Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

UTILIZZO DI KOTLIN PER SVILUPPO DI APPLICAZIONI MOBILI: UN CASO DI STUDIO

Relatore:
Dott.
LUCA BEDOGNI

Presentata da:
ALESSIO KOCI

Sessione III
Anno Accademico 2016/17

A Claudio Maracci ...

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

L'idea di realizzare un'applicazione gestionale su piattaforma Android, è nata dall'esigenza di risolvere alcune problematiche reali riscontrate durante l'organizzazione di eventi, spese e commissioni da svolgere fra gruppi di persone.
Lo sviluppo iniziale si avvaleva di Java come linguaggio di programmazione, successivamente con l'annuncio ufficiale a Maggio 2017 del supporto di Google a Kotlin, come nuovo linguaggio di programmazione per Android, l'applicazione è stata riscritta completamente, utilizzando Kotlin.
Lo sviluppo dell'applicazione in Kotlin ha permesso di analizzare le funzionalità del nuovo linguaggio sia teoricamente che con un'implementazione pratica che ha reso possibile anche un confronto diretto con Java.
In questa tesi verranno illustrate le principali funzionalità e caratteristiche del linguaggio Kotlin, e del BaaS Firebase utilizzato per la gestione della parte server. Dal terzo capitolo invece verrà presa in considerazione l'applicazione, mostrando l'architettura ad alto livello dell'infrastruttura dei servizi server e delle funzionalità dell'applicazione. Successivamente verranno analizzate porzioni di codice rilevanti e discusse le implementazioni sviluppate per la realizzazione della parte client dell'applicazione. Infine nell'ultimo capitolo viene descritto l'attuale stato dell'arte, il confronto fra Java e Kotlin ed eventuali sviluppi futuri.

Capitolo 1

Kotlin

1.1 Storia

Kotlin è 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 è 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, rivelò durante un'intervista[2] che non esisteva ancora nessun linguaggio con le potenzialità e facilità d'uso richieste dal team di JetBrains, ad eccezione del linguaggio Scala che offriva grossi vantaggi nello sviluppo, ma aveva un tempo di compilazione molto lento, JetBrains scelse

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

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

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, è sempre stato uno dei linguaggi più 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 facilità sintattica, potenzialità 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 miglioramento di Java, ma dato che anche Android utilizzava la JVM, gli sviluppatori cercarono di adattarlo e renderlo ottimizzato anche per Android, come Java.
La comunità di sviluppatori che cominciò 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]

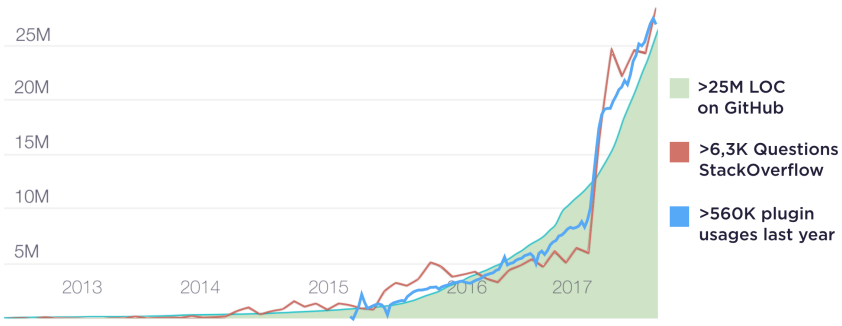

Figura 1.1: Grafico delle linee di codice scritte in Kotlin su Github.

