
 }

The new alarm time is then updated into the database, rather than adding a new entry

*lifecycleScope*.*launch***{** *withContext*(Dispatchers.IO)**{** db.alarmDao().updateAlarmMs(alarmID, newAlarmMs)  
 **}  
}**

Because the recyclerview is using liveData, when the alarms are fired the user can see the changes. Because the viewmodel sorts the list to have the most recent alarm on top, when a looping alarm is fired it automatically moves down the list:

viewModel.getAllAlarms().observe(this, *Observer* **{** val sortedList = **it**.*sortedBy* **{it**.milliseconds**}**

When a user clicks on an alarm in the recyclerview, it is deleted. This is not possible to so identically as the main program for several reasons. The lifecyclescope used in the main activity is not usable in the MyAdapter adapter class, as it does not have a lifecycle like an activity. Instead, we code a custom scope for the coroutine:

var job = *Job*()  
override val coroutineContext: CoroutineContext  
 get() = Dispatchers.Main + job

The alarmManager is also not accessible in the normal way due to the change of context, and the pendingIntent has to be supplied with the different context:

val context = myViewHolder.itemView.*context*val pi = PendingIntent.getBroadcast(context, (alarms[index].id).toInt(), Intent("alarmTask"), PendingIntent.*FLAG\_UPDATE\_CURRENT*)  
val alarmMgr = context.getSystemService(Context.*ALARM\_SERVICE*) as AlarmManager  
alarmMgr.cancel(pi)

**Evaluation**

Whilst my app is functional and non-trivial – it does function as an alarm and send alerts at the given frequency – there are some obvious improvements to be made. The design of the app is lacking, using mostly plain, unstyled default elements. The main styled element is the floating action button on the main activity.

The current implementation uses a separate activity for the user to pick an alarm time. An improvement to this could be to make this into a popup box or dialog box that appears over the main activity. It could also be implemented with fragments.

The times that a user can currently pick for the alarm is quite restrictive in when they can be set and the frequency. An improvement would be to allow the user to pick any day for the alarm, or give the user checkboxes for whatever days the alarm is wanted to go off on. When the alarm is fired here, it would query the database and see when the alarm is next wanted and use the current epoch to work out the current day and how many milliseconds in the future to re-set the alarm to go off on.

Currently the alarm just sends a single notification when an alarm is triggered. A more alarm-like option would be for the app to send continuous notifications to the user at regular interval.

Most alarm apps include a snooze feature, which is missing in this app. This would be achieved by adding an amount of time to the alarm once fired, if prompted by the user.