

Mobile Programming

Android 5 – Services, Broadcast Receiver



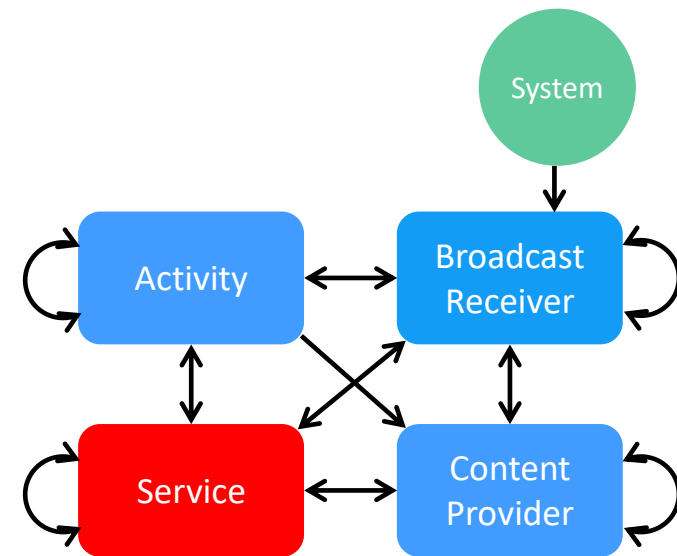
Nicola Keller

Inhalt

- Services
 - Komponente ohne UI
 - Für «wahrnehmbare» Hintergrundaktivitäten
- WorkManager
 - Für generelle Arbeiten im Hintergrund
- Broadcast Receiver
 - Empfänger von (System-)Nachrichten
 - Interner «Message Bus»

Service-Komponente

- Ursprünglich in Android zur Kapselung und Erledigung von Hintergrundarbeiten eingeführt
- Seit API 26 (Android 8, Oreo) stark eingeschränkt, wegen System-performance Issues (zu viele Services in zu vielen Apps)
- Nun nur noch als Spezialfall «Foreground»-Service empfohlen (und zum Export von App-Logik)
 - Klassischer Fall: Music Player, Navigation (z.B. Easy Ride: SBB Mobile, Download: Swisstopo)
- Für frühere Anwendungsbereiche wird nun Verwendung von *WorkManager* empfohlen
 - Z.B. Location Update, Background Sync, etc.



Android: Service-Konzept

<https://developer.android.com/guide/components/services.html>
<https://developer.android.com/reference/android/app/Service.html>

■ Was bietet ein Service?

- Eine Möglichkeit, dem System mitzuteilen, dass eine gewisse Arbeit im Hintergrund erledigt werden soll
- Eine Möglichkeit, gewisse Funktionalität (API) zu exportieren und anderen Applikationen anzubieten

Achtung: Eingeschränkte
Verwendung ab API 26!

`startService()` - **Auftrag** für
einen Service erteilen

■ Was ist ein Service **nicht**?

- Kein separater Worker-Thread (per se)
- Kein eigener Prozess (nur wenn so definiert)

`bindService()` – Öffnet
stehende Verbindung für
Kommunikation mit Service

■ Lang andauernde Operationen & Nebenläufigkeit?

- Ein Service kann (und sollte) einen eigenen Thread starten, um lang andauernde Operationen zu erledigen

Lebensarten eines Services + Lifecycle Methods

■ **Foreground**

- Es muss eine Notification angezeigt werden
- Läuft weiter auch wenn die App nicht im Foreground ist, muss explizit beendet werden
- Beispiel: Music player

■ **Background**

- Service nicht Sichtbar für den User
- Wenn App nicht im Foreground ist der Service limitiert

<https://developer.android.com/about/versions/oreo/background>

■ **Bound**

- Ein Service ist gebunden wenn die Applikation in mit `bindService()` aufruft.
- Ein gebundener Service definiert eine Client-Server Schnittstelle für Services
- Existiert nur solange wie mindestens an einen Service gebunden

