**Android Go**

**E**

**Build for Billions**

Report del progetto di

Sistemi Embedded

Università degli studi di Padova

Docente: Carlo Fantozzi

A cura di : Albertini Pietro, Varotto Francesco, Zanocco Giovanni

Indice

[1. ANDROID GO 3](#_Toc42621949)

[1.1. INTRODUZIONE 3](#_Toc42621950)

[1.2. SPECIFICHE 3](#_Toc42621951)

[1.3. OTTIMIZZAZIONE PRESTAZIONI 4](#_Toc42621952)

[1.3.1. UN ESEMPIO: MAPS VS MAPS GO 4](#_Toc42621953)

[1.4. CONCLUSIONI 5](#_Toc42621954)

[1.5. VERSO BUILD FOR BILLIONS 6](#_Toc42621955)

[2. BUILD FOR BILLIONS 6](#_Toc42621956)

[2.1. CONNECTIVITY2.1.1 IMAGES 7](#_Toc42621957)

[2.1.2. NETWORKING 7](#_Toc42621958)

[2.2. DEVICE CAPABILITY 9](#_Toc42621959)

[2.2.2. GRAPHICS AND SCREEN SIZE 9](#_Toc42621960)

[2.2.3. RETROCOMPATIBILITà 10](#_Toc42621961)

[2.2.4. UTILIZZO DELLA MEMORIA 11](#_Toc42621962)

[2.3. CONSUMO DI DATI 11](#_Toc42621963)

[2.4. USO DELLA BATTERIA 13](#_Toc42621964)

[2.5. INTERFACCIA UTENTE 13](#_Toc42621965)

[2.6. PROFILER 13](#_Toc42621966)

[2.7. PERFORMANCE TIPS 15](#_Toc42621967)

[3. SITOGRAFIA 17](#_Toc42621968)

# ANDROID GO

## INTRODUZIONE

Android Go fu inizialmente rilasciato il 5 dicembre 2017 in concomitanza con la versione di Android 8.1 con il nome di Oreo (go edition).

Voleva presentarsi come una distribuzione per Android destinata ai mercati emergenti, in particolare quello cinese e indiano.

Le specifiche base su cui si fonda Android go, sono molto basse, come ad esempio la RAM minore di 1-2Gb oppure il limitato spazio di storage (solitamente con un tetto massimo di 8Gb), dato che questi paesi non hanno ancora una ricchezza tale da permettere di comprare i modelli top di gamma.

Un altro aspetto fondamentale è la ricerca di risparmio nel campo della batteria, ma soprattutto dell’utilizzo internet, dato che in posti come l’India, si ha ancora una copertura internet limitata in certi luoghi.

## SPECIFICHE

Come accennato precedentemente Android Go *deve* basarsi su specifiche molto basse, e per questo motivo è un sistema operativi con grossissime ottimizzazioni. Ottimizzazioni sia a livello di Sistema Operativo, sia di applicazioni implementate di default.

[Perciò pure gli sviluppatori che mirano ai mercati orientali, o in generale, mirano al maggior pubblico possibile, devono prendere in considerazione questo fatto.]

Partiamo dalla prima cosa che un utente qualsiasi nota avviando il proprio smartphone: il launcher.

Google promette che l’avvio del cellulare e delle proprie applicazioni si avviano con un 15% di velocità in più rispetto ad uno smartphone con le stesse specifiche ma con sistema operativo Android “standard”.

Per di più Android go permette un grosso risparmio per quanto riguarda lo spazio in memoria, sia RAM che per lo storage.

Infatti questo sistema operativo è talmente ottimizzato da far risparmiare fino a 270Mb di memoria RAM. Questo vuol dire che l’utente ha la possibilità di avviare e lasciare in background più applicazioni contemporaneamente, senza sovraccaricare eccessivamente lo smartphone. Ovviamente non ci si può aspettare la stessa prontezza e reattività di un cellulare top di gamma, però per un dispositivo che nei mercati indiani può costare anche 30$ ciò è degno di nota.

Per quanto riguardo lo spazio disponibile nello storage, invece, Android garantisce fino a ben 900Mb si spazio che su un cellulare dalle basse specifiche vale a dire anche il 5-10% di spazio disponibile in più, col quale l’utente può decidere di scaricare nuove applicazioni, immagazzinare documenti di lavoro, foto di famiglia, video di amici.

Parlando di internet Google mette a disposizione dell’utente una modalità “data saver” che consente di limitare l’utilizzo di internet.

## OTTIMIZZAZIONE PRESTAZIONI

Ma come si fa ad ottenere tale risparmi di spazio e tali miglioramenti di prestazioni in telefono di basse specifiche?

Innanzitutto Google si è impegnata a sviluppare versioni “Go” delle applicazioni più usate, come possono essere

* Youtube Go
* Google Go
* Gallery Go
* Maps Go
* Etc…

Che sono molto più leggere delle loro versioni standard. Più tardi analizzeremo l’esempio di Google Maps Go, per capire dove avvenga il risparmio di risorse.

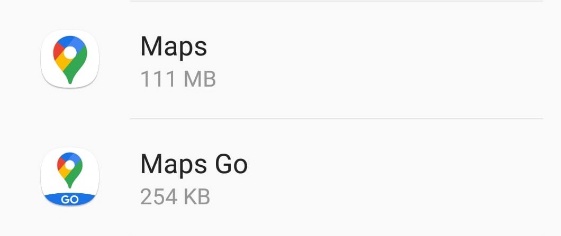
In secondo luogo, Android go fornisce una serie di impostazioni che ci permettono di monitorare meglio lo stato del telefono (spazio disponibile, stato batteria) e ci dà strumenti per controllare gli aspetti che più pesao nelle performance del cellulare, come il “data saver” accennato brevemente nel Capitolo 1.2.

