10-Background Services

Android - Service

Un componente importante dell'applicazione nella piattaforma Android è un servizio.

Un servizio:

- Rappresenta un punto di ingresso aggiuntivo per un'applicazione
- È un componente dell'applicazione che può eseguire operazioni a lungo termine in background e non fornisce un'interfaccia utente

> Nota
> Un altro componente dell'applicazione che può avviare un servizio e continuerà ad eseguirsi in background anche se l'utente passa a un'altra applicazione. Inoltre, un componente può collegarsi a un servizio per interagire con esso e persino effettuare comunicazioni interprocesso (IPC), ad esempio un servizio potrebbe gestire transazioni di rete, riprodurre musica, eseguire operazioni di I/O su file o interagire con un provider di contenuti, tutto ciò in background.

Un servizio può assumere essenzialmente due forme principali:

- Started (Avviato) (Unbound)
- Collegato (Bound)

Started Service

Un servizio viene avviato quando un componente (come un'attività) lo avvia chiamando `startService()`, una volta avviato, un servizio può eseguire indefinitamente in background, anche se il componente che lo ha avviato viene distrutto.
Di solito, un servizio avviato esegue una singola operazione e non restituisce un risultato al chiamante, ad esempio potrebbe scaricare o caricare un file tramite la rete.

> Nota
> Una volta completata l'operazione, il servizio dovrebbe interrompersi da solo

Bound Service

Un servizio è collegato (bound) quando un componente dell'applicazione si collega ad esso chiamando `bindService()`, offre un'interfaccia client-server che consente ai componenti di interagire con il servizio, inviare richieste, ottenere risultati e persino farlo attraverso processi diversi mediante comunicazione interprocesso (IPC).

> Nota
> Un servizio collegato è in esecuzione solo finché un altro componente dell'applicazione è collegato ad esso

Diversi componenti possono collegarsi al servizio contemporaneamente, ma quando tutti si scollegano, il servizio viene distrutto.

Concurrency and Services

Un servizio Android viene eseguito nel thread principale.
Pper impostazione predefinita, non crea un proprio thread e non viene eseguito in un processo separato a meno che non venga lo specificato diversamente.

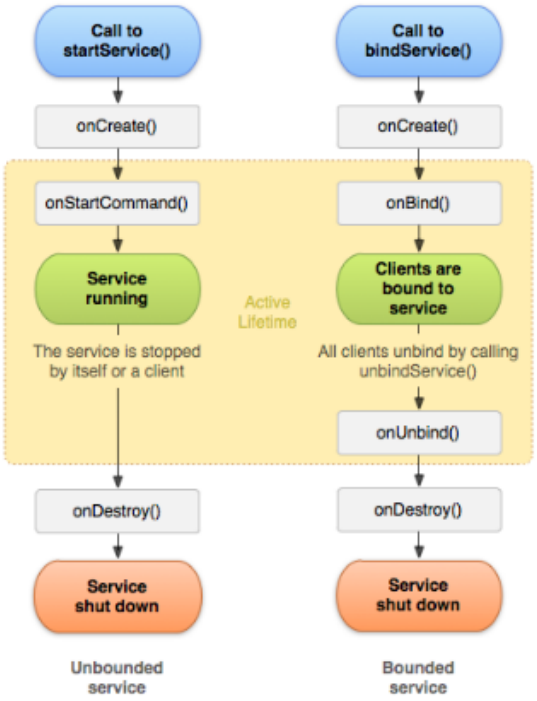
> Nota
> Se il servizio eseguirà operazioni intensive per la CPU o operazioni di blocco (ad esempio, riproduzione di file MP3 o operazioni di rete), bisognerebbe creare un nuovo thread all'interno del servizio per gestire l'elaborazione e ridurre il rischio di errori "Application Not Responding" (ANR)

Service - Life Cycle

Come un'attività, un servizio ha metodi di callback del ciclo di vita che si può implementare per monitorare le modifiche dello stato del servizio e svolgere operazioni nei momenti appropriati.

I metodi sono:

- `onCreate()`
- `onStartCommand()` o `onStart()`
- `onBind()`, `onUnbind()`, `onRebind()`
- `onDestroy()`




```
public class ExampleService extends Service {
        @Override
        public void onCreate() {
                // ...
        }
        @Override
        public int onStartCommand(Intent intent, int flags, int startId) {
                // ...
        }
        @Override
        public IBinder onBind(Intent intent) {
                // ...
        }
        @Override
        public boolean onUnbind(Intent intent) {
                // ...
        }
        @Override
        public void onRebind(Intent intent) {
                // ...
        }
        @Override
        public void onDestroy() {
                // ...
        }
}
```

Il ciclo di vita di un servizio avviene tra le chiamate `onCreate()` e `onDestroy()`. Come un'attività, un servizio esegue la sua configurazione iniziale in `onCreate()` e rilascia tutte le risorse rimanenti in `onDestroy()`, ad esempio un servizio di riproduzione musicale potrebbe creare il thread in cui verrà riprodotta la musica in `onCreate()` e poi interrompere il thread in `onDestroy()`

> Nota
> I metodi `onCreate()` e `onDestroy()` vengono chiamati per tutti i servizi, indipendentemente che siano creati da `startService()` o `bindService()`.

Il ciclo di vita attivo inizia con una chiamata a `onStartCommand()` o `onBind()` ricevendo rispettivamente l'intent passato a `startService()` o `bindService()`.

Se il servizio è avviato il ciclo di vita attivo termina allo stesso istante in cui termina l'intero ciclo di vita del servizio, viene considerato attivo finché `onsStartCommand()` ritorna.

> Nota
> Se un servizio è bound il suo ciclo di vita termina alla chiamata di `onUnbind()`

Creating a Service (Unbounded)

Creare un servizio in Android comporta l'estensione della classe `Service`, 'aggiunta di un tag di servizio nel file `AndroidManifest.xml` e la sovrascrittura dei metodi `onCreate()`, `onStart()`, `onStartCommand()` e `onDestroy()`.

Se si sta implementando un servizio collegato (bounded Service), si dovrebbe implementare alcuni metodi aggiuntivi che si analizzerà più avanti.

> Nota
> I metodi `onStart()` e `onStartCommand()` sono essenzialmente gli stessi, l'unica differenza è che `onStart()` è stato deprecato nelle versioni API 5 e successive

Se il servizio:

- Viene creato dal sistema con `Context.startService(...)`, il metodo `onCreate()` viene chiamato appena prima di `onStart()` o `onStartCommand()`.
- È collegato tramite una chiamata al metodo `Context.bindService()`, il metodo `onCreate()` viene chiamato appena prima di `onBind()`. In questo caso, `onStart()` e `onStartCommand()` non vengono chiamati.

Creating a Service

```
public class LocationTrackingService extends Service{
	@Override
	public IBinder onBind(Intent arg0) {
		return null;
	}
	@Override
	public void onCreate() {
		super.onCreate();
	}
	@Override
	public void onDestroy() {
		super.onDestroy();
	}
	@Override
	public void onStart(Intent intent, int startId) {
		super.onStart(intent, startId);
		startServiceTask();
	}
	@Override
	public int onStartCommand(Intent intent, int flags, int startId) {
		startServiceTask();
		return super.onStartCommand(intent, flags, startId);
	}
	@Override
	public boolean onUnbind(Intent intent) {
		return super.onUnbind(intent);
	}
	//...
}
```

Service and Android Manifest

Il sistema non è a conoscenza di un servizio a meno che non sia stato definito nel file `AndroidManifest.xml`utilizzando il tag `<service>`.