Lebensarten eines Services + Lifecycle Methods

■ Zwei Lebensarten

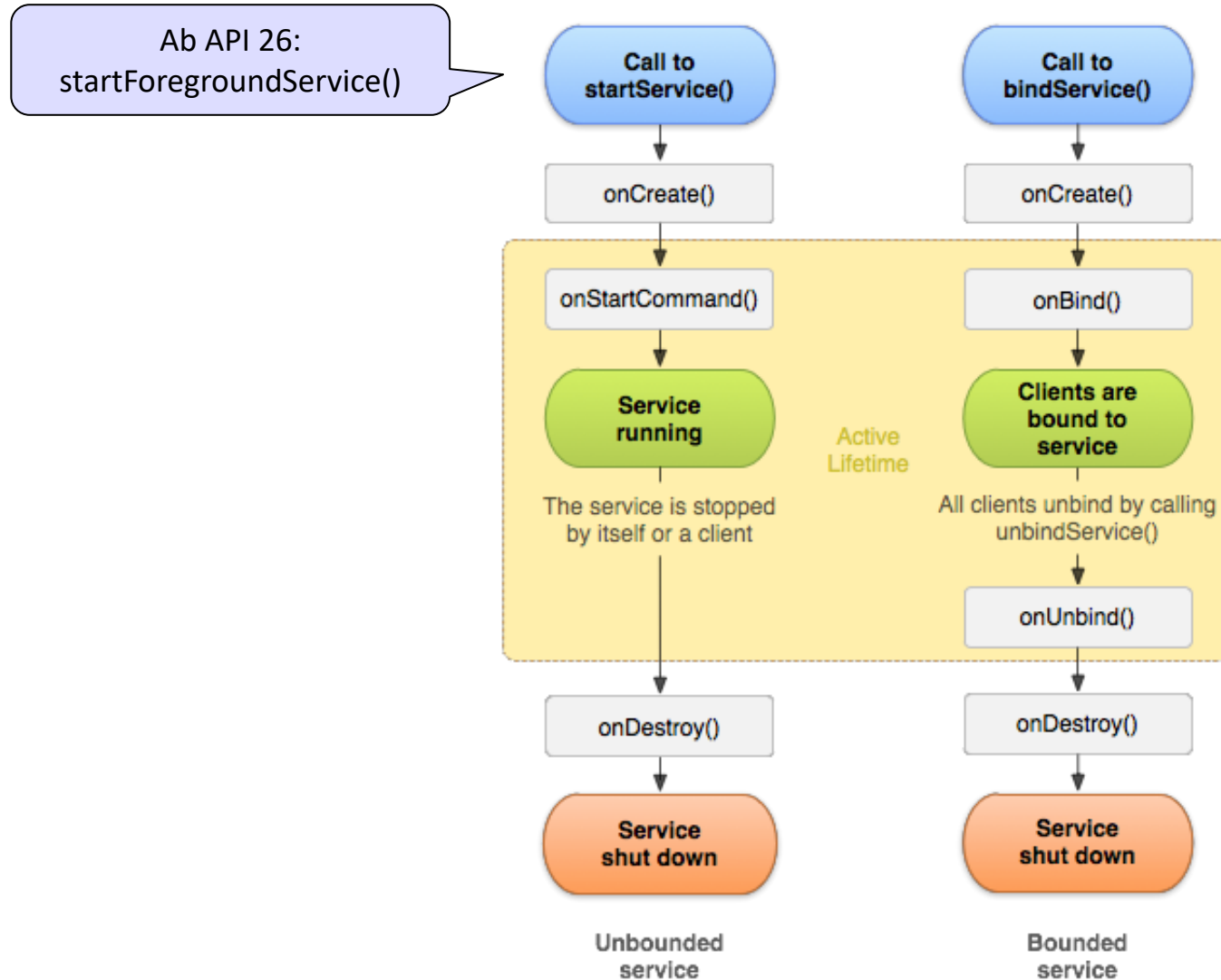
– Aufruf durch **startForegroundService(...)**

- `onCreate()` : Bei Erzeugung
- `onStartCommand()` : Auftragsbehandlung
- `onDestroy()` : Bei Beendung (durch Service selbst, durch Applikation oder durch System)

– Aufruf durch **bindService(...)**

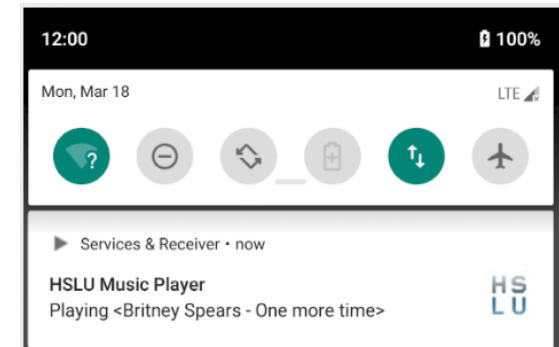
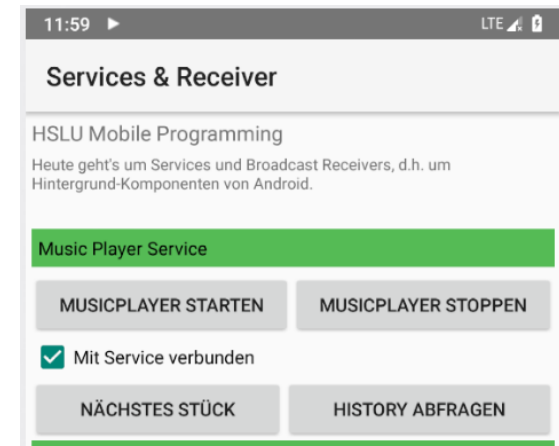
- `onCreate()` : s.o.
- `onBind()` : Wenn Komponente Verbindung herstellt
- `onUnbind()` : Wenn Komponente Verbindung beendet
- `onDestroy()` : s.o.

Lebenszyklus eines Service + Lifecycle Methods



Demo: Foreground Service

- Für Hintergrund-Arbeiten, die für Benutzer wahrnehmbar sind, z.B. *Music Player*
- Zeigt Notification an, während er läuft
- Benötigt Permission `FOREGROUND_SERVICE`
 - «Normal Permission», d.h. wird automatisch vergeben
- Interaktion mit / Steuerung Service oft über Binding
 - Folgt später



Foreground Service: Music Player Service (1)

```
class DemoMusicPlayerService : Service() {
```

```
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
        startPlayer()  
        return START_STICKY  
    }
```

Wird bei `startService()` aufgerufen.
Achtung: *main Thread*!

```
    private fun startPlayer() {  
        playThread?.let { it ->  
            if (it.isAlive()) { return } }  
        stopSignal.set(false)  
        startPlayThread()  
        startForeground(NOTIFICATION_ID, createNotification("-- waiting --"))  
    }
```

Macht aus Bg-
Service einen Fg-
Service

```
    override fun onDestroy() {  
        stopPlayer()  
        stopForeground(true) } }
```

Wird bei `stopService()`
aufgerufen

Es gibt keinen `onStopService()` Hook,
nur `onDestroy()`.