Per capire le maggiori differenze tra queste due versioni prendiamo in esame MAPS e MAPS GO.

### UN ESEMPIO: MAPS VS MAPS GO

In questi paragrafi vedremo un esempio di cosa varia dalla versione standard alla versione go di una tra le applicazioni per smartphone più utilizzate: Maps e Maps Go.

#### DIMENSIONI

Se andiamo a vedere la memoria occupara da queste due applicazioni notiamo subito una differenza abissale.

1- Confronto dimensioni

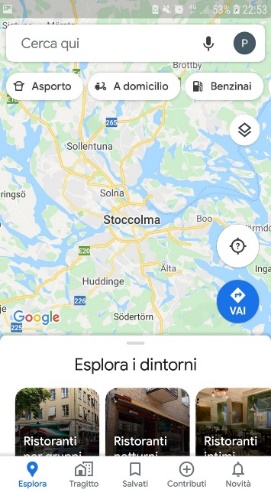
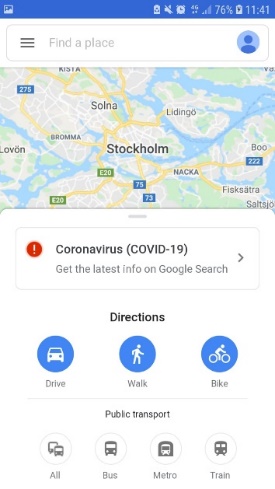
Maps “standard” pesa più di 400 volte la sua rivale Maps go!

Ma come può essere che due applicazioni sviluppate da una stessa azienda e che hanno *circa* le stesse funzioni, abbiano questa differenza di dimensione?

Ovviamente ci sono aspetti grafici e diversità di funzioni che nella versione Go non vengono implementate. Andiamo ora ad analizzare i principali.

#### INTERFACCIA

Parliamo ora delle differenze nell’interfaccia grafica.

Nalla Figura 2 si può vedere l’UI(user-interface) della varsione standard dell’applicazione di Maps, mentre nella Figura 3 si può vedere quella della verione Go.

Le prima differenza che saltano all’occhio è che laschermata iniziale di Maps Go è molto più scarna della sorella versione standard.

Infatti la Navigation Bar (la barra di navigazione in basso nello schermo) della versione standard nella versione Go non appare, l’effetto traslucido della barra in alto, non ci sono le shortcut sotto la barra di ricerce che permettono di cercare “cose utili” mentre si è in viaggio, e altro.

Figura 3-Maps Go Home

Figura 2-Maps Home

Altra differenza, un po’ più “nascosta”, ma molto importante è la possibilità di fare ricerche a voce; feature presente solo nella versione standard.

#### NAVIGAZIONE

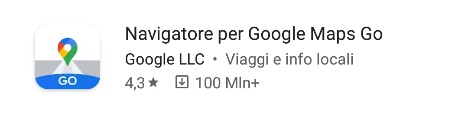
**Altra differenza veramente importante, è che la verisione Go NON permette la navigazione. Nel senso che se Google Maps, al suo interno ha incorporati dei provider per la navigazione, Go ne è sprovvisto. C’è comunque la possibilità di navigare nella versione standard di Maps… in che modo? C’è la necessita di scaricarsi dal Play Store un’app apposita chiamata (nello store italiano): *Navigatore per Google Maps Go* [Figura 4]*.*

Figura 2-Navigatore per Maps Go

#### INFORMAZIONI SUI LUOGHI

Quando andiamo in cerca di un’attività in particolare, come può essere un ristorante, bar, sala giochi, Google Maps ci fornisce tutte le informazioni necessarie per capire se “ha senso” andare in quel determinato posto o preferirne un altro. Le informazioni più comuni che si trovano solo:

* La valutazione
* Gli orari di apertura/chiusura
* I livelli di prezzo
* L’indirizzo specifico
* Etc…

Tutte queste informazioni si possono trovare anche nella versione Go.

## CONCLUSIONI

Abbiamo visto dove, principalmente, un’applicazione in versione Go risparmia spazio e dove no. Sostanzialmente tutti le feature fondamentali le implementa, mentre tutti gli extra devono essere scaricati a parte. Una cosa che nei punti precedenti non è stata detta, ma molto importante, è che Maps Go per funzionare si “appoggia” Google Chrome. Questo significa che se cerchiamo di avviare Maps Go in un dispositivo dove Google Chrome è disabilitato, Maps Go non funzionerebbe, cosa che non accade con la versione standard di Maps che, al suo interno, possiede già tutto il necessario per funzionare.

Detto ciò si può vedere come Android go non vada a risparmiare dove non si può: e cioè nel reparto sicurezza.

Google, infatti, utilizza diversi strumenti per aiutare l’utente medio a stare al sicuro da persone malintezionate con strumenti come Google Play Protect, che è un anti-malware che migliora in tempo reale grazie ad algoritmi di machine learning. Questa feature è sempre in funzione, analizza il dispositivo in maniera automatica e cerca di risolvere tutte le problematiche riguardanti la sicurezza.

## VERSO BUILD FOR BILLIONS

Ora che abbiamo visto cos’è Android go e capito come mai sia necessario sviluppare un’app dai prerequisiti bassi per avere il maggior pubblico possibile, andiamo a vedere quali sono gli accorgimenti per sviluppare *concretamente* un’applicazione del genere.

Un’applicazione sviluppata per una grandissima audience viene soprannominata: *build for billions*, che letteralmente di traduce: “sviluppata per miliardi”, in riferimento al numero di abitanti dei paesi in via di sviluppo quali India e Cina che assieme hanno circa della popolazione mondiale.

