# MODUL 8
# Services, WorkManager, and Notifications

## THEME DESCRIPTION

This module will introduce students to the concepts of managing long-running tasks in the background of an app.

## WEEKLY LEARNING OUTCOME (SUB-LEARNING OUTCOME)

Students will be able to trigger a background task, create a notification for the user when a background task is complete, and launch an application from a notification

## TOOLS/SOFTWARE USED

- Android Studio

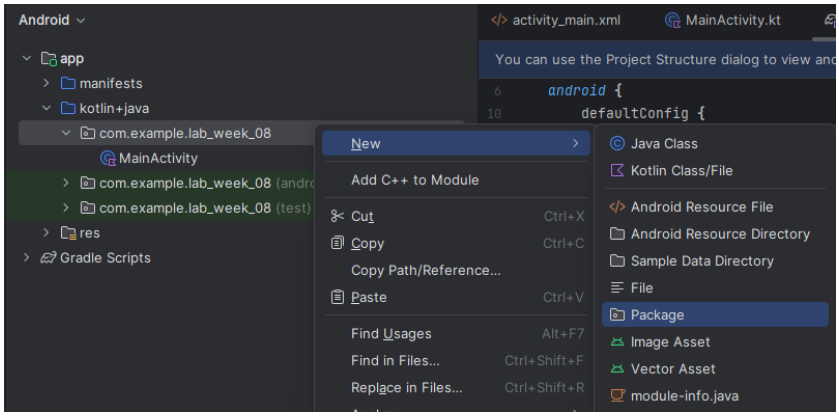
## PRACTICAL STEPS

### Part 1 - Executing background work with the WorkManager class

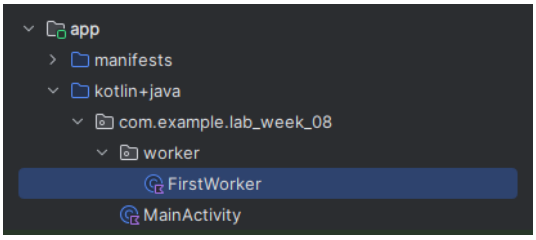
1. Open Android Studio and click **New Project.**
2. Choose the **Empty Views Activity** to start with.
3. Name your project **“LAB_WEEK_08”**.
4. Set the minimum SDK to **“API 24: Android 7.0 (Nougat)”**.
5. Click **Finish**, and let your android application build itself.
6. In this part, we will be focusing on how we can **create a worker for our background process** in Android. First, import the necessary **Dependencies** to your **build.gradle.kts (Module :app)** and don’t forget to **Gradle Sync**.

```
implementation(libs.androidx.work.runtime)
```

7. Next, create a new path inside your package and name it **“com.example.lab_week_08.worker”.**




8. Inside the new package, add a new **Kotlin Class File** and name it **FirstWorker.kt.**




9. To make our new class file into a worker class, update your **FirstWorker.kt** to the code below.

```
class FirstWorker(
    context: Context, workerParams: WorkerParameters
) : Worker(context, workerParams) {
    //This function executes the predefined process based on the input
    //and return an output after it's done
    override fun doWork(): Result {
        //Get the parameter input
        val id = inputData.getString(INPUT_DATA_ID)

        //Sleep the process for 3 seconds
        Thread.sleep(3000L)

        //Build the output based on process result
        val outputData = Data.Builder()
            .putString(OUTPUT_DATA_ID, id)
            .build()

        //Return the output
        return Result.success(outputData)
```

```
    }
    companion object {
        const val INPUT_DATA_ID = "inId"
        const val OUTPUT_DATA_ID = "outId"
    }
}
```

