# UNIVERSITÁ DI BOLOGNA

## SCUOLA DI INGEGNERIA E ARCHITETTURA

---

Corso di Laurea in Ingegneria Informatica
Laboratorio di Amministrazione di Sistemi


# Caricamento dinamico di componenti software realizzati in Kotlin in ambiente Android


Tesi di Laurea di
Matteo Pellegrino

Relatore
Prof. Marco Prandini

Correlatore
Dott. Andrea Melis


---


Anno accademico 2017-2018

*Ai miei genitori*

**Sommario**

La società moderna è stata travolta da un'inarrestabile onda tecnologica alimentata dai dati sensibili degli utenti. Una vastità di prodotti software poggiano su di un modello di business legato intrinsecamente ai dati più o meno personali forniti dagli utenti durante il normale utilizzo del prodotto stesso. Questo è trasversale alle varie aziende produttrici, a prescindere dalla loro natura, dimensione e obiettivi di mercato. Questa tesi vuole essere un approccio tecnico, con lo scopo di fornire uno strumento attraverso il quale sia possibile ottenere gli stessi risultati di elaborazione delle informazioni senza che i dati personali, necessari alla computazione, vengano inviati ad entità software estranee al controllo dell'utente o quanto meno limitarne la dispersione. La tesi è strutturata come segue:

Il primo capitolo analizza nel dettaglio il problema ed espone gli ambienti tecnologici volti alla soluzione.

Nel secondo capitolo viene esposta la ricerca e lo studio di un'architettura per la soluzione al problema. Sono qui marcate le scelte di progetto conseguite a seguito di esperimenti fallimentari.

Il terzo capitolo illustra in modo approfondito lo studio effettuato sulla sicurezza informatica nell'ambito utile alle esigenze di questa tesi.

Il quarto capitolo descrive l'implementazione effettuata, risultato pratico di questo elaborato. Si definisce qui inoltre il modo d'uso del prototipo creato e le direttive per contribuire al progetto.

Nel quinto capitolo sono tratte le conclusioni e gli sviluppi futuri.

# Indice

# Capitolo 1

# Introduzione

## 1.1 Analisi del problema

Nel quotidiano uso di determinati servizi software l'utente medio, inconsapevolmente, fornisce i propri dati sensibili. Questo fenomeno è esplicito nei sistemi mobili (smartphone): una moltitudine di applicazioni elabora i dati personali al di fuori del dispositivo, implicando la loro trasmissione, la cui sicurezza e protezione dipendono direttamente dal fornitore del prodotto software, secondo sua discrezione. Si vuole dunque trovare un'alternativa architetturale secondo cui sia possibile recuperare il codice d'elaborazione e portarlo all'utilizzo dell'utente sul proprio dispositivo. Il processo di recupero deve essere sicuro e prudente affinché il codice sia autenticato. L'analisi del problema è di carattere generale, così che lo sia la ricerca della soluzione e la sua implementazione. Ad ogni modo lo studio qui effettuato sarà propenso ad un progetto specifico, volto alla creazione di un'applicazione per la mobilità urbana.

## 1.2 Android

Android è un sistema operativo mobile sviluppato da Google e basato sul kernel di Linux. L'intero codice è open source: la possibilità di creare una propria distribuzione, a partire da quella originale, ha causato una forte diversificazione nel mercato, soprattutto riguardo l'interfaccia grafica. I dispositivi mobili Android sono attualmente i più venduti a livello globale[1] (Figura 1.1).

Android sfrutta una macchina virtuale come ambiente di esecuzione, da cui i vantaggi di avere il codice del nucleo separato da quello delle applicazioni e indipendenza dalla piattaforma. Questa Dalvik Virtual Machine (DVM) è simile alla più nota Java VM (Figura 1.2a): oltre a compilare i sorgenti in Java Byte code, esegue un ulteriore processo di compilazione (DEX) per creare Dalvik Byte code. Durante l'esecuzione di ogni applicazione i file

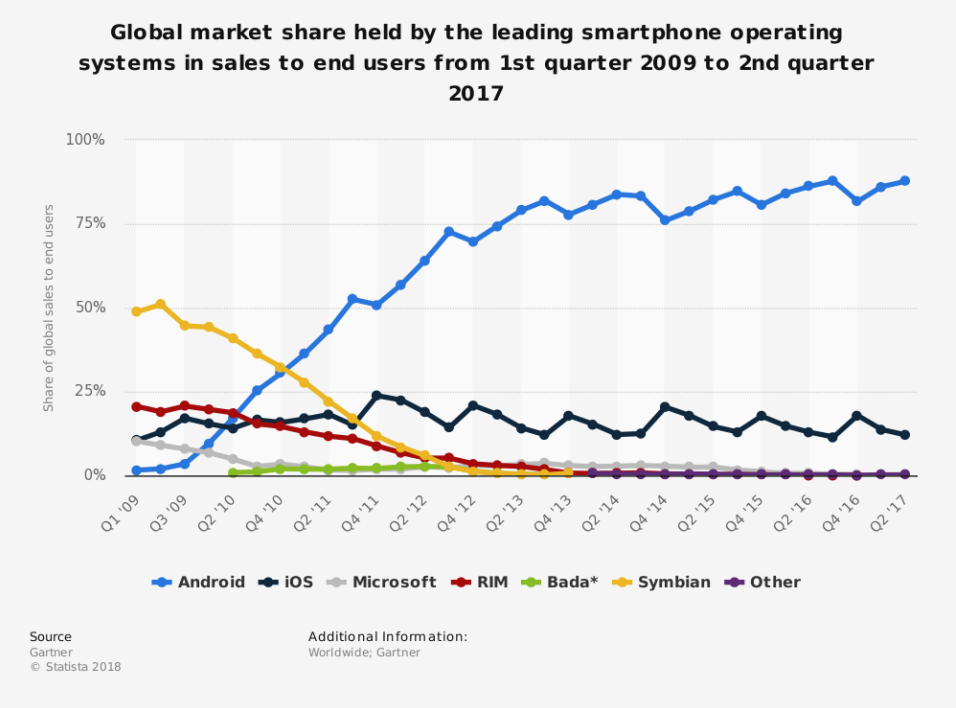
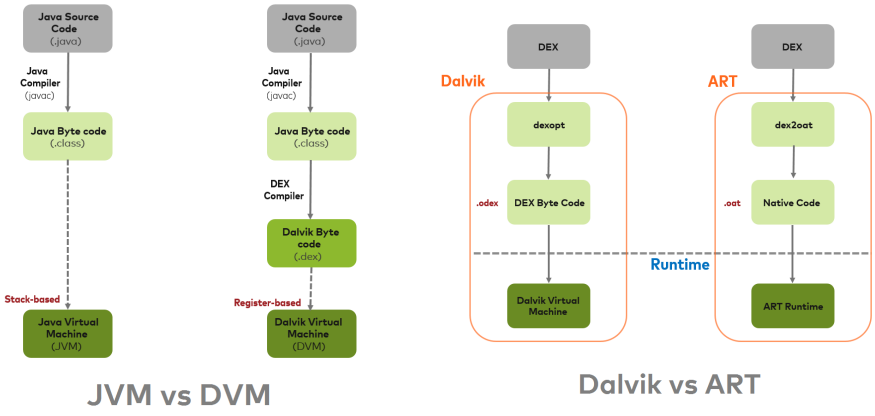



Figura 1.1: Quota globale di mercato dei sistemi operativi mobili nelle vendite agli utenti finali dal 1° trimestre 2009 al 2° trimestre 2017

di Dalvik Byte code vengono tradotti in codice macchina. Il compilatore DEX viene eseguito ogni volta che una determinata app è in esecuzione, traduce parte del codice byte code man mano che vi è la necessità di utilizzo, ottenendo così un comportamento incrementale in termini di memoria cache utilizzata. Questo tipo di compilatore è detto JIT (Just in time).

Dalla versione 4.4 (Android KitKat) è stato introdotto ART (Android runtime) che ha sostituito completamente l'ambiente di esecuzione basato su DVM (Figura 1.2b). ART possiede un compilatore di tipo AOT (Ahead Of Time), secondo il quale il codice macchina viene prodotto nell'installazione dell'applicazione e non più durante la sua esecuzione[2]. I vantaggi principali riscontrati sono maggiore velocità di esecuzione, avvio delle applicazioni più rapido, prestazioni di batteria migliorate e garbage collector più avanzato. A discapito vi è un'installazione più corposa e una occupazione di spazio maggiore (il codice macchina è tradotto una sola volta e salvato in modo persistente).




(a) Java vs Dalvik VM

(b) DVM vs ART

Figura 1.2: Ambiente di esecuzione Android

## 1.3 Kotlin

Figura 1.3: Logo Kotlin

Kotlin è un nuovo linguaggio di programmazione tipizzato staticamente sviluppato da JetBrains, leader nel campo IDE (Integrated Development Environment). Il progetto è iniziato nel 2010 e fin da subito è stato reso open source. La prima pubblicazione ufficiale è avvenuta nel Febbraio 2016, con un grande riscontro positivo da parte della comunità. Kotlin ha sia costrutti orientati agli oggetti che funzionali, potendo persino mischiarli. Questo nuovo linguaggio risulta al 100% interoperabile con Java, conseguenza del suo stato dell'arte. Infatti Kotlin compila Java bytecode per la JVM, così come Java. Questo ha permesso fin da subito un pratico utilizzo nella comunità, inoltre Google supporta ufficialmente Kotlin come linguaggio di prima classe in ambiente Android. Il principale IDE per Android ha come nucleo IntelliJ, prodotto da JetBrains e base di ogni altro suo derivato. Per questo motivo si può ben dedurre che il supporto è eccellente, favorendo una veloce diffusione del linguaggio. Kotlin non migliora solo i punti sfavorevoli di Java, ma volge come uno strumento con più ampio raggio di applicazione. Attualmente può creare applicazioni per Android, JVM, browser (compilando Javascript) e applicazioni Native, inoltre possiede in fase sperimentale il supporto multipiattaforma (write once run anywhere). La comunità ha risposto bene: sempre più librerie e tool vengono sviluppati in Kotlin, per una migliore interoperabilità. Visti i progressi, si è deciso dunque di utilizzare Kotlin per l'implementazione tecnica di questa tesi, oltre a motivi pratici per

un migliore sviluppo del codice. Riassumiamo di seguito i principali vantaggi [3], il cui approfondimento non è di nostro interesse in questo studio:

- Sintassi famigliare
- Interpolazione di stringhe
- Inferenza di tipo
- Cast intelligenti (*auto-casts*)
- Equals impliciti
- Argomenti di default e nominati
- Blocchi sintattici più leggibili (espressione *when*)
- Data Class
- Overloading di operatori
- Dichiarazioni distruttive
- Ranges
- Funzioni di estensioni
- Null safety
- Lambdas migliori

# Capitolo 2

# Architettura

## 2.1 Panoramica

In questo capitolo sono mostrate l'ideazione e la scelta architetturale del progetto tenendo presente le tecnologie obiettivo descritte nell'introduzione. L'idea di base è la creazione di un modulo, un componente software riutilizzabile, che permetta a qualsiasi applicazione Android di usufruire delle funzionalità implementate. Secondo questi criteri il modulo definisce delle interfacce (API) che ne descrivono il comportamento e ne indicano il modo d'uso: tutti gli strumenti (classi e funzioni) necessari all'utilizzatore sono esposti pubblicamente, mentre quelli legati all'implementazione sono nascosti e non recuperabili. In questo modo si ottiene un giusto equilibrio tra rigidità e flessibilità: l'utilizzatore non può alterare l'implementazione compromettendola, ma può al contempo estenderne le funzionalità. Questo modulo dovrà permettere di scaricare dinamicamente del codice via rete e offrire il supporto per poterlo eseguire a tempo di esecuzione.

L'architettura (figura 2.1) prevede dunque tre componenti: un modulo, un'applicazione ed un servitore. L'applicazione si interfaccia con il modulo per richiedere determinate funzionalità, mentre il servitore espone via rete il codice da scaricare. Il modulo e il servitore devono essere tra di loro interoperabili, ovvero è necessario specificare il dialogo tra i due, dando la possibilità all'utilizzatore di definirne uno proprio. Il modulo effettua una richiesta al servitore, al seguito del quale avviene il download del codice che verrà reso disponibile all'applicazione dinamicamente.

In accordo con le migliori pratiche della comunità di sviluppatori Android, il modulo prevede al suo interno un'applicazione d'esempio per mostrare dei casi d'uso tipici. Il progetto è open source all'indirizzo `https://github.com/arabello/KSLL`

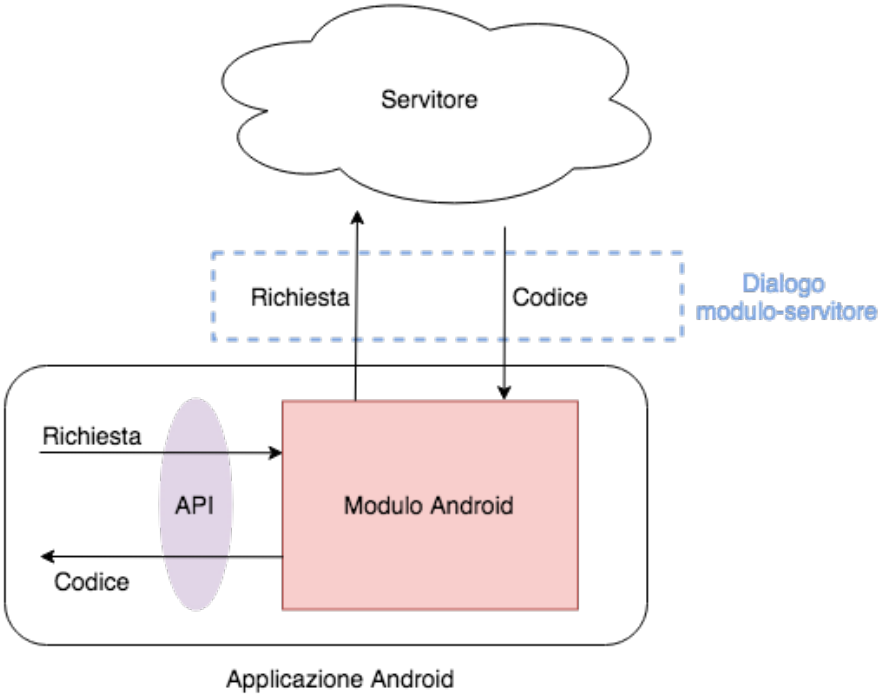



Figura 2.1: Panoramica architetturale

## 2.2 Kotlin Script Engine

Contrariamente a quel che si può immaginare, data la stretta parentela con Java, Kotlin risulta essere anche un linguaggio di scripting. Infatti può essere usato per compiti di automazione del sistema operativo o per creare delle macro, similmente al Bash, e può essere interpretato o valutato come può esserlo Javascript. Questa proprietà intrinseca del linguaggio è rilevante per il problema che dobbiamo affrontare. Se l'obiettivo è quello di trasmettere il codice ed eseguirlo a tempo di esecuzione, l'utilizzo di un linguaggio interpretato è favorevole principalmente per due motivi: innanzitutto il contenuto della trasmissione è ridotta pressoché alla sola sequenza di caratteri che esprime il codice da eseguire, inoltre l'esecuzione risiede in una semplice valutazione, messa a disposizione dal linguaggio stesso. Java supporta da sempre la valutazione di linguaggi di scripting attraverso il package `javax.script`, con la possibilità di estenderne le funzionalità a nuovi linguaggi. Infatti con la versione 1.1 di Kotlin è stato integrato il supporto alle API di `javax.script`, meglio note come `JSR-223`.