# BUILD FOR BILLIONS

Sviluppare un’applicazione Android rivolta ai mercati emergenti significa necessariamente dover fare i conti con una serie di limiti tecnici. In queste aree, come accennato poco fa, la maggioranza degli utenti utilizza dispositivi con schermi a bassa risoluzione, memorie meno capienti e processori meno performanti rispetto a quelli presenti in altri mercati, mentre l’accesso a Internet è possibile solo attraverso connessioni lente e costose.  
Per permettere agli sviluppatori di affrontare queste e altre problematiche, Android Developers ha messo a disposizione all’interno della propria documentazione una serie di linee guida raccolte sotto il nome di “Build for billions guidelines” e suddivise in cinque macro-categorie, di seguito brevemente illustrate.

1. *Connectivity*Si preoccupa di ottimizzare l’utilizzo della rete internet al fine di garantire un’esperienza d’uso fluida anche a quegli utenti che non dispongono di una connessione veloce e affidabile.
2. *Device Capability*Contiene svariati suggerimenti su come sviluppare un’applicazione che funzioni in maniera ottimale su una varietà sempre maggiore di versioni di Android e su dispositivi dalle caratteristiche diverse.
3. *Data cost*

Analizza vari aspetti riguardanti la dimensione degli APK, la lunghezza del codice e la possibilità di permettere agli utenti di modificare alcuni parametri dell’applicazione.

1. *Battery consumption*

Fornisce alcune indicazioni su come ottimizzare il consumo della batteria da parte dell’applicazione.

1. *User interface and content*Si focalizza sulla progettazione di un’interfaccia utente reattiva che renda il più possibile impercettibili i limiti imposti dall’hardware.

## CONNECTIVITY2.1.1 IMAGES

Di seguito vengono descritte tre modalità con cui è possibile agevolare il download di immagini dalla rete.

1. WebP è un formato di compressione per immagini sviluppato da Google che permette di ridurre i tempi di caricamento e risparmiare larghezza di banda.  
   A parità di qualità dell’immagine, un file WebP risulta essere più leggero rispetto alle controparti PNG e JPG, inoltre la sua codifica con perdita presenta una definizione pressoché identica all’originale.  
   Questo formato è supportato nella forma “lossy” nei dispositivi Android a partire da Android 4.0 (API level 14: Ice Cream Sandwich) mentre nella forma “lossless” a partire da Android 4.2 (API level 17: Jelly Bean).
2. È possibile minimizzare la quantità di dati scambiati attraverso la rete semplicemente richiedendo che le immagini da scaricare abbiano una dimensione di rendering specifica per il tipo di connessione in uso e per le caratteristiche del dispositivo.
3. *Glide* e *Picasso* sono due librerie specializzate nel caricamento e nell’archiviazione di immagini remote. Esse permettono di ridurre sia i tempi di caricamento delle immagini sia l’utilizzo di memoria da parte dell’applicazione. Tra le loro funzioni principali vi sono: un meccanismo intelligente di riutilizzo delle risorse, la possibilità di poter configurare a proprio piacimento la ricerca in cache di un’immagine e il supporto a vari tipi di *placeholder*.

## NETWORKING

Una gestione ottimale delle risorse passa anche e soprattutto attraverso un uso accurato della rete Internet.  
Su questo principio si basano le seguenti indicazioni:

1. L’utente deve essere in grado di interagire in maniera utile con l’applicazione anche in assenza di un collegamento a Internet. Questo può essere reso possibile attraverso una serie di accorgimenti tecnici, quali ad esempio: l’archiviazione dei dati in memoria locale, il *caching*, e la gestione di una coda di richieste da processare nel momento in cui la connessione risulti nuovamente disponibile.  
   Inoltre, per rendere queste dinamiche quanto più possibile trasparenti all’utente, l’applicazione potrebbe notificare la mancanza di una connessione solamente quando le venga richiesto di comunicare con la rete, oppure potrebbe permettere all’utente di compiere qualsiasi tipo di operazione, comprese quelle che necessitano di un collegamento a Internet, rimandando però l’esecuzione di quest’ultime a un secondo momento.
2. *Room*

*Room* è una libreria ORM (Object-relational mapping)fornita da Android che permette il salvataggio di dati all’interno di un database locale. In sintesi, essa consente a un’applicazione di rispondere alle richieste dell’utente limitando la quantità di informazioni che è necessario reperire ex novo dalla rete.

1. *DiskLruCache*Si tratta di un’implementazione in Java di una *Disk-cache* di tipo LRU (**Least Frequently Used**). Ogni entrypresenta una chiave di tipo stringa, corrispondente alla *regex* **[a-z0-9\_-]{1,64},** e un numero fissato di valori, composti da sequenze di byte di lunghezza compresa tra 0 e *Integer.MAX\_VALUE*.  
   I metodi principali per la gestione della cache sono:

edit(String key)Permette di creare o aggiornare i valori dell’entry avente *key* come chiave. Restituisce un oggetto di tipo [*DiskLruCache.Editor*](https://jakewharton.github.io/DiskLruCache/com/jakewharton/disklrucache/DiskLruCache.Editor.html), oppure *null* se è in corso un’altra modifica sulla stessa entry.

get(String key)Ritorna un oggetto di tipo [*DiskLruCache.Snapshot*](https://jakewharton.github.io/DiskLruCache/com/jakewharton/disklrucache/DiskLruCache.Snapshot.html), il quale rappresenta, in un determinato istante, lo stato dell’entry avente *key* come chiave. Viene restituito null nel caso in cui l’entry non esista o non sia leggibile in quel momento.