Foreground Service: Music Player Service (2)

```
<manifest>
...
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
...
<service
    android:name=".service.DemoMusicPlayerService"
    android:description="@string/musicplayerservice_desc"
    android:exported="false" />
...
</manifest>
```

```
return NotificationCompat.Builder(this, MainActivity.CHANNEL_ID)
```

```
    .setOngoing(true)
    .setContentTitle("HSLU Music Player")
    .setTicker("HSLU Music Player")
    .setContentText(musicTitle)
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .setOngoing(true)
    .setSmallIcon(android.R.drawable.ic_media_play)
    .setLargeIcon(BitmapFactory.decodeResource(resources, R.mipmap.ic_launcher))
    .setWhen(System.currentTimeMillis())
    .setCategory(Notification.CATEGORY_SERVICE)
    .build()
```

Channel muss 1x für App
angelegt werden, am besten
in Main Activity (siehe Docs)

Notification zuoberst,
kann von User nicht
weggeklickt werden

Foreground Service: Music Player Service (3)

■ Service starten

```
private fun startPlayerClicked() {  
    requireActivity().startService(Intent(context, DemoMusicPlayerService::class.java))  
}
```

Kann natürlich auch noch Parameter mitgeben im Intent!

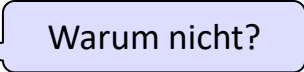
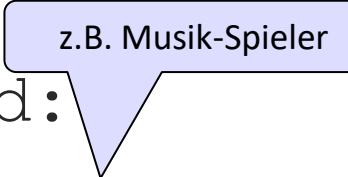
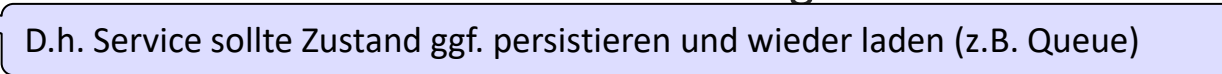
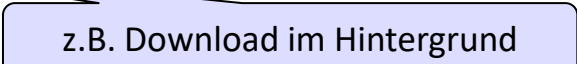
■ Service stoppen

```
private fun stopPlayerClicked() {  
    requireActivity().stopService(Intent(context, DemoMusicPlayerService::class.java))  
}
```

Allgemeines Muster

- Service wird bei App-Start oder aufgrund Event als Foreground-Service gestartet und bleibt alive
- Startet einen Worker-Thread oder Thread-Pool, der aktiv bleibt (zu Beginn ggf. idle)
- Mittels `bindService()` kann *synchron* kommuniziert werden

Service: STICKY oder NOT_STICKY?

- Was soll mit Service passieren, wenn das System den Applikationsprozess zerstört und später wieder herstellt?
 - `onStartCommand()` retourniert gewünschte Verhaltensweise
 - Nicht wichtig für gebundene Services! 
- Mögliche Rückgabewerte von `onStartCommand`: 
 - `START_STICKY`: Service soll nach Wiederherstellung automatisch gestartet werden - `onStartCommand()` wird erneut aufgerufen, aber ohne Intent
 - `START_NOT_STICKY`: Service wird nicht automatisch neu gestartet nach Wiederherstellung 
 - `START_REDELIVER_INTENT`: Wie `START_STICKY`, aber der ursprüngliche Intent wird noch einmal ausgeliefert, damit parametrisierte Reinitialisierung möglich 

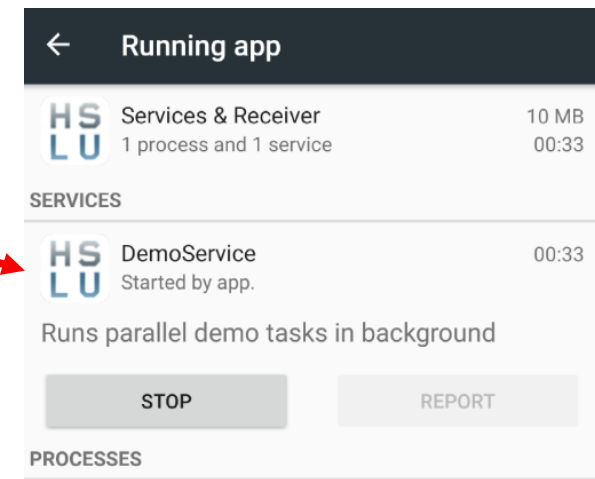
Tipp: Laufende Services auf Gerät anzeigen

User können laufende Services auf Gerät anzeigen und manuell stoppen:

- Developer Mode freischalten
- Settings > Developer options > Running services
- Deshalb: In Service-Deklaration im Manifest `android:description` verwenden, um Service zu beschreiben (nur String-ref erlaubt!)

