Appunti di Mobile Applications and Cloud Computing

Matteo Salvino

Contents

Chapter 1

Introduzione

L' **architettura multi-strato** indica un'architettura software in cui le varie funzionalità del software sono logicamente separate su più strati in comunicazione fra loro. Nell'ambito delle applicazioni web questi strati sono:

- **strato di presentazione**: E' il livello più alto dell'applicazione. Esso mostra le informazioni relative a servizi, elaborate dagli strati di livello più basso.
- **strato logico**: E' il livello che controlla le funzionalità di un'applicazione eseguendo elaborazioni dettagliate. In particolare, gestisce lo scambio di informazioni con lo strato di presentazione con varie elaborazioni e lo strato dei dati.
- **strato dei dati**: E' il livello che si occupa dei dati. In tale strato le informazioni vengono memorizzate e recuperate. Il suo compito è di mantenere i dati neutrali e indipendenti dallo strato logico.

Ciascuno strato comunica direttamente con gli strati adiacenti, richiedendo ed offrendo servizi. Tale approccio fornisce un modello per gli sviluppatori per creare un'applicazione flessibile, riutilizzabile e di facile manutenibilità, visto che è possibile modificare separatamente uno specifico strato, senza influenzare i rimanenti.

Un **mashup** è un'applicazione web ibrida, cioè tale da essere costituita da contenuti provenienti da diverse fonti. Ad esempio, un'applicazione per la prenotazione di un tavolo in un particolare ristorante, dove viene mostrata

la sua ubicazione utilizzando il servizio Google Maps, le recensioni dei clienti passati per decidere se è un buon ristorante o meno, etc.

Il **Fog computing** è un'architettura orizzontale utile a distribuire senza soluzione di continuità risorse e servizi di calcolo, immagazzinamento di dati, controllo e funzionalità di rete sull'infrastruttura che connette il cloud all'Internet delle cose (IoT). In altri termini, il Fog rappresenta un miglioramento e un'estensione del paradigma Cloud in supporto ad applicazione IoT che debbano rispettare dei precisi parametri di qualità del servizio per essere processati, quali latenza e banda disponibile per determinate connessioni, visto che il Cloud non li supporta efficientemente a causa dello spostamento dei dati dai confini della rete verso strutture di elaborazione centralizzate.

1.1 Metodi di progettazione

La progettazione e l'implementazione di un'applicazione viene raggiunta secondo dei determinati metodi di progettazione. Lo scopo di tali metodi è di guidare la struttura del software in modo da raggiungere un alto livello di flessibilità e manutenibilità del software. Si tratta di un organizzazione del codice in diverse componenti, in modo che la modifica all'interno di un componente non influenzi gli altri. Esistono differenti metodi di progettazione, fra cui:

- **MVC**: E' un metodo di progettazione in grado di separare la logica di presentazione dei dati dalla logica di business. Esso è costituito da tre componenti:
  - **Modello** cattura il comportamento dell'applicazione in termini di dominio del problema, indipendentemente dall'interfaccia utente. Esso gestisce dirittamente i dati, la logica e le regole dell'applicazione.
  - **Vista**: si tratta di un qualsiasi rappresentazione in output di informazioni.
  - **Controller**: si occupa della gestione dei comandi dell'utente ricevuti attraverso il componente precedente e li attua modificando lo stato degli altri due componenti.
- **MVP**: E' un metodo derivato dal precedente approccio, per facilitare test di unità automatici e migliorare la separazione dei concetti nella

logica di presentazione:

  - **Modello**: è un'interfaccia che definisce i dati da mostrare.
  - **Vista**: è un'interfaccia passiva che mostra i dati e inoltra i comandi dell'utente al presentatore per agire su tali dati.
  - **Presentatore**: esso agisce sul modello e sulla vista. In particolare, recupera i dati dal modello e li elabora per mostrarli nella vista.

  A differenza dell'approccio MVC, il modello e la vista non possono comunicare direttamente.

- **MVVM**: un pattern software architetturale costituito dai seguenti elementi:

  - **Modello**: il modello è un'implementazione del modello di dominio dell'applicazione che include un modello di dati insieme con la logica di business e validazione.
  - **Vista**: La vista è responsabile della definizione della struttura e l'aspetto di ciò che l'utente vede sullo schermo.
  - **ViewModel**: esso è un intermediario tra la vista e il modello, ed è responsabile per la gestione della logica e della vista. In altri termini, esso fornisce i dati dal modello in una forma che la vista li può utilizzare facilmente.
  - **Binder**: è il meccanismo fondamentale di tale pattern, con il quale il view model e la vista vengono costantemente mantenuti sincronizzati. Questo implica che le modifiche ai dati apportate dall'utente attraverso la vista verranno automaticamente riportate nel view model (e viceversa), risparmiando tale onere allo sviluppatore.

1.2 Framework

Un **framework** è un framework software progettato per supportare lo sviluppo di siti web dinamici, applicazione web e servizi web. Il suo scopo è quello di alleggerire il lavoro associato allo sviluppo delle attività più comuni di un'applicazione web da parte dello sviluppatore. Molti framework forniscono

ad esempio delle librerie per l'accesso alle basi di dati o per la gestione della sessione dell'utente.

Ci sono diversi tipi di applicazioni mobili:

- siti web per dispositivi mobili: inoltrare i contenuti web ai dispositivi mobili utilizzando lo stile di navigazione dei tradizionali siti web.
- web mobile: applicazioni web che mimano il comportamento e l'aspetto di applicazioni native, così come la navigazione.
- Android/iOS native: applicazioni installate sui dispositivi acquistate dall'app store.
- Ibride: mix fra applicazioni native e web
- Native: applicazioni in C/C++.

1.3 Metodologia Agile

La metodologia Agile si riferisce a un insieme di metodi di sviluppo del software emersi a partire dai primi anni 2000 e fondati su un insieme di principi comuni. La gran parte dei metodi agile tenta di ridurre il rischio di fallimento sviluppando il software in finestre di tempo limitate chiamate iterazioni. Ogni iterazione è un piccolo progetto a se stante e deve contenere tutto ciò che è necessario per rilasciare un piccolo incremento nelle funzionalità del software: planning, analisi dei requisiti, progettazione, test e documentazione. Anche se viene ottenuto un risultato intermedio, deve comunque essere pubblicato e cercare di avvicinarsi sempre di più alle richieste del cliente nelle successive iterazioni. L'obiettivo di tale metodologia è la completa soddisfazione del cliente. I principi su cui si basa tale metodologia sono i seguenti: le persone e le interazioni sono più importanti dei processi e degli strumenti, è più importante avere software funzionante che documentazione, bisogna collaborare con i clienti oltre che rispettare il contratto, e bisogna essere pronti a rispondere ai cambiamenti oltre che aderire alla pianificazione.

1.4 Behavior Driven Design

Il BDD è una metodologia di sviluppo del software basata sul TDD. Il BDD combina le tecniche generali e i principi del TDD, con idee prese dal domain-driven design e dal design orientato agli oggetti, per fornire agli sviluppatori software degli strumenti e dei processi condivisi per collaborare nello sviluppo software. In particolare esso specifica che i test di una determinata porzione del software devono essere specificati in termini del desiderato comportamento di tale porzione. Il comportamento di ciascuna sezione viene specificato utilizzando delle user stories. Ogni user story adotta la seguente struttura:

- **As a**: la persona o il ruolo che beneficerà di tale feature;
- **I want**: la feature;
- **so that**: il beneficio o il valore della feature.

Chapter 2

Sensori

I sensori permettono di essere consapevoli dell'ambiente (dove siamo, come ci muoviamo, etc.). Queste informazioni arricchiscono il modo in cui un utente interagisce con lo smartphone. Loro sono dei micro sistemi elettro-meccanici, utilizzati per misurare delle quantità fisiche. In particolare, sono costituiti da due parti:

- **Parte meccanica**: essa sfrutta qualche legge fisica
- **Parte Elettrica**: essa viene utilizzata per trasdurre la quantità meccanica in un valore elettrico leggibile (per esempio l'accelerazione in volts).

Le figure principali di merito di un sensore sono:

- **Sensitività**: il voltaggio del segnale di output generato per unità di input
- **Larghezza di banda**: l'intervallo delle frequenze nel quale la sensitività rimane costante all'intervallo della banda di tolleranza
- **Risoluzione**: il più piccolo incremento di accelerazione misurabile
- **Intervallo di misura**: ad esempio +/- 3g
- **Rumore**: il più basso valore che può essere letto prima che i dati vengano sovrapposti al rumore elettrico.

