Lab 4 - Bound Service

Android applications we wrote so far have all relied on direct interaction with the user and no “hidden” processing. Often, however, applications have to conduct some work in the background independently on the user interaction. For instance, a music player should play music even when you don’t interact with the app and when the screen is off; your emails should be downloaded without you actively refreshing the app, etc.

Android has a number of classes that help us with background processing. These include:

- **Service** - for long-running background tasks
- **IntentService** - for shorter background tasks
- **AsyncTasks** - for shorter background tasks tightly connected with the user interface
- **Workers** - for periodic background tasks
- **Threads**, **Handlers**, and **HandlerThreads** - lower-level classes for background processing
- Finally, in Kotlin we have **Coroutines** as yet another abstraction for background processing

In this lab we will see how we can perform long-running processing in the background and at the same time keep updating the UI using a **Bound Service**. Bound Service is a concept that includes a regular **Service** that is bound to another component (usually an **Activity** or a **Fragment**). This other component can then call methods of the **Service**.

We will write an app that counts seconds since a button was pushed. The counting should continue even if the app is not visible on the screen. We will see what happens if the counting code is in the Activity, then we will move it to the Service (so it runs in the background), and finally, we will run the Service in the foreground to ensure that the time is counted even when the app is closed.

This is going to be a rather long lab, so to speed things up, we will start with the initial code you can find here: https://bitbucket.org/pbdfrita/pbdfrita-pbd2023-lab-4

Processing in Activity

Open `MainActivity.kt` file. Find `runButtonClick`. This function is called when a user clicks the button. The Boolean variable `isTimerRunning` tells us whether the timer is ticking or not. If not, the `startTimer` function is called to take note of the starting time. The `updateUiStartRun` is then called to set the text of the button to “`Stop`”, so that the user can stop the count using the same button.

Run the app - you will notice that besides the button text change, nothing else happens.

We need to refresh the **TextView** every once in a while to see the time ticking. For that, we will use a **Handler**. This class is associated with a **Thread** (in this case the main thread) and contains a queue where messages can be posted and later processed. We can post messages with a delay equal to our desired UI refresh delay (say 1000 milliseconds) and when

handling them simply update the **TextView** and post another message to refresh the **TextView** again after a given delay.

The code already contains the `updateTimeHandler` that represents our **Handler**. Uncomment the code where the **Handler** is defined and instantiated and the code where messages are posted and cleared from the **Handler** queue (in `updateUIStartRun` and `updateUIStopRun`). MSG_UPDATE_TIME is a constant code that our Handler sends as a message - we only use it to keep track of what is the content of the message, but we don't really use it. The delay at which the handler re-sends the message, UPDATE_RATE_MS, is more significant. Spend some time till you understand what's going on here.

Test the application again - you should see the timer ticking when you click on the start button. However, if you exit the app or rotate the screen, the timer will be reset. This is not the desired behavior - let's fix it!

Processing in Service and Binding

Open `TimerService.kt` file. This is a **Service** that will be started from the **Activity** and the **Activity** will bind to the **Service**. When a user clicks on the button this **Service** will note the starting time of the click. The **Activity** will also query the **Service** periodically to get information on how many seconds have elapsed since the starting moment.

Move the fields `startTime`, `endTime`, and `isTimerRunning` to **TimerService** class. Set these to zero or false (as appropriate). Move functions `startTimer`, `stopTimer`, and `elapsedTime` from **MainActivity** to **TimerService** class.

The **Service** can now do the counting, but we need to communicate with it somehow. For that, we should create a **Binder** in the **Service** and return it when an **Activity** binds to **Service**, which is signaled by `onBind` method. Define a new internal class called **RunServiceBinder** that extends **Binder**, it should look like this:

```
inner class RunServiceBinder : Binder() {

   val service: TimerService

       get() = this@TimerService

}
```

Pay attention to this structure - we have a `service` property for which we are redefining the getter.

