ChatGPT

100 funzioni e funzionalità comuni per lo sviluppo di app Android in Java

50 Funzioni Java comuni nello sviluppo Android

1. **findViewById(int)**: Restituisce una `View` dal layout a partire dal suo ID, permettendo al codice di accedere e manipolare quell'elemento dell'interfaccia utente.
2. **setContentView(int)**: Imposta il layout (risorsa XML) da visualizzare in un' `Activity`, caricando e rendendo visibili le viste definite in quel file di layout.
3. **setOnClickListener(...)**: Registra un listener/callback su una `View` (es. un pulsante) da eseguire quando l'utente fa clic su di essa, così da gestire l'evento di click nel codice.
4. **Toast.makeText(...).show()**: Mostra un breve messaggio di feedback all'utente in un piccolo riquadro popup (toast) sullo schermo, che scompare automaticamente dopo una breve durata.
5. **AlertDialog.Builder**: Classe utilizzata per costruire e mostrare un dialogo di avviso (finestra popup) con titolo, messaggio e bottoni di azione, utile per chiedere conferme o notificare qualcosa all'utente.
6. **TextView.setText() / getText()**: Metodi per impostare (`setText`) o ottenere (`getText`) il testo da componenti UI testuali come `TextView` o `EditText`, usati rispettivamente per aggiornare il contenuto visualizzato o leggere input utente.
7. **RecyclerView e Adapter**: Componente UI avanzato per liste scrollabili ad alte prestazioni; visualizza elenchi di elementi riciclando le viste già create. Richiede un `Adapter` per fornire i dati e creare/collegare le singole celle dell'elenco in modo efficiente.
8. **Snackbar.make(...).show()**: Mostra un messaggio breve nella parte inferiore dello schermo (simile al Toast) con la possibilità di includere un pulsante di azione. Utile per fornire conferme (es. "Elemento eliminato" con opzione "Annulla").
9. **runOnUiThread(Runnable)**: Esegue un blocco di codice sul thread principale dell'app (UI thread). È utilizzato quando un thread in background necessita di aggiornare l'interfaccia utente, garantendo che le modifiche alle viste avvengano sul thread UI appropriato.
10. **getSharedPreferences(String name, int mode)**: Apre (o crea) un file di preferenze condivise identificato da un nome. Consente di memorizzare dati semplici in forma di coppie chiave-valore (ad es. impostazioni utente) in modo persistente all'interno della memoria dell'app.
11. **SharedPreferences.Editor.putX(...)**: Metodo dell'editor di SharedPreferences per salvare un valore (String, int, boolean, ecc.) associato a una chiave. Ad esempio `putString("chiave", "valore")` inserisce una stringa nelle preferenze. Le modifiche vanno confermate con `.apply()` o `.commit()`.
12. **SharedPreferences.getX(...)**: Metodo per leggere un valore dalle SharedPreferences dato il nome della chiave (ad es. `getString("chiave", defaultValue)`). Restituisce il valore salvato (o un valore di default se la chiave non esiste), permettendo di recuperare impostazioni o dati precedentemente memorizzati.
13. **SQLiteOpenHelper**: Classe di supporto per la gestione di database SQLite locali. Fornisce metodi callback `onCreate()` e `onUpgrade()` per creare o aggiornare il database alla prima apertura, semplificando l'accesso a SQLite e la gestione della versione del database.
14. **SQLiteDatabase.insert()/update()**: Metodi per inserire nuovi record (`insert`) o aggiornare record esistenti (`update`) in una tabella di un database SQLite. Si usano tipicamente passando il nome della tabella, i valori (tramite `ContentValues`) e, per update, una clausola WHERE per selezionare i record da modificare.