Settings > About
Phone > Build # > Tap
7 times

```
<service
    android:name=".service.DemoService"
    android:exported="false"
    android:description="@string/demoservice_desc"/>
<service
    android:name=".service.DemoIntentService"
    android:exported="false"
    android:description="@string/intenservice_desc"/>
```

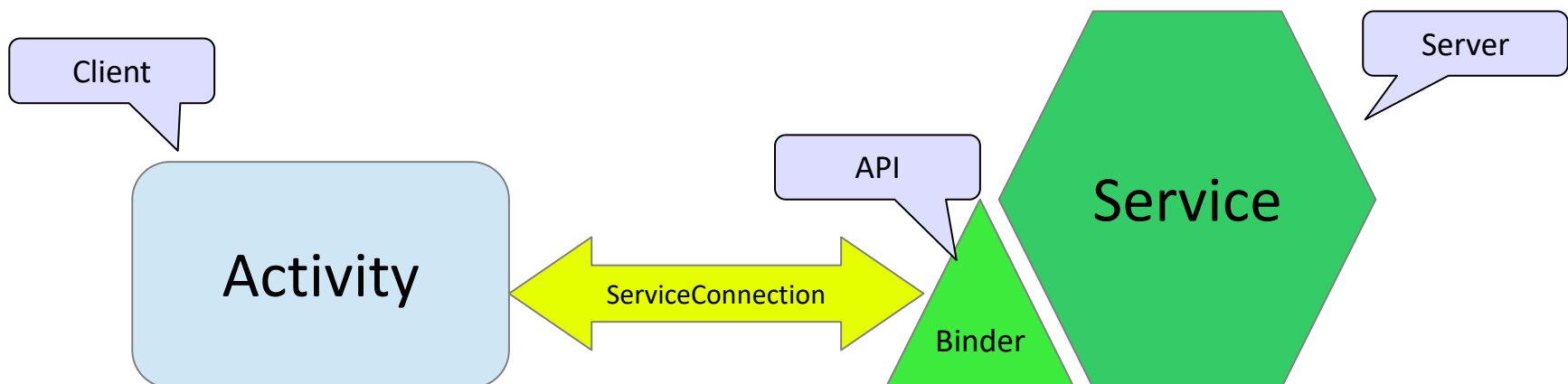


Gebundene Services

Ein Service kann gebunden werden

- Mittels `bindService(intent, connection, flag)`
- Client kommuniziert mit Service über `ServiceConnection`
- Damit kann Funktionalität einer App exportiert werden
 - Insbesondere mit einem „Remote Service“
- Bindung lösen mit `unbindService(connection)`