[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, arrivò 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 potenzialità di Kotlin ne rimasero molto soddisfatti per la praticità, la stabilità ed i suoi benefici sintattici e funzionali, Kotlin è 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 garantito attraverso una buona integrazione con 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 è 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 funzionalità, fino a raggiungere nel 2017 la prima versione stabile: la 1.0.
Kotlin è 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 problematiche concrete riscontrati dagli stessi sviluppatori. Prendendo spunto dalle problematiche di Java e da buone regole introdotte da alcuni linguaggi imperativi e funzionali, Kotlin è stato modellato in modo tale da aggiungere funzionalità 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 è stata la buona integrazione del suo plugin con Android Studio. Il supporto dato dal plugin è tale da supportare il programmatore in ogni momento, proponendo la riscrittura di porzioni di codice per renderlo più coinciso, allertare il programmatore in caso di possibili puntatori nulli, offrire una conversione automatica del codice Java in Kotlin e ove possibile cercare di avvisare lo sviluppatore su possibili problemi di prestazione ed errori sintattici.
Le caratteristiche più importanti offerte da Kotlin sono l'interoperabilità 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 consentendo di differenziare il tipo nullabile e il tipo non-nullabile, prevenendo quindi errori di "NullPointerExpetions".
Il codice prodotto in Kotlin è inoltre più compatto, coinciso e meno verboso grazie alle dataclass, il supporto delle lambda function e altri costrutti utili.

1.2.1 Interoperabilità

I linguaggi Kotlin e Java sono fortemente intercompatibili, permettendo quindi a entrambi i linguaggi di coesistere all'interno dello stesso progetto e di richiamare 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 World in Kotlin

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

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

Listing 1.2: Hello World 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” produrrà un JAR eseguibile da Java “Hello.jar”

Listing 1.3: Compilazione di un programma Kotlin

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

É possibile chiamare classi Java all’interno di funzioni Kotlin e viceversa:

Listing 1.4: Chiamare Java da Kotlin

```
class KotlinClass {
    fun kotlinDoSomething() {
        val javaClass = JavaClass()
        javaClass.javaDoSomething()
        println(JavaClass().prop)
    }
}
```

Listing 1.5: Chiamare Kotlin da 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'esecuzione 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 dell'APK, Kotlin introduce circa 7000 metodi aggiuntivi a run-time che corrispondono ad un aumento di 1MB nell'APK finale.
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 sarà necessario utilizzare librerie esterne come: Guava, ButterKnife che spesso vengono importate in progetti Android aumentando considerevolmente la dimensione finale dell'APK.
In termini di performance Kotlin pone alcuni miglioramenti prestazionali nelle funzioni di ordine superiore e lambda function, dimostrandosi più ottimizzato e veloce nei confronti di Java, che ha introdotto queste nuove funzionalità solo dalla versione 8[6].
Altri miglioramenti di performance si possono notare nella memorizzazione delle variabili, poiché Kotlin utilizza una buona gestione dell'Autoboxing, permettendo di istanziare un oggetto solo quando è strettamente necessario, altri miglioramenti riguardano, le inline-function e le funzioni di ordine superiore che risultano più veloci rispetto a Java, poiché Kotlin supporta le lambda function nativamente a differenza di Java che per supportarle crea oggetti o utilizza chiamate virtuali.
Kotlin quindi non comporta alcun svantaggio al programmatore oltre all'aumento della dimensione finale dell'APK corrispondente a qualche MB aggiuntivo, di conseguenza pur non aggiungendo sostanziosi miglioramenti in termini di performance rispetto a Java, offre numerosi vantaggi sintattici.

[6] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

1.2.3 Coroutines

Dalla versione 1.1 Kotlin, introduce in fase sperimentale le Coroutines, consentendo agli sviluppatore di testarne le funzionalità, semplicemente aggiungendo nel file di configurazione "build.gradle", presente all'interno dell sorgente del progetto Android, un'eccezione:

Listing 1.6: Gradle Coroutines

```
kotlin {
 experimental {
  coroutines 'enable'
 }
}
```

Le coroutines offrono un modo per scrivere sequenzialmente, programmi che operano in maniera asincrona, la differenza sostanziale rispetto a Java e altri linguaggi è il modo e l'ordine con cui vengono scritte la parti di codice asincrone. Le coroutines permettono di scrivere istruzioni, una dopo l'altra, con la possibilità di sospendere l'esecuzione e attendere momentaneamente che un risultato sia disponibile e successivamente riprendere l'esecuzione, aumentano la facilità di lettura del codice e migliorando l'utilizzo della memoria a differenza dei thread.
Le operazioni che utilizzano i thread spesso riguardano la gestione di processi che operano in rete (network I/O), per leggere file locali o che sfruttano i thread per effettuare dei calcoli con un uso intensivo della CPU e GPU, bloccando l'utilizzo del dispositivo fino alla loro terminazione.
La soluzione offerta da Java è quella tradizionale, che consiste di creare un thread che opera in background, ma in termini di prestazioni è svantaggioso poiché creare e gestire molti thread è un operazione costosa e complessa.
Attraverso le coroutines di Kotlin invece è possibile sospendere funzioni che possono interrompere l'esecuzione del programma principale e riprenderle successivamente, queste funzioni vengono chiamate "funzioni di sospensione" e sono contrassegnate con la keyword "suspend".