Grazie a questo supporto è possibile caricare il Kotlin Script Engine built-

in ed invocare delle valutazioni di codice Kotlin. Purtroppo non vi è tutt'ora una soddisfacente documentazione a riguardo, ancor meno per la portabilità su Android. La repository ufficiale [4] di Kotlin offre una classe di test su `JSR-223` in ambiente JVM, un buon punto di partenza per i nostri casi d'uso.

### 2.2.1 JSR-223 Problemi di prestazioni e portabilità

La classe di test fornita ufficialmente poggia su JVM. Effettuiamo dunque i primi esperimenti sul medesimo ambiente, tralasciando per il momento la complessità che potrebbe intercorrere in Android. Oracle, creatore di JSR-223, offre precise API [5] a riguardo; consultandole è possibile riassumere il principale flusso di utilizzo:

1. Creare un oggetto `ScriptEngineManager`
2. Recuperare un oggetto `ScriptEngine` dal gestore
3. Valutare lo script usando il metodo `eval()` dell'oggetto `ScriptEngine` prima ottenuto

Ovviamente l'implementazione di `javax.script.ScriptEngine` necessaria al funzionamento non è di nostra competenza, ma del supporto Kotlin offerto dalla versione 1.1.

Dopo una corretta impostazione delle dipendenze (`kotlin-compiler` e `kotlin-script-util`) e di alcune classi di test accessorie si prova il funzionamento con successo. Tuttavia l'esito architetturale è sfavorevole. Si nota che con semplici script di prova dal basso overhead il tempo di valutazione è veramente alto. Ad esempio, prendendo in esame il seguente test dalla classe ufficiale Kotlin il tempo di valutazione è di circa 6 secondi come riportato nella figura 2.2.

```
1 @Test
2     fun testSimpleEval() {
3         val engine =
              ScriptEngineManager().getEngineByExtension("kts")!!
4         val res1 = engine.eval("val x = 3")
5         Assert.assertNull(res1)
6         val res2 = engine.eval("x + 2")
7         Assert.assertEquals(5, res2)
8     }
```

Queste prestazioni non risultano per nulla adeguate alla nostra situazione, alla quale dovremmo oltretutto aggiungere i tempi di ricezione del codice attraverso internet. Inoltre, come anticipato, stiamo effettuando questi test sulla JVM di un calcolatore di moderna generazione. Queste prestazioni su

Android possono solo peggiorare in quanto, in media, i dispositivi mobili possiedono risorse hardware minori.

Proseguiamo comunque il nostro iter e tentiamo di portare la corrente struttura su Android. Come specificato prima per il funzionamento del motore di interprete sono necessarie due dipendenze, una delle quali (`kotlin-compiler`) sembra non essere compatibile con l'ambiente Android. Infatti alla prima compilazione si può notare l'errore seguente:

```
Caused by: com.android.builder.dexing.DexArchiveBuilderException:
Failed to process [...]/kotlin-compiler.jar
```

L'errore è causato da una classe facente parte del processo di dexing spiegato nell'introduzione. La dipendenza non risulta compatibile in quanto non predispone affatto dei file DEX necessari, addirittura Android non riesce a crearli di per sé.

Alla luce di problemi di prestazioni e compatibilità non resta che cercare una soluzione alternativa, radicalmente diversa da questa appena affrontata. Per convincerci di questa scelta si riporta un commento [6] di Dmitry Jemerov, membro del team JetBrains di Kotlin riguardo proprio la dipendenza in oggetto e le prestazioni discusse su Android:

> You can try packaging the Kotlin compiler as an Android app, and this could work, but there is no ready-made library that you can simply plug in into your app. Also note that the Kotlin compiler has fairly high CPU and memory requirements, which means that compiling even small files on old Android devices could take minutes. I don't know what problem you're trying to solve, but I'm pretty sure that running the Kotlin compiler on the user's phone is not the optimal way to do that.

Tradotto

> Puoi provare ad impacchettare il compilatore Kotlin come un app Android, e potrebbe funzionare, ma non esistono librerie apposite pronte all'uso che puoi semplicemente inserire nella tua app. Inoltre nota che il compilatore Kotlin ha requisiti di CPU

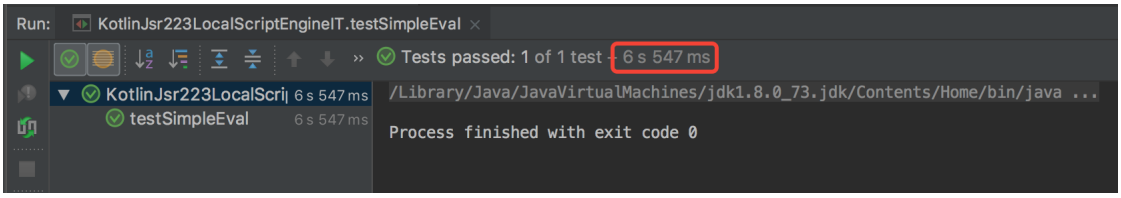



Figura 2.2: Tempo di esecuzione di un metodo di test

> e memoria notevolmente alti, ciò significa che compilare perfino piccoli file su vecchi dispositivi Android potrebbe richiedere minuti. Non conosco il problema che stai cercando di risolvere, ma sono abbastanza sicuro che eseguire il compilatore Kotlin sul telefono di un utente non è la soluzione ottimale.

## 2.3 Caricamento di librerie a tempo di esecuzione

Troviamo ora un'alternativa che possa in qualche modo evitare le problematiche riscontrate nella sezione precedente:

- Prestazioni
- Portabilità

Riguardo la prima è sufficiente sostituire gli script pensati inizialmente con del codice Kotlin compilato. Riuscendo ad invocare metodi di classi compilate le prestazioni sono paragonabili a quelle che si avrebbero in un normale programma. L'unica differenza è quella di non conoscere a priori il numero, il nome e l'implementazione delle classi compilate. Tutto ciò deve avvenire a tempo di esecuzione tramite una pratica abbastanza comune denominata *reflection* (riflessione). Quasi ogni linguaggio, tra cui Kotlin, predispone di classi dedicate ad eseguire processi di reflection, tramite le quali è possibile leggere ed invocare a tempo di esecuzione codice non conosciuto a priori. Sebbene la reflection sia di sua natura estremamente flessibile e adattabile ad ogni situazione, si vuole introdurre nel nostro progetto un filo di rigidità. Per esempio possiamo esigere che le funzionalità siano reperibili da un'unica classe, il cui nome viene specificato dal fornitore del codice, semplificando estremamente l'utilizzo della reflection e mitigando la necessità di specificare una moltitudine di classi di cui non ne si conosce a priori l'interoperabilità. Questa classe diverrebbe un *Service Access Point* (punto di accesso al servizio) e in questo modo un qualsiasi terzo, intento a creare una libreria per il sistema, avrebbe bisogno solo del nome della classe. L'implementazione è a sua discrezione e le funzionalità esposte sono semplicemente i metodi pubblici della classe SAP (Service Access Point). Per riassumere, il progetto di studio è un modulo Android scritto in Kotlin capace di reperire dinamicamente librerie messe a disposizione da servitori omogenei, che dovranno dialogare con il modulo per esporre parametri utili al caricamento delle librerie stesse.

Queste possono essere scritte sia in Kotlin che in Java, infatti, per i motivi illustrati nell'introduzione, la loro compilazione crea ugualmente bytecode interpretabile dalla JVM. Malgrado ciò, bisogna avere l'accortezza di predisporre la libreria creata affinché sia leggibile in ambiente Android, compilandola tramite processo di dexing. Questo avviene in automatico se l'ambiente di sviluppo è predisposto e vengono forniti le opportune impostazioni a riguardo. Per esempio IDE come Android Studio, IntelliJ IDEA

e Eclipse possono essere usati per creare librerie che compilino per Android. Altrimenti è possibile effettuare la procedura manualmente: dopo aver scaricato l'Android SDK (Software Development Kit), è sufficiente invocare il comando relativo come segue:

```
1 $SDK_HOME/build-tools/X.X.X/dx --dex --output="ouput.jar"
    "input.jar"
```

Per togliere ogni ambiguità futura nominiamo il modulo Android oggetto di questo progetto come *Kotlin Safety Library Loader* (Caricatore sicuro di libreria Kotlin) in breve *KSLL*. Il nome stesso racchiude gli obiettivi prefissati in questa sezione:

**Kotlin** Uso di un linguaggio moderno e fortemente versatile

**Safety** Il codice scaricato deve essere autenticato e fidato

**Library** Il codice è racchiuso in librerie adeguatamente formate

**Loader** Compito ultimo del modulo, caricare le librerie reperite

### 2.3.1 Schema architetturale

In figura 2.3 è riportato il funzionamento di KSLL e delle sue interazioni con l'esterno. Si nota anzitutto l'introduzione di una nuova entità denominata *Internal Storage* (archiviazione interna); essa corrisponde allo strumento Android per la persistenza dei dati, necessaria al mantenimento delle librerie. Si elenca di seguito il flusso di azioni in ordine temporale:

1. Un utilizzatore richiede una libreria remota identificandola tramite un *URL*. Questo rappresenta un punto di accesso di un servitore al quale è possibile richiedere un dialogo secondo le specifiche.

2. KSLL interroga il servitore la prima volta e attende una risposta ben formata il cui contenuto definisce dei *meta dati*. Questi sono utilizzati per la corretta gestione della libreria e saranno descritti nel dettaglio nel capitolo dedicato all'implementazione. KSLL verifica se la libreria è già stata scaricata precedentemente. Questo in previsione di un meccanismo di cache, di notevole utilità se siamo sicuri che la libreria da dover scaricare non differisca da quella pronta localmente. In tal modo sarà possibile avere un incremento delle prestazioni annullando il tempo di attesa del download. Se la libreria non è pronta localmente si salta al punto 3, altrimenti avviene un'ulteriore verifica riguardo la sua versione. Si prevede un impostazione di KSLL secondo il quale è possibile indicare se la libreria deve essere aggiornata o no. Sulla base di questa impostazione e dell'effettivo confronto tra la versione locale con quella remota, si dispone il download o un salto al punto 5.

3. KSLL si occupa di interrogare nuovamente il servitore, questa volta per l'effettivo download della libreria. Una volta scaricata, avviene il controllo della firma digitale riguardo i dati ottenuti. Se, e solo se, la libreria è ritenuta autentica e fidata allora si prosegue con la memorizzazione al punto successivo, altrimenti viene scartata e sollevata un'eccezione.

4. La libreria è considerata valida, quindi viene memorizzata in via persistente sulla memoria di massa interna del dispositivo. Sarà necessario creare un percorso unico sulla base dei meta dati ottenuti e l'aggiunta di uno o più file accessori per la memorizzazione di questi.

5. Una volta assicurato di avere i file necessari si prosegue all'effettivo caricamento della classe SAP utilizzando il meccanismo di reflection.

