Indice

Introduzione

SOMMARIO: 1. Internet of Things – 2. Google e IoT – 3. Obiettivi e struttura tesi

1. Internet of Things

L'Internet delle cose, o meglio *Internet of Things*, è prima di tutto un'idea, quella di poter comunicare con ogni cosa e di far interagire gli oggetti anche fra loro. L'IoT rappresenta la rete dove è possibile riconoscere oggetti che acquisiscono intelligenza e capacità di "parola". Queste cose intelligenti, o *smart objects*, presentano delle caratteristiche fondamentali quali la possibilità di essere individuati, localizzati, di essere connessi alla rete, di poter elaborare dati e di poter interagire con l'ambiente esterno.

Si supponga che la mattina ci sia del traffico sul solito tragitto tra casa e lavoro, la sveglia smart, connessa alla rete, recepita quest'informazione, potrebbe suonare prima del previsto. Oppure si pensi a tutti quegli oggetti che bisogna utilizzare regolarmente, o assumere, come un farmaco, che potrebbero avvertire l'utente del mancato utilizzo. Insomma, le possibilità sono molteplici e variano a seconda dei vari domini applicativi, come ad esempio la domotica, la robotica, la telemetria, le reti wireless di sensori, la sorveglianza, i sistemi embedded e molto altro.

Si stima che gli oggetti intelligenti superino quota trenta miliardi in numero nel giro di tre anni. Si capisce, quindi, quanto questa sia ormai una realtà affermata e quanto sia giusto spendere e investire in questo ambiente. Infatti, la crescita di IoT può comportare anche un notevole risparmio energetico, sia a livello personale, con la domotica e *smart-home*, sia ad un livello più alto, *smart-city* e *smart industry*.

2. Google e IoT

Google, date le ottime premesse dell'IoT, ha deciso di non rimanere in disparte e ha infatti introdotto, dopo un anno e mezzo di fase di test, nel maggio 2018, Android Things 1.0 ovvero una versione stabile del famoso sistema operativo del colosso di Mountain View per IoT. L'idea è stata quella di creare un

ambiente robusto e sicuro, compatibile con altri servizi offerti dalla stessa casa produttrice. Google fornisce, quindi, una piattaforma robusta per hardware certificato, un ricco insieme di API di sviluppo e aggiornamenti software e di sicurezza. Quello che rimane è la costruzione del prodotto che si intende sviluppare, che è lasciata a coloro che risultino interessati al sistema offerto dalla casa del motore di ricerca più famoso al mondo.

Il loro obiettivo primario non è solo quello di aiutare le compagnie a vedere i loro prodotti disponibili sul mercato il più velocemente possibile, ma anche di migliorarne la qualità. Infatti, Google mette a disposizione molte tecnologie già robuste e sviluppate quali quelle relative al *machine learning*, riconoscimento vocale e servizi *cloud*. Ad esempio, nel Consumer Electronics Show, tenutosi nel Gennaio 2018, alcune compagnie mostravano prototipi di altoparlanti smart basati su Android Things che usavano per il riconoscimento vocale il famoso software Google Assistant.

3. Obiettivi e struttura tesi

La tesi verte su quale sia la metodologia di sviluppo fornita da Android per la programmazione mirata all'Internet delle cose. In particolare, si vuole capire quali siano i vantaggi e, di contro, le limitazioni che il sistema offre. Per farlo, si va ad analizzare il tutto sviluppando in prima persona un'app per Things. Si vuole infatti osservare, partendo da una conoscenza molto basilare sia di programmazione in ambiente Android che di programmazione per sistemi embedded, quanto la metodologia di sviluppo sia *newbie-friendly*, quanto questa sia diversa da altre tipologie di sviluppo e quanto tempo sia veramente necessario per ottenere risultati tangibili.

La tesi si articola in tre parti sostanzialmente:

- La prima parte è dedicata all'ambiente Android. In particolare si vuole osservare le caratteristiche fondamentali, ovvero che cos'è,

quali sono i suoi componenti fondamentali, che cos'è Android Studio e quali sono le caratteristiche di Android Things.

- La seconda parte è dedicata allo sviluppo di un'applicazione per Android Things. Si descrive innanzitutto il funzionamento dell'app e quali siano i componenti hardware e software utilizzati. Si mostra infine il codice sviluppato, analizzato in dettaglio e possibili esempi d'uso dell'applicazione.
- La terza parte si basa sulle conclusioni.

CAPITOLO I

Ambiente Android

SOMMARIO: 1. Android: architettura e struttura app – 1.1 Componenti fondamentali di un'applicazione – 1.2 Attivazione dei componenti – 1.3 *Manifest file* – 1.4 Risorse delle applicazioni – 1.5 Ciclo di vita di un'applicazione – 1.6 Ciclo di vita di una *activity* – 2. Android Studio – 3. Android Things

1. Android: architettura e struttura app

Android è un sistema operativo basato sul kernel Linux, sviluppato da Google e progettato principalmente per dispositivi mobili. Android è inoltre un progetto open source, guidato da Google, chiamato AOSP (*Android Open Source Project*). La stessa Google usa questo progetto come base per creare la sua versione di Android, poi usata da altri produttori.

Tra i vari strumenti software utili alla programmazione in Android è necessario sottolineare è Android SDK, pacchetto di strumenti con cui si interagisce spesso tramite l'IDE (ambiente di sviluppo integrato).

Le applicazioni Android possono essere scritte usando vari linguaggi di programmazione, ovvero Kotlin, Java e C++. Gli strumenti SDK di Android compilano il codice, insieme con altri dati e file di risorse, in un APK, *Android package*, ossia un file di archivio con suffisso .apk . Un file APK contiene tutti gli elementi della App ed è il file che i dispositivi basati su Android usano per installare l'applicazione.

Ogni applicazione Android vive protetta da varie caratteristiche di sicurezza quali:

- Il sistema operativo Android è un sistema Linux multi-utente in cui ogni applicazione è un utente diverso.
- Il sistema assegna ad ogni app un esclusivo ID utente Linux e inoltre imposta i permessi per tutti i file in un app, cosicché solo l'utente avente l'ID corretto possa accedervi.
- Ogni processo ha la sua macchina virtuale (VM), in modo tale che il codice di un applicazione sia eseguito in maniera isolata dalle altre app.
- Ogni applicazione funziona in un proprio processo Linux che il sistema Android inizializza, quando un componente di un app deve essere eseguito, o spegne, quando il processo non è più necessario o per recuperare memoria.

Un’applicazione non può avere accesso a parti del sistema per i quali non è stato garantito il permesso. Questo crea un ambiente sicuro, perché ogni app può solo lavorare con i file utili per svolgere il suo lavoro e niente più. Tuttavia, un’app può condividere i dati con un’altra applicazione in vari modi:

- È possibile fare in modo che due applicazioni condividano lo stesso ID utente Linux, in tal caso sarebbe possibile per ognuna di esse di accedere ai file dell’altra; potrebbero inoltre condividere lo stesso processo per risparmiare risorse di sistema.
- Un’applicazione può richiedere i permessi per accedere ai dati del dispositivo come i contatti dell’utente, i messaggi, la camera e il Bluetooth. In tal caso l’utente deve fornire i permessi personalmente.