Queste funzioni di sospensione, sono normali funzioni con parametri e valori di ritorno che permettono di sospendere una coroutine, il vantaggio è che la sospensione e la ripresa di queste funzioni è ottimizzata per avere un costo quasi nullo, inoltre la libreria può decidere di proseguire l'esecuzione senza la sospensione, se il risultato è già disponibile.

Listing 1.7: Esempio Kotlin Coroutines

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

Async è una normale funzione (non di sospensione) che contiene una funzione di sospensione 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 quello 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", queste sono seguite 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 tipo del suo primo assegnamento).

- **Var**: Variabile mutabile, permette ad una variabile di modificare il suo valore con un riassegnamento durante l'esecuzione 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 può 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 inizializzare una variabile.

- **Lazy**: Consente di delegare ad una funzione l'inizializzazione della variabile, il risultato della funzione verrà assegnato alla variabile, in seguito quando verrà effettuato l'accesso alla variabile la funzione non sarà rieseguita ma verrà solamente passato il valore.
- **Late**: Permette di posticipare l'inizializzazione di una variabile, se si tenterà di accedere alla variabile prima che essa venga inizializzata si riceverà un errore. Late è stato principalmente introdotto per supportare la "dependency injection", ma può essere comunque utilizzato dal programmatore per scrivere codice efficiente.

Listing 1.8: Esempio Late e Lazy Initialization in Kotlin

```
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 le loro corrispondenti classi (Int, Boolean, Byte, Long, Short, Float, Double, Char).
Uno dei principali cambiamenti introdotti da Kotlin è stato quello di rendere accessibile allo sviluppatore tutte le variabili, 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 funzionalità e metodi aggiuntivi, inoltre il tipo primitivo non può 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 controllerà se l'oggetto è strettamente necessario o può essere sostituito dal suo corrispondente tipo primitivo.

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

Tabella 1.1: Variabili primitive in Kotlin

L'unico tipo introdotto da Kotlin è "Nothing", un tipo senza istanze, molto simile al concetto del tipo "Any".
Any è superclasse di tutti i tipi, Nothing contrariamente è 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 è presente un loop infinito, oppure per inizializzare una variabile che non contiene nessun elemento, infatti è la base per definire le funzioni emptyList(), emptySet(), introdotte da Kotlin.

1.3.2 Optional

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

Listing 1.9: Esempio 1 Safe call operator Kotlin

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

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

Listing 1.10: Esempio 2 Safe call operator Kotlin

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

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

Listing 1.11: Smart Casting in Kotlin

```
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 già in altri linguaggi. Il string Template consiste nel fare riferimento a variabili, durante la rappresentazione di stringhe, aggiungendo il prefisso "$" al nome della variabile, nel caso si volesse accedere ad una sua proprietà è necessario utilizzare le parentesi graffe dopo il prefisso. "$".

Listing 1.12: Esempio String template Kotlin

```
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 visibilità di una funzione di default è "public" ma come in Java può

essere modificata, indicando il tipo di visibilità, 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 eviterà di inserire controlli all'interno di funzioni o addirittura creare un'altra funzione con parametri diversi, aumenta la leggibilità del codice, rendendolo più diretto e comprensivo.
Un'ultima caratteristica riguardante i parametri delle funzioni è la possibilità 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:Boolean = true){...}
buyItem(id=23) // oppure semplicemente: buyItem(23)
```

Tutte le funzioni devono restituire un valore, qualora non ci fosse, il valore di default assegnato da Kotlin è "Unit" corrispondente a "Void" in Java, l'unica eccezione viene fatta per le "Single expression functions", ovvero funzioni che vengono scritte in una sola linea, solitamente formate da un'unica espressione.