1. Un’applicazione che implementi una coda di priorità permette di servire le richieste più urgenti con maggiore rapidità. Questo meccanismo può essere utile soprattutto nei casi in cui il dispositivo non disponga di una connessione stabile, in quanto consente di visualizzare per prime le informazioni ritenute importanti, evitando un allungamento dei tempi di attesa dovuto al caricamento di risorse non essenziali.  
   Ad esempio, nel caso in cui una risorsa web includa elementi testuali misti a file multimediali, quest’ultimi dovrebbero avere una priorità di caricamento più bassa.
2. È possibile monitorare lo stato della connessione di un dispositivo invocando i seguenti metodi:

public boolean isActiveNetworkMetered()

Classe di appartenenza: ConnectivityManager

Richiede: Manifest.permission.ACCESS\_NETWORK\_STATE

Restituisce true nel caso in cui la connessione in uso sia classificata come “metered”, ovvero sensibile al trasferimento di grandi quantità di dati.

public NetworkInfo getActiveNetworkInfo() (deprecato a partire da API level 29)Classe di appartenenza: ConnectivityManager  
Richiede: Manifest.permission.ACCESS\_NETWORK\_STATE  
Restituisce un oggetto di tip NetworkInfo contenente informazioni sulla connessione in uso, se questa è attiva, oppure null se nessuna tra le connessioni di default è disponibile per il trasferimento dati.

public NetworkCapabilities getNetworkCapabilities(Network network)Classe di appartenenza: ConnectivityManager  
Richiede: Manifest.permission.ACCESS\_NETWORK\_STATE  
Resituisce un oggetto di tipo NetworkCapabilities contenente informazioni sulla qualità e sul tipo della connessione fornita in ingresso come parametro, oppure null se la connessione è sconosciuta.

public void registerDefaultNetworkCallback ([ConnectivityManager.NetworkCallback](https://developer.android.com/reference/android/net/ConnectivityManager.NetworkCallback) networkCallback)  
Classe di appartenenza: ConnectivityManager  
Richiede: Manifest.permission.ACCESS\_NETWORK\_STATE  
Permette a un’applicazione di registrarsi per ricevere una notifica ogni volta che il sistema rileva un cambiamentio nello stato della rete. Esistono cinque diverse varianti di questo metodo, ciascuna con un’intestazione diversa per numero e tipo dei parametri d’ingresso.

public int getNetworkType() (deprecato a partire da API level R)Classe di appartenenza: TelephonyManager  
Richiede: Manifest.permission.READ\_PHONE\_STATE  
Resituisce, sottoforma di costante, il tipo di connessione in uso.

public int getDataState()Classe di appartenenza: TelephonyManager  
Resituisce, sottoforma di costante, lo stato della rete cellulare (es. DATA\_DISCONNECTED, DATA\_CONNECTING, DATA\_CONNECTED, DATA\_SUSPENDED, DATA\_DISCONNECTING).

1. Nei casi in cui la qualità della connessione risulti particolarmente scarsa, può essere utile impedire a un’applicazione di scaricare contenuti multimediali, o in alternativa consentire il download soltanto dei file a risoluzione più bassa.  
   È inoltre consigliato l’utilizzo di *placeholder* (ad esempio sfruttando le funzionalità offerte dal package ”androidx.palette.graphics”) in modo da permettere all’utente di godere di un’esperienza visiva uniforme anche in mancanza di una connessione stabile.  
   Al contrario, se un dispositivo dispone di una connessione veloce e affidabile, è possibile fruire più rapidamente dei contenuti effettuando il *prefetch*, ovvero archiviando preventivamente in cache i file a cui l’utente potrebbe voler accedere nell’immediato futuro.

## DEVICE CAPABILITY

## GRAPHICS AND SCREEN SIZE

In questa sezione vengono discusse alcune tecniche che è possibile adottare per consentire a un’applicazione di supportare correttamente dispositivi con schermi dalle dimensioni e risoluzioni diverse.

1. Il *density-independent pixel* (dp) è unità di misura alternativa al *pixel* (px) che permette di visualizzare dei contenuti su uno schermo in maniera indipendente dalla dpi (“dots per inch”, ovvero il numero di punti, o pixel, presenti in un pollice di lunghezza).  
   L’uso dei dp offre il grande vantaggio di non doversi preoccupare di come la propria applicazione verrà visualizzata sui vari dispositivi, in quanto permette di “scalare” gli elementi dell’interfaccia in base alla risoluzione e alla densità di pixel di ogni singolo schermo.
2. È importante, soprattutto nello sviluppo di un’applicazione rivolta ai mercati emergenti, assicurarsi che gli elementi grafici vengano visualizzati correttamente anche su device dalle specifiche hardware limitate. In particolare, per fare in modo che la propria app funzioni in maniera ottimale anche su tali dispositivi, è consigliabile ridurre, se non addirittura eliminare, la presenza di elaborazioni grafiche particolarmente pesanti, come animazioni e transizioni.
3. È possibile che su dispositivi con schermi dalle dimensioni ridotte non sia possibile visualizzare tutti gli elementi previsti per una singola activity. Per questo motivo è necessario che ogni applicazione sia in grado di modificare la propria interfaccia rendendo visibili solamente gli elementi strettamente necessari al suo utilizzo.  
   Per facilitare il compito degli sviluppatori, Google ha messo a disposizione una serie di linee guida, raccolte sotto il nome di “Material Design guidelines”, in cui vengono illustrati alcuni principi utili alla progettazione di layout per l’interfaccia utente.

## RETROCOMPATIBILITà

Di seguito vengono illustrate tre diverse modalità attraverso le quali è possibile incrementare il grado di retrocompatibilità di un’applicazione.

