Riassunto Applicazioni Mobili

Hardware Abstraction Layer (HAL) Strato software tra il kernel Linux e il resto della piattaforma Android. Nasconde i dettagli dell'hardware, esponendo API per uniformare l'accesso al basso livello del dispositivo. Raggruppa i dispositivi per tipologia, permettendo di gestirli correttamente in base al tipo. In base all'hardware utilizzato, HAL si occupa di caricare le librerie corrette.

Spetta ai produttori hardware implementare la HAL, mettendo a disposizione degli sviluppatori software le stesse interfacce e funzioni.

Android Runtime (ART) Implementazione specifica della Java Virtual Machine (JVM) per Android, ottimizzata per dispositivi con vincoli di memoria.

Interpreta il codice Java in Bytecode (APK), assieme a risorse e librerie. In seguito interpreta il Bytecode in codice macchina.

Utilizza due tipi di compilazione:

- Ahead of time (AOT): ART compila l'app al momento dell'installazione, in modo da averla già precompilata quando dovrà eseguirla.
- Just in time (JIT): ART compila le parti più critiche del codice durante l'esecuzione, per tenerle sempre aggiornate.

ART una un daemon per controllare e compilare periodicamente app non precompilate.

Dalvik Implementazione della JVM prima di ART (fino ad Android KitKat). Compilava JIT i file DEX bytecode al momento dell'esecuzione dell'app. ART invece dispone di uno stack più capiente, migliore gestione degli errori, ottimizzazione del garbage collector e usa compilazione AOT per prestazioni migliori.

Componenti Punti di accesso all'app, dichiarati nel manifest.

1. Activity: singola schermata con logica e UI;
2. Service: esegue operazioni in background, senza UI;
3. Broadcast Receiver: rimane in ascolto di eventi, li riceve ed esegue brevi operazioni in base ad essi;
4. Content provider: gestisce l'accesso ai dati tra app diverse.

Ogni componente ha un lifecycle definito, ed è attivabile tramite intent[1]. Il sistema gestisce il loro ciclo di vita e possono essere terminati in ordine di priorità se sono necessarie risorse. Ogni componente viene eseguito in un processo separato (sandbox) come utente Linux.

Activity Schermata con cui l'utente interagisce (entrypoint) con un layout proprio. Un'applicazione può avere più activities che si susseguono nella navigazione. Un'activity può attivarne un altra tramite intent. Come gli altri componenti, ha un lifecycle.

1. `onCreate()`: si occupa di creare l'activity, ma non la rende ancora visibile;
2. `onStart()`: rende visibile l'activity, ma non ci si può ancora interagire;
3. `onResume()`: l'activity è visibile e interattiva (running). Solo un'activity alla volta può essere in questo stato;
4. `onPause()`: l'activity è solo parzialmente visibile, interrotta da qualcosa. Chiama `onResume()` per riprendere l'esecuzione;
5. `onStop()`: l'activity non è più visibile. Viene chiamato quando l'activity sta per essere distrutta o un altro componente richiede interamente il primo piano;
6. `onRestart()`: un'activity stopped viene riattivata chiamando `onStart()`
7. `onDestroy()`: l'activity viene distrutta, liberando memoria. Avviene quando cambia il layout dello schermo o il SO ha bisogno di spazio.

[1] Tranne i Content Provider.

Dato che Android può terminare l'activity in un qualsiasi momento, è importante avere un meccanismo per gestire i dati.
Nel ciclo di vita di un'activity sono individuabili 3 loop:
Entire Lifetime Tra `onCreate()` e `onDestroy()` ⟶ esiste;
Visible Lifetime Tra `onStart()` e `onStop()` ⟶ visibile;
Foreground Lifetime Tra `onResume()` e `onPause()` ⟶ interattiva.

Nella stessa app , un'activity può essere rimpiazzata da un'altra. La precedente viene salvata nel backstack. Per ripristinare l'activity precedente, quella corrente viene rimossa (`pop()`) dal backstack e distrutta.

Fragment Porzione della UI in un'activity, riutilizzabile in più activity. Ha un lifecycle gestito dal FragmentManager, subordinato a quello dell'activity che lo contiene. Possono esserci più fragment attivi e visibili nella stessa activity. I fragment possono essere aggiunti o rimossi durante l'esecuzione della activity. Ogni set di cambiamenti all'activity è chiamato transaction, salvato in un backstack per permettere la navigazione a ritroso. Il backstack delle activity è mantenuto dal sistema, mentre quello dei fragment è mantenuto dalla activity che li contiene.

Portrait e Landscape Mode Quando si ruota il telefono, Android distrugge l'activity e la ricrea per adattare le risorse alla configurazione. Lo stesso vale quando si cambia lingua o tema.
1. Distrugge l'activity: `onPause()` ⟶ `onStop()` ⟶ `onDestroy()`;
2. Ricrea l'activity `onCreate()` ⟶ `onStart()` ⟶ `onResume()`.