1.1 Componenti fondamentali di un’applicazione

Ci sono quattro tipi di componenti che formano una App:

- *Activity*
- *Service*
- *Broadcast receiver*
- *Content provider*

Ognuno di questi componenti determina un punto attraverso il quale il sistema o l’utente può accedere. Ognuno ha una distinta finalità e un distinto ciclo di vita che definisce quando il componente è creato o distrutto.

Activity: un’*activity* è il punto di accesso per l’interazione con l’utente. Rappresenta un singolo schermo con un’interfaccia utente. Ad esempio, un’app per le email potrebbe avere un’*activity* per mostrare la lista delle nuove email, una per comporne una nuova da inviare, un’altra per la lettura. Sebbene le *activity* lavorino insieme per formare una coesiva esperienza per l’utente, nell’esempio

precedente ognuna di queste è indipendente dalle altre. Infatti, un'altra applicazione può far partire una qualsiasi *activity* dell'app delle email, se questa lo permette. Ad esempio, un'app per la fotocamera può inizializzare l'*activity* presente nell'app delle email che permette la composizione di una nuova mail per permettere all'utente di inviare una fotografia. Un'*activity* facilita i seguenti punti chiave dell'interazione tra utente e sistema:

- Tenere traccia di cosa interessa al momento all'utente (quello che è sullo schermo) per assicurare che il sistema continui ad eseguire il processo che sta ospitando l'*activity*.
- Sapere se i processi usati in precedenza contengano cose su cui l'utente potrebbe voler ritornare (*stopped activities*) e conservarli dando loro maggiore priorità.
- Aiutare l'app ad eliminare il suo processo cosicché l'utente possa ritornare alle *activity*, con il loro stato precedente recuperato.

Service: un *service* è un componente che permette di tenere in esecuzione un'app in background per qualsiasi tipo di ragione. Viene eseguita in background per svolgere operazioni di lunga durata o per svolgere il lavoro per processi remoti. Un *service* non fornisce un'interfaccia utente. Ad esempio, un *service* potrebbe far suonare la musica in sottofondo mentre l'utente è in un'app diversa, o potrebbe acquisire dati dalla rete senza bloccare l'interazione dell'utente con un'*activity*. Un altro componente, come l'*activity*, può inizializzare un *service* e lasciarlo in esecuzione, o unirsi a questo per interagirci. Ci sono due distinti tipi semantici di *service* che dicono al sistema come gestire un'app:

- Gli *started services* dicono al sistema di mantenere in esecuzione le app finché non concludono il loro compito. Questi potrebbero essere usati per sincronizzare dei dati o per far suonare la musica in sottofondo. Quest'ultimi due esempi rappresentano tipi diversi di *started services* che il sistema gestisce in maniera differente: la musica in sottofondo è qualcosa di cui l'utente è cosciente, così l'app dice al sistema di voler essere in primo piano con una notifica che

informa l'utente di questo; in tal caso, il sistema si impegna a tenere in esecuzione il processo di quel *service* per non scontentare l'utente. L'altro tipo di *started services* è quello di un regolare *service* di background, di cui l'utente non è direttamente consapevole, cosicché il sistema ha più libertà nel gestire il suo processo; in tal caso, potrebbe permettere al sistema di farsi terminare (e di ripartire in un secondo momento eventualmente) se c'è bisogno di RAM per altri processi più importanti per l'utente.

- I *bound services* si eseguono perché altre app (o il sistema) vogliono far uso del service. Questo è praticamente un *service* che fornisce un API ad un altro processo. Il sistema, in questo modo, sa che c'è una dipendenza tra i processi; se il processo A è legato al *service* nel processo B, il sistema capisce che ha bisogno di mantenere il processo B (e il suo *service*) in esecuzione per A. Ulteriormente, se il processo A è qualcosa a cui l'utente tiene, allora il sistema sa che deve trattare allo stesso modo anche il processo B.

Per la loro flessibilità (in meglio e in peggio), i *service* risultano essere dei blocchi fondamentali per qualsiasi concetto di sistema di alto livello. Sfondi interattivi, notifiche, salva schermi, metodi di input, servizi di accessibilità, e molte altre funzioni fondamentali del sistema sono tutte costruite come *service*, che le applicazioni implementano, e a cui il sistema le unisce quando devono essere eseguite.

Broadcast receiver: un *broadcast receiver* è un componente che permette al sistema di comunicare ad un'app eventi al di fuori del regolare *user flow*, permettendole di rispondere ad avvisi trasmessi a livello di sistema. Poiché i *broadcast receiver* sono un altro ben definito punto di accesso ad un'app, il sistema può consegnare trasmissioni anche ad app che non sono in esecuzione. Ad esempio, un'app può programmare una sveglia per inviare una notifica all'utente riguardo un evento imminente e, consegnando quella sveglia ad un *broadcast receiver* dell'app, non c'è bisogno che l'app rimanga in esecuzione finché la sveglia non si spegne. Molte trasmissioni si originano dal sistema (ad esempio una

notifica che annuncia che lo schermo si sta per spegnere, la batteria è quasi scarica, o che è stato fatto uno screenshot), ma anche le app possono farle avviare (ad esempio per informare un'altra app che sono stati scaricati dei dati sul dispositivo e che sono dunque disponibili all'uso). Sebbene molti *broadcast receiver* non mostrino un'interfaccia utente, essi potrebbero creare una notifica nella barra di stato per avvertire l'utente quando si verifica un evento trasmesso. Più comunemente, un *broadcast receiver* è solo una via di passaggio per altri componenti ed è pensato per svolgere una mole di lavoro modesta.

Content provider: un *content provider* gestisce blocchi di dati condivisi tra le app che si possono immagazzinare in un file di sistema, in un database SQLite, sul web, o in una qualsiasi locazione di memoria a cui l'app può accedere. Attraverso il *content provider*, le altre app possono richiedere o modificare i dati se lo stesso lo consente. Per esempio, il sistema Android fornisce un *content provider* che gestisce le informazioni dei contatti dell'utente. Per cui, qualsiasi app con i giusti permessi può richiedere al *content provider* di leggere e scrivere informazioni riguardo una determinata persona. È facile pensare al *content provider* come l'astrazione di un database, infatti ci sono molte API e supporto interno basati su questa idea. Tuttavia, i *content provider* assumono un significato diverso se visti da una prospettiva *system-design*. I *content provider* sono anche utili per leggere e scrivere dati che sono privati per un'applicazione e non condivisi con essa.

Un aspetto unico del design del sistema Android è che un'applicazione può eseguire un componente di un'altra app. Ad esempio, se si vuole in una prima app che un utente scatti una foto con la fotocamera, c'è probabilmente una seconda app che già lo può fare e la prima app può sfruttare l'*activity* dell'altra app senza bisogno di svilupparla da zero. Infatti, è possibile eseguire l'*activity* nell'app della camera che scatta una fotografia e, una volta fatto, la foto ritorna nella prima app, così da poterla utilizzare. Per l'utente, è come se la camera fosse parte di quell'app.