15. **SQLiteDatabase.query() & Cursor**: Metodo per eseguire query di selezione sul database SQLite. Restituisce un `Cursor` che punta ai risultati della query, permettendo di iterare sui record ottenuti e leggere le colonne desiderate (es. `cursor.getString(indiceColonna)`).
16. **Room (Jetpack)**: Libreria ORM di Jetpack che semplifica l'uso di SQLite. Consente di definire entità (classi di dati annotate) e DAO (interfacce con query SQL annotate) in Java, generando automaticamente il codice per le operazioni sul database. Facilita la persistenza locale evitando di scrivere query SQL manuali.
17. **openFileOutput(String, int) / openFileInput(String)**: Metodi del `Context` per scrivere su file interni all'app o leggerli. `openFileOutput` crea/apre un file nella memoria interna privata dell'app e restituisce un `OutputStream` per scriverci dati. `openFileInput` fa il contrario, restituendo un `InputStream` per leggere da un file precedentemente salvato.
18. **Accesso a memoria esterna**: Uso di API come `Environment.getExternalStorageDirectory()` (deprecated nelle versioni più recenti) o il più moderno **MediaStore** per leggere/scrivere file nella memoria esterna (es. scheda SD o storage condiviso). Questo consente di salvare o recuperare foto, documenti e altri file accessibili anche fuori dall'app, ma richiede permessi di storage (o l'uso del sistema di Storage Access Framework nelle versioni recenti di Android).
19. **ContentResolver.query(...)**: Metodo per accedere a dati esposti da un Content Provider. Consente di interrogare (query) URI pubblici (es. la rubrica contatti, le immagini della galleria, ecc.) e ottenere un `Cursor` con i risultati. In pratica, serve per leggere dati di altre app o del sistema (come contatti, media) tramite il loro provider.
20. **HttpURLConnection**: Classe Java standard (`java.net.HttpURLConnection`) per effettuare richieste HTTP verso un server. Permette di aprire una connessione a un URL, impostare metodo (GET, POST, ...), intestazioni, inviare dati (nel caso di POST) e leggere la risposta (stream di input) dal server. Utile per networking di base senza librerie esterne.
21. **OkHttpClient**: Libreria client HTTP efficiente fornita da Square (OkHttp). Consente di inviare richieste HTTP/HTTPS in modo semplice e performante, gestendo in modo integrato dettagli come caching, compressione e pooling delle connessioni. Si usa creando una richiesta (`Request`), quindi chiamando `OkHttpClient.newCall(request).execute()` (o enqueue per asincrono) per ottenere la risposta.
22. **Retrofit**: Libreria di alto livello per chiamate HTTP/REST. Permette di definire le API web come interfacce Java annotate (indicando endpoint, metodi GET/POST, parametri, ecc.) e genera automaticamente le implementazioni. Gestisce la serializzazione/deserializzazione JSON in oggetti Java (spesso con Gson) e semplifica l'esecuzione asincrona delle chiamate (es. con `enqueue` che fornisce callback onResponse).
23. **AsyncTask**: Classe (ora deprecata) che facilitava l'esecuzione di operazioni in background con possibilità di aggiornare la UI. Si sovrascrivono metodi come `doInBackground()` (operazioni in background su thread separato) e `onPostExecute()` (eseguito sul thread UI al termine per aggiornare interfaccia). Era comunemente usata per compiti brevi come chiamate di rete o lettura database senza bloccare il thread principale.
24. **Thread & Runnable**: Meccanismo di base per gestire operazioni in parallelo. Creare un nuovo `Thread` e fornirgli un oggetto `Runnable` (il cui metodo `run()` contiene il codice da eseguire) permette di avviare un'operazione in background. Utile per compiti concorrenti personalizzati, ad esempio calcoli intensivi o operazioni di I/O, assicurandosi poi di sincronizzare con il thread principale per aggiornare la UI.
25. **Handler.postDelayed(...)**: Utilizzo di un `Handler` (associato al Looper del thread principale o di un thread in background) per posticipare l'esecuzione di codice. Con `postDelayed(Runnable, delayMillis)` si può programmare un Runnable da eseguire dopo un certo ritardo. Inoltre, un Handler può inviare messaggi tra thread, utile per comunicare risultati dal background al thread UI.

26. **WorkManager**: Libreria Jetpack per pianificare task in background a lungo termine in modo affidabile. Permette di programmare lavori asincroni (anche periodici o con vincoli, ad es. solo su Wi-Fi o dispositivo in carica) che il sistema esegue garantitamente, anche se l'app viene chiusa o il dispositivo viene riavviato. È ideale per sincronizzazioni, upload differiti e operazioni che devono persistere oltre la vita dell'app in foreground.
27. **JSONObject & JSONArray**: Classi utili per lavorare con dati in formato JSON. `JSONObject` rappresenta un oggetto JSON (coppie chiave-valore), `JSONArray` un array di oggetti/valori JSON. Forniscono metodi per analizzare stringhe JSON (ad es. ottenere valori con `getString("nomeChiave")`, `getInt`, ecc.) e per costruire JSON in modo programmatico.
28. **Gson**: Libreria di Google per la serializzazione/deserializzazione JSON. Consente di convertire facilmente stringhe JSON in oggetti Java e viceversa, evitando parsing manuale. Si definiscono classi Java corrispondenti ai dati JSON e Gson si occupa di popolarle dagli attributi JSON o generare il JSON da un oggetto Java.
29. **WebView.loadUrl(...)**: Metodo per caricare e visualizzare contenuti web all'interno di un componente WebView. Permette di inserire una mini-browser nell'app per mostrare pagine HTML/JavaScript. Ad esempio `webView.loadUrl("https://sito.com")` renderizza quella pagina dentro l'app, utile per visualizzare contenuti online senza uscire dall'app.
30. **Glide/Picasso (caricamento immagini)**: Librerie ampiamente usate per caricare immagini da URL e visualizzarle in una `ImageView`. Gestiscono automaticamente operazioni costose come il download in background, caching delle immagini scaricate, e resizing per adattarle alla view, evitando di bloccare il thread principale. Esempio: `Glide.with(context).load(url).into(imageView)`.
31. **NotificationCompat.Builder**: Classe (della support library AndroidX) per creare notifiche in modo compatibile con diverse versioni di Android. Si impostano proprietà come titolo, testo, icona piccola, priorità, azioni, ecc., dopodiché con `build()` si ottiene una notifica pronta da inviare tramite NotificationManager.
32. **NotificationManager.notify(int, Notification)**: Metodo che pubblica la notifica costruita nel sistema Android. Accetta un ID univoco (int) per identificare la notifica e l'oggetto `Notification`. Mostra la notifica nell'area notifiche a tendina e nella barra di stato. L'ID può essere usato successivamente per aggiornare o rimuovere la notifica.
33. **createNotificationChannel(...)**: Metodo richiesto da Android 8.0 (Oreo) in poi per impostare un canale di notifica prima di inviare notifiche di un certo tipo. Un NotificationChannel definisce importanza, suoni, vibrazione ecc. per gruppi di notifiche. Deve essere creato e registrato una volta (di solito all'avvio dell'app) tramite NotificationManager, dopodiché le notifiche inviate con quel channel ID seguiranno quelle impostazioni.
34. **PendingIntent**: Oggetto che incapsula un `Intent` destinato ad essere eseguito in futuro da un'altra applicazione (tipicamente Android system). Si usa spesso nelle notifiche: ad esempio un PendingIntent creato con `PendingIntent.getActivity()` punta a un'Activity specifica dell'app e viene attivato dal sistema quando l'utente tocca la notifica, aprendo così l'app. Si utilizza anche con AlarmManager o widget per azioni posticipate.
35. **AlarmManager**: Manager di sistema che consente di programmare l'esecuzione futura di un'azione. Può attivare un `PendingIntent` ad un orario preciso o a intervalli ricorrenti (es. sveglie, notifiche a orari programmati, esecuzione periodica di codice). Supporta allarmi esatti o approssimati e operazioni ripetute (ma su Android recenti ha limitazioni per il risparmio energetico).
36. **FusedLocationProviderClient**: API fornita da Google Play Services per ottenere la posizione geografica in modo semplice ed efficiente. Combina in modo "fuso" (fused) diverse fonti (GPS, reti cellulari, Wi-Fi) per fornire la posizione attuale ottimizzata. Permette di richiedere aggiornamenti periodici di posizione o l'ultima posizione nota tramite metodi come `getLastLocation()` o `requestLocationUpdates()`.