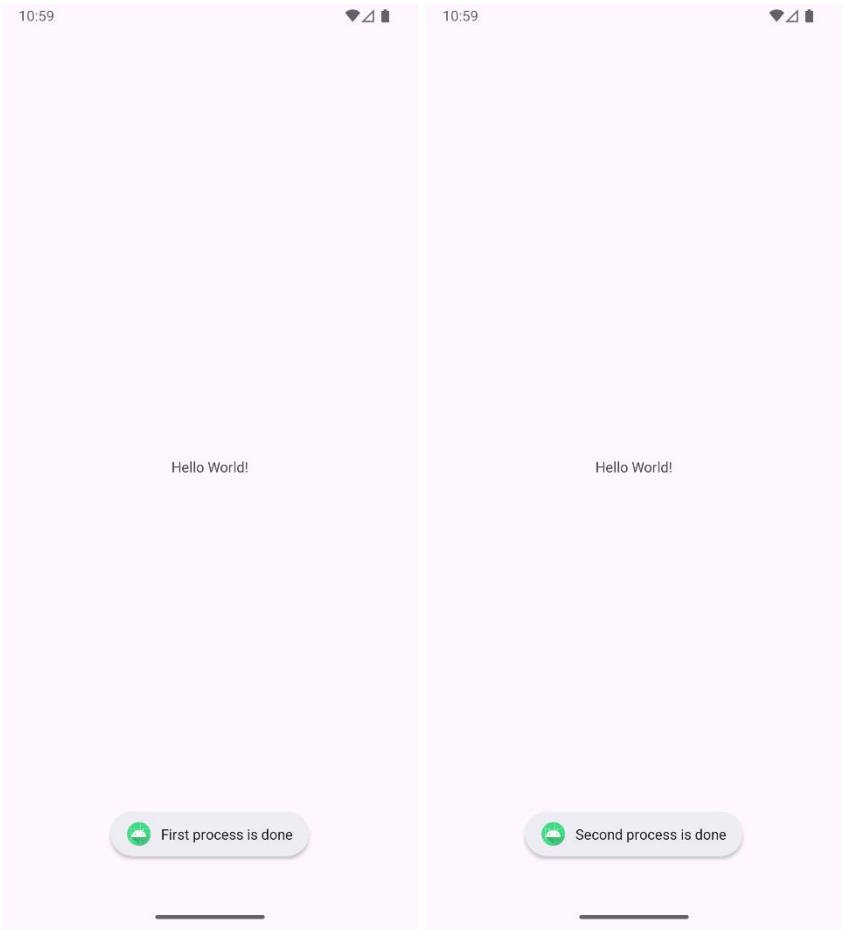
10. You may see that we're using `Thread.sleep(3000L)` to simulate a **long and heavy background process**. Since this is only a tutorial, we're just gonna pretend that the 3 seconds is the amount of time the process will take.
11. We've just made **one worker.** Remember that we can make **multiple workers**. Let's repeat the above process and make another worker named **SecondWorker.kt**. Update the code just like in **Step 9** but change the **class name** to **SecondWorker**.
12. Our workers are done, now let's move on to **MainActivity.kt**. Update it to the code below. You may notice keywords like **WorkManager**, **OneTimeWorkRequest**, **Constraints**. Make sure to read all the comments in the provided code snippet to learn more about it.

```
class MainActivity : AppCompatActivity() {
    //Create an instance of a work manager
    //Work manager manages all your requests and workers
    //it also sets up the sequence for all your processes
    private val workManager = WorkManager.getInstance(this)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v,
insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right,
systemBars.bottom)
            insets
        }
        //Create a constraint of which your workers are bound to.
        //Here the workers cannot execute the given process if
        //there's no internet connection
        val networkConstraints = Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .build()
```

```
        val id = "001"

        //There are two types of work request:
        //OneTimeWorkRequest and PeriodicWorkRequest
        //OneTimeWorkRequest executes the request just once
        //PeriodicWorkRequest executed the request periodically

        //Create a one time work request that includes
        //all the constraints and inputs needed for the worker
        //This request is created for the FirstWorker class
        val firstRequest = OneTimeWorkRequest
            .Builder(FirstWorker::class.java)
            .setConstraints(networkConstraints)
            .setInputData(getIdInputData(FirstWorker
                .INPUT_DATA_ID, id)
            ).build()

        //This request is created for the SecondWorker class
        val secondRequest = OneTimeWorkRequest
            .Builder(SecondWorker::class.java)
            .setConstraints(networkConstraints)
            .setInputData(getIdInputData(SecondWorker
                .INPUT_DATA_ID, id)
            ).build()

        //Sets up the process sequence from the work manager instance
        //Here it starts with FirstWorker, then SecondWorker
        workManager.beginWith(firstRequest)
            .then(secondRequest)
            .enqueue()

        //All that's left to do is getting the output
        //Here, we receive the output and displaying the result as a toast message
        //You may notice the keyword "LiveData" and "observe"
        //LiveData is a data holder class in Android Jetpack
        //that's used to make a more reactive application
        //the reactive of it comes from the observe keyword,
        //which observes any data changes and immediately update the app UI

        //Here we're observing the returned LiveData and getting the
        //state result of the worker (Can be SUCCEEDED, FAILED, or CANCELLED)
        //isFinished is used to check if the state is either SUCCEEDED or FAILED
        workManager.getWorkInfoByIdLiveData(firstRequest.id)
            .observe(this) { info ->
```

```
            if (info.state.isFinished) {
                showResult("First process is done")
            }
        }
        workManager.getWorkInfoByIdLiveData(secondRequest.id)
            .observe(this) { info ->
            if (info.state.isFinished) {
                showResult("Second process is done")
            }
        }
    }

    //Build the data into the correct format before passing it to the worker as
input
    private fun getIdInputData(idKey: String, idValue: String) =
        Data.Builder()
            .putString(idKey, idValue)
            .build()

    //Show the result as toast
    private fun showResult(message: String) {
        Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
    }
}
```