Quando il sistema inizializza un componente, fa partire il processo di quell'app, se non è già in esecuzione, e istanzia le classi necessarie per quel componente. Per esempio, se una prima app inizializza l'*activity* per scattare una foto nell'app della camera, l'*activity* è eseguita nel processo dell'app della camera e non nel processo della prima app. Pertanto, a differenza delle applicazioni di altri sistemi, le app Android non hanno un singolo punto di accesso (non c'è un metodo *main()*).

1.2 Attivazione dei componenti

Poiché il sistema esegue ogni applicazione in processi diversi con permessi che limitano l'accesso alle altre app, una prima app non può attivare direttamente il componente di una seconda app. Tuttavia, il sistema Android può. Per attivare il componente di un'altra app, bisogna inviare un messaggio al sistema che specifica l'intenzione di inizializzare quel particolare componente. È poi il sistema che inizializza il componente.

Tre dei quattro tipi di componenti (*activity*, *service* e *broadcast receiver*) sono attivati da un messaggio asincrono chiamato *intent* (il *content provider* non è attivato dagli *intent* ma da una richiesta di un oggetto `ContentResolver` per ragioni di sicurezza). Gli *intent* legano i singoli componenti gli uni con gli altri durante l'esecuzione. Si possono immaginare come dei messaggeri che richiedono un'azione da un componente, che appartenga alla medesima app o meno.

Un *intent* è creato con un oggetto `Intent`, che definisce un messaggio per attivare o uno specifico componente (*intent* esplicito), o uno specifico tipo di componente (*intent* implicito). Per le *activity* e i *service*, un *intent* definisce un'azione da svolgere (ad esempio, vedere o inviare qualcosa) e potrebbe specificare l'URI (*Uniform Resource Identifier*) dei dati su cui agire, tra le varie altre cose di cui il componente inizializzato potrebbe aver bisogno. Per esempio, un *intent* potrebbe trasmettere una richiesta ad una *activity* per mostrare un'immagine o per aprire un sito web. In alcuni casi, è possibile inizializzare

un'*activity* per ricevere un risultato che la stessa potrebbe restituire tramite un *intent*.

Per i *broadcast receiver*, l'*intent* definisce semplicemente l'avviso che è stato trasmesso. Per esempio, una trasmissione, per indicare che il livello della batteria è basso, include solo una stringa.

1.3 Manifest File

Prima che il sistema Android esegua un'app, questo ha bisogno di sapere che il componente esiste leggendolo dal *manifest file* dell'applicazione, ovvero il file `AndroidManifest.xml`. L'app deve dichiarare tutti i suoi componenti in questo file, che deve essere posto all'inizio della directory del progetto. Altre funzioni del *manifest file*, oltre a quella della dichiarazione dei componenti dell'app, sono:

- Identificare ogni permesso da ottenere dall'utente che l'app richiede, come l'accesso ad Internet o alle informazioni dei contatti dell'utente.
- Dichiarare il livello minimo di API che richiede l'app, basato su quali API l'app usa.
- Dichiarare quali caratteristiche software o hardware sono richieste dalla applicazione, come la fotocamera, i servizi bluetooth, o uno schermo multitouch.
- Dichiarare librerie API a cui l'app ha bisogno di essere collegata (oltre quelle del *framework* di Android).

Nel *manifest file* bisogna dichiarare tutti i componenti dell'app usando i seguenti elementi:

- `<activity>` per le *activity*
- `<service>` per i *service*
- `<receiver>` per i *broadcast receiver*
- `<provider>` per i *content provider*

Activity, *service* e *content provider* che sono inclusi nel file sorgente, ma che non sono dichiarati nel *manifest file*, non sono visibili al sistema e, di conseguenza, non possono mai essere eseguiti. Tuttavia, i *broadcast receiver* possono sia essere dichiarati nel *manifest file* che creati dinamicamente nel codice come oggetti `BroadcastReceiver` e registrati con il sistema chiamando il metodo `registerReceiver()`.

Come discusso nel sotto-paragrafo precedente, un *intent* può essere usato per inizializzare un'*activity*, un *service* o un *broadcast receiver*. Lo si può usare nominando esplicitamente il componente (usando il nome della classe del componente) oppure, implicitamente, descrivendo il tipo di azione da eseguire e, facoltativamente, i dati su cui agire. L'*intent* implicito permette al sistema di trovare un componente sul dispositivo che può eseguire l'azione e lo inizializza. Se ci sono più componenti capaci di eseguire quell'azione descritta dall'*intent*, l'utente seleziona quale utilizzare.

Il sistema identifica i componenti sul dispositivo che possono rispondere ad un i*ntent* confrontando l'*intent* ricevuto con gli *intent filter* forniti nel *manifest file* delle altre applicazioni sul dispositivo. Gli *intent filter* possono essere dichiarati nel *manifest file* aggiungendo l'elemento `<intent-filter>` come figlio dell'elemento di dichiarazione del componente.

C'è una vasta varietà di dispositivi forniti del sistema Android e non tutti forniscono le stesse caratteristiche e le stesse capacità. Per prevenire che un'app sia installata su un dispositivo che manca di funzioni necessarie per la stessa, è importante definire chiaramente un profilo per i tipi di dispositivi che l'app supporta dichiarando i requisiti software e hardware nel *manifest file*. La maggior parte di queste dichiarazioni sono solo informative e il sistema non le legge, ma alcuni servizi esterni, come Google Play, sono in grado di recepirle filtrando nella ricerca le app non adatte al dispositivo dell'utente.

1.4 Risorse delle applicazioni

Un'app Android non è composta dal solo codice, richiede risorse che sono separate dal file sorgente, come immagini, file audio e qualsiasi altra cosa legata alla presentazione visuale dell'app. Per esempio, è possibile definire animazioni, menu, stili, colori, e il layout dell'interfaccia utente dell'*activity* con dei file XML. Usando queste risorse, è reso più facile l'aggiornamento di varie caratteristiche dell'app, infatti non se ne modifica il codice. Fornire dei gruppi di risorse alternativi permette di ottimizzare l'app sviluppata per una grande varietà di configurazioni dei dispositivi, come lingue diverse o grandezze dello schermo differenti.

Per ogni risorsa che viene inclusa nel progetto Android, gli strumenti *SDK build* definiscono un preciso ID in formato intero, che è possibile usare per riferirsi alle risorse dal codice dell'app o ad altre risorse definite in XML. Ad esempio, se un'app contiene un'immagine chiamata logo.png (salvata nella cartella res/drawable/), gli strumenti SDK generano un ID di risorsa chiamato R.drawable.logo . Questo ID si associa ad un intero specifico per l'app, che può essere utilizzato per riferirsi all'immagine e per poi inserirlo nell'interfaccia utente.

Uno degli aspetti del fornire risorse separate dal codice sorgente è la possibilità di definire risorse diverse per configurazioni di dispositivi diversi. Per esempio, definendo stringhe per l'interfaccia utente in XML, è possibile tradurre le stringhe in altre lingue e poi salvare quelle stringhe in file separati. Allora Android applica le stringhe con la lingua appropriata per l'interfaccia utente basandosi su un *qualifier* della lingua che viene aggiunto al nome della cartella della risorsa (come res/values-it/ per i valori italiani delle stringhe) e le impostazioni della lingua dell'utente.