L'implementazione delle librerie non è rilevante al fine di ottenere compatibilità con il modulo, ma è sufficiente che sia omogenea l'interfaccia e che si rispettino le specifiche del servitore. Questo rende KSLL riutilizzabile in sistemi differenti. Legittimamente si può assumere che nello stesso sistema

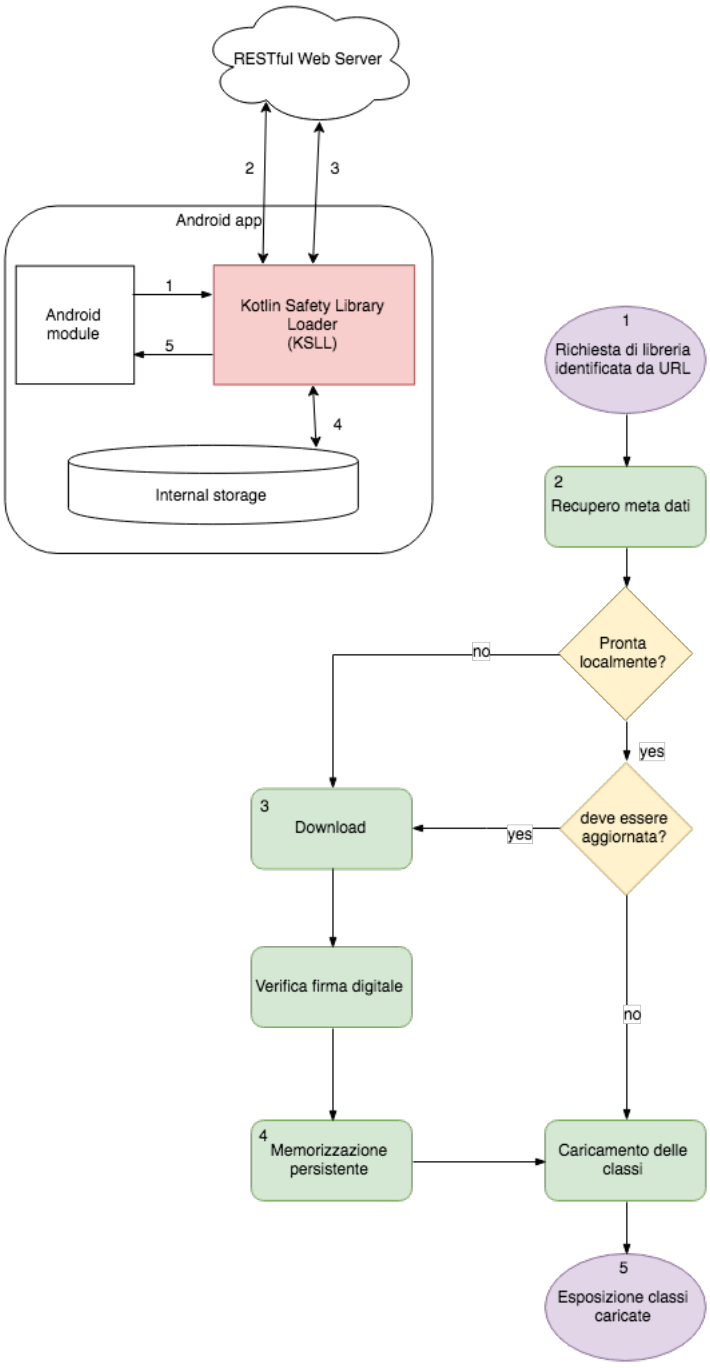



Figura 2.3: Schema di funzionamento previsto

le librerie siano simili ed espongano metodi con firme uguali, lasciando la loro implementazione a ciascun servitore. Per esempio il sistema per l'applicazione di mobilità urbana, citata nell'introduzione, prevede una libreria per ciascun servitore i cui metodi hanno la stessa firma nota, mentre il codice interno può non essere uguale. L'applicazione usa quindi KSLL per reperire dal servitore della zona attuale la corretta implementazione della libreria ed invocarne le funzionalità.

### 2.3.2 Specifiche del servitore

Il servitore deve esporre delle informazioni essenziali, di seguito elencate.

**URL**
: Indirizzo di una risorsa web da cui poter scaricare i dati binari effettivi della libreria.

**Nome della classe SAP**
: Il nome completo di una classe pubblica presente all'interno dalla libreria e che opera come punto di accesso al servizio. Fondamentale per il processo di reflection e per il caricamento delle funzionalità a tempo di esecuzione.

**Versione**
: Codice rappresentante la versione corrente. Tramite questo valore KSLL può decidere se effettuarne l'aggiornamento rispetto alla libreria pronta localmente.

**Estensione**
: Formalmente le librerie contenti file DEX hanno estensione `.dex`, `.jar`, `.zip` o `.apk`. Sebbene non sia un vincolo stringente questa informazione può essere usata per salvare il file usando un formato standard.

**Firma**
: Impronta della libreria risultato di una *funzione di hash*. Deve esserci accordo tra il modulo e il servitore riguardo la codifica di caratteri usata per la sua rappresentazione. I dettagli verranno discussi nella sezione dedicata alla sicurezza.

### 2.3.3 Persistenza in Android

Android fornisce diverse soluzioni per salvare i dati relativi a un'applicazione. La scelta dipende dalla situazione specifica, dal tipo di dati e dalla loro dimensione e accessibilità. Le soluzioni previste sono *archiviazione di file interni*, *archiviazione di file esterni*, *preferenze condivise* e *database*.

**Archiviazione di file interni** Tramite questo tipo di memorizzazione è possibile salvare in modo persistente file di qualsiasi tipo in uno spazio privato visibile e accessibile solamente dalla nostra applicazione. Il sistema Android prevede una cartella privata per ogni applicazione installata sul dispositivo e delle API per permette la lettura, scrittura e modifica su di essa. Con questo tipo di archiviazione, i dati saranno persi nel momento in cui l'applicazione viene disinstallata.

**Archiviazione di file esterni** A differenza della precedente soluzione questo tipo di archiviazione non è privato, ma accessibile sia da altre applicazioni che dall'utente stesso. L'accesso non è sempre garantito in quanto si basa su una memoria persistente esterna, ovvero fisicamente rimovibile (SD card o memoria USB). Rappresenta un ottima alternativa all'archiviazione privata per file di notevole grandezza o se vi è la necessità di condividere risorse con l'utente e le altre applicazioni.

**Preferenze condivise** Nel caso le informazioni da memorizzare siano veramente poche e dalla struttura primitiva come booleani, interi o stringhe, Android mette a disposizioni delle API per la memorizzazione persistente di coppie chiave-valore. La gestione è completamente a carico del sistema, limitandone la flessibilità.

**Database** Android fornisce supporto alla tecnologia *SQLite*, permettendo la memorizzazione di dati tramite database e tutte le funzionalità relative come query e viste.

La soluzione più adeguata alle nostre esigenze, come suggerisce anche il portale ufficiale [7] per lo sviluppo Android, è l'*archiviazione di file interni*: permette facilmente di salvare con visibilità privata le librerie, la cui grandezza è contenuta.

# Capitolo 3

# Indagine sulla sicurezza

## 3.1 Cenni di crittografia

Il termine crittografia deriva dall'unione di due parole greche, *kryptós* (nascosto) e *graphía* (scrittura). Indica lo studio della codifica e decodifica delle informazioni. La crittografia riveste un ruolo fondamentale nella sicurezza informatica ponendosi come soluzione principale a molte problematiche connesse alla protezione delle informazioni. In generale, per raggiungere un desiderato livello di sicurezza, è opportuno riferirsi a cinque categorie di funzionalità:

**Confidenzialità** Permette di nascondere il contenuto di un'informazione tramite cifratura dei dati, in modo tale da poter essere letto solamente dal destinatario legittimo.

**Integrità** Questa funzionalità assicura che i dati non siano alternati durante la comunicazione, a prescindere dal fatto che possano essere cifrati o meno.

**Autenticazione** Non meno importante è la verifica dell'origine delle informazioni, intesa come autenticazione dell'entità che le ha generate. Permette di avere la certezza che l'altro capo della comunicazione sia veramente l'entità con cui si pensa di essere in connessione.

**Controllo degli accessi** In sinergia con la precedente, è possibile limitare i servizi o le risorse offerte secondo le autorizzazioni concesse.

**Non ripudio** I servizi che implementano questa funzionalità garantiscono che una certa entità partecipante ad una transazione o comunicazione non possa rinnegare la sua stessa partecipazione. Corrisponde alla firma autografa su documenti legali come bonifici e contratti; presenta la sua controparte crittografica attraverso la *firma digitale*.

Gli algoritmi crittografici si suddividono in due famiglie: algoritmi a *chiave simmetrica* e algoritmi a *chiave pubblica (asimmetrica)*.

## 3.2 Crittografia a chiave simmetrica

Gli algoritmi a chiave simmetrica sono stati i primi ad avere implementazioni pubbliche. Il primo vero standard fu il *Data Encryption Standard (DES)* proposto da un gruppo di ricerca IBM e approvato dalla National Security Agency (NSA) nel 1977. Prende il suo posto nel 2001 l'*Advanced Encryption Standard (AES)*, rendendo di fatto obsoleto il DES. Attualmente AES utilizza una chiave di lunghezza variabile oltre i 64 bit, a differenza di DES con chiave fissa a 56 bit.

Questo tipo di crittografia si basa su un'unica chiave matematica sia per la cifratura, che per la decifratura dei dati. Il mittente utilizza la chiave per cifrare il messaggio e inviarlo in totale sicurezza. Il destinatario utilizzerà la medesima chiave per decifrarlo e ottenere il contenuto in chiaro.

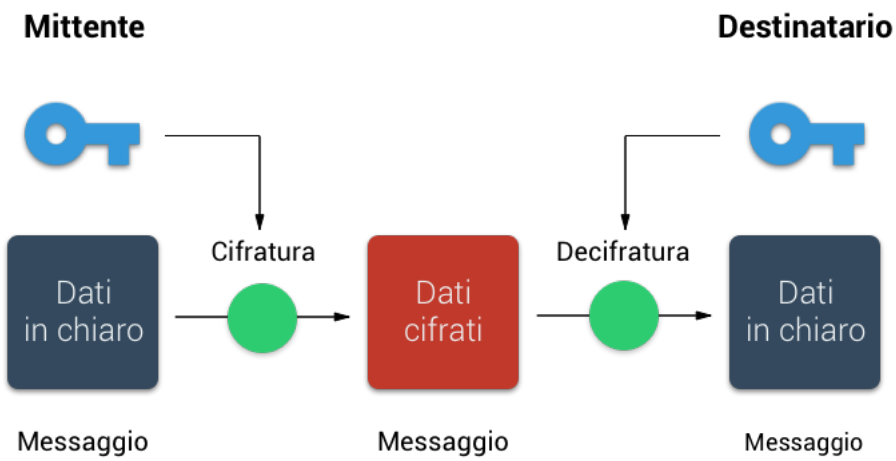



Figura 3.1: Schema di funzionamento di un algoritmo a chiave simmetrica

Il principio matematico alla base di questo algoritmo è l'applicazione della funzione di decifratura sul messaggio $M$ cifrato. Questo processo funziona se le due funzioni di cifratura $E$ e decifratura $D$ si basano sulla stessa chiave $K$.

$$D_k(E_k(M)) = M$$

Sebbene l'aspetto matematico è semplice e sicuro si pone un problema architetturale: la distribuzione della chiave. Affinché l'algoritmo funzioni la chiave deve essere conosciuta da ciascun componente della comunicazioni e da nessun altro. In un sistema ristretto, in cui la chiave può essere persino consegnata manualmente, il problema non si pone. Attualmente, con le dimensioni di internet, questo meccanismo non è fattibile, infatti se $n$ è il numero di partecipanti le chiavi totali da distribuire sono

$$\frac{n * (n-1)}{2}$$

Inoltre, nel caso in un cui un mal intenzionato riesca a prendere possesso della chiave e a sostituirsi ad un partecipante, non vi è modo di verificare l'autenticità dell'origine dei dati.

## 3.3 Crittografia a chiave pubblica

Contemporaneamente alla pubblicazione di DES, Whitfield Diffie e Martin E.Hellman introducono nel 1976 l'utilizzo di una chiave pubblica per la cifratura. Su questa base Rivest, Shamir e Adleman realizzano il primo algoritmo crittografico a chiave pubblica, denominato RSA. Il cuore del funzionamento si basa sul fatto di avere due chiavi (*asimmetria*), pubblica e privata, in modo tale che un messaggio cifrato usando quella pubblica sia decifrabile solo tramite la corrispettiva chiave privata. Le chiavi sono generate accoppiate e non esistono collisioni tra coppie di chiavi.

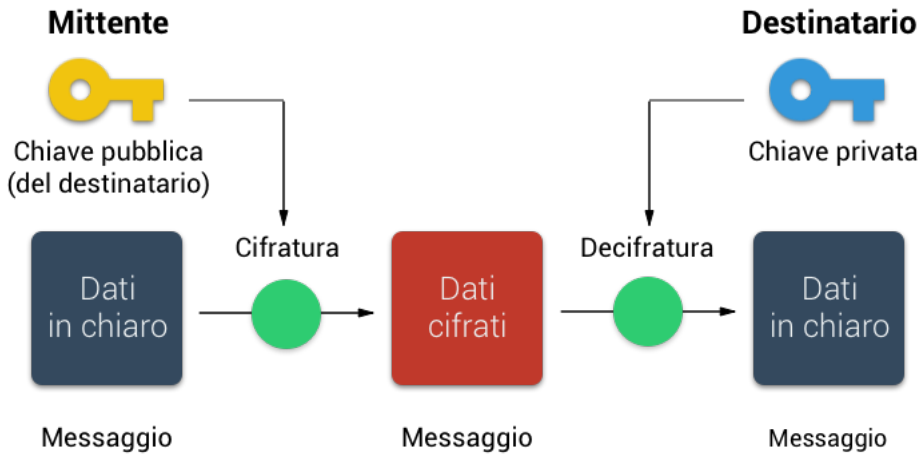



Figura 3.2: Schema di funzionamento di un algoritmo a chiave asimmetrica

Sia $M$ il messaggio in chiaro, $C$ quello cifrato, $E$ la funzione di cifratura, $D$ quella di decifratura, $K_{pub}$ la chiave pubblica del destinatario e $K_{priv}$ quella privata, il principio di funzionamento si riassume come segue:

$$C = E(K_{pub}, M)$$
$$M = D(K_{priv}, C)$$

L'enorme vantaggio è la possibilità di distribuire pubblicamente la prima chiave, mentre quella privata deve essere mantenuta da una sola entità. A

differenza della crittografia simmetrica, è possibile implementare questo algoritmo su larga scala, infatti se $n$ sono i partecipanti coinvolti, il numero di chiavi da distribuire rimane $n$.

Vediamo ora alcuni dettagli matematici riguardo l'algoritmo RSA per capire dove risiede la sua sicurezza. In questa crittografia viene utilizzata l'algebra modulare e la fattorizzazione di interi di grandi dimensioni. Per la creazione delle chiavi bisognare seguire questi passi:

1. Scegliere due numeri interi $p$ e $q$ molto grandi e primi. Il loro prodotto genera un numero $N$ detto modulo.

2. Calcolare la funzione di Eulero $T = (p-1) * (q-1)$ e scegliere un numero intero $e$ tale per cui tra $T$ ed $e$ non ci siano fattori comuni (escluso 1). La chiave pubblica è dunque $K_{pub} = (e, N)$

3. Per il calcolo della chiave privata è necessario calcolare un numero intero $d$ per cui $e * d \ mod \ T = 1$. In questo modo si ottiene $K_{priv} = (d, N)$

Sia $M$ messaggio in chiaro e $C$ il corrispettivo cifrato, le relazioni sono:
Cifrazione

$$C = M^e mod N$$

Decifrazione

$$M = C^d mod N$$

La sicurezza dell'algoritmo deriva dalla difficoltà di determinare, nell'algebra modulare, i fattori primi di un numero intero molto grande. La lunghezza della chiave determina dunque il fattore di sicurezza. Per avere un termine di paragone, a partire dal 2016, la suite di algoritmi commerciali per la sicurezza dell'NSA [8] specifica per RSA una chiave di almeno 3072 bit.

## 3.4 Firma digitale

Un documento elettronico firmato digitalmente ha la garanzia di integrità, autenticità e non ripudio. Infatti, come per la firma autografa, quella digitale permette di assicurare al destinatario che il mittente abbia deliberatamente firmato il documento, che non possa rinnegarne la paternità, che nessun altro possa aver effettuato la stessa firma sullo stesso documento e che non vi siano state alterazioni durante la comunicazione.

La firma digitale viene realizzata tramite algoritmi a chiave pubblica, con il supporto di particolari funzioni dette di *hash*. Queste permettono di generare l'impronta digitale (*digital fingerprint o digest message)* di un documento. Le funzioni di hash sono notevolmente più efficienti degli algoritmi di cifratura asimmetrici come l'RSA. La firma digitale nasce dalla congiunzione dei due:

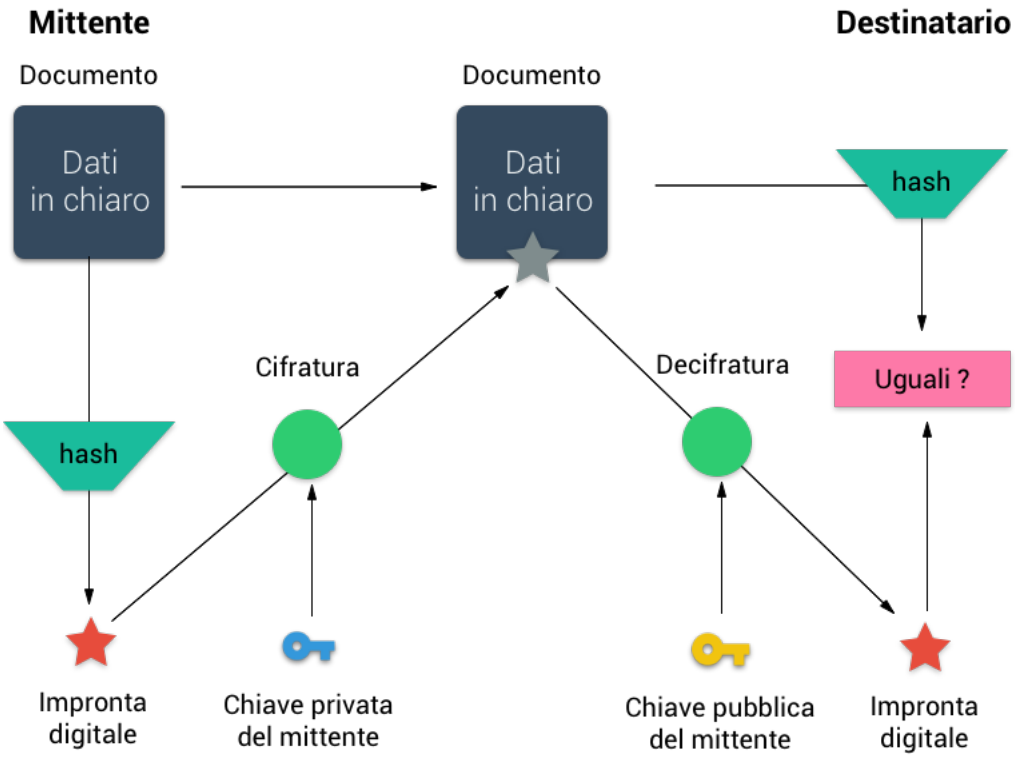



Figura 3.3: Schema di funzionamento della firma digitale

1. Viene applicata una funzione di hash ad un documento. Il risultato (impronta digitale) sarà di lunghezza fissa indipendentemente dalla grandezza del documento.
2. Tramite algoritmo RSA si cifra, usando la chiave privata, l'impronta digitale solamente.
3. Il documento (in chiaro) viene inviato insieme all'impronta digitale cifrata.
4. Qualsiasi destinatario per determinare l'autenticità e l'integrità del documento applica la stessa funzione di hash sul documento e confronta il risultato con l'impronta digitale decifrata usando la chiave pubblica del mittente legittimo. Se la verifica fallisce vuol dire che la chiave privata del mittente non corrisponde a quella pubblica in possesso al destinatario (qualcuno si è sostituito al mittente legittimo), oppure il documento è stato alterato.

La funzione di hash deve presentare due proprietà:

- Difficilmente invertibile, ovvero dato $x$ è facile calcolare $f(x)$, ma molto difficile computazionalmente risalire a $x$ partendo da $f(x)$
- Resistente alle collisioni, cioè dato $x$ e $y$ per cui $x \neq y$ allora deve essere $f(x) \neq f(y)$

Quest'ultima proprietà assicura che l'impronta digitale calcolata a partire da un documento sia sola ed unica per quel documento. Le prime versioni di certe funzioni di hash hanno riscontrato negli anni questo tipo di problema, perdendo la loro efficacia e implicando un aggiornamento o la loro deprecazione. Al giorno d'oggi i principali algoritmi per la firma digitale sono RSA e il *DSA (Digital Signature Algorithm)*, mentre le più usate funzioni di hash sono *MD5 (Message Digest 5)* e *SHA (Secure Hash Algorithm)*. I vantaggi di uno rispetto all'altro sono discussi nella sezione dedicata alle considerazioni progettuali.

## 3.5 Infrastruttura a chiave pubblica

Nei sistemi asimmetrici la distribuzione della chiave pubblica genera comunque un problema di autenticità. Non è scontato che un certa chiave pubblica appartenga effettivamente all'interlocutore con cui si vuole dialogare. Basti pensare che, nel caso in cui fosse l'interlocutore stesso a distribuire la sua chiave pubblica, è sufficiente che un malintenzionato sostituisca la sua chiave con quella dell'interlocutore e tutte le nuove comunicazioni sarebbero interamente gestite dalla terza persona. Un modello del genere sarebbe possibile se la consegna avvenisse di persona, così da conoscere veramente chi fornisce una determinata chiave. Questa conoscenza prestabilita tra interlocutori non è possibile su larga scala attraverso la rete. I *certificati elettronici* risolvono il problema della distribuzione globale delle chiavi, attraverso uno strumento affidabile, sicuro e altamente scalabile. I certificati garantiscono agli utenti finali autenticità e integrità tramite l'impiego di una *Autorità di Certificazione (Certification Authority o CA)* che li emette e gestisce. In figura 3.4 è mostrata la struttura di un certificato digitale definito dallo standard X.509 come specifica RFC5280. [9]

Le informazioni più rilevanti sono il nome univoco del possessore di una chiave pubblica, il valore della chiave, la validità temporale del certificato e la firma digitale di un'autorità di certificazione per assicurare autenticità e integrità del certificato stesso.

Il supporto necessario alla gestione delle chiavi pubbliche e dei certificati affinché possano essere usati su larga scala è fornito dalle *PKI (Public Key Infrastructure)*. Alla base dell'infrastruttura di una PKI vi è il concetto di fiducia di terze parti (*third-party trust)* secondo il quale due entità si fidano l'una dell'altra in quanto condividono una fiducia verso una terza parte comune. Questa parte è l'Autorità di Certificazione (CA).

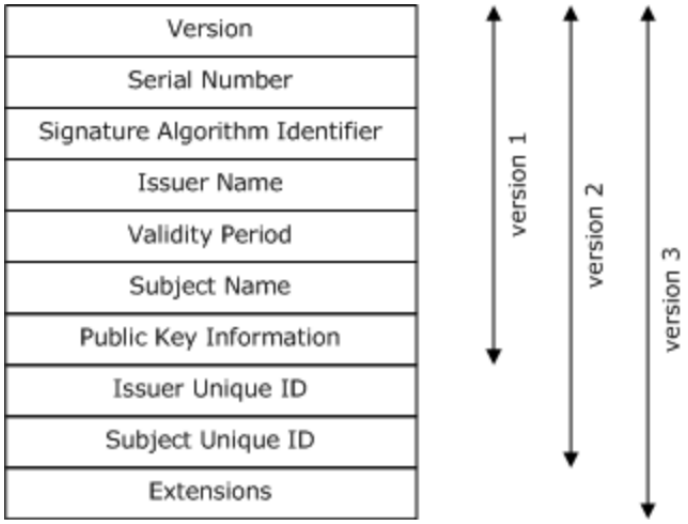



Figura 3.4: Struttura di un certificato digitale secondo X.509 [10]

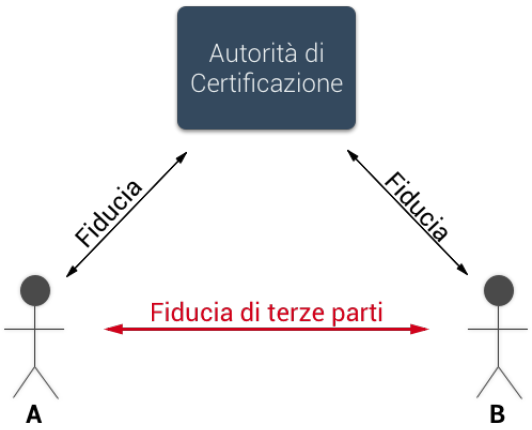



Figura 3.5: Fiducia di terze parti (*third-party trust*)

Vediamo ora i principali componenti di un'infrastruttura a chiave pubblica illustrati in figura 3.7.

**Autorità di Registrazione (Registration Authority RA)** Entità incaricata di verificare l'identità di una persona o società a seguito di una richiesta di certificato (*Certificate Signing Request CSR*). Questo permette di abilitare il richiedente in un dominio di fiducia. La funzionalità può essere effettuata dall'Autorità di Certificazione e delegata ad esterni.

**Autorità di Certificazione (Certification Authority CA)** Fulcro della PKI, gestisce l'intero ciclo di vita di un certificato: generazione, aggiornamento, sostituzione (scadenza temporale) e revoca (se le condizioni di emissione non sono più valide). Queste funzionalità non possono essere delegate ad altre entità. Compito della CA è stabilire relazioni di fiducia con altre CA. Queste relazioni a cascata generano la *catena di fiducia (chain of trust)*: un dato certificato può presentare al suo interno un riferimento ad un altro detto *intermediario*. La catena si completa con un certificato auto firmato detto radice *(root certificate)*, a cui corrisponde una root CA.

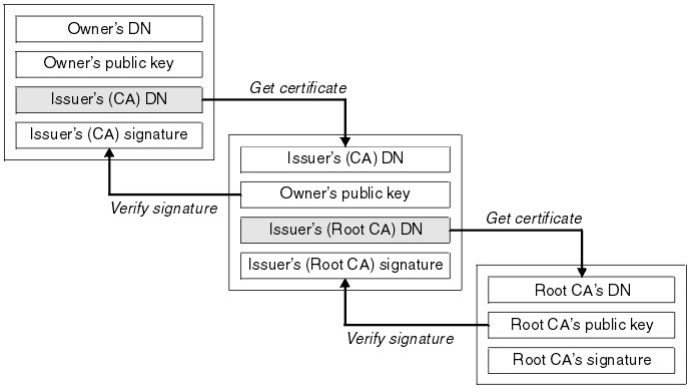



Figura 3.6: Chain of trust [11]

**Autorità di Convalida (Validation Authority VA)** Entità che offre un servizio per la validazione dei certificati. Mantiene una lista di certificati.

**Sistema di directory** Sistema fisico distribuito da cui sono reperibili i certificati a chiave pubblica e quelli revocati. Il protocollo di comunicazione è *LDAP (Lightweight Directory Access Protocol)* basato su TCP/IP.

**Database** Oltre al sistema di directory, le CA mantengono privatamente dei database con il principale scopo di backup e memorizzazione dei certificati in disuso.

**Utenti** Gli utenti finali sono i possessori di software client del sistema PKI capaci di interagire con la directory o l'eventuale CA.

## 3.6 Considerazioni progettuali

In conclusione a questa indagine vediamo come applicare i concetti di sicurezza al nostro modulo KSLL. La dimensione della libreria è da considerare contenuta, applicare dunque la cifratura dei dati con chiave asimmetrica,

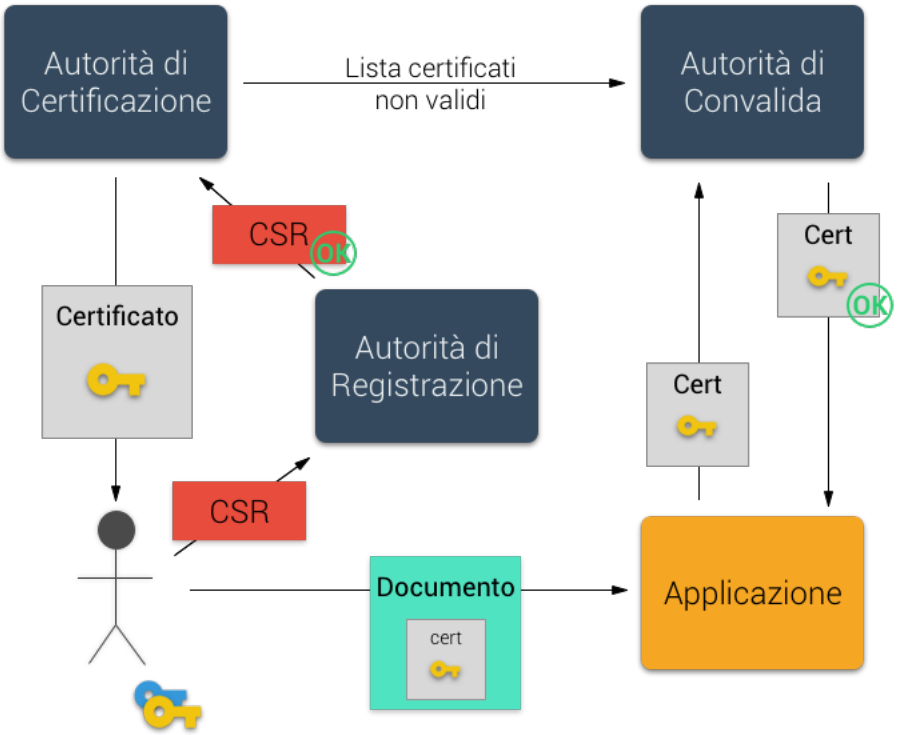



Figura 3.7: Schema di funzionamento di un'infrastruttura a chiave pubblica

ormai uno standard, comporterebbe un basso utilizzo di risorse e tempi accettabili. Tuttavia la confidenzialità intrinseca delle informazioni non è una priorità; si vuole invece conferire al sistema integrità e autenticità riguardo le librerie da reperire. Per questo motivo non aggiungiamo complessità legata alla cifratura del contenuto della libreria, ma scegliamo l'utilizzo di una firma digitale. Sono di seguito presentate le scelte progettuali riguardo l'algoritmo di firma e funzione di hash, mentre i dettagli implementativi sono riportati nel capitolo dedicato.

### 3.6.1 DSA e RSA

Sicuramente la scelta ricade su algoritmi a chiave pubblica rispetto a chiave simmetrica per tutti i vantaggi discussi precedentemente, seguendo così lo standard di fatto della comunità. Rimane la scelta tra i vari algoritmi, di cui i più noti RSA e DSA. Tralasciando gli aspetti aspetti matematici, il principale fattore di nostro interesse sono le prestazioni, in quanto a livello di sicurezza sono eguali. DSA si presenta più veloce in cifratura rispetto la

decifratura, mentre le proprietà di RSA sono esattamente opposte [12]. Nel nostro sistema la cifratura avviene un numero di volte minore rispetto alla decifratura. Infatti una volta cifrata l'impronta digitale di una libreria, finché la libreria rimane la stessa, non vi è la necessità di ripetere l'operazione. Invece la decifrazione deve avvenire ad ogni download della libreria. Per questo motivo la scelta ricade naturalmente su RSA.

### 3.6.2 MD5 e SHA

Le principali famiglie di funzioni di hash sono MD e SHA. Attualmente l'ultima versione MD5 ha problemi di collisione [13], ovvero è possibile trovare due documenti diversi a cui corrisponde la stessa funzione hash. Similmente SHA-1 è ormai deprecato in favore delle versioni più recenti. La scelta è dunque su SHA-3 con almeno 224 bit, come afferma *NIST (National Institute of Standards and Technology)*. Il massimo che si può ottenere oggi (2018) è SHA-3 a 512 bit.

Android presenta diverse soluzioni algoritmiche [14] riguardo la firma digitale, tra cui quelle qui scelte (RSA con SHA512).

### 3.6.3 PKI in Android

Il sistema Android supporta le PKI sia per tecnologie HTTPS che SSL [15]. A partire dalla versione 4.2 (Jelly Bean), contiene più di 100 Autorità di Certificazione aggiornate in ogni pubblicazione. Espone inoltre le API per permettere la gestione in caso di errore. Per esempio, il seguente codice

```
1 URL url = new URL("https://wikipedia.org");
2 URLConnection urlConnection = url.openConnection();
3 InputStream in = urlConnection.getInputStream();
4 copyInputStreamToOutputStream(in, System.out);
```

può generare delle eccezioni nel caso in cui:

- L'Autorità di Certificazione non è riconosciuta, ovvero non è presente nella lista predisposta nel sistema Android
- Il certificato è auto firmato
- Nella configurazione del servitore manca un certificato intermediario

A seguito di queste problematiche è necessario operare manualmente, altrimenti Android si occupa di tutte le validazioni e al programmatore rimane solo la logica di business propria. Nel primo caso la CA non è fidata da Android, questo può capitare soprattutto se è privata; è possibile via codice

insegnare ad Android, nel contesto della propria applicazione, a fidarsi di tale CA. La verifica logica ovviamente è a discrezione del programmatore. Questa soluzione è applicabile anche al secondo caso, il quale indica che il servitore si comporta come una propria CA. Nel terzo caso invece il servitore non ha fornito l'intera catena di fiducia, ma solo il proprio certificato e la root CA. Questa pratica, sebbene non del tutto corretta, è comune soprattutto per il risparmio di banda. In ambiente desktop questo problema è implicitamente risolto dal browser, che mantiene memoria cache delle CA intermediarie visitate da altri siti. Android invece si fida solo delle root CA [15], quindi è necessario che il servitore fornisca l'intera catena.

Android è un sistema aperto, sebbene le API di più alto livello permettono una facilità d'uso maggiore, quelle più profonde possono essere usate, con cautela, per raggiungere certe configurazioni particolari. Infatti può essere eseguita una pratica denominata *Pinning*: nel contesto della propria applicazione è possibile limitare le CA fidate a un numero ristretto, corrispondente ai soli servitori con cui l'applicazione dovrebbe dialogare. In questo modo si limitano eventuali pericoli rapportati alle numerevoli root CA predefinite nel sistema Android.

# Capitolo 4

# Implementazione

Per poter sviluppare in ambiente Android è necessario fornirsi dei giusti strumenti. La scelta dell'IDE (Integrated Development Environment) ricade su Android Studio, ufficiale e supportato da Google. L'esecuzione del codice può avvenire su dispositivo fisico o virtuale; all'interno di Android Studio vi è tutto il necessario per scaricare e lanciare un emulatore, nel caso non si disponga di uno smartphone Android, solitamente preferito per questioni di prestazioni. Il progetto che si vuole implementare non richiede particolari funzionalità, quindi la versione minima di Android supportata dal modulo può essere molto bassa. Considerando la distribuzione ufficiale [16] possiamo scegliere la versione 16 delle API come la minima supportata, coprendo di fatto il 99.7% del mercato globale. Le versioni supportate sono, in ordine crescente, *Jelly Bean, KitKat, Lollipop, Mashmallow, Nougat, Oreo.*

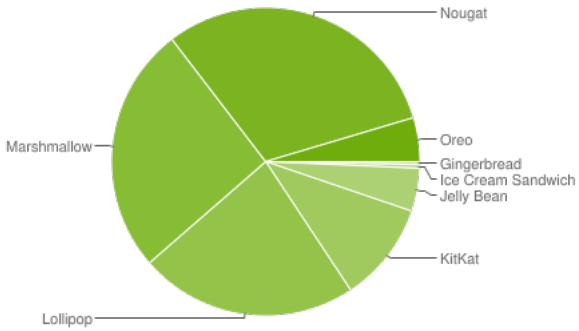



Figura 4.1: Dati raccolti durante un periodo di 7 giorni terminante il 16 aprile 2018. Qualsiasi versione con una distribuzione inferiore allo 0,1% non è mostrata. [16]

In questo capitolo con il termine utilizzatore e utente s'intende un programmatore che usufruisce del modulo KSLL per lo sviluppo di un'applicazione.

## 4.1 Struttura

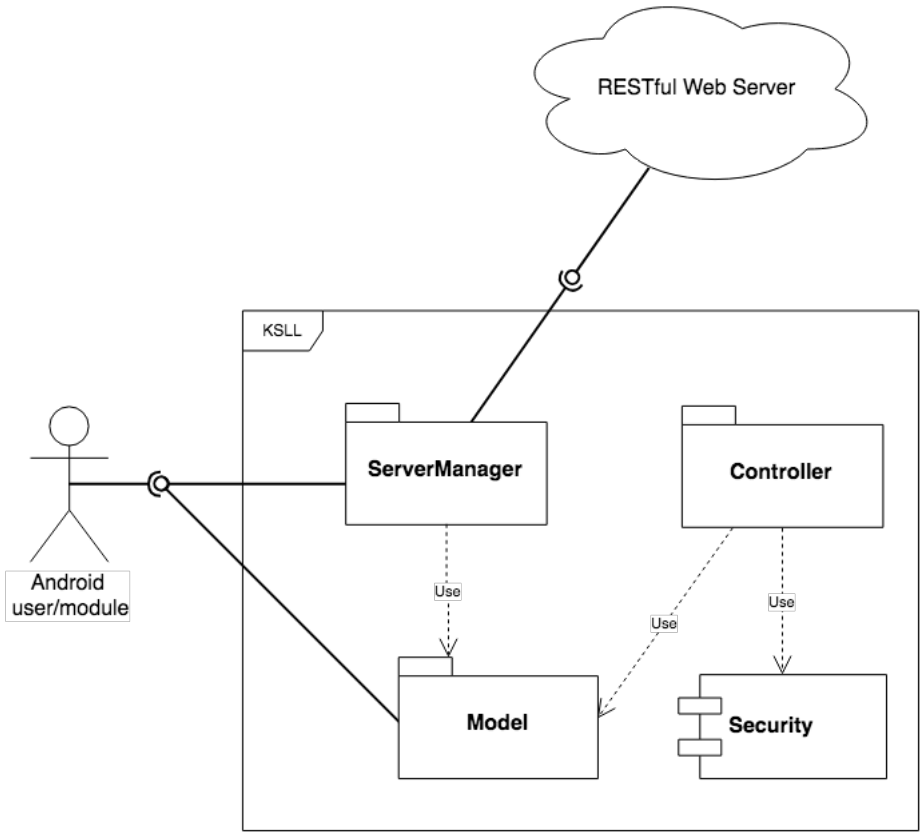



Figura 4.2: Diagramma dei package e visibilità

In figura 4.2 è rappresentata la struttura del modulo tramite diagramma dei package. Un utilizzatore esterno di KSLL ha visibilità limitata ai componenti sufficienti e necessari al corretto interfacciamento con il modulo. Il resto è nascosto e non a disposizione. Il package `ServerManager` si occupa della prima interazione con un server, allo scopo di accordarsi riguardo i metadati. Inoltre `ServerManager` è visibile all'utilizzatore, in quanto si prevede un meccanismo tale per cui si possa indicare in che modo il modulo debba interagire con il servitore, qualora l'implementazione predisposta non sia compatibile con il servitore in questione. Nel `Model` sono racchiuse le classi rappresentanti le librerie; queste sono il risultato che il modulo deve fornire all'utilizzatore, dunque visibili dall'esterno. Il package `Controller` e il sottomodulo `Security` invece racchiudono la logica di business.

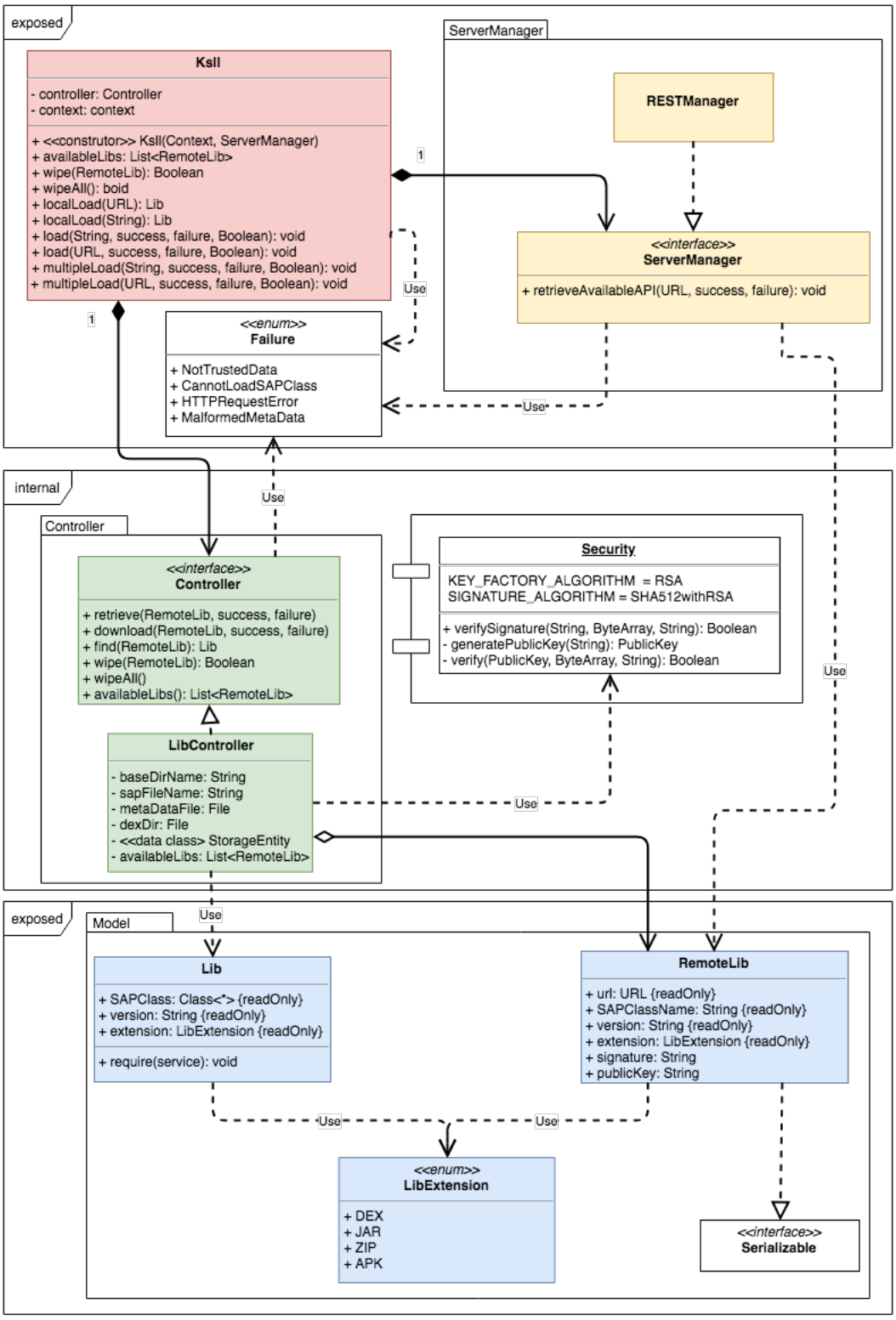



Figura 4.3: Diagramma delle classi

## 4.2 Modello

Il modello costituisce la rappresentazione logica dell'entità libreria, soggetto primario di questo progetto. Essendo il risultato finale dell'elaborazione che viene consegnato all'utilizzatore, è pensato e realizzato affinché contenga poche informazioni in favore della semplicità d'uso. Per rappresentazione logica s'intende che le classi del modello effettuano *information hiding* riguardo la rappresentazione fisica: la libreria non è altro che uno o più file, ma il modello, come si può vedere dalla figura 4.4, non contiene alcun campo file o tipi simili. Questo per due ragioni:

**Trasparenza** L'utente utilizza l'entità logica in modo trasparente: i file di una libreria, i percorsi nel disco fisso e i metodi di lettura o scrittura non sono di suo interesse, dunque nascosti.

**Integrità** Non esponendo alcuna informazione riguardo la rappresentazione fisica, come i file, l'utente non è in grado di cancellare o alterare direttamente la libreria.

Quest'ultimo punto è di notevole rilevanza. Il modulo espone precisi metodi per eliminare le librerie, con una propria logica interna. Se le classi di modello esponessero file o percorsi legati ad una libreria, l'utente potrebbe aggirare i metodi predisposti ed effettuare operazioni di eliminazione o scrittura non voluti.

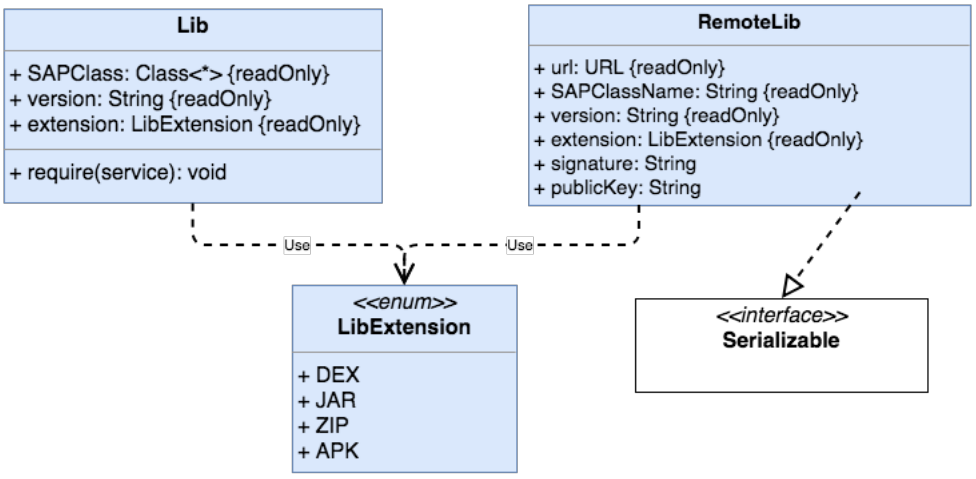



Figura 4.4: Classi del modello

Una libreria è rappresentata da due classi distinte: `Lib e RemoteLib`. Questo è dovuto al fatto che, nella logica di business, vi sono delle transizioni per cui esiste il concetto di libreria remota (`RemoteLib`) prima di ottenere la corrispettiva libreria locale (`Lib`). Le funzionalità rispettive sono:

**Lib** Rappresenta la libreria locale, scaricata e fisicamente presente nel disco fisso del dispositivo. L'utente dispone di istanze di questa classe per trarre le funzionalità della libreria: attraverso il metodo `require` ottiene un'istanza di *SAPClass* e una lista dei metodi associati invocabili tramite meccanismo di reflection. I campi `version` e `extension` sono di utilità informativa.

**RemoteLib** Rappresenta la libreria remota, residente in un servitore e non ancora pronta all'utilizzo. Contiene esattamente un campo per ogni metadato esposto dal servitore. Viene usato dall'utilizzatore esclusivamente per estendere le funzionalità del modulo: nell'aggiunta di un nuovo `ServerManager`, questa classe è indispensabile per definire il dialogo con il servitore. Viene inoltre utilizzata internamente per tenere traccia di quali librerie si dispongono nel disco fisso; per questo motivo estende l'interfaccia `Serializable`. Questa interfaccia è ben nota in ambiente JVM: è un marcatore che permette al sistema di interpretare l'oggetto come serializzabile, rendendolo effettivamente scrivibile in modo persistente.

L'enum `LibExtension` è semplicemente una struttura dati per mantenere tutte le estensioni accettate come libreria dal nostro sistema. Dispone degli adeguati metodi per la conversione da e verso una stringa.

### 4.2.1 Classe di dati Kotlin

Il linguaggio Kotlin rende la scrittura del modello estremamente semplice e coincisa. Fornisce la parola riservata `data` la quale rende una normale classe una classe di modello. Per esempio con la seguente istruzione

```
1 data class RemoteLib(val url: URL,
2                       val SAPClassName: String,
3                       val version: String,
4                       val extension: LibExtension,
5                       val signature: String,
6                       val publicKey: String): Serializable
```

si ottiene l'equivalente in Java di

```
1 public class RemoteLib implement Serializable{
2    private URL url;
3    private String SAPClassName;
4    private LibExtension extension;
5    private String signature;
6    private String publicKey;
```

```
7
8     public RemoteLib(URL url, String SAPClassName, LibExtension
          extension, String signature, String publicKey){
9        this.url = url;
10       this.SAPClassName = SAPClassName;
11       this. extension = extension;
12       this.signature = signature;
13       this.publickKey = publicKey;
14    }
15
16    public URL getUrl(){
17       return url;
18    }
19
20    public String getSAPClassName(){
21       return SAPClassName;
22    }
23
24    public LibExtension(){
25       return extension;
26    }
27
28    public String getSignature(){
29       return signature;
30    }
31
32    public String getPublicKey(){
33       return publicKey;
34    }
35 }
```

Come si può notare la quantità di codice necessario in Java è altamente svantaggiosa rispetto a Kotlin. In un progetto di media o grande dimensione fa la differenza. Inoltre la `data class` ricorre all'utilizzo della parola riservata `val` per specificare attributi di sola lettura. Questa particolarità riduce notevolmente la complessità e la possibilità di errore nell'utilizzo del modello.

### 4.2.2 Funzioni in linea

Kotlin fa largo uso di funzioni di ordine superiore (*Higher-Order Functions*), ovvero quelle funzioni che ne prendono altre come parametri oppure ne ritornano una. Questa è una grande flessibilità e una caratteristica chiave del linguaggio, ma contemporaneamente contribuisce all'overhead delle prestazioni. Ogni funzione passata come parametro o come tipo di ritorno viene dichiarata dal compilatore, il quale implicitamente crea un oggetto funzione e la corrispettiva chiamata [17]. Tutto questo può avere un costo notevole,

soprattutto in blocchi critici come può essere il thread di UI di Android. La soluzione del linguaggio a questo problema è l'introduzione della parola riservata `inline`, la quale forza una funzione ad essere espansa nel codice tramite il meccanismo delle lambda (4.3.1). In questo modo la funzione parametro non ha il costo di dichiarazione e chiamata, con la controparte dell'aumento delle righe di codice (espansione ad ogni funzione `inline`). Dal momento in cui la natura di questo progetto è di essere un modulo, un componente riutilizzabile, non possiamo sapere a priori come verrà utilizzata la classe `Lib`, in particolare il metodo `require` (utile al caricamento delle funzionalità della libreria). Questa situazione ha imposto l'utilizzo di `inline` per questo metodo: l'implementazione è di poche righe, dunque il numero di codice totale replicato è contenibile, mentre l'overhead è arginato dall'espansione della funzione parametro.

## 4.3 Controllore

Il controllore rappresenta la logica dell'applicativo volta alla gestione e al mantenimento delle librerie e delle informazioni di supporto. Funge da ponte tra il modello e il punto di accesso del modulo: la struttura è tale per prendere in ingresso delle `RemoteLib`, generate da un `ServerManager`, elaborarle e restituire le corrispondenti `Lib`. Utilizza una classe di supporto `Security` per risolvere le problematiche legate alla firma digitale.

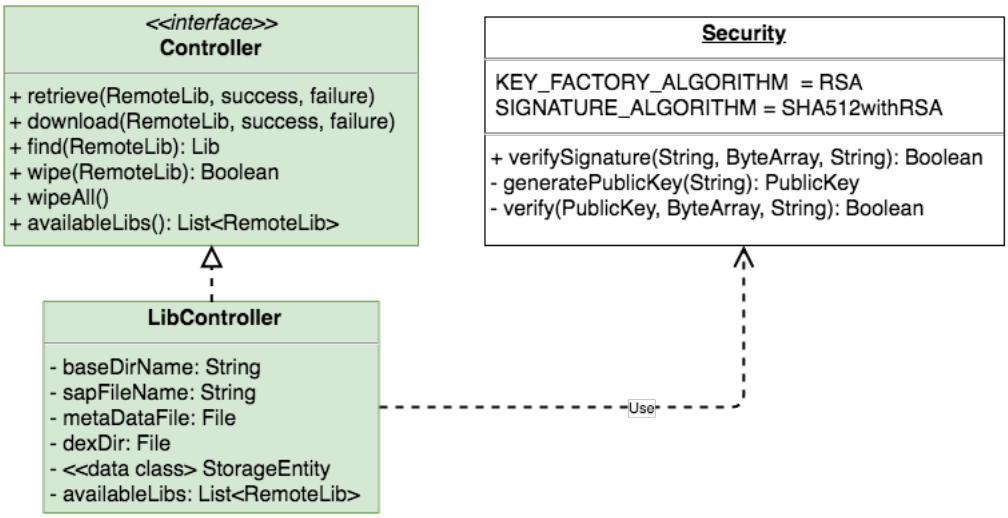



Figura 4.5: Classi del controllore

`Controller` è un'interfaccia che definisce nel dettaglio le operazioni necessarie affinché l'architettura prenda vita. Un generico controllore deve sapere scaricare una libreria remota (`download`), ricercarla localmente nello spazio di archiviazione (`find`) ed eliminarla (`wipe`). Il metodo di recupero

(`retrieve`) combina i primi due: ricerca localmente e se necessario effettua lo scaricamento. Inoltre espone le funzionalità di lettura di tutte le librerie disponibili (`availableLibs`) e di cancellazione totale (`wipeAll`). Essendo un'interfaccia, per sua natura, prescinde dal come queste operazioni vengano effettuate. Per esempio il metodo `download` non fornisce alcuna specifica sul come scaricare e dove salvare le informazioni ottenute.

#### Struttura di archiviazione

Il modulo ha una sua implementazione a riguardo nominata `LibController`. Questa si basa su un sistema di archiviazione di file e direttori, semplice e versatile. L'implementazione usa una `data class StorageEntity` che rappresenta fisicamente una libreria (in riferimento al paragrafo 4.2.1). Il modulo supporta l'utilizzo e il reperimento di molteplici librerie, oltre alla gestione degli aggiornamenti. Per implementare queste funzionalità si è pensato di organizzare il sistema di archiviazione tramite le seguenti direttive:

- Direttorio padre nominato *ksll* per non creare conflitto nello spazio condiviso con l'applicazione in cui il modulo viene utilizzato
- Un file, figlio immediato del direttorio padre, contenente la serializzazione di una lista di `RemoteLib`. Serve per ricordarsi di tutti i metadati di ciascuna libreria presente nel sistema di archiviazione locale.
- Un direttorio per ogni libreria. Questo deve avere un percorso unico per prevenire problemi di collisione. Una libreria è reperibile da un web URL indicato nei metadati del servitore (salvati come descritto nel punto precedente). Si è deciso di utilizzarlo come base da cui creare il percorso, essendo di sua natura già univoco a livello globale.
- All'interno di ogni direttorio di libreria, un file per il contenuto effettivo nominato usando il formato `versione.estensione` e un altro per memorizzare il nome della classe SAP da utilizzare per quella libreria.

L'insieme di questi tre elementi compongono un'istanza della classe `StorageEntity`. In figura 4.6 un'istantanea del sistema di archiviazione di un'applicazione d'esempio contenente due librerie. Da notare che l'URL è completo: protocollo, dominio, percorso relativo e porta, così da supportare molteplici librerie provenienti dallo stesso dominio.

### 4.3.1 Kotlin lambda

Il controllore fa ampio utilizzo delle funzioni lambda, dette anche letterali, caratteristica chiave del linguaggio Kotlin. In figura 4.5 si può notare che alcuni metodi hanno dei parametri di nome *success* e *failure*. Questi non sono tipi, bensì indicano delle funzioni di ritorno che vengono invocate

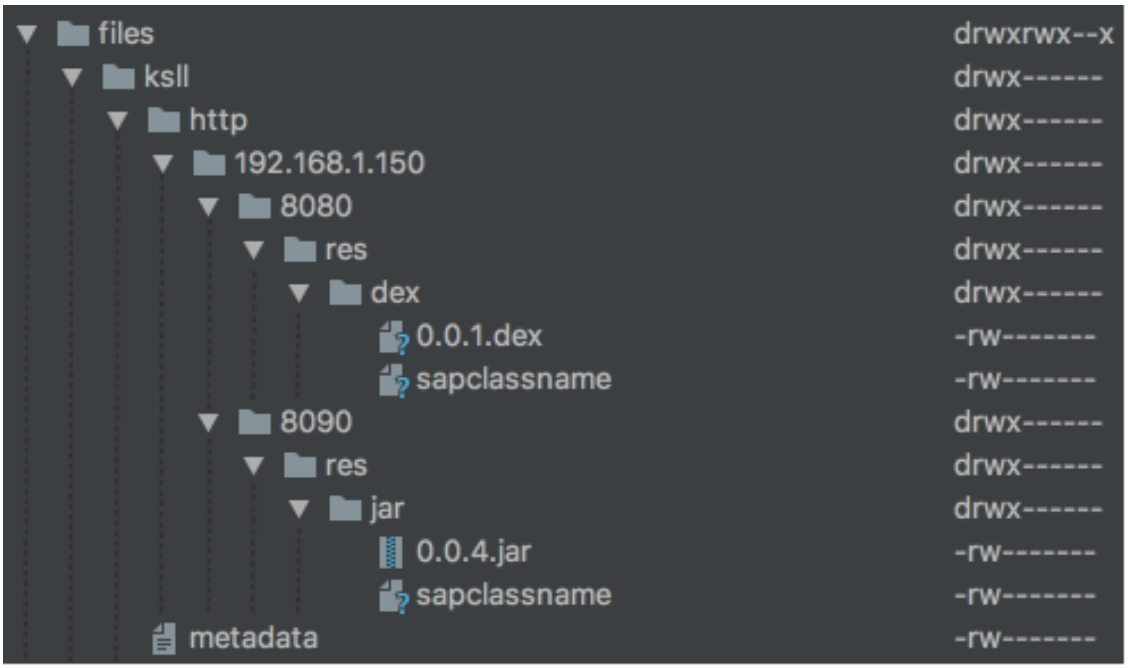



Figura 4.6: Struttura di archiviazione

all'interno del metodo. Queste funzioni a loro volta possono avere o meno parametri. Dunque quando il metodo viene invocato è necessario specificare come parametri due funzioni. In questo caso è possibile utilizzare le funzioni lambda che consistono in una espressione implementata direttamente nel punto di chiamata, aggirando la dichiarazione e i costi associati. Ad esempio vediamo la firma del metodo `retrieve`

```
1 fun retrieve(remoteLib: RemoteLib, success: (lib: Lib) -> Unit =
      {}, failure: (cause: Failure) -> Unit = {})
```

e una sua invocazione

```
1 ctrl.retrieve(remoteLib, {lib ->
2                   if (shouldUpdate && lib.version !=
                        remoteLib.version)
3                       [...]
4               }, failure)
```

In questo esempio la funzione *success* (secondo parametro) è passata come espressione lambda, mentre *failure* tramite il metodo classico usando una variabile contente a priori la funzione.

Da notare, infine, che i parametri della funzione di ritorno dichiarati nella firma sono quelli esposti all'uso, come il parametro `Lib`. Questo concetto è dettagliato nel paragrafo 4.6, fondamentale per il corretto utilizzo del modulo.

## 4.4 Sicurezza

Lo logica riguardante la sicurezza è racchiusa in unica classe marcata dalla parola riservata `object`. Kotlin permette così di avere un *Singleton* pattern integrato nel linguaggio: la classe `Security` (fig. 4.5) avrà una sola ed unica istanza accessibile attraverso tutto il modulo. Le funzionalità di questa classe sono volte all'implementazione di verifica della firma digitale. L'unico metodo esposto è `verifySignature` a cui è sufficiente passare una chiave pubblica, i dati (documento) da verificare e la firma digitale.

In riferimento al paragrafo 3.6.3, il progetto non si è spinto fino all'implementazione della PKI in Android, dovuto al principale fatto che sarebbe stato necessario l'utilizzo di un vero certificato HTTPS e un server che lo mantenesse. Invece, la chiave pubblica attualmente viene esposta dal servitore e mantenuta come informazione chiara in `RemoteLib`, soluzione adeguata per un ambiente di sviluppo.

La chiave pubblica è di per sé una sequenza di byte. Per essere scritta, trasmessa e letta ha bisogno di una rappresentazione testuale. Si è scelto di usare la codifica `Base64` permettendo di convertire i dati binari in stringhe di testo ASCII. Così deve essere passata la chiave pubblica al metodo `verifySignature`, mentre i dati sono passati come sequenza di byte. Per ottenerla è sufficiente l'utilizzo di un flusso di lettura dal file di libreria scaricato. La verifica della firma digitale avviene nel metodo `download` del controllore, se fallisce viene invocata la funzione di fallimento. Il servitore dovrebbe seguire lo stesso standard di rappresentazione per evitare incongruenze lato cliente. L'utilizzo di una codifica differente può causare il fallimento della verifica di autenticità, sollevando un errore legato alla sicurezza pur essendo di altra natura.

Sempre in figura 4.5 si può notare che la classe `Security` presenta due costanti dal valore *RSA* e *SHA512withRSA*. La prima indica l'algoritmo da usare per interpretare la chiave pubblica presa in ingresso, mentre la seconda è l'algoritmo di firma digitale. Questo conferma l'implementazione dei concetti discussi al paragrafo 3.6.

## 4.5 Gestore del servitore

L'interfaccia `ServerManager` definisce il contratto non scritto tra cliente e servitore, ovvero in che modo, dato un web URL, sia possibile ottenere una o più librerie remote (`RemoteLib`). Il contratto è detto non scritto perché deve essere l'implementazione a determinare la logica interna di definizione del contratto stesso. Il gestore del servitore risulta estremamente semplice:

```
1  /**
2   * An implementation of this interface should work for a specific
3   * type of web server that provides an API about libraries that
4   * can be downloaded. The implementation is based on a
5   * non written contract, such as the structure and the content of
        the response
6   *
7   * @author Matteo Pellegrino matteo.pelle.pellegrino@gmail.com
8   */
9  interface ServerManager {
10
11     /**
12      * Async http request the [url], elaborate the response
13      * and invoke [success] or [failure]
14      *
15      * @param url The URL at which a web server is ready to respond
            (with a KSLL meta inf)
16      * @param success Callback function on successful response
17      * @param failure Callback function on error
18      */
19     fun retrieveAvailableAPI(url: URL, success: (remoteLib:
           List<RemoteLib>) -> Unit, failure: (cause: Failure) -> Unit)
20 }
```

L'unico metodo, che rispecchia il comportamento dell'API, definisce l'ingresso (`URL`) e il risultato, che può avere successo (`List<RemoteLib>`) o fallire (`Failure`).

Il modulo dipende esclusivamente dall'interfaccia, risultando altamente riusabile. Questo è un semplice, ma elegante uso della *Dependency Inversion Principle*, secondo il quale ogni dipendenza dovrebbe mirare all'interfaccia piuttosto che alla sua concreta implementazione. Il principio limita la propagazione degli errori e permette l'aggiunta di funzionalità in modo flessibile, senza la necessità di modifiche. L'utilizzatore potrà infatti implementare il contratto con un altro tipo di servitore, non fornito dal modulo, e passarlo al punto di accesso, estendendo di fatto la funzionalità di KSLL senza modificare il suo codice interno.

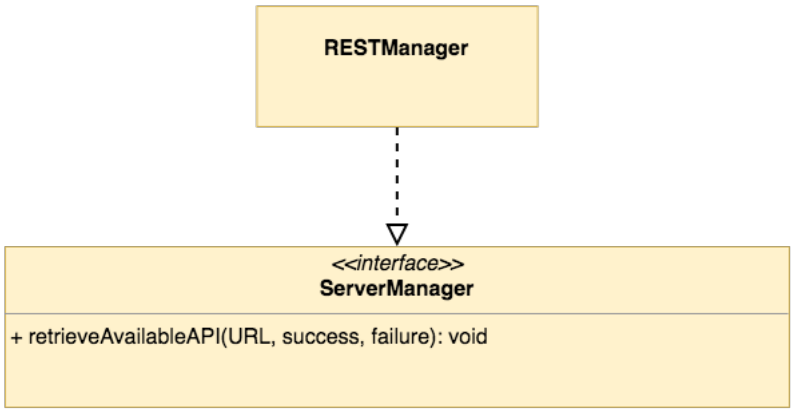



Figura 4.7: Classi del gestore del servitore

### 4.5.1 Supporto a servizi web RESTful

L'implementazione di `ServerManager` predisposta nel modulo è a supporto dei servitori *RESTful* e del formato di interscambio dati *JSON*. Questa scelta è dovuta all'ampio utilizzo dell'architettura *Representational State Transfer (REST)* riscontrato nei sistemi distribuiti. Questa tecnologia ha da qualche anno soppiantato i sistemi basati su *SOAP o WSDL*, in quanto notevolmente più semplice e flessibile [18]. I servizi web RESTful hanno reso l'accoppiamento cliente-servitore più debole, permettendo di servire clienti di differenti tipologie, linguaggio e architettura. Non è d'interesse in questo studio approfondire l'argomento, si vuole solo sottolineare che il supporto ai servitori RESTful-JSON, piuttosto che ad altri, è giustificato dalla notorietà della tecnologia e dal semplice interfacciamento.

La classe `RESTManager` implementa il comportamento: esegue una richiesta HTTP puntando l'URL specificato come parametro nel metodo `retrieveAvailableAPI`, legge la risposta in formato JSON e invoca la funzione di ritorno `success` o, in caso di errore, `failure`. La funzione `success` prevede come parametro una lista di `RemoteLib`, creata in funzione delle informazioni ricevute dal servitore. Di seguito, un esempio di metadati in formato JSON.

```
[
   {
      "url": "http://localhost:8080/res/dex",
      "sapclassName": "it.matteopellegrino.ksla.KslaSAP",
      "version": "0.0.1",
      "extension": "dex",
      "signature": "DdglDaYEz1ApDmO...df/ydQ1SM=",
      "publicKey":"MIGfMA0GCSq...2+AgtGQIDAQAB"
   }
]
```

I metadati sono esattamente i campi che costituiscono la classe di modello `RemoteLib`. Un servitore deve offrire almeno queste informazioni per il corretto funzionamento. Da notare che l'esempio riporta un JSON Array e non un semplice JSON object; `RESTManager` supporta sia una risposta di un singolo oggetto (libreria) sia un array di oggetti. Il secondo caso implica che il servitore sta offrendo più librerie differenti. Non è necessario specificare quale delle due opzioni utilizzare: l'implementazione tenta di leggere un array, se fallisce utilizza la logica per un singolo oggetto. Se in entrambi i casi vi siano problemi di lettura viene invocata la funzione di ritorno `failure`.

Il valore corrispondente alla chiave *url* indica l'indirizzo della risorsa fisica (libreria), usato per scaricare i dati binari e memorizzarli sul sistema di archiviazione interno. Questo implica che l'*endpoint* per i metadati è diverso da quello per il download della libreria.

## 4.6 Punto di accesso e utilizzo del modulo

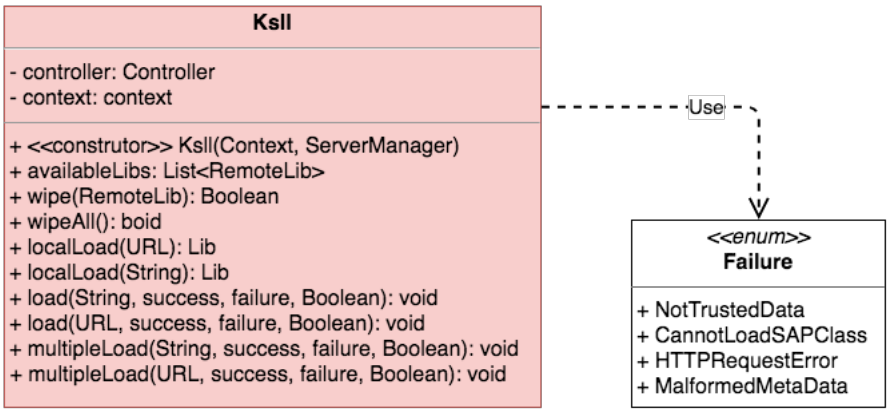



Figura 4.8: Classi del *top level* package

La classe `Ksll` è il punto di accesso per l'utilizzo del modulo. Il suo costruttore richiede un parametro di tipo `Context` e uno di tipo `ServerManager`. Il primo è un'interfaccia alle informazioni globali riguardo l'ambiente dell'applicazione. L'implementazione è fornita dal sistema ed è disponibile in qualsiasi componente Android. Tra le varie funzioni consente l'accesso a risorse specifiche dell'applicazione, viene usato principalmente nella logica di archiviazione (controllore). Il secondo è l'interfaccia discussa nel paragrafo precedente.

Ottenuta un'istanza di `Ksll` è possibile invocare diverse funzionalità, corrispondenti ai metodi implementati mostrati in figura 4.8.

`availableLibs` Ritorna una lista di librerie (`RemoteLib`) attualmente disponibili in locale e pronte all'uso.

`wipe(RemoteLib)` Elimina una specifica libreria rimuovendo ogni file persistente ad essa associata.

`wipeAll()` Invoca il precedente metodo per ogni libreria disponibile al momento.

`localLoad(URL)` Cerca localmente la libreria identificata da un `URL` passato come parametro. Ritorna un'istanza di `Lib` se è stata trovata, `null` altrimenti. L'`URL` specificato deve essere quello che punta alla risorsa fisica della libreria, non quello generico per i metadati; corrisponde al campo `RemoteLib.url`.

`localLoad(String)` Come la precedente, ma è possibile passare come parametro uno stringa anziché un'istanza di `URL`.

`load(URL, success, failure, Boolean)` Attua ogni procedura necessaria per contattare il servitore all'indirizzo `URL`, legge i metadati e scarica la libreria esposta. Se sono esposte più librerie solo la prima viene considerata. Il secondo e terzo parametro sono funzioni di ritorno ad uso specifico del programmatore; vengono invocate quando è pronto un risultato. Ogni operazione è asincrona. Il quarto parametro (opzionale) abilita il controllo di versione, permettendo l'aggiornamento automatico tramite sovrascrizione. Il valore predefinito è `true`, se non si vuole aggiornare in automatico è necessario passare il valore `false`. N.B.: `URL` indica l'indirizzo del servitore in cui sono esposti i metadati

`load(String, success, failure, Boolean)` Come la precedente, potendo specificare l'indirizzo come stringa.

`multipleLoad(URL, success, failure, Boolean)` Esegue la stessa logica di `load(URL, success, failure, Boolean)`, ma per ogni libreria esposta dal servitore, non solamente la prima. In questo caso le funzioni di ritorno di successo e fallimento sono invocate molteplici volte indipendentemente per ogni libreria. Invece il quarto parametro è comune, il controllo di versione viene effettuato per ogni libreria. Questo metodo può essere invocato anche non conoscendo a priori se il servitore espone una o più librerie.

`multipleLoad(String, success, failure, Boolean)` Come la precedente, potendo specificare l'indirizzo come stringa.

Di seguito un breve esempio di utilizzo che richiede una libreria e stampa le funzionalità a disposizione:

```
1 val ksll = Ksll(baseContext, RESTManager())
2
3 ksll.load("http://192.168.1.150:8080", { remoteLib ->
4    remoteLib.require{ obj, methods ->
5       methods.forEach{ println(it) }
6    }
7 }, { error ->
8    println("Error: $error")
9 })
```

### 4.6.1 Funzioni di estensione

In riferimento all'esempio precedente, il metodo `require`, appartenente alla classe `Lib`, è in realtà dichiarato in un file differente, precisamente in quello del punto di accesso (classe `Ksll`), nel seguente modo:

```
1 inline fun Lib.require(service: (obj: Any, methods: List<Method>)
      -> Unit) {
2     val inst = SAPClass.newInstance()
3     val mthds = SAPClass.methods.filter { it.declaringClass !=
          Any::class.java }
4     service(inst, mthds)
5 }
```

Questo introduce le *funzioni di estensione*, caratteristica non disponibile in Java. Kotlin offre la possibilità di estendere una classe con nuove funzionalità senza utilizzare l'ereditarietà o alcun pattern di progettazione. Questo viene fatto tramite dichiarazioni speciali chiamate *estensioni*. Kotlin supporta sia le funzioni che le proprietà di estensione [19]. Queste non modificano effettivamente le classi che estendono: non inseriscono nuovi membri in una classe, ma si limitano a creare nuove funzioni invocabili dalle istanze della classe. Le funzioni di estensione sono esplicitate staticamente, caratteristica da considerare in particolari casi di ereditarietà.

Nel progetto si è voluto estendere il metodo `require` delle classi `RemoteLib` e Lib principalmente per organizzazione del codice. Essendo queste classi unicamente di modello, ho personalmente preferito dichiarare il metodo, che introduce una certa logica di business, nel file del punto di accesso. In questo modo quel file è l'unico a contenere ogni operazione (classe e funzioni) invocabile dall'utilizzatore. Questa organizzazione è preferibile per facilitare future modifiche: le classi di modello sono da ritenere abbastanza rigide, mentre la logica di accesso ed utilizzo può variare più frequentemente.

A seguito di questa sezione implementativa è risultato naturale aggiungere alcune funzioni di estensioni che replicassero i metodi della classe `Ksll`, rendendoli accessibili a tipi di dato esterni. I metodi `localLoad, load, multipleLoad` sono di fatto invocabili anche da oggetti di tipo `URL` e `String`.

```
val ksll = Ksll(baseContext, RESTManager())

"http://192.168.150.1:8080".load(ksll, success, failure);
URL("http://192.168.1.150:8080").load(ksll, success, failure);

"http://192.168.1.150:8080".multipleLoad(ksll, success, failure);
URL("http://192.168.1.150:8080").multipleLoad(ksll, success,
    failure);

"http://192.168.1.150:8080".localLoad(ksll, success, failure);
URL("http://192.168.1.150:8080").localLoad(ksll, success, failure);
```

### 4.6.2 Eccezioni

Come si è notato da questi piccoli esempi, KSLL non introduce una grande complessità di utilizzo: i metodi esposti sono pochi, essenziali e dall'utilizzo immediato. In un contesto del genere, il lancio di eccezioni, estese da classi di `Exception`, risulterebbe oneroso per l'utilizzatore. Ogni qual volta ci fosse la necessità di richiede l'utilizzo di una libreria sarebbe necessario l'utilizzo di un blocco di controllo `try catch`. Inoltre, la maggior parte dei metodi sono asincroni e richiedono un'attesa dipendente dalla connessione internet, rendendo la maggior parte delle volte il blocco di controllo futile o problematico all'utilizzo. Invece, si è preferito utilizzare una funzione di ritorno, nominata `failure`, presente in ogni metodo esposto. In caso di errore viene invocata questa funzione, invece di sollevare un'eccezione nativa. L'implementazione della funzione `failure` deve essere fornita dall'utilizzatore, a sua discrezione. La funzione prevede un parametro denominato *cause* (causa) di tipo `Failure`.

```
/**
 * Type of errors passed to failure callback as parameter.
 * Exceptions are wrapped into these types
 *
 * @author Matteo Pellegrino matteo.pelle.pellegrino@gmail.com
 */
enum class Failure{
    NotTrustedData,
    HTTPRequestError,
    MalformedMetaData
}
```

`Failure` è un *enum* rappresentante ogni tipo di errore che può verificarsi nella logica interna del modulo. Il parametro *cause* sarà, al momento dell'in-

vocazione, del tipo assegnato, così che l'utilizzatore può facilmente basare la sua logica di business di conseguenza. Per esempio:

```
ksll.load("http://192.168.1.150:8080", ... , {error ->
          val msg = when(error){
              Failure.NotTrustedData -> "Signature verification
                  failed. Library not trusted."
              Failure.HTTPRequestError -> "Connection problem.
                  Cannot retrieve library."
              else -> {
                  "Unexpected error. Cannot load library."
              }
          }

          Toast.makeText(this, msg, Toast.LENGTH_LONG).show()
      }
})
```

**NotTrustedData** Il controllo di firma digitale è risultato negativo. I dati sono stati alterati o provengono da una fonte non fidata. In tal caso la libreria viene scartata e non è utilizzabile.

**HTTPRequestError** Errore generico di connessione o risposta dal servitore inutilizzabile. Può verificarsi ad ogni richiesta via rete. Non compromette necessariamente l'utilizzo di una libreria, ma può renderla momentaneamente non disponibile (connessione internet assente).

**MalformedMetaData** I metadati forniti dal servitore sono errati o non sufficienti ai fini del corretto funzionamento del modulo. La verifica di questo errore implica l'errata implementazione del servitore o del `ServerManager`.

## 4.7 Dipendenze

L'unica dipendenza di terzi è una libreria denominata Fuel (`https://github.com/kittinunf/Fuel`) utilizzata per effettuare richieste HTTP, soprattutto nel controllore e nella classe `Ksll`. Si è rivelata di grande utilità per la semplicità d'uso e la minimizzazione del codice necessario. La sua licenza di utilizzo è largamente aperta [20].

## 4.8 Applicazione d'esempio

L'intero codice del progetto è reperibile all'indirizzo `https://github.com/arabello/KSLL`. Oltre al modulo è disponibile un'applicazione Android d'esempio all'utilizzo di KSLL. In figura 4.9 sono riportati due screenshots delle *Activity* disponibili. La prima è un elenco delle librerie attualmente disponibili localmente con le informazioni dei metadati; un pulsante di aggiunta permette di inserire un URL e reperire una nuova libreria. Cliccando su una di esse si raggiunge la pagina di dettagli, in cui vengono elencati le funzionalità (metodi) di cui la libreria dispone.

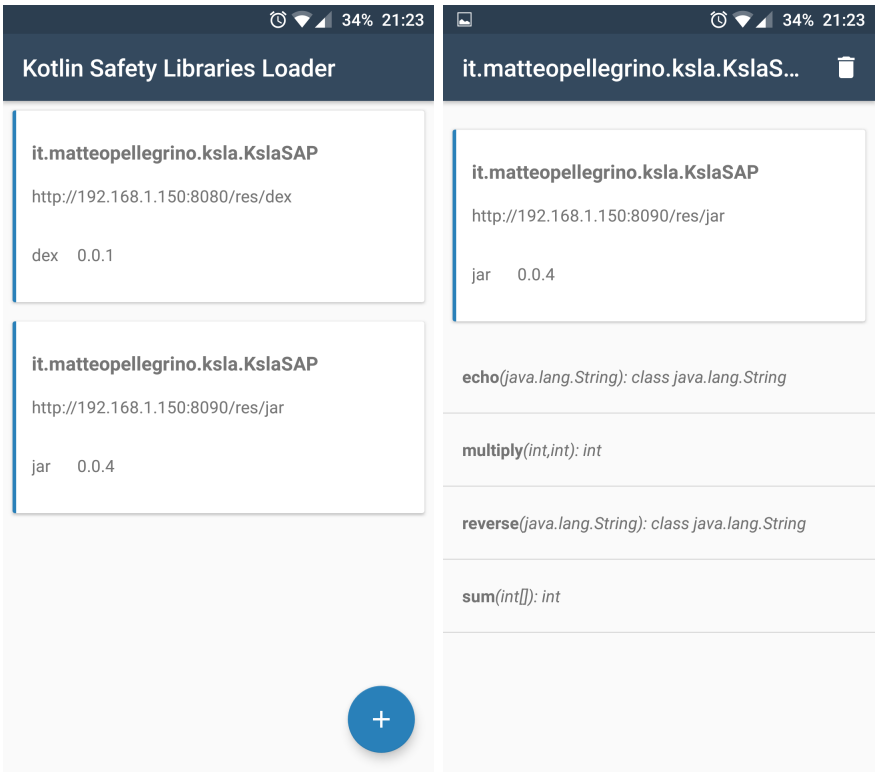



Figura 4.9: Screenshot dell'applicazione di esempio

Questo esempio può essere di ispirazione per l'utilizzo del modulo o persino un punto di partenza su cui costruire la propria applicazione Android. Al link specificato precedentemente è disponibile un file (`README.md`), che riassume i concetti esposti in questo paragrafo. La presenza di questo file, nella directory principale del progetto, è una convenzione utilizzata dalla community di sviluppatori, alla quale con piacere prendiamo parte.

# Capitolo 5

# Conclusioni

## 5.1 Analisi dei risultati

L'obiettivo iniziale del progetto, in sintesi, era rendere disponibile all'utente il codice per l'elaborazione di dati sensibili, senza che questi escano dal contesto privato del suo dispositivo. Il risultato di questa tesi è uno strumento capace di risolvere questo tipo di problematica, in differenti contesti, attraverso un componente software riutilizzabile. Affinché lo scopo prefissato raggiunga pienamente l'utente, il codice scaricato deve essere autocontenuto e non deve disperdere a sua volta i dati sensibili. Questo passo è a completa discrezione dell'utilizzatore, in quanto non è possibile sapere a priori in quale contesto il modulo verrà utilizzato.

Per esempio, come accennato nell'introduzione, uno dei contesti di utilizzo risiede in un'applicazione per la mobilità urbana. Questa prevede una serie di servitori, organizzati secondo la localizzazione dell'utente, che dispongono di diverse implementazioni della stessa libreria. Tramite il modulo KSLL è dunque possibile scaricare il codice di una determinata implementazione in funzione della localizzazione. Il codice deve essere fidato; il modulo si occuperà di verificare l'autenticità della fonte e l'integrità dei dati.

## 5.2 Sviluppi futuri

Gli sviluppi futuri immediati del progetto dovrebbero riguardare prima di tutto l'integrazione della Android PKI. L'attuale implementazione di verifica della firma digitale è ottima in un contesto di sviluppo, in quanto semplice da impostare e collaudare, ma non è adeguata in un ambiente di produzione. Sarebbe opportuno utilizzare un canale sicuro per il dialogo con i servitori e verificare la loro autenticità da un certificato HTTPS, tramite le API della Android PKI. Secondariamente, sarebbe necessario disporre la possibilità di dialogo con diversi tipi di servitori che trattano i dati in formato diverso, come ad esempio XML, predisponendo le implementazioni adeguate

di `ServerManager`. Sebbene un utilizzatore possa fornire la sua implementazione, il modulo dovrebbe fornirne di proprie per ampliare il supporto ai servitori più comuni.

# Elenco delle figure

# Bibliografia


[1] Statista. (2018). Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017, indirizzo: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[2] A. Sinhal. (2017). Closer Look At Android Runtime: DVM vs ART, indirizzo: https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924.

[3] M. Vinther. (2017). Why you should totally switch to Kotlin, indirizzo: https://medium.com/@magnus.chatt/why-you-should-totally-switch-to-kotlin-c7bbde9e10d5.

[4] (2018). JSR223 API example project, indirizzo: https://github.com/JetBrains/kotlin/tree/master/libraries/examples/kotlin-jsr223-local-example.

[5] Oracle. (). The Java Scripting API, indirizzo: https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html.

[6] D. Jemerov. (2018), indirizzo: https://discuss.kotlinlang.org/t/how-to-use-kotlin-compiler-on-android/6513/4.

[7] Google. (2018). Data and file storage overview, indirizzo: https://developer.android.com/guide/topics/data/data-storage#filesInternal.

[8] NSA. (2016), indirizzo: https://cryptome.org/2016/01/CNSA-Suite-and-Quantum-Computing-FAQ.pdf.

[9] IETF. (2008), indirizzo: https://www.ietf.org/rfc/rfc5280.txt.

[10] Microsoft. (), indirizzo: https://msdn.microsoft.com/en-us/library/windows/desktop/bb540819(v=vs.85).aspx.

[11] Symantec. (2017), indirizzo: https://knowledge.symantec.com/support/ssl-certificates-support/index?page=content&actp=CROSSLINK&id=SO16297.

[12] (2013), indirizzo: https://security.stackexchange.com/questions/5096/rsa-vs-dsa-for-ssh-authentication-keys.

[13] V. N. Database. (2009), indirizzo: `https://www.kb.cert.org/vuls/id/836068`.

[14] Google. (2018), indirizzo: `https://developer.android.com/reference/java/security/Signature`.

[15] ——, (2018), indirizzo: `https://developer.android.com/training/articles/security-ssl`.

[16] ——, (2018). Distribution Dashboard, indirizzo: `https://developer.android.com/about/dashboards/`.

[17] JetBrains. (2018), indirizzo: `https://kotlinlang.org/docs/reference/inline-functions.html`.

[18] A. Rodriguez. (2015), indirizzo: `https://www.ibm.com/developerworks/library/ws-restful/index.html`.

[19] JetBrains. (2018), indirizzo: `https://kotlinlang.org/docs/reference/extensions.html`.

[20] K. Vantasin. (2018), indirizzo: `https://github.com/kittinunf/Fuel/blob/master/LICENSE.md`.