13. Run your application, after 3 seconds the **"First process is done"** toast should appear, then after another 3 seconds the **"Second process is done"** toast should appear.




**COMMIT to GITHUB at this point**
**Commit Message: "Commit No. 1 – add first & second worker"**

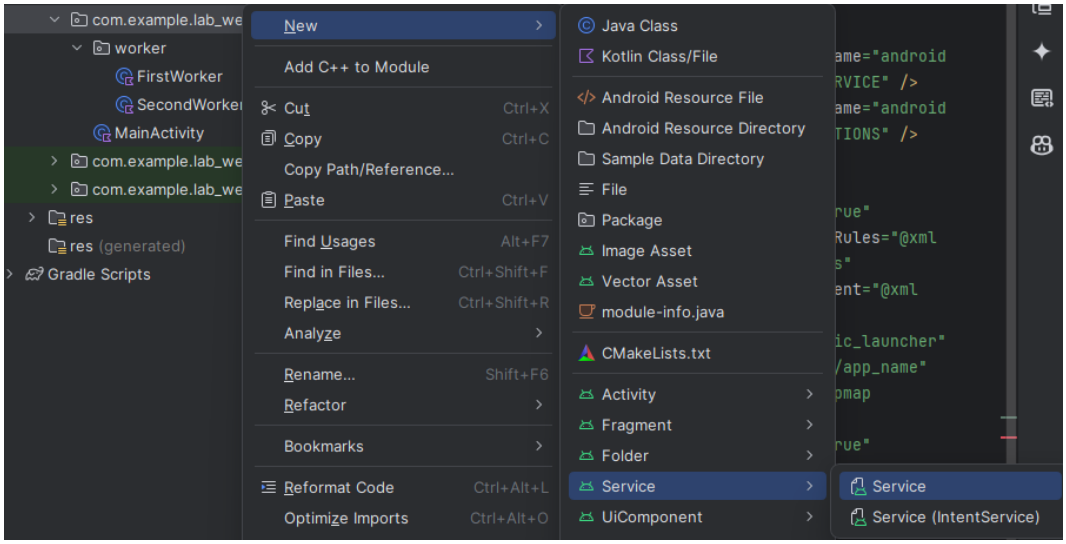
## Part 2 - Tracking your Background Work with a Foreground Service

1. Continue your "**LAB_WEEK_08"** project.
2. Now that you have successfully implemented the **Workers** to do your background processes, we will now create a **Foreground Service** to track those hidden background processes in Android. Before you move on, you should at least know the difference between a **Service** and a **Worker**:
    - **Worker** - Mainly for background processes and the system automatically executes the process in a different thread than the main thread (Main thread is the thread that controls the UI).
    - **Service** - Can be used for foreground or background processes and it executes the process on the main thread on default. If you want to use a different thread, this has to be done manually.

3. First, let's add the necessary permissions and service to your **AndroidManifest.xml**. Update it to the code below.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

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

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.LAB_WEEK_08"
        tools:targetApi="31">
        <service
            android:name=".NotificationService"
            android:enabled="true"
            android:exported="false"
            android:foregroundServiceType="dataSync"/>
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