Hier **nicht** behandelt



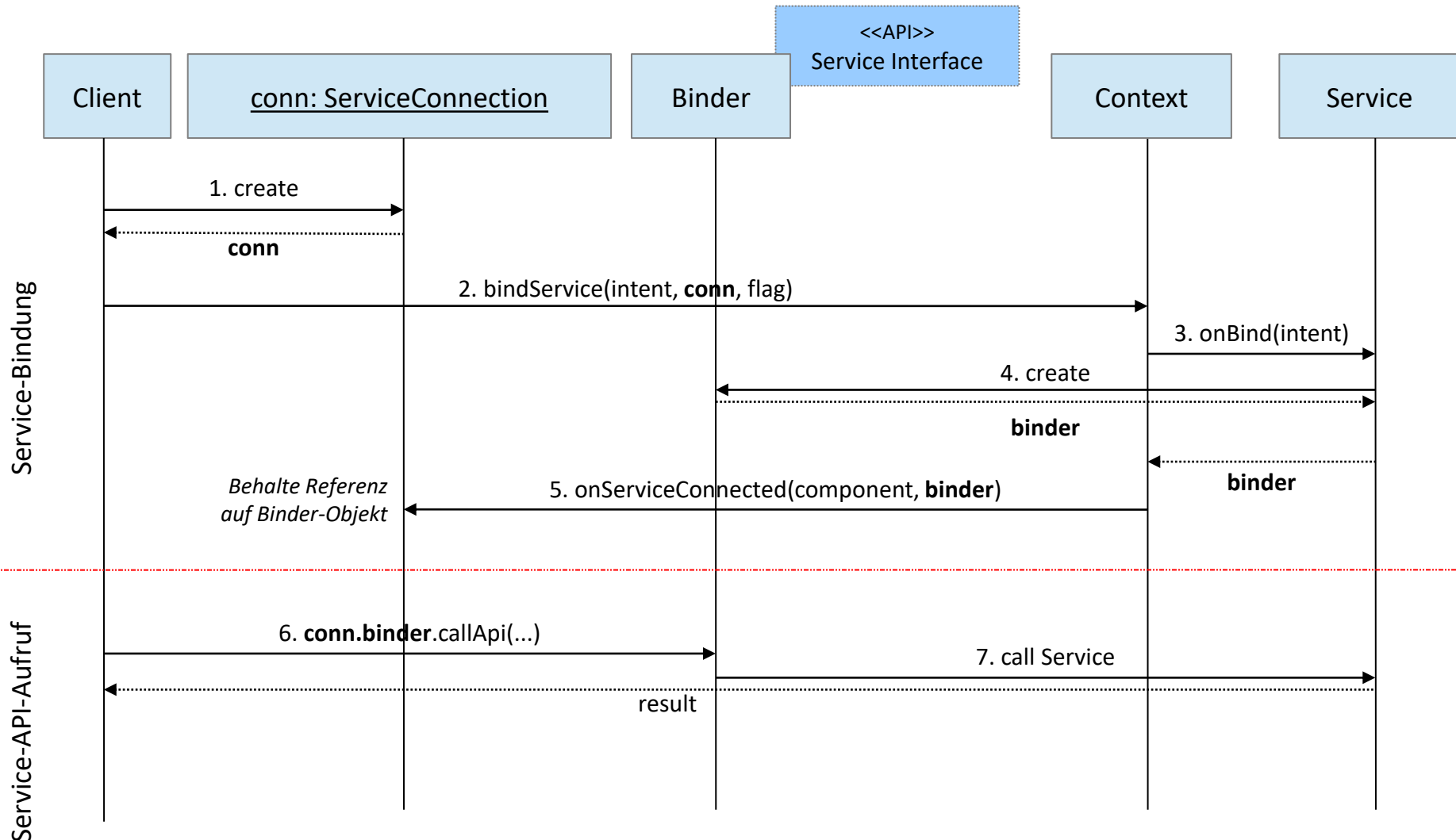
Gebundene Services: Involvierte Klassen

- Service Interface (MusicPlayerApi)
 - Definiert API des Services
- Binder (MusicPlayer.MusicPlayerApiBinder)
 - Implementiert Service Interface, wird dem Client bei erfolgreicher Verbindung übergeben (Service-Stub / -Handle)
- Service Connection (MusicPlayerConnection)
 - Definiert Callbacks für erfolgreiche Verbindung oder verlorene Verbindung, erhält Binder-Objekt (=API) bei Erfolg
- Service (MusicPlayerService)
 - Implementiert `onBind(intent)` und gibt Binder-Objekt zurück
- Client (MainActivity)
 - Ruft `bindService(intent, connection, flag)` resp. `unbindService(connection)` auf

Klassenname in Beispiel-Code, siehe spätere Folie (Übung 5)

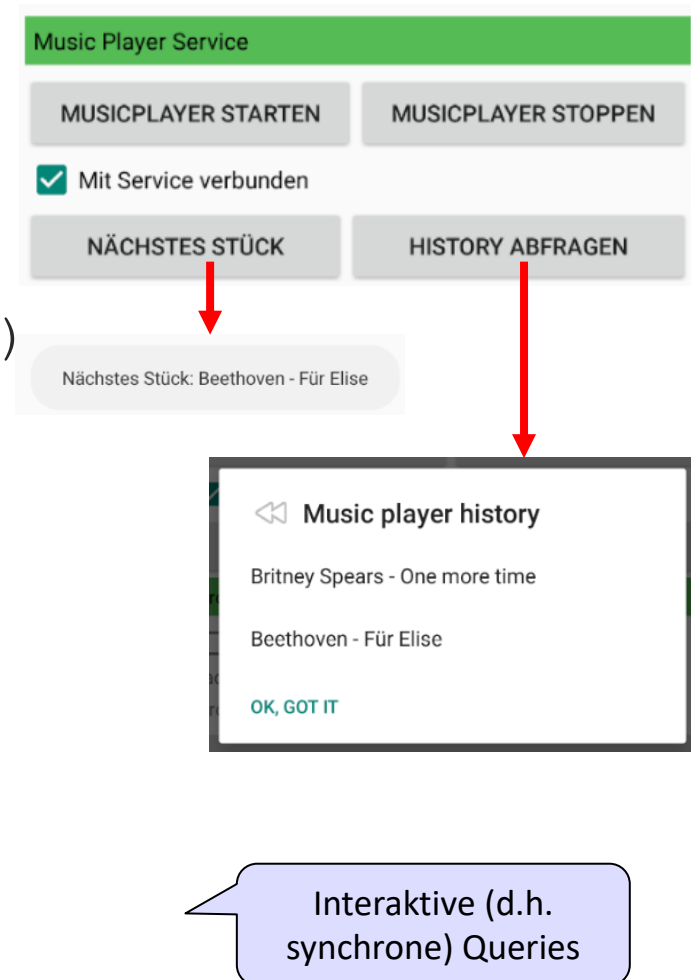
Callback-Handler

Ablauf: Service-Bindung & Service API-Aufruf



Demo: gebundener Service (Siehe Übung 5)

- `MusicPlayerService`
 - Kann gebunden werden
 - d.h. implementiert `onBind()`
 - Ablauf siehe Sequenz-Diag. vorangehende Folie!
- `MusicPlayerApi`
 - Bietet 2 Methoden an:
 - `playNextItem()`
 - `getHistory()`



Gebundener Service: Music Player Service (4)

```
interface MediaPlayerApi {  
    fun playNext(): String  
    fun queryHistory(): List<String>  
}
```

Service API Definition

```
override fun onBind(intent: Intent?): IBinder? {  
    return musicPlayerApi  
}
```

Gibt Binder-Instanz
zurück (=API Instanz)

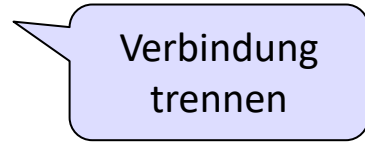
```
inner class MediaPlayerApiImpl : Binder(), MediaPlayerApi {  
    override fun playNext(): String {  
        return this@DemoMusicPlayerService.playNext()  
    }  
  
    override fun queryHistory(): List<String> {  
        return this@DemoMusicPlayerService.queryHistory()  
    }  
}
```