Android supporta molti *qualifiers* per le risorse alternative. Un *qualifier* è un piccolo testo che va incluso nel nome delle cartelle delle risorse in modo da definire la configurazione del dispositivo per la quale dovrebbero essere usate le

stesse. Ad esempio, si potrebbero creare diversi layout per le *activity*, dipendentemente dalle dimensioni dello schermo e dall'orientamento dello stesso. Si potrebbe volere una posizione diversa per i bottoni a seconda dell'orientamento, in basso per quello a ritratto e a sinistra per quello panoramico. Per cambiare layout a seconda dell'orientamento, si potrebbero definire due layout differenti e applicare l'opportuno qualificatore al nome della cartella di ogni di questi. Allora, il sistema, in corrispondenza dell'orientamento opportuno, applicherebbe automaticamente il layout giusto.

1.5 Ciclo di vita di un'applicazione

Android sa che il fattore fondamentale della sopravvivenza di un sistema mobile è la corretta gestione delle risorse. Si pensi ad uno smartphone: è un dispositivo che non solo si occupa di chiamate ed SMS, ma offre pagine web, giochi, comunicazione sui "social" per molto tempo ogni giorno. Inoltre, capita sempre più spesso che non venga mai spento impedendo così una fase molto comune nella vita dei PC: l'arresto del sistema con conseguente liberazione della memoria e pulizia di risorse temporanee assegnate.

Android fa in modo di tenere in vita ogni processo il più a lungo possibile. Ciò non toglie che in alcune circostanze ed in base alle risorse hardware a disposizione, il sistema operativo si troverà nella necessità di dover liberare memoria abbattendo processi. La discriminante per la chiusura di un processo è quanto un'applicazione, candidata all'eliminazione, sia importante per la *user experience*. Maggiore sarà l'importanza riconosciuta, minori saranno le probabilità che venga arrestata. Così facendo Android tenta di raggiungere il suo duplice scopo: preservare il sistema e salvaguardare l'utente.

I processi possono essere classificati, in ordine di importanza decrescente, come:

- Processi in *foreground*: sono quelli che interagiscono direttamente o indirettamente con l'utente. Stiamo parlando delle applicazioni che, ad esempio,

contengono l'*activity* attualmente utilizzata o i *service* ad essa collegati. Questi sono i processi che Android tenterà di preservare maggiormente. Importante notare che, comunque, anche le applicazioni in *foreground* non sono del tutto al sicuro. Se ad esempio il sistema non disponesse di risorse sufficienti a mantenerli tutti in vita, si troverebbe costretto ad arrestarne qualcuno;

- Processi visibili: non sono importanti come quelli in *foreground* ma vengono anch'essi grandemente tutelati da Android. Infatti, avendo componenti ancora visibili all'utente anche se non vi interagiscono più, svolgono comunque un ruolo particolarmente critico. Anche in questo caso si tratta di *activity* visibili e s*ervice* ad esse collegati;
- Processi *service*: contengono dei *service* in esecuzione che generalmente svolgono lavori molto utili all'utente anche se non direttamente collegati con ciò che egli vede nel display. Il loro livello di priorità può essere considerato medio: importanti sì ma non tanto quanto i processi di cui ai precedenti due punti;
- Processi in "*background*": contengono *activity* non più visibili all'utente. Questa è una categoria solitamente molto affollata composta dal gran numero di applicazioni che l'utente ha usato e messo poi in disparte, ad esempio premendo il tasto *Home*. Non sono considerati molto importanti e sono dei buoni candidati all'eliminazione in caso di scarsità di risorse;
- Processi "*empty*": sono praticamente vuoti nel senso che non hanno alcuna componente di sistema attiva. Vengono conservati solo per motivi di cache, per velocizzare la loro riattivazione qualora si rendesse necessaria. Come ovvio, sono i candidati "numero 1" all'eliminazione da parte del sistema operativo.

1.6 Ciclo di vita di un'activity

Una delle più note illustrazioni della programmazione Android è questa:

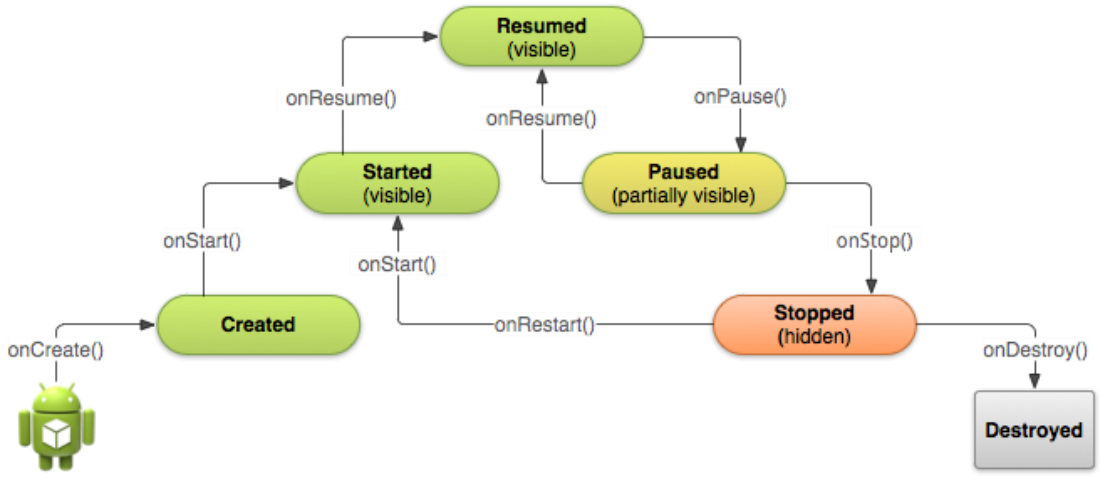


Figura 1 - Ciclo di vita di una Activity

La si può trovare sulla documentazione ufficiale, nelle pagine in cui viene spiegato il ciclo di vita di un'*activity*. Si tratta di una serie di stati attraverso i quali l'esistenza dell'*activity* passa. In particolare, nell'illustrazione riportata, gli stati sono rappresentati dalle figure colorate. L'ingresso o l'uscita da uno di questi stati viene notificato con l'invocazione di un metodo di *callback* da parte del sistema. Il codice inserito in tali metodi dovrà essere allineato con la finalità del metodo stesso affinché l'app possa essere "un buon cittadino" dell'ecosistema Android.

Quando un'*activity* va in esecuzione per interagire direttamente con l'utente vengono obbligatoriamente invocati tre metodi:

- *onCreate*: l'*activity* viene creata. Il programmatore deve assegnare le configurazioni di base e definire quale sarà il layout dell'interfaccia;
- *onStart*: l'*activity* diventa visibile. È il momento in cui si possono attivare funzionalità e servizi che devono offrire informazioni all'utente;
- *onResume*: l'*activity* diventa la destinataria di tutti gli input dell'utente.

