# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Kotlin

## 1.1 Storia

Kotlin é un linguaggio di programmazione open source[1], basato sulla JVM (Java Virtual Machine), con la caratteristica di essere orientato agli oggetti e staticamente tipizzato.
Lo sviluppo di Kotlin é iniziato nel 2010 dall'azienda JetBrains, conosciuta nel modo dello sviluppo software Java per la realizzazione di diversi IDE ( Integrated development environment ), tra cui: Intellij IDEA, sul quale si basa l'attuale IDE di Google dedicato alla programmazione Android, chiamato: Android Studio.
Il team di JetBrains, scelse di iniziare a realizzare un nuovo linguaggio di programmazione, per risolvere problematiche riscontrate durante la realizzazione dei loro IDE, in particolare Dmitry Jemerov, un programmatore di JetBrains e sviluppatore di Kotlin, riveló durante un'intervista [2] che non esisteva ancora nessun linguaggio con le potenzialitá, facilitá d'uso richieste dal team di JetBrains, ad eccezzione del linguaggio Scala che offriva grossi vantaggi nello sviluppo, ma aveva un tempo di compilazione molto lento, Jet-

[1] https://github.com/JetBrains/kotlin

[2] https://www.infoworld.com/article/2622405/java/jetbrains-readies-jvm-based-language.html

Brains scelse quindi di realizzare il linguaggio Kotlin basandosi sulla JVM, in seguito nel 2012 il progetto Kotlin venne resto Open Source sotto licenza Apache 2 license.
Java fin dalle sue prime versioni, é sempre stato uno dei linguaggi piú usati e conosciuti ma presenta diverse imperfezioni e problemi che spinse il team di JetBrains a iniziare lo sviluppo di un suo linguaggio, con una sintassi semplice che prendesse in considerazione alcuni spunti e idee introdotte da linguaggi come CSharp, Scala, Groovy, ECMAScript, Go, Python, ma continuasse a basarsi sulla JVM.
L'idea di utilizzare pattern e idee di linguaggi preesistenti permise agli sviluppatori di introdurre la facilitá sintattica, potenzialitá e caratteristiche testate a lungo da altri linguaggi, senza dover apportare grosse innovazioni, rendendo quindi il linguaggio leggibile e comprensibile anche da chi non lo conoscesse.
Lo sviluppo di Kotlin si basava principalmente sul miglioraramento di Java, ma dato che anche Android utilizzava la JVM, gli sviluppatori cercarono di adattarlo e renderlo ottimizzato anche per Android.
La comunitá di sviluppatori che cominció a utilizzare Kotlin crebbe enormemente, tra il 2016 e il 2017, in solo un anno, le linee di codice scritte in Kotlin su progetti presenti su GitHub quadruplicarono passando da 2.4 milioni a 10 milioni [3].

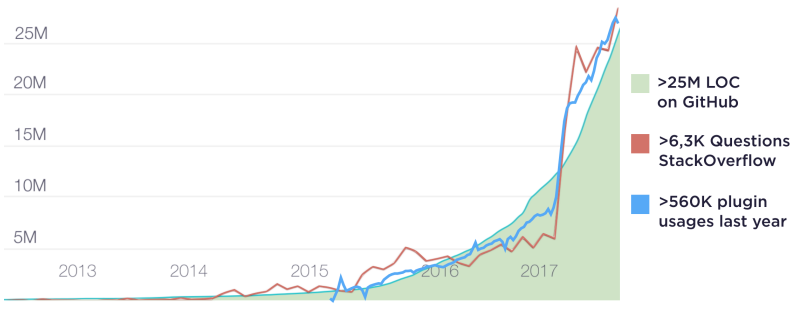


Figura 1.1: Kotlin incremento.

[3] https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1/

Nel 2015 Google prese in considerazione l'utilizzo di Kotlin come plugin per Android Studio, e dopo vari test nel 2017 durante la conferenza Google IO ( 2017 ), arrivó l'annuncio che ufficializzava [4] Kotlin come nuovo linguaggio di programmazione per lo sviluppo di applicazioni Android, senza escludere e rinunciare a Java, su cui si basa l'SDK di Android.
Gli stessi sviluppatori Android, dopo aver testato le potenzialitá di Kotlin ne rimasero molto soddisfatti per la praticitá, la stabilitá e i suoi benefici sintattici e funzionali, Kotlin é infatti un linguaggio molto coinciso, espressivo, strutturato sulla tipizzazione che mette a disposizione costrutti per evitare errori a puntatori nulli.
Il supporto completo di Kotlin su Android venne raggiunto attraverso una buona integrazione con l'IDE Android Studio 3.0 e un plugin Kotlin per le versioni precedenti delle IDE, qualsiasi progetto che utilizzava Java poteva essere parzialmente o completamente convertito in Kotlin.

## 1.2 caratteristiche

Il linguaggio Kotlin é stato sviluppato in ambito aziendale e non accademico, come spesso accade per altri linguaggi, rimanendo in fase beta per 7 anni, periodo in cui gli stessi programmatori che lavoravano presso JetBrains ne testarono le funzionalitá, fino a raggiungere nel 2017 la prima versione stabile: la 1.0.
Kotlin é nato quindi all'interno di un team di sviluppatori che dopo anni di esperienza acquisita con Java e altri linguaggi hanno realizzato un linguaggio mirato a risolvere problemi concreti riscontrati dagli stessi sviluppatori.
Prendendo spunto dalle problematiche di Java e da buone regole introdotte da alcuni linguaggi imperativi e funzionali, Kotlin é stato modellato in modo tale da aggiungere funzionalitá utili sia a livello sintattico che a livello prestazionale, offrendo quindi al programmatore strumenti, caratteristiche e

