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 usa 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()`

[1] Tranne i Content Provider.

7. `onDestroy()`: l'activity viene distrutta, liberando memoria. Avviene quando cambia il layout dello schermo o il SO ha bisogno di spazio.

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 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 salva 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 tipo MIME 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 Possono essere solo informative o anche attive se cliccandole si apre l'app o si esegue qualche altra azione. Possono utilizzare i pending intent per eseguire delle azioni.
Ogni notifica può avere:

- Icona piccola;
- Nome dell'app;
- Timestamp;
- Icona più grande;
- Titolo;
- Testo.

Le notifiche possono essere modificate anche dopo che sono comparse, come nel caso delle progress bar.
Da Android 8 è obbligatorio che ogni notifica sia associata ad un canale.
Il system service notification manager è il componente del OS responsabile della gestione delle notifiche.

Processo Istanza di un'app in esecuzione, isolata in un sandbox dalle altre per motivi di sicurezza. Ogni app Android dispone di un processo linux e user ID propri. All'interno di un processo possono esserci più thread in esecuzione. I processi possono essere terminati dal OS in caso di carenza di memoria. I processi di un componente vengono specificati nel manifest con `android:process`.

Thread Flusso di esecuzione all'interno di un processo. In ogni processo c'è sempre un thread principale, il main thread/UI thread. Questo si occupa di gestire gli eventi e mostrare la UI.

Si possono creare thread aggiuntivi (worker threads) per delegare operazioni bloccanti. Ogni compito che dura più di qualche millisecondo dovrebbe essere delegato ad un worker thread dedicato. I thread condividono la memoria del processo e hanno un ID proprio.
I thread modificano indirettamente la UI con:
- message passing con il main thread:
- modificando direttamente un mutable live data osservato dalla UI.

Coroutine Operazione computazionale il cui svolgimento può essere sospeso e ripreso. Esegue un blocco di codice in concorrenza senza essere legata ad uno specifico thread. Una coroutine è composta da:

Scope ambiente che tiene traccia delle coroutines create ed offre metodi per interagire con le coroutines;

Context dati del contesto nel quale è in esecuzione la coroutine:
- Job che la coroutine esegue;
- Dispatcher: thread su cui eseguire la coroutine.

Ci sono 3 tipi di dispatcher:
- Main: esegue sul main thread;
- IO: ottimizzato per eseguire operazioni di input-output su disco e rete;
- Default: ottimizzato per operazioni CPU intensive.

Le coroutine sono avviate con il metodo `launch()` da uno scope con un dispatcher. Sono utilizzate per operazioni la cui durata impieghi più di qualche millisecondo: accesso a DB con Room o fetch in rete con Retrofit.
Una coroutine è più leggera di un thread, ed ammette async-await al suo interno.

Multithreading e Message Passing Non si accede a componenti della UI dal worker thread, ma si aggiornano gli elementi tramite message passing.
Due thread comunicano scambiandosi dati e segnali utilizzando 3 elementi:
- Message loop: coda di messaggi associati ad un thread. Un looper rimane in ascolto di nuovi messaggi e gestisce la coda;
- Handler: oggetto che processa i messaggi in arrivo al thread. Se è vuoto, è il sender del messaggio a specificare il comportamento del receiver, inviando al receiver un runnable che lui eseguirà;
- Message: oggetto che può essere inviato/ricevuto da un thread.

Thread e Servizi

Thread	**Servizio**
Consigliato per eseguire operazioni ripetitive di durata finita. • Esegue il codice in parallelo al main Thread. • Implementa runnable, che specifica un comportamento nel metodo `run()`. Non sopravvive alla chiusura dell'activity o dell'app. Il suo ciclo di vita va gestito manualmente. • Utile per operazioni asincrone.	**Consigliato per esecuzioni continue in background.** • Un service è un componente che può eseguire operazioni di durata prolungata e task in background; • Ha un suo lifecycle (`onStart()`, `onCreate()`, ...) come un'activity, ma non ha una UI; • Sopravvive alla chiusura dell'app.

Un foreground service rimane attivo continuamente, non può essere scartato dal sistema per recuperare memoria. Va inserito nel manifest il permesso `FOREGROUND_SERVICE`.