Un tipo di sensore è il **sensore inerziale**, il quale sfrutta il concetto di inerzia e la classica legge di newton $F = m \cdot a$. Un'altro tipo di sensore è quello di **gravità**, il quale sfrutta la legge di gravitazione universale. La sensitività del sensore risulta pari $\frac{M}{K}$. La frequenza naturale è $\omega_n = \sqrt[2]{\frac{K}{M}}$, la quale diminuisce con la sensitività. Un'altro tipo di sensore è il **sensore giroscopio**, il quale sfrutta la forza di Coriolis. La forza può essere facilmente compresa grazie alla conservazione del momento della forza. Si tratta di una forza apparente che una massa vede quando ad esempio si muove su una traiettoria circolare. Sfortunatamente il magnetometro di uno smartphone non misura solamente il campo magnetico della Terra. Molto spesso in ambienti interni siamo in presenza di dipoli magnetici, i quali perturbano la misura del campo magnetico terrestre. Queste perturbazioni possono esser causate da dispositivi elettromagnetici, strutture oppure altre oggetti ferro magnetici come chiavi, etc. Il **sensore NFC** si basa sul seguente principio: un campo magnetico variabile B_1 generato in una bobina da una corrente variabile potrebbe indurre un'altro campo variabile B_2 in un'altra bobina. L'effetto netto della forza di Lorentz nella seconda bobina è una forza elettromotrice. L'energia può essere allora trasferita da un circuito ad un altro. Tale campo trasporta energia che potrebbe attivare un semplice circuito elettronico privo di batteria. Tale circuito potrebbe eseguire delle semplici computazioni e memorizzare i dati. Uno smartphone agisce con un circuito attivo fornendo l'energia. Il circuito passivo viene attivato quando il circuito attivo è abbastanza vicino in modo tale che possano comunicare oppure memorizzare dati. Le lettura dai sensori sono misurate utilizzando il framework del dispositivo.Il **problema di Wahba** descrive il problema di trovare la matrice di rotazione ottimale partendo da differenti misure (affette da errori o rumore). Ottenere una rotazione precisa è molto importante visto che essa determina la rotazione del dispositivo. La matrice di rotazione contiene tutte le informazioni per determinare l'orientazione. Ogni orientazione può essere espressa come una sequenza di tre differenti rotazioni lungo tre assi differenti (angolo di Eulero) $R = R_x \cdot R_y \cdot R_z$. Possiamo definire diverse tipologie di angolo:

- **Azimuth**: l'angolo tra la direzione attuale del compasso del dispositivo e il nord magnetico.
- **Pitch**: l'angolo tra il piano parallelo allo schermo del dispositivo e il piano parallelo al terreno.

- **Roll**: l'angolo tra il piano perpendicolare allo schermo del dispositivo e il piano parallelo al terreno.

I **quaternioni** sono la generalizzazione di numeri complessi. Le loro unità possono essere utilizzate per rappresentare l'orientazione. L'orientazione è fornita come un vettore e un angolo di rotazione. Tre componenti rappresentano il vettore e una componente l'angolo. Quindi è preferibile rispetto alla matrice di rotazione (4 scalari invece di 9). L' area disponibile per mostrare l'applicazione è chiamata **viewport**. Un'altro sensore è il **GPS**, il quale è costituito da tre segmenti: il segmento spaziale, il segmento di controllo e il segmento utente. L'aeronautica militare degli stati uniti sviluppa, gestisce e opera il segmento spaziale e il segmento di controllo. Il segmento spaziale comprende da 24 a 32 satelliti. Il segmento di controllo di compone di una stazione di controllo principale, una stazione di controllo alternativa, varie antenne dedicate e condivise e stazioni di monitoraggio. Il segmento utente infine è composto dai ricevitori GPS. Il principio di funzionamento si basa su un metodo di posizione sferico chiamato **trilaterazione**, che parte dalla misura del tempo impiegato da un segnale radio a percorrere la distanza satellite-ricevitore. Poiché il ricevitore non conosce il momento in cui è stato trasmesso il segnale dal satellite, per il calcolo della differenza dei tempi il segnale inviato dal satellite è di tipo orario, grazie all'orologio atomico presente sul satellite: il ricevitore calcola l'esatta distanza di propagazione dal satellite a partire dalla differenza tra l'orario pervenuto e quello del proprio orologio sincronizzato con quello a bordo del satellite, tenendo conto della velocità di propagazione del segnale. L'orologio a bordo dei ricevitori GPS è però molto meno sofisticato di quello a bordo dei satelliti e deve essere frequentemente corretto. La sincronizzazione di questo orologio avviene all'accensione del dispositivo ricevente, utilizzando le informazioni che arrivano da un quarto satellite. Se il ricevitore avesse la stessa tipologia di orologio presente nei satelliti, sarebbero sufficienti le informazioni fornite da tre satelliti, ma nella realtà non è così e quindi il ricevitore deve risolvere un sistema di quattro incognite (longitudine, latitudine, altitudine e tempo).

Chapter 3

Introduzione ad Android

L'architettura hardware di un dispositivo è costituita da:

- **Processore in banda base**: il quale ha il proprio sistema operativo in tempo reale, il quale pone i seguenti vincoli: il software che interagisce con le torri cellulari deve essere certificato, è proibito provare ad accedere alle stazioni di base e ha dei stringenti vincoli temporali.
- **Processore dell'applicazione**: esso esegue un sistema Linux basato sul kernel. Esso ha almeno due modalità: utente (non privilegiata) e kernel (privilegiata). Esse sono codificate in un registro della CPU come bits. Ogni tentativo di eseguire un'istruzione privilegiata quando tale processore è in modalità utente lancia un'eccezione. L'unico modo per cambiare modalità è utilizzare una specifica istruzione a livello utente (es. chiamata).

Android costruisce un ricco framework sopra al sottostrato di Linux, ma il suo nucleo fa affidamento su Linux per virtualizzare tutte le operazioni. L'ereditarietà di Linux consente ad Android di utilizzare le sue stesse caratteristiche di sicurezza: i permessi, le capacità, SELinux e altre protezioni di basso livello. Il modello di sicurezza di Linux è in relazione con il modello di sicurezza Unix. Questo modello fornisce le seguenti primitive:

- Ogni utente ha il proprio numero unico identificativo (UID): due utenti potrebbero condividere lo stesso UID, ma questo in effetti significa che, in base a come il sistema è costruito, questo rappresenta un singolo utente con due combinazioni username/password.

- Ogni utente ha un id numerico di gruppo primario: simile allo username, il nome del gruppo non conta, e alcuni GID sono riservati per il sistema.
- Un utente può appartenere ad altri gruppi.

Android utilizza le precedenti primitive ma offre una differente interpretazione: gli utenti sono concessi per singole applicazioni, non per utenti umani. Un utente non può accedere alle risorse di un altro utente. Questo isolamento consente alle applicazioni di essere eseguite una accanto all'altra, senza essere in grado di influenzarsi a vicenda. Il **sandboxing** è un'ambiente in cui ogni applicazione durante l'installazione riceve la sua propria home directory. I permessi di Linux assegnati a questa directory permettono solamente al proprietario dell'applicazione di scrivere a e leggere da questa directory. Lo scopo di un permesso è di proteggere la privacy di un utente Android. Le applicazioni Android devono richiedere dei permessi per accedere a informazioni riservate dell'utente, così come per determinate caratteristiche del sistema (es. internet e la fotocamera). Utilizzare i permessi del filesystem per i files e i drivers del dispositivo, è possibile limitare i processi nell'accedere ad alcune funzionalità del dispositivo. Se un'applicazione ha richiesto l'accesso a qualche dispositivo e l'utente l'ha approvato, questa applicazione ha anche assegnato GID del dispositivo.

3.1 Routines interne

Android ha introdotto un meccanismo chiamato **Position Independent Executable (PIE)**, che esegue delle operazioni correttamente indipendentemente dal suo indirizzo assoluto. Esso viene comunemente utilizzato per le librerie condivise, così che il codice della libreria può essere caricato in una locazione dello spazio di indirizzi di ogni programma, senza utilizzare altre locazioni di memoria, ad esempio dedicate ad altre librerie condivise. Utilizzando tale approccio risparmiamo del tempo di rilocazione, a differenza del codice assoluto che doveva essere caricato ad una specifica locazione di memoria per funzionare correttamente. La protezione contro attacchi di buffer overflow viene aumentata, utilizzando la tecnica **Address Space Layout Randomization**, il quale rende parzialmente casuale l'indirizzo delle funzioni di libreria e delle più importanti aree di memoria. In questo modo l'attaccante che cerca di eseguire del codice malevolo su un computer è

costretto a cercare gli indirizzi del codice e dei dati che gli servono prima di poterli utilizzare, causando una serie di crash del programma. Infatti, accedendo ad un indirizzo errato, si ottengono dati errati, e se viene chiamata una funzione con l'indirizzo errato verrà generata un'eccezione, causando la terminazione del programma. L'attaccante può sfruttare queste eccezioni per collezionare sempre più informazioni in modo da riuscire a conoscere prima o poi il corretto indirizzo di memoria di cui necessita. Questa attività di raccolta di informazioni genera una serie di crash del programma, che quindi sono ben visibili all'utente.

