Mobile Applications

7. Background work, notifications, and alarms in Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction

- Android apps often need to perform work:
  - In the background (outside the UI)
  - At a later time
- This unit introduces several elements for that:
  - Services: traditional way for background tasks
  - `WorkManager`: new API for background tasks
  - Notifications: nessages displayed outside the app UI to inform the user
  - Alarms: for time scheduling

Table of contents

2. Services

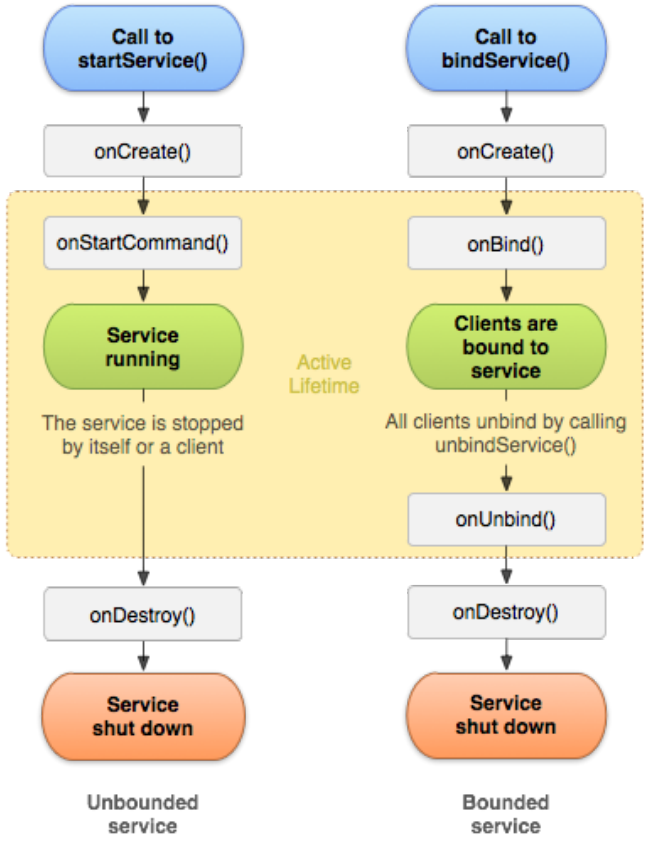
- A **Service** is a type of app component for long-running operations without a UI
- There are two types of Services in Android:

1. **Started service**. There are two types:
   - Background: Does not requires a notification
     - For example, and app can use a background service to compact its storage
   - Foreground: Requires a notification
     - For example, an audio app that use a foreground service to play an audio track
2. **Bound service**. Allows interaction with other components
   - For example, and app that offers a service to other apps

https://developer.android.com/guide/components/services

2. Services

- The lifecycle for started and bound services is as follows:



https://developer.android.com/guide/components/services

2. Services

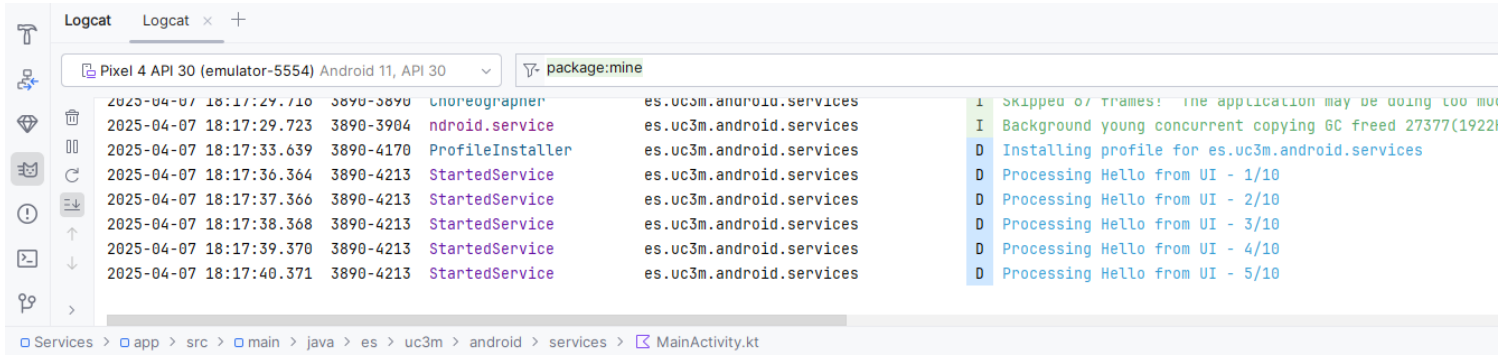
- To create services in Java/Kotlin:
  - We need to extend the `Service` class and register it in the manifest
  - Bound services must override the lifecycle method `onBind()`
    - This method returns an instance of `IBinder` which is an interface that allows clients to invoke methods defined by the service
  - In started services, the `onBind()` method should return `null`
- To start services in Java/Kotlin:
  - Background services are started by calling with the context method `startService()`
  - Foreground services requires are started by calling with the context method `startForegroundService()` and display a notification
  - Bound services are initiated by calling the context method `bindService()`

