14- service

Un servizio in Android è un componente applicativo che esegue operazioni in background senza fornire un'interfaccia utente. I servizi possono essere utilizzati da altri componenti dell'applicazione anche se appartengono a processi diversi. Esistono due tipi principali di servizi:

1. **Servizi avviati (Started Services)**
2. **Servizi associati (Bound Services)**

Servizi Avviati -Started Services

Un servizio è considerato "avviato" quando un componente dell'applicazione (come un'attività) lo avvia chiamando il metodo **startService(...).**

- Una volta avviato, un servizio può **funzionare indefinitamente in background**, anche se il componente che l'ha avviato viene distrutto, a condizione che mostri la sua presenza nella barra di stato (ForegroundService).
- In generale, un servizio avviato esegue una singola operazione e non restituisce alcun risultato al componente chiamante.
    - Ad esempio, può scaricare un file dalla rete e salvarlo nel file system del dispositivo.
- Quando il **compito è terminato, il servizio può arrestarsi autonomamente**.
    - Quando il task che causa il service, finisce se non ha ricevuto altre richieste,e il service si ferma e si distrugge

Servizi Associati – Bound Service

Un servizio è chiamato "associato" quando un componente dell'applicazione lo invoca utilizzando il metodo **bindService(...).**

- altro component si è connesso a questo component con il metodo bindService che richiede un intent
    - Un canale di comunicazione viene aperto tra l'application e il service, esponendo delle API
    - Può invocare metodi
    - Service bound può appartenere allo stesso processo dell'applicazione corrente o essere in un processo diverso, in quel caso Android fornisce metodi opportuni

Questo offre un'interfaccia client-server che consente al componente invocante di interagire con il servizio inviando richieste e ottenendo risultati, supportato da un meccanismo di comunicazione tra processi (IPC).
Il ciclo di vita di un servizio associato è legato al componente a cui è connesso.
Diversi componenti possono essere associati allo stesso servizio contemporaneamente, ma quando tutti i componenti associati si disconnettono, il servizio viene distrutto.

Utilizzo dei Servizi

Qualsiasi componente dell'applicazione può utilizzare un servizio, per usare i servizi, devo sapere il nome e avere il context per creare un Intent.
Un service runna sul main thread del processo ospitante/hosting

- Di default, non crea altri thread
- Se svolge operazioni costose, queste devono essere svolte in un altro thread

nota: service solitamente sono legati ad un behaviour della mia app

- Nel Manifest, vanno inseriti i services che la mia applicazione vuole fornire
    - Può essere dichiarato privato nel file di manifesto per impedirne l'accesso da altre applicazioni.

Classe android.app.Service – *using service*

La classe **Service** è la classe base da cui derivano tutte le implementazioni di servizi.
Richiede la sovrascrittura di alcuni metodi per gestire aspetti chiave del ciclo di vita e per fornire un meccanismo di associazione dei componenti.

- **onStartCommand(...)**: Il sistema chiama questo metodo ogni volta che un altro componente, come un'attività, richiede l'attivazione del servizio tramite il metodo *startService(...)*.
  - Una volta eseguito questo metodo, il servizio è avviato e può funzionare in background indefinitamente.
  - Per fermarlo, è necessario chiamare stopSelf(...) o stopService(...) passando l'intent appropriato.
- **onBind(...)**: Il sistema chiama questo metodo quando un altro componente vuole associarsi al servizio invocando bindService(...).
  - *Questo metodo dovrebbe restituire un oggetto che implementa l'interfaccia IBinder o null.*
    - *iBinder: con metodi che sono inviabili su diverse applicazioni se necessario*
    - *null: nessuna iterazione permessa*
- **onCreate()**: Il sistema chiama questo metodo quando l'oggetto servizio è appena stato creato, prima che venga chiamato onStartCommand(...) o onBind(...).
    - *non ha parametri*
    - *invoato all'inizio del lifecycle del service, subito dopo il constructor*
- **onDestroy()**: Il sistema chiama questo metodo quando il servizio non è più utilizzato e deve essere distrutto. I servizi devono implementare questo metodo per rilasciare le risorse.
    - invocato alla fine del lifecycle
    - siamo responsabili di rilasciare le risorse che abbiamo preso durante il lifecycle del servizio stesso
    - dopo questa callback, il reference del servizio viene scartato da android