37. **LocationManager**: API nativa di Android per accedere ai servizi di localizzazione (parte del framework Android). Consente di richiedere aggiornamenti GPS o di rete tramite `requestLocationUpdates()` e di ottenere la posizione corrente. Si possono impostare criteri di precisione e intervallo e fornire un `LocationListener` che riceve le coordinate man mano che vengono rilevate.
38. **Geocoder**: Classe che effettua la geocodifica e la reverse-geocodifica. In pratica converte indirizzi testuali in coordinate geografiche (latitudine/longitudine) e viceversa. Utile, ad esempio, per ottenere un indirizzo leggibile (via, città) a partire da una posizione GPS dell'utente, oppure trovare coordinate di un indirizzo inserito dall'utente.
39. **Google Maps API (MapView/MapFragment)**: Permette di integrare le mappe di Google nell'app. Usando un `MapView` o `SupportMapFragment`, l'app può mostrare una mappa interattiva su cui è possibile disegnare marker (punti di interesse), linee, poligoni o tracciare la posizione dell'utente. Richiede la configurazione di una Google Maps API Key e offre controlli integrati per zoom, panoramica, ecc.
40. **ContextCompat.checkSelfPermission(...)**: Metodo per verificare se la nostra app ha uno specifico permesso runtime già concesso. Ad esempio `checkSelfPermission(context, Manifest.permission.CAMERA)` restituisce `PERMISSION_GRANTED` se l'utente ha già autorizzato l'uso della fotocamera, oppure `PERMISSION_DENIED` in caso contrario. È usato per decidere se mostrare la richiesta di permesso.
41. **ActivityCompat.requestPermissions(...)**: Metodo per richiedere all'utente uno o più permessi runtime. Genera la comparsa di una finestra di dialogo di sistema in cui l'utente può consentire o negare i permessi (es. posizione, fotocamera). Si passa l'Activity corrente, un array di permessi (stringhe di Manifest) e un requestCode; la risposta dell'utente arriverà nel metodo callback onRequestPermissionsResult().
42. **onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults)**: Metodo di callback dell'Activity che viene invocato dopo che l'utente ha risposto a una richiesta di permessi. Fornisce il requestCode originale, l'array di permessi richiesti e un array di risultati (grantResults) indicanti per ogni permesso se è stato concesso o negato. In questo metodo si implementa la logica per gestire il caso in cui il permesso venga negato (ad es. disabilitando funzionalità) o concesso (procedendo ad usare la funzionalità protetta dal permesso).
43. **getSystemService(...)**: Metodo del `Context` che permette di ottenere istanze dei servizi di sistema di Android. Ad esempio `getSystemService(LAYOUT_INFLATER_SERVICE)` restituisce un LayoutInflater per "gonfiare" viste da XML, `getSystemService(SENSOR_SERVICE)` dà il SensorManager per accedere ai sensori hardware, `getSystemService(VIBRATOR_SERVICE)` dà il Vibrator per far vibrare il dispositivo, e così via per molti servizi (Location, Notification, Connectivity...).
44. **ConnectivityManager**: Servizio di sistema recuperabile via `getSystemService(CONNECTIVITY_SERVICE)`. Consente di ottenere info sullo stato della connessione di rete (se c'è connettività, se è Wi-Fi o mobile, ecc.). Tramite i suoi metodi (come `getActiveNetworkInfo()` su versioni più vecchie o le API NetworkCapabilities più recenti) si può capire se l'app è online, ricevere notifiche di cambiamento rete, o avviare richieste specifiche su una rete.
45. **MediaPlayer**: Classe per la riproduzione di contenuti multimediali audio/video. Permette di caricare un file audio/video da risorsa, file o URL (`setDataSource`), prepararlo (`prepare()` o `prepareAsync()`), e riprodurlo (`start()`). Supporta controlli come `pause()`, `stop()`, `seekTo()`, e listener per eventi di completamento o errori. Utile per aggiungere musica di sottofondo, riproduzione di brani, effetti sonori, ecc.
46. **startActivityForResult(...)**: Metodo usato per avviare un'`Activity` quando ci si aspetta che quest'ultima restituisca un risultato. Si passa un Intent e un requestCode; l'Activity di destinazione partirà e potrà al termine restituire dati. Ad esempio, si può chiamare la fotocamera

