Lezione 21

Programmazione Android

- Tecnologie di rete
  - Networking TCP/IP
  - Il Connectivity manager
  - Bluetooth
  - Wi-fi direct
  - NFC
- Accesso ai servizi Google
  - Google APIs for Android

Networking TCP/IP

Visione generale

- Il networking via TCP/IP su Android è **completamente standard**
- Vale tutto quanto conoscete sul networking in Java
- Il S.O. Utilizza tutti i “trucchi” soliti per garantire (best-effort) il delivery dei vostri pacchetti
  - Routing su reti di diverso tipo (wi-fi o cellulari)
  - Gestione dinamica al variare della connessione

Esempio
ovvero
Networking in Java in 2 slide

- Server-side

```
try {
   ServerSocket ss = new ServerSocket(8080);
   while (!done) {
      Socket s = ss.accept();
      servi(s);
   }
} catch (…) { … }
```

- Client-side

```
try {
   Socket s = new Socket(server, 8080);
   ordina(s);
} catch (…) { … }
```

Esempio
ovvero
Networking in Java in 2 slide

- Letture e scritture su rete avvengono tramite gli *stream* tipici di Java
  - Dal Socket si estraggono InputStream e OutputStream

```
InputStream is = s.getInputStream();
OutputStream os = s.getOutputStream();
```

  - Letture e scritture avvengono come normale
    - Tipicamente, incapsulando gli stream in BufferedStream, PrintWriter, DataInput/OutputStream, ObjectInput/OutputStream, ecc.

Concorrenza

- Naturalmente, le operazioni su rete possono essere lente e/o bloccanti
  - **Mai** accedere alla rete nel thread UI!
- È anche poco comune avere un server sul cellulare
  - Il server dovrebbe, per sua natura, essere in esecuzione *sempre...*
  - L'approccio di Android è che i programmi utente dovrebbero essere in esecuzione *mai...*
    - Per ricevere notifiche push, si usano altre strade

Batteria

- Le operazioni su rete consumano in genere molta energia (specialmente in trasmissione)
  - Necessaria perché i segnali radio siano ricevuti a destinazione
- È opportuno cercare di minimizzare il consumo
  - Trasferire pochi dati (aiuta anche con i costi)
  - Mantenere delle cache quando possibile
  - Fare *coalescing*: poche comunicazioni “grosse”
    - Ogni ri-accensione dei modem 3G richiede tempo e energia
    - Se dovete fare una comunicazione, fatene tante insieme!

Il ConnectivityManager

ConnectivityManager

- Il ruolo del ConnectivityManager è di fornire informazioni sulle reti accessibili al dispositivo
  - Momento per momento
- Si tratta dell'ennesimo servizio di sistema:

```
ConnectivityManager cm = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
```

- Ogni network è rappresentato da un oggetto

```
NetworkInfo[] ni = cm.getAllNetworkInfo();
NetworkInfo currni = cm.getActiveNetworkInfo();
NetworkInfo wfni = cm.getNetworkInfo(NetworkInfo.TYPE_WIFI);
```

NetworkInfo

- Un oggetto NetworkInfo mantiene tutte le informazioni disponibili su un particolare network
- Tipi supportati:
  - TYPE_BLUETOOTH
  - TYPE_DUMMY
  - TYPE_ETHERNET
  - TYPE_MOBILE
    - TYPE_MOBILE_DUN, TYPE_MOBILE_HIPRI, TYPE_MOBILE_MMS, TYPE_MOBILE_SUPL
  - TYPE_VPN
  - TYPE_WIFI
  - TYPE_WIMAX

API 21+
Nota: negli USA, Google è un VNO (virtual network operator), e instrada le chiamate vocali sulle sue reti VPN...

NetworkInfo

- Una volta ottenuto un NetworkInfo, si possono chiedere le caratteristiche di dettaglio della rete
  - Info: getType(), getTypeName(), getDetailedState(), getExtraInfo()
  - Stato: isAvailable(), isConnected(), isFailOver(), isRoaming() isConnectedOrConnecting(),
  - Errori: getReason()

Notifiche

- Il ConnectionManager invia degli Intent broadcast ogni volta che cambiano le condizioni della rete
  - Action CONNECTIVITY_ACTION
  - Extras contiene ulteriori info
- Potete registrare un BroadcastReceiver nel manifest, con un intent filter
  - Ma in questo caso, il vostro codice verrà eseguito **sempre**, anche quando l'app non è in esecuzione
- Oppure, registrare dinamicamente il receiver
  - Di solito, in onCreate() / onDestroy()

Ma ricordate i limiti ai broadcast sulle versioni di Android più recenti!

Uso delle notifiche per ottimizzare la batteria

- Un'applicazione può ricevere notifiche che segnalano il passaggio a un nuovo tipo di rete
  - Per esempio, fra wi-fi e cellulare
- In molti casi, è utile modificare i pattern di accesso alla rete in base al tipo (e alle preferenze)
  - Per esempio, scaricare un podcast solo su wi-fi, ma ricevere le notifiche sulla disponibilità di una nuova puntata anche su 3G
- Nei casi di polling, ricordarsi degli inexact alarms
  - Si può fare coalescing anche fra app diverse!

Bluetooth

Bluetooth

- Bluetooth è la più diffusa tecnologia per le *personal area network*
  - L'idea è di offrire connettività solo a dispositivi che sono in prossimità fisica alla persona dell'utente
  - Protocollo complicato, prevede molti *profili* per diverse classi di dispositivi
    - Auricolari, trasferimento file, streaming di audio in casa, dispositivi medicali, ecc.
- Ci interessa in particolare il profilo RFCOMM
  - “Comunicazioni seriali in radiofrequenza”

Pairing & co.

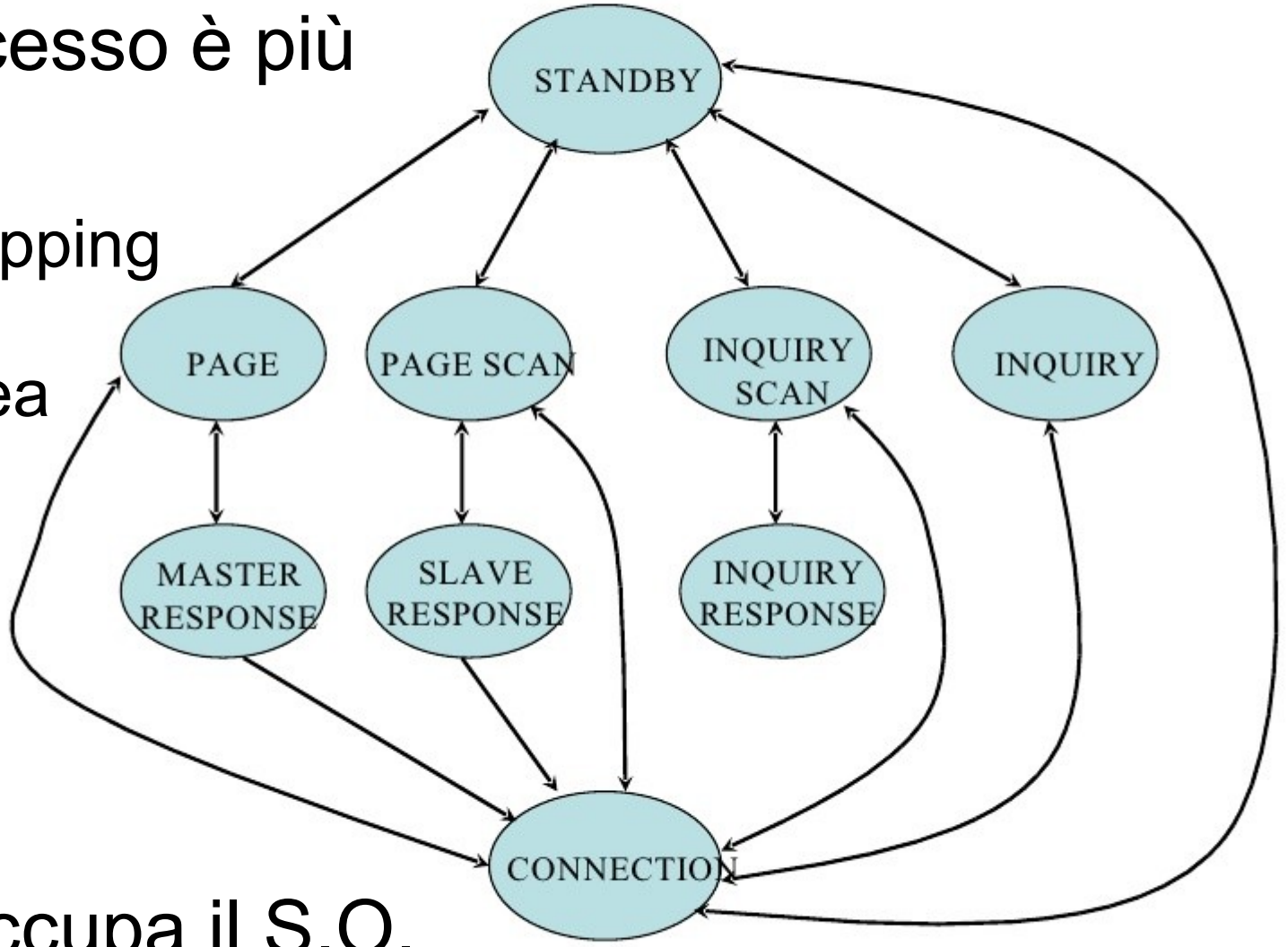
- Bluetooth prevede che i dispositivi debbano essere *accoppiati* prima di poter scambiare dati a livello applicativo
  - Naturalmente, possono scambiare dati anche prima (per esempio, i loro nomi e classi), ma solo a livello di protocollo
- L'accoppiamento deve confermare la volontà dell'utente/proprietario di entrambi i dispositivi
  - Per questo motivo, richiede in genere un segnale esplicito da parte sua: passkey (PIN)

Pairing & co.

- Un dispositivo Bluetooth può quindi essere in diversi stati
  - Spento: modulo Bluetooth non attivato
  - Acceso, ma non discoverable
  - Discoverable ma non paired
  - Discoverable e paired
  - Paired e non discoverable
- D'altra parte, l'altro dispositivo può attivamente cercare dispositivi discoverable o no

Pairing & co.

- In realtà il processo è più complesso
  - Frequency hopping su 12 canali diversi per area geografica
  - Point-to-point, piconet, scatternet
  - Master/slave
- … ma se ne occupa il S.O.



Le classi principali

- Nella maggior parte dei casi, si usano 4 classi
  - **BluetoothAdapter** – rappresenta la scheda di rete
  - **BluetoothDevice** – rappresenta un dispositivo
  - **BluetoothServerSocket** e **BluetoothSocket**
    - Analoghi a ServerSocket e Socket del networking TCP/IP
- Esistono poi diverse altre classi
  - Specializzazioni per particolari profili
  - Classi che descrivono i meta-dati dei profili
  - Non le vedremo...

BluetoothAdapter

- In teoria, un dispositivo potrebbe avere più adapter Bluetooth; occorre quindi prendere una particolare istanza, per esempio:

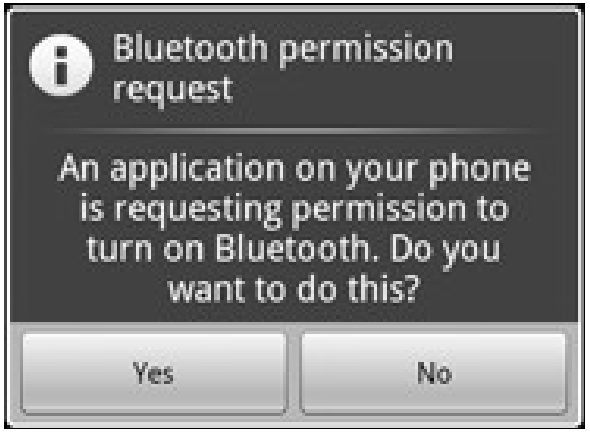
```
BluetoothAdapter bta =BluetoothAdapter.getDefaultAdapter();
```

- Se bta è null, il dispositivo non supporta BT
- Altrimenti, si controlla se BT è abilitato, e in caso contrario si chiede all'utente di abilitarlo

```
if (!bta.isEnabled()) {
  Intent i=new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
  startActivityForResult(i, 1);
}
```

Abilitare Bluetooth

- L'utente a questo punto è libero di decidere se abilitare o meno BT
- L'activity che risponde all'intent è un dialog di sistema



- Verrà poi chiamata la onActivityResult() passando un risultato che indica l'esito

```
void onActivityResult(int code, int res, Intent data) {
  If (code==1) {
    If (res==RESULT_OK) { /* BT abilitato! */ }
    if (res==RESULT_CANCEL) { /* niet! */ }
  } …
```

Abilitare Bluetooth

- In alternativa, è anche possibile registrarsi per ricevere l'intent broadcast con action BluetoothAdapter.ACTION_STATE_CHANGED
- Gli extra dell'intent contengono informazioni sullo stato corrente
  - L'app può anche fare “piggybacking” – *non* richiede lei di attivare il BT, ma è pronta a partire se qualcun altro lo attiva
  - Stessa tecnica per scoprire la disattivazione

Discovery

- Il secondo passo consiste nello scoprire con quali BluetoothDevice possiamo comunicare
- Dispositivi paired in passato (e registrati):

```
Set<BluetoothDevice> devs = bta.getBondedDevices()
```

- Dispositivi discoverable e in range:

```
IntentFilter f = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(this,f);
bta.startDiscovery();
```

  - Il nostro onReceive() riceverà tanti intent ACTION_FOUND quanti sono i dispositivi in range
    - Ciascuno ha negli extra una descrizione completa

Discovery

- Esempio di receiver:

```
void onReceive(Context c, Intent i) {
  String action = i.getAction();
  if (BluetoothDevice.ACTION_FOUND.equals(action)) {
    BluetoothDevice dev =
              i.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
  } …
}
```

- Per rendersi discoverable a propria volta:

```
Intent i=new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
startActivityForResult(i, 2);
```

  - Vale quanto detto per l'abilitazione (RESULT_OK/CANCELED)

Collegamento

- Una volta che sia stato effettuato il pairing, i due dispositivi possono comunicare
- Uno farà da server, l'altro da client
  - Il client è l'iniziatore della connessione
  - Il server si mette in attesa di connessione
- La connessione è ammessa solo se **entrambi i device presentano lo stesso UUID**
  - L'UUID può essere un accordo privato
  - Svolge lo stesso ruolo del numero di porta in TCP/IP

Collegamento – server

```
String myname = "it.unipi.di.sam.bttest server";
UUID myid = UUID.fromString("550e8400-e29b-41d4-a716-446655440000");
BluetoothServerSocket bss =
                    bta.listenUsingRfcommWithServiceRecord(myname,myid);
BluetoothSocket bs = bss.accept();
bss.close();
servi(bs);
```

- L'intero processo è simile a quello per TCP/IP
  - Tuttavia, raramente il server rimane attivo in attesa di ulteriori connessioni; più spesso si tratta di connessioni singole
  - Come al solito: mai nel thread UI!

Collegamento – client

- Il client deve specificare a quale device e servizio vuole connettersi
  - Il device è uno dei device paired (ottenuto come visto prima)
  - Il servizio è identificato dall'UUID

```
UUID hisid = UUID.fromString("550e8400-e29b-41d4-a716-446655440000");
BluetoothDevice btd = … ;
BluetoothSocket bs = btd.createRfcommSocketToServiceRecord(hisid);
bs.connect();  /* ← exception se la connessione non riesce */
ordina(bs);
```

Scambio di dati

- Una volta che sia il server che il client hanno ottenuto un loro BluetoothSocket, il processo è il solito
  - Si ottengono InputStream e OutputStream per ogni socket
  - Direttamente o tramite wrapping si procede allo scambio di dati
  - Problemi di rete si traducono in fallimenti delle letture/scritture o eccezioni
  - Al termine, si chiama close() per chiudere la RFCOMM

Wi-fi direct

Wi-fi direct

- Wi-fi direct è una tecnologia relativamente nuova per le connessioni point-to-point via wi-fi
  - Disponibile da Android 4.0 in poi
  - Casi d'uso analoghi a Bluetooth, ma molto più veloce e con range assai più esteso
  - I dispositivi devono avere entrambi wi-fi, ma **non** necessitano di una base station
- Consente anche di creare **gruppi p2p**
  - Dispositivi multipli, tutti in range, che comunicano liberamente

Design del framework

- A differenza dei network layer precedenti, lo strato wi-fi direct di Android è stato sviluppato nativamente
- Fa quindi uso del design più tipico di Android
  - Si lanciano intent per chiedere azioni
  - Si ricevono notifiche via listener
- La classe WifiP2pManager fornisce i metodi che semplificano la vita

In effetti, il nome originale della tecnologia era "Wi-Fi P2P", quando il P2P era cool.
Quando sono cominciate le cause contro i pirati del P2P, il nome "WiFi Direct" improvvisamente suonava meglio...

WifiP2pManager

- Il solito servizio di sistema

```
WifiP2pManager wfdm = (WifiP2pManager)
                    getSystemService(Context.WIFI_P2P_SERVICE);
```

- L'inizializzazione del sistema ci fornisce un canale da usare poi come handle

```
Channel ch = wfdm.initialize();
```

- I metodi del WifiP2pManager consentono di chiedere le principali funzioni
  - Le risposte arriveranno tramite intent broadcast o attraverso chiamate a dei listener

WifiP2pManager

- Metodi principali:

Method	Description
initialize()	Registers the application with the Wi-Fi framework. This must be called before calling any other Wi-Fi Direct method.
connect()	Starts a peer-to-peer connection with a device with the specified configuration.
cancelConnect()	Cancels any ongoing peer-to-peer group negotiation.
requestConnectInfo()	Requests a device's connection information.
createGroup()	Creates a peer-to-peer group with the current device as the group owner.
removeGroup()	Removes the current peer-to-peer group.
requestGroupInfo()	Requests peer-to-peer group information.
discoverPeers()	Initiates peer discovery
requestPeers()	Requests the current list of discovered peers.

Discovery

- Il processo di Discovery inizia con una chiamata a discoverPeers():

```
wfdm.discoverPeers(ch, new WifiP2pManager.ActionListener()
{
    @Override
    public void onSuccess() {
        ...
    }

    @Override
    public void onFailure(int reasonCode) {
        ...
    }
});
```

Indica solo che è andata bene

Codice di fallimento

Discovery

- Una volta che il processo di discovery è concluso, il sistema invia un broadcast intent
  - WIFI_P2P_PEERS_CHANGED_ACTION
- A questo punto, la nostra applicazione (che avrà "visto" l'intent tramite un receiver) può chiamare requestPeers()

```
void onReceive(Context c, Intent i) {
  String action = i.getAction();
  if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
    wfdm.requestPeers(ch, pllist);
  } …
}
```

Altro listener!

Discovery

- Quest'ultimo listener sarà un WifiP2pManager.PeerListListener
  - Unico metodo: onPeersAvailable(WifiP2pDeviceList peers)
- Finalmente, peers contiene la lista dei dispositivi raggiungibili!
- È ora possibile iniziare una connessione

Connessione

- Assumendo che dev sia un dispositivo dalla lista, possiamo creare una **configurazione** corrispondente, ed effettuare una connessione

```
WifiP2pConfig cfg = new WifiP2pConfig();
cfg.deviceAddress = dev.deviceAddress;
wfdm.connect(ch, cfg, new ActionListener() {
    @Override
    public void onSuccess() {
      …
    }
    @Override
    public void onFailure(int reason) {
      …
    }
});
```

Trasferimento

- La buona notizia:
  - Una volta che i dispositivi sono connessi, si possono usare le normali socket TCP/IP
  - Compaiono come dispositivi su una rete locale privata
  - Niente di nuovo da scoprire!

NFC

NFC

- All'altro estremo della gamma, abbiamo le comunicazioni NFC
  - NFC = near field comunication
  - Distanza massima: 4cm (a contatto)
- Tre casi d'uso:
  - Lettura / scrittura di *tag NFC*
  - NFC P2P (comunicazione, es.: Android Beam)
  - Simulazione di un *tag NFC*

Leggere Tag NFC

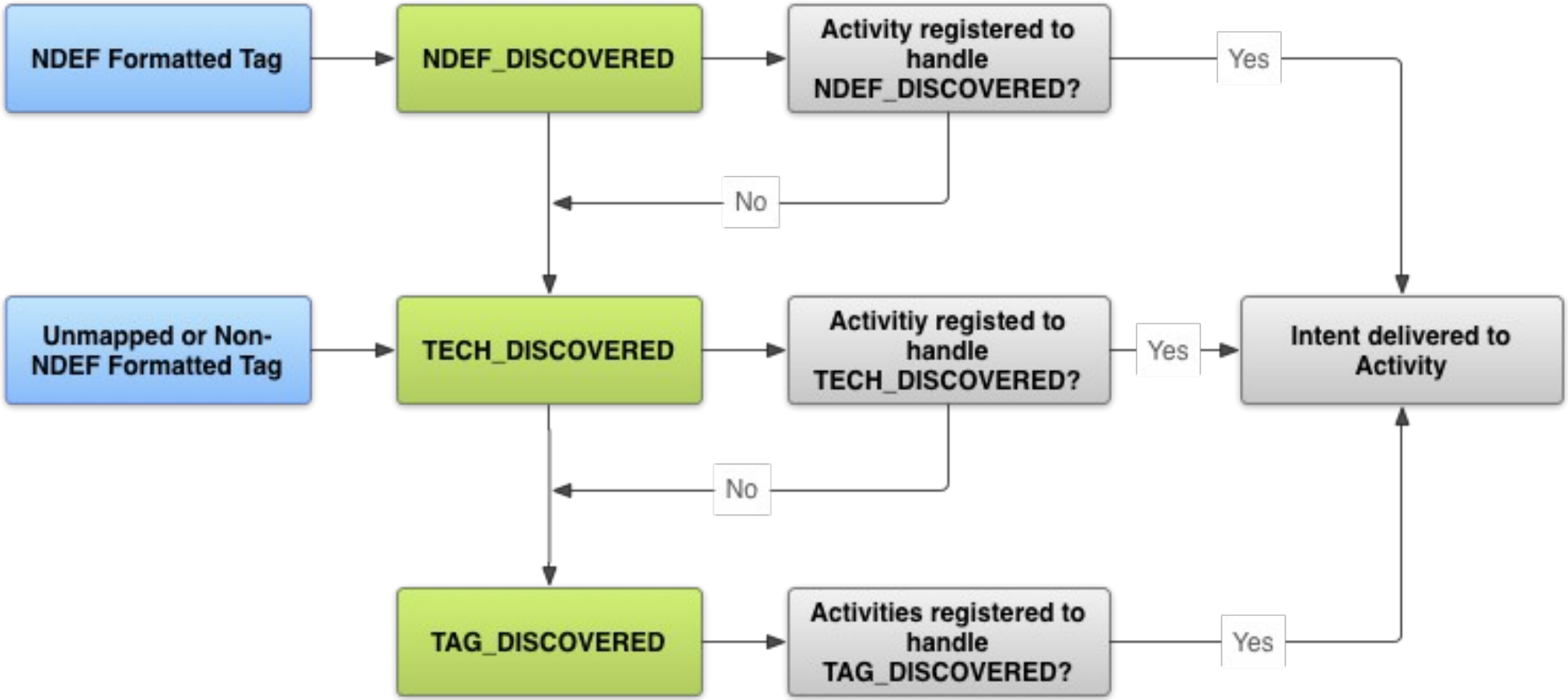
- I tag NFC fungono da “memoria di massa”
  - Capacità molto limitata
- Le applicazioni possono registrarsi per essere attivate quando il dispositivo è posto a contatto con un tag NFC

```
<intent-filter>
    <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

- L'intent contiene fra gli extra le informazioni del tag
  - Esistono alcuni altri intent per casi più specifici

Leggere Tag NFC

Lo standard NDEF specifica una (lunga) serie di formati per i messaggi NFC



Spedire messaggi P2P (Android Beam)

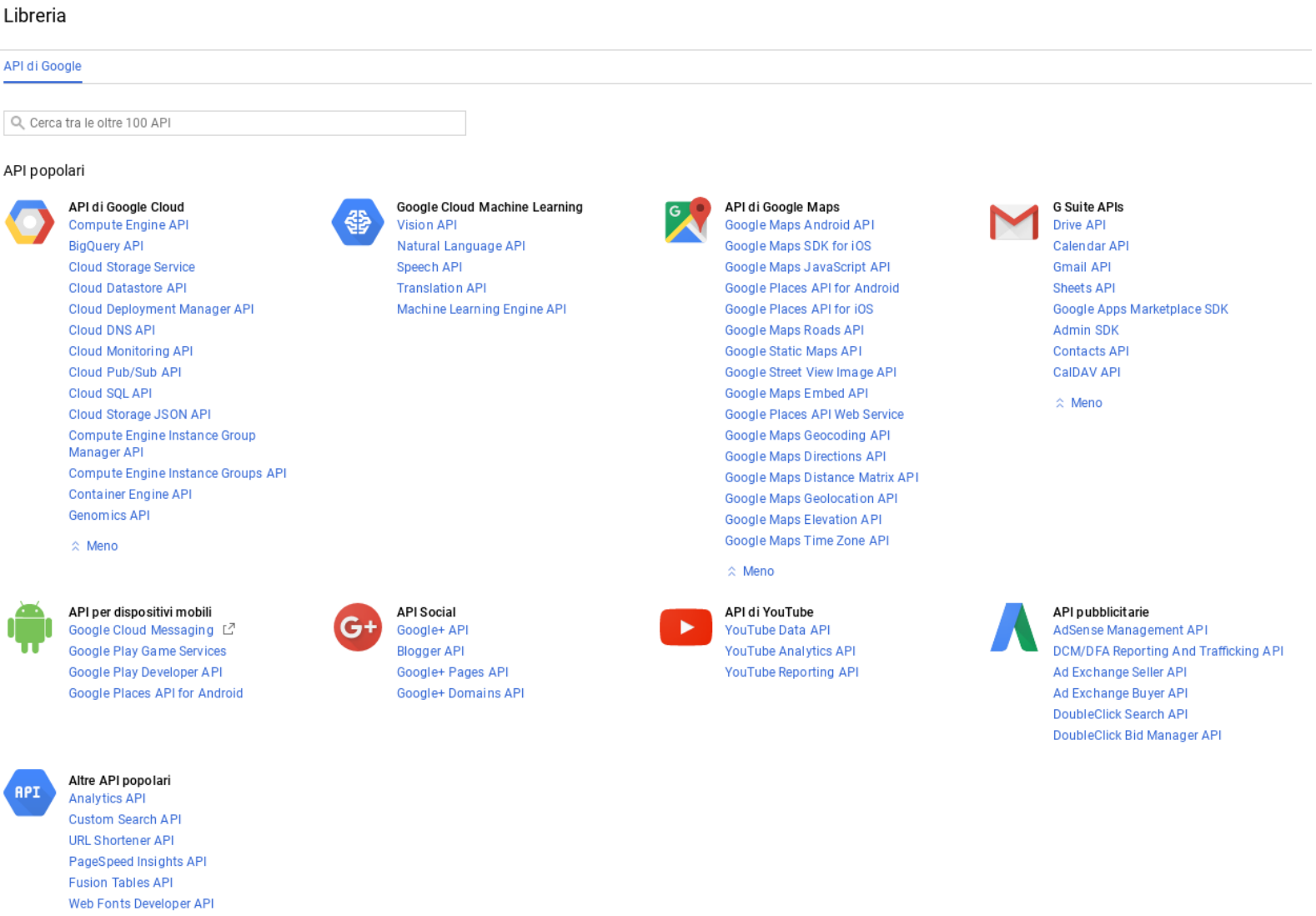
- La classe NfcAdapter offre il metodo **setNdefPushMessage**(msg, activities…)
  - msg è un NdefMessage
  - activities è un insieme di activity della nostra app
- Dopo aver impostato il push message, in automatico:
  - Quando un altro dispositivo con NFC P2P è in range
  - Se una delle activities è in foreground
  - Il msg viene spedito

Esiste anche una variante in cui si registra una callback, e si crea il msg sul momento

Google APIs

Google APIs

- Oltre 100 APIs
- Ciascuna con dozzine o centinaia di metodi
- Così tanti servizi da richiedere un motore di ricerca interno con una sezione delle API più popolari!
- Marketplace di APIs a pagamento



API Explorer

https://developers.google.com/apis-explorer

Google

Search for services, methods, and recent requests... Loading...

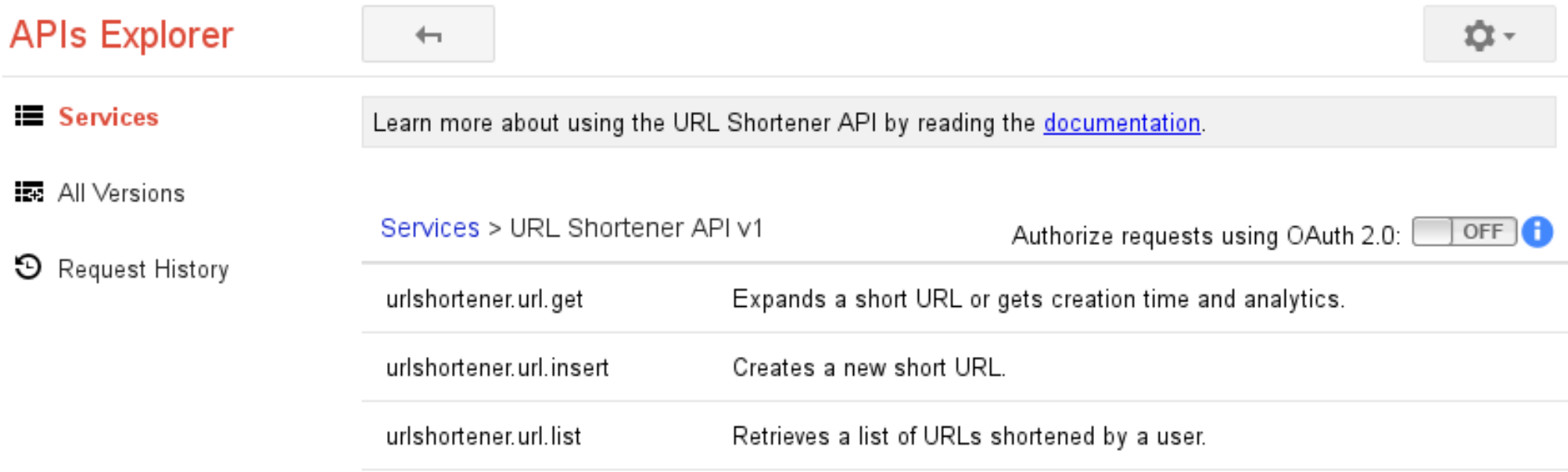
APIs Explorer

- Services
- All Versions
- Request History

Accelerated Mobile Pages (AMP) URL API	v1	This API contains a single method, batchGet. Call this method to retrieve the AMP URL (and equivalent AMP Cache URL) for given public URL(s).
Ad Exchange Buyer API	v1.4	Accesses your bidding-account information, submits creatives for validation, finds available direct deals, and retrieves performance reports.
Ad Exchange Buyer API II	v2beta1	Accesses the latest features for managing Ad Exchange accounts and Real-Time Bidding configurations.
Ad Exchange Seller API	v2.0	Accesses the inventory of Ad Exchange seller users and generates reports.
Admin Reports API	reports_v1	Fetches reports for the administrators of G Suite customers about the usage, collaboration, security, and risk for their users.
AdSense Host API	v4.1	Limited Availability Generates performance reports, generates ad codes, and provides publisher management capabilities for AdSense Hosts.
AdSense Management API	v1.4	Accesses AdSense publishers' inventory and generates performance reports.
APIs Discovery Service	v1	Provides information about other Google APIs, such as what APIs are available, the resource, and method details for each API.
BigQuery API	v2	A data platform for customers to create, manage, share and query data.
BigQuery Data Transfer Service API	v1	Transfers data from partner SaaS applications to Google BigQuery on a scheduled, managed basis.
Blogger API	v3	Limited Availability API for access to the data within Blogger.
Books API	v1	Searches for books and manages your Google Books library.
Calendar API	v3	Manipulates events and other calendar data.
Cloud Monitoring API	v2beta2	Accesses Google Cloud Monitoring data.
Cloud Source Repositories API	v1	Access source code repositories hosted by Google.
Cloud Spanner API	v1	Cloud Spanner is a managed, mission-critical, globally consistent and scalable relational database service.
Cloud SQL Administration API	v1beta4	Creates and configures Cloud SQL instances, which provide fully-managed MySQL databases.
Cloud Storage JSON API	v1	Stores and retrieves potentially large, immutable data objects.

API Explorer

https://developers.google.com/apis-explorer

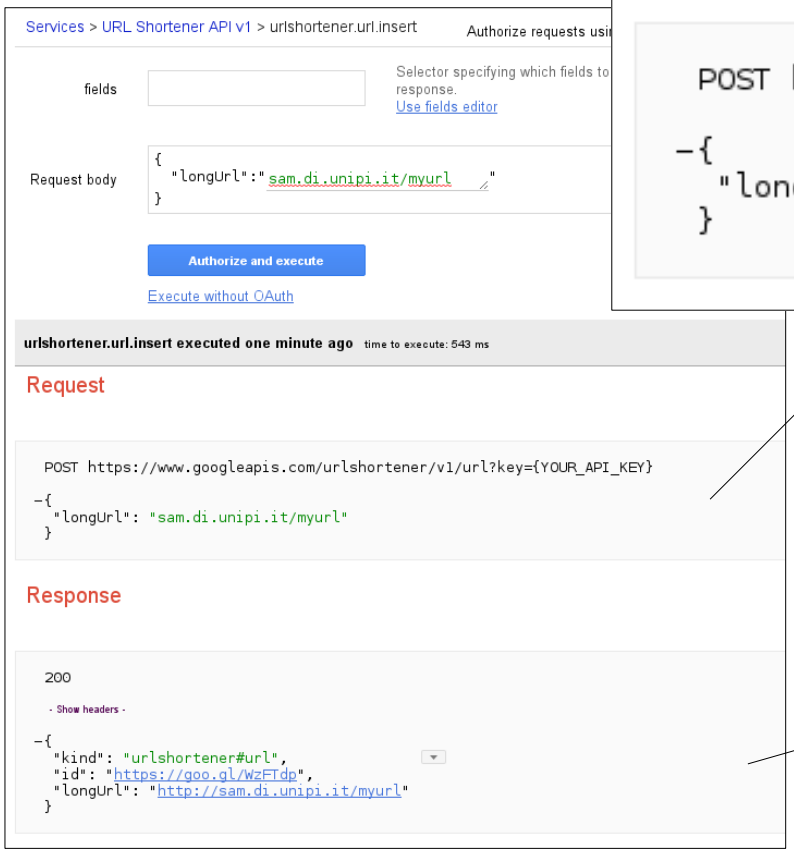


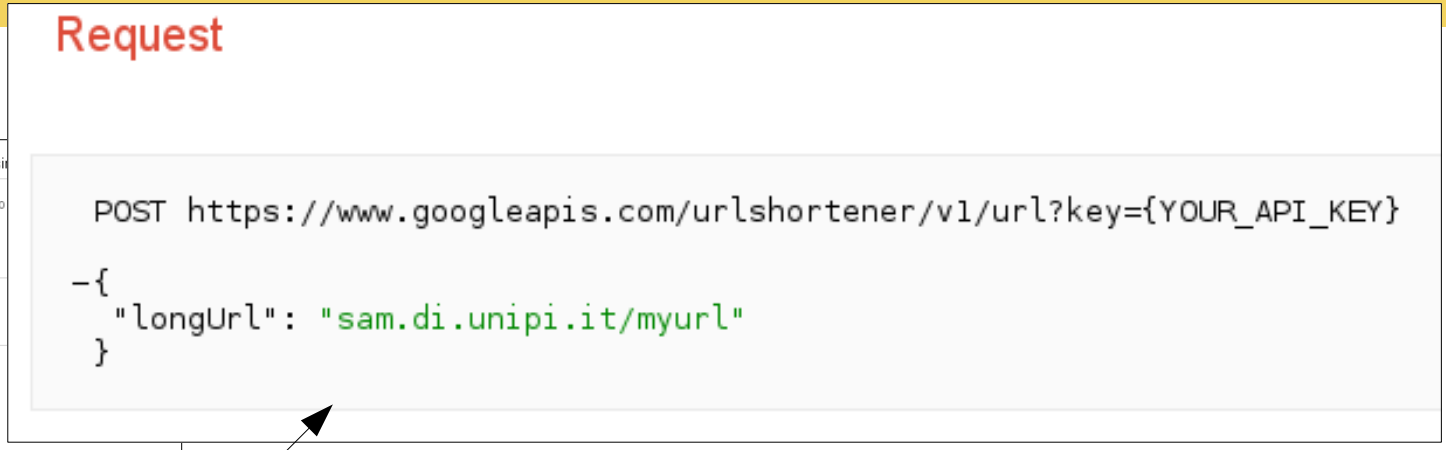
Ogni API offre un insieme di metodi che possono essere chiamati in stile REST.

Ogni chiamata incluse una **richiesta** (HttpRequest), tipicamente con un *payload* JSON che fornisce gli argomenti.

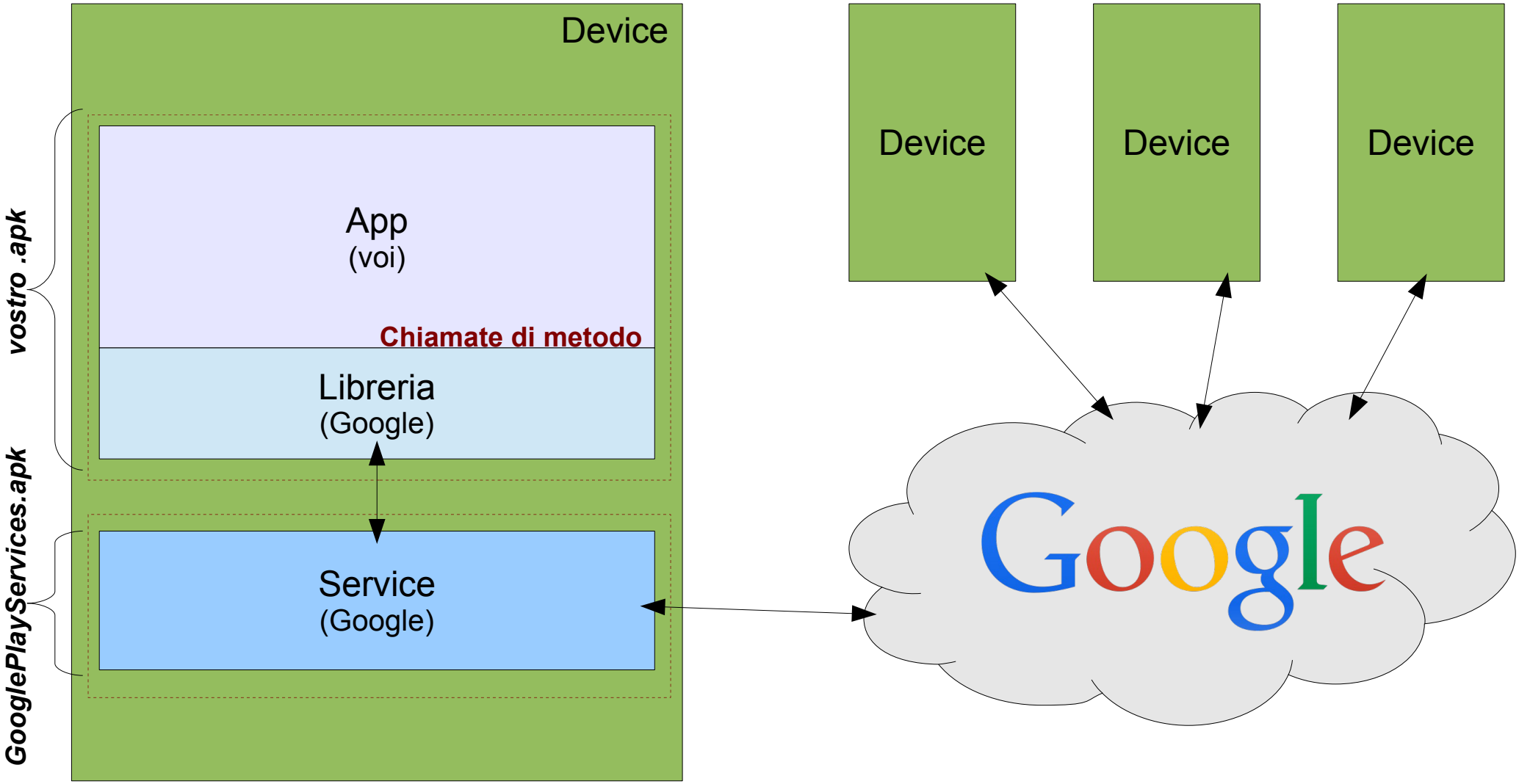
La **risposta** è una coppia <*codice*,*corpo*> in cui il *codice* è un error code HTTP (200=ok, 404=forbidden, ecc.), mentre il *corpo* è un oggetto JSON i cui campi rappresentanto il risultato della chiamata.

Esempio: Url shortener

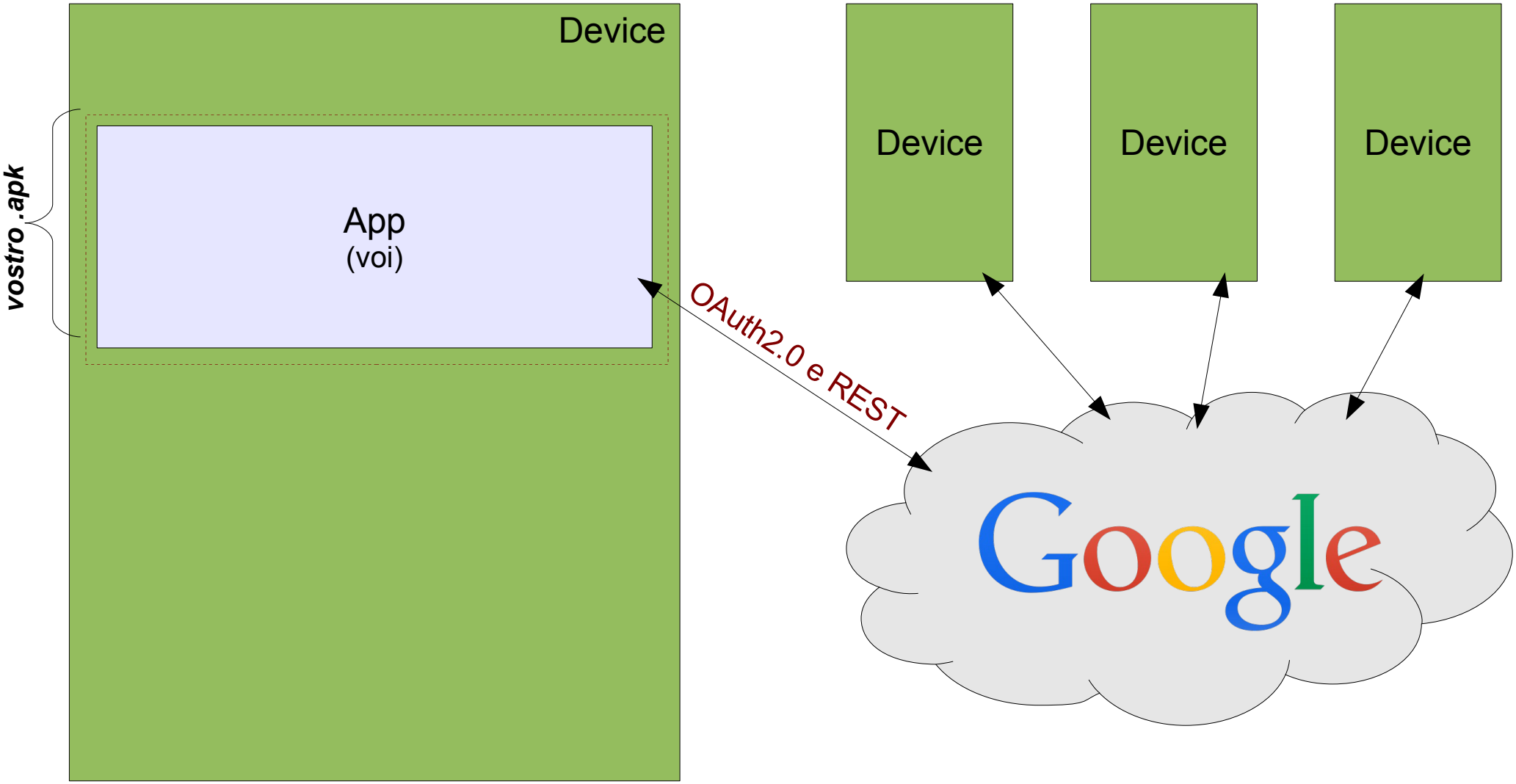




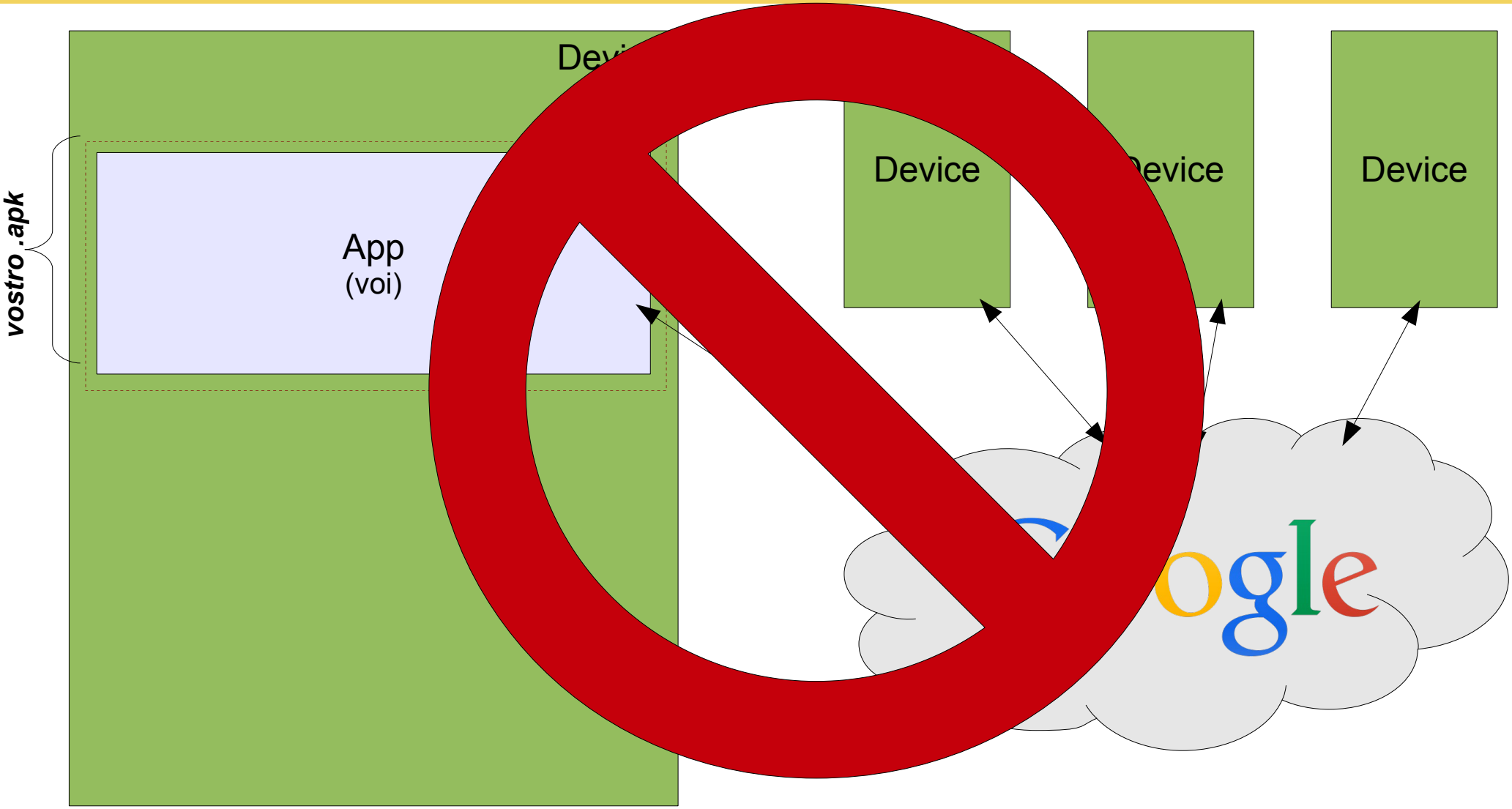
Architettura "Google APIs for Android"



Architettura alternativa



Architettura alternativa



Connessione a Google API

- Usiamo il pattern *Builder*, aggiungendo due interfacce e un gruppo di API:

```
GoogleApiClient gapi = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(new GoogleApiClient.ConnectionCallbacks() {
        public void onConnected(Bundle hint) {
            // Possiamo usare l'API
        }
        public void onConnectionSuspended(int res) {
            // Accesso all'API sospeso – ma tornerà!

        }
    })
    .addOnConnectionFailedListener(new GoogleApiClient.OnConnectionFailedListener() {
        public void onConnectionFailed(ConnectionResult res) {
                // Connession fallita. Addio mondo crudele!
        }
    })
    .addApi(Wearable.API) // Lista delle Google API a cui vogliamo accedere
    .build();
```

Spesso (ma non sempre) fatto nella onCreate() dell'activity

Connessione a GoogleAPI

- Si può anche, come al solito, fare implementare le interfacce all'Activity in questione, attivare più API insieme, ecc.
  - E magari scrivere di meno!
- Per esempio:

```
gapi = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .addApiIfAvailable(Wearable.API)
    .addApi(Drive.API)
    .addScope(Drive.SCOPE_FILE)
    .build();
```

Wear solo se disponibile.

Poi si può usare gapi.hasConnectedApi(Wearable.API) per scoprire a runtime se l'API è disponibile.

Drive sempre – altrimenti è un errore

Connessione a GoogleAPI

- L'ultimo passo è anche il più semplice!
- Per avviare la connessione alle API selezionate:

```
gapi.connect();
```

- Schemi tipici:
  - Uso solo durante l'attività
    - gapi.connect() nella onStart(), gapi.disconnect() nella onStop()
  - Uso solo in certi casi specifici
    - gapi.connect(); operazione; gapi.disconnect()
  - Uso continuo
    - gapi.connect() nella onCreate(), gapi.disconnect() nella onDestroy()

Connessione a GoogleAPI

- La gestione degli errori invece è una tragedia
  - Dopo una chiamata a connect() può darsi che…
    - Tutto bene: onConnected(hint)
    - Qualcosa storto: onConnectionFailed(res)
- In caso di fallimenti, è possibile che il sistema offra una ***risoluzione*** implicita
  - Esempio: il client non era autenticato
  - Spesso le risoluzioni necessitano di azione dell'utente
  - In ogni caso… possiamo esprimere l'*intenzione* di rimediare!

Connessione a GoogleAPI

- Esempio

```
public void onConnectionFailed(ConnectionResult res) {
    if (risincorso) {
        // Stiamo già cercando di risolvere l'errore
        return;
    } else if (res.hasResolution()) {
        try {
            risincorso = true;
            res.startResolutionForResult(this, REQUEST_RESOLVE_ERROR);
        } catch (SendIntentException e) {
            // Il tentativo di risoluzione ha causato errore. Riproviamo.
            gapi.connect();
        }
    } else {
        // Niente da fare, dialog d'errore all'utente
        risincorso = true;
        showErrorDialog(res.getErrorCode());
    }
}
```

GooglePlayServicesUtil.getErrorDialog(errorCode, this.getActivity(), REQUEST_RESOLVE_ERROR);

Connessione a GoogleAPI

- Se abbiamo avviato un tentativo di risoluzione, verrà chiamato più avanti il nostro onActivityResult() con RESULT_OK
  - via startResolutionForResult()
  - via GooglePlayServicesUtil.getErrorDialog()
- A quel punto, vale la pena di riprovare la connect()
- In caso contrario… siamo alla disperazione, errore permanente!

GoogleAPIs

- Alcune delle API disponibili:
  - Search – è Google!
  - Ads – pubblicità e simili
  - Analytics – analisi di traffico
  - AppInvite – gestione beta tester
  - Cast – Chromecast e simili
  - Auth e Identity – identità degli utenti
  - Drive – storage condiviso
  - Fit – informazioni sull'attività fisica
  - Games – aspetti sociali dei giochi
  - Location, Maps, Panorama e Nearby – geocose
  - Plus – accesso a G+
  - Wallet – pagamenti elettronici
  - Wear – dispositivi indossabili

GoogleAPIs

- Trovate un elenco completo di API a https://developers.google.com/products/
- Alcune richiedono particolari relazioni commerciali, o che registriate la vostra applicazione sulla Developer Console, o non sono rilevanti per Android

Operazioni su GoogleAPI

- Le operazioni su GoogleAPI sono per loro natura **asincrone** e **fallibili**
  - Per forza: passano da rete, sotto c'è REST
- Tutte le chiamate vengono fatte tramite metodi statici di classi di libreria corrispondenti alle API
  - Hanno come parametro `gapi`
  - In molti casi, restituiscono un PendingResult
  - Sul PendingResult si può agire in tre modi
    - Registrando una **callback** da chiamare quando il risultato è pronto
    - **Sospendendo** il thread in attesa che il risultato sia pronto
    - **Cancellando** l'operazione (e testando se è stata cancellata)

Operazioni su GoogleAPI
Callback

- Come di consueto...
  - Si chiama **setResultCallback**(ResultCallback<R> rc) sul PendingResult
  - Quando l'operazione sarà conclusa, il risultato (di tipo R) verrà passato a rc
  - R estende Result
    - Al momento, sono definite 71 sottoclassi di Result nella libreria, con i diversi risultati specifici di diverse chiamate
    - Result fornisce di suo solo getStatus(), la quale restituisce un'oggetto Status
      - Status a sua volta ha un codice di successo (successo / interrotto / fallito / cancellato), una stringa che rappresenta un messaggio di stato, e un PendingIntent

Operazioni su GoogleAPI
Callback

- Esempio:

```
private void loadFile(String filename) {
    Query q = new Query.Builder()
        .addFilter(Filters.eq(SearchableField.TITLE, filename))
        .build();
    Drive.DriveApi.query(gapi, q)
        .setResultCallback(new ResultCallback() {
            public void onResult(DriveApi.MetadataBufferResult r) {
                MetadataBuffer mdb=r.getMetaDataBuffer();
                if (mdb!=null) {
                    for (Metadata md: mdb) {
                        int size=md.getFileSize();
                        String desc=md.getDescription();
                        ...
                    }
                    mdb.release();
                }
            }
    });
}
```

Di setResultCallback() esiste anche la versione con timeout: scaduto il tempo indicato, la chiamata si considera fallita.

Operazioni su GoogleAPI
Sospensione

- Sul PendingResult si chiama la **await**()
- Il thread chiamante viene bloccato finché il risultato non è disponibile
  - Quindi: **mai** chiamare sul thread UI!
- La await() restituisce il risultato
  - Di tipo R, sottoclasse di Result
  - Lo stesso oggetto che sarebbe passato a onResult()

Di await() esiste anche la versione con timeout: scaduto il tempo indicato, la chiamata si considera fallita.

Operazioni su GoogleAPI
Cancellazione

- Su un PendingResult non ancora completato si può chiamare la **cancel**()
  - Se era stata impostata una callback, questa viene chiamata con un parametro Result
  - Se un thread era bloccato su una await(), viene risvegliato e si ritorna un Result
- In entrambi i casi, il Result conterrà come status code l'indicazione dell'interruzione
- Il metodo **isCanceled**() di PendingResult indica se il PendingResult è stato cancellato o meno