Listing 1.15: Esempio Single Expression Function in Kotlin

```
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 già scritto e migliorarne la sua leggibilità.
Il concetto chiave risiede quindi nel creare tante funzioni che eseguono determinate operazioni e restituiscono un valore al chiamante. Kotlin amplia le funzionalità 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 una funzione stessa, o restituirne una.
Queste funzioni sono utilizzate molto nella programmazione funzionale ma sono state introdotte anche nei 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 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 è un linguaggio orientato agli oggetti, oltre a mantenere i concetti fondamentali della programmazione ad oggetti, rimuove alcune verbosità caratteristiche di molti linguaggi orientati agli oggetti.
Istanziare una classe in Kotlin è molto semplice e intuitivo poiché occorre chiamare direttamente il costruttore, senza dover utilizzare keywords aggiuntive come "new", presente in Java.

Listing 1.19: Esempio classe in Kotlin

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

1.5.1 Data Class

Nella programmazione Java 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 più coinciso e leggibile.

Listing 1.20: Esempio Data Class in Kotlin

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

Listing 1.21: Esempio classe in 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 più comuni strutture di controllo come if..else, try..catch, for e while introducendo il controllo "when" e aggiungendo funzionalità aggiuntive rispetto a Java.

Listing 1.22: Sintatti "when" in 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 è una dichiarazione che valuta una valore e ne restituisce un risultato, e si differisce da una semplice dichiarazione che non restituisce nulla.

Listing 1.23: Espressione - Flussi di Controllo

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

Listing 1.24: Dischiarazione - Flussi di Controllo

```
val number = 2
```

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

Listing 1.25: Esempio espresioni in Kotlin

```
// Flusso di controllo come 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 la funzione "findViewById()" che prende come parametro il riferimento alla View con la quale si vuole interagire.
Il continuo utilizzo 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[7] 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 più semplice ed intuitiva ed è: 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, ed importando il riferimento al layout.

Listing 1.26: Esempio Kotlin Android Extensions

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

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

Capitolo 2

Firebase

2.1 Storia

Firebase è 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 ed altri strumenti di analisi e sviluppo.
Il modello su cui si basa la piattaforma è relativamente recente poichè appoggiandosi al cloud computing, fornisce uno servizio globale e uniforme per connettere client differenti offrendo una sincronizzazione dei dati in tempo reale.
Lo sviluppo di Firebase iniziò dall'omonima azienda che nel 2011 sviluppò la piattaforma con l'idea di fornire un servizio in grado di sincronizzare dati in tempo reale, successivamente ricevette grande interessa da parte di Google che nel 2014 acquistò[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++, Unity) per interagire con essi.
I servizi[2] offerti da Firebase sono circa 20, realizzati per facilitare lo sviluppo e la gestione del backend, permettendo ad uno sviluppatore di concentrarsi maggiormente sulla parte client e meno sulla manutenzione e gestione del backend.
Tra i vari servizi forniti, quelli più 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 tempo reale dei dati fra diversi client.
- **Firebase Storage**: Servizio che offre il trasferimento e l'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 eseguire script JavaScript ogni volta che vi è un cambiamento nel database.
- **Firestore Database**: Database NoSQL basato su documenti con il supporto della sincronizzazione in tempo reale.

[2]https://firebase.google.com/products/

2.3 Database

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

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

RealTime Database è un cloud database NoSQL che memorizza i dati in un unico file JSON. Il database è organizzato attraverso un modello gerarchico ad albero con un unica radice ed ha una struttura senza schema che può cambiare nel tempo.
Utilizzando l'SDK, tutte le richieste effettuate al RealTime Database vengono memorizzate localmente nella cache del client e vengono aggiornate in tempo reale al susseguirsi di modifiche ed eventi all'interno del Database.
Quando il dispositivo precedentemente offline riacquista la connessione, l'SDK del Realtime Database sincronizza le modifiche dei dati locali con gli aggiornamenti remoti che si sono verificati mentre il client era offline, risolvendo automaticamente eventuali conflitti.