Android pone a riposo l'*activity* nel momento in cui l'utente sposta la sua attenzione su un'altra attività del sistema, ad esempio apre un'applicazione diversa, riceve una telefonata o semplicemente – anche nell'ambito della stessa applicazione – viene attivata un'altra *activity*. Anche questo percorso, passa per tre metodi di *callback*:

- *onPause* (l'inverso di *onResume*) notifica la cessata interazione dell'utente con l'*activity*;
- *onStop* (contraltare di *onStart*) segna la fine della visibilità dell'*activity*;
- *onDestroy* (contrapposto a *onCreate*) segna la distruzione dell'*activity*.

Si consideri che i metodi di *callback* sono concepiti a coppie (un metodo di avvio con un metodo di arresto: *onCreate–onDestroy*, *onStart–onStop*, *onResume–onPause*) e solitamente il lavoro fatto nel metodo di avvio – in termini di funzionalità attivate e risorse allocate – verrà annullato nel corrispondente metodo di arresto. Una situazione che illustra l'invocazione dei più importanti metodi di *callback* è l'uso degli *intent* per passare da un'*activity* all'altra.

2. Android Studio

L'ambiente di sviluppo Android Studio è figlio di IntelliJ, un IDE molto intuitivo ed efficiente prodotto dalla società JetBrains. Questo ambiente di sviluppo, ottenibile dal sito ufficiale di Android, è stato pensato all'insegna della flessibilità e della praticità. È scaturito dalla stessa Google e nasce appositamente per Android, integrandosi con tutto il suo ecosistema. Permette di realizzare progetti per smartphone e tablet, nonché per dispositivi indossabili, Android Auto, Android TV e Android Things. Il colosso di Mountain View ha a disposizione un universo di servizi cloud, ed Android Studio offre a tutti i progetti un ponte per creare app che dialoghino con essi.

All'inizio di ogni nuovo progetto, l'IDE propone diversi template che rappresentano i tipi più in voga di applicazioni e la configurazione è affidata ai file *build* di Gradle. Quest'ultimo è uno strumento di *build automation* che permette una configurazione molto flessibile con una sintassi mutuata dal linguaggio Groovy. Un contributo utilissimo che offre Gradle è la gestione delle dipendenze in stile Maven. Quando si ha bisogno di integrare librerie di sviluppo prodotte da Android o da sviluppatori di terze parti è possibile inserire direttive che permetteranno di recuperarle direttamente in rete tramite "coordinate" costituite da *group id*, *artifact id* e versione.

Altre caratteristiche messe a disposizione da Android Studio sono:

- Editor per layout visuale usabile in modalità *drag and drop*;
- Supporto a ProGuard e preparazione del pacchetto di installazione;
- Accesso a SDK Manager per la personalizzazione dell'SDK scaricato e AVD (*Android Virtual Device*) Manager per la gestione degli emulatori;
- *Inline debugging*, funzionalità che rende più immediata l'ispezione del codice durante il debug, affiancando alle righe di linguaggio Java i valori ed i riferimenti collegati agli oggetti;
- Monitoraggio delle risorse di memoria e della CPU utilizzate dall'app, per seguire l'allocazione dinamica di oggetti ed effettuare *dump* della memoria *heap*, da analizzare successivamente.

Un aspetto importante è che il progetto è tuttora in continuo ampliamento. Ad esempio, dalla versione 1.3, si è avuta una ricongiunzione tra SDK (per lo sviluppo in Java) ed NDK (il *Native Developemente Kit*, pensato per sviluppatori C/C++) per il quale è stato aggiunto *editing* e *debugging* in Android Studio. Nella seconda major release, invece, si è puntato all'incremento della produttività e alla riduzione dei tempi di sviluppo con la nascita dell'*Instant Run* che mira a vedere in azione con grande rapidità le modifiche apportate al codice senza aspettare lunghi *rebuild* del progetto e al potenziamento degli emulatori che spesso –

soprattutto su macchine meno dotate – sono stati la nota dolente delle giornate lavorative dei programmatori.

3. Android Things

Android Things permette di sviluppare, prodotti per il mercato di massa su una piattaforma di fiducia, senza una precedente conoscenza di design di sistemi embedded. Riduce gli elevati costi di sviluppo e i rischi inerenti alla produzione di una idea. Quando si è pronti per la produzione di massa di una grande quantità di dispositivi, il costo scala linearmente e i costi per test ingegneristici si riducono grazie agli aggiornamenti forniti da Google. Al momento con le caratteristiche di Android Things è possibile:

- Sviluppare usando Android SDK e Android Studio
- Sviluppare un prodotto che non richiede il *power management*
- Sviluppare un prodotto che è connesso ad Internet via Wifi o Ethernet
- Usare produzioni hardware certificate
- Aggiungere display, fotocamere e interfacce audio compatibili con il *System-on-Chip* (SoM) e accedere a queste tramite il *framework* Android
- Integrare periferiche aggiuntive tramite le API relative all'input-output attraverso delle periferiche (GPIO, I2C, SPI, UART, PWM)
- Usare la *Console Android Things* per aggiornamenti più sicuri

Android Things fornisce una piattaforma hardware su cui è possibile sviluppare e programmare. Le board di sviluppo certificate basate sulla architettura SoM danno vari benefici per un veloce inizio:

- Parti Integrate: gli SoM integrano gli SoC (*System-on-Chip*), RAM, memoria flash, Wifi, Bluetooth e altri componenti su una singola board e viene proposta con tutte le certificazioni FCC. Quando si vuole produrre in massa il

dispositivo, è possibile ottimizzare il design della board comprimendo i moduli esistenti su un circuito stampato per ridurre costi e spazio.

- BSP Google: Il *Board Support Package* (BSP) è gestito da Google e questo comporta che non è necessario sviluppare un *kernel* o *firmware*. Questo garantisce una piattaforma di fiducia su cui programmare con aggiornamenti e riparazioni ricorrenti da parte di Google.
- Hardware differenziati: I partner di Android forniscono board di sviluppo con differenti SoM e impostazioni di fabbrica che si adattano ai bisogni, dando maggiore scelta e flessibilità. Quando si è pronti, è possibile trasformare i prototipi in prodotti modificandoli per farli aderire a delle impostazioni di fabbrica specifiche, tutto mentre si conserva lo stesso software.

Android Things estende il *framework* Android con API aggiuntive fornite dalla libreria di supporto per Things, che permettono di integrare i nuovi tipi di hardware non presenti sui dispositivi mobili. Sviluppare applicazioni per dispositivi *embedded* è diverso dal farlo per dispositivi mobili per vari motivi:

- Accesso più flessibile alle periferiche hardware e driver rispetto ai dispositivi mobili
- Le app di sistema non sono presenti per ottimizzare l'avvio e le richieste di memoria
- Le app sono mandate in esecuzione automaticamente all'avvio per immergere l'utente nell'esperienza dell'applicazione
- I dispositivi mostrano solo un'applicazione all'utente, invece di app multiple come sui dispositivi mobili

CAPITOLO II

Sviluppo di applicazioni Android per Internet of Things

SOMMARIO: 1. Descrizione applicazione sviluppata – 2. Descrizione componenti hardware – 2.1 Periferiche di input/output – 2.2 Connessione dei dispositivi – 3. Creazione del progetto in Android Studio – 4. Codice del programma – 4.1 Libreria del driver – 5. Esempi d'uso

1. Descrizione applicazione sviluppata

L'applicazione che si intende realizzare prevede l'accensione di un diodo LED tramite pressione di un pulsante. Il componente programmabile, in questo caso, è una Raspberry Pi Model B, uno dei dispositivi certificati e riconosciuti che supportano Android Things. La board, opportunamente collegata e programmata, recepisce la pressione del bottone, con una delle sue periferiche di input, per poi determinare l'accensione del led tramite una delle sue periferiche di output. La programmazione viene effettuata con il software Android Studio configurato per lo sviluppo di app per Things. In particolare, si mira a scrivere un codice che permetta la comunicazione di dati, sia in entrata che in uscita, dal dispositivo programmabile.