di sistema per scattare una foto o aprire i contatti per selezionarne uno, attendendo un risultato che arriverà nel callback onActivityResult del chiamante.

47. **onActivityResult(int requestCode, int resultCode, Intent data)**: Metodo di callback chiamato quando un'Activity avviata con startActivityForResult termina e restituisce un risultato. Nel metodo, usando il requestCode si identifica quale richiesta è conclusa, si verifica se resultCode indica successo, quindi si prelevano i dati dall'Intent `data` (ad es. l'URI della foto scattata, o i dettagli di un contatto selezionato) per poi utilizzarli nell'app.
48. **Intent (fotocamera)**: Utilizzo di un Intent implicito `MediaStore.ACTION_IMAGE_CAPTURE` per avviare l'app Fotocamera di sistema. Si costruisce l'Intent magari specificando dove salvare la foto, quindi con `startActivityForResult` si lascia che l'utente scatti una foto. L'immagine catturata sarà restituita all'app (come thumbnail nel `data` Intent, o salvata in un percorso fornito) permettendo di usarla, ad esempio, per impostare un avatar o salvare una foto.
49. **Intent (galleria)**: Utilizzo di un Intent implicito con azione `ACTION_PICK` o `ACTION_GET_CONTENT` sul contenuto immagini (`MediaStore.Images.Media.EXTERNAL_CONTENT_URI`) per aprire l'app Galleria/selezione documenti. In questo modo l'app permette all'utente di scegliere una foto o un video già presente sul dispositivo. Il risultato della selezione (URI del file scelto) viene poi restituito in onActivityResult per poterlo visualizzare o elaborare.
50. **BroadcastReceiver**: Componente atto a ricevere **broadcast** di Intent inviati dal sistema o dall'app stessa. Ad esempio, Android lancia broadcast per eventi come cambio della connettività, batteria quasi esaurita, avvio completato del dispositivo, ricezione di SMS, ecc. Un BroadcastReceiver registrato (nel Manifest o dinamicamente) intercetta l'evento e permette di eseguire del codice in risposta (anche quando l'app non è visibile), abilitando reazioni a eventi globali del sistema o comunicazione tra applicazioni.

50 Funzionalità implementabili comuni nelle app Android

1. **Registrazione utente**: Funzionalità che consente all'utente di creare un nuovo account nell'app, solitamente fornendo dati come nome, email e password. L'app salva queste credenziali (in un database remoto o locale) per permettere all'utente di autenticarsi e identificarsi in futuro.
2. **Login utente con credenziali**: Permette a un utente già registrato di accedere all'app usando username/email e password. L'app verifica le credenziali inserite confrontandole con quelle registrate (magari tramite un server) e, se corrette, sblocca le funzionalità riservate all'utente autenticato.
3. **Login con Google**: Opzione di autenticazione che consente all'utente di loggarsi tramite il proprio account Google. L'app integra il servizio Google Sign-In, evitando di creare un nuovo nome utente/password e sfruttando le credenziali Google esistenti per velocizzare l'accesso.
4. **Login con Facebook**: Simile al login con Google, questa funzionalità permette all'utente di autenticarsi utilizzando il proprio account Facebook. L'app reindirizza verso l'SDK di Facebook per il login e riceve i dati di profilo autorizzati, facilitando l'accesso senza registrazione manuale.
5. **Autenticazione biometrica**: Utilizzo di sensori biometrici del dispositivo (come impronta digitale o riconoscimento facciale) per consentire accessi rapidi e sicuri. Ad esempio, l'utente può sbloccare l'app o confermare operazioni sensibili tramite impronta, invece di reinserire una password.
6. **Acquisti in-app**: Integrazione del sistema di acquisti digitali di Google Play all'interno dell'app. Consente agli utenti di comprare contenuti o funzionalità premium (come livelli aggiuntivi, rimuovere pubblicità, abbonamenti) direttamente nell'app. L'implementazione gestisce il processo di pagamento tramite Play Store e fornisce conferma all'app sull'esito dell'acquisto.
7. **Pagamenti tramite servizi esterni**: Supporto a pagamenti gestiti fuori dal circuito dello store, ad esempio tramite PayPal, Stripe o inserimento di carta di credito in-app. Utile per app di e-

commerce o servizi che vendono beni fisici o abbonamenti multi-piattaforma: l'utente può effettuare transazioni di denaro in sicurezza all'interno dell'app verso un servizio esterno.