Nel blocco XML, è possibile specificare il nome del servizio e che il servizio sia abilitato, se si desidera si può aggiungere ulteriormente un `intent-filter` per avviare, controllare e inviare comandi specifici al servizio.

Gli intent filters vengono utilizzati per informare il sistema sugli intent impliciti che un servizio/attività può gestire (sono consentiti più IntentFilters). Ogni filtro descrive una capacità del componente, ovvero un insieme di intent che il componente è disposto a ricevere, in pratica i filtri consentono di filtrare gli intent di un tipo desiderato, escludendo gli intent indesiderati, ma solo quelli impliciti indesiderati (che non specificano una classe di destinazione).

> Nota
> Un intent esplicito viene sempre consegnato al suo destinatario, indipendentemente dal suo contenuto, il filtro non viene consultato. Tuttavia, un intent implicito viene consegnato a un componente solo se riesce a passare attraverso uno dei filtri del componente

```
<service android:enabled="true" android:name="LocationTrackingService">
	<intent-filter>
		<action android:name="it.unipr.dsg.tracker.LocationTrackingService.SERVICE" >
</action>
	</intent-filter>
</service>
```

Starting and stopping a Service

Avviare un servizio è una procedura semplice basata sulla creazione di un intent, si può utilizzare un Intent implicito o esplicito.

Con un Intent implicito, bisognerebbe definire un IntentFilter per il servizio e utilizzare una delle azioni definite per avviare il servizio (utile per esporre l'Intent per applicazioni esterne).

Con un Intent esplicito, bisognerebbe utilizzare direttamente il Context e il nome della classe del servizio, proprio come avviare un'attività.

Il codice per interrompere il servizio è essenzialmente lo stesso di quello per avviare il servizio, ma con una chiamata al metodo `stopService()`.

```
// intent implicito
Intent service = new Intent("it.unipr.dsg.tracker.LocationTrackingService.SERVICE");
startService(service); / stopService(service);

// intent esplicito
Intent service = new Intent(mContext,LocationTrackingService.class);
startService(service); / stopService(service);
```

Service and User Notification

I servizi non possono gestire l'interfaccia utente (UI), una volta in esecuzione un servizio può notificare l'utente di eventi utilizzando notifiche Toast o notifiche nella barra di stato (Status Bar Notifications).

> Nota
> Una notifica Toast è un messaggio che appare sulla superficie della finestra corrente per un breve periodo e poi scompare, mentre una notifica nella barra di stato fornisce un'icona nella barra di stato con un messaggio, che l'utente può selezionare per intraprendere un'azione (ad esempio, avviare un'attività).

Di solito, una notifica nella barra di stato è la tecnica migliore quando un'attività in background è stata completata (come il completamento del download di un file) e l'utente può agire di conseguenza.

Quando l'utente seleziona la notifica dalla vista estesa, la notifica può avviare un'attività (ad esempio, per visualizzare il file scaricato).

Creating a Bound Service

A volte è utile avere un maggiore controllo su un servizio rispetto alle sole chiamate di sistema per avviare e arrestare le sue attività.

Un servizio vincolato (bound service) è un servizio che consente ai componenti dell'applicazione di collegarsi ad esso chiamando `bindService()` per creare una connessione a lungo termine (e in generale non consente ai componenti di avviarlo chiamando `startService()`).

> Nota
> Bisognerebbe creare un servizio vincolato quando si desidera interagire con il servizio dalle attività e altri componenti nell'applicazione o per esporre alcune funzionalità della tua applicazione ad altre applicazioni tramite la comunicazione tra processi (IPC)

Per creare un servizio vincolato bisogna implementare il metodo di callback `onBind()` per restituire un `IBinder` che definisce l'interfaccia per la comunicazione con il servizio, gli altri componenti dell'applicazione possono quindi chiamare `bindService()` per ottenere l'interfaccia e iniziare a chiamare i metodi del servizio.

Il servizio esiste solo per servire il componente dell'applicazione che è ad esso vincolato, quindi quando non ci sono componenti collegati al servizio, il sistema lo distrugge (non è necessario arrestare un servizio vincolato come invece occorre fare quando il servizio viene avviato tramite `onStartCommand()`).

Ci sono tre modi per definire l'interfaccia `IBinder`:

1. Estensione della classe `Binder`, se il servizio è privato alla'pplicazione e viene eseguito nello stesso processo del client (cosa comune), si dovrebbe creare la propria interfaccia estendendo la classe `Binder` e restituendo un'istanza di essa da `onBind()`. Il client riceve il Binder e può utilizzarlo per accedere direttamente ai metodi pubblici disponibili sia nell'implementazione del Binder che nel Servizio stesso.
2. Utilizzo di un Messenger, se si ha bisogno che l'interfaccia funzioni attraverso processi diversi, si può creare un'interfaccia per il servizio con un Messenger. In questo modo, il servizio definisce un Handler, che risponde a diversi tipi di oggetti Message e che è alla base di un Messenger che può quindi condividere un `IBinder` con il client, consentendo al client di inviare comandi al servizio utilizzando oggetti Message. Inoltre, il client può definire un proprio Messenger in modo che il servizio possa inviare messaggi di risposta. Questo è il modo più semplice per effettuare la comunicazione tra processi (IPC), poiché il Messenger accoda tutte le richieste in un unico thread, quindi non è necessario progettare il servizio per essere thread-safe.
3. AIDL (Android Interface Definition Language), se si ha bisogno di supportare interfacce più complesse e interagire con altri processi o applicazioni esterne. È un linguaggio di definizione delle interfacce specifico di Android che consente di definire interfacce con metodi più complessi, supportando la comunicazione tra processi in modo robusto. Questo è il metodo più potente ma anche più complesso rispetto all'utilizzo di Binder o Messenger.

> Nota
> Il punto 1 definisce la tecnica preferita quando il servizio è semplicemente un lavoratore in background per l'applicazione, l'unico motivo per cui non si creerebbe l'interfaccia in questo modo è se il servizio viene utilizzato da altre applicazioni o in processi separati

Using AIDL

Utilizzando AIDL (Android Interface Definition Language) si esegue tutto il lavoro per scomporre gli oggetti in tipi primitivi comprensibili dal sistema operativo, che li confeziona per eseguire la comunicazione tra processi (IPC).

La tecnica precedente, utilizzando un Messenger, è effettivamente basata su AIDL come struttura sottostante, crea una coda di tutte le richieste del client in un singolo thread, quindi il servizio riceve le richieste una alla volta. Tuttavia, se si desidera che il servizio gestisca più richieste contemporaneamente, si può utilizzare AIDL direttamente.
In questo caso, il servizio deve essere in grado di eseguire multithreading e deve essere costruito in modo thread-safe.

> Nota
> Per utilizzare AIDL direttamente bisogna creare un file `.aidl` che definisce l'interfaccia di programmazione permettere agli strumenti di sviluppo Android SDK utilizzano questo file per generare una classe astratta che implementa l'interfaccia e gestisce la comunicazione IPC, che si può estendere all'interno del servizio.

```
// IRemoteService.aidl
package com.example.android;
// Declare any non-default types here with import statements
/** Example service interface */
interface IRemoteService {
        /** Request the process ID of this service, to do evil things with it. */
        int getPid();
        /** Demonstrates some basic types that you can use as parameters
        * and return values in AIDL.
        */
        void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,
                                                double aDouble, String aString);
}
```

Extending the Binder class

Se il servizio viene utilizzato solo dall'applicazione locale e non ha bisogno di funzionare tra processi, si può implementare la propria classe `Binder` che offre al client un accesso diretto ai metodi pubblici del servizio.