2. Descrizione componenti hardware

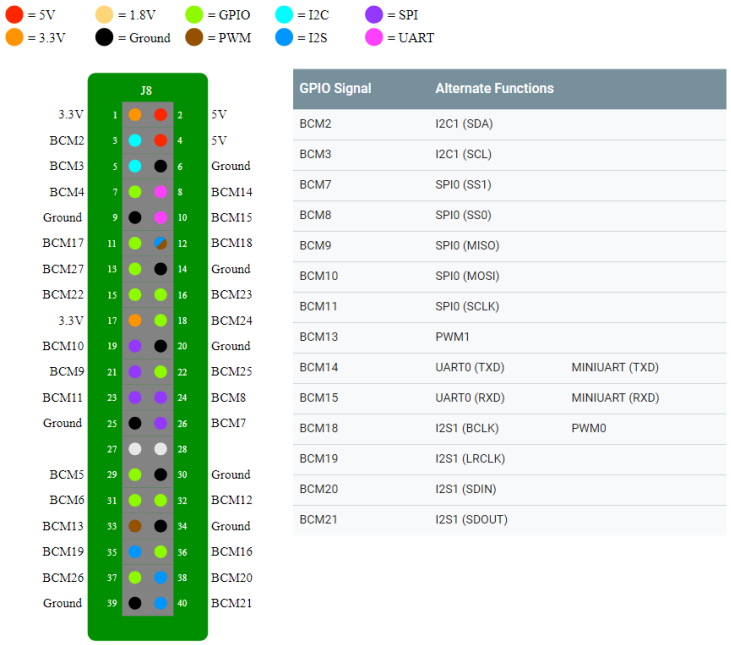
Il kit di progetto prevede come componenti hardware i seguenti dispositivi:

- Una breadboard
- Un pushbutton
- Due resistenze (con valori opportuni legati alle specifiche del LED e del bottone)
- Un LED
- Otto cavi jumper (quattro con connettori maschio-femmina, quattro con connettori maschio-maschio)
- Piattaforma di sviluppo per Android Things, in questo caso, Raspberry Pi Model B

La Raspberry Pi Model B è una delle più recenti versioni del più famoso computer a singola board del mondo. Possiede una CPU ARM Cortex-A53 quad-core a 64-bit funzionante a 1.2 GHz, quattro porte USB 2.0, rete wireless e non, video output con HDMI, 40 connettori pin GPIO per progetti con interfaccia fisica.

La Raspberry possiede pin che sono multiplexati tra le varie funzioni della board. Alcune di queste funzioni non possono essere usate simultaneamente, come attivare il Bluetooth e usare la porta periferica UART0.

Il seguente diagramma di piedinatura mostra le locazioni delle porte disponibili in base ai connettori visibili sulla board:



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

Figura 2 - Diagramma di piedinatura della Raspberry PI 3 Model B

2.1 Periferiche di input/output

Android Things fornisce API per le periferiche di input/output per comunicare con sensori e attuatori usando protocolli e interfacce standard a livello industriale.

- General Purpose Input/Output (GPIO) – Si usa quest'API per sensori semplici come rilevatori di movimento, rilevatori di prossimità e pulsanti che riportano il loro stato come un valore binario – alto o basso.
- Pulse Width Modulation (PWM) – Si usa quest'API per servo motori, motori in continua e luci che richiedono un segnale proporzionale per fornire un preciso controllo sull'uscita

- Comunicazione seriale – Si usano queste API per trasferire carichi più grandi di dati tra due o più dispositivi smart connessi sullo stesso bus locale. I protocolli seriali implementati sono I2C, SPI (entrambi sincroni) e UART (asincrono).

2.2 Connessione dei dispositivi

Prima di iniziare a scrivere il codice, è necessario collegare le periferiche dal kit di sviluppo alla board per ottenere il corretto funzionamento dell'applicazione:

1. Si collega un lato del bottone con il pin GPIO di input scelto, in questo caso il pin BCM21, e l'altro lato a massa.
2. Si collega lo stesso pin di input al morsetto dell'alimentazione a 3.3 V tramite un resistore di pull-up.
3. Si collega il pin GPIO di output scelto, in questo caso BCM6, ad un lato di un resistore serie.
4. Si collega l'altro lato del resistore all'anodo del LED.
5. Si collega il catodo del LED a massa.

È possibile vedere lo schema rappresentativo del circuito nella figura seguente:

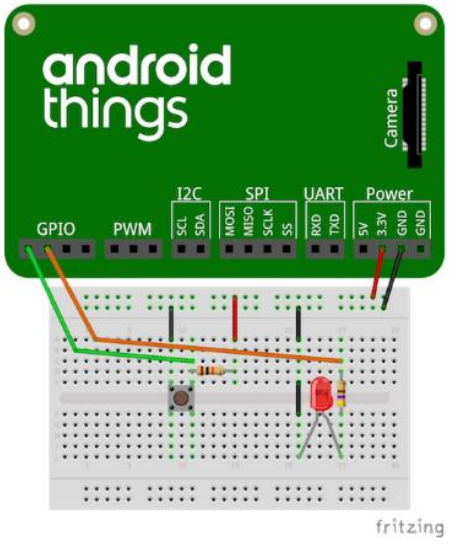


Figura 3 - Circuito rappresentativo del programma LED-Pushbutton

3. Creazione del progetto in Android Studio

Le app sviluppate per Things sfruttano lo stesso design usato per le app per cellulari e tablet. Questa similitudine comporta che è possibile modificare app esistenti per farle eseguire su sistemi embedded o svilupparne di nuove con conoscenze già possedute da chi programma in ambiente Android.

Android Studio rende facile la creazione di app per Android in diversi *form factors* come telefoni, tablet, TV, dispositivi indossabili (*Wear devices*). Ciò che interessa ora, è la nuova funzione introdotta nel 2017 nel software che permette la programmazione for Things. Infatti, il *project wizard* alla creazione di un nuovo progetto presenta la possibilità di sceglierlo come *form factor*. Successivamente si sceglie il livello di API opportuno, che, dato la recente introduzione in Android Studio della programmazione for Things, risulta uno dei più recenti, ovvero 27 o superiore. In seguito il processo per la creazione del progetto è il medesimo di quello per lo sviluppo di una qualsiasi altra app, con l'inserimento di una nuova *activity* e l'opportunità di inserire o meno un file di UI layout. Quest'ultimo aspetto è interessante perché si sottolinea come lo sviluppo di questo tipo di app, la maggior parte delle volte, è pensato per dispositivi manchevoli di uno schermo o di una qualsivoglia forma di output visiva. Da sottolineare che, per quanto riguarda la possibile scelta di un'*activity* predefinita, è introdotta l'*Android Things Peeripheral Activity* che prevede l'inserimento immediato dei metodi onCreate() e onDestroy() di cui si parlerà in seguito con l'introduzione del programma sviluppato.