8. **Pubblicità in-app**: Inserimento di annunci pubblicitari nell'app per monetizzare. Può trattarsi di banner (piccoli annunci statici o animati in una parte dello schermo), interstitial (annunci a tutto schermo mostrati in transizioni), o video reward (video pubblicitari che l'utente può scegliere di guardare in cambio di una ricompensa nell'app). Generalmente gestiti tramite network come Google AdMob integrando i rispettivi SDK.
9. **Notifiche push**: Sistema per inviare notifiche remote agli utenti. L'app si registra a un servizio di push (come Firebase Cloud Messaging) e può ricevere messaggi dal server anche quando non è attiva. Ciò permette, ad esempio, di notificare l'utente di nuovi messaggi, aggiornamenti, promozioni o eventi importanti attraverso le classiche notifiche Android nell'area dedicata.
10. **Sincronizzazione in background**: Meccanismo che mantiene i dati dell'app aggiornati sincronizzandoli periodicamente con un server, senza intervento dell'utente. Può avvenire tramite servizi di background o WorkManager, garantendo che l'app invii/riceva aggiornamenti (es. nuove email, feed di notizie) anche quando non è aperta, rispettando limiti di batteria e rete.
11. **Modalità offline**: Capacità dell'app di funzionare (almeno parzialmente) senza connessione Internet. I dati necessari vengono salvati in cache o in un database locale, così l'utente può visualizzarli o usarli offline. Eventuali modifiche fatte offline (es. messaggi inviati, form compilati) verranno sincronizzate con il server alla riconnessione.
12. **Barra di navigazione inferiore**: Un menu di navigazione posizionato in fondo allo schermo (Bottom Navigation Bar) contenente tipicamente icone o etichette per le sezioni principali dell'app. Toccando le voci della barra, l'utente può passare rapidamente da una sezione all'altra (es. Home, Ricerca, Profilo, Impostazioni) con un'esperienza di navigazione coerente.
13. **Menu di navigazione laterale**: Conosciuto anche come Navigation Drawer, è un menu a scomparsa che scorre dal lato sinistro o destro dello schermo. Contiene voci di menu aggiuntive o impostazioni (spesso troppe per stare in una barra inferiore). L'utente lo apre toccando un'icona hamburger ≡ o tramite swipe laterale, e vi trova opzioni di navigazione secondarie, sezioni meno usate o informazioni account.
14. **Modalità scura (Dark Mode)**: Tema grafico alternativo che applica colori scuri all'interfaccia (sfondi neri o grigio scuro e testo chiaro). Molte app offrono la possibilità di passare dal tema chiaro a quello scuro manualmente o in base alle impostazioni di sistema/orario. La modalità scura riduce l'affaticamento visivo in ambienti bui e può diminuire il consumo energetico su schermi OLED.
15. **Schermate di onboarding**: Pagine introduttive mostrate al primo avvio dell'app (o alla prima apertura dopo un aggiornamento significativo). Di solito sono una serie di slide con testi e illustrazioni che presentano le caratteristiche principali dell'app o ne spiegano il funzionamento, aiutando l'utente a capire il valore e richiedendo eventualmente permessi iniziali (come accesso a GPS o notifiche) in modo guidato.
16. **Schermata di splash**: Schermata iniziale visibile durante il caricamento dell'app, spesso contenente il logo o il nome dell'app su sfondo semplice. Serve a dare un feedback immediato all'utente mentre le risorse dell'app vengono caricate in memoria all'avvio. Generalmente viene mostrata per pochi secondi prima di passare alla schermata principale.
17. **Supporto multilingua**: Localizzazione dell'app in più lingue. Tutti i testi dell'interfaccia (etichette, messaggi, bottoni) sono tradotti nelle lingue previste e caricati in base alle impostazioni del dispositivo (o a una scelta manuale nell'app). Ciò rende l'app fruibile da utenti di diverse regioni, adattando anche formati di data/ora, numeri e valute dove necessario.
18. **Supporto accessibilità**: Adeguamento dell'app per utenti con disabilità. Include elementi come: descrizioni testuali (content description) per elementi grafici, in modo che i screen reader (es. TalkBack) possano enunciarli; possibilità di navigare l'app tramite tastiera o comandi vocali; contrasto elevato e dimensioni font regolabili per ipovedenti; evitamento di contenuti

lampeggianti per utenti fotosensibili. Queste accortezze garantiscono che anche utenti con necessità speciali possano utilizzare l'app.