3.2 Gestione della memoria

Android non utilizza la tecnica dello scambio dei files, ma piuttosto gestisce la memoria dividendola in pagine e mappandola. Questo significa che la memoria che viene modificata da un'applicazione rimane residente nella RAM e non è possibile effettuare il paging. L'unico modo per rilasciare la memoria da un'applicazione è di rilasciare i riferimenti degli oggetti che tale applicazione mantiene, rendendo la memoria disponibile per il garbage collector. Per mantenere un ambiente multi-tasking funzionale, Android pone un limite sulla dimensione dell'heap per ciascun applicazione. La dimensione dipende da quanta memoria RAM è disponibile sul dispositivo. Se la nostra applicazione raggiunge la capacità dell'heap e prova ad allocare ancora memoria, esso può ricevere un errore **Out Of Memory**. Nel caso in cui il sistema abbia bisogno di risorse, può decidere di terminare alcuni processi per liberare la RAM. In android, OOM killer è differente da quello tradizionale di Linux, visto che i processi sono classificati in base allo stato dell'applicazione corrente. Dopo che un'applicazione è stata terminata per richiedere risorse, può essere ripristinata, visto che il sistema operativo memorizza lo stato dell'applicazione nella cache. Un'applicazione Android in esecuzione è costituita da diversi processi, classificati come foreground, visibili, servizi, nascosti e vuoti. Per consentire lo scambio delle informazioni fra processi viene utilizzato **Inter-Process Communication**. Esso viene regolato dal **Binder**, il quale è un framework che contiene un oggetto centrale chiamato Binder driver che si occupa di gestire tutte le chiamate IPC. Il meccanismo di alto livello del Binder è chiamato **Intent**.

3.3 Sequenza di boot

Premendo il bottone di accensione, il codice ROM di avvio inizia l'esecuzione da una determinata locazione direttamente collegata alla ROM. Esso carico il **Boot Loader** nella RAM. Il boot loader è il codice che è eseguito prima del sistema operativo del dispositivo. Esso è costituito da codice di basso livello, il quale comunica al dispositivo come trovare ed eseguire il **kernel del sistema**. Tipicamente si trova nella memoria non volatile del sistema del dispositivo. Il kernel di Android è simile al kernel di Linux, visto che inizializza la cache, la memoria protetta, lo scheduling e carica i drivers. Quando queste attività sono state completate, esso cerca il processo **init** nel file system. Tale processo è il primo processo, infatti viene spesso chiamato il processo di root oppure il genitore di tutti gli altri processi. Tale processo esegue lo script init.rc che descrive i servizi del sistema, il file system e altri parametri che hanno bisogno di essere inizializzati. **Zygote** è un processo di macchina virtuale che viene eseguita durante la fase di avvio del sistema, lanciato dal processo init. In Java, noi sappiamo che diverse istanze di macchine virtuali appariranno in memoria per ogni applicazione che verrà eseguita. Se Android lancia diverse istanze di macchine virtuali Dalvik per ogni applicazione, allora consumerà molta memoria e tempo. Per questo motivo Android utilizza il processo Zygote. Infatti, esso permette di condividere il codice fra le macchine virtuali e avere dei minimi tempi di avvio. Il **System Server** è il primo componente Java da eseguire nel sistema, poichè esso esegue tutti i servizi Android come il manager della telefonia e il bluetooth, registrandoli con il gestore dei servizi. Le applicazioni accederanno a tali servizi attraverso un riferimento al gestore dei servizi.

3.4 Android Runtime

ART è un sistema software introdotto con Android 5.0 per andare a sostituire la macchina virtuale Dalvik. La macchina virtuale Dalvik è basata su tecnologia *ust-in-time*, con il quale ogni applicazione viene compilata solamente in parte dallo sviluppatore e sarà poi di volta in volta compito della macchina virtuale eseguire il codice e compilarlo definitivamente il linguaggio macchina in tempo reale, per ogni esecuzione dell'applicazione stessa. ART invece è basato su tecnologia *ahead-of-time* che esegue l'intera compilazione del codice durante l'installazione dell'applicazione e non durante l'esecuzione

dell'applicazione stessa. Questo approccio richiederà un tempo maggiore per la fase di installazione, ma permette di incrementare le prestazioni e la gestione delle risorse in fase di esecuzione.

3.5 Applicazioni

Un'applicazione è generata utilizzando un linguaggio di programmazione. Essa sfrutta le librerie del sistema ma può anche effettuare direttamente delle system calls. Le librerie condivise sono tipicamente collegate al codice a run time. Il codice dell'applicazione e delle librerie viene copiato nella memoria principale e collegate fra di loro. Se due applicazioni utilizzano la stessa libreria, una copia di tale libreria è collegata in qualsiasi applicazione.

3.5.1 Applicazioni native

Le applicazioni native sono applicazioni sviluppate specificamente per un sistema operativo. Esse accedono alle librerie condivise e vengono eseguite direttamente dalla CPU. Vi sono due metodi per sviluppare questo tipo di applicazioni: **NDK** e **gcc**. I vantaggi di questo tipo di applicazioni sono:

- Maggiore velocità, affidabilità e migliore reattività oltre che una risoluzione superiore che assicura una migliore UX.
- Semplice accesso a tutte le funzionalità del dispositivo.
- Notifiche push possibili solamente per questo tipo di applicazioni. Queste notifiche permettono di avvisare gli utenti e di attirare la loro attenzione ogni volta che lo desideriamo.
- Non necessitano obbligatoriamente di internet per funzionare.

Un possibile svantaggio è il loro costo di sviluppo vista la richiesta di sviluppatori esperti per un determinato ambiente e di manutenzione per le diverse versioni da aggiornare per le diverse piattaforme.

3.5.1.1 Native Development Kit

NDK è un insieme di strumenti che ci consentono di utilizzare codice C e C++ con Android. Lo strumento di default di compilazione di Android Studio per

compilare librerie native è CMake. Esso viene utilizzato per controllare il processo di compilazione del software utilizzando una semplice piattaforma e files di configurazione indipendenti dal compilatore. Esso sfrutta il progetto **LLVM**. Essa è un'infrastruttura di compilazione scritta in C++, progettata per l'ottimizzazione di programmi in fase di compilazione, di linking, di esecuzione e di non utilizzo. Usando tale approccio, il programmatore può creare una macchina virtuale per linguaggi che la richiedono, un compilatore dipendente dall'architettura considerata e software di ottimizzazione del codice indipendenti dal tipo di linguaggio utilizzato o dalla piattaforma. La rappresentazione intermedia (IR) è indipendente sia dal linguaggio che dall'architettura; essa si pone fra il codice sorgente in un dato linguaggio e un generatore di codice per una specifica architettura. In questa fase il codice viene ottimizzato/trasformato sulla base di specifiche analisi. L'architettura di un'applicazione può seguire i seguenti approcci:

- E' scritta in Java/Kotlin con pochi metodi scritti in C/C++ che vengono acceduti attraverso la **Java Native Interface**.
- Tutte le activity sono implementate in C/C++.

JNI è un framework del linguaggi Java che consente al codice Java di richiamare il codice nativo scritto in altri linguaggi di programmazione, come C oppure C++. La sua principale applicazione è quella di richiamare all'interno di programmi Java porzioni di codice che svolgono funzionalità non portabili e che non possono essere implementate in linguaggio Java puro. Tale framework permette ai metodi nativi di utilizzare gli oggetti Java nello stesso modo in cui il codice Java li utilizza. Un metodo nativo può creare, ispezionare e aggiornare questi oggetti. Le funzioni native sono implementate in files .c oppure .cpp separati. Quando la JVM invoca la funzione, essa fornisce un puntatore JNIEnv, un puntatore jobject ad qualsiasi altro argomento dichiarato dal metodo Java. Il puntatore JNIEnv è una struttura che contiene l'interfaccia della JVM, la quale include tutte le funzioni necessarie per interagire con la JVM e per lavorare con gli oggetti Java. Il puntatore jobject è un riferimento all'oggetto Java all'interno del quale il metodo nativo è stato dichiarato. Un tipico utilizzo è la mappatura di tipi di dato nativi a tipi di dato Java.