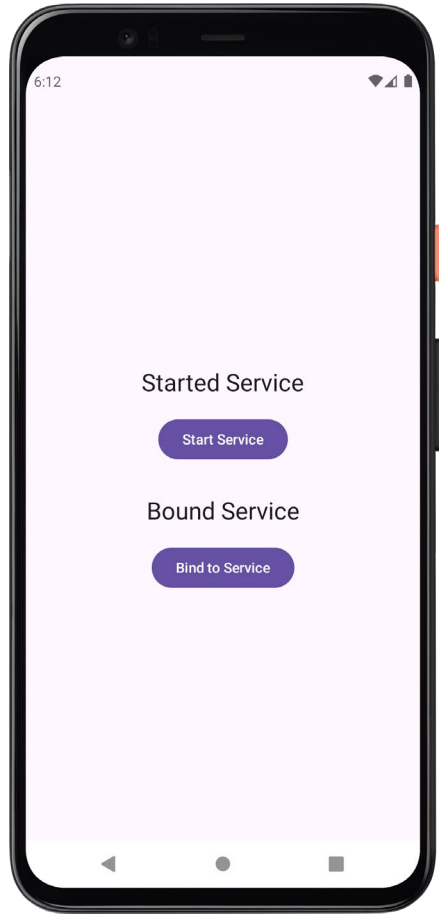
https://developer.android.com/develop/background-work/services/bound-services

2. Services

- The following demo created an started (background) service and a bound service:

The background service simply receives a message from the UI and log it 10 times





2. Services

A companion object allows us to define members (properties and functions) that belong to the class itself rather than to instances of the class

```
class StartedService : Service() {
    companion object {
        const val TAG = "StartedService"
        const val EXTRA_INPUT = "extra_input"
    }

    override fun onBind(intent: Intent?): IBinder? = null

    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        val input = intent?.getStringExtra(EXTRA_INPUT) ?: getString(R.string.no_input)

        CoroutineScope(Dispatchers.IO).launch {
            for (i in 1..10) {
                @SuppressLint("StringFormatMatches")
                Log.d(TAG, getString(R.string.processing, input, i))
                delay(1000)
            }
            stopSelf(startId)
        }

        return START_NOT_STICKY
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d(TAG, getString(R.string.service_destroyed))
    }
}
```

Returns `null` because this is a started service (not a bound service)

We uses a coroutines to run tasks on a background thread (not the main thread)

To shut down the service when done

If Android kills the service, it won't restart automatically (unlike *START_STICKY*)

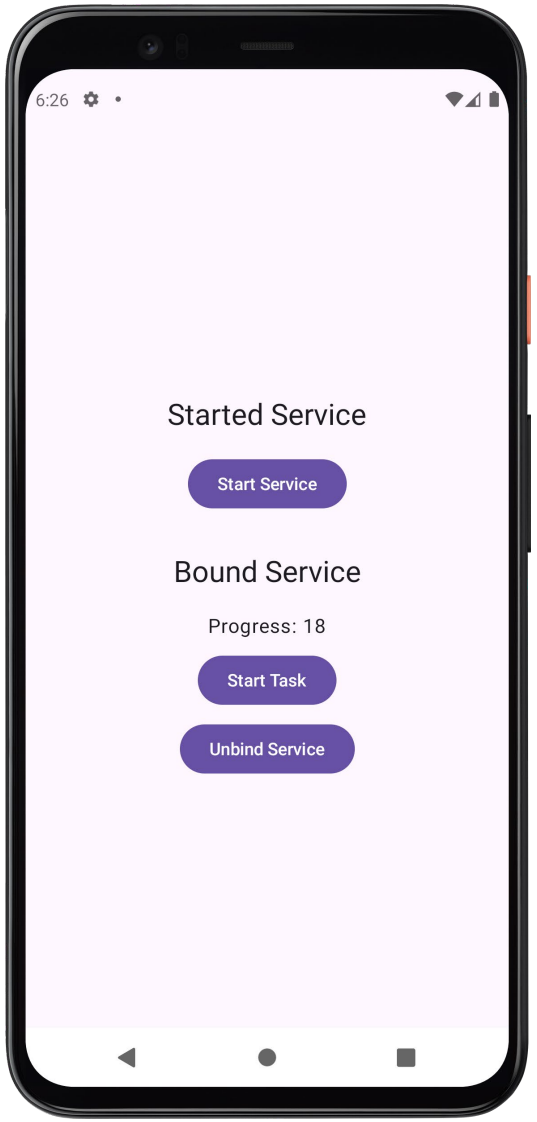
We need to declare our service (as app component) in the manifest file

```
<service
    android:name=".StartedService"></service>
```

2. Services


```
class BoundService : Service() {
    private val binder = LocalBinder()
    private val _progress = MutableStateFlow(0)
    val progress: StateFlow<Int> = _progress

    inner class LocalBinder : Binder() {
        fun getService(): BoundService = this@BoundService
    }

    override fun onBind(intent: Intent): IBinder = binder

    fun startTask() {
        CoroutineScope(Dispatchers.IO).launch {
            for (i in 1..100) {
                delay(100)
                _progress.value = i
            }
        }
    }
}
```