Un service può essere avviato con `startService()`, oppure essere legato ad un componente con `bindService()`. Un bound service termina quando tutti i componenti associati fanno unbind. Un service viene eseguito come qualsiasi altro componente sul main thread. Se fa operazioni complesse bisogna comunque usare coroutine o thread. Se il main thread non è impegnato, un service è l'unico componente in grado di mantenerlo attivo.

Broadcast Receiver Componente che rimane in ascolto di eventi (Intent) specifici lanciati dal sistema o da altre app (anche ad app chiusa). Il broadcast receiver deve essere registrato nel manifest tramite un intent filter.
L'intent può essere inviato tramite i metodi `sendBroadcast(intent)` o `sendOrderedBroadcast(intent)` per intent ordinati per priorità. Quando avviene l'intent specificato, viene invocato `onReceived()`, dove si inseriranno le operazioni da eseguire. Al termine della gestione dell'evento, il broadcast viene distrutto.
Il broadcast receiver ha priorità bassa per l'esecuzione in background. Deve eseguire task brevi, poiché è possibile che il SO lo elimini per liberare spazio.

Data Management e Room Si può utilizzare un database locale con SQLlite. Per utilizzarlo si crea un DBHelper che contiene i metodi per gestire il database. L'output di ogni query sul database è un cursor: un puntatore alle colonne ottenute dalla query. Le operazioni sui db sono bloccanti e vengono eseguite in thread separati.
Room è il framework raccomandato per android. Genera codice SQL a partire da annotazioni (insert, update, delete e query). Room fornisce un abstraction layer su SQLite, semplificandone l'utilizzo.

- Database: contiene il database holder, rappresenta il punto di accesso al database;
- Data Access Object (DAO): interfacce con metodi per accedere alle tabelle dei db;
- Entities: per ogni entity, Room crea una tabella nel db.

Room supporta le observable queries: queries che ritornano oggetti live data, così le modifiche al database vengono immediatamente notificate al live data. L'aggiornamento dei database avviene tramite il migration environment di room: ogni migration class deprecato una startVersion e una endVersion del db.
Il database Room è una Single Source of Truth (SSOT): assicura che la richiesta per i dati sia sempre fatta verso una singola sorgente. Ogni volta che si chiede un dato salvato nel db, si fornisce il LiveData del db. Una classe intermedia repository fornisce una classe che gestisce le chiamata alle risorse in maniera univoca.

HTTP Comunicazioni con la rete avvengono tramite il sistema richiesta-risposta di HTTP. I client HTTP sono implementati con:

- HTTPClient (deprecato);
- HTTPUrlConnection: implementazione lightweight di un client HTTP, utile per applicazioni client-server.

Le connessioni HTTP devono essere gestite su un thread separato. Ci sono librerie più specifiche per gestire le chiamate HTTP:

- Volley: meccanismi di caching e chiamate async;
- Retrofit: serializzazione e de-serializzazione automatica dei contenuti.

Content Provider Interfaccia standard che permette a un'app di esporre i propri dati (come contatti, immagini, file, note, ecc.) ad altre app, oppure di accedere a dati provenienti da altre applicazioni. Un ContentProvider è un componente simile ad un REST web service, utile per accedere a dati condivisi da altre app. Alle app esterne appare come un'interfaccia per il db

tramite cui è possibile fare query. Il content provider deve essere registrato nel manifest per poter essere visibile dalle altre app.

Geolocation API Fa parte del context-aware computing: la computazione dipende dal contesto esterno. I dati di contesto possono essere primari se grezzi, e secondari se hanno subito manipolazioni. La geolocalizzazione è possibile tramite:

- Global Positioning System (GPS): triangola la posizione basandosi sul delay della ricezione di segnali da 3 satelliti.
- Wi-Fi localization: localizzazione basata sulla posizione dei punti di accesso Wi-Fi memorizzati nel database Google Location Service.
- Cellular localization: localizzazione basata sulla posizione del ripetitore telefonico a cui è connesso lo smartphone, memorizzata in un database pubblico.