3.5.2 Applicazioni ibride

Il meccanismo principale che ci consente di utilizzare Java e il codice nativo è il **Foreign Function Invocation (FFI)**. Esso è un meccanismo attraverso il quale un programma scritto in un linguaggio di programmazione può chiamare routines oppure fare uso di servizi scritto in un altro linguaggio. **Java Native Interface** è un'interfaccia che consente al codice java in esecuzione su una macchina virtuale java (JVM) di chiamare ed essere chiamato da applicazioni native e librerie scritte in altri linguaggi. Essa consente ai programmatori di scrivere dei metodi nativi per gestire situazioni quando un'applicazione non può essere scritta interamente in linguaggio Java, ad esempio quando le librerie Java non supportano una caratteristica specifica della piattaforma. JNI consente ai metodi nativi di utilizzare gli oggetti Java nello stesso modo in cui il codice Java utilizza questi oggetti. Inoltre, è possibile utilizzare codice Javascript dal codice Java dell'applicazione e viceversa. Infatti, basta rendere i metodi che si vogliono condividere lato Android pubblici, inserire l'annotazione @JavascriptInterface e aggiungere l'interfaccia con Javascript all'applicazione.

3.5.2.1 Cordova

Cordova fornisce un insieme di APIs che possono essere utilizzate per accedere ai servizi dei sistemi operativi mobili nativi come Camera, Geolocalizzazione, etc. Per accedere a tali servizi del dispositivo, vengono utilizzati degli oggetti Javascript attraverso delle chiamate alle APIs di Cordova. La sua architettura è costituita dai seguenti componenti:

- **Webview**: essa fornisce un'interfaccia utente di un'applicazione Cordova e può essere anche utilizzata per essere un componente all'interno di un'applicazione ibrida più grande.
- **Web App**: essa è definita come la parte principale dove il codice dell'applicazione risiede. E' semplicemente una pagina web creata utilizzando HTML, CSS e Javascript. Tipicamente, viene fornito un file locale (index.html) attraverso il quale è possibile fare riferimento al codice e ad altre risorse che sono necessarie per l'esecuzione dell'applicazione. L'applicazione viene eseguita in una WebView all'interno di un container nativo dell'applicazione, attraverso un motore di rendering HTML. Questo container è costituito da un file principale (config.xml), responsabile della comunicazione delle informazioni sull'applicazione.

- **Cordova plugin**: essi sono definiti come una parte integrale dell'ecosistema Cordova che forniscono un'interfaccia per Cordova e per la comunicazione fra componenti nativi. Essi forniscono anche un'interfaccia per le APIs standard del dispositivo (ci consentono di invocare del codice nativo dal codice Javascript). I plugin vengono classificati in **Principali**, i quali forniscono l'accesso ai servizi del dispositivo all' applicazione come Camera, Batteria, etc., e **Personalizzati** che forniscono dei bindings addizionali per servizi che non sono necessariamente disponibili su tutte le piattaforme.

Quando l'applicazione viene eseguita, Cordova carica la pagina di startup dell'applicazione (index.html) nella WebView dell'applicazione e fornisce il suo controllo alla WebView. Quest'ultima consente ad un utente di interagire con l'applicazione inserendo dei dati nei campi di input, cliccando dei bottoni e osservare i risultati ottenuti. Per accedere ai servizi del dispositivo come Contatti oppure Camera, Cordova fornisce un insieme di APIs Javascript che possono essere utilizzate dai sviluppatori dal loro codice Javascript. Le chiamate verranno tradotto in chiamate di API di codice nativo del dispositivo utilizzando un particolare strato di bridge. Le APIs native possono essere accedute attraverso i plugin di Cordova.

3.5.2.2 Xamarin

Xamarin è una piattaforma open-source per sviluppare applicazioni moderne e performanti per iOS, Android e Windows con .NET. Xamarin offre una strato di astrazione che gestisce la comunicazione del codice condiviso con il codice della piattaforma sottostante. Inoltre, esso viene eseguito in un ambiente controllato che fornisce dei metodi di supporto come l'allocazione della memoria e il garbage collection. Tale piattaforma consente a gli sviluppatori di condividere circa il 90% delle loro applicazioni fra diversi sistemi operativi. Xamarin è costruito sopra a **Mono**, una versione open-source del framework .NET basato su gli standard .NET ECMA. Le applicazioni Xamarin.Android compilano dal linguaggio C# in un linguaggio intermedio, il quale è compilato con la tecnologia *just-in-time* in linguaggio macchina quando viene eseguita l'applicazione. Tali applicazioni vengono eseguite nell'ambiente di esecuzione Mono, fianco a fianco con la macchina virtuale ART. Inoltre, tale piattaforma fornisce dei bindings ai namespaces Android.* e Java.*. L'ambiente Mono effettua delle chiamate a tali namespaces attraverso un **Managed Callable**

Wrapper e gli viene restituito il risultato attraverso un **Android Callable Wrapper**.

3.5.3 Applicazioni mobile web

Un'applicazione mobile web è un'applicazione che gira su un server web e che viene utilizzata attraverso un browser web, a differenza di una tradizionale applicazione desktop. Questo significa che gli utenti non dovranno installare l'applicazione sui loro smartphone e quindi la capacità di memoria del dispositivo non verrà influenzata. L'interfaccia utente deve essere ottimizzata per i dispositivi mobili che hanno schermi più piccoli e capacità funzionali legate al tocco con le mani. Tipicamente realizzare un'applicazione di questo tipo richiede meno disponibilità economica rispetto alle applicazioni native. Nonostante ciò, esse possono essere molto lente, non sfruttano al massimo le potenzialità del dispositivo e soprattutto sono totalmente dipendenti dalla connessione internet, non è possibile inviare notifiche push.

3.5.3.1 Architettura

L'idea per mostrare un'applicazione mobile web è di evitare di caricare continuamente la pagina di un determinato sito, bensì memorizzarla localmente. Ma come viene mostrata una pagina web nella nostra applicazione ? Viene eseguito il processo di rendering, nel quale il codice HTML viene tradotto, viene creato l'albero del dominio, viene tradotto il codice CSS, rilevando le sue regole e le direttive del layout. In tal modo abbiamo costruito un albero di rendering. In seguito viene eseguito il processo di layout, attraverso il quale viene assegnata la posizione fisica a ogni elemento da mostrare. Le posizioni vengono calcolate in base al viewport (i.e. area sullo schermo) disponibile. Ogni elemento HTML viene mostrato in una scatola chiamata **modello di scatola CSS**. Le sue dimensioni sono calcolate in base alla propria scatola genitore (container). La scatola genitore di livello più alto rappresenta la scatola del viewport. Molti browser supportano il meta tag viewport, il quale consente di settare il viewport per il processo di rendering. In particolare, esso consente di fornire il reale viewport attuale, così che il rendering viene effettuato sulla dimensione reale. Per gestire differenti viewport possiamo utilizzare due approcci:

- Creare delle regole CSS per ogni tipologia di schermo. In generale, sono necessarie delle elaborazioni lato server.

- Definire delle regole CSS per lo schermo attuale, lato client (web reattivo). Esiste un framework chiamato **Bootstrap 4** che consente di definire delle scatole flessibili per il layout CSS, definendo come un container si adatta al viewport. Tale framework fornisce 5 breakpoints: **xs** per dispositivi extra small ($<$ 576px), **sm** per dispositivi small ($\geq$ 576px), **md** per dispositivi intermedi ($\geq$ 768px), **lg** per dispositivi grandi ($\geq$ 992px) e **xl** per dispositivi molto grandi ($\geq$ 1200px).

3.5.4 Processo di compilazione

Il processo di compilazione coinvolge molti strumenti e processi che convertono il nostro progetto in un Android Application Package (APK). Esso per un tipico modulo di applicazione Android segue questi passi:

1. i compilatori convertono il codice sorgente in files DEX, i quali includono il bytecode in esecuzione sui dispositivi Android e ogni altra cosa nelle risorse compilate.

2. il gestore dell'APK combina i files DEX e le risorse compilate in un singolo APK. Prima che la nostra applicazione possa essere installata e rilasciata su un dispositivo Android, l'APK deve essere firmata.