Cloud Firestore è un cloud database NoSQL basato sulla memorizzazione dei dati sotto forma di documenti e collezioni, anche la sua struttura è senza schema e può quindi cambiare nel tempo.
Il modello di memorizzazione dei dati è basato sui documenti che possono contenere stringhe e numeri, date, oggetti complessi e annidati.
Questi documenti sono archiviati in raccolte, chiamate collezioni che contengono i documenti, ma è anche possibile creare sub-collezioni all'interno dei documenti e creare strutture gerarchiche di dati che scalano man mano che il database cresce.
Gli unici limiti imposti da Firestore sono: la dimensione di un singolo documento che è di 1 MiB (1,048,576 bytes) e un massimo di 100 collezioni annidate.

L'SDK del database Firestore mantiene i dati aggiornati attraverso una buona gestione del caching, i client invece utilizzando appositi listener offrono una sincronizzazione dei dati in tempo reale.
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 può 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: Tipi di dato Firestore

L' interrogazione del database Firestore attraverso query risulta essere molto espressiva ed efficiente. La creazione delle query permette di filtrare i dati all'interno di un documento o filtrare collezioni, 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 è 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 of-

frendo 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, una buona memorizzazione dei dati e una espressività delle query maggiore.

RealTime Firebase	**Firestore**
Memorizza i dati in un unico file JSON	Memorizza i dati in collezioni 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: Confronto dei due database Firebase

2.3.1 Database Rules

Firebase offre per i suoi due database la possibilità di inserire delle restrizioni e regole di sicurezza per l'accesso al Database, chiamate Database Rules. Le Database Rules determinano chi ha accesso in lettura e scrittura al database o a collezioni di dati all'interno del database, queste regole sono gestite utilizzando il pannello di controllo di Firebase, una volta scritte le regole queste vengono applicate automaticamente ad ogni modifica. Ogni richiesta di lettura e scrittura di dati nel database sarà 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 è possibile effettuare.
Firebase permette di scrivere le regole utilizzando un file in formato JSON dove vengono definite le regole in base alla collezione in cui si trovano i dati, alla validazione dei dati o in base all'utente registrato 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 collezione da indicizzare per supportare l'ordine e l' interrogazione

Tabella 2.3: Firebase Database Rules

Le regole per il database RealTime vengono memorizzate in formato JSON sui server Firebase, un esempio di alcune regole applicate è 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 validazione dei dati in un formato più semplice ed espressivo.
Tutte le regole di sicurezza 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, lettura, validazione è possibile creare funzioni ausiliarie per semplificare e rendere più intuitiva la scritture delle regole.
Un esempio di scrittura di alcune regole su un database Firestore è il seguente:

Listing 2.2: Firestore Database 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 script backend in risposta a modifiche effettuate nel database Firestore o RealTime, o sullo storage Firebase. I linguaggi utilizzati per scrivere le Cloud Functions sono JavaScript e TypeScript, una volta scritta una funzione essa viene memorizzata e gestita

dai server Firebase 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 è il seguente:

- Lo sviluppatore scrive il codice per una nuova funzione, selezionando un provider di eventi (Realtime Database, Firestore, Storage) 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 della funzione viene eseguito.
- Se sono presenti più eventi da gestire contemporaneamente, Google creerà più instanze per gestire il lavoro più velocemente.
- Quando lo sviluppatore aggiorna la funzione distribuendo il codice aggiornato, tutte le istanze per la vecchia versione vengono cancellate e sostituite da nuove istanze.
- Quando uno sviluppatore cancella una funzione, tutte le istanze vengono cancellate e la connessione tra la funzione e il provider dell' evento viene rimossa.

Cloud Functions mette a disposizione quattro tipi di controllo sul database:

Evento	**Trigger**
onCreate	Attivato quando si scrive un documento per la prima volta.
onUpdate	Attivato quando esiste già 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: Controlli Cloud Functions

La stesura di una funzione viene effettuata definendo funzioni JavaScript o TypeScript ed indicando il documento o la collezione 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 utilizzate anche interagendo con gli altri servizi Firebase, un esempio di integrazione tipico potrebbe essere quello di creare l'anteprima di un immagine e salvarla su Firebase Storage:

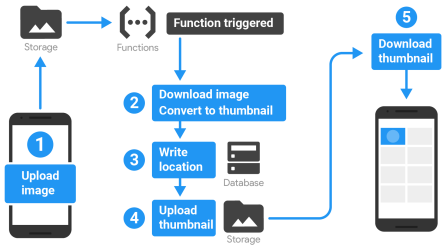

Figura 2.1: Firebase Storage e Cloud Functions esempio 1

In questo esempio, una funzione del servizio Cloud Function rimane in ascolto su cambiamenti nello Storage in atttesa che un'immagine venga caricata, Quando un utente aggiungerà un immagine nello storage, verrà richiamata la funzione che scaricherà l'immagine e ne creerà una versione miniaturizzata, in seguito scriverà il riferimento della miniatura sul database, in modo che un'applicazione client possa trovarla e utilizzarla. Un'altro esempio è l'utilizzo delle cloud functions per effettuare controlli su un tipo di linguaggio inappropriato all'interno di una chat,forum o commento: la funzione esamina il testo, rimuove il linguaggio volgare e restituisce il testo revisionato.

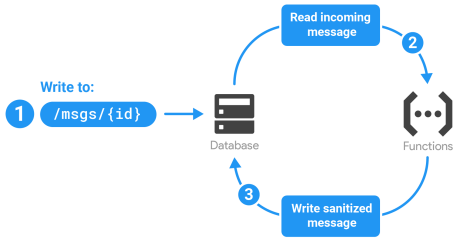

Figura 2.2: Firebase Storage e Cloud Functions esempio 2

2.5 Cloud Messaging

Firebase Cloud Messaging (FCM) è una soluzione di messaggistica multi-piattaforma che consente di inviare messaggi tra dispositivi con la possibiltà, di notificare a una o più applicazioni client che sono disponibili nuovi dati.
è possibile inviare messaggi tramite l'Admin SDK[3] o le API HTTP e XMPP, rispettando la dimensione massima di un messaggio corrispondente a 4KB.
I messaggi possono essere di due tipologie: messaggi di notifica e messaggi di dati, la differenza fra i due tipi di messaggi è il loro contenuto: i messaggi di notifica contengono un insieme predefinito di chiavi visualizzabili dall'utente, mentre i messaggi di dati contengono solo un insieme di chiave-valore definito da chi invia il messaggio.
Entrambi i messaggi richiedono di definire il campo obbligatorio "token" che è il riferimento ad un dispositivo o di un gruppo di dispositivi, questo token è generato tramite l'SDK di Firebase e deve essere memorizzato su un server per poter essere riutilizzato, in caso contrario l'SDK consente di aggiornare il token del dispositivo.

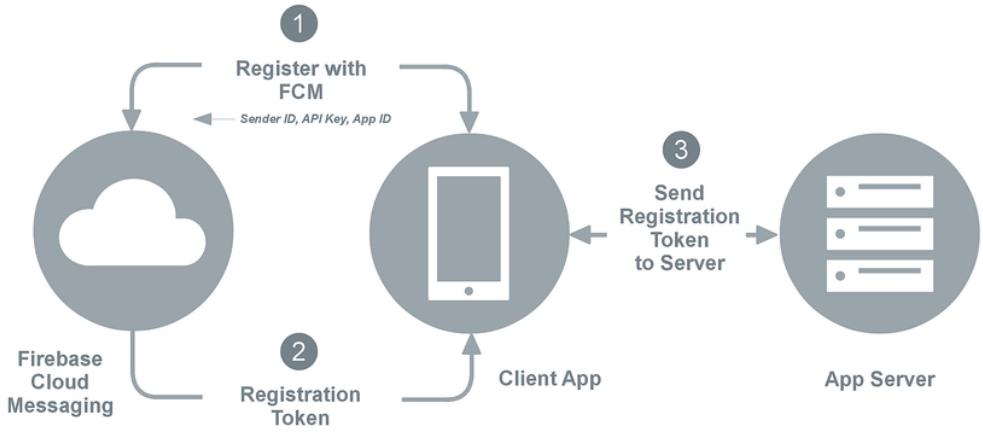

Figura 2.3: Processo di generazione di un token per il servizio FCM.