Si tiene traccia della dei dati in un bundle chiamato instance state dell'activity. Con esso, Android tiene traccia dello stato di ogni view.
1. Prima di essere stoppata , l'activity salava il bundle (override di `onsaveInstanceState()`);
2. Prima di essere riavviata, l'activity carica i dati (override di `onrestoreInstanceState()`).

Questo metodo è pensato per la persistenza di piccoli dati. Per quantità di dati maggiori si utilizza un ViewModel, che sopravvive ai cambi di configurazione e permette una gestione migliore.

Activity Backstack e Task In un'app, le activities sono memorizzate una sopra l'altra, e per crearne una si usano gli intent. Quando l'utente preme "indietro", Android preleva l'elemento precedente dallo stack delle activity eliminando il primo. Questa operazione è fatta dal ActivityManagerService. Quando una nuova attività è avviata, viene messa in cima allo stack. Quella sottostante esiste ancora ma è congelata. Navigando a ritroso si esegue un pop delle activity. Se si preme indietro nella root activity, quindi quando lo stack è vuoto, l'app viene terminata su Android ≤ 11, mentre su Android ≥ 12 il task va in background.
Un task è una collezione (backstack) di activity correlate. Un task può essere in primo piano se la sua activity in cima è "running", altrimenti è in background viene mostrato nella UI "recent activities". Una singola app può avere più task distinti, ciascuno con il proprio backstack.

View Classe base di tutti i componenti grafici della UI. È l'oggetto che rappresenta e gestisce gli eventi di un elemento visivo sullo schermo, come un pulsante, un campo di testo o un'immagine. Le view sono il paradigma per la costruzione di una ui responsiva in Android.
Le view possono essere create in 2 modi:
1. Declarative mode: dichiarandole nell'XML e accedendovi da codice tramite il loro id;
2. Programmatic mode: creando la view direttamente con codice kotlin.

L'handling degli eventi può avvenire in 3 modi:
1. Direttamente dell'xml;
2. Tramite event handler;
3. Tramite event listener: ogni view delega la reazione ad un evento a un oggetto che implementa l'interfaccia listener (Single Abstract Method, un solo metodo astratto da

implementare). Ogni listener gestisce elementi di un singolo tipo e contiene un solo metodo di callback.

Layout Struttura che definisce l'organizzazione visiva delle View e dei componenti UI sullo schermo. Gli oggetti ViewGroup (un layout estende ViewGroup) sono container invisibili di view che definiscono un layout. Layout statici in Android sono:

- LinearLayout: organizza le view su una singola riga o colonna;
- RelativeLayout;
- TableLayout;
- FrameLayout;
- AbsoluteLayout;
- ConstraintLayout: default di Android, organizza la view secondo vincoli. Ogni view ha associato dei dei constraint definito in relazione ad un altra view precedentemente dichiarata.

Dynamic Layout Usato quando si ha la necessità di popolare a runtime un layout con delle view. Si usano le sottoclassi di AdapterView. Tramite un adapter si ottengono i dati da una sorgente, che vengono mappati dentro gli elementi dell'AdapterView. L'adapter prende in input il contesto, il layout di una singola view del layout dinamico e la struttura contenente i dati da mappare.

Recycler View Libreria che permette di mostrare grandi set di dati in maniera efficiente, creando gli elementi del layout in maniera dinamica, quando necessari. Quando gli elementi escono dallo schermo, le view non sono distrutte, ma riutilizzate per gli altri elementi da mostrare. Per implementarla:

- si definisce una CardView che rappresenta il layout del singolo elemento;
- si definisce la struttura dati di un elemento;
- si definisce un ViewHolder, ovvero il container in cui verrà inserito il contenuto a runtime;
- si definisce un Adapter che prende in input un set di dati e genera un ViewHolder per ogni elemento;
- si usa un layout manager che organizza gli item della lista secondo un determinato stile.

Resources Dato che Android deve girare su tanti dispositivi diversi tra loro, con feature e UI variabile, si separa il codice dalle risorse. Questo è un approccio dichiarativo basato su file XML: tutto ciò che non è codice è una risorsa. L'accesso alle risorse nel codice avviene tramite la classe R: un file generato e mantenuto da Android che contiene gli id per tutte le risorse nella directory `res`. Ogni risorsa ha associato un ID composto da:

- tipo della risorsa (string, color,...);
- nome della risorsa (attributo XML `Android:name`).

Se la risorsa è una view, l'id va specificato esplicitamente con `Android:id="@+id/nome-view"` (`@` indica che la stringa a seguire sarà un ID e `+`, che la classe R deve aggiungerlo). Un app Android dovrebbe fornire risorse alternative per supportare specifiche configurazioni di device. A runtime, Android analizza la configurazione del dispositivo e carica le risorse corrette.