1. L’attributo *android:targetSdkVersion* presente nel file *AndroidManifest.xml* informa il sistema che una particolare app è stata testata utilizzando una certa versione di Android. In caso il valore di questo attributo fosse più basso dell’API level utilizzato dal dispositivo, il sistema operativo potrebbe attivare dei meccanismi di compatibilità per assicurarsi che l’applicazione continui a funzionare come previsto. Per questo motivo, se si vuole mantenere aggiornata la propria app, è importante che la targetSdkVersion corrisponda all’API level più recente.  
   Analogamente, l’attributo *android:minSdkVersion* indica l’API level più basso supportato da un’applicazione. Diminuire il valore di questo attributo permette di aumentare il numero di dispositivi su cui essa è in grado di funzionare correttamente, ma allo stesso tempo costringe gli sviluppatori a utilizzare un set di funzionalità limitato rispetto a quello offerto da versioni di Android SDK più recenti (se la minSdkVersion non viene esplicitamente indicata dallo sviluppatore, il sistema imposta automaticamente il suo valore a 1, il che significa che quella particolare applicazione è, almento teoricamente, retrocompatibile con qualsiasi versione di Android).
2. È importante che l’applicazione funzioni correttamente se usata in varie versioni del sistema operativo Android. Questo può essere fatto grazie all’utilizzo delle *Librerie di Supporto Android*, implentando *android.support.\** che dalle verisioni API successive alla 27 sono state deprecate e sono state sostituite con *androidx*. Il package *Librerie di Supporto Android* contiene diverse librerie che possono essere incluse nelle propria applicazione, ognuna di esse riguardanti uno specifico range di versioni Android, e uno specifico insieme di feature; vediamo qui di seguito una breve lista:

* Librerie di Supporto v4 e v7: sono le librerie di supporto di dimensioni maggiori. Al loro interno si possono trovare funzionalità di supporto per i componenti delle app, feature dell’UI, accessibilità e molto altro.
* Material Design: qua si trovano una gran quantità di componenti e modelli grafici per le applicazioni.
* Multidex: questa libreria provvede supporto per applicazioni con più file DEX .

1. Anche i Google Play Services aiutano nella funzione di retrocompatibilità.

Questi services includono anche classi come *GcmNetworkManager* che serve ad avere un forte meccanismo di sincronizzazione con dei server nei quali sono disponibili nuove informazioni.

Questo metodo comunque verrà deprecato il 1/11/2020, ed è quindi *consigliato* migrare a *Jetpack WorkManager*. Per di più nel Google Play Store gli aggiornamenti di Google Play Services sono automatici e quindi attraverso l’ Android SDK Manager, vengono consegnate anche *librerie client* che aiutano lo sviluppatore a programmare.

## UTILIZZO DELLA MEMORIA

In questo paragrafo vedremo come gestire la memoria in maniera efficiente, in modo tale da rendere più gradevole l’utilizzo del dispositivo mobile, rendendo più fluida l’esperienza di utilizzo.

1. Il primo consiglio è di gestire dinamicamente la memoria. Questo assicura una maggiore stabilità quando l’app si passa a dispositivi con dimensioni RAM anche parecchio diverse.

Per di più esistono metodi che ci aiutano a capire quanto il dispositivo sia “stressato” a runtime. In questo modo noi possiamo capire meglio quali azioni possiamo compiere senza che l’utente ne risenta troppo e quali magari evitare.

Uno di questi metodi è senza dubbio *isLowRamDevice()* che ritorna un booleano. Se il valore ritornato è *true* allora abbiamo tra le mani un device con poca RAM, quindi sapremo che non potremo far partire troppi processi in contemporanea, altrimenti l’esperienza dell’utente potrebbe risentirne.

Un altro metodo che possiamo utilizzare è *getMemoryInfo()* che provvede informazioni circa lo stato della memoria, includendo la memoria disponibile, la memoria totale e il livello di memoria dove il sistema comincia ad uccidere i processi. Analizzando questi valori si può capire se la nostra applicazione risulterà essere troppo pesante.

1. Molto importante, inoltre, è evitare di far eseguire processi dalla lunga durata, che potrebbero rallentare il device. Per evitare questi lunghi processi in background si può utilizzare, ad esempio, il già accennato *GcmNetworkManager.*
2. Per controllare l’uso della memoria Android Studio provvede lo sviluppatore di alcuni accessori molti utili, tra cui il *Profiler*, che mostra il consumo di memoria a run-time. In realtà il Profiler (che riprenderemo più tardi), è molto utile anche per controllare i consumi di batteria e internet.

## CONSUMO DI DATI

Questo paragrafo è in parte collegato col paragrafo 2.1.2.. Andremo, infatti, a vedere quali sono i consigli che Android Deveper ci fornisce per creare un’app che sia agevolmente scaricabile al pubblico dei Build For Billions. In alcuni paesi in via di sviluppo, infatti, gli abbonamenti telefonici costano molti soldi, quindi ridurre la dimensione della propria applicazione vuol dire un minor consumo di dati del proprio abbonamento. Andiamo a vedere come si può diminuire la dimensione di un applicazione.

1. Il primo consiglio è quello di ridurre i settaggi grafici, considerato che solitamente l’aspetto grafico è quello che più incide nella dimensione di un’app.

Ad esempio, se usiamo una grafica basata su “vettori” (icone, loghi…) è consigliato usare il formato SVG, dato che con esso è possibile scalare un immagine senza perdere qualità, e soprattutto utilizzando l’SVG l’apk risulterà più leggero. Se invece usiamo immagini non basate su vettori, come possono essere delle foto, si consiglia di utlizzare il formato immagine WebP che consente una maggiore compressione senza (o con la minima) perdita di dettaglia. Usare una compressione ben fatta, come si può intuire, fa risparmiare molto spazio nella propria applicazione. Si pensi che un’immagine WebP è il 26% più piccola di una PNG.

1. Un altro suggerimento è quello di andare a risparmiare spazio nel codice. Un primo accorgimento che si può fare, è verificare che le varie librerie implementate siano ottimizzate per dispositivi mobile.