Service API
Implementierung

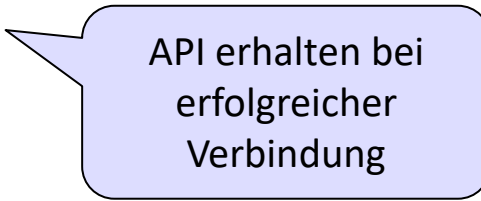
Gebundener Service: Client (5)

Verbinden

```
private fun bindService() {  
    val demoService = Intent(context, DemoMusicPlayerService::class.java)  
    demoServiceConnection = MusicPlayerConnection()  
    demoServiceConnection?.let { it ->  
        requireActivity().bindService(demoService, it, Context.BIND_AUTO_CREATE)  
    }  
}
```

Verbindung
trennen

```
private fun unbindService() {  
    requireActivity().unbindService(demoServiceConnection as ServiceConnection)  
    demoServiceConnection = null  
    main_stopServiceButton.isEnabled = true  
}
```

API erhalten bei
erfolgreicher
Verbindung

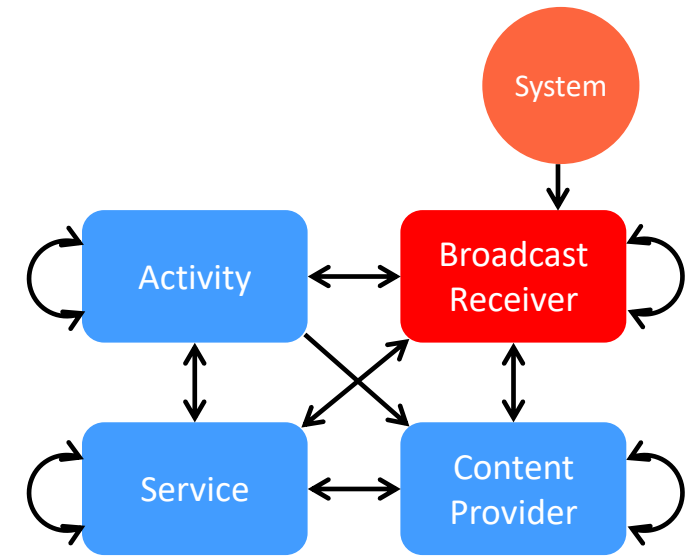
```
class MusicPlayerConnection : ServiceConnection {  
    private var musicPlayerApi: MusicPlayerApi? = null  
  
    override fun onServiceDisconnected(name: ComponentName?) {  
        musicPlayerApi = null  
    }  
  
    override fun onServiceConnected(name: ComponentName?, service: IBinder?) {  
        musicPlayerApi = service as MusicPlayerApi  
    }  
}
```



Broadcast Receiver

Broadcast Receiver

- Broadcasts sind Nachrichten
- App-interner «Message Bus»
- Alle Komponenten können Broadcasts verschicken und sich für Empfang registrieren
- System verschickt ebenfalls Nachrichten bei gewissen Events (App installiert, Timer, ...)
- **Seit API 26 (Android 8, Oreo) stark eingeschränkt**, wegen Performance (zuviele Handler, zuviele Msg)
- Nun werden nur noch sehr wenige Events global verteilt



z.B. «OnBootCompleted», gehen wir nicht näher drauf ein

gutes Video

- <https://developer.android.com/guide/components/broadcasts>

Broadcasts verschicken

D.h. können Daten mit sich tragen

- Broadcasts werden als Intents verschickt
- Implizit (in App) via `LocalBroadcastManager`
`.getInstance(this).sendBroadcast(intent);`
- Explizit (an andere App) über
 - `Context::sendBroadcast(intent)`
- Empfang von Broadcasts: Receiver (Empfänger)
 - Empfänger werden dynamisch im Code registriert
 - `registerReceiver(receiver, filter)`
 - Können auch statisch im Manifest definiert werden
 - Tag: `<receiver ...>`

Aufruf erfolgt auch wenn App nicht gestartet! Nur für explizite Broadcasts.

Globaler Broadcast Receiver (1)

Don't: Keine AsyncTasks!
Keinen Service binden!
Keinen Dialog anzeigen!

Do: Activity starten,
Service starten,
Notification schicken, ...

- BR ist immer nur so lange aktiv, wie Bearbeitung der empfangenen Nachricht braucht
 - Sonst inaktiv, wird vom System gelöscht!
 - (Erneute) Erzeugung „on demand“
 - Nur für Empfang von expliziten Broadcasts geeignet
- BR hat kein UI
 - Notifications benutzen für Kommunikation mit Benutzer...
 - ...oder Service starten für Hintergrund-Aufgaben

mit hoher
Priorität!