> Nota
> Questo funziona solo se il client e il servizio sono nella stessa applicazione e processo, altrimenti è necessaria la comunicazione tra processi - IPC. Ad esempio, questo funzionerebbe bene per un'applicazione musicale che ha bisogno di collegare un'attività al proprio servizio che sta riproducendo musica in background

Per configurare un Binder bisogna:

1. Creare un'istanza di Binder che contenga:
   - Metodi pubblici che il client può chiamare
   - Restituire:
     - L'istanza corrente di `Service`, che ha metodi pubblici che il client può chiamare
     - Un'istanza di un'altra classe ospitata dal servizio con metodi pubblici che il client può chiamare
2. Restituire questa istanza di `Binder` dal metodo di callback `onBind()`
3. Nel client ricevere il Binder dal metodo di callback `onServiceConnected()` e effettua chiamate al servizio vincolato utilizzando i metodi forniti

```
public class LocalService extends Service {
        // Binder given to clients
        private final IBinder mBinder = new LocalBinder();
        // Random number generator
        private final Random mGenerator = new Random();
        /**
        * Class used for the client Binder. Because we know this service always
        * runs in the same process as its clients, we don't need to deal with IPC.
        */
        public class LocalBinder extends Binder {
                LocalService getService() {
                // Return this instance of LocalService so clients can call public methods
                return LocalService.this;
                }
        }
```

```
        @Override
        public IBinder onBind(Intent intent) {
                return mBinder;
        }
        /** method for clients */
        public int getRandomNumber() {
                return mGenerator.nextInt(100);
        }
}
```

Binding to a Service

I componenti dell'applicazione (client) possono collegarsi a un servizio chiamando `bindService()`, il sistema quindi chiama il metodo `onBind()` del servizio, il quale restituisce un `IBinder` per interagire con il servizio.

Il collegamento è asincrono, `bindService()` ritorna immediatamente e non restituisce l'`IBinder` al client, che per riceverlo deve creare un'istanza di `ServiceConnection` e passarla a `bindService()`.

> Nota
> `ServiceConnection` include un metodo di callback che il sistema chiama per consegnare l'IBinder

> Nota
> Solo le attività (activities), i servizi (services) e i provider di contenuti (content providers) possono collegarsi a un servizio.
> Non è possibile collegarsi a un servizio da un ricevitore di trasmissione (broadcast receiver)

I servizi possono essere concatenati, un servizio avviato può collegarsi ad altri servizi che ha avviato per ulteriore disaccoppiamento delle responsabilità.

Per collegarsi a un servizio dal client bisogna:

- Implementare `ServiceConnection`, sovrascrivendo i due metodi di callback:
  - `onServiceConnected()`, il sistema chiama questo metodo per consegnare l'`IBinder` restituito dal metodo `onBind()` del servizio
  - `onServiceDisconnected()`, il sistema Android chiama questo metodo quando la connessione al servizio viene persa inaspettatamente, ad esempio quando il servizio è crashato o è stato terminato. Questo metodo non viene chiamato quando il client si scollega intenzionalmente dal servizio
- Chiamare `bindService()`, passando l'implementazione di `ServiceConnection`
- Quando il sistema chiama il metodo di callback `onServiceConnected()`, si può iniziare a effettuare chiamate al servizio, utilizzando i metodi definiti dall'interfaccia
- Per disconnettersi dal servizio bisogna `unbindService()`. Quando il client viene distrutto, si disconnetterà automaticamente dal servizio, ma bisognerebbe sempre scollegarsi quando si ha finito di interagire con il servizio o quando l'attività viene messa in pausa, in modo che il servizio possa essere terminato quando non viene utilizzato

```
LocalService mService;
private ServiceConnection mConnection = new ServiceConnection() {
        // Called when the connection with the service is established
        public void onServiceConnected(ComponentName className, IBinder service) {
                // Because we have bound to an explicit
                // service that is running in our own process, we can
                // cast its IBinder to a concrete class and directly access it.
                LocalBinder binder = (LocalBinder) service;
                mService = binder.getService();
        }
        // Called when the connection with the service disconnects unexpectedly
        public void onServiceDisconnected(ComponentName className) {
                Log.e(TAG, “onServiceDisconnected");
                mService = null;
        }
};
```

Con questa `ServiceConnection`, il client può collegarsi a un servizio passandola a `bindService()`, ad esempio:

- Il primo parametro di `bindService()` è un intent che specifica esplicitamente il nome del servizio a cui collegarsi (anche se l'Intent potrebbe essere implicito)
- Il secondo parametro è l'oggetto `ServiceConnection`
- Il terzo parametro è un flag che indica le opzioni per il collegamento, di solito dovrebbe essere `BIND_AUTO_CREATE` per creare il servizio se non è già attivo

```
Intent intent = new Intent(this, LocalService.class);
bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
```

Bound Service Notes

Di solito bisognerebbe abbinare la creazione e la distruzione del collegamento durante i momenti di avvio e chiusura del ciclo di vita del cliente ad esempio:

- Se si ha bisogno di interagire con il servizio solo mentre l'attività è visibile, bisognerebbe abbinare il collegamento durante `onStart()` e distaccarlo durante `onStop()`
- Se si desidera che l'attività riceva risposte anche quando è in background, si può abbinare il collegamento durante `onCreate()` e distaccarlo durante `onDestroy()`. Bisogna essere consapevoli che ciò implica che l'attività deve utilizzare il servizio per tutto il tempo in cui è in esecuzione (anche in background), quindi se il servizio è in un altro processo, aumenti il carico di lavoro di quel processo e aumenta la probabilità che il sistema lo termini

> Nota
> Di solito non si dovrebbe abbinare e distaccare durante `onResume()` e `onPause()` dell'attività, poiché questi callback si verificano ad ogni transizione del ciclo di vita e dovresti mantenere il processo di elaborazione che avviene in queste transizioni al minimo. Inoltre, se più attività nell'applicazione si collegano allo stesso servizio e c'è una transizione tra due di queste attività, il servizio potrebbe essere distrutto e ricreato mentre l'attività corrente si distacca (durante la pausa) prima che la successiva si colleghi (durante la ripresa)

Running a Service in the Foreground

I servizi dovrebbero rendere l'utente consapevole del fatto che stanno in esecuzione. Un servizio in primo piano:

- È un servizio associato a qualcosa di cui l'utente è consapevole e quindi non è candidato per essere terminato dal sistema quando la memoria è scarsa
- Deve fornire una notifica per la barra di stato, che viene posizionata sotto l'intestazione "In corso", il che significa che la notifica non può essere eliminata a meno che il servizio non venga interrotto o rimosso dal primo piano

Un servizio musicale che riproduce file multimediali dovrebbe essere impostato per essere eseguito in primo piano, poiché l'utente è esplicitamente consapevole della sua operazione. La notifica nella barra di stato potrebbe indicare la canzone corrente e consentire all'utente di avviare un'attività per interagire con il lettore musicale.

Il metodo per avviare un servizio in primo piano è `startForeground()` e richiede due parametri, un intero che identifica univocamente la notifica e la `Notification` per la barra di stato.