When the **Activity** gets this binder it will call `service` to get a reference to a running service. In **TimerService** instantiate a private variable `serviceBinder` and assign a new instance of **RunServiceBinder** to it. Return this `serviceBinder` from `onBind` (instead of returning `null`).

The **Service** should be started when the **Activity** starts. In `onStart` create an `Intent i` with `TimerService:class.java` component and start it using `startService(i)`. To bind the **Activity** to the **Service**, we need a **ServiceConnection** object which will ensure that we get a reference to a **RunServiceBinder** "channel" to our **Service**. First, create two more fields in **MainActivity**: `timerService` of type `TimerService?` and `serviceBound`, a Boolean set to `false` initially. Then uncomment the code that defines and instantiates a **ServiceConnection** object.

We just have to fix a few methods that take care of the UI in **MainActivity**. `runButtonClick` is used to check whether the timer is running and if not, it would start the timer, otherwise it would stop the timer. However, the timer is now moved to the **Service**, thus, you should fix the function so that it checks

whether the service is bound (use `serviceBound` field) and whether `timerService.isTimerRunning()` is `false` or `true` and then either start the timer (using `timerService.startTimer()`) or stop the timer (using `timerService.stopTimer()`). Pair these calls with `updateUIStartRun()` and `updateUIStopRun()` as before. `updateUITimer` should be fixed to get the elapsed time from `timerService.elapsedTime()` but only if the service is bound. Finally, when the **Activity** is stopped it should unbind the **Service** - write the unbinding code in `onStop`:

```
if (serviceBound) {

   unbindService(mConnection)

   serviceBound = false

}
```

Run the app now. You will see that rotating the screen or exiting and re-entering the app does not prevent the timer from ticking. Unlike the **Activity**, the **Service** does not get stopped when a user navigates back from the app.

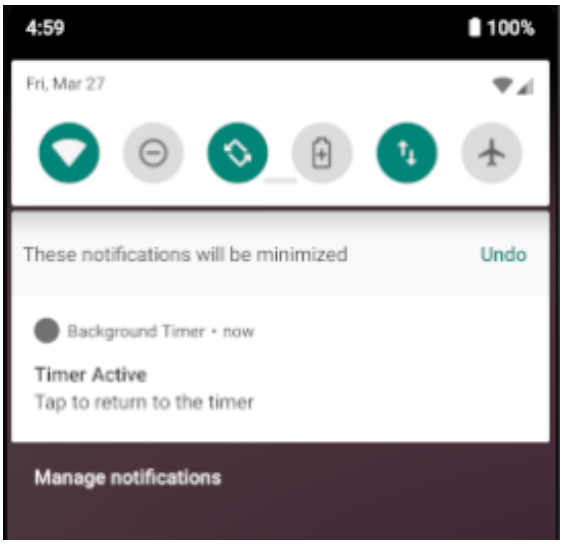
Processing in Foreground Service

Nevertheless, there is still a high chance that a **Service** is killed if it runs in the background. Furthermore, a user should probably know whether the timer is ticking or not, even if the app is not open. After all, you have an indicator that a music player is playing music even when you don't use the app directly - there is a persistent notification in the notification bar that tells you that the player is active. This notification indicates that a so-called "foreground service" is active. We will now move our **Service** to the foreground when a user exits the **Activity**.

First, we need permission to run a **Service** in the foreground. Open the `AndroidManifest.xml` file and add the following line just before the application tag:

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

Next, we need a notification to show when the service is in the foreground. Go to **TimerService** and uncomment the code for creating a notification. The code might look daunting, but it's really not that complex. First, you have to `createNotificationChannel`. Starting from Android API 26 you need to specify one or more channels that your app uses for notifications. This is so that users can block certain channels and allow some others (e.g. block "commented" but allow "liked" notifications from the Facebook app). Here we create a single channel with the ID `channelID` and use it to show our notification - call `createNotificationChannel` at the end of `onCreate` in the **Service**.