Globaler Broadcast Receiver (2)

■ Im Manifest

```
<receiver android:name=".BootCompletedReceiver">  
  <intent-filter>  
    <action android:name="android.intent.action.BOOT_COMPLETED" />  
  </intent-filter>  
</receiver>
```

XML Intent-Filter

■ Dedizierte Broadcast Receiver-Klasse

```
class BootCompletedReceiver : BroadcastReceiver() {  
  
    private val context: Context? = null  
  
    override fun onReceive(context: Context?, intent: Intent?) {  
        // do something, when boot has completed, e.g. start a service...  
    }  
}
```

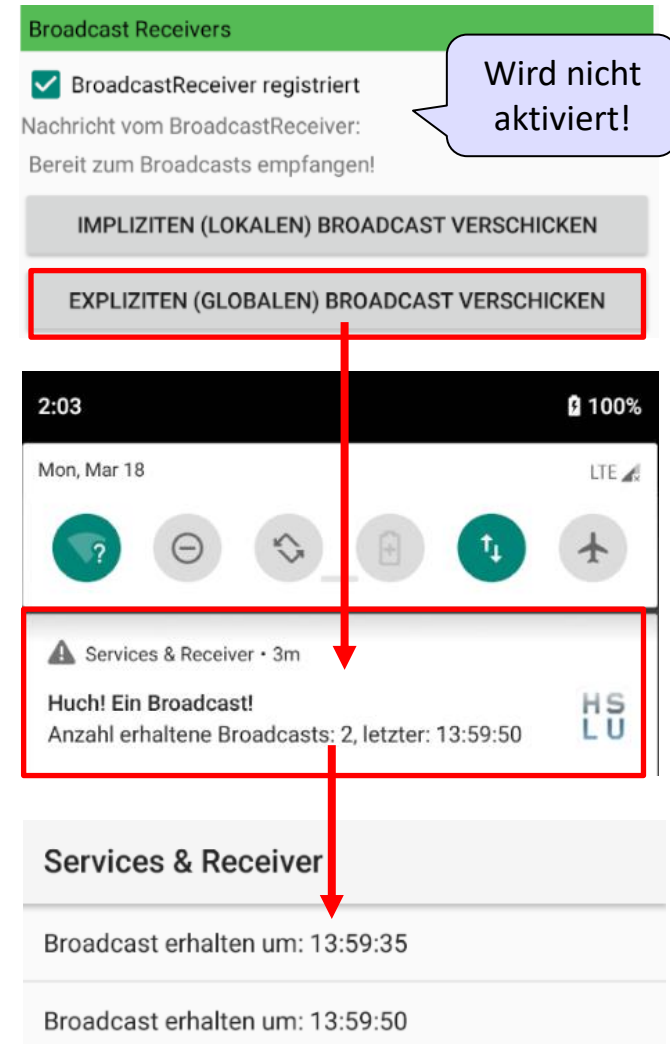
■ Expliziten Broadcast an andere App versenden

```
val broadcastIntent = Intent("ACTION_MY_BROADCAST")  
broadcastIntent.setPackage("ch.hslu.mobpro.other")  
sendBroadcast(broadcastIntent)
```

Wichtig:
Empfänger-ID!

Demo: Statischer BR (globale Message)

- App registriert Broadcast statisch Receiver im Manifest
- Versand globaler Broadcast adressiert an App via Package-ID
- Empfängt nur die expliziten Events und zeigt diese mit Notification an
 - So konfiguriert, dass Activity mit Details angezeigt wird bei Klick auf Notification



Lokale Broadcasts: «App Message-Bus»

■ BR erzeugen & registrieren im Code

```
myBroadcastReceiver = object : BroadcastReceiver() {  
    private var counter = 0  
    override fun onReceive(context: Context, intent: Intent) {  
        //do work  
    }  
}  
IntentFilter filter = new IntentFilter("ACTION_MY_BROADCAST");  
LocalBroadcastManager.getInstance(this)  
    .registerReceiver(myBroadcastReceiver, filter);
```

Hinweis: Muss
nicht eine innere
Klasse sein

Filter auf Action-Code

■ Nachricht versenden (Emitter)

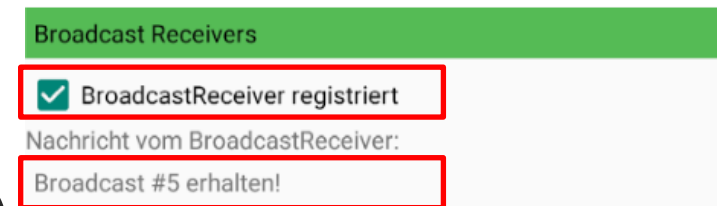
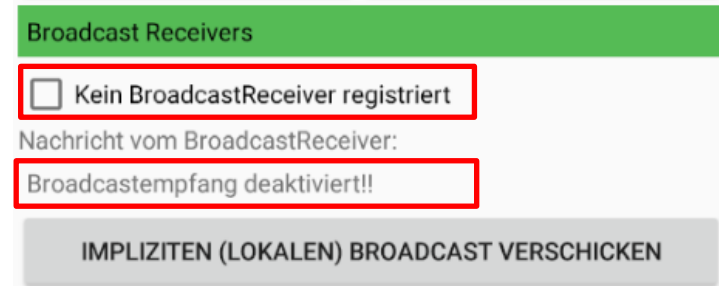
```
Intent localBroadcast = new Intent(" mobpro.DOWNLOAD_COMPLETE ");  
localBroadcast.putExtra("file", "Terminator2.mp4");  
LocalBroadcastManager.getInstance(this).sendBroadcast(localBroadcast);
```

■ BR deregistrieren (wenn nicht mehr benötigt)

```
LocalBroadcastManager.getInstance(requireContext()).unregisterReceiver(it)
```

Demo: dynamischer BR (on/off, Übung 5)

- MainActivity registriert eigenen Broadcast Receiver auf ACTION_MY_BROADCAST
 - On/Off-CheckBox
 - Registrierung im Code
 - `(un)registerReceiver(...)`
- Eigener Broadcast Receiver (z.B. anonyme Klasse) zählt Anzahl empfangene Broadcasts und zeigt diese auf View an
 - ☞ Empfang ausschliesslich, wenn BR registriert!