All'interno del service bisognerebbe creare la Notifica e chiamare il metodo `startForeground()` come illustrato nel seguente codice:

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.ticker_text),
System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);

// this message will be triggered when the user clicks on the notification
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

notification.setLatestEventInfo(this,
getText(R.string.notification_title),getText(R.string.notification_message),
pendingIntent);

startForeground(ONGOING_NOTIFICATION, notification);
```

Per rimuovere il servizio dal primo piano bisogna chiamre `stopForeground()`, il parametro booleano indica se rimuovere anche la notifica dalla barra di stato.

> Nota
> Questo metodo non ferma il servizio, tuttavia se si interrompe il servizio mentre è ancora in esecuzione in primo piano, allora la notifica viene rimossa anche essa

> Nota
> Le applicazioni che mirano a API Android superiori alla versione 9 (livello API 28) e utilizzano servizi in primo piano devono richiedere il permesso (normale) `FOREGROUND_SERVICE`, altrimenti il sistema lancerà una `SecurityException` durante l'esecuzione

JobIntentService

La classe `JobIntentService` fornisce una struttura semplice per eseguire un'operazione su un singolo thread in background, consentendo di gestire operazioni di lunga durata senza influire sulla reattività dell'interfaccia utente.

Un IntentService però ha alcune limitazioni:

- Non può interagire direttamente con l'interfaccia utente, per visualizzare i risultati nell'UI bisogna inviarli a un'activity

- Le richieste di lavoro vengono eseguite in modo sequenziale, se un'operazione è in esecuzione in un JobIntentService e ne si invia un'altra, la seconda richiesta attende fino a quando la prima operazione non è terminata
- Un'operazione in esecuzione su un IntentService non può essere interrotta.

> Nota
> `JobIntentService` può essere adottato per operazioni semplici in background.

Definizione di sevizio estendendo la classe `JobIntentService`

```
public class MyJobIntentService extends JobIntentService {
        @Override
        protected void onHandleWork(@NonNull Intent intent) {
                try {
                        startServiceWork(intent);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Mandare un lavoro al servizio

```
Intent serviceIntent = new Intent();
serviceIntent.putExtra(MyJobIntentService.RANDOM_LIMIT_EXTRA, 100);

//jobId A unique job ID for scheduling; must be the same value for all work enqueued for the
same class.
int jobId = 1000;
MyJobIntentService.enqueueWork(mContext, MyJobIntentService.class, jobId, serviceIntent);
```

JobIntentService - Manifest

Per utilizzare un JobIntentService nell'applicazione bisgna dichiararlo all'interno del file `AndroidManifest.xml`

```
<service
        android:name=".service.MyJobIntentServiceName"
        android:permission="android.permission.BIND_JOB_SERVICE"
        android:exported="false"/>
```

Il flag `exported="false"` indica che il servizio è disponibile solo per l'applicazione corrente

---

Intent

Tre dei componenti fondamentali di un'applicazione sono:

- Attività (Activities)
- Servizi (Services)
- Ricevitori di trasmissioni (Broadcast Receivers)

E vengono attivati tramite messaggi chiamati "intents" (intenzioni).
La messaggistica degli intents è una funzionalità per il collegamento a tempo di esecuzione tra i componenti della stessa o diverse applicazioni.

Un oggetto `Intent` è una struttura dati passiva che contiene una descrizione astratta di un'operazione da eseguire, o spesso nel caso delle trasmissioni, una descrizione di qualcosa che è accaduto e viene annunciato. Sono caratterizzati da un identificatore e possono includere informazioni aggiuntive (`Bundle`) che possono essere aggiunte come `extras` (extra).

Esistono meccanismi separati per inviare intenzioni a ciascun tipo di componente:

- Un oggetto `Intent` viene passato a `Context.startActivity()` o `Activity.startActivityForResult()` per avviare un'attività o far svolgere qualcosa di nuovo a un'attività esistente (può anche essere passato a `Activity.setResult()` per restituire informazioni all'attività che ha chiamato `startActivityForResult()`)
- Un oggetto `Intent` viene passato a:
  - `Context.startService()`, per avviare un servizio o fornire nuove istruzioni a un servizio in corso
  - `Context.bindService()`, per stabilire una connessione tra il componente chiamante e un servizio di destinazione. Se necessario, si può anche avviare il servizio se non è già in esecuzione
- Gli oggetti `Intent` passati a uno qualsiasi dei metodi di trasmissione (come `Context.sendBroadcast()`, `Context.sendOrderedBroadcast()` o `Context.sendStickyBroadcast()`) vengono inviati a tutti i ricevitori di trasmissione interessati, molte tipologie di queste hanno origine nel codice di sistema
- Gli intent sono particolarmente utili con i servizi Android per scambiare informazioni e notifiche tra l'oggetto chiamante e il servizio

Intent FIlter

Per informare il sistema riguardo alle intenzioni implicite che possono gestire, le attività (activities), i servizi (services) e i ricevitori di trasmissioni (broadcast receivers) possono avere uno o più filtri di intenti (intent filters).

Ciascun filtro descrive una capacità del componente, ovvero un insieme di intenzioni che il componente è disposto a ricevere. Il filtro, di fatto, filtra le intenzioni di un determinato tipo, scartando invece le intenzioni indesiderate, ma solo quelle implicite indesiderate (ovvero quelle che non specificano una classe di destinazione).

> Nota
> Un'intenzione esplicita (explicit intent) viene sempre recapitata al suo destinatario, indipendentemente da ciò che contiene, il filtro non viene consultato

> Nota
> Un'intenzione implicita (implicit intent) viene recapitata a un componente solo se può attraversare uno dei filtri del componente stesso

Un filtro di intenti è un'istanza della classe `IntentFilter`, tuttavia, poiché il sistema Android deve conoscere le capacità di un componente prima di poterlo avviare, i filtri di intenti generalmente non vengono impostati nel codice Java, ma nel file manifesto dell'applicazione (`AndroidManifest.xml`) come elementi `<intent-filter>`.

> Nota
> L'eccezione sarebbe rappresentata dai filtri per i ricevitori di trasmissioni che vengono registrati dinamicamente tramite `Context.registerReceiver()`, in questo caso, vengono creati direttamente come oggetti `IntentFilter`

```
<activity android:name=".MainActivity"
                      android:label="@string/app_name" >

        <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
        <intent-filter>
                <action android:name="it.unipr.dsg.tracker.NEW_LOCATION" ></action>
        </intent-filter>
</activity>

<service android:enabled="true" android:name="LocationTrackingService">
        <intent-filter>
                <action android:name="it.unipr.dsg.tracker.LocationTrackingService.SERVICE" >
</action>
        </intent-filter>
</service>
<receiver android:name="it.unipr.dsg.tracker.LocationTrackingReceiver">
        <intent-filter>
                <action android:name="it.unipr.dsg.tracker.NEW_LOCATION" ></action>
        </intent-filter>
</receiver>
```

Broadcast Receiver

Un broadcast receiver è una classe che estende `BroadcastReceiver` e che viene registrata come ricevitore in un'applicazione Android tramite il file `AndroidManifest.xml` (o tramite il codice). In alternativa a questa registrazione statica, è anche possibile registrare un `BroadcastReceiver` dinamicamente tramite il metodo `Context.registerReceiver()`.

Questa classe sarà in grado di ricevere intent (generati internamente o esternamente - ad esempio, generati dal sistema).

> Nota
> Gli intenti possono essere generati tramite il metodo `Context.sendBroadcast()`

La classe `BroadcastReceiver` definisce il metodo `onReceive()` solo nel quale l'oggetto `yourBroadcastReceiver` sarà valido. In seguito, il sistema Android potrà riciclare il `BroadcastReceiver`.