3. il gestore dell'APK firma la nostra APK utilizzando la chiave di debug oppure di rilascio:

    - se stiamo compilando una versione di debug dell'applicazione, cioè un'applicazione intesa solamente per profilazione e testing, il gestore firma la nostra applicazione con la chiave di debug.
    - se stiamo compilando una versione di rilascio dell'applicazione che abbiamo intenzione di rilasciare esternamente, il gestore firma la nostra applicazione con la chiave di rilascio.

4. prima di generare l'APK finale, il gestore utilizza lo strumento zipalign per ottimizzare l'applicazione in modo da utilizzare meno memoria quando verrà eseguita sul dispositivo dell'utente finale.

Android studio per automatizzare e gestire il processo di compilazione utilizza il toolkit **Gradle**. La personalizzazione delle configurazioni di compilazione è molto flessibile, infatti possiamo modificare le variabili dell'ambiente di

compilazione che troviamo nel file .properties. Il processo di compilazione è controllato da diversi scripts utilizzando un DSL (linguaggio specifico del dominio). Il file setting.gradle comunica a Gradle quali moduli devono essere inclusi quando compiliamo l'applicazione. Il file build.gradle è un file di alto livello che definisce le configurazioni di compilazione che verranno applicate a tutti i moduli del progetto. Per default, tale file utilizza un blocco buildscript per definire le repositories e le dipendenze Gradle che sono comuni a tutti i moduli nel progetto. Possiamo definire diverse tipologie di dipendenze:

- dipendenza su un modulo di libreria locale: il sistema di compilazione compila il modulo della libreria e incapsula il contenuto compilato nell'APK.
- dipendenza binaria locale: i files JAR già esistono.
- dipendenza binaria remota: implementazione di una libreria remota.

Chapter 4

Framework Android

Adesso introdurremo delle definizioni di alcuni componenti presenti nel framework Android, come:

- **Drawable**: esso è un'astrazione generale per qualcosa che può essere disegnato sullo schermo.
- **Vista**: essa a differenza del precedente componente è anche in grado di gestire gli eventi.
- **Immagine digitale**: un'immagine digitale W x H è una matrice di pixels (Width, Height). Un pixel è la più piccola unità visuale che può essere controllata. Quando si parla di dimensione ci riferiamo al numero totale di pixels. Inoltre, spesso viene definito l'aspect ratio W/H.
- **Immagine vettoriale**: essa descrive un' immagine in termini di punti, i quali sono connessi da linee e curve per formare poligoni ed altre forme. I formati più popolari sono: PDF, SVG e EPS.
- **Schermo**: Uno schermo ha una superficie X x Y, una diagonale $\sqrt[2]{X^2 + Y^2}$, e un aspect ratio $AR = Y : X$, dove Y è il numero più grande. Spesso le dimensioni degli schermi vengono fornite come la lunghezza della diagonale in inches (1 in = 2.54 cm). E' possibile determinare anche quanti pixel sono presenti per inch (PPI).

In generale fornita un'immagine W x H, non possiamo dire nulla sulla sua risoluzione spaziale, questo perchè tale risoluzione dipende dalle dimensioni dello schermo X x Y dove verrà mostrata l'immagine. Ha senso incrementare

sempre di più il PPI ? L'occhio umano ha la sua massima risoluzione (300 PPI), così non ha senso andare oltre il limite. Infatti schermi con elevati PPI mappano un pixel software su diversi pixels hardware.

4.1 Il sistema delle viste

Esso è un sistema per organizzare l'interfaccia grafica. **ViewGroup** è una vista che contiene altre viste (ad esempio il Linear Layout). Un **container** è un altro tipo di ViewGroup utilizzato per ospitare del contenuto dinamico, oppure del contenuto che non può essere adattato alle dimensioni dello schermo (supporto dello scrolling). La gestione del contenuto dinamico richiede un'implementazione precisa per ottimizzare le prestazioni: il contenuto dovrebbe essere recuperato da uno sorgente lenta (internet), utilizzare un thread separato per eseguire il download di contenuti, gli elementi della vista devono essere aggiunti e aggiornare la vista. Gli **stili** e i **temi** su Android ci consentono di separare i dettagli della progettazione dell'applicazione dalla struttura e dal comportamento dell'interfaccia utente. Uno **stile** è una collezione di attributi che specificano l'apparenza per una singola vista. Uno stile può specificare attributi come il colore e la dimensione del font, il colore di sfondo, etc. Un **tema** è un tipo di stile che viene applicato all'intera applicazione, non solo ad una singola vista.

4.2 Activity

Un'applicazione è costituita da almeno un'activity, la quale è controllata dalla GUI nello schermo. Essa viene eseguita all'interno del thread principale e dovrebbe reagire all'input dell'utente (gestita dal gestore delle Activity). Le activity nel sistema sono gestite attraverso una pila posteriore (back stack). In particolare, quando un activity viene creata, essa viene posizionata in cima a tale pila e diventa attiva. Le activity precedenti rimangono nella pila in attesa della distruzione dell'activity attualmente attiva. L'insieme delle activity lanciate da un utente è chiamato **Task**. Tutte le componenti dell'applicazione vengono eseguite all'interno del thread principale. Tuttavia, possono essere creati altri thread per eseguire delle operazioni che richiedono molto tempo.
Ogni activity viene creata dal launcher quando viene selezionata l'icona cor-

rispondente. Essa può essere creata anche da un'altra activity oppure da un altro componente software. Un activity può essere in tre differenti stati: esecuzione, pausa e arrestata. Un activity in esecuzione è in primo piano e ha il focus, così l'utente può interagire con essa. Un activity in pausa è parzialmente visibile ma non ha il focus. Questo si può verificare quando un'altra activity viene eseguita ma non ricopre l'intero schermo. Un activity è arrestata quando viene completamente oscurata da un'altra. Le activity in pausa e arrestate possono essere terminate dal sistema quando ha bisogno delle risorse. Quando un activity viene creata, viene chiamato il metodo onCreate(). Esso inizializza tutte le viste, i dati, etc, ed è seguito dal metodo onStart(). Questo metodo è chiamato prima che l'activity diventi visibile e quando l'activity viene riavviata dopo lo stato di arresto (onRestart() -¿ onStart()). Il metodo onResume() viene chiamato dopo il metodo onStart() oppure quando l'activity viene ripristinata dallo stato di pausa. Il metodo onPause() viene chiamato prima del ripristino di un'altra activity. Esso è utilizzato per eseguire il commit di cambiamenti non salvati e per arrestare i threads, le animazioni, etc. Esso deve essere veloce perchè l'altra activity non può essere ripristinata fino a quando l'activity corrente non abbia terminato i processi sopra citati. Il metodo onStop() è chiamato quando l'activity non è più visibile all'utente. Se l'activity viene ripristinata, questo metodo è seguito da onRestart(), altrimenti se l'activity deve essere distrutta viene chiamato il metodo onDestroy(). Per memorizzare un piccolo ammontare di dati che possono essere serializzati e deserializzati può essere utilizzato un **Bundle**. Tuttavia, esso diventa inefficiente al crescere dell'ammontare dei dati da memorizzare. Android risolve tale problema, fornendo un modo per memorizzare i dati attraverso il **ViewModel**. Un activity è creata se è il target di un messaggio speciale chiamato **Intent**.

4.2.1 Intent

Un intent è un oggetto di messaggistica che può essere utilizzato per richiedere un'azione da un'altro componente dell'applicazione. Esso facilita la comunicazione fra le componenti, ma può essere anche utilizzato per eseguire un'activity o un servizio. Gli intent consentono di chiamare un'activity ed ottenere il risultato. L'activity chiamante non attenderà in modo sincrono il risultato, ma l'activity chiamata quando avrà elaborato il risultato chiamerà il metodo setResult, il quale eseguirà il metodo onActivityResult dell'activity chiamante. Quando un'activity vuole passare dei dati ad un'altra activity,

ha bisogno di settare dei campi extra nell'intent. Ad esempio, possiamo raggiungere questo scopo utilizzando un bundle, il quale può inglobare tipi di dato primitivi e strutture dati. Un intent viene classificato in:

- **Unicast**: intent intesi solamente per uno specifico componente oppure per una determinata azione da compiere. Gli intent di questo tipo si dividono in:
  - **Espliciti**: quando il target dell'intent è dichiarato all'interno di esso.
  - **Impliciti**: quando l'intent specifica solamente l'azione che il componente dovrebbe compiere. Se sono disponibili diverse activity che possono eseguire l'azione richiesta, allora l'utente ha bisogno di selezionarne una.
- **Broadcast**: intent che comunica delle informazioni a tutti i componenti (vedi Broadcast Receivers).
- **Pending**: intent con i quali possono essere attivati dei componenti in futuro (utilizzato per le notifiche).