Ci sono tre tipi di permessi per accedere alla posizione dell'utente:

- ACCESS_FINE_LOCATION: posizione esatta solo ad app aperta;
- ACCESS_COARSE_LOCATION: posizione approssimativa solo ad app aperta;
- ACCESS_BACKGROUND_LOCATION: posizione ad app chiusa.

FusedLocationProvider è un servizio di Android che ottimizza le richieste GPS delle varie app per migliorare la durata della batteria. Sostituisce il LocationListener inefficiente. Il FusedLocationProviderClient invia la location request ad intervalli predefiniti, e il servizio risponde appena riesce.
Il geocoding converte un indirizzo in latitudine/longitudine, implementato dalla classe Geocoder.
Il geofencing permette di ottenere informazioni su quando l'utente entra/esce da un'area precisa, basata su "recinti geografici". GeofencingClient prende in input coordinate e raggio dell'area da recintare, e un pending intent. Quando avviene un geofencing event, il pending intent viene lanciato. Si usa un broadcast receiver per catturare gli intent di geofencing.

System Services Componenti che espongono le funzionalità del framework Android per interagire direttamente con l'hardware. Alcuni permettono di programmare un'azione da eseguire in un momento futuro:

- AlarmManager: permette di lanciare intent (passando un pending intent) nel futuro, specificando in maniera esatta quando avvenire (anche in maniera ripetitiva). Specificando WAKEUP l'intent verrà lanciato anche se il dispositivo è in standby. In caso di riavvio del dispositivo bisogna reimpostare gli alarms.
- JobScheduler;
- WorkManager: permette di eseguire task nel futuro, in maniera asincrona e senza necessità di reimpostarlo nel caso si riavvii il dispositivo. L'azione viene eseguita in un intervallo di tempo flessibile nel futuro, in base alle risorse di cui ha bisogno il sistema. Sfrutta JobScheduler, AlarmManager e BroadcastReceiver, ma non sostituisce l'alarm manager per programmare task in un momento esatto nel futuro. Sfrutta una classe Worker che definisce il task da eseguire nel metodo `doWork()`.

Sensori e Activity Recognition API Sensori del telefono forniscono informazioni di contesto primario, tra cui movimento, ambiente e posizione.
L'activity recognition API aggrega informazioni di contesto primario dei vari sensori per riconoscere tipi di attività che l'utente sta svolgendo. Per accedere ai sensori non sono necessari i permessi, ma per l'activity recognition API si.
Tramite l'activity recognition API è possibile specificare un elenco di ActivityTransition da monitorare e un PendingIntent da lanciare quando almeno una di queste avviene.

Creazione di app in iOS e Android

	Android	**iOS**
Linguaggio	Java/Kotlin	Objective C/Swift
Pattern strutturale	MVVM	MVC
IDE	Android Studio	XCode
View/Component lifecycle	Ciclo di vita completo con numerose fasi	Ciclo di vita ridotto, dettato interamente dal OS
Gestione eventi	Listeners per events, Observables e LiveData per interagire con la UI	Actions con Targets e Outlets per interagire con la UI
Architettura OS	Basata su Linux	Darwin, basato su Unix (vantaggi simili ad Android)
Applicazioni/Processi	User Linux con UserID	User Darwin con UserID
Compilazione app	ART⟶ AOT+JIT	Compilata in codice macchina prima di consegnare l'app allo user. Le app sul app store sono già in codice macchina.
Deploy e rilascio	Play store come app bundle/ APK modulare	App store come app unica
Focus	Portabilità	Prestazioni

Architettura iOS e Android Entrambe sono a livelli, ma:

- iOS è costruito su una base di OS Unix chiamata Darwin, mentre Android è costruito direttamente sul kernel Linux. In entrambi i casi le app sono user, eseguiti in un sandbox dedicato;
- iOS esegue Address Space Layout Randomization (ASLR) per proteggere la memoria da attacchi di tipo buffer overflow;
- Android Fornisce solamente delle linee guida per il design, mentre iOS impone vincoli per le componenti grafiche (Cocoa Touch), garantendo un'estetica consistente tra le app. Dunque la UI di iOS è meno personalizzabile di quella Android.
- iOS non necessita di un HAL globale: dato che l'hardware è proprietario, il SO si interfaccia direttamente con le componenti. Non c'è necessità di garantire una compatibilità più ampia, a differenza di Android dove dove l'architettura hardware non è nota a priori.