> Nota
> Pertanto, non è possibile eseguire alcuna operazione asincrona nel metodo `onReceive()`

Il metodo `sendBroadcast()` consente di inviare intent di broadcast, questi "messaggi di broadcast" vengono intercettati da Activity, Service e BroadcastReceiver autonomi che dichiarano l'appropriato filtro nell`'AndroidManifest.xml`.

Example 1

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
                        package="package.name.test"
                        android:versionCode="1"
                        android:versionName="1.0" >

        <uses-sdk android:minSdkVersion="10" />

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

        <application
                android:icon="@drawable/icon"
                android:label="@string/app_name" >

                <receiver android:name="MyPhoneReceiver" >
                        <intent-filter>
                                <action android:name="android.intent.action.PHONE_STATE" >
</action>
                        </intent-filter>
                </receiver>
        </application>
</manifest>
```

```
package package.name.test;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.util.Log;

public class MyPhoneReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                Bundle extras = intent.getExtras();
                if (extras != null) {
                        String state = extras.getString(TelephonyManager.EXTRA_STATE);
                        Log.d("TEST", state);
                        if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
                                String phoneNumber =
extras.getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
                                Log.d("TEST", phoneNumber);
                        }
                }
        }
}
```

Example 2

```
<activity
        android:name=".MainActivity"
        android:label="@string/app_name" >
        <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
        <intent-filter>
                <action android:name="it.unipr.dsg.tracker.NEW_LOCATION" ></action>
        </intent-filter>
</activity>
```

```
public class ActivityLocationTrackingReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                Bundle extras = intent.getExtras();
                Log.d(MainActivity.TAG,"ActivityLocationTrackingReceiver ---> Received
Intent: " + intent.getAction());
        }
}
```

Alarm Manager

Gli allarmi (basati sulla classe `AlarmManager`) offrono un modo per eseguire operazioni basate sul tempo al di fuori del ciclo di vita dell'applicazione, ad esempio è possibile utilizzare un allarme per avviare un'operazione a lungo termine, come avviare un servizio una volta al giorno per scaricare una previsione del tempo (simile ai `cronjob`) e presentano le seguenti caratteristiche:

- Consentono di inviare intent a tempi e/o intervalli predefiniti
- È possibile utilizzarli insieme ai broadcast receiver per avviare servizi ed eseguire altre operazioni
- Operano al di fuori dell'applicazione, quindi è possibile utilizzarli per attivare eventi o azioni anche quando l'applicazione non è in esecuzione e anche se il dispositivo stesso è in modalità di sospensione
- Ti aiutano a ridurre al minimo i requisiti di risorse dell'applicazione, programmando gli allarmi per eseguire operazioni specifiche e ridurre l'uso continuo della CPU o di altre risorse.

Best practices

Ogni scelta che si fa nella progettazione del proprio allarme ripetitivo può avere conseguenze su come l'applicazione utilizza (o abusa) delle risorse di sistema.

Aggiungere casualità (jitter) a tutte le richieste di rete che vengono attivate a seguito di un allarme ripetitivo:

- Esegue qualsiasi lavoro locale quando l'allarme viene attivato, con "lavoro locale" ci si riferisce a qualsiasi operazione che non coinvolge un server o non richiede dati dal server
- Allo stesso tempo, pianificare l'allarme contenente le richieste di rete per essere attivato in un periodo di tempo casuale

> Nota
> Bisogna mantenere la frequenza dell'allarme al minimo. Non bisogna svegliare il dispositivo inutilmente (questo comportamento è determinato dal tipo di allarme)

Non bisogna rendere il tempo di attivazione dell'allarme più preciso di quanto sia necessario, utilizzando `setInexactRepeating()` invece di `setRepeating()`.
Quando si utilizza `setInexactRepeating()`, Android sincronizza gli allarmi ripetitivi provenienti da più applicazioni e li attiva contemporaneamente, riducendo il numero totale di volte in cui il sistema deve attivare il dispositivo, e di conseguenza ridurre così il consumo della batteria.

> Nota
> A partire da Android 4.4 (API Livello 19), tutti gli allarmi ripetitivi sono inesatti, bisogna tener presente presente che, anche se `setInexactRepeating()` è un miglioramento rispetto a `setRepeating()`, può ancora sovraccaricare un server se ogni istanza di un'applicazione effettua richieste al server nello stesso momento. Pertanto per le richieste di rete bisogna aggiungere casualità agli allarmi

> Nota
> Gli allarmi ripetitivi basati su un tempo di attivazione preciso non scalano bene, è meglio utilizzare `ELAPSED_REALTIME` se possibile, i diversi tipi di allarme sono descritti in maggior dettaglio nella sezione seguente

Alarm Types

- `ELAPSED_REALTIME`, attiva l'intent in sospeso in base al tempo trascorso dal momento in cui il dispositivo è stato avviato, ma non sveglia il dispositivo. Il tempo trascorso include eventuali periodi in cui il dispositivo era in modalità di sospensione (sleep)
- `ELAPSED_REALTIME_WAKEUP`, sveglia il dispositivo e attiva l'intent in sospeso dopo il periodo di tempos specificato dal momento dell'avvio del dispositivo
- `RTC`, attiva l'intent in sospeso all'orario specificato ma non sveglia il dispositivo
- `RTC_WAKEUP`, sveglia il dispositivo per attivare l'intent in sospeso all'orario specificato

Esempio

Sveglia il dispositivo per attivare un allarme non ripetitivo (one-time) tra un minuto

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
// ...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);
alarmMgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, SystemClock.elapsedRealtime()+60*1000,
alarmIntent);
```

Sveglia il dispositivo per attivare l'allarme tra 30 minuti e ogni 30 minuti successivamente

```
alarmMgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
SystemClock.elapsedRealtime() + AlarmManager.INTERVAL_HALF_HOUR,
AlarmManager.INTERVAL_HALF_HOUR, alarmIntent);
```

Sveglia il dispositivo per attivare l'allarme approssimativamente alle 14:00 (2:00 p.m.), e ripete una volta al giorno alla stessa ora

```
// Set the alarm to start at approximately 2:00 p.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 14);
// With setInexactRepeating(), you have to use one of the AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
AlarmManager.INTERVAL_DAY, alarmIntent);
```

Sveglia il dispositivo per attivare l'allarme esattamente alle 8:30 a.m. e poi ogni 20 minuti successivamente

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
// ...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

// Set the alarm to start at 8:30 a.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 8);
calendar.set(Calendar.MINUTE, 30);

// setRepeating() lets you specify a precise custom interval--in this case,
// 20 minutes.
alarmMgr.setRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
        1000 * 60 * 20, alarmIntent);
```

Cancel Alarm