19. **Funzione di ricerca interna**: Implementazione di una barra o di una schermata di ricerca che permetta all'utente di trovare rapidamente contenuti all'interno dell'app. Ad esempio cercare articoli in un'app di news, prodotti in un'app di shopping o brani in un'app musicale. La funzionalità di solito include suggerimenti durante la digitazione e filtro dei risultati in base alla query inserita.
20. **Pull-to-refresh**: Gesto standard per aggiornare i contenuti. L'utente, trovandosi su una lista (ad es. elenco di news o feed social), può trascinare verso il basso lo schermo oltre il primo elemento e rilasciare, attivando un refresh manuale. Un indicatore di caricamento compare brevemente e l'app effettua un aggiornamento (ad es. richiesta al server per nuovi dati) mostrando poi i nuovi contenuti in cima alla lista.
21. **Paginazione/infinite scroll**: Gestione efficiente di liste molto lunghe caricando i dati in modo graduale. Invece di caricare centinaia di elementi tutti insieme, l'app ne mostra ad esempio 20 e poi carica altri elementi man mano che l'utente scorre verso il fondo (infinite scroll). In alternativa, con paginazione esplicita l'utente può premere un bottone "Carica altro" o navigare tra pagine di risultati. Ciò migliora le performance e l'esperienza utente in elenchi estesi.
22. **Condivisione di contenuti**: Permette all'utente di condividere testo o file dall'app verso altre app installate (ad esempio condividere un articolo via WhatsApp, email, Facebook, etc.). Di solito realizzato invocando l'intent di sistema di condivisione (`Intent.ACTION_SEND`) che apre il pannello di condivisione Android, mostrando le app disponibili e passando ad esse il contenuto (link, testo, immagini) selezionato dall'utente.
23. **Funzionalità di chat in tempo reale**: Integrazione di un sistema di messaggistica istantanea all'interno dell'app. Gli utenti possono inviare e ricevere messaggi testuali (e spesso emoji, immagini, vocali) in tempo reale, con aggiornamenti immediati dell'interfaccia quando arrivano nuovi messaggi. Può essere implementata usando servizi come WebSocket o Firebase Realtime Database/Firestore per notificare i nuovi messaggi istantaneamente a tutti i client connessi.
24. **Integrazione di mappe**: Inclusione di una mappa interattiva nell'app, tipicamente tramite Google Maps SDK. Consente di mostrare all'utente mappe navigabili, eventualmente con la propria posizione corrente indicata. L'app può aggiungere marker/pin su luoghi specifici, tracciare itinerari o zone evidenziate. Utile per funzionalità come mostrare indirizzi di punti vendita, fornire indicazioni stradali, visualizzare utenti/oggetti su una mappa, ecc.
25. **Utilizzo della localizzazione GPS**: Funzionalità che sfrutta il GPS (e altre fonti di localizzazione) per ottenere la posizione geografica dell'utente e offrire servizi contestuali. Ad esempio un'app di consegna cibo usa la posizione per conoscere l'indirizzo dell'utente, un social network può geotaggare i post, un'app meteo mostra il meteo locale. Richiede la gestione dei permessi runtime di localizzazione e l'uso delle API Location.
26. **Geofencing**: Implementazione di "recinti geografici", ovvero aree virtuali definite da coordinate GPS e raggio. L'app registra uno o più geofence e può ricevere notifiche dal sistema quando l'utente entra o esce da queste aree. Ciò permette funzionalità come: inviare una notifica "Benvenuto nel negozio X" quando l'utente entra in prossimità di un punto vendita, o attivare/disattivare automaticamente una funzione (es. modalità silenziosa) in certe zone.
27. **Accesso alla fotocamera**: Integrazione che permette all'app di utilizzare la fotocamera del dispositivo. Ad esempio per scattare foto, registrare video o effettuare scansioni di documenti/codici. Può essere realizzato aprendo l'app Fotocamera nativa tramite Intent e recuperando il risultato, oppure incorporando direttamente l'uso della fotocamera nell'app (via Camera API/CameraX) per maggior controllo (es. un'app di fotografia con filtri in tempo reale).
28. **Scansione codici QR**: Funzionalità che attiva la fotocamera e decodifica i QR code (o codici a barre). Molte app la includono per permettere all'utente di, ad esempio, aggiungere amici inquadrando un loro codice QR, effettuare pagamenti, riscattare coupon o ottenere informazioni/scaricare app tramite QR. Si implementa usando librerie specializzate (come ZXing,

ZBar o Google ML Kit) che analizzano il frame video della fotocamera alla ricerca di codici e restituiscono il contenuto testuale codificato.