Un **intent-filter** è un'espressione nel manifesto dell'applicazione che specifica il tipo di intent che il componente vorrebbe ricevere. Dichiarando un intent filter per un activity, rendiamo possibile ad altre applicazioni di lanciare direttamente tale activity con un determinato tipo di intent. Ad esempio, supponiamo che l'activity A nel package PA vuole chiamare l'activity B nel package PB. L'activity B definisce un intent-filter contenente un'azione personalizzata e la categoria di default. L'activity A crea un intent con l'azione personalizzata e chiama startActivity. Il sistema trova il componente appropriato da eseguire confrontando il contenuto dell'intent con il contenuto dell'intent filter dichiarato nel manifesto di altre applicazioni sul dispositivo. Le activity vengono eseguite come due utenti separati.
Un **Broadcast Receiver** è un componente che ci consente di registrare gli eventi di un'applicazione o del sistema. Tutti i ricevitori registrati per un evento vengono notificati da Android una volta che l'evento si verifica. La parte di implementazione della registrazione consiste nella creazione di un intent filter per indicare gli specifici intent broadcast cui il ricevitore deve ascoltare, facendo riferimento alla stringa dell'azione dell'intent broadcast. Quando viene trovata una corrispondenza broadcast, il metodo onReceive()

del ricevitore viene chiamato; a questo punto il metodo ha a disposizione 5 secondi per eseguire le attività necessarie prima di restituire il controllo. I ricevitori broadcast non hanno bisogno di essere eseguiti di continuo. Gli intent broadcast sono degli oggetti intent che diventano broadcast attraverso una chiamata al metodo sendBroadcast() della classe Activity(). Inoltre, per fornire un sistema di comunicazione tra le componenti dell'applicazione, gli intent broadcast vengono anche utilizzati dal sistema Android per notificare le applicazioni interessate per quanto riguarda gli eventi principali del sistema (ad esempio la connessione/disconnessione del cavo di alimentazione).

4.3 Permessi

Android è un sistema operativo a privilegi separati, nel quale ogni applicazione viene eseguita con una diversa identità di sistema (Linux user id e group id). Le parti del sistema sono anche separate in diverse identità. In tal modo, Linux separa le applicazioni l'una dalle altre e dal sistema. Visto che ogni applicazione Android opera in un processo sandbox, le applicazioni devono richiedere esplicitamente i permessi a loro necessari per ulteriori capacità non fornite nella sandbox nel manifesto. Dipende da quanto l'area è sensibile, il sistema potrebbe consentire i permessi automaticamente oppure potrebbe chiedere all'utente se approvare oppure rifiutare la richiesta. Quando un permesso viene concesso per un' applicazione, il sistema assegnerà un Group id supplementare corrispondente al permesso. Se la piattaforma del dispositivo è Android 6.0 (livello API 23) o superiore, e il targetSdkVersion dell'applicazione è 23 o superiore, l'applicazione richiederà i permessi all'utente a tempo di esecuzione. Un utente può revocare i permessi in qualsiasi momento, così l'applicazione ha bisogno di controllare se ha i permessi ogni volta che viene eseguita. Se la piattaforma del dispositivo è Android 5.1 (livello API 22) o inferiore, oppure se il targetSdkVersion è 22 o inferiore, il sistema chiederà di fornire i permessi quando l'utente installerà l'applicazione. Se aggiungiamo un nuovo permesso ad una versione aggiornata dell'applicazione, il sistema chiederà all'utente di consentire tale permesso quando l'utente aggiornerà l'applicazione.

4.4 Servizi

Un servizio è un componente dell'applicazione che può eseguire delle lunghe operazioni in background, il quale non fornisce alcuna interfaccia utente. Il ciclo di vita dei servizi è indipendente dal componente che lo ha lanciato, così loro possono continuare la loro esecuzione anche se tale componente ha terminato la sua esecuzione. I servizi si dividono in due categorie:

- **Foreground**: un servizio di questo tipo esegue alcune operazioni che sono visibile all'utente. Per esempio, un'applicazione per la musica potrebbe utilizzare un servizio di foreground per procedere con l'esecuzione di un brano. Essi devono mostrare un'icona nella barra di stato del dispositivo, invocando tipicamente il metodo startForeground(id, notification). Il sistema difficilmente terminerà questo tipo di servizi.

- **Background**: un servizio di background esegue un operazione non direttamente visibile all'utente. Per esempio, se un'applicazione utilizza un servizio per comprimere la memoria che utilizza, questo sarà probabilmente un servizio di background. Il sistema ha più probabilità di terminare un servizio di background rispetto a uno di foreground. A partire da Oreo, per ragioni di prestazioni il sistema impone delle restrizioni sui servizi di background in esecuzione quando l'applicazione stessa non è in foreground. Per eseguire delle operazioni di background viene raccomandato l'utilizzo dello **JobScheduler**.

I servizi possono assumere due comportamenti:

- **Started service**: esso esegue una singola operazione e non restituisce il risultato direttamente al chiamante. Il servizio può essere eseguito in background indefinitivamente, anche se il componente che lo ha lanciato viene distrutto. Loro vengono eseguiti chiamando la funzione startService(). Quando il servizio ha completato la sua attività, esso dovrebbe chiamare il metodo stopSelf(). Alternativamente, un servizio in esecuzione può essere arrestato chiamando il metodo stopService(). Il metodo onBind() deve restituire null.

- **Bound service**: esso offre un'interfaccia client-server che consente ai componenti di interagire con il servizio, inviare richieste, ricevere risultati e persino fare questo fra processi con la comunicazione fra processi

(IPC). Un servizio di questo tipo verrà eseguito fino a quando il componente a cui è associato continuerà l'esecuzione. Loro vengono lanciati con una chiamata al metodo bindService(), ma permettono delle interazioni con il componente che li ha lanciati. Diversi componenti possono essere collegati ad uno stesso servizio simultaneamente. Per dissociarsi da un servizio, il componente chiama il metodo unbindService(). Quando tutti i componenti collegati ad un servizio vengono dissociati, il sistema lo termina. Un servizio di questo tipo può essere anche lanciato con una chiamata a startService() e poi i componenti possono associarsi ad esso invocando la funzione bindService(). In questo caso il servizio non verrà terminato quando tutti i componenti si saranno dissociati da esso.

Lo **JobScheduler** viene utilizzato quando un'attività non richiede un tempo esatto, ma potrebbe essere schedulato basato sulla combinazione dei requisiti dell'utente e del sistema. Per esempio, un'applicazione potrebbe aggiornare le notizie di mattina, ma potrebbe attende fino a quando il dispositivo sarà carico e connesso al wifi per aggiornarle, per preservare le risorse del sistema e i dati dell'utente. Tale classe permette di stabilire le condizioni oppure i parametri per eseguire una determinata attività. Inoltre, esso calcola il miglior istante temporale per schedulare l'esecuzione dell'attività richiesta.
Un servizio viene eseguito sul thread principale, così se esso deve eseguire una lunga operazione dovrebbe creare un nuovo thread che esegue tale attività. **Intent Service** semplifica tale operazione. Intent Service è una sottoclasse di Service e predispone un thread per gestire le attività in background asincronamente. Inoltre, essa processa una richiesta alla volta in base alla politica FIFO. Una volta che il servizio ha gestito tutte le richieste, esso termina. Per processare una richiesta, il metodo onHandleIntent() viene chiamato, così il programmatore deve implementarlo. Visto che Intent Service è un servizio, deve essere necessariamente registrato nel manifesto dell'applicazione.

4.5 Frammento

Un frammento è una parte di un activity, il quale contribuisce all'interfaccia utente di tale activity. Un frammento può essere pensato come ad una sotto activity, mentre lo schermo completo con il quale l'utente interagisce è chiamato come activity. Un activity funge da collante per diversi frammenti. Un frammento ha il proprio layout e comportamento, così come un proprio ciclo

di vita. Possiamo aggiungere oppure rimuovere dei frammenti in una singola activity per costruire un interfaccia utente multi riquadro. Un frammento può essere utilizzato in diverse activity. Il ciclo di vita di un frammento è strettamente in relazione con il ciclo di vita dell'activity ospitante. Ad esempio, quando un activity è messa in pausa, tutti i frammenti disponibili nell'activity verranno messi in pausa. I frammenti hanno delle callback extra rispetto alle activity, per gestire le interazioni con l'activity per eseguire azioni come la costruzione o distruzione dell'interfaccia utente del frammento. Questi metodi addizionali sono:

- onAttach() chiamato quando il frammento è stato associato con un activity.
- onCreateView() chiamato per creare la gerarchia delle viste associata al frammento.
- onActivityCreated() chiamato al completamento del metodo onCreate() dell'activity ospitante.
- onDestroyView() chiamato quando la gerarchia delle viste associata al frammento viene rimossa.
- onDetach() chiamato quando il frammento viene dissociato dall'activity.