Il progetto creato ha inserito al suo interno due oggetti fondamentali per lo sviluppo: una dipendenza alla libreria di supporto per Android Things e una *activity* di lancio di default.

I dispositivi per Android Things mostrano API tramite librerie di supporto che non sono nell'Android SDK. La dipendenza alla libreria di supporto è inserita automaticamente dal *project wizard* nel file `build.gradle`. Viene anche

aggiunto la stringa `<uses-library>` al manifest file per rendere questa libreria precompilata disponibile al classpath dell'applicazione durante l'esecuzione.

Un'applicazione pensata per essere eseguita su dispositivi embedded deve dichiarare un'activity nel *manifest file* come principale *entry point* dopo l'accensione dell'apparecchio. Il *project wizard* applica un *intent filter* contenente i seguenti attributi:

- Action: ACTION_MAIN
- Category: CATEGORY_DEFAULT
- Category: CATEGORY_HOME

Per rendere più facile lo sviluppo dell'applicazione, questa stessa *activity* include un CATEGORY_LAUNCHER *intent filter* cosicché Android Studio possa lanciarla come *activity* di default quando esegue il processo di *debug* o quando installa e mette in funzione. Quanto scritto è illustrato nel codice seguente:

```
<application>
    <uses-library android:name="com.google.android.things"/>
    <activity android:name=".HomeActivity">
        <!—Lancia l'activity come default per Android Studio -->
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category
android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>

        <!—Lancia l'activity automaticamente all'accensione e la
rilancia in caso di riavvio -->
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.HOME"/>
            <category
android:name="android.intent.category.DEFAULT"/>
        </intent-filter>
    </activity>
</application>
```

4. Codice del programma

Il codice del programma viene ora analizzato spiegando opportunamente tutti i metodi analizzati, tenendo presente che le classi implementate sono due: BoardDefaults e ButtonActivity (la principale).

La classe BoardDefaults è introdotta per riuscire a individuare automaticamente i nomi dei pin di input/output a cui sono connessi il bottone e il LED. Il codice è presentato di seguito:

```
package com.example.davide.bottoneled

import android.os.Build;

@SuppressWarnings("WeakerAccess")
public class BoardDefaults {
    private static final String DEVICE_RPI3 = "rpi3";
    private static final String DEVICE_IMX6UL_PICO = "imx6ul_pico";
    private static final String DEVICE_IMX7D_PICO = "imx7d_pico";

     // Restituisce il pin di GPIO a cui è connesso il LED.
    public static String getGPIOForLED() {
        switch (Build.DEVICE) {
            case DEVICE_RPI3:
                return "BCM6";
            case DEVICE_IMX6UL_PICO:
                return "GPIO4_IO22";
            case DEVICE_IMX7D_PICO:
                return "GPIO2_IO02";
            default:
                throw new IllegalStateException("Unknown Build.DEVICE
" + Build.DEVICE);
        }
    }
     //Restituisce il pin di GPIO a cui è connesso il bottone.
    public static String getGPIOForButton() {
        switch (Build.DEVICE) {
            case DEVICE_RPI3:
                return "BCM21";
            case DEVICE_IMX6UL_PICO:
                return "GPIO2_IO03";
            case DEVICE_IMX7D_PICO:
                return "GPIO6_IO14";
            default:
                throw new IllegalStateException("Unknown Build.DEVICE
" + Build.DEVICE);
        }
    }
}
```

Questa classe, una volta determinato il dispositivo con cui si sta programmando in ButtonActivity, restituisce sottoforma di stringhe i nomi dei pin opportuni con i metodi getGPIOForLED() e getGPIOForButton(). Entrambi i metodi presentano quindi banalmente uno switch-case che varia il risultato in base alla board selezionata.

La classe ButtonActivity presenta il programma vero e proprio. Si osservano prima di tutto le librerie importate:

```
import android.app.Activity;
import android.os.Bundle;

import com.google.android.things.contrib.driver.button.Button;
import
com.google.android.things.contrib.driver.button.ButtonInputDriver;
import com.google.android.things.pio.Gpio;
import com.google.android.things.pio.PeripheralManager;
import android.util.Log;
import android.view.KeyEvent;

import java.io.IOException;
```

Tra queste si sottolineano quelle relative ai driver del pulsante (com.google. android.things.contrib.driver.button.Button e com.google.android.things.contrib. driver.button.ButtonInputDriver) e quelle introdotte con Android Things per la comunicazione con le periferiche di cui si richiameranno successivamente alcuni metodi (com.google.android.things.pio.Gpio e com.google.android.things.pio. PeripheralManager).

All'inizio del codice, nell'activity principale, si aggiungono due oggetti, mLedGpio e mButtonInputDriver, rappresentativi nel programma dei rispettivi dispositivi fisici.

```
private Gpio mLedGpio;
private ButtonInputDriver mButtonInputDriver;
```

Il primo metodo inserito è onCreate(). Al suo interno di nota subito l'inizializzazione dell'oggetto pioService della classe PeripheralManager introdotta con una delle librerie sopramenzionate:

```
PeripheralManager pioService = PeripheralManager.getInstance();
```

Nello stesso metodo c'è ora bisogno di configurare i pin di input/output sia per il LED che per il pulsante. Per quanto riguarda il LED è necessario prima aprire la porta adeguata con cui è stato collegato e poi definirne la direzione, ovvero se si tratta di una porta di input o di output. In questo caso, viene impostato un pin di GPIO di output e viene inizializzato con uno stato falso, basso:

```
try {
    Log.i(TAG, "Configuring GPIO pins");
    //Apertura del pin di GPIO
    mLedGpio = pioService.openGpio(BoardDefaults.getGPIOForLED());
    //Impostazione della direzione del pin e inizializzazione dello
stato
    mLedGpio.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
} catch (IOException e) {
    Log.e(TAG, "Error configuring GPIO pins", e);
}
```

Si noti come il pin scelto sia ottenuto richiamando la classe BoardDefaults permettendo il riutilizzo del programma anche nel caso in cui si cambiasse board di sviluppo.

Sempre nel metodo onCreate(), si continua registrando il pulsante la cui pressione determina l'accensione del LED:

```
try {
    Log.i(TAG, "Registering button driver " +
BoardDefaults.getGPIOForButton());
    // Inizializza e registra l'InputDriver che emetterà SPACE key
events
    // ad un cambio di stato del GPIO
    mButtonInputDriver = new ButtonInputDriver(
            BoardDefaults.getGPIOForButton(),
            Button.LogicState.PRESSED_WHEN_LOW,
            KeyEvent.KEYCODE_SPACE);
    mButtonInputDriver.register();

} catch (IOException e) {
    Log.e(TAG, "Error configuring GPIO pins", e);
}
```

Si noti che si inizializza il pulsante definendo il pin a cui esso è collegato, in maniera analoga a quanto fatto per il LED, lo stato logico e il tipo di KeyEvent.