Dichiarare un Servizio nel Manifest File

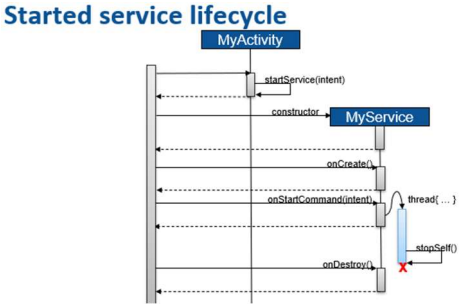
È necessario dichiarare tutti i servizi nel file di manifesto dell'applicazione. Questo si fa aggiungendo un elemento di tipo <service> come figlio dell'elemento <application>.

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
```

L'attributo android:name è obbligatorio, ma possono essere aggiunte altre proprietà (permessi del servizio, ecc.).

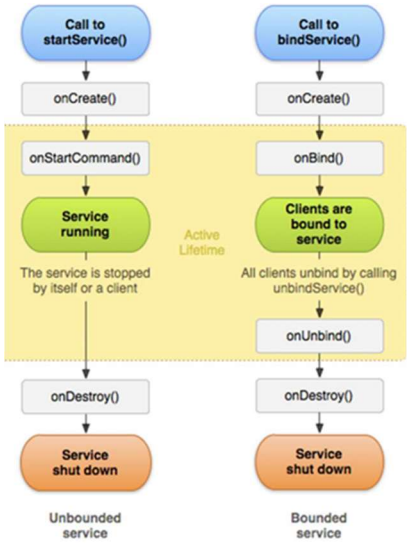
Ciclo di Vita di un Servizio Avviato - startService

- **startService(intent)**: Avvia il servizio.
- **onCreate()**: Metodo chiamato quando il servizio viene creato.
- **onStartCommand(intent)**: Metodo chiamato per gestire le richieste di avvio del servizio.
- **stopSelf()**: Metodo per fermare il servizio.
- **onDestroy()**: Metodo chiamato quando il servizio viene distrutto.



Ciclo di Vita di un Servizio Associato – bind service

- **bindService***(intent, connection, flag)*: Associa il servizio.
- **onCreate()**: Metodo chiamato quando il servizio viene creato.
- **onBind***(intent):* Metodo chiamato per gestire l'associazione del servizio.
- **onUnbind()**: Metodo chiamato quando il servizio viene disassociato.
- **unbindService***(connection):* Disassocia il servizio.
- **onDestroy()**: Metodo chiamato quando il servizio viene distrutto.



Estendere la Classe Service

Un servizio può essere implementato estendendo la classe Service. Nel metodo onCreate(...) è possibile creare e avviare un **HandlerThread** per gestire le richieste. Nel metodo **onStartCommand**(...), per ogni richiesta in arrivo viene inviato un messaggio al gestore fornendo dei dettagli.

Il metodo onStartCommand(...) deve restituire un intero che indica come il sistema gestirà il servizio in caso di arresto a causa di mancanza di spazio:

- **START_NOT_STICKY**: Il servizio non viene ricreato a meno che non ci siano richieste di Intent in sospeso.
  - Uccidilo e dimenticati di lui
- **START_STICKY**: Il servizio viene ricreato ma l'ultimo Intent non viene restituito.
  - Servizio importante ma la computazione che sta eseguendo ora, non è rilevante
- **START_REDELIVER_INTENT**: Il servizio viene ricreato e l'ultimo Intent viene restituito.
  - Processo deve evitare duplicazioni nei suoi side effect

```
class HelloService : Service {
  private lateinit var mServiceLooper:Looper;
  private lateinit var mServiceHandler:ServiceHandler;