Android fornisce un manager chiamato **Fragment Manager** per aggiungere/rimuovere/trovare un frammento. Tipicamente, tale manager utilizza un **FrameLayout** per fornire l'area dove il frammento mostrerà la propria vista. Quando un frammento è creato programmaticamente, l'activity ospitante può utilizzare un Bundle per fornire dei dati al frammento. Diversi frammenti in una stessa activity possono comunicare utilizzando l'activity ospitante.

4.6 RecyclerView

E' una tipologia di vista che ci permette di mostrare una lista di dati. Le liste possono essere molto complesse, mostrare una serie di destinazioni, oppure dei dati inerenti a transazioni, etc. I benefici nell'utilizzare questa vista sono i seguenti:

- Per default, RecyclerView processa solamente gli elementi attualmente presenti sullo schermo. Quando l'utente effettua uno scrolling, tale componente si accorge che dei nuovi elementi devono essere mostrati ed effettua abbastanza lavoro per mostrarli.
- Quando un elemento non è più visibile sullo schermo, la sua vista viene riciclata. Ciò significa che la sua vista verrà riutilizzata da un'altro elemento, aggiornando il rispettivo contenuto.
- Quando un elemento cambia, invece di progettare l'intera lista, tale componente può aggiornare solamente tale elemento.

4.7 Multithreading

Quando viene eseguita l'activity principale di un'applicazione, essa viene eseguita sul thread principale. Questo è responsabile per lo smistamento degli eventi alla vista all'interno dello schermo. Il thread dell'interfaccia utente dovrebbe essere molto reattivo a gli input dell'utente, così lunghe operazioni non dovrebbe essere eseguite su di esso. Ci sono diversi modi per implementare il multithreading. Il primo è di utilizzare la programmazione standard dei thread, estendendo la classe Thread ed effettuando l'override del metodo run(). Se il thread di lavoro ha bisogno di aggiornare l'interfaccia utente, deve comunicare con il thread principale attraverso un meccanismo basato su messaggi: il thread principale crea un oggetto handler e il thread di lavoro lo utilizza per ottenere un messaggio vuoto e lo invia al thread principale. L'handler consente di inviare oggetti Message e Runnable. Un'altro modo per implementare il multithreading è utilizzare la classe AsyncTask. Essa semplifica l'interazione tra il thread principale e il thread di lavoro: essa consente di eseguire operazioni in background (metodo doInBackground()) e pubblicare i risultati nel thread principale (metodo onPostExecute()). Possiamo anche eseguire operazioni di background con Intent Services oppure utilizzando il framework Volley, il quale semplifica le richieste HTTP.

Chapter 5

Memorie

Dove memorizzare i dati di un'applicazione ? Abbiamo le seguenti possibilità:

- **SharedPreference**: consente di memorizzare dati privati (booleani, float, int, long e string) come coppie chiave-valore. E' una buona opzione per un piccolo ammontare di dati (es. stato di un gioco).
- **Preference**: è stato aggiunto in android X, per memorizzare le preferenze di un utente mostrate a quest'ultimo tramite le impostazioni dell'applicazione.
- **Memoria interna**: può essere utilizzata la memoria interna del dispositivo per memorizzare i dati di un utente. Tipicamente viene creata una cartella privata quando l'applicazione viene installata.
- **Memoria esterna**: può essere utilizzata la memoria esterna del dispositivo (SD) per memorizzare i dati di un utente, i quali sono accessibili da chiunque.
- **SQLite**: possiamo memorizzare i dati in un database con una determinata struttura dati. Viene raccomandato l'accesso attraverso la **Room Persistency Library**, con il quale la correttezza delle query verrà controllata a tempo di compilazione. Per accedere ai dati dell'applicazione utilizzando tale libreria, dobbiamo lavorare con i **Data Access Objects (DAO)**. Questo insieme di oggetti DAO costituisce il componente principale di Room, visto che ogni DAO include dei metodi che offrono un'accesso astratto al database dell'applicazione. Accedere al database utilizzando questi oggetti, ci permette di separare differenti

componenti dell'architettura del database. Prima di tutto per utilizzare un DAO come interfaccia bisogna inserire l'annotazione @Dao. Per le operazioni CRUD verranno utilizzate le rispettive notazioni @Insert, @Query, @Update e @Delete.

- **Provider di contenuti**: esso è una repository di informazioni condivise fra diverse applicazioni. Fornisce un pieno controllo per le operazioni di lettura e scrittura, e un'interfaccia uniforme che può essere implementata in un file oppure in un database. Alcuni esempi di provider di contenuti sono Contatti, Calendario, etc. L'URI che viene utilizzato per richiedere una determinata informazione assume la seguente forma $content : //authority/path/id$, dove **content** significa che vogliamo accedere al provider di contenuti, **authority** rappresenta il nome del provider, **path** è 0 oppure diversi segmenti che indicano i dati a cui vogliamo accedere e **id** specifica l'elemento. Possiamo utilizzare un provider di contenuti attraverso un intent con target il provider stesso. Inviare un intent all'applicazione Contatti del dispositivo ci consente di accedere al provider dei Contatti indirettamente. Infatti, l'intent lancia l'interfaccia utente dell'applicazione Contatti del dispositivo, nel quale un utente può effettuare delle operazioni relative ai Contatti. In questo modo l'utente può prendere un contatto dalla lista e restituirlo all'applicazione, modificare i dati di un contatto esistente, inserire un nuovo contatto e rimuovere un contatto.

La libreria **DataBinding** consente di associare dei valori nel layout ad un oggetto di dati. Essa consente di aggiungere dei listeners e notificare il cambiamento dei dati all'interfaccia utente. **Firebase** è un database in tempo reale. Molti database richiedono delle chiamate HTTP per ottenere e sincronizzare dei dati. Quando colleghiamo la nostra applicazione a Firebase, non ci stiamo connettendo attraverso una normale chiamata HTTP, ma ci stiamo connettendo attraverso una WebSocket. Esse sono molto più veloci di HTTP. Tutti i nostri dati verranno aggiornati automaticamente attraverso una singola WebSocket con una velocità dipendente dalla rete dell'utente. Quando un utente modifica dei dati, tutti gli utenti connessi ricevono i dati aggiornati quasi istantaneamente.

Chapter 6

Web API

6.1 Remote Procedure Call

Una RPC è quando un programma invoca una procedura per eseguirla su una macchina differente da quella su cui è in esecuzione il programma. Il chiamante (client) realizza una chiamata locale al componente software (stub) che comunica con l'oggetto remoto. Nel lato chiamato (server) c'è un componente (scheletro) che riceve la chiamata dello stub e realizza una chiamata locale al codice del server. RPC utilizza il protocollo HTTP e i dati sono rappresentati in JSON oppure XML. Generalmente, il flusso di esecuzione di RPC è il seguente:

1. Il client chiama il proprio stub come in una tradizionale chiamata a procedura locale.
2. Lo stub del client inserisce i parametri in un messaggio e realizza una chiamata di sistema per inviare il messaggio.
3. Il sistema operativo del client invia il messaggio dalla macchina del client alla macchina del server.
4. Il sistema operativo del server fornisce i pacchetti in entrata allo stub del server.
5. Lo stub del server estrae i parametri del messaggio.
6. Infine, lo stub del server chiama la procedura del server.

La risposta segue gli stessi passi, ma in ordine inverso.

6.1.1 XML-RPC

Esso è un protocollo RPC il quale utilizza XML per codificare i dati. I dati che possono essere convertiti in XML sono scalari, arrays oppure structs. Esso invia una richiesta HTTP al server che implementa il protocollo. Ci sono tre tipi di messaggio: messaggio di richiesta, messaggio di risposta e messaggio di errore. L'elemento principale del messaggio di richiesta è il **methodCall**. Esso contiene gli elementi **methodName** e **params**. Il primo contiene il nome della procedura invocata e il secondo contiene una lista di valori per i parametri della procedura. L'elemento principale del messaggio di risposta è **methodResponse**. Esso contiene un elemento **params**, il quale è una lista di valori.