The second service in this demo is a bound service that exposes a integer value every 100ms and allows clients to start a task that updates this progress



2. Services


```
@Composable
fun ServiceDemoApp(modifier: Modifier = Modifier) {
    var serviceBound by remember { mutableStateOf(false) }
    var boundService: BoundService? by remember { mutableStateOf(null) }
    val context = LocalContext.currentQ

    val connection = remember {
        object : android.content.ServiceConnection {
            override fun onServiceConnected(
                name: android.content.ComponentName?, service: IBinder?
            ) {
                val binder = service as BoundService.LocalBinder
                boundService = binder.getService()
                serviceBound = true
            }

            override fun onServiceDisconnected(name: android.content.ComponentName?) {
                serviceBound = false
            }
        }
    }

    // ...
```

Creates a `ServiceConnection` object to handle binding lifecycle

```
    // Bound Service
    Text(stringResource(R.string.bound_service), style = MaterialTheme.typography.headlineSmall)
    Spacer(modifier = Modifier.height(16.dp))

    if (serviceBound) {
        val progress by boundService?.progress?.collectAsState() ?: mutableIntStateOf(0)

        Text(stringResource(R.string.progress, progress))
        Spacer(modifier = Modifier.height(8.dp))
        Button(onClick = { boundService?.startTask() }) {
            Text(stringResource(R.string.start_task))
        }
        Spacer(modifier = Modifier.height(8.dp))
        Button(onClick = {
            context.unbindService(connection)
            serviceBound = false
            boundService = null
        }) {
            Text(stringResource(R.string.unbind_service))
        }
    } else {
        Button(onClick = {
            val intent = Intent(context, BoundService::class.java)
            context.bindService(intent, connection, BIND_AUTO_CREATE)
        }) {
            Text(stringResource(R.string.bind_to_service))
        }
    }
}
```

When bound: displays progress (collected as state from the service)

When not bound: shows a button to bind to the service

Table of contents

3. WorkManager

- Modern Android discourages direct background execution
- The recommended solution in JetPack Compose is **WorkManager**
- Benefit:
  - More battery-efficient compared to traditional background services
- Features:
  - Runs even if app is closed
  - Supports retries and constraints
- Use cases:
  - Sync data
  - Periodic tasks

https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started

3. WorkManager

- To use WorkManager, we need the following dependency in our project:

build.gradle.kts (app)

```
dependencies {
    implementation(libs.androidx.work.runtime.ktx)

}
```

libs.version.toml

```
[versions]
workManager = "2.11.1"

[libraries]
androidx-work-runtime-ktx = { group = "androidx.work", name = "work-runtime-ktx", version.ref = "workManager" }
```

3. WorkManager

- In Kotlin, we can define a background task (*worker*) for `WorkManager` by extending `CoroutineWorker`
- The worker logic is implemented in `doWork()`, which is the method `WorkManager` executes in the background

```
class MyWorker(
    appContext: Context, workerParams: WorkerParameters
) : CoroutineWorker(appContext, workerParams) {

    override suspend fun doWork(): Result {
        val message = inputData.getString(KEY_MESSAGE).orEmpty()
        Log.d(TAG, "Started: $message")
        for (i in 1..5) {
            Log.d(TAG, "Step $i/5 - $message")
            delay(1000)
        }
        Log.d(TAG, "Finished: $message")
        return Result.success()
    }

    companion object {
        const val TAG = "DemoWorker"
        const val KEY_MESSAGE = "key_message"
    }
}
```

In this example, the worker reads an input message, writes progress messages to Logcat, waits one second in each iteration to simulate work

3. WorkManager


```
Button(onClick = {
    val request = OneTimeWorkRequestBuilder<MyWorker>().setInputData(
        Data.Builder().putString(MyWorker.KEY_MESSAGE, oneTimeMsg).build()
    ).build()

    oneTimeWorkId = request.id
    workManager.enqueue(request)
}) {
    Text(stringResource(R.string.start_one_time_work))
}

Button(onClick = {
    val request = PeriodicWorkRequestBuilder<MyWorker>(
        15, TimeUnit.MINUTES
    ).setInputData(
        Data.Builder().putString(
            MyWorker.KEY_MESSAGE, periodicMsg
        ).build()
    ).build()
    workManager.enqueueUniquePeriodicWork(
        PERIODIC_WORK_NAME, ExistingPeriodicWorkPolicy.UPDATE, request
    )
}) {
    Text(stringResource(R.string.start_periodic_work))
}

Button(onClick = {
    workManager.cancelUniqueWork(PERIODIC_WORK_NAME)
}) {
    Text(stringResource(R.string.stop_periodic_work))
}
```