  // Handler receives messages from thread
  class ServiceHandler(looper:Looper) : Handler(looper) {

     override fun handleMessage(msg: Message) {
     // Implementation
        when(msg.what) {
          1 -> { /* do somenthing */ }
          2 -> { /* do something else */ }
          else -> { /* … */}
        }
        // Stops the process using the reqeustID,
        // In this way it does not stop the service
        // in the middle of the operation of another job
        stopSelf(msg.arg1);
      }
  }
```

```
override fun onCreate() {

  // Create a thread to service request.
  // The thread has background priority
  // so operations that require a lot of CPU
  // do not affect the main thread
  val thread = HandlerThread("ServiceStartArguments",
          Process.THREAD_PRIORITY_BACKGROUND);
  thread.start();

  mServiceLooper = thread.getLooper();
  mServiceHandler = ServiceHandler(mServiceLooper);
}
```

```
override fun onStartCommand(Intent intent,
                            int flags, int startId): Int {
    Toast.makeText(this, "processing request starting",
                       Toast.LENGTH_SHORT).show();

    // For each request received,
    // send a message to the looper
    // together with a start ID,
    // to find when to terminate it
    val msg = mServiceHandler.obtainMessage(
                           intent.getIntExtra("cmd",0));
    msg.arg1 = startId;
    mServiceHandler.sendMessage(msg);

    // should the service be stopped, it will be resumed
    return START_STICKY;
}

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

Limitazioni sull'Esecuzione in Background

A partire da Android 8.0 (API level 26), sono state imposte limitazioni sull'esecuzione in background.
Un servizio in primo piano (foreground) esegue un'operazione visibile all'utente e mostra una notifica sulla barra delle applicazioni.
Quando un servizio in background è in esecuzione, l'utente potrebbe non notare la sua presenza

- questa modalità è consentita solo in situazioni limitate.

Applicazioni in Primo Piano e in Background

Un'app è considerata in primo piano se **ha un'attività visibile** (anche se è in pausa), ha un servizio in primo piano, **o un'altra app in primo piano è connessa ad essa**.

In tutti gli altri casi, l'app è considerata in background.

Un'app in background può rinnovare la finestra di esecuzione se gestisce un'attività visibile all'utente

- Un Firebase Cloud Messaging in arrivo o un messaggio SMS
- Esecuzione di un PendingIntent da una notifica
- Avvio di un servizio VPN

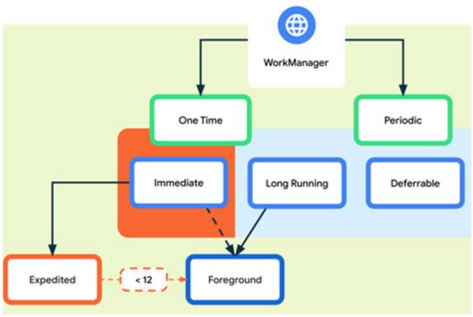
Gestione dei Task in Background

L'API WorkManager di Jetpack offre un modo semplice per programmare task asincroni differibili, garantendo l'esecuzione affidabile anche se l'app o il dispositivo si riavviano.

```
dependencies {
  implementation "androidx.work:work-runtime-ktx:2.7.1"
}
```

Bisogna definire un'istanza builder, i tipi di lavoro gestiti da WorkManager includono:

- **Immediate**: Task che devono iniziare immediatamente e completarsi rapidamente.
  - **OneTimeworkRequest**
- **Long Running**: Task che possono durare più di 10 minuti.
  - **WorkRequest**
- **Deferrable**: Task programmati che possono iniziare in un secondo momento e possono essere eseguiti periodicamente.
  - **PeriodickWoekRequest**



Nota: se voglio copiare db su server, conviene eseguirlo come background task

WorkManager, ritorna un result in modo tale da poter comprendere se tutto è andato bene o no