4. Next, create a new **Service Class File** called **NotificationService**. Update it to the code below. You may notice that the code snippet is quite long and complicated, **Make Sure** that you **Read** all of the **Comments** available in the code snippet.




```
class NotificationService : Service() {
   //In order to make the required notification, a service is required
   //to do the job for us in the foreground process

   //Create the notification builder that'll be called later on
   private lateinit var notificationBuilder: NotificationCompat.Builder
   //Create a system handler which controls what thread the process is being
executed on
   private lateinit var serviceHandler: Handler

   //This is used to bind a two-way communication
   //In this tutorial, we will only be using a one-way communication
   //therefore, the return can be set to null
   override fun onBind(intent: Intent): IBinder? = null

   //this is a callback and part of the life cycle
   //the onCreate callback will be called when this service
   //is created for the first time
   override fun onCreate() {
       super.onCreate()

       //Create the notification with all of its contents and configurations
       //in the startForegroundService() custom function
       notificationBuilder = startForegroundService()

       //Create the handler to control which thread the
       //notification will be executed on.
       //'HandlerThread' provides the different thread for the process to be
executed on,
       //while on the other hand, 'Handler' enqueues the process to HandlerThread
to be executed.
       //Here, we're instantiating a new HandlerThread called "SecondThread"
```

```
        //then we pass that HandlerThread into the main Handler called
serviceHandler
        val handlerThread = HandlerThread("SecondThread")
            .apply { start() }
        serviceHandler = Handler(handlerThread.looper)
    }

    //Create the notification with all of its contents and configurations all set up
    private fun startForegroundService(): NotificationCompat.Builder {
        //Create a pending Intent which is used to be executed
        //when the user clicks the notification
        //A pending Intent is the same as a regular Intent,
        //The difference is that pending Intent will be
        //executed "Later On" and not "Immediately"
        val pendingIntent = getPendingIntent()

        //To make a notification, you should know the keyword 'channel'
        //Notification uses channels that'll be used to
        //set up the required configurations
        val channelId = createNotificationChannel()

        //Combine both the pending Intent and the channel
        //into a notification builder
        //Remember that getNotificationBuilder() is not a built-in function!
        val notificationBuilder = getNotificationBuilder(
            pendingIntent, channelId
        )

        //After all has been set and the notification builder is ready,
        //start the foreground service and the notification
        //will appear on the user's device
        startForeground(NOTIFICATION_ID, notificationBuilder.build())
        return notificationBuilder
    }

    //A pending Intent is the Intent used to be executed
    //when the user clicks the notification
    private fun getPendingIntent(): PendingIntent {
        //In order to create a pending Intent, a Flag is needed
        //A flag basically controls whether the Intent can be modified or not later
on
        //Unfortunately Flag exists only for API 31 and above,
        //therefore we need to check for the SDK version of the device first
        //"Build.VERSION_CODES.S" stands for 'S' which is the API 31 release name
```

```
        val flag = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S)
FLAG_IMMUTABLE else 0

        //Here, we're setting MainActivity into the pending Intent
        //When the user clicks the notification, they will be
        //redirected to the Main Activity of the app
        return PendingIntent.getActivity(
            this, 0, Intent(
                this,
                MainActivity::class.java
            ), flag
        )
    }

    //To make a notification, a channel is required to
    //set up the required configurations
    //A notification channel includes a couple of attributes:
    //channel id, channel name, and the channel priority
    private fun createNotificationChannel(): String =
        //Unfortunately notification channel exists only for API 26 and above,
        //therefore we need to check for the SDK version of the device.
        //"Build.VERSION_CODES.O" stands for 'Oreo' which is the API 26 release name
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            //Create the channel id
            val channelId = "001"
            //Create the channel name
            val channelName = "001 Channel"
            //Create the channel priority
            //There are 3 common types of priority:
            //IMPORTANCE_HIGH - makes a sound, vibrates, appears as heads-up
notification
            //IMPORTANCE_DEFAULT - makes a sound but doesn't appear as heads-up
notification
            //IMPORTANCE_LOW - silent and doesn't appear as heads-up notification
            val channelPriority = NotificationManager.IMPORTANCE_DEFAULT

            //Build the channel notification based on all 3 previous attributes
            val channel = NotificationChannel(
                channelId,
                channelName,
                channelPriority
            )
```

```
            //Get the NotificationManager class
            val service = requireNotNull(
                ContextCompat.getSystemService(this,
                    NotificationManager::class.java)
            )
            //Binds the channel into the NotificationManager
            //NotificationManager will trigger the notification later on
            service.createNotificationChannel(channel)

            //Return the channel id
            channelId
        } else { "" }

    //Build the notification with all of its contents and configurations
    private fun getNotificationBuilder(pendingIntent: PendingIntent, channelId:
String) =
        NotificationCompat.Builder(this, channelId)
            //Sets the title
            .setContentTitle("Second worker process is done")
            //Sets the content
            .setContentText("Check it out!")
            //Sets the notification icon
            .setSmallIcon(R.drawable.ic_launcher_foreground)
            //Sets the action/intent to be executed when the user clicks the
notification
            .setContentIntent(pendingIntent)
            //Sets the ticker message (brief message on top of your device)
            .setTicker("Second worker process is done, check it out!")
            //setOnGoing() controls whether the notification is dismissible or not
by the user
            //If true, the notification is not dismissible and can only be closed by
the app
            .setOngoing(true)

    companion object {
        const val NOTIFICATION_ID = 0xCA7
        const val EXTRA_ID = "Id"

        //this is a LiveData which is a data holder that automatically
        //updates the UI based on what is observed
        //It'll return the channel ID into the LiveData after
        //the countdown has reached 0, giving a sign that
        //the service process is done
```

```
    private val mutableID = MutableLiveData<String>()
    val trackingCompletion: LiveData<String> = mutableID
  }
}
```