Intent Utilizzati per navigare tra le activity di un app o app esterne. Un intent è un oggetto messaggero che descrive un operazione da fare e un bundle di dati su cui eseguirla. Permette il late runtime binding tra componenti. Esistono due tipi di intent:

Espliciti si specifica direttamente il nome del componente destinatario (package e classe). Sono usati per la navigazione interna dell'app.

Impliciti si dichiara un'azione, un URI e una category, lasciando al SO il compito di risolvere a runtime il componente più idoneo. Se ci sono più destinatari possibili, l'utente sceglie quello da usare tramite una choose activity.

I componenti di un intent sono:

- Name del componente destinatario, opzionale solo per gli intent impliciti;

- Action: stringa che identifica l'operazione da eseguire, permette di identificare quale componente può eseguire l'operazione;
- Data e Type: URI e MIME type dei dati su cui operare;
- Category: altre informazioni di routing;
- Extras: coppie chiave-valore per parametri addizionali;
- Flags: istruzioni aggiuntive su come avviare il componente.

Il SO risolve un intent controllando i manifest di tutte le applicazioni e analizzando gli intent filters di ognuno. Sugli intent filter vengono eseguiti 3 test:

1. Action field test: almeno una delle actions deve combaciare;
2. Category field test: almeno una delle categorie deve combaciare;
3. Data field test: l'URI dei dati dell'intent viene confrontato con parti dell'URI specificate nel filter.

Intent con Risultati utilizzati quando si vuole avviare un activity tramite un intent per ottenerne dei risultati, ad esempio scegliere un immagine dalla galleria. Per ottenere risultati da un intent si utilizza l'Activity Result API: permette al sender dell'intent di ricevere dal receiver l'intent originale arricchito con i risultati richiesti.

1. Il sender crea l'intent utilizzando `registerActivityForResult()`, che restituisce un launcher. Il launcher specifica quale risultato dovrà aspettarsi il sender e rimane in attesa di esso con una callback.
2. Per definire il tipo di risultato atteso, il launcher usa un contract;
3. Se non si usano i contract, si ottiene in risposta un'ActivityResult.

Pending Intent Quando un intent non deve essere lanciato immediatamente, ma in un momento futuro non definito. Il pending intent è particolarmente utile quando non è l'app stessa a definire l'intent, ma qualcun'altro (ad esempio una notifica).
Si avvolge l'intent in un PendingIntent, specificandone il tipo e il componente che lo lancerà (con un builder).

Android Permission System Certe funzionalità che richiedono l'accesso a dati protetti devono richiedere permessi specifici. Questi permessi sono dichiarati nel file manifest e controllati nel momento in cui la funzionalità associata è richiesta. La richiesta di permesso può essere fatta a runtime utilizzando l'Activity Result API.

MVC, MVP e MVVM Sono pattern architetturali.

Model (domain logic) gestisce i dati dell'app, la connessione con i layer del database e network;

View (UI) visualizza i dati e gestisce le interazioni con l'utente;

Per la business logic ci sono tre pattern:

Model View Controller (VMC) il controller è attivo, mentre model e view sono passivi;

Model View Presenter (MVP) la view è attiva, mentre il presente agisce come mediatore uno-a-uno con la view.

Model View ViewModel (MVVM) la view è attiva, contenente la business logic, mentre il ViewModel contiene i LiveData osservati nella view. ViewModel salva i dati della UI in maniera lifecycle aware: i dati sopravvivono indipendentemente dal ciclo di vita dell'activity e si separa il possesso dei dati dalla logica della UI.
Si possono avere più UI controller per lo stesso view model, ma solamente un view mode per uno stesso UI controller. Un ViewModel può essere utilizzato in più view (relazione uno a molti). Nel view model non sono mai referenziati elementi della view: dipendenza in un verso solo.

LiveData e Observable Sono alla base dell'architettura MVVM. Gli observable sono classi che implementano il pattern observer, permettendo ad altre classi di essere notificate quando i dati

osservati vengono modificati. L'oggetto observable ha una lista nascosti di oggetti che vengono notificati (chiamando la loro callback function) quando avviene un cambiamento nello stato.
I LiveData sono lifecycle observable components che notificano solamente observer che si trovano in stato attivo chiamando il loro metodo `onChange()`. I LiveData sono istanziati nel ViewModel.
LiveData e Observables si basano sulla lifecycle awareness: sono observer del lifecycle dell'activity, in modo da poter reagire ai cambi di stato del ciclo di vita.

Android Navigation Framework Facilita la navigazione tramite:

- NavHostFragment: indica la struttura dell'app, un'activity con tanti fragment contenuti in un NavHostFragment che agisce da container;
- Navigation Controller: gestisce la navigazione;
- Navigation Graph: risorsa XML che connette le destinazioni tramite eventi chiamati action.

Ogni action contiene un campo tipo, un'ID per l'azione e uno per la destinazione.
Mantiene un backstack delle transaction tra i fragment, fornendo il back button per navigare a ritroso.

Notifiche