Viene infine richiamato il metodo register() proprio della libreria del driver importata ad inizio codice.

Prima di passare ai metodi atti al funzionamento coordinato dei dispositivi, si osservi il metodo onStop():

```
@Override
protected void onStop(){
    Log.d(TAG, "onStop called.");
    if (mButtonInputDriver != null) {
        mButtonInputDriver.unregister();
        try {
            Log.d(TAG, "Unregistering button");
            mButtonInputDriver.close();
        } catch (IOException e) {
            Log.e(TAG, "Error closing Button driver", e);
        } finally{
            mButtonInputDriver = null;
        }
    }

    if (mLedGpio != null) {
        try {
            Log.d(TAG, "Unregistering LED.");
            mLedGpio.close();
        } catch (IOException e) {
            Log.e(TAG, "Error closing LED GPIO", e);
        } finally{
            mLedGpio = null;
        }
    }
    super.onStop();
}
```

In questa funzione si fa in modo che sia il pulsante che il LED vengano rilasciati quando l'activity viene fermata.

Bisogna quindi implementare dei metodi per rispondere agli eventi determinati dal pulsante:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_SPACE) {
        // Accende il LED
        setLedValue(true);
        return true;
    }

    return super.onKeyDown(keyCode, event);
}
```

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_SPACE) {
        // Spegne il LED
        setLedValue(false);
        return true;
    }

    return super.onKeyUp(keyCode, event);
}
```

onKeyDown() e onKeyUp() definiscono il valore che il LED deve assumere in relazione allo stato del pulsante. In onKeyDown() si determina il valore *true* per il LED, provocandone l'accensione, quando il pushbutton è premuto. In opposizione in onKeyUp() si determina il valore *false* per il LED, provocandone lo spegnimento, quando il pulsante è rilasciato. Questo è reso possibile dall'ultimo metodo inserito nel codice, setLedValue(), dove viene impostato il valore da assegnare al pin di output collegato con il diodo:

```
private void setLedValue(boolean value) {
    try {
        mLedGpio.setValue(value);
    } catch (IOException e) {
        Log.e(TAG, "Error updating GPIO value", e);
    }
}
```

4.1 Libreria del driver

Un elemento chiave nella stesura del programma è stato il driver del pulsante, richiamato con una libreria all'inizio del programma. Questa libreria e molte altre relative ai più disparati dispositivi nel mondo dell'elettronica sono presenti nell'*Android Things driver library* (libreria di tipo *open source*). Questi driver astraggono le caratteristiche della comunicazione a basso livello con molte periferiche hardware comuni.

Per importare una libreria in un'applicazione bisogna seguire semplici passaggi:

- Cercare il driver corrispondente alla periferica scelta

- Aggiungere la dipendenza del driver al file `build.gradle`:

```
dependencies {
    ...
    compile 'com.google.android.things.contrib:driver-
button:1.0'
}
```

- Aggiungere i permessi richiesti per il driver nel *manifest file*:

```
<uses-permission
android:name="com.google.android.things.permission.MANAGE_INP
UT_DRIVERS" />
```

- Inizializzare la classe del driver con le appropriate risorse delle periferiche di input/output
- Chiudere la connessione quando l'app non ha più bisogno della risorsa

5. Esempi d'uso

Sebbene l'applicazione possa risultare banale, questa è il punto di partenza per lo sviluppo di app ben più complesse. Essenzialmente tutto questo si può riportare all'accensione di una lampadina, o di un qualsiasi altro dispositivo che si voglia attivare manualmente, ma non bisogna escludere la possibilità, arricchendo anche in minima parte il codice, di ottenere dei programmi più adatti alle situazioni della vita quotidiana. Si potrebbero, ad esempio, aggiungere più pulsanti e più LED di colori differenti per poter rappresentare stati diversi segnalati da colori diversi. Un altro possibile esempio d'uso potrebbe essere quello di utilizzare questo programma per sviluppare un gioco (l'accensione del LED risulta casuale e bisogna premere il pulsante più velocemente possibile oppure premere ripetutamente il pulsante fino ad uno spegnimento casuale del LED per monitorare la frequenza di pressione).

Conclusioni

Conclusioni

Si vuole, in questa parte conclusiva, tirare le somme sulla metodologia di sviluppo osservata, analizzandola sotto diversi punti di vista, ovvero quello relativo alla programmazione e alla parte hardware. In particolare, si intendono sottolineare eventuali vantaggi o limiti, anche facendo confronti con altre metodologie studiate.

Per quanto riguarda la programmazione, coloro i quali erano già avvezzi allo sviluppo di app in ambiente Android, non incontreranno difficoltà alcuna nello sviluppo di app per Things. Il software Android Studio, infatti, rende particolarmente facile adattarsi a quelle piccole differenze che potrebbero incorrere nello sviluppo dell'app. Esso presenta inoltre l'utilissima possibilità, soprattutto per applicazioni embedded rivolte alla comunicazione con altri dispositivi, di programmare app per Things e per smartphone (o anche altri *form factor* eventualmente) nello stesso progetto. Altre metodologie di sviluppo prevedono invece due programmazioni diverse e separate per applicazioni che devono trasmettere informazioni e dati fra loro. O meglio, i sistemi operativi non coincidenti fra dispositivi posso determinare una netta separazione nello sviluppo delle due applicazioni, cosa che non accade con Android. I veri limiti, se presenti, non sono tanto da attribuire allo sviluppo per Things, quanto al software Android Studio e allo sviluppo in generale delle app in ambiente Android. Un altro svantaggio, ma facilmente superabile nel tempo, è la scarsa quantità di librerie relative agli attuatori e sensori da collegare al dispositivo programmabile (per quanto Google abbia introdotto un sito, *Android Things driver library*, dove gli stessi rivenditori possono inserire librerie relative al loro prodotto)

Per quel che concerne la parte hardware, bisogna rimarcare alcuni aspetti non del tutto positivi di Android Things. I requisiti di sistema per i dispositivi che supportano Android Things sono abbastanza alti quando comparati con quelli relativi a dispositivi basati su microcontrollori. Ovviamente ci sono dei vantaggi nell'avere un maggiore calcolo computazionale, ad esempio questo è necessario

per il riconoscimento facciale e vocale, elementi disponibili già in Android. Tuttavia, se questi dispositivi vanno ad essere integrati con il *cloud*, cosa che accade con ogni probabilità dato che sono sviluppati per IoT, allora non c'è una grande necessità di potenza di calcolo nella parte di sistema relativa all'*user interface*. Inoltre, i dispositivi che attualmente supportano Android Things sono relativamente pochi, addirittura solo due per la prototipazione (anche se uno di questi è la Raspberry Pi, dispositivo affidabile e facilmente reperibile).

In definitiva, la metodologia di sviluppo analizzata è consigliata per coloro i quali possiedono già un certo background di programmazione e di conoscenze relative ai sistemi embedded. In particolare, è suggerita per gli sviluppatori di applicazioni decisamente dispendiose, da un punto di vista computazionale, e per quelli che si vogliono affidare alla sicurezza dei servizi che Google offre ormai da anni e che continua ad aggiornare.