```
// If the alarm has been set, cancel it.
if (alarmMgr!= null) {
        alarmMgr.cancel(alarmIntent);
}
```

Power management

#daqui

Doze and App Standby

A partire da Android 6.0 (API level 23), che introduce due funzionalità di risparmio energetico, le quali prolungano la durata della batteria per gli utenti, gestendo il comportamento delle applicazioni quando il dispositivo non è collegato a una fonte di alimentazione:

- Doze, riduce il consumo della batteria ritardando l'attività in background della CPU e della rete per le applicazioni quando il dispositivo rimane inutilizzato per lunghi periodi di tempo. Mentre il dispositivo si trova in modalità Doze, l'accesso delle applicazioni a determinate risorse che consumano molta batteria viene ritardato fino alle finestre di manutenzione
- Standby, ritarda l'attività di rete in background per le applicazioni con cui l'utente non ha interagito di recente

Setting	Jobs *	Alarms †	Network ‡	Firebase Cloud Messaging §
User Restricts Background Activity				
Restrictions enabled:	Never	Never	Never	Messages discarded in Android P+ starting January 2019
Doze				
Doze active:	Deferred to window	Regular alarms: Deferred to window While-idle alarms: Deferred up to 9 minutes	Deferred to window	High priority: No restriction Normal priority: Deferred to window
App Standby Buckets (by bucket)				
Active:	No restriction	No restriction	No restriction	No restriction
Working set:	Deferred up to 2 hours	Deferred up to 6 minutes	No restriction	No restriction
Frequent:	Deferred up to 8 hours	Deferred up to 30 minutes	No restriction	High priority: 10/day
Rare:	Deferred up to 24 hours	Deferred up to 2 hours	Deferred up to 24 hours	High priority: 5/day

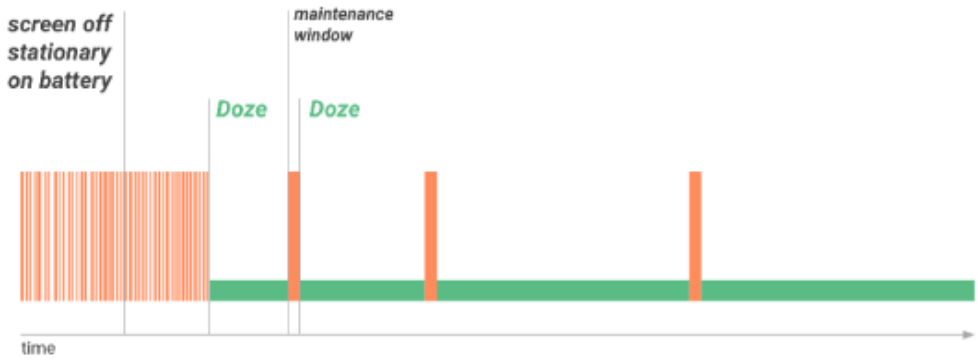
Understanding Doze

Se un utente lascia un dispositivo scollegato e fermo per un certo periodo di tempo, con lo schermo spento, il dispositivo entra in modalità Doze. In questa modalità, il sistema cerca di conservare la batteria limitando l'accesso delle app a servizi di rete e CPU intensivi ritardando lavori, sincronizzazioni e allarmi standard.

Periodicamente, il sistema esce dalla modalità Doze per un breve periodo di tempo per consentire alle app di completare le loro attività ritardate. Durante questa finestra di manutenzione, il sistema esegue tutte le sincronizzazioni, i lavori e gli allarmi in sospeso e consente alle app di accedere alla rete.

Al termine di ogni finestra di manutenzione, il sistema rientra nuovamente in modalità Doze, sospendendo l'accesso alla rete e ritardando i lavori, le sincronizzazioni e gli allarmi. Nel tempo, il sistema pianifica le finestre di manutenzione con una frequenza sempre minore, contribuendo a ridurre il consumo della batteria nei casi di inattività prolungata quando il dispositivo non è collegato a un caricatore.

Appena l'utente sveglia il dispositivo muovendolo, accendendo lo schermo o collegando un caricatore, il sistema esce dalla modalità Doze e tutte le app tornano all'attività normale



Doze Restrictions

Le seguenti restrizioni si applicano alle app durante la modalità Doze:

- L'accesso alla rete è sospeso
- Il sistema ignora i wake lock (blocco della CPU, del display, delle interfacce radio)
- Gli allarmi standard di `AlarmManager` (inclusi `setExact()` e `setWindow()`) sono posticipati alla prossima finestra di manutenzione
- Se si ha bisogno di impostare allarmi che si attivino durante la modalità Doze bisogna utilizzare `setAndAllowWhileIdle()` o `setExactAndAllowWhileIdle()`
- Gli allarmi impostati con `setAlarmClock()` continuano a funzionare normalmente, il sistema esce dalla modalità Doze poco prima che scattino tali allarmi

- Il sistema non esegue scansioni Wi-Fi
- Il sistema non consente l'esecuzione degli adattatori di sincronizzazione (sync adapters)
- Il sistema non consente l'esecuzione di JobScheduler

Adapt Your App to Doze

Doze può influenzare le app in modo diverso, a seconda delle funzionalità che offrono e dei servizi che utilizzano. Molte app funzionano normalmente durante i cicli di Doze senza modifiche, però in alcuni casi è necessario ottimizzare il modo in cui l'app gestisce la rete, gli allarmi, i job e le sincronizzazioni.

> Nota
> Le app dovrebbero essere in grado di gestire efficientemente le attività durante ogni finestra di manutenzione.

Doze potrebbe avere un impatto particolare sulle attività gestite dagli allarmi e timer di `AlarmManager`, poiché gli allarmi nelle versioni di Android 5.1 (API livello 22) o inferiori non vengono attivati quando il sistema è in modalità Doze.

Per aiutare nella pianificazione degli allarmi, Android 6.0 (API livello 23) introduce due nuovi metodi di `AlarmManager`:

- `setAndAllowWhileIdle()`
- `setExactAndAllowWhileIdle()`

Con questi metodi, è possibile impostare allarmi che si attiveranno anche se il dispositivo è in modalità Doze

La restrizione di Doze sull'accesso alla rete potrebbe anche influenzare la applicazione, specialmente se dipende da messaggi in tempo reale come le notifiche.

> Nota
> Se una applicazione richiede una connessione persistente alla rete per ricevere messaggi, bisognerebbe utilizzare Firebase Cloud Messaging (FCM) se possibile

---

Job Scheduler

JobScheduler è garantito per completare il lavoro assegnato, ma poiché opera a livello di sistema, può anche utilizzare diversi fattori per pianificare in modo intelligente il lavoro in background in modo da eseguirlo insieme ai job di altre app. Questo ci consente di ridurre al minimo il tasso di utilizzo, ottenendo così un chiaro risparmio di batteria.

> Nota
> A partire dall'API 24, `JobScheduler` tiene anche conto della pressione sulla memoria, ottenendo così un chiaro vantaggio complessivo per i dispositivi e i loro utenti

L'obiettivo con `JobScheduler` era trovare un modo per far sì che il sistema si assuma parte del carico di creazione di app performanti. Come sviluppatore, fai la tua parte per creare un'app che non si blocchi, ma ciò non sempre si traduce in una durata della batteria sana per il dispositivo.
Pertanto, introducendo `JobScheduler` a livello di sistema si può concentrarsi sulla raggruppamento di richieste di lavoro simili, il che si traduce in un miglioramento significativo sia per la durata della batteria che per la memoria.

Il lavoro che si desidera pianificare dovrebbe essere definito in un `JobService`, che sarà effettivamente un Service che estende la classe `JobService`.