The minimum interval for periodic works is **15 minutes**

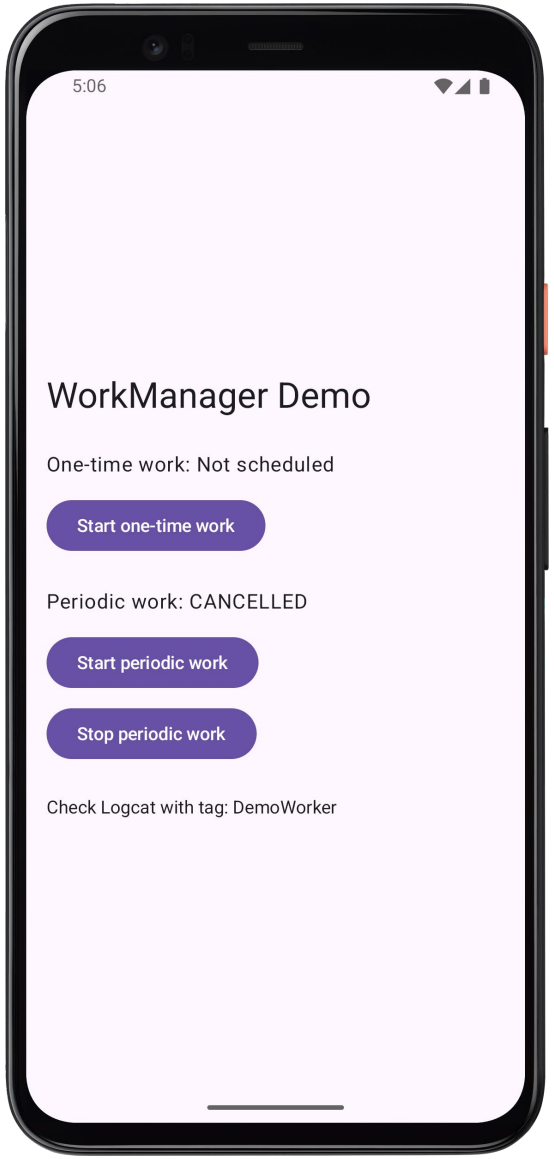
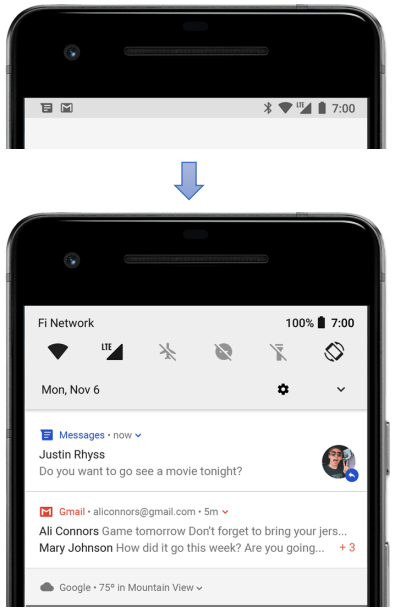


Table of contents

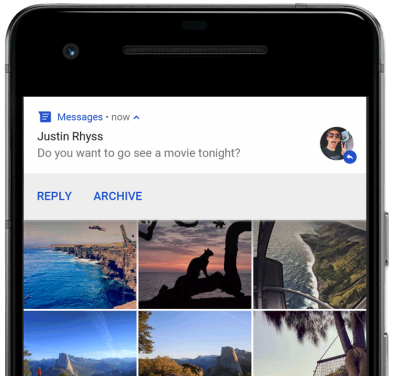
4. Notifications

- A notification is a message that Android displays outside an app's UI to provide the user with reminders or other information
- There are different formats for Android notifications:

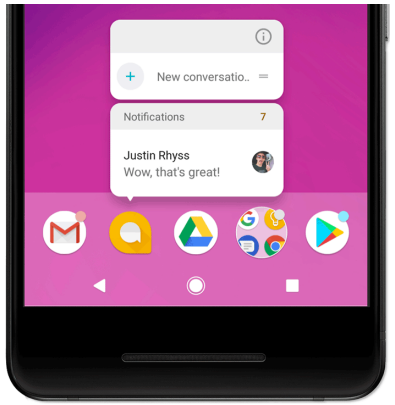
Status bar



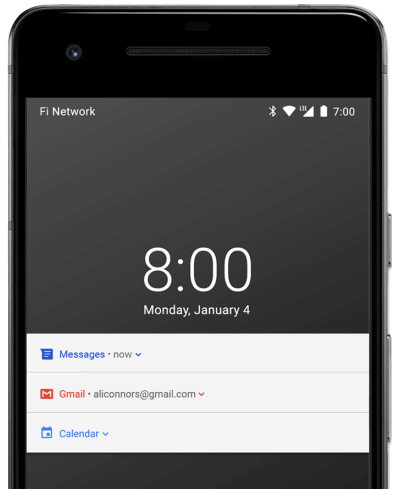
Heads-up notification



App icon badge



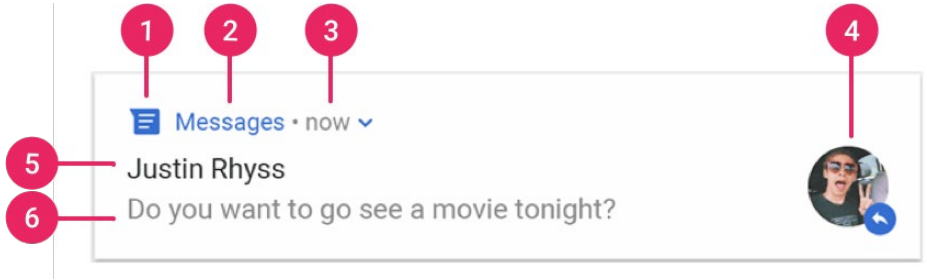
Lock screen



https://developer.android.com/guide/topics/ui/notifiers/notifications

4. Notifications - Anatomy

- The design of a notification has different elements:
  1. Small icon: required; set using `setSmallIcon()` in the notifications builder
  2. App name: provided by the system
  3. Time stamp: provided by the system. It can be overridden using `setWhen()`
  4. Large icon: optional; set using `setLargeIcon()`
  5. Title: optional; set using `setContentTitle()`
  6. Text: optional; set using `setContentText()`