In secondo luogo si può tagliare il codice non utilizzato. Per far ciò ci vengono in aiuto strumenti come ProGuard, che si accorge delle parti di codice non utilizzate e provvede a rimuoverle. Oppure si può attivare la funzione di shrinking quando l’app esegue il build.

Questo è possibile settando nel *build.gradle,*  *minifyEnabled=True* e *shrinkResources=True* [Figura 3].

Come si fa per le librerie, anche per le API dobbiamo verificare di aver implementato solo quelle necessarie

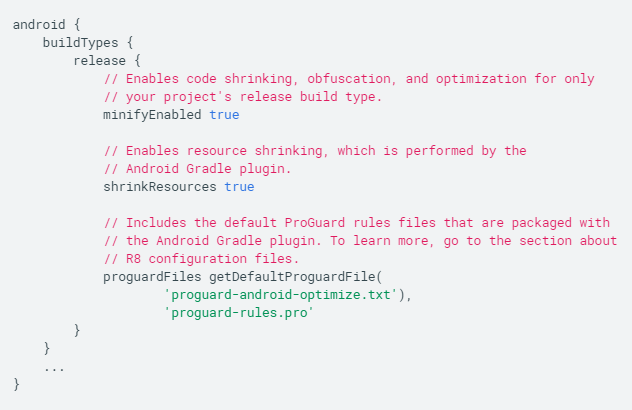


Figura 3-Shrinking

1. Una feature che è sentitmente consigliato di implementare è la possibilità di spostae l’app in una scheda SD. Per far ciò si deve andare nel *manifest* e settare *android:installLocation=”preferExternal”* se si vuole che l’app sia installata nell’SD, oppure lasciarla in *auto* se si lascia al sistema la possibilità di decidere. In entrambi i casi è possibile, dopo aver installato l’app, spostarla da una posizione all’altra.

## USO DELLA BATTERIA

Come abbiamo detto più volte, i cellulari ai quali *Build for BIllions* mira hanno delle specifiche molto basse, tra cui anche la durata della batteria. È quindi importante che l’applicazione che andremo a sviluppare non vada a consumare più energia del necessario, comunque rimanendo su valori non troppo alti.

Questo si può verificare tramite vari test di prestazione, oppure grazie al già accenato *Profiler* di Android Studio.

Andiamo ora a vedere come si può risparmiare sul consumo della batteria.

1. L’applicazione deve avere in background il minor numero possibile dei activity, e soprattutto in background deve cercare di minimizzare al massimo il numero di sensori attivi; soprattutto sensori di GPS, che succhiano una grandissima quantità di energia. In questo caso si consiglia di evitare l’uso di *LocationManager,* e al suo posto utilizzare i servizi API di localizzazione Google. Infatti questi ultimi, anche se leggermente inferiori per precisione, consumano molta meno batteria. Uno svantaggio di questi ultimi, però, è quello di dover implementare installare i *Google Play Services.*

## INTERFACCIA UTENTE

Un’ UI reattiva e interattiva è ciò che ogni utente vuole. Perciò in questo paragrafo vedremo che linee guida uno sviluppatore dovrebbe seguire.

1. **In primo luogo è importate associare un feedback ad ogni elemento che interagisce con l’utente.

Figura 4-Profilo di rendering GPU

Ad esempio all pressione di un’icona si dovrebbe inserire un feedback tattile (vibrazione) oppure un effetto visivo di pressione. Per migliorare l’interazione con l’utente è consigliato far uso dei *Material Design*, data la vastità di aspetti grafici e consigli che mette a disposizione.

1. L’occhio umano riesce al meglio bene uno schermo da 60fps, quindi una *risoluzione* inferiore vuol dire che le animazioni dello schermo non sono fluide, e ciò potrebbe essere alquanto fastidioso. Perciò assicurarsi la consistenza dei 60 fps rende l’utilizzo dell’applicazione molto più gradevole. Per quanto riguarda questo aspetto nelle impostazioni di ogni cellulare android esiste un’impostazione di monitoraggio chiamata *Profilo di rendering GPU* [Figura 4, i dettagli e il significato di tale grafico si può trovare in *https://developer.android.com/topic/performance/rendering/inspect-gpu-rendering#gpu\_rendering\_output*]*.*

Per mantenere la consistenza dei 60fps è consigliato,ad esempio, rimuovere tutte le animazioni non necessarie.

1. Inoltre un detto dice: “il buongiorno si vede dal mattino”, che si può rispecchiare anche nei dispositivi mobile. Infatti un’applicazione con un avvio rapido ha sicuramente un impatto positivo sull’utente. Esistono diversi oggetti del *Material Design* che si possono usare, tra cui il *Branded Lauch Screen*. Cosa serve quest’ultimo? Se l’applicazione ci impiega troppo tempo ad avviarsi, far comparire una schermata di avvio è il miglior modo per “nascondere” questo ritardo.

## PROFILER

In questo paragrafo andremo ad analizzare lo strumento di monitoraggio che Android Studio di Fornisce: il Profiler. Saper utilizzare correttamente questo strumento di monitoraggio quando si sviluppa un’applicazione *Build for Billions* è essenziale.