[3]https://firebase.google.com/docs/admin/setup

La ricezione dei messaggi di Cloud Messaging è possibile solo se si utilizza l'apposito SDK e si ottiene un token, necessario per inviare un messaggio ad 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 utilizzando le funzioni offerte dall'SDK. La classe "FirebaseInstanceIdService" viene utilizzata per la gestione dei token, mentre la classe "FirebaseMessagingService" viene utilizzata per la gestione dei messaggi ricevuti.
Il token di registrazione può cambiare quando l'applicazione elimina il token, ripristina, reinstalla, disnistalla l'app, o quando vengono eliminate le cache e i dati dal dispositivo.

2.5.1 Messaggi

I messaggi di dati inviabili tramite l'SDK o richieste HTTPS possono essere di due tipi:

- **Downstream Messaging**
- **Upstream Messaging**

I messaggi Downstream, permettono di inviare una notifica push dal Server verso il Client

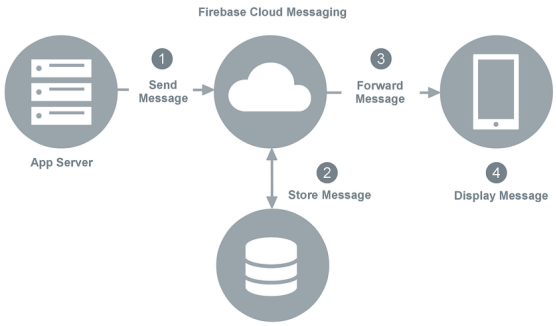

Figura 2.4: Esempio downstream messaging - Firebase Cloud Messaging

- Il server (per esempio Cloud Functions) invia il messaggio a FCM.
- Se il client non è disponibile, il server FCM memorizza il messaggio in una coda per la successiva trasmissione. I messaggi vengono conservati nella memoria FCM per un massimo di 4 settimane.
- Quando il dispositivo client è disponibile, FCM inoltra il messaggio all'applicazione.
- L'applicazione client riceve il messaggio da FCM, lo elabora e lo visualizza all'utente.

In alternativa oltre all'invio di un messaggio ad un singolo dispositivo FCM consente la creazione di gruppi di utenti, permettendo con un singolo token di inviare un messaggio a tutti i dispositivi appartenenti ad un gruppo. Il funzionamento dei messaggi Upstream invece è analogo e permette di inviare messaggi da un dispositivo client ad un server (ad esempio Cloud Functions).
Sulla base del modello di pubblicazione/iscrizione invece, FCM consente anche di inviare un messaggio a più dispositivi che si sono registrati ad un particolare argomento.

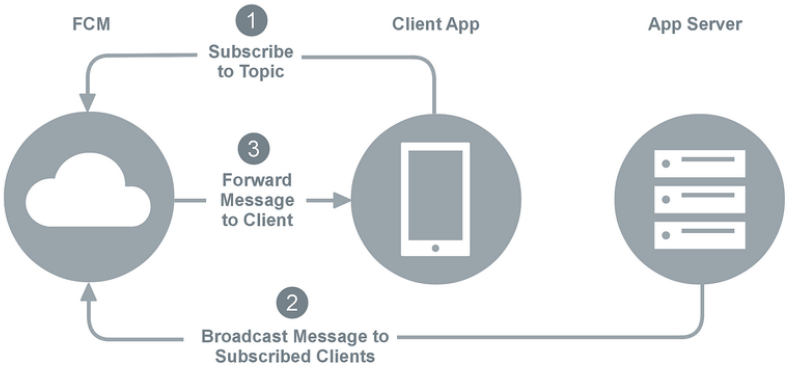

Figura 2.5: Esempio topic messaging - Firebase Cloud Messaging

- L'applicazione client si iscrive ad un argomento inviando un messaggio di sottoscrizione al server FCM.
- Il server invia messaggi tematici a FCM per la distribuzione.
- FCM inoltra messaggi tematici ai client che si sono registrati a tale argomento.

Per iscriversi a un argomento, l'applicazione client chiama la seguente funzione:

Listing 2.4: FCM topic

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

Per annullare l'iscrizione, l'applicazione client deve richiamare "unsubscribeFromTopic()" con il nome del topic dal quale disiscriversi.