Architettura iOS	**Architettura Android**
Core OS: kernel costruito su UNIX, media l'interazione con le componenti hardware.	Linux Kernel: si interfaccia con l'hardware, gestendo portabilità, sicurezza e batteria.
	Hardware Abstraction Layer: interfaccia standard definita da android e implementata dai costruttori del dispositivo per interfacciarsi con i componenti hardware.
Core Services: fornisce i servizi essenziali alle app, a partire dai costrutti fondamentali di livello inferiore.	C/C++ Librarie (NDK): scrivere codice che interagisce direttamente con l'hardware.

Media: gestisce i servizi audio, video, grafica e animazione messi a disposizione delle applicazioni.	Android Runtime (ART): implementazione della JVM per compilare java in bytecode (AOT+JIT) e interpretare bytecode.
Cocoa Touch: layer di astrazione, fornisce librerie per al programmazione di app iOS (estetica, multitasking, notifiche,...)	Api Framework: API che espongono alle app i servizi forniti da Android. System App: applicazioni di default del sistema (con privilegi).

Play Store e App Store Dato che Swift è un linguaggio compilato, non può essere interpretato a runtime, ma è per la sua interezza da compilare prima del suo rilascio. Nei file .app dell'app store è presente codice macchina. Dunque è più pesante di un'app Android perché è già compilata, così come lo sarà quando installata sul telefono. La app è la stessa per ogni dispositivo: iOS richiede che all'interno dell'app sia incluso il supporto per tutte le architetture, le risorse per tutte le lingue supportate, gli asset e il codice compilato universalmente per tutti i dispositivi.
Android invece usa APK modulari o App bundle più leggere, poiché include solo:
- Architettura della CPU richiesta;
- Densità di schermo corretta;
- Lingua selezionata dall'utente.

Sarà poi ART a compilare l'app scaricata nella versione corretta.
L'approccio di Apple è possibile solo perché possiede anche l'hardware proprietario. Il numero limitato di dispositivi permette di precompilare il codice, includendo la compatibilità per ogni dispositivo possibile. Il codice macchina garantisce prestazioni migliori.
Dato che Android si interfaccia con molti dispositivi, necessita di determinare quello giusto al momento del download, e compilare l'app nella versione compatibile. Per questo l'app Android è più leggera di quella iOS. ART fornisce maggiore portabilità.

Compilazione iOS e compilazione Android

	iOS	**Android**
IDE	Il codice non viene compilato direttamente in codice macchina, ma passa attraverso due step intermedi: • Swift Intermediate Language (SIL): ottimizza Swift per ridurre il gap di astrazione tra Swift e IR ed esegue un analisi per cercare errori. • Intermediate Representation (IR): traduce il codice in una rappresentazione più generale usata dal LLVM[2] per generare codice macchina.	Il codice viene compilato in DEX bytecode (Dalvik Executable) e caricato sul Play Store.
Dispositivo	Scarica il codice macchina già compilato dal App Store. L'interprete esegue direttamente il codice	DEX viene eseguito da ART: • AOT compilation converte il DEX in un file `.OAT` installato sul dispositivo, che è machine code

[2]Low Level Virtual Machine (LLVM): compilatore che ottimizza il codice, generando codice eseguibile dal processore.

	macchina quando l'app viene lanciata (maggiore efficienza).	eseguibile direttamente. Un daemon ART controlla periodicamente se ci sono app da ricompilare. • JIT compilation compila il codice al momento dell'esecuzione. Sarebbe possibile anche l'interpretazione diretta del DEX bytecode ma è meno efficiente.

iOS UIKit e Android View Differenze tra iOS UIKit e le view di Android

Android View	**iOS UIKit**
• Compilato e interpretato; • Target-Action-Outlet (late binding dalla view al controller); • Layout Multipli; • Diversi file di risorse; • Completamente personalizzabile; • Supporta pienamente il lavoro in background.	• Completamente compilato; • Listener (late binding dal controller alla view); • AutoLayout; • Modificatori per ogni elemento dell'interfaccia; • Vincolato ad uno standard (Cocoa Touch); • Supporto limitato per il lavoro in background.