29. **Selezione immagini dalla galleria**: Consente all'utente di scegliere una foto o un video già presente sul dispositivo per utilizzarlo nell'app (ad esempio impostare un'immagine di profilo, allegare una foto a un post/messaggio, etc.). Si realizza tipicamente aprendo un selettore multimediale via Intent (come ACTION_PICK sul content URI delle immagini) e gestendo l'URI del file selezionato per poi caricarlo o inviarlo al server.
30. **Riproduzione video integrata**: L'app offre la possibilità di riprodurre video direttamente al suo interno, anziché aprirli in app esterne. Ciò implica avere un player video integrato (spesso implementato con VideoView o librerie come ExoPlayer) che supporti controlli play/pausa, barra di avanzamento, eventualmente fullscreen, e gestisca flussi streaming o file locali. Molto comune in app di media, social (per video nel feed) o e-learning.
31. **Riproduzione audio**: Simile alla precedente ma per contenuti audio. L'app può riprodurre musica, podcast, note vocali o effetti sonori. Include controlli come play/pausa, skip, e gestisce l'audio in background (es. continua a suonare anche se l'utente passa ad altra app, con notifica di controllo multimedia). Spesso implementato tramite `MediaPlayer` o librerie come ExoPlayer per audio.
32. **Registrazione audio**: Permette all'utente di registrare suoni o voce tramite il microfono del dispositivo. Ad esempio un'app di messaggistica consente di inviare messaggi vocali, un'app per appunti vocali registra memo audio, un'app di musica registra canto/chitarra. Si implementa tramite `MediaRecorder` o nuove API (Jetpack) assicurandosi di gestire i permessi del microfono. Il file audio risultante può essere salvato localmente o inviato a un server.
33. **Connettività Bluetooth**: Funzionalità che abilita l'app a interfacciarsi con dispositivi Bluetooth. Può includere la scansione e l'abbinamento con dispositivi vicini (es. cuffie, smartwatch, dispositivi IoT), la trasmissione di dati o comandi a un device (es. controllo di un robot/Bluetooth toy) o la lettura di dati da sensori BLE (come cardiofrequenzimetri, beacon). Richiede permessi Bluetooth e, per BLE, spesso localizzazione.
34. **Integrazione NFC**: Permette di utilizzare la comunicazione Near Field Communication. Ad esempio leggere tag NFC (adesivi, badge, carte) avvicinando il telefono, per ottenere informazioni o autenticare l'accesso a un servizio; oppure effettuare pagamenti contactless via NFC (integrando servizi tipo Google Pay). L'app registra un filtro per tipi di tag/supporta gli standard NFC e poi reagisce quando il dispositivo NFC legge un tag compatibile mentre l'app è in primo piano.
35. **Uso di sensori del dispositivo**: Accesso e utilizzo di sensori come accelerometro, giroscopio, sensore di prossimità, luce ambientale, bussola, ecc., per arricchire l'esperienza utente. Ad esempio, usare l'accelerometro per conteggiare passi o rilevare movimenti (shake per aggiornare), il giroscopio per controllare un gioco con il movimento del telefono, il sensore di prossimità per spegnere lo schermo durante una chiamata VoIP, o la bussola per un'app di mappe che mostra l'orientamento.
36. **Funzionalità di realtà aumentata (AR)**: Integrazione di AR nell'app, usando la fotocamera e i sensori per sovrapporre elementi digitali al mondo reale. Esempi: un'app di arredamento che mostra modelli 3D di mobili nella stanza inquadrata, filtri facciali in app di foto/selfie, giochi in AR (come Pokémon GO) dove oggetti virtuali compaiono nell'ambiente ripreso dalla fotocamera. Si realizza con librerie/piattaforme come ARCore (Google) o ARKit (iOS) per il tracking dell'ambiente e il rendering di contenuti 3D.
37. **Widget per la schermata Home**: Fornire un widget collegato all'app che l'utente può aggiungere nella home del dispositivo Android. Il widget mostra informazioni aggiornate o controlli rapidi senza aprire l'app (ad es. widget meteo con temperatura aggiornata, widget player musicale con pulsanti play/pausa, o calendario con prossimi eventi). Richiede la definizione di un layout e una classe `AppWidgetProvider` che gestisce l'aggiornamento periodico dei contenuti.