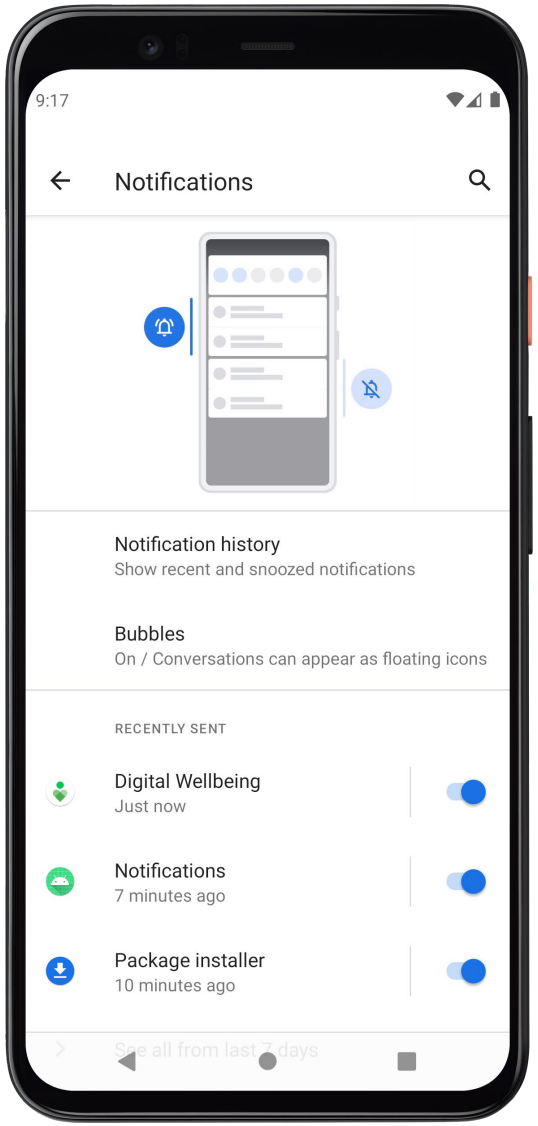


Nevertheless, some of these elements are not available in specific Android devices

https://developer.android.com/guide/topics/ui/notifiers/notifications

4. Notifications - Channels

- Starting in Android 8.0 (Oreo), all notifications must be assigned to a channel
  - A channel is a categorization mechanism that allows us to group notifications into different types based on their content, importance, or other factors
- Users can customize different aspects of notifications (Settings →App & notifications → Notifications)
  - For instance, users can disable notifications of specific apps



4. Notifications - Status bar

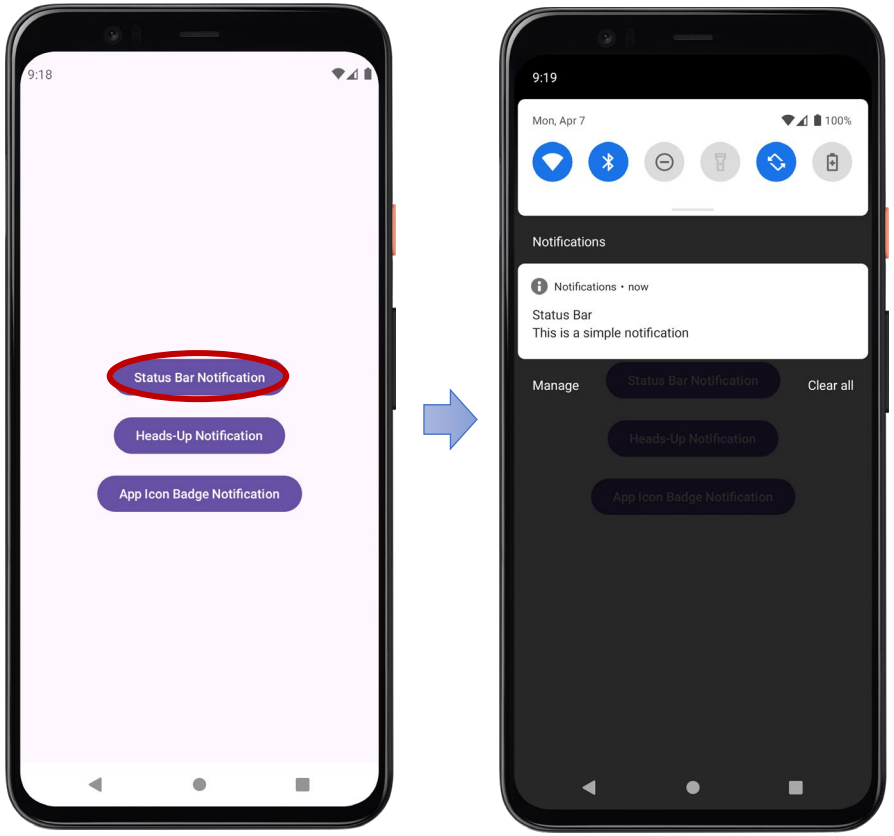
- The examples repository contains a project implementing different kinds of notifications