```
public class MyService extends JobService {
        public MyService() {
        }
        // Metodo chiamato dal sistema quando è il momento di eseguire un lavoro
        @Override
        public boolean onStartJob(JobParameters params) {
                return false;
        }

        // Metodo chiamato dal sistema quando è il momento di terminare un lavoro prima del
su completamento
        @Override
        public boolean onStopJob(JobParameters params) {
                return false;
        }
}
```

> Nota
> Il `JobService` sarà eseguito sul thread principale, questo significa che bisogna gestire le task asincrone

Manifest

Come ogni servizio bisogna aggiungere il `JobScheduler` nel file `AndroidManifest.xml`

```
<service
        android:name=".sync.DownloadArtworkJobService"
        android:permission="android.permission.BIND_JOB_SERVICE"
        android:exported="true"/>
```

Conditions

Bisogna consoiderare le condizioni che devono essere vere affinché il proprio job venga eseguito.

> Nota
> Il grande vantaggio di JobScheduler è che non esegue il lavoro basandosi unicamente sul tempo, ma piuttosto sulle condizioni. Ciò significa che non è più necessario impostare un allarme ripetitivo per attivarsi ogni poche ore per controllare se è il momento giusto per sincronizzare con il server, per fare ciò si può definire questa richiesta tramite l'oggetto `JobInfo`

```
JobScheduler jobScheduler = (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);
jobScheduler.schedule(new JobInfo.Builder(LOAD_ARTWORK_JOB_ID,
                                          new ComponentName(this,
DownloadArtworkJobService.class))

.setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
                                          .build());
```

Available Conditions

Le condizioni sono:

- Tipo di rete (metered/non-metered), se il job richiede l'accesso alla rete, bisogna includere questa condizione, si può specificare una rete con tariffazione o senza tariffazione, o qualsiasi tipo di rete. Ma se non si chiama questa condizione durante la creazione del `JobInfo`, il sistema presumerà che c'è bisogno di alcun accesso alla rete e non sarà in grado di contattare il server
- Caricamento e inattività (Idle), se l'applicazione deve eseguire lavori che richiedono molte risorse, è altamente consigliabile aspettare fino a quando il dispositivo non è collegato all'alimentazione e/o inattivo.

> Nota
> Il termine "inattività" non è lo stesso della modalità Doze, significa semplicemente che lo schermo è spento e il dispositivo non è stato utilizzato per un po' di tempo

- Aggiornamento del Content Provider, a partire dall'API 24 si può utilizzare un cambiamento del Content Provider come trigger per eseguire alcuni lavori. È necessario specificare l'URI del trigger, che verrà monitorato con un `ContentObserver`. Si può anche specificare un ritardo prima che il job venga attivato, se si desidera assicurarsi che tutte le modifiche si propaghino prima di eseguire il job
- Criteri di ritrasmissione (Backoff criteria), si può specificare la propria politica di ritrasmissione/back-off. Il valore predefinito è una politica esponenziale, ma se si imposta la propria e quindi si restituisce `true` per ri-pianificare un job (con `onStopJob()`, ad esempio), il sistema utilizzerà la propria politica specificata invece di quella predefinita
- Latenza minima e scadenza di sovrascrittura, se il job non può iniziare per almeno una certa quantità di tempo (X), o non può essere ritardato oltre un orario specifico, si può specificare questi valori qui. Anche se non tutte le condizioni sono state soddisfatte, il job verrà eseguito entro la scadenza (si può verificare il risultato di `isOverrideDeadlineExpired()` per determinare se ci si troca in questa situazione), e se le condizioni sono state soddisfatte, ma non è trascorso il tempo di latenza minima, il job verrà mantenuto in sospeso.
- Periodico, se si ha delle attività che devono essere eseguite regolarmente, si può configurare un job periodico. Questa è un'ottima alternativa a un allarme ripetitivo per la maggior parte degli sviluppatori. Poiché lo si configura una volta, lo si pianifica, e il job verrà eseguito una volta in ciascun periodo specificato
- Persistente, qualsiasi lavoro che deve essere mantenuto anche dopo un riavvio del dispositivo può essere contrassegnato come persistente qui. Una volta che il dispositivo si riavvia, il job verrà ripianificato in base alle condizioni

> Nota
> La propria applicazione ha bisogno dell'autorizzazione `RECEIVE_BOOT_COMPLETED` per farlo funzionare

- Extra, se il job ha bisogno di alcune informazioni dall'applicazione per eseguire il lavoro, si può passare dati di tipo primitivo come extra nel JobInfo

Schedule & Cancel

Si può schedulare un lavoro usando `JobScheduler`, che si può ottenere dal sistema, e successivamente chiamando il metodo `schedule()` utilizzando l'oggetto `JobInfo`.

```
JobScheduler jobScheduler = (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);

jobScheduler.schedule(new JobInfo.Builder(LOAD_ARTWORK_JOB_ID,
                      new ComponentName(this, DownloadArtworkJobService.class))
                      .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
                      .build());

JobScheduler jobScheduler = (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);
jobScheduler.cancel(LOAD_ARTWORK_JOB_ID);
```

Background Tasks & Sleep Mode

La modalità di sospensione di Android principalmente spegne la CPU, proseguendo, anche le connettività non essenziali (WiFi, GPS) seguiranno la CPU.

> Note
> La modalità di sospensione non ferma la radio GSM o CDMA (per chiamate in entrata, SMS e pacchetti IP) e AlarmManager.

Per mantenere attiva un'attività in background quando il dispositivo entra in modalità di sospensione, si può utilizzare:

- La classe `PowerManager` per impostare un `WakeLock` (mantiene attiva la CPU e/o lo schermo) e/o un `WifiManager.WifiLock` per mantenere attiva la connettività Wi-Fi
- `AlarmManager` e `AlarmReceiver` per ricevere periodicamente un intent e riavviare un'attività in background

---

Status Bar Notifications

La notifica della barra di stato aggiunge un'icona alla barra di stato del sistema (con un messaggio di testo opzionale `ticker-text`) e un messaggio di notifica nella finestra delle notifiche. Quando l'utente seleziona la notifica, Android attiva un intent definito dalla notifica (di solito per avviare un'Activity).

> Nota
> È anche possibile configurare la notifica per avvisare l'utente con un suono, una vibrazione e luci lampeggianti sul dispositivo

Una notifica della barra di stato dovrebbe essere utilizzata in ogni caso in cui un servizio in background che deve avvisare l'utente di un evento che richiede una risposta.

> Nota
> Un servizio in background non dovrebbe mai avviare un'attività da solo per ricevere l'interazione dell'utente, dovrebbe invece creare una notifica della barra di stato che avvierà l'attività quando selezionata dall'utente

Un'Activity o un Service possono avviare una notifica della barra di stato, poiché un'attività può eseguire azioni solo mentre è in esecuzione in primo piano e la sua finestra ha il focus, di solito si creerà notifiche della barra di stato da un servizio, in questo modo, la notifica può essere creata dal background mentre l'utente sta utilizzando un'altra applicazione o mentre il dispositivo è in stato di sospensione.

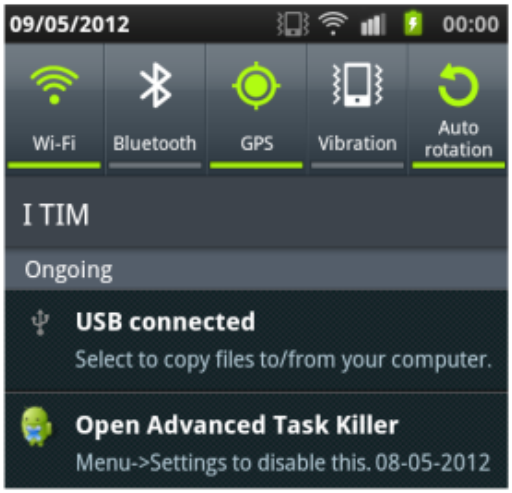
> Nota
> Le classi dedicate alla creazione e gestione delle notifiche sono `Notification` e `NotificationManager`

Un oggetto `Notification` definisce i dettagli del messaggio di notifica che viene visualizzato nella barra di stato e nella finestra delle notifiche, e ogni altra impostazione di allerta, come suoni e luci lampeggianti. L'oggetto in questione richiede:

- Un icona per la barra di stato
- Un titolo ed un messaggio, a meno che non si definisce un layout di notifica personalizzato
- Un `PendingIntent`, da essere risvegliato quando la notifica viene selezionata

Le impostazioni opzionali per la notifica della barra di stato includono:

- Un messaggio `ticket-text` per la barra di stato
- Un suono di allerta
- Un impostazione di vibrazione
- Un'impostazione per il lampeggio del LED



Il `NotificationManager` è un servizio di sistema di Android che esegue e gestisce tutte le notifiche della barra di stato, non si istanzia direttamente il `NotificationManager`. Per fornirgli una notifica, bisogna ottenere un riferimento al `NotificationManager` con `getSystemService()` e, quando si desidera notificare l'utente, bisogna passargli la notifica tramite `notify()`.