Per accedere al profiler in Android Studio basta andare su **Run** -> **Profile app.**

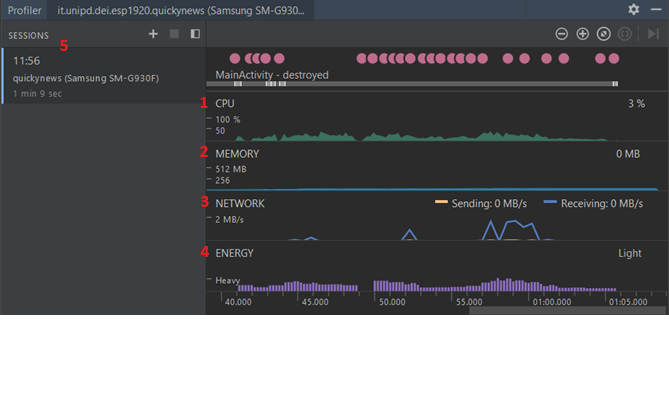


Figura 5 – Profiler di Quicky News, app che verrà descritta meglio nel capitolo 3

Il profiler ci dà informazioni utili sui vari consumi di un’app sul dispositivo.

Andiamo a vedere i numeri in rosso a cosa si riferiscono per capire meglio che informazioni ci dà il profiler:

1. CPU: qua vediamo la percentuale di consumo di CPU rispetto al totale di quella disponibile. Si può quindi capire, che più bassa è la percentuale, più il dispositivo risulterà fluido.
2. MEMORY: questa linea azzurra indica quanta memoria (in MB) stiamo occupando. Solitamente questa linea ha un picco iniziale, quando l’app di avvia, e poi rimane lineare se facciamo uso stabile dell’utilizzo della memoria (come si può vedere in Figura 5)
3. NETWORK: abbiamo già detto più volte che in un progetto dedicato a mercati emergenti è essenziale monitorare l’utilizzo di internet. Questa sezione del profiler ci aiuta proprio per questo scopo. È essenziale non avere un costante consumo di internet, né tanto meno, avere dei picchi, nei quali si consuma molto internet.
4. ENERGY: qui lo sviluppatore può visualizzare quanta energia elettrica la nostra applicazione sta consumando. È molto semplice da capire dato che Android Studio ci indica se il consumo è *light, medium* o *heavy*.

Per di più cliccando sopra a uno dei grafici qua sopra citati ci viene descritto più in dettaglio, da dove derivano i consumi.

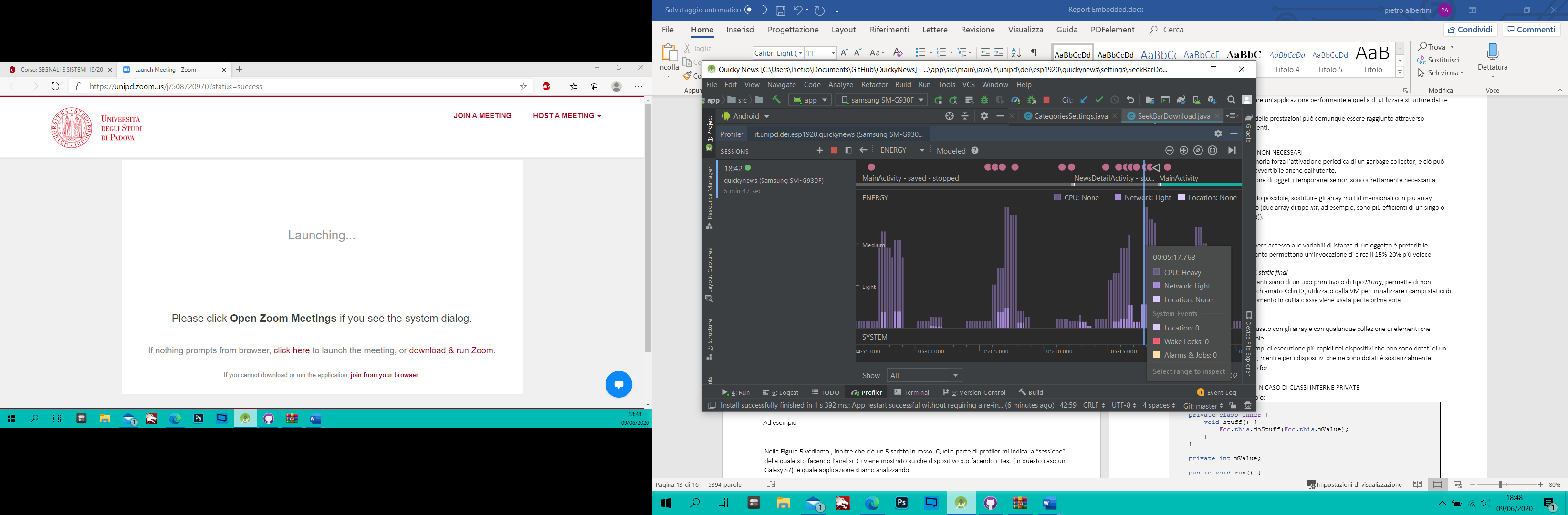
Ad esempio premendo su ENERGY vedremo un grafico simile a quello di Figura 6. In questo caso ci viene mostrato come Internet consumi poca batteria, mentre la cpu, la sforzi abbastanza (è stato fatto apposta a trovare un momento dove l’app stava “facendo fatica” per marcare la differenza tra “heavy”e light”.

Figura 6 - Zoom consumi energia

Nella Figura 5 vediamo , inoltre che c’è un 5 scritto in rosso. Quella parte di profiler mi indica la “sessione” della quale sto facendo l’analisi. Ci viene mostrato su che dispositivo sto facendo il test (in questo caso un Galaxy S7), e quale applicazione stiamo analizzando.

Sopra al numero 1 si vede anche che c’è scritto “MainActivity-destroyed”. In quella parte di schermo là possiamo capire a quale *activity* l’analisi si sta riferendo, e in quale stato è una certa activity.

# PERFORMANCE TIPS

Come ultimo paragrafo di *Build for Billions* vengono presentate le ultime regole per ottenere un’applicazione performante .