First, we create the notifications channels

The channel is composed by an identifier, a name, an importance level, and a description

```
private fun createNotificationChannels() {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is not in the Support Library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        // Channel for standard notifications
        val standardChannel = NotificationChannel(
            STANDARD_CHANNEL_ID,
            STANDARD_CHANNEL_NAME,
            NotificationManager.IMPORTANCE_DEFAULT
        ).apply {
            description = STANDARD_CHANNEL_DESCRIPTION
        }

        // Channel for heads-up notifications
        val headsUpChannel = NotificationChannel(
            HEADS_UP_CHANNEL_ID,
            HEADS_UP_CHANNEL_NAME,
            NotificationManager.IMPORTANCE_HIGH // Required for heads-up
        ).apply {
            description = HEADS_UP_CHANNEL_DESCRIPTION
        }

        // Register the channels with the system
        val notificationManager: NotificationManager =
            context.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(standardChannel)
        notificationManager.createNotificationChannel(headsUpChannel)
    }
}
```

4. Notifications - Status bar



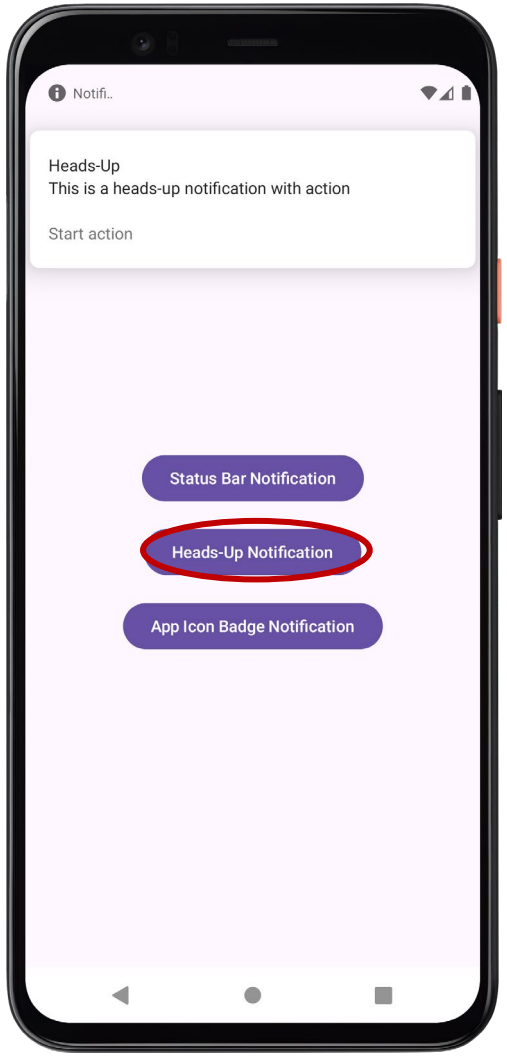
Then, at some point we launch the notifications

```
@SuppressLint("MissingPermission")
fun statusBarNotification(title: String, content: String) {
    val builder = NotificationCompat.Builder(context, STANDARD_CHANNEL_ID)
        .setSmallIcon(android.R.drawable.ic_dialog_info)
        .setContentTitle(title)
        .setContentText(content)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)

    with(NotificationManagerCompat.from(context)) {
        notify(NOTIFICATION_ID, builder.build())
    }
}
```

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

This permission is required in the manifest

4. Notifications - Heads-up




```
@SuppressLint("MissingPermission")
fun headsUpNotification(title: String, content: String) {
    val builder = NotificationCompat.Builder(context, HEADS_UP_CHANNEL_ID)
        .setSmallIcon(android.R.drawable.ic_dialog_info)
        .setContentTitle(title)
        .setContentText(content)
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setFullScreenIntent(null, true)
        .addAction(R.drawable.ic_launcher_foreground,
            context.getString(R.string.start_action), getPendingIntent())
        .setAutoCancel(true)

    with(NotificationManagerCompat.from(context)) {
        notify(NOTIFICATION_ID + 1, builder.build())
    }
}

fun getPendingIntent(): PendingIntent {
    val intent = Intent(Intent.ACTION_DIAL, "tel:666555444".toUri())
    return PendingIntent.getActivity(context, 0, intent, PendingIntent.FLAG_IMMUTABLE)
}
```

A `PendingIntent` is a token we give to a foreign app (e.g. `NotificationManager`) which allows this app to execute a given intent