6.1.2 JSON-RPC

Esso è un protocollo RPC il quale utilizza JSON per codificare i dati. La richiesta deve contenere tre proprietà: **metodo** è una stringa con il nome del metodo da invocare, **params** è un array di oggetti da fornire come parametro al metodo definito e **id** è un valore di un qualsiasi tipo, utilizzato per abbinare la risposta con la richiesta a cui si stà rispondendo. Anche il responso deve contenere tre proprietà: **result** sono i dati restituiti dal metodo invocato, se un errore si verifica mentre invochiamo il metodo, questo valore deve essere null, **error** è uno specifico codice di errore se si verifica un errore, altrimenti è null, **id** è l'id della richiesta a cui si stà rispondendo. E' possibile utilizzare le notifiche: in questo caso l'id deve essere settato a null. JSON-RPC può essere utilizzato come un semplice sistema di richiesta e responso, oppure per inviare delle notifiche a tutti i clients.

6.1.3 REST

Essa è la più semplice forma di RPC. Le richieste sono mappate a metodi HTTP (GET, POST). Il nome del metodo è un parametro della chiamata e il risultato può essere formattato come dati XML oppure JSON. Il suo nome deriva da RESTful Web Services: il protocollo è stateless, ogni risorsa è identificata da un URI e le operazioni CRUD sono mappate a verbi HTTP (POST, GET, PUT, DELETE).

6.2 Gestione delle risorse

Naturalmente, bisogna gestire le risorse sulla base di diversi aspetti:

- **Autenticazione**: stabilire un'associazione di fiducia tra il client e l'identità del principale.
- **Autorizzazione**: determinare quali risorse e operazioni un'utente autenticato può eseguire.
- **Controllo dell'accesso**: controllare se un operazione su una risorse è consentita, ad esempio possiamo controllare la validità del token di accesso che viene rilasciato ad un utente dopo aver effettuato l'autenticazione.
- **Tasso di controllo**: un controllo sul numero massimo di richieste da processare.

Quando recuperiamo i dati da una sorgente lenta, sono più appropriate delle chiamate asincrone, altrimenti l'interfaccia utente viene bloccata. In particolare, l'utente invia la richiesta e procede con il suo proprio flusso di esecuzione. Quando le risorse saranno pronte, riceverà una notifica. Per aggiornare delle risorse effettuiamo prima dei cambiamenti sulla nostra copia locale, poi li aggiorniamo asincronamente. Se la rete non è disponibile utilizziamo un database locale e un meccanismo di sincronizzazione che verrà eseguito quando la rete sarà nuovamente disponibile. Ad esempio, nel caso di una READ quando la rete è disponibile, restituisco un elemento placeholder, recupero i dati aggiornati e sostituisco il placeholder. Nel caso in cui la rete non fosse disponibile, mostro un messaggio all'utente. Ad esempio, nel caso di una CREATE, se la rete è disponibile utilizzo un thread secondario per aggiornare il server, altrimenti memorizzo tale richiesta. Quando la rete sarà disponibile, le operazioni appese saranno eseguite in sequenza.

Quando diversi utenti (applicazioni) condividono un singolo dato su cui vengono eseguite delle operazioni di lettura e scrittura concorrenti, la sua consistenza potrebbe essere un problema (Teorema CAP). **Firebase** è un database in tempo reale che si occupa di questi problemi. In caso di un file, la sincronizzazione è gestita da uno speciale algoritmo e i conflitti vengono risolti, ad esempio realizzando una copia dei dati.

6.3 JSON Web Token

Esso è uno standard Internet per creare dei token di accesso basati su JSON che sostengono un determinato numero di richieste. Per esempio, un server potrebbe generare un token che viene assegnato alla richiesta del login come amministratore, e lo fornisce al client. Il client potrebbe utilizzare il token per provare che è loggato come amministratore. I tokens sono firmati con la chiave privata del server, così che entrambi possono verificare se il token sia legittimo o meno. Un token è costituito da tre informazioni: **Header**, il quale identifica quale algoritmo è stato utilizzato per generare la firma, il **Payload** che contiene una serie di richieste e la **firma**, il quale valida in sicurezza il token. Le richieste standard che possono essere inserite nel payload sono:

- iss: identifica l'ente che ha rilasciato il token.
- sub: identifica l'ente a cui è stato rilasciato il token.
- aud: identifica il destinatario del token. Ogni ente che rilascia dei token deve auto identificarsi con un valore, altrimenti il token deve essere rifiutato.
- exp: rappresenta la data di scadenza del token, dopo la quale esso non deve essere accettato.
- nbf: rappresenta la data di inizio di validità del token.
- iat: identifica la data in cui è stato rilasciato il token.
- jti: identificatore unico (case sensitive) del token anche fra differenti issuers.

Chapter 7

Cloud Computing

Il **Cloud Computing** è un modo per utilizzare le infrastrutture IT senza il bisogno di installare uno specifico hardware relativo all'infrastruttura da utilizzare. L'infrastruttura IT può essere una macchina virtuale, una piattaforma software utilizzata per sviluppare ed eseguire applicazioni su diverse macchine, oppure un'applicazione software. Una delle caratteristiche principali del Cloud Computing è la capacità di fornire risorse astratte, oppure risorse virtualizzate fornite come un servizio e con una propria interfaccia. Le tipologie principali di CC sono:

- **IaaS**: Essi sono dei servizi di cloud computing che forniscono APIs di alto livello utilizzate per dereferenziare vari dettagli di basso livello dell'infrastruttura di rete considerata. Questi servizi sono sostenuti da data centers su larga scala costituiti da migliaia di computers. Tutte le istruzioni di un programma sono eseguiti all'interno di una macchina virtuale: questa fornisce isolamento, sicurezza, affidabilità e delle migliori prestazioni. Un'altra caratteristica fondamentale di tale infrastruttura è la mobilità dell'applicazione. Questo viene raggiunto incapsulando il sistema operativo ospite all'interno della macchina virtuale e consentendo di essere sospeso, completamente serializzato, migrato ad una piattaforma differente e ripristinato immediatamente oppure memorizzato per un futuro ripristino. Ci sono due tipologie di macchine virtuali: **native** ed **emulate**. Le macchine virtuali native sono costituite da una macchina logica che viene eseguita sulla stessa macchina fisica. Macchine logiche e fisiche possiedono lo stesso ISA e le istruzioni sulle macchine fisiche sono in gran parte eseguite su una CPU reale. Nella macchine virtuali emulate, le macchine fisiche e

logiche sono differenti e possono avere ISA differenti. E' presente un hardware e un livello di linguaggio (Java) di emulazione.

- **PaaS**: Essa è una piattaforma che offre un ambiente in cui gli sviluppatori creano e rilasciano le applicazioni. Essa supporta differenti linguaggi di programmazione (Java, Ruby) e databases. Gli utenti posso decidere la dimensione della macchina virtuale, la locazione, etc. e avere una console web per creare applicazioni. Loro possono utilizzare un IDE per sviluppare un'applicazione ed utilizzare un SDK o CLI per rilasciarla. I vantaggi principali di tale tipologia di CC sono: consente programmazione di alto livello riducendo la complessità, l'intero processo di sviluppo dell'applicazione è più efficiente visto che utilizza un'infrastruttura built-in, la manutenzione dell'applicazione è semplice. Alcuni svantaggi sono: gli sviluppatori potrebbero non essere in grado di utilizzare tutti gli strumenti convenzionali, ritrovandosi così bloccati in determinate piattaforme. Un esempio di PaaS è il motore delle app di Google; infatti, esso è un servizio cloud per eseguire applicazioni web nel data center di Google. Ha una trasparente scalabilità, semplice configurazione e supporta diversi linguaggi di programmazione. Le applicazione vengono eseguite all'interno di una sandbox per ragioni di sicurezza.

- **SaaS**: SaaS è un modello di cloud computing nel quale un provider ospita delle applicazioni e le rende disponibili ai clienti in rete. Esso offre un elevata scalabilità e dei pagamenti flessibili, il quale consente ai clienti di accedere a dei servizi addizionali su richiesta. Molte applicazioni SaaS possono essere eseguite direttamente da un browser web, senza alcuni download o installazione richiesta, richiedono solamente alcuni plugins. Con SaaS è semplice per le aziende ottimizzare la loro manutenzione e il supporto, visto che ogni cosa può essere gestita dai fornitori: applicazioni, dati, middleware, virtualizzazione, etc. Un esempio di SaaS sono le applicazioni di Google.