5. The code above basically sets up the **Notification Configuration** needed for your app. Now let's start the notification with all its contents to your device. Add the code below after your **getNotificationBuilder** method and before your **companion object** in **NotificationService.kt**. Make sure to also read all the comments.

```
//This is a callback and part of a life cycle
//This callback will be called when the service is started
//in this case, after the startForeground() method is called
//in your startForegroundService() custom function
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int
{
   val returnValue = super.onStartCommand(intent,
       flags, startId)

   //Gets the channel id passed from the MainActivity through the Intent
   val Id = intent?.getStringExtra(EXTRA_ID)
       ?: throw IllegalStateException("Channel ID must be provided")

   //Posts the notification task to the handler,
   //which will be executed on a different thread
   serviceHandler.post {
       //Sets up what happens after the notification is posted
       //Here, we're counting down from 10 to 0 in the notification
       countDownFromTenToZero(notificationBuilder)
       //Here we're notifying the MainActivity that the service process is
done
       //by returning the channel ID through LiveData
       notifyCompletion(Id)
       //Stops the foreground service, which closes the notification
       //but the service still goes on
       stopForeground(STOP_FOREGROUND_REMOVE)
       //Stop and destroy the service
       stopSelf()
   }
```

```
    return returnValue
}

//A function to update the notification to display a count down from 10 to
0
private fun countDownFromTenToZero(notificationBuilder:
NotificationCompat.Builder) {
    //Gets the notification manager
    val notificationManager = getSystemService(NOTIFICATION_SERVICE) as
NotificationManager

    //Count down from 10 to 0
    for (i in 10 downTo 0) {
        Thread.sleep(1000L)
        //Updates the notification content text
        notificationBuilder.setContentText("$i seconds until last warning")
            .setSilent(true)
        //Notify the notification manager about the content update
        notificationManager.notify(
            NOTIFICATION_ID,
            notificationBuilder.build()
        )
    }
}

//Update the LiveData with the returned channel id through the Main Thread
//the Main Thread is identified by calling the "getMainLooper()" method
//This function is called after the count down has completed
private fun notifyCompletion(Id: String) {
    Handler(Looper.getMainLooper()).post {
        mutableID.value = Id
    }
}
```

6. The code above sets up what to do when the **Foreground Service** is **Started** up. In this case, it tells the **Handler** to **Post** the Notification and let it countdown from 10 to 0 afterward. It also detects if the countdown has reached 0, then the **Notification** will be **Closed** and the **Service** will be **Stopped** and **Destroyed**.

7. Now let's call the **Service Class** that we've just made in your **MainActivity.kt**. First, let's request for notification permission (required for Android 13 (API 33) and above) Add the code below inside **onCreate**.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContentView(R.layout.activity_main)
    ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets
->
        val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
        v.setPadding(systemBars.left, systemBars.top, systemBars.right,
systemBars.bottom)
        insets
     }
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
       if (checkSelfPermission(android.Manifest.permission.POST_NOTIFICATIONS) !=
PackageManager.PERMISSION_GRANTED) {