La prima regola per riuscire a sviluppare un’applicazione performante è quella di utilizzare strutture dati e algoritmi efficienti.  
Tuttavia, un sensibile miglioramento delle prestazioni può comunque essere raggiunto attraverso l’adozione di alcuni semplici accorgimenti.

1. EVITARE DI CREARE OGGETTI NON NECESSARI  
   L’allocazione continua di memoria forza l’attivazione periodica di un garbage collector, e ciò può provocare un rallentamento avvertibile anche dall’utente.  
   È quindi sconsigliata la creazione di oggetti temporanei se non sono strettamente necessari al funzionamento del codice.  
   Inoltre è buona norma, quando possibile, sostituire gli array multidimensionali con più array monodimensionali in parallelo (due array di tipo *int*, ad esempio, sono più efficienti di un singolo array di oggetti di tipo (*int*, *int*)).
2. PREFERIRE METODI STATICI  
   Quando non sia necessario avere accesso alle variabili di istanza di un oggetto è preferibile utilizzare metodi statici in quanto permettono un’invocazione di circa il 15%-20% più veloce.
3. DEFINIRE LE COSTANTI COME *static final*  
   Questo, nel caso in cui le costanti siano di un tipo primitivo o di tipo *String*, permette di non invocare un metodo speciale chiamato <clinit>, utilizzato dalla VM per inizializzare i campi statici di una classe ed eseguito nel momento in cui la classe viene usata per la prima vota.
4. UTILIZZARE IL *for-each*  
   Questo costrutto può essere usato con gli array e con qualunque collezione di elementi che implementi l’interfaccia Iterable.  
   Esso permette di ottenere tempi di esecuzione più rapidi nei dispositivi che non sono dotati di un compilatore JIT (Just-In-Time), mentre per i dispositivi che ne sono dotati è sostanzialmente equivalente a un classico ciclo for.
5. PREFERIRE *package* A *private* IN CASO DI CLASSI INTERNE PRIVATE  
   Si consideri il seguente esempio:  
     
   Si noti che la classe privata *Inner*, interna rispetto a *Foo*, fa uso nel suo metodo *stuff()* di un metodo e di una variabile appartenenti alla classe esterna e dichiarati come *private*.  
   In questo caso, nonostante le specifiche del linguaggio lo permettano, la VM considera *Inner* e *Foo* come due classi separate e segnala come illegale il tentativo di *Inner* di accedere ai *field* privati di *Foo.*  
   Per ovviare a questo problema, il compilatore aggiunge automaticamente i seguenti due metodi:

public class Foo {  
    private class Inner {  
        void stuff() {  
            Foo.this.doStuff(Foo.this.mValue);  
        }  
    }  
  
    private int mValue;  
  
    public void run() {  
        Inner in = new Inner();  
        mValue = 27;  
        in.stuff();  
    }  
  
    private void doStuff(int value) {  
        System.out.println("Value is " + value);  
    }  
}

Essi vengono invocati da *Inner* per accedere alla variabile *mValue* e al metodo *doStuff()*.  
Per evitare questo *overhead*, nei casi in cui non sia obbligatorio utilizzare *private*,è preferibile usare *package* come modificatore di accesso per quei blocchi di codice appartenenti alla classe esterna che vengono richiesti da una classe interna.

static int Foo.access$100(Foo foo) {  
    return foo.mValue;  
    }

static void Foo.access$200(Foo, int value) {  
    foo.doStuff(value);  
    }

1. EVITARE DI UTILIZZARE I *floating-point*  
   Generalmente variabili di tipo *float* e *double* vengono gestite due volte più lentamente rispetto a variabili di tipo *int*.  
   Inoltre, nonostante il tipo *float* richieda la metà dello spazio rispetto al tipo *double*, quest’ultimo è quasi sempre da preferire.
2. CONOSCERE E UTILIZZARE LE LIBRERIE  
   Questo può portare a notevoli miglioramenti in termini di performance.  
   Ad esempio, Android riporta che su un Nexus One con compilatore JIT utilizzare il metodo *arraycopy()* della classe *System* assicura un tempo di esecuzione nove volte più veloce rispetto a un normale ciclo.
3. USARE CON ATTENZIONE I METODI NATIVI  
   Sviluppare la propria applicazione in codice nativo non è necessariamente più conveniente rispetto a utilizzare Java, sia dal punto di vista dell’efficienza che della portabilità.  
   Inoltre, per quanto riguarda la portabilità, va considerato il fatto che oltre a dover compilare il proprio codice in più versioni, una per ogni architettura, è possibile anche che il codice nativo compilato per un dispositivo con un certo processore risulti non ottimizzato o addirittura ineseguibile su un altro dispositivo con lo stesso processore.
4. “MITI” SULLE PRESTAZIONI  
   Su dispositivi non dotati di un compilatore JIT l’invocazione di un metodo attraverso una variabile di tipo interfaccia è più lenta rispetto alla stessa invocazione con una variabile di un tipo “concreto” (ad esempio, *HashMap* invece di *Map*). Tuttavia, questa differenza è solamente di circa il 6%, mentre arriva praticamente ad azzerarsi nei dispositivi che fanno uso di un compilatore JIT.  
   Inoltre, mentre nei dispositivi in cui JIT è assente gli accessi a variabili locali impiegano circa il 20% di tempo in meno rispetto agli accessi a *field*, nei dispositivi in cui esso è presente la differenza tra i due è pressoché nulla.

# SITOGRAFIA

* *https://www.android.com/versions/go-edition/*
* *https://developers.google.com/speed/webp/*
* *https://developer.android.com/studio/build/shrink-code.html*
* *https://appus.software/blog/difference-between-locationmanager-and-google-location-api-services*
* *https://material.io/*
* *https://developer.android.com/training/articles/perf-tips*
* *https://developer.android.com/docs/quality-guidelines/build-for-billions*