```
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager mNotificationManager = (NotificationManager) getSystemService(ns);
```

Quando si desidera inviare una notifica alla barra di stato, bisogna passare la Notifica al `NotificationManager` con `notify(int, Notification)`. Il primo parametro è l'ID univoco per la notifica e il secondo è l'oggetto Notifica.

> Nota
> L'ID identifica in modo univoco la notifica all'interno dell'applicazione. L'ID è necessario se si deve aggiornare la notifica o (se la tua applicazione gestisce diversi tipi di notifiche) selezionare l'azione appropriata quando l'utente torna alla tua applicazione tramite l'intent definito nella notifica

Per cancellare la notifica della barra di stato quando l'utente la seleziona dalla finestra delle notifiche, bisogna aggiungere il flag `FLAG_AUTO_CANCEL` alla notifica.

> Nota
> È possibile anche cancellarla manualmente con `cancel(int)`, passando l'ID della notifica, o cancellare tutte le tue notifiche con `cancelAll()`

Normalmente Android considera tutte le attività all'interno di un'applicazione come parte del flusso dell'interfaccia utente di quella applicazione, quindi avviare semplicemente l'attività in questo modo può far sì che venga mescolata con lo stack di back dell'applicazione normale in modi indesiderati.

Per farla comportare correttamente, nella dichiarazione del manifesto per l'attività devono essere impostati gli attributi:

- `android:launchMode="singleTask"`
- `android:taskAffinity=""`
- `android:excludeFromRecents="true"`

La dichiarazione completa dell'attività per questo esempio è:

```
<activity android:name=".app.IncomingMessageInterstitial"
                        android:label="You have messages"
                        android:theme="@style/ThemeHoloDialog"
                        android:launchMode="singleTask"
                        android:taskAffinity=""
                        android:excludeFromRecents="true">
</activity>
```

```
// Prepare the intent triggered if the notification is selected
Intent intent = new Intent(this, MainActivity.class);
PendingIntent pIntent = PendingIntent.getActivity(this, 0, intent, 0);

// Build the notification
// Use NotificationCompat.Builder instead of just Notification.Builder to support older
Android versions
Notification n = new NotificationCompat.Builder(this)
                                        .setContentTitle("MobDev - YouTube App")
                                        .setContentText("New Video List Available")

.setSmallIcon(R.drawable.icon_notification_exclamation_mark)
                                        .setContentIntent(pIntent)
                                        .setAutoCancel(true)
                                        .build();

NotificationManager notificationManager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
notificationManager.notify(0, n);
```

Status Bar Notifications + Actions

```
Notification n = new Notification.Builder(this)
                                        .setContentTitle("Missed Call")
                                        .setContentText("Home")
                                        .setSmallIcon(R.drawable.user__icon)
                                        .setContentIntent(pIntent)
                                        .setAutoCancel(true)
                                        .addAction(R.drawable.reply_icon, "Call back",
callBackIntent)
                                        .addAction(R.drawable.delete_icon, "Message",
messageIntent).build();
```

È possibile aggiungere azioni specifiche alla prorpia notifica.
Un'azione consente all'utente di passare direttamente dalla notifica in arrivo a un'attività (definita tramite un intent) nell'app, dove è possibile visualizzare uno o più eventi o svolgere ulteriori attività.

Android 8 & Notifications Channels

A partire da Android 8.0 (API livello 26), i canali di notifica consentono di creare un canale personalizzabile dall'utente per ogni tipo di notifica che si desidera visualizzare.

> Nota
> I canali di notifica forniscono un sistema unificato per aiutare gli utenti a gestire le notifiche

Gli utenti possono gestire la maggior parte delle impostazioni associate alle notifiche utilizzando un'interfaccia utente di sistema coerente, ovviamente tutte le notifiche inviate allo stesso canale di notifica hanno lo stesso comportamento.

Quando un utente modifica il comportamento per una delle seguenti caratteristiche, ciò si applica al canale di notifica:

- Importanza
- Suono
- Luci
- Vibrazione
- Visualizza sulla schermata di blocco
- Override "Non disturbare" (Do Not Disturb)

Creating a notification channel

```
NotificationManager mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
String id = "my_channel_01";

// The user-visible name of the channel.
CharSequence name = getString(R.string.channel_name);

// The user-visible description of the channel.
String description = getString(R.string.channel_description);
int importance = NotificationManager.IMPORTANCE_HIGH;
NotificationChannel mChannel = new NotificationChannel(id, name, importance);

// Configure the notification channel.
mChannel.setDescription(description);
```

```
mChannel.enableLights(true);

// Sets the notification light color for notifications posted to this
// channel, if the device supports this feature.
mChannel.setLightColor(Color.RED);
mChannel.enableVibration(true);
mChannel.setVibrationPattern(new long[]{100, 200, 300, 400, 500, 400, 300, 200, 400});
mNotificationManager.createNotificationChannel(mChannel);
```

Example Code

```
NotificationManager notificationManager =
(NotificationManager)getSystemService(Context.NOTIFICATION_SERVICE);
NotificationCompat.Builder builder = null;

// Nuovo approccio con channels
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
        int importance = NotificationManager.IMPORTANCE_DEFAULT;
        NotificationChannel notificationChannel = new NotificationChannel("ID", "Name",
importance);
        notificationManager.createNotificationChannel(notificationChannel);
        builder = new NotificationCompat.Builder(getApplicationContext(),
notificationChannel.getId());
        } else {
        builder = new NotificationCompat.Builder(getApplicationContext());
}

Notification notification = builder.setContentTitle("MobDev - App")
                                                        .setContentText("New Data Available")

.setSmallIcon(R.drawable.icon_notification_exclamation_mark)

.setContentIntent(pIntent).setAutoCancel(true).build();
notificationManager.notify(0, notification);
```

Notification Style

A partire da Android 4.1, oltre alla visualizzazione normale della notifica, è anche possibile definire una vista più ampia (big view) che viene mostrata quando la notifica viene espansa.
Ci sono tre stili da utilizzare con la vista più ampia:

- "big picture" (immagine grande)
- "big text" (testo grande)
- "Inbox" (casella di posta)

```
String longText = "bla bla bla bla ... :) ";
Notification noti = new Notification.Builder(this).
                                                // [...]
.setStyle(new Notification.BigTextStyle().bigText(longText))
```