requestPermissions(arrayOf(android.Manifest.permission.POST_NOTIFICATIONS), 1)
       }
    }
```

8. Add the code below after **showResult** function.

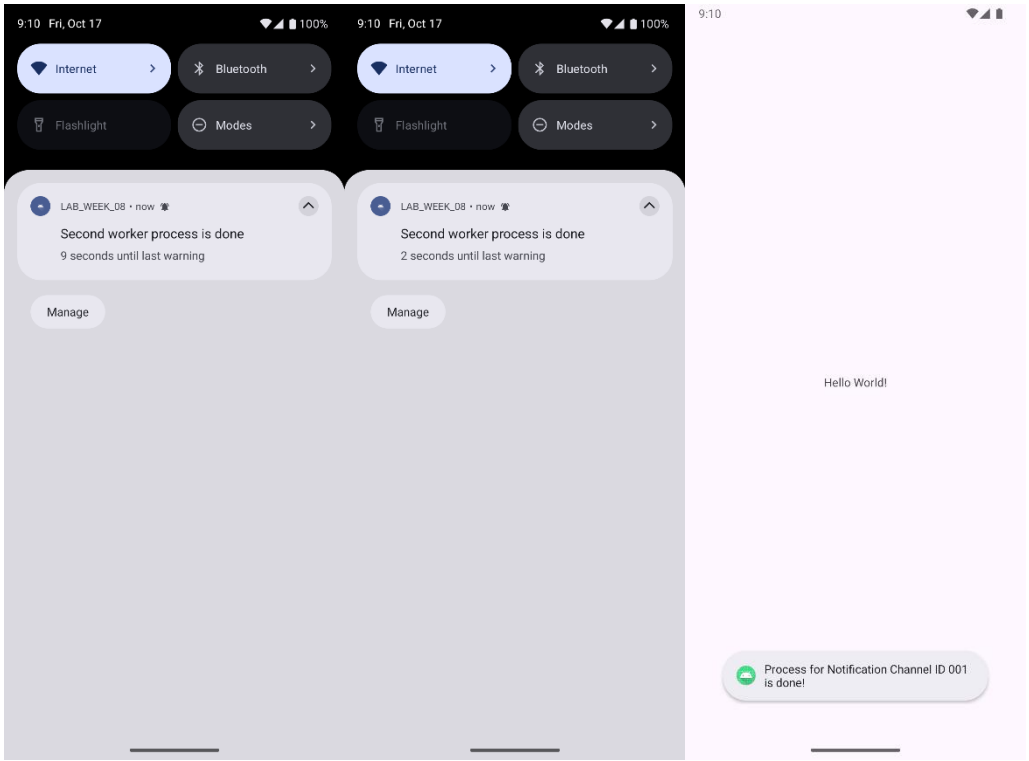
```
//Launch the NotificationService
private fun launchNotificationService() {
   //Observe if the service process is done or not
   //If it is, show a toast with the channel ID in it
   NotificationService.trackingCompletion.observe(
       this) { Id ->
       showResult("Process for Notification Channel ID $Id is done!")
   }

   //Create an Intent to start the NotificationService
   //An ID of "001" is also passed as the notification channel ID
   val serviceIntent = Intent(this,
       NotificationService::class.java).apply {
       putExtra(EXTRA_ID, "001")
   }

   //Start the foreground service through the Service Intent
   ContextCompat.startForegroundService(this, serviceIntent)
}

companion object{
```

```
    const val EXTRA_ID = "Id"
}
```

9. Last but not least, call the **launchNotificationService** method after your **SecondWorker** is done.

```
workManager.getWorkInfoByIdLiveData(secondRequest.id)
    .observe(this) { info ->
    if (info.state.isFinished) {
        showResult("Second process is done")
        launchNotificationService()
    }
}
```

10. Run your application and after the **Second Worker Toast** has appeared, a **Notification** should appear in your device counting down from 10 to 0, then giving a **Toast** at the end that says "**Channel id 001 process is done!**" (Don't forget to allow the app to send notifications). **Note**: Re-run the application if the notification seems to appear late



**COMMIT to GITHUB at this point**
**Commit Message: "Commit No. 2 – add first foreground service"**

## ASSIGNMENT

Continue your **LAB_WEEK_08** project. Add **1** more **Worker** named **ThirdWorker** and **1** more **Foreground Service** named **SecondNotificationService**. Make sure that **SecondNotificationService** is called after the **ThirdWorker** is done. Here's a detailed chronological order of the processes:

1. **FirstWorker** executed
2. **SecondWorker** executed
3. **NotificationService** executed
4. **ThirdWorker** executed
5. **SecondNotificationService** executed

Feel free to change the **Notification Count Down** timer to avoid any toast collisions.

**COMMIT to GITHUB at this point**
**Commit Message: "Commit No. 3 – add third worker & second foreground service"**