4. Notifications - App icon badge


```
@SuppressLint("MissingPermission")
fun badgeNotification(title: String, content: String) {
    val notification = NotificationCompat.Builder(context, STANDARD_CHANNEL_ID)
        .setSmallIcon(R.drawable.baseline_notifications_24)
        .setContentTitle(title)
        .setContentText(content)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
        .setContentIntent(getPendingIntent())
        .setAutoCancel(true)
        .setNumber(5) // This makes the badge appear
        .setBadgeIconType(NotificationCompat.BADGE_ICON_SMALL)
        .build()

    with(NotificationManagerCompat.from(context)) {
        notify(NOTIFICATION_ID + 2, notification)
    }
}
```

The app icon badge (also known as *notification dot*) is a visual indicator displayed on the app's icon to convey certain information or notifications to the user

4. Notifications - Lock screen

- Notifications can appear on the lock screen as of Android 5
  - This feature can be useful, for example, in messaging apps
- To control the level of detail visible in the notification from the lock screen, call `setVisibility()` and specify one of the following values:
  - `VISIBILITY_PUBLIC`: the notification full content shows on the lock screen
  - `VISIBILITY_SECRET`: no part of the notification shows on the lock screen
  - `VISIBILITY_PRIVATE`: only basic information, such as the notification icon and the content title, shows on the lock screen. The notification full content doesn't show

We see an example of lock screen notification in the alarms section

https://developer.android.com/develop/ui/views/notifications/build-notification#lockscreenNotification

Table of contents

5. Alarms

- Alarms allows us to perform scheduled tasks, i.e., time-based operations outside the lifetime of an Android app
  - For example: an alarm clock, calendar reminder
- There are two main types of alarms:
  - One-time alarms: Triggered at a single specified time in the future. Once the alarm goes off, it is automatically canceled
  - Repeating alarms: Triggered repeatedly at regular intervals. It can be cancelled programmatically

```
val alarmManager = context.getSystemService(Context.ALARM_SERVICE) as AlarmManager
```

Alarms are managed using the class `AlarmManager`

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

This permission is required in the manifest

https://developer.android.com/training/scheduling/alarms

5. Alarms

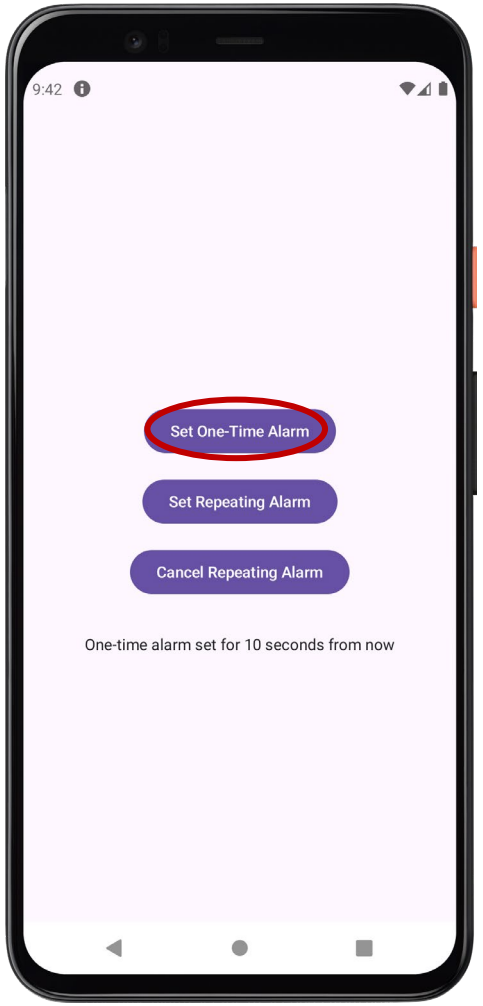
- The examples repository contains an app using a couple of basic alarms

```
class AlarmsHelper(private val context: Context, private val alarmManager: AlarmManager) {

    fun setOneTimeAlarm() {
        // Set the alarm to trigger 10 seconds from now
        val triggerTime = System.currentTimeMillis() + 10_000

        alarmManager.setExactAndAllowWhileIdle(
            AlarmManager.RTC_WAKEUP, triggerTime, getPendingIntent()
        )
    }

    private fun getPendingIntent(): PendingIntent {
        val intent = Intent(context, AlarmReceiver::class.java).apply {
            putExtra(MSG_KEY, context.getString(R.string.repeating_alarm_msg))
        }
        val pendingIntent = PendingIntent.getBroadcast(
            context, REPEATING_ALARM_REQUEST_CODE, intent, PendingIntent.FLAG_IMMUTABLE
        )
        return pendingIntent
    }

}
```

Schedules an exact alarm to trigger at a precise time, even if the device is *asleep* (or *Doze mode*, i.e., attempting to conserve battery)

Alarms can be triggered depending on this value:

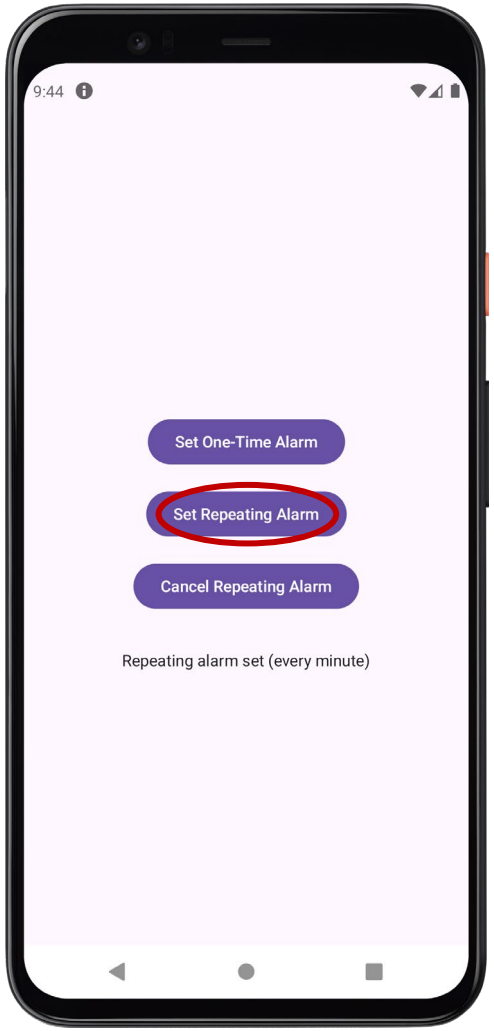
- *ELAPSED_REALTIME* : Based on the amount of time since the device was booted. It does not wake the device up if it is asleep
- *ELAPSED_REALTIME_WAKEUP* : Based on the amount of time since the device was booted. It wake the device up if it is asleep
- *RTC* : Using absolute time. It does not wake the device up if it is asleep
- *RTC_WAKEUP* : Using absolute time. It wake the device up if it is asleep



5. Alarms


```
fun setRepeatingAlarm() {
    // Set the alarm to start approximately 10 seconds from now and repeat every minute
    val triggerTime = System.currentTimeMillis() + 10_000
    val repeatInterval = 60_000L // 1 minute in milliseconds

    alarmManager.setRepeating(
        AlarmManager.RTC_WAKEUP, triggerTime, repeatInterval, getPendingIntent()
    )
}

fun cancelRepeatingAlarm() {
    val intent = Intent(context, AlarmReceiver::class.java)
    val pendingIntent = PendingIntent.getBroadcast(
        context,  REPEATING_ALARM_REQUEST_CODE, intent, PendingIntent.FLAG_IMMUTABLE
    )

    alarmManager.cancel(getPendingIntent())
    pendingIntent.cancel()
}
```

When using `setRepeating()`, Android synchronizes multiple repeating alarms and fires them at the same time (to reduce the use of the battery). Therefore, the repeating interval is not exact



5. Alarms

- We can enable a screen lock (Settings →Security → Screen lock) to see the lock screen notification in this demo app

```
    val builder = NotificationCompat.Builder(context, CHANNEL_ID)
        .setSmallIcon(android.R.drawable.ic_dialog_info)
        .setContentTitle(context.getString(R.string.alarm_notification))
        .setContentText(message)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
        .setContentIntent(pendingIntent)
        .setVisibility(NotificationCompat.VISIBILITY_PUBLIC)

    with(NotificationManagerCompat.from(context)) {
        notify(NOTIFICATION_ID, builder.build())
    }
}
```

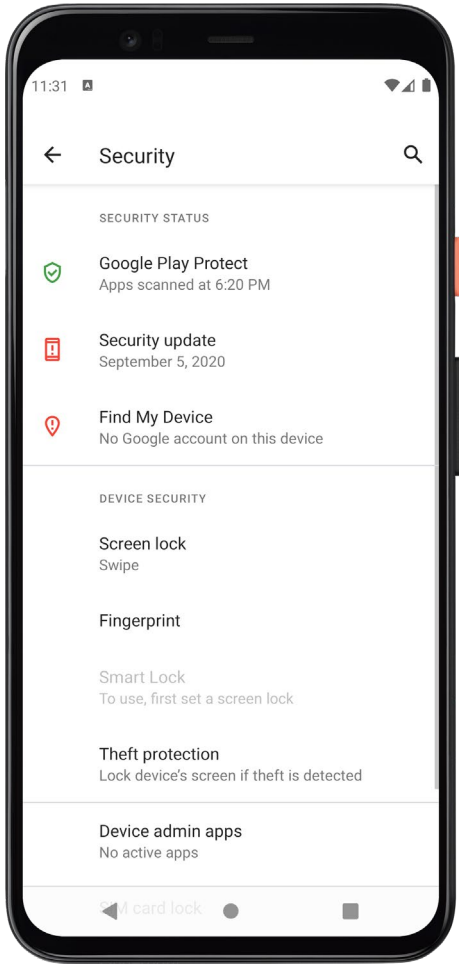


Table of contents

6. Takeaways

- A service is an app component that runs in the background to perform long-running general-purpose operations
- There are two types of services: started (background and foreground), and bounded (which offers a client-server interface to interaction between app components)
- In modern apps using Jetpack Compose, the `WorkManager` API is the preferred option for background workers
- A notification is a message that Android displays outside an app's user interface to provide the user with reminders or other information
- Alarms are scheduled tasks that allows to perform time-based operations outside the lifetime of an Android app