2.5.2 Parametri Cloud Messaging

Oltre al destinatario è possibile definire anche la priorità 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 è offline
Clickaction	L'azione associata al click della notifica

Tabella 2.5: Cloud Messaging parametri

2.5.3 Priorità

Esistono due opzioni per assegnare la priorità di consegna ai messaggi: normale ed alta priorità. Il recapito di messaggi normali e di alta priorità funziona in questo modo:

- **Priorità normale**: I messaggi vengono inviati immediatamente quando l'app è in primo piano, quando invece il dispositivo è in modalità Doze o l'applicazione è in standby 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 il dispositivo sarà nuovamente online.

- **Alta priorità**: Il server Cloud Messaging tenta di inviare immediatamente il messaggio ad alta priorità, consentendo al servizio, attraverso l'SDK di cambiare lo stato del dispositivo e di eseguire alcune elaborazioni limitate (compreso un accesso alla rete molto limitato).

2.6 FirebaseUI

FirebaseUI è un insieme di librerie open-source[4] disponibile per diverse piattaforme, che consentono di semplificare lo sviluppo di un applicazione che utilizza Firebase.
La libreria offre una versione semplificata per gestione dell'autenticazione, fornendo metodi che si integrano con i più comuni social, migliorando la comunicazione tra le View dell'interfaccia utente e il database, e facilita inoltre il collegamento e le richieste con il servizio Firebase Storage.
FirebaseUI dispone di moduli separati per utilizzare le varia librerie dedicate ai servizi:

- FirebaseUI Auth

- FirebaseUI Firestore

[4]https://firebaseopensource.com/projects/firebase/firebaseui-android/

- FirebaseUI Database
- FirebaseUI Storage

FirebaseUI-Auth cerca di gestire tutte le possibili casistiche che si possono riscontrare durante il login e la registrazione di un nuovo utente. La libreria FirebaseUI-Auth offre una integrazione con i social login più diffusi (Google, Facebook, Twitter) e una buona integrazione con Smart Lock per memorizzare e recuperare le credenziali, consentendo l' accesso automatico e il single-tap sign-in, gestendo anche casi d'uso più 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.
FirebaseUI-Firestore semplifica la comunicazione e l'interazione dei dati fra Cloud Firestore e l'interfaccia utente dell' applicazione, fornendo un adapter personalizzato (FirestoreRecyclerAdapter) che consente la manipolazione e la sincronizzazione automatica dei dati, senza dover scrivere codice ripetitivo per ogni adapter che si interfaccia con il database.

Capitolo 3

Architettura

L'applicazione è stata realizzata per la piattaforma mobile Android utilizzando il linguaggio di programmazione Kotlin, e altre librerie open-source. Lo sviluppo della parte client si basa sul pattern Model View Presenter (MVP) e l'organizzazone dei file segue le classiche direttive di Android, differenziando quindi Activity, Fragment, Adapter e Servizi.
Il client quando richiede dei dati al database memorizza la risposta all'interno delle cache, in modo da consentire all'utente di interagire con l'applicazione anche in modalità offline, senza quindi necessitare di una connessione ad internet. Quando l'utente riacquisterà la connessione, tutti i cambiamenti apportati offline verranno inviati al database FireStore che si occuperà di risolvere eventuali conflitti, ed aggiornare il database e le cache del client attraverso l'SDK.
La parte server invece è stata realizzata utilizzando come BaaS Firebase e i suoi servizi per la gestone del database, autenticazione, notifiche e lo storage. Ogni servizio offerto da Firebase interagisce in maniera diretta o indiretta con tutti gli altri servizi, facilitando la gestisce dell'autenticazione, del database e dello storage online.

3.1 Architettura Firebase

La gestione del backend è stata realizzata utilizzando la piattaforma Firebase e i suoi servizi. I servizi utilizzati per la gestione del backend sono:

1. **Auth**: Servizio per gestire l'autenticazione degli utenti
2. **Firestore**: Database real-time per la memorizzazione di tutti i dati utilizzati dall'applicazione.
3. **Storage**: Spazio di archiviazione utilizzato per salvare gli avatar degli utenti e le immagini dei gruppi.
4. **Cloud Functions**: Servizio utilizzato per monitorare i cambiamenti all'interno del database Firestore.
5. **Cloud Messaging**: Servizio utilizzato per gestire ed inviare notifiche ai dispositivi.