The second function `createNotification` builds a notification and attaches a **PendingIntent** to it. This **PendingIntent** will start the **MainActivity** once a user clicks on the notification.

To start a notification in the foreground let's define a `foreground()` function in our **Service** and put `startForeground(NOTIFICATION_ID, createNotification())` in it. The first parameter is a constant integer of your choice. To move the **Service** to the background create a `background()` function and put `stopForeground(true)` in it.

Who decides whether the **Service** should run in the foreground or not? The **MainActivity**. If the **MainActivity** is bound to the **Service** that means that the user is actively interacting with the app, thus the **Service** should be in the background. In `onServiceConnected` add `timerService.background()` immediately after you finish binding the **Service**.

If the user is leaving the **MainActivity** and the timer is running, the **Service** should be moved to the foreground. If the user is leaving the **MainActivity** but the timer is not running, then the **Service** should be stopped. Thus, in `onStop`, if the Service is bound, check whether the service is running (`timerService?.isTimerRunning`) and if so, move the service to the foreground (`timerService?.foreground()`). Otherwise, stop the service using `stopService(Intent(this, TimerService::class.java))`

Test the app. When you start the counter and exit the **MainActivity** you should get a notification indicating that the **Service** is still alive and ticking.

Foreground Service Notification Action Button

Let's finish the lab by adding an action button to the notification. This button will let the user quickly stop the service. We will stop the service by sending an Intent as if we are starting the Service. However, we will add an action of type `ACTION_STOP`. Any **Intent** to start the **Service** calls `onStartCommand`. Thus, in this function we will check whether the **Intent** that called the **Service** contains an action of type `ACTION_STOP`. If so, we will stop the **Service**.

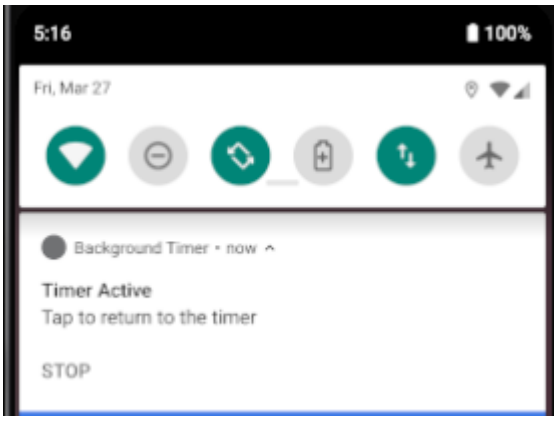
In `createNotification` define the **Intent** and add the action to it:

```
val actionIntent = Intent(this, TimerService::class.java)

actionIntent.action = ACTION_STOP

val actionPendingIntent =
PendingIntent.getService(this, 0, actionIntent,
PendingIntent.FLAG_UPDATE_CURRENT)
```

Then, create an action button in the notification by adding:

```
      .addAction(android.R.drawable.ic_media_pause,
"Stop", actionPendingIntent)
```

at the end of the notification building expression.



To stop the service when `ACTION_STOP` arrives, in `onStartCommand` check whether the **Intent** contains `ACTION_STOP` action (use `intent.action`) and if so, call `stopForeground(true)` and `stopSelf()` to stop the **Service**. Note that we should also modify the **Intent** to start the **Service** in **MainActivity** by giving it a different action. E.g. add

```
i.action = ACTION_START
```

in `onStart` of the **MainActivity**.

Finally, test the code. When a user clicks on the button the timer should tick. When the user leaves the app, a notification should demonstrate that the **Service** is still active. Finally, clicking on "`STOP`" will stop the **Service**.

If you did not attend and work on this solution during the in-person lab slots, your code must be committed to a private repository named **PBD2023-LAB-4** in your Bitbucket account. User **pbdfrita** must be added as a read-only member of this repository. The code must be committed by Sunday (April 2nd) 23:59.

Happy coding!