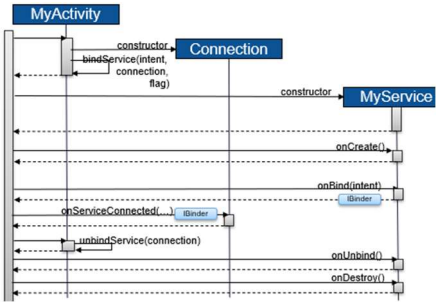
Esempio di implementazione di un worker:

```
class MyWorker(appCtx: Context, params: WorkerParameters): Worker(appCtx, params) {
  override fun doWork(): Result {
    // Esegui il lavoro qui
    return Result.success() // o .failure() o .retry()
  }
}

val constraints = Constraints.Builder()
    .setRequiresDeviceIdle(true)
    .build()

val myRequest = OneTimeWorkRequestBuilder<MyWorker>() //voglio che questo lavoro sia
                                                        eseguito una volta sola
    .setConstraints(constraints)
    .build()

WorkManager.getInstance(context).enqueue(myRequest)

Il sistema lo runnerà appeana possibile
```

Collegamento a un Servizio Associato -BoundService

Un componente può connettersi a un servizio associato invocando il metodo **ctx.bindService(...)**, fornendo un'implementazione **ServiceConnection** che riceve notifiche sul ciclo di vita dell'associazione.

- Un bound service opera come un server in un'interfaccia client-server
  - Consente alle attività (o ad altri componenti) di associarsi a se stesse per inviare richieste e ricevere risposte, possibilmente oltre i limiti del processo
- Implementa il metodo onBind() che ritorna un oggeto che implementa l'interfaccia IBinder
  - servizio fornisce _____ e lo usa per activity
  - Activity fornisce tutto ciò che serve
  - Quando activity non usa più, chiama onUnBound
  - Se Servizio avrà CounterOfActivity=0 → distrutto



Dobbiamo fornire l'istanza di connessione e altri servizi.

Esempio di attività:

```
class MainActivity : AppCompatActivity() {
    private var svc: BoundService.MyLocalBinder? = null

    val conn = object : ServiceConnection {
        override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
            svc = service as BoundService.MyLocalBinder
            // Notifica qui che il servizio è ASSOCIATO
        }

        override fun onServiceDisconnected(name: ComponentName?) {
            svc = null
            // Notifica qui che il servizio è disassociato
        }
    }

    private fun bind(): Boolean {
        val intent = Intent(this, BoundService::class.java)
        return bindService(intent, conn, Context.BIND_AUTO_CREATE)
    }
}
```

Implementazione dell'interfaccia IBinder

L'interfaccia IBinder consente ai client di accedere alle funzionalità fornite dal servizio. Esistono tre possibili implementazioni:

1. **Estensione della classe Binder**: semplice, funziona solo quando i client sono nello stesso processo del servizio.
    1. Espongo vari metodi
    2. Nel binder ritorno istance che incapsula l'indirizzo
2. **Utilizzo di un Messenger**: complessità media, richiede la creazione di un HandlerThread che processa i messaggi.
3. **Creazione di un componente remoto tramite AIDL**: alta complessità, consente ai client di comunicare con il servizio anche da processi diversi.

```
class BoundService : Service() {

        private val myBinder = MyLocalBinder(this);

        override fun onBind(intent: Intent?): IBinder {
                return myBinder;
        }

        fun getSomeServiceFunctionality(someParam:Int): String {
                //perform here the request job
                  return "SomeServiceResult";
        }

        fun getSomeOtherFunctionality(whatever: String): Int {
                //perform here the request job
                  return 1;
        }

        class MyLocalBinder(val service:BoundService) : Binder()
}
```

Esempio di implementazione tramite Binder:

```
class BoundService : Service() {
    private val binder = MyLocalBinder()

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

    override fun onBind(intent: Intent?): IBinder? = binder}
```

```
class MainActivity : AppCompatActivity() {

    private var svc: BoundService.MyLocalBinder? = null

    val conn = object:ServiceConnection {
        override fun onServiceConnected(name: ComponentName?,
                                        service: IBinder?) {
            svc = service as BoundService.MyLocalBinder
            //Notify here that service is BOUND
        }
        override fun onServiceDisconnected(name: ComponentName?) {
            svc = null
            //Notify here that service is now unbound
        }
    }
    private fun bind():Boolean {
        val intent = Intent(this, BoundService::class.java)
        return bindService(intent, conn, Context.BIND_AUTO_CREATE)
    }
}
```

se tutto Ok → onServiceConnection

→ risetti il servizio