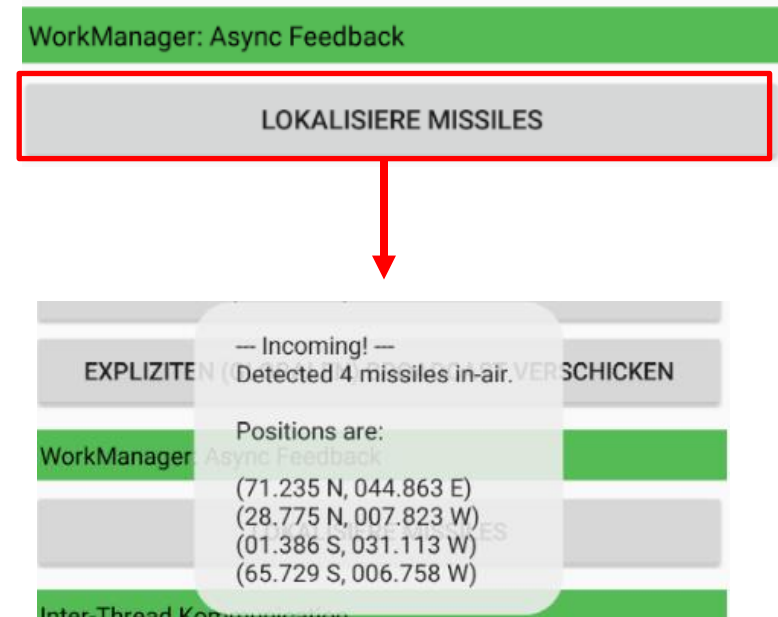
WorkManager + Livedata

WorkManager für Hintergrundtasks

- Erinnere letzte Vorlesung: repetitive oder einmalige Background-Tasks, die aufschiebbar sind, sollten via WorkManager erledigt werden
- Wie findet die App heraus, wenn der Task abgeschlossen ist?
- Z.B. mittels LiveData vom WorkManger
 - Verarbeitung in diesem Fall nur, wenn App noch läuft
 - z.B. um Liste heruntergeladener Files zu refreshen

Demo: WorkManager

- Lang andauernder Task:
Missile-Positionen erkennen
- Worker task wird an
WorkManager übergeben
- Sobald Positionen bestimmt
sind, werden diese mittels
Resultat gesetzt.
- Das LiveData observed das Resultat und generiert eine
Toast.



Worker mit Resultat-Event

WorkRequest erstellen
und erfassen

```
val getLocationTask = OneTimeWorkRequest.  
    Builder(LocalizeMissilesWorker::class.java).build()  
WorkManager.getInstance(requireContext()).enqueue(getLocationTask)
```

```
class LocalizeMissilesWorker(context: Context, params: WorkerParameters)  
: Worker(context, params) {  
    override fun doWork(): Result {  
        Log.i("LocalizeMissilesWorker", "Getting location of missiles...")  
        val output: Data = workDataOf(Pair("missilePositions", data));  
        return Result.success(output)    }  
}
```

Erfolg/Misserfolg melden

Output setzen,
Nur primitive Typen
und Listen davon
erlaubt

Worker Result Observen

Observe

Livedata with id

```
val workInfo: LiveData<WorkInfo> =
    WorkManager.getInstance(requireContext()).getWorkInfoByIdLiveData(getLocationTask.id)
workInfo.observe(viewLifecycleOwner, Observer { workInfo ->
    if (workInfo != null && workInfo.state == WorkInfo.State.SUCCEEDED) {
        val missilePositions: Array<String>? =
            workInfo.outputData.getStringArray("missilePositions")
        missilePositions?.let { missilePositionsList ->
            if (missilePositionsList.isNotEmpty()) {
                showToast(
                    String.format(
                        "--- Incoming! ---\nDetected %d missiles in-air.\n\nPositions
                        are:\n\n%s",
                        missilePositionsList.size,
                        missilePositionsList.joinToString("\n")
                    )
                )
            } else {
                showToast("No missiles detected.")
            }
        }
    }
})
```

WorkManager: Weitere Informationen

- Benötigte Gradle-Dependency:

```
implementation 'androidx.work:work-runtime-ktx:2.7.1'
```

- Das gezeigte Background-Work Beispiel mit einem Worker funktioniert selbstredend auch mit Threads
- Worker kann auch als wiederkehrender Task registriert werden oder mit einem anfänglichen Delay
- Arbeiten können auch in Graphen mit Abhängigkeiten definiert werden
- Für mehr Info siehe *How-To Guides* unter <https://developer.android.com/topic/libraries/architecture/workmanager>



Übung 5

Zur Übung 5

- Eigener gebundener Service
 - Simuliert Music Player
 - Starten & stoppen
 - Inkl. Service-API
- Eigener Broadcast Receiver
 - Dynamische Registration im Code
 - Eigene Broadcast-Action
- Optional: WorkManager verwenden