[4] https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html

implementazioni semplici e utili ma molto potenti.
Un altro aspetto importante su cui il team di JetBrains ha prestato molta attenzione é stata la buona integrazione del suo plugin con Android Studio. Il supporto dato dal plugin é tale da supportare il programmatoreFirebase logo in ogni momento, proponendo la riscrittura di porzioni di codice per renderlo piú coinciso, allertare il programmatore in caso di possibili puntatori nulli, offrire una conversione automatica del codice Java in Kotlin e ove possibile cercherá di allertare lo sviluppatore su possibili errori e problemi di prestazione e sintattici.
Le caratteristiche piú importanti offerte da Kotlin sono l'interoperabilitá con Java, permettendo l'utilizzo di librerie Java e Kotlin simultaneamente, l'introduzione di alcune caratteristiche dei linguaggi di ordine superiore, la tipizzazione statica delle variabili, l'inferenza di tipo e soprattutto il null-safety permettendo di differenziare il tipo nullabile e il tipo non-nullabile, prevenendo quindi errori di null pointer expetions.
Il codice prodotto in Kotlin è inoltre piú compatto, coninciso e meno verboso grazie alle dataclass, il supporto delle lambda function e altri costrutti utili.

### 1.2.1 Interoperabilitá

I linguaggi Kotlin e Java sono fortemente intercompatibili, permettendo quindi a entrambi i linguaggi di coesistere all'interno dello stesso codice e di richimare funzioni e parti di codice in Java da Kotlin e viceversa, poiché entrambi i linguaggi producono Java Bytecode [5].
Prendiamo in considerazione il classico esempio "Hello word" scritto in Kotlin e in Java:

Listing 1.1: Hello.kt in Kotlin

```
fun main(args : Array<String>) {
  println("Hello, world!")
}
```

[5] http://kotlinlang.org/docs/reference/java-interop.html

Listing 1.2: Hello.java in Java

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Il compilatore di Kotlin prendendo come input il file "Hello.kt" produrrá un JAR eseguibile da Java "Hello.jar"

Listing 1.3: Compilatore kotlin

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar //
$ java -jar Hello.jar
$ Hello, world!
```

-includeruntime é un opzione del compilatore per produrre un eseguibile jar, includendo le runtime di Kotlin ( 800Kb)

Listing 1.4: da Java a Kotlin

```
// Calling Java code from Kotlin
class KotlinClass {
    fun kotlinDoSomething() {
        val javaClass = JavaClass()
        javaClass.javaDoSomething()
        println(JavaClass().prop)
    }
}
```

Listing 1.5: da Kotlin a Java

```
// Calling Kotlin code from Java
public class JavaClass {
    public String getProp() { return "Hello"; }
    public void javaDoSomething() {
```

```
        new KotlinClass().kotlinDoSomething();
    }
}
```

### 1.2.2 Performance

I tempi di compilazione e d'eseguizione di un programma scritto in Kotlin sono molto simili a Java poiché entrambi producono bytecode per la JVM. Nei progetti Android, utilizzare la libreria Kotlin non comporta un grande aumento nella dimensione finale dell'APK, Kotlin aggiunge circa 7000 metodi aggiuntivi a runtime che corrispondono ad un aumento di 1MB nell'APK. L'impatto di questa libreria aggiuntiva, anche se aumenta la dimensione dell'APK, porta tanti vantaggi poiché grazie alle nuove caratteristiche introdotte da Kotlin, non sará necessario utilizzare librerie esterne come RxJava, Guava, butterknife che spesso vengono aggiunte aumentando coniderevolmente la dimensione finale dell'APK.
In termini di performance Kotlin pone alcuni miglioramenti prestazionali nelle funzioni di ordine superiore e lambda function, dimostrandosi piú ottimizzato e veloce nei confronti di Java, che ha introdotto queste nuove funzionalitá solo dalla versione 8. Altri miglioramenti si possono notare nella memorizzazione delle variabili, poiché Kotlin utilizza una buona gestione dell'Autoboxing e alcuni miglioramenti nelle funzioni locali.

### 1.2.3 Coroutines

Dalla versione 1.1 Kotlin introduce in fase sperimentale le Coroutines, permettendo agli sviluppatore di testarne le funzionaliá, semplicemente aggiungendo nel Grandle un eccessione

Listing 1.6: Grandle Coroutines

```
kotlin {
 experimental {
```

```
  coroutines 'enable'
 }
}
```

Le Coroutines sono un modo per scrivere sequenzialmente programmi che operano in maniera asincrona, la differenza sostanziale é il modo in cui vengono scritte la parti di codice asincrone, infatti le coroutines permettono di scrivere linee di codice una dopo l'altra con la possibilitá di sospendere l'esecuzione e attendere momentaneamente che un risultato sia disponibile e successivamente riprendere l'esecuzione, aumentano la facilitá di lettura del codice e migliorando l'utilizzo della memoria a differenza dei thread.
Queste operazioni spesso riguardano la gestione di processi che operano in rete ( network I/O) o che sfruttano i thread per un uso intensivo della CPU e GPU, bloccando l'utilizzo del dispositivo fino alla loro terminazione.
La soluzione offerta da Java é quella tradizionale, che consiste nel creare un thread che lavora in background, ma in termini di prestazioni é svantaggioso poiché creare e gestire molti thread é un operazione costosa e complessa.
Attraverso le Coroutines invece é possibile sospendere funzioni che possono interrompere l'eseguzione del programma principale e riprenderle successivamente, queste funzioni vengono chiamate "funzioni di sospensione" e sono contrassegnate con la keyword "suspend".
La sospensione e la ripresa di queste funzioni é ottimizzata per avere un costo quasi nullo.

Le funzioni di sospensione, sono normali funzioni con parametri e valori di ritorno che permettono di sospendere una coroutine (la libreria puó dedicere di proseguire l'eseguizione senza la sospensione, se il risultato é giá disponibile)

Listing 1.7: Grandle Coroutines

```
suspend fun doSomething(foo: Foo): Bar {
    ...
```

```
}
fun <T> async(block: suspend () -> T)
async {
    ...
    doSomething(foo)
    ...
}
```

Async é una normale funzione ( non di sospensione ) che contiene un una funzione di sospenzione all'interno.

Le Coroutine sono completamente implementate attraverso una tecnica di compilazione (nessun supporto da parte della JVM o del sistema operativo), fondamentalmente, ogni funzione di sospensione viene trasformata in una macchina di stato, dove gli stati corrispondono a sospendere le chiamate. Subito prima della sospensione, lo stato successivo viene memorizzato in un campo di una classe generata dal compilatore insieme alle variabili locali. Alla ripresa di quella routine, le variabili locali vengono ripristinate e la macchina procede dallo stato successivo a quella della sospensione.

Molti meccanismi asincroni disponibili in altri linguaggi possono essere implementati con Kotlin utilizzano le coroutines, come ad esempio async/await di CSharp, "channels and select" di Go e "generators/yeald" di python.

## 1.3 Variabili

Kotlin utilizza due keyword differenti per dichiarare una variabile ( var e val ) seguita dal nome che si vuole assegnare alla variabile e il tipo opzionale della variabile ( nel caso non fosse definito il tipo, Kotlin attraverso il Type Inference, riconosce automaticamente il tipo della variabile in base al suo primo assegnamento)

- Var: Variabile mutabile, permette ad una variabile di modificare il suo valore con un riassegnamento durante l'eseguzione del programma o posticiparne l'inizializzazione, indicando solamente il tipo della variabile

- Val: Variabile Immutabile, permette di dichiarare una variabile di sola lettura ( equivalente a final in Java). L'inizializzazione di una variabile val non puó essere posticipata

Un'ultima caratteristica introdotta da Kotlin nell'inizializzazione di una nuova variabile sono la "Lazy Initialization " e la "Late Initialization", due nuovi modi per inizializzre una variabile.

- Lazy: consente di delegare ad una funzione l'inizializzazione della variabile, il risultato della funzione verrá assegnato alla variabile, in seguito quando verrá effettuato l'accesso alla variabile la funzione non sará rieseguita ma verrá solamente passato il valore

- Late: permette di posticipare l'inizializzazione di una variabile, se si tenterá di acceddere alla variabile prima che essa venga inizializzata si riceverá un errore. Late é stato principalmente introdotto per supportare la "dependency injection", ma puó essere comunque utilizzato dal programmatore per scrivere codice efficiente

Listing 1.8: Late init

```
lateinit var prova: String
val lazyString = lazy { readStringFromDatabase() }
```

### 1.3.1 Autoboxing

Java pone due differenze quando si parla di variabili, mette a disposizione i principali tipi primitivi (int, boolean, byte, long, short, float, double, char)

e i le loro corrispondenti classi (Int, Boolean, Byte, Long, Short, Float, Double, Char).
Uno dei principali cambiamenti introdotti da Kotlin é stato quello di rendere accessibile allo sviluppatore tutte le varibili come se fossero oggetti.
La differenza fra i tipi primitivi e gli oggetti risiede nel loro utilizzo, i primi indicano solamente il tipo di una variabile, mentre gli oggetti incapsulano il tipo e ne aggiungono funzionalitá e metodi aggiuntivi, inoltre il tipo primitivo non puó assumere valore nullo.
Kotlin operando ad alto livello, rimuove e astrae le due distinzioni poiché di default quando viene inizializzata una nuova variabile, la identifica come un oggetto, consentendo allo sviluppatore di utilizzare i metodi aggiuntivi ad esso associati e solo in fase di compilazione il compilatore di Kotlin controllerá se l'oggetto é strettamente necessario o puó essere sostituito dal suo corrispondente tipo primivio.

| Tipo | Oggetto | Dimensione |
|---|---|---|
| int | Int | 32 bits |
| boolean | Boolean | 1 bits |
| byte | Byte | 8 bits |
| long | Long | 64 bits |
| short | Short | 16 bits |
| float | Float | 32 bits |
| double | Double | 64 bits |
| char | Char | 16 bits |

L'unico tipo introdotto da Kotlin é "Nothing", un tipo senza istanze, molto simile al concetto del tipo Any.
Any é superclasse di tutti i tipo, Nothing contrariamente é la sottoclasse di tutti i tipi.
Nothing viene utilizzato dal compilatore per indicare che una funzione non ritorna nessun valore, in particolare viene utilizzato per indicare che é presente un loop infinito, oppure per inizializzare una variabile che non contiene nessun

elemento, infatti é la base per definire le funzioni emptyList(), emptySet(), introdotte da Kotlin.

### 1.3.2 Optional

Le variabili sono pressocché le stesse che sono presenti in Java, con la particolariá che Kotlin cerca di evitare alcuni problemi dovuti a referenze a puntatori nulli ( NullPointerException ).
Kotlin richiede che una variabile a cui assegnamo un valore nullo sia dichiarata con l'operatore "?", in caso contrario mostrerá un errore in fase di compilazione.

Listing 1.9: Esempio

```
var esempio1: String? = null //corretto
var esempio2: String = null //errore
```

Il safe call operator "?" serve ad indicare che la variabile puó assumere in qualsiasi momento un valore nullo, e lascia al programmatore la responsabilitá e la possibilitá di accedervi ugualmente per leggerne il valore, con l'utilizzo dell operatore "!!".

Listing 1.10: Esempio !!

```
val nome = getName()!!
```

In altrnativa, attraverso il Smart Casting, l'operatore "!!" si puó omettere, poiché il compilatore capisce automaticamente che la variabile non potrá assumere il valore nullo.

Listing 1.11: Smart Casting

```
fun getName(): String? {..}
val name = getName()
if (name != null) {
  println(name.length)
}
```

```
//forma contratta
println(name?.length)
```

### 1.3.3 String Template

La gestione delle stringhe in Kotlin si differisce dalla gestione di Java per l'aggiunta di nuove caratteristiche, tra cui il "String Template" disponibile giá in altri linguaggi. Lo string Template consiste nel fare riferimento a variabili, durante la rappresentazione di stringhe, aggiungendo il prefisso "$" al nome della variabile, nel caso si volesse accedere ad una sua proprietá é necessario utilizzare le parentesi graffe dopo il prefisso. "$".

Listing 1.12: Esempio String template

```
val name = "Sam"
%val str = hello $name. Your name has ${name.length} characters
```

## 1.4 Funzioni

Le funzioni sono definite utilizzando la parola "fun" seguite dal nome della funzione, i parametri opzionali e il valore di ritorno anch'esso opzionale.
La visibilitá di una funzione di default é "public" ma come in Java puó essere modificata, indicando il tipo di visibilitá, seguito dalla definizione della funzione.

Listing 1.13: Esempio Funzione Kotlin

```
fun saluta(nome: String): String {
  return "Ciao $nome"
}
```

Gli argomenti delle funzioni in Kotlin possono assumere il valore passato dal chiamante della funzione oppure avere un valore di default.
Questa caratteristica oltre ad essere utile al programmatore che eviterá di

inserire controlli all'interno di funzioni o addirittura creare un'altra funzione con parametri diversi, aumenta la leggibilitá del codice, rendendolo piú diretto e comprensivo.
Un'ultima caratteristica riguardante i parametri delle funzioni é la possibilitá di indicare l'ordine e il valore a cui assegnare il dato passato per parametro, indicando il nome del parametro della funzione seguito dal simbolo "="

Listing 1.14: Esempio Kotlin Parametri

```
fun buyItem(id:String, status: color = true){...}
buyItem(id=23) // oppure semplicemente: buyItem(23)
```

Tutte le funzioni devono restituire un valore, qualora non ci fosse, il valore di default é "Unit" corrispondente a "void" in java, l'unica eccessione viene fatta per le "Single expression functions", ovvero funzioni che vengono sritte in una sola linea, solitamente formate da un' unica espressione.

Listing 1.15: Esempio Single Exp. Function

```
fun quadrato(k: Int) = k * k
```

### 1.4.1 Funzioni Locali

Nei linguaggi di programmazione le funzioni sono state introdotte per ridurre la ripetizione di codice giá scritto e migliorarne la sua leggibilitá.
Il concetto chiave risiede quindi nel creare tante funzioni che eseguono determinate operazioni e restituiscono un valore al chiamante. Kotlin amplia le funzionalitá delle funzioni rendendo disponibili le "Local Function" ovvero funzioni che possono essere definite e richiamate all'interno di altre funzioni, con il vantaggio di poter accedere a variabili definite nello scope esterno.

Listing 1.16: Esempio Funzioni locali

```
fun printArea(width: Int, height: Int): Unit {
  fun calculateArea(): Int = width * height
  val area = calculateArea()
```

```
    println("The area is $area")
  }
```

### 1.4.2 Funzioni di ordine superiore

Le funzioni di ordine superiori sono funzioni che possono accettare come argomento di una funzione: una funzione stessa, o di restituirne una.
Queste funzioni sono utilizzate molto nella programmazione funzionale ma sono state introdotte anche in linguaggi imperativi, ispirandosi al paradigma della programmazione funzionale.

Listing 1.17: Funzioni ordine superiore

```
fun esempio(str: String, fn: (String) -> String): Unit {
  val prova = fn(str)
  println(prova)
}
```

Java ha incominciato a includere le lambda nel suo linguaggio solo dalla versione 8, costringendo a tutti gli utilizzatori di versioni precedenti alla 8 di affidarsi a delle librerie esterne che forzavano l'utilizzo delle lambda su Java. Kotlin supporta le lambda nativamente senza l'utilizzo di librerie esterne e limitazioni aggiuntive. Le lambda function e altri costrutti di linguaggi funzionali possono essere utili anche durante la programmazione di un applicazione Android, ove é richiesto ad esempio il passaggio di funzioni come parametro ad altre funzioni, chiamate asincrone verso un server esterno o localmente per interagire con l'interfaccia utente.

Listing 1.18: Esempio Kotlin Programmazione funzionale

```
val lista: List = listOf("OS Windows", "Smartphone", "OS LINUX",
    "OS Android", "RAM", "OS IOS", "Scarpe")
println(strings.filter { it.startsWith("OS") }.map {
    it.toLowerCase() }.joinToString())
```

```
// "os windows, os linux, os android, os ios"

//lambda
val sum = { x: Int, y: Int -> x + y }
```

## 1.5 Classi

Kotlin come Java é un linguaggio orientato agli oggetto, oltre a mantanere i concetti fondamentali della programmazione ad oggetti, rimuove alcune verbositá caratteristiche di molti linguaggi orientati agli oggetti.
Istanziare una classe in Kotlin é molto semplice e intuitivo poiché occorre chiamare direttamente il costruttore, senza dover utilizzare keywords aggiuntive come "new".

Listing 1.19: Esempio class Kotlin

```
val palla = Pallone(type="calcio", color="red")
```

### 1.5.1 Data Class

Nella programmazione Java e di altri linguaggi orientati agli oggetti vengono create classi per rappresentare modelli che verranno usati dall'applicazione per memorizzare valori o altri oggetti, questi modelli devono contenere i metodi get e set per leggere e settare i valori di un oggetto, Kotlin per rendere meno verbosa la creazione di classi, introduce il marcatore "data" permettendo al programmatore di scrivere solamente il costruttore senza dover pensare alla creazione dei metodi get, set, equals, toString, hashCode, tipicamente utilizzati nelle stesura di classi Java.
Questo rende il codice delle classi in Kotlin molto piú coinciso e leggibile.

Listing 1.20: Esempio dataclass Kotlin

```
data class User(val name: String, var password: String)
```

Listing 1.21: Esempio classe Java

```
public class User {
 private String name;
 private String password;
 public User(String name, String password) {
  this.name = name;
  this.password = password;
 }
 public String getName() {
  return this.name;
 }
 public String getPassword() {
  return this.password;
 }
 public void setName(String name) {
  this.name = name;
 }
 public void setPassword(String password) {
  this.password = password;
 }
```

## 1.6 Strutture e flussi di controllo

Kotlin utilizza le piú comuni strutture di controllo come if..else, try..catch, for e while introducendo il controllo "when" e aggiungendo funzionaliá aggiuntive rispetto a Java.

Listing 1.22: When kotlin

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
```

```
    x.isOdd() -> print("x is odd")
    else -> print("none of the above")
}
```

Un espressione é una dichiarazione che valuta una valore e ne restituisce un risultato, e si differeisce da una semplice dischiarazione che non restituisce nulla.

Listing 1.23: Espressione

```
"hello".startsWith("h")
```

Listing 1.24: Dischiarazione

```
val number = 2
```

Le comuni strutture di controllo in Java sono considerate dichiarazioni che non valutano nessun valore, in Kotlin invece, tutti i flussi di controllo restituiscono un valore.

Listing 1.25: esempio espresioni kotlin

```
//Tradizionale
var max: Int
if (a > b)
  max = a
else
  max = b
// Espressione
val max = if (a > b) a else b
```

## 1.7 Kotlin Android Extensions

La struttura di un classico progetto Android comporta il continuo utilizzo di View Binding attraverso l'utilizzo di keyword come "findViewById()" che

prende come parametri il riferimento alla view con la quale si vuole interagire.
L'utilizzo continuo di questa funzione oltre a rendere poco leggibile il codice, provoca spesso numerosi bug dovuti ad un cattivo assegnamento di identificativi, Kotlin lasciando inalterato il funzionamento interno di findViewById, ha scelto di utilizzare un approccio simile a molte librerie[6] che cercarono di risolvere il problema della chiamata a findViewById ogni qualvolta si volesse interagire con un elemento di un layout.
Il funzionamento di queste librerie era molto semplice e intuitivo, utilizzavano le annotazioni offerte da Java per evitare codice inutile, bastava infatti fare riferimento all'ID della view e successivamente utilizzarla, in base al nome dato dal programmatore, infine in fase di compilazione veniva generato il codice con il findViewById().
La soluzione offerta da Kotlin risulta essere ancor piú semplice ed intuitiva ed é: Kotlin Android Extensions, nata come una libreria assestante per poi essere integrata all'interno del linguaggio dalla versione 1.0, permettendo di fare riferimento ad una view scrivendo direttamente il suo Id, e importando il riferimento al layout.

Listing 1.26: Esempio Java

```
TextView demo = findViewById(R.id.textView) as TextView;
demo.setText("hello")
```

Listing 1.27: Esempio Java + Libreria esterna

```
@BindView(R2.id.user) EditText username;
username.setText("hello")
```

Listing 1.28: Esempio KAE

```
import kotlinx.android.synthetic.main.activity_main.*
username.setText("hello");
```

[6] https://github.com/JakeWharton/butterknife

# Capitolo 2

# Firebase

## 2.1 Storia

Firebase é una piattaforma Mobile backend as a service (MBaaS) che consente di interfacciare applicazioni mobili e web app ad un cloud backend, fornendo allo sviluppatore servizi utili per la gestione degli utenti, storage, notifiche push e altri strumenti di analisi e sviluppo.
Questo tipo di modello é relativamente recente poiché si basa sul cloud computing, fornendo uno servizio globale e uniforme per connettere client differenti con la sincronizzazione dei dati in tempo reale.
Lo sviluppo di Firebase inizió dall'omonima azienda che nel 2011 sviluppó la piattaforma con l'idea di fornire un servizio in grado di sincronizzare dati in tempo reale, successivamente ricevetté grande interessa da parte di Google che nel 2014 acquistó[1] Firebase e altre startup simili, integrandole con i suoi servizi Google Cloud Platform.

[1]https://techcrunch.com/2014/10/21/google-acquires-firebase-to-help-developers-build-better-realtime-apps/

## 2.2 Servizi

Firebase offre diversi servizi, mettendo a disposizione anche SDK (software development kit ) e API ( Application programming interface) multipiattaforma ( Android, Ios, JavaScript, C++ e Unity) per interagire con essi. I servizi offerti da Firebase sono circa 20, realizzati per facilitare lo sviluppo e la gestione del backend, permettendo ad uno sviluppatore di concentrarsi solamente sulla parte client.
Tra i vari servizi forniti quelli piú utili nell' ambito dello sviluppo software sono:

- **Firebase Cloud Messaging**: Soluzione cross-platform per la gestione di notifiche push su Android iOS e Web
- **Firebase Auth**: Servizio per la gestione degli utenti con il supporto del social login per Facebook, Github, Twitter, Google
- **Realtime Database**: Database noSQL, con il supporto della sincronizzazione in real teim dei dati fra diversi client in diversi linguaggi di programmazione
- **Firebase Storage**: Servizio che offre trasferimento e hosting sicuro dei file
- **Firebase Hosting**: Web hosting che fornisce file utilizzando CDN (content delivery network) e HTTP Secure (HTTPS)
- **Cloud Functions**: Servizio che permette di scrivere script JavaScript eseguiti ogni volta che vi é un cambianeto nel database Firestore/Real Time Database
- **Firestore Database**: Database noSQl basato su documenti, con accesso ai dati offilne scalabile e con il supporto della sincronizzazione in real time

## 2.3 Database

Google offre due differenti database con supporto della sincronizzazone dei dati in tempo reale:

- **RealTime Database**
- **Cloud Firestore**

**RealTime Database** é un cloud database NoSQL che memorizza i dati in un unico file JSON. Il database risultante risulta avere una struttura ad albero con un unica radice, la struttura é senza schema e puó quindi cambiare nel tempo.
Utilizzando l'SDK del RealTime Database i dati vengono mantenuti localmente, e anche quando sono offline, gli eventi in tempo reale continuano a incendiarsi, dando all' utente finale un' esperienza reattiva.
Quando il dispositivo riacquista la connessione, il Realtime Database sincronizza le modifiche dei dati locali con gli aggiornamenti remoti che si sono verificati mentre il client era offline, unendo automaticamente eventuali conflitti.

**Cloud Firestore** é un cloud database NoSQL basato sulla memorizzazione dei dati sottoforma di documenti e collezzioni di documenti, anche la sua struttura é senza schema e puó quindi cambiare nel tempo.
Il modello di memorizzazione dei dati é basato sui documenti che possono contenere dati come stringhe e numeri, oggetti complessi e annidati.
Questi documenti sono archiviati in raccolte, che sono collezzioni per i documenti utilizzate per organizzare i dati e creare query.
É inoltre possibile creare subcollezioni all'interno dei documenti e creare strutture gerarchiche di dati che scalano man mano che il database cresce.
Gli unici limiti inposti da Firestore sono la dimensione di un singolo documento che é di 1 MiB (1,048,576 bytes) e un massimo di 100 collezzioni annidate.
Il database Firestore mantiene i dati aggiornati attraverso una buona comunicazione client-server, i client attraverso l'utilizzo delle librerie SDK e di

listener offrono una sincronizzazione in tempo reale del dati.
L'aggiunta di listener oltre a informare il client su modifiche effettuate nel database, permette di memorizzare le richieste effettuate in precedenza e mantenere una copia delle risposte del server nella cache, offrendo quindi un supporto offline dei dati.

I tipi di dato messi a disposizione da Firestore sono:

| **Tipo** | **Descrizione** |
|---|---|
| Array | non puó contenere un altro valore array. |
| Boolean | falso, vero |
| Date | Memorizzato in formato timestamp |
| Float | precisione numerica a 64 bit |
| Geo Point | Punto geografico contentente latitudine e longitudine |
| Integer | Intero Numerico a 64 bit |
| Map | Rappresenta un oggetto |
| Null | valore nullo |
| Reference | Riferimento ad un'altro documento nel database |
| String | Stringa di testo codificata in UTF-8. |

Tabella 2.1: Data type firestore

L' interrogazione del database Firestore attraverso query risulta essere molto espressivo ed efficiente. É possibile creare query per filtrare solo alcuni dati all'interno di un documento o filtrare collezzioni, con la caratteristiche basilari delle interrogazioni: l'ordinamento, il filtraggio e limiti sui risultati di una query. Si possono filtrare anche sottocampi di un oggetto map, ma non é possibile filtrare o ordinare un elemento di tipo Reference che serve solo ad indicare il riferimento di un documento all'interno del database.
Cloud Firestore offre un SDK con una buona integrazione per dispositivi mobili Android, iOS e web apps, ma permette l'utilizzo dei servizi anche offrndo SDK aggiuntive per altri linguaggi di programmazione come: NodeJS, Java,

Python, e GO.

Ricapitolando, possiamo definire Firestore come una nuova versione di Firebase RealTime con una miglior struttura interna di memorizzazione dei dati e una espressivitá della query maggiore.

| **RealTime Firebase** | **Firestore** |
|---|---|
| Memorizza i dati in un unico file JSON | Memorizza i dati in collezzioni contententi documenti |
| Supporto per i dati offline su Android e iOS | Supporto per i dati offline su Android, iOS e Web |
| Depp Query con ordinamento e condizioni sui dati limitate | Query con ordinamento, condizioni sui dati, indicizzazione, alte performance |
| Memorizzazione di dati come singole operazioni. | Memorizzazione e transizioni sui dati atomiche. |
| Validazione dei dati manuale, e settaggio manuale di regole di protezione sui dati | Validazione dei dati automatica, e regole di protezione sui dati manuali |

Tabella 2.2: Cofronto dei due database Firebase

### 2.3.1 Database Rules

Firebase offre per i suoi due database la possibilitá di inserire delle restrizioni e regole di sicurezza per l'accesso, chiamate Database Rules.
Le Database Rules determinano chi ha accesso in lettura e scrittura al database o a collezzioni di dati all'interno del database, queste regole sono presenti sui server Firebase e vengono applicate automaticamente ad ogni modifica.
Ogni richiesta di lettura e scrittura di dati nel database sará completata solo se le regole lo consentono.
Entrambi i database: Real time e Firestore supportano le Database Rules, le differenze fra i due servizi riguardano il metodo con cui vengono scritte le

regole e il tipo di controlli che é possibile effetuare.
Firebase permette di sccrivere le regole utilizzando un file in formato JSON in cui vengono definite le regole in base alla collezzione in cui si trovano i dati, alla validazione dei dati o in base all'utente loggato su Firebase Auth.
Le regole applicate al database RealTime hanno una sintassi simile a JavaScript e mettono a disposizione del programmatore quattro tipi di controlli:

| **Tipo** | **Descrizione** |
|---|---|
| .read | Descrive se e quando i dati possono essere letti dagli utenti. |
| .write | Descrive se e quando i dati possono essere scritti dagli utenti |
| .validate | Definisce l'aspetto di un valore formattato correttamente, se ha attributi figli e il tipo di dato |
| .indexOn | Specifica una collezzione da indicizzare per supportare l' ordine e l' interrogazione |

Tabella 2.3: Firebase Rules

La stesura delle regole per il database RealTime viene memorizzata in formato JSON sui server Firebase, un esempio di alcune regole applicate su un database é il seguente:

Listing 2.1: Firebase Rules esempio

```
{
  "rules": {
    "users": {
      "$uid": {
        ".write": "$uid === auth.uid"
      }
    },
    "collection2": {
     ".validate": "newData.isString() && newData.val().length < 100"
   }
  }
```

```
}
```

Le regole di sicurezza del database Firestore sono simili a quelle del Real-Time Database ma prevedono un controllo degli accessi e della convalida dei dati in un formato piú semplice ma espressivo.
Tutte le regole di sicurezza Cloud Firestore sono costituite da dichiarazioni di corrispondenza chiamate "match" che identificano i documenti nel database e consentono la creazione di espressioni che controllano l' accesso a tali documenti.
Oltre alle regole di scrittura, letttura, condalidazione é possibile creare funzioni ausiliarie per semplificare e rendere piú intuitivo la scritture delle regole.
Un esempio di scrittura di alcune regole su un database Firestore é il seguente:

Listing 2.2: Firestore Rules

```
service cloud.firestore {
  match /databases/{database}/documents {
    function signedInOrPublic() {
      return request.auth.uid != null || resource.data.visibility
          == 'public';
   }
  match /cities/{city} {
      allow read: if request.auth.uid != null;
      allow create: if
          exists(/databases/$(database)/documents/users/$(request.auth.uid))
  }
 }
}
```

## 2.4 Cloud Functions

Le Cloud Functions consentono di eseguire automaticamente il codice backend in risposta a modifiche effettuate nel database Firestore o RealTi-

me.
I linguaggi utilizzati per scrivere le Cloud Functions sono JavaScript e TypeScript, una volta scritta una funzione essa viene memorizzata e gestita dai server Google e man mano che il carico aumenta o diminuisce, Google scala automaticamente il numero di istanze di server virtuali necessari per eseguire le funzioni.

Il ciclo di vita di una funzione é il seguente:

- Lo sviluppatore scrive il codice per una nuova funzione, selezionando un provider di eventi (Realtime Database, Firestore) e definisce le condizioni in cui la funzione deve essere eseguita
- Lo sviluppatore tramite un tool a linea di comanda invia la funzione sui server di Firebase
- Quando il provider dell' evento genera un evento che corrisponde alle condizioni della funzione, il codice viene eseguito
- Se sono presenti piú eventi da gestire contemporaneamente, Google creerá piú instanze per gestire il lavoro piú velocemente
- Quando lo sviluppatore aggiorna la funzione distribuendo il codice aggiornato, tutte le istanze per la vecchia versione vengono ripulite e sostituite da nuove istanze
- Quando uno sviluppatore cancella la funzione, tutte le istanze vengono ripulite e la connessione tra la funzione e il provider dell' evento viene rimossa

Cloud Functions mette a disposizione 4 tipi di controllo sui dati del database:

| Evento | Trigger |
|---|---|
| onCreate | Attivato quando si scrive un documento per la prima volta. |
| onUpdate | Attivato quando esiste giá un documento e se ne modifica un valore. |
| onDelete | Attivato quando un documento con dati viene eliminato. |
| onWrite | Attivato quando si attiva onCreate, onUpdate o onDelete. |

Tabella 2.4: Firestore Rules

La scrittura di una funzione viene effettuata con la scrittura di funzioni JavaScript o TypeScript, indicando il documento o collezzione sui cui si vuole fare riferimento seguito dal tipo di evento:

Listing 2.3: Cloud functions esempio 1

```
exports.myFunctionName =
    functions.firestore.document('users/koci').onWrite((event) => {
    ...
  });
exports.modifyUser =
    functions.firestore.document('users/{userID}').onWrite(event =>
    {
    //  documento sottoforma di oggetto
    var document = event.data.data();
    // documento precedente alla modificarlo
    var oldDocument = event.data.previous.data();
    ...
});
```

### 2.4.1 Esempi di utilizzo

Le cloud functions possono essere utilizzati anche interagendo con gli altri servizi Firebase, un esempio di integrazione tipico potrebbe essere quello di creare un'anteprima di un immagine e salvarla su Firebase storage:

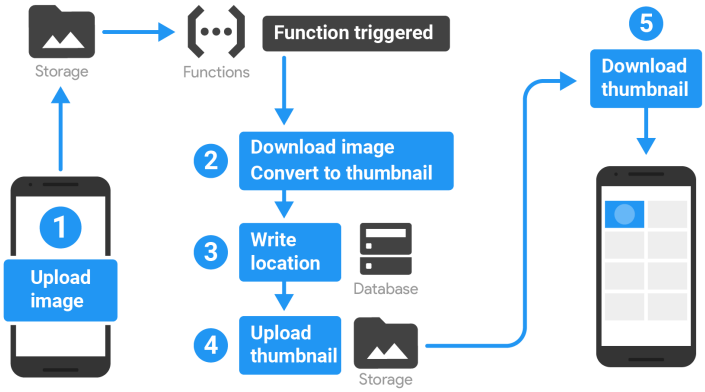


Figura 2.1: Firebase Storage e Cloud Functions esempio 1

In questo esempio un funzione si attiva quando un immagine viene caricata in sullo storage firebase, la funzione scarica l'immagine e ne crea una versione miniaturizzata, in seguito scrive il riferimento della miniatura sul database, in modo che un'applicazione client possa trovarla e utilizzarla.

Un'altro esempio é l'utilizzo delle cloud functions per effettuare controlli su un tipo di linguaggio inappropriato all'interno di una chat,forum o commento:

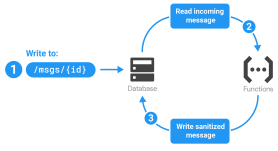


Figura 2.2: Firebase Storage e Cloud Functions esempio 2

Quando un messaggio viene scritto all'interno della collection "msgs" la funzione che ascolta gli eventi di scrittura ne recupera i dati, elabora il testo per rilevare la presenza di un linguaggio non appropriato e riscrivere il messaggio ricevuto correggendolo o eliminando parti di linguaggio volare, applicando cosi una moderazione del testo lato server.

## 2.5 Cloud Messaging

Firebase Cloud Messaging (FCM) é una soluzione di messaggistica multipiattaforma che consente di inviare messaggi tra dispositivi con la possibiltá di notificare a una o piú applicazioni client che sono disponibili nuovi dati di sincronizzazione.

I messaggi si differenziano in due tipi:

- Messaggi di notifica
- Messaggi di dati

É possibile inviare messaggi tramite l'Admin SDK o le API HTTP e XMPP, il carico utile massimo per entrambi i tipi di messaggio é 4KB, tranne quando si inviano messaggi dalla console Firebase, che impone un limite di 1024 caratteri.
La differenza fra i due tipi di messaggi é la loro composizione: i messaggi di notifica contengono un set predefinito di chiavi visualizzabili dall' utente, mentre i messaggi di dati, contengono solo le coppie di valori definite dall'utente.
Entrambi i messaggi richiedono di definire il campo obbligatorio "token" che é un riferimento ad dispositivo o ad un gruppo di dispositivi, questo token é generato tramite l'SDK di Firebase e deve essere memorizzato su un server per poter essere riutilizzato, in caso contrario l'SDK permette di aggiornare il token del dispositivo.

### 2.5.1 Invio messaggi

Firebase Cloud Messaging é usufruibile solo se si utilizza l'apposito SDK e si ottiene un token, necessario per inviare un messaggio a un dispositivo specifico.
All'avvio iniziale dell' applicazione, l'SDK FCM genera un token di registrazione per l'istanza dell' applicazione client.
I token e la ricezione dei messaggi possono essere controllati creando una classe che ha come estensione la classe "FirebaseInstanceIdService" per la gestione dei token e la classe "FirebaseMessagingService" per la gestione dei messaggi ricevuti.

Il token di registrazione puó cambiare quando:

- L' applicazione elimina l' ID istanza
- L' applicazione viene ripristinata su un nuovo dispositivo
- L' utente disinstalla/riinstalla l' app
- L' utente cancella i dati delle app.

FCM tramite l'SDK e richieste HTTPS consente anche la creazione di gruppi di utenti, permettendo con un solo token di inviare un singolo messaggio a piú istanze di un' applicazione in esecuzione su dispositivi appartenenti a un gruppo. Tutti i dispositivi di un gruppo condividono una comune chiave di notifica, che é il token utilizzato da FCM per inviare i messaggi a tutti i dispositivi del gruppo.

Sulla base del modello di pubblicazione/iscrizione invece, FCM consente di inviare un messaggio a piú dispositivi che hanno si sono registrati ad un particolare argomento.
Le applicazioni client possono iscriversi a qualsiasi argomento esistente o creare un nuovo argomento. Quando un' applicazione client sottoscrive un

nuovo nome di argomento (uno che non esiste giá per il vostro progetto Firebase), un nuovo argomento di tale nome viene creato in FCM e ogni cliente puó successivamente sottoscriverlo.
Per iscriversi a un argomento, l' applicazione client chiama la funzione

Listing 2.4: FCM topic

```
FirebaseMessaging.getInstance().subscribeToTopic("news");
```

Per annullare l' iscrizione, l' applicazione client deve richiamare unsubscribeFromTopic () con il nome dell' argomento.

### 2.5.2 Parametri

Oltre al destinatario é possobile definire anche la prioritá di un messaggio, la durata, il suono, l'icona, il tempo di vita e altri parametri opzionali.

I principali parametri messi a disposizione da Firebase sono:

| **Paramentro** | **Descrizione** |
|---|---|
| Title | Titolo della notifica. |
| Body | Testo della notifica |
| Sound | Suono da riprodurre quando il dispositivo riceve la notifica. |
| Sottotitolo | Sottotitolo della notifica. |
| Icon | Icona della notifica |
| Timetolive | Specifica per quanto tempo (in secondi) il messaggio deve essere conservato se il dispositivo é offline. |
| Clickaction | L' azione associata ad un click della notifica. |

Tabella 2.5: FCM parametri

### 2.5.3 Prioritá

Esistono due opzioni per assegnare la prioritá di consegna ai messaggi: normale e ad alta prioritá. Il recapito di messaggi normali e ad alta prioritá

funziona in questo modo:

- Prioritá normale: i messaggi vengono inviati immediatamente quando l' app é in primo piano, quando invece il dispositivo é in modalitá Doze o l' applicazione é in standby app la consegna potrebbe essere ritardata per risparmiare la batteria, i messaggi in questo caso richiedono di pianificare un job FJD (Firebase Job Dispache) o un JobIntentService per gestire la notifica quando la rete sará nuovamaente disponibile.

- Alta prioritá: Il server Cloud Messaging tenta di inviare immediatamente il messaggio ad alta prioritá, consentendo al servizio di attivare un dispositivo sleeping, tramite l'SDK, e di eseguire alcune elaborazioni limitate (compreso un accesso alla rete molto limitato).

## 2.6 FirebaseUI

FirebaseUI é un insieme di librerie open-source per Firebase che consentono di semplificare lo sviluppo di un applicaione che utilizza Firebase. I miglioramenti vengono apportati attraverso una versione semplificata dell' autenticazione di Firebase fornendo metodi di facile utilizzo che si integrano con i piú comuni fornitori di identitá come Facebook, Twitter e Google, migliorare la gestone delle view con la sincronizzazione in tempo reale del database e funzionalitá aggiuntive per il servizio Firebase Storage

FirebaseUI dispone di moduli separati per utilizzare Firebase Realtime Database, Cloud Firestore, Firebase Auth e Cloud Storage.

- FirebaseUI Autorizzare

- FirebaseUIUI Firebase Firestore

- FirebaseUI Database

- FirebaseUI storage

FirebaseUI-Auth mira a massimizzare l'integrqzione integra con Smart Lock for Password per memorizzare e recuperare le credenziali, consentendo l' accesso automatico e single-tap sign-in, gestendo anche casi d'uso piú complessi come il recupero dell'account e il collegamento di account multipli che sono sensibili alla sicurezza e difficili da implementare correttamente utilizzando le API di base fornite da Firebase Auth.

FirebaseUI-Firestore semplifica il collegamento dei dati da Cloud Firestore all' interfaccia utente dell' applicazione, fornendo un adapter personalizzato per Firestore (FirestoreRecyclerAdapter) che consente la manipolazione automatica della sincronizzazione tra view e il database Firestore.