38. **Modalità Picture-in-Picture (PiP)**: Supporto alla modalità multi-finestra in cui l'app, quando l'utente esce (es. premendo home), continua a mostrare un piccolo riquadro flottante sullo schermo con un contenuto in esecuzione (tipicamente video o navigazione GPS). L'utente può così guardare un video in un angolino mentre usa altre app. L'implementazione richiede l'uso delle API PiP (disponibili da Android 8.0) e l'app deve gestire la riduzione a mini-player e il ripristino a schermo intero.
39. **Funzionalità di Drag & Drop**: Consente all'utente di trascinare e rilasciare elementi dell'interfaccia. Può essere interno all'app (ad es. riordinare una lista trascinando elementi su o giù, spostare eventi nel calendario) o tra app diverse in split-screen (trascinare testo/immagini da un'app all'altra, se supportato). Android offre API per avviare un drag (`startDragAndDrop`) su una view e rilevare il drop in un'altra, permettendo interazioni più naturali con contenuti trascinabili.
40. **Gesture di zoom (pinch-to-zoom)**: Implementazione del classico gesto multi-touch di "pizzicare" con due dita per zoomare. Usato comunemente su immagini e mappe all'interno dell'app: l'utente può ingrandire o ridurre i dettagli avvicinando o allontanando pollice e indice sullo schermo. Si ottiene gestendo gli eventi touch multipoint o utilizzando componenti che supportano lo zoom (es. PhotoView per immagini).
41. **Deep linking**: Configurazione di link esterni (URL) che aprono direttamente l'app verso una specifica schermata o contenuto. Ad esempio, cliccando su un link www.mioecommerce.com/prodotto/123 su mobile, si apre direttamente l'app di ecommerce (se installata) mostrando la pagina di quel prodotto. Si realizza definendo intent filter nel Manifest per gli URL di destinazione e gestendo i parametri ricevuti per navigare alla sezione appropriata nell'app.
42. **Richiesta di valutazione dell'app**: Meccanismo per incoraggiare l'utente a lasciare una recensione positiva sullo store. Tipicamente dopo un certo periodo di utilizzo o dopo un evento soddisfacente (es. completamento di un livello, acquisto riuscito) l'app mostra un pop-up che chiede di valutare l'app. Se l'utente accetta, viene indirizzato alla pagina dello store per dare stelle e commenti. Implementabile tramite API ufficiali (es. In-App Review API di Google Play) o aprendo direttamente l'intent del market.
43. **Funzione di preferiti/segnalibri**: Permette all'utente di salvare elementi nell'app per ritrovarli facilmente in seguito. Può trattarsi di articoli in un'app di news (lista "segna come preferito"), prodotti in un'app di shopping (wishlist), video in un'app di streaming ("Aggiungi a Guarda più tardi"), ecc. L'app offre un pulsante/icona (es. a forma di stella o cuore) per salvare/rimuovere dai preferiti e una sezione dove visualizzare tutti gli elementi salvati.
44. **Gestione profilo utente**: Una sezione dedicata in cui l'utente può vedere e modificare i propri dati personali e impostazioni. Ad esempio, l'utente può aggiornare la foto profilo, cambiare nome visualizzato, email, password, oppure impostare preferenze (notifiche on/off, privacy) relative al proprio account. Questa funzionalità include spesso la visualizzazione dei dettagli account e opzioni come disconnettersi o eliminare l'account.
45. **Integrazione di analitiche**: Inclusione di un servizio di analytics per raccogliere dati sull'uso dell'app. Lo sviluppatore integra SDK come Google Firebase Analytics, Mixpanel o similari, e l'app invia eventi (es. "schermata X aperta", "bottone Y cliccato", "acquisto effettuato") a tali servizi. I dati aggregati aiutano a comprendere il comportamento degli utenti, quali funzionalità sono più usate, dove migliorare e misurare il successo di funzionalità o campagne.
46. **Monitoraggio crash e errori**: Implementazione di strumenti di crash reporting (ad esempio Firebase Crashlytics, Sentry) nell'app. In caso di arresto anomalo (crash) o errori gravi, l'SDK cattura automaticamente stack trace e contesto dell'errore e lo invia in background allo sviluppatore. Questo aiuta a individuare bug e problemi su diversi dispositivi e sistemi operativi e a migliorarne la stabilità nelle versioni successive.
47. **Protezione con PIN/impronta**: Funzionalità di sicurezza che permette all'utente di impostare un codice PIN o password numerica, oppure di usare l'impronta digitale/biometria, per accedere all'app o a specifiche sezioni sensibili. Ad esempio, un'app di note può proteggere note private

richiedendo un PIN, oppure un'app bancaria richiede l'impronta per aprirsi. Ciò aggiunge un livello extra di protezione oltre all'eventuale login.

48. **Chiamate audio/video VoIP**: Funzionalità che consente agli utenti di effettuare chiamate vocali o video chiamate direttamente nell'app via Internet. Viene implementata usando protocolli VoIP/WebRTC: l'app gestisce la connessione audio/video tra due o più utenti, permettendo conversazioni in tempo reale con streaming del microfono e della fotocamera. Tipico di app di messaggistica, incontri online o collaborazione, dove gli utenti possono parlare o vedersi senza usare la rete telefonica tradizionale.
49. **Accesso ai contatti del dispositivo**: L'app, con il permesso dell'utente, legge la rubrica dei contatti sul telefono per funzioni come: trovare altri utenti che utilizzano la stessa app (matching dei numeri/email), invitare amici ad usare l'app tramite SMS/WhatsApp, o auto-compilare campi di nominativi/numero in funzionalità di condivisione. È una funzionalità comune nei social network e app di comunicazione per aiutare l'utente a connettersi con conoscenti già presenti.
50. **Integrazione calendario**: Funzione che consente all'app di interagire con il calendario di sistema. Ad esempio un'app di produttività può creare un evento nel calendario del telefono per ricordare una scadenza o sincronizzare gli appuntamenti dell'utente; oppure un'app di prenotazione può aggiungere al calendario un evento per la prenotazione effettuata. Viceversa, l'app potrebbe leggere eventi esistenti (con permesso) per, ad esempio, evitare conflitti di orario o mostrare all'utente i propri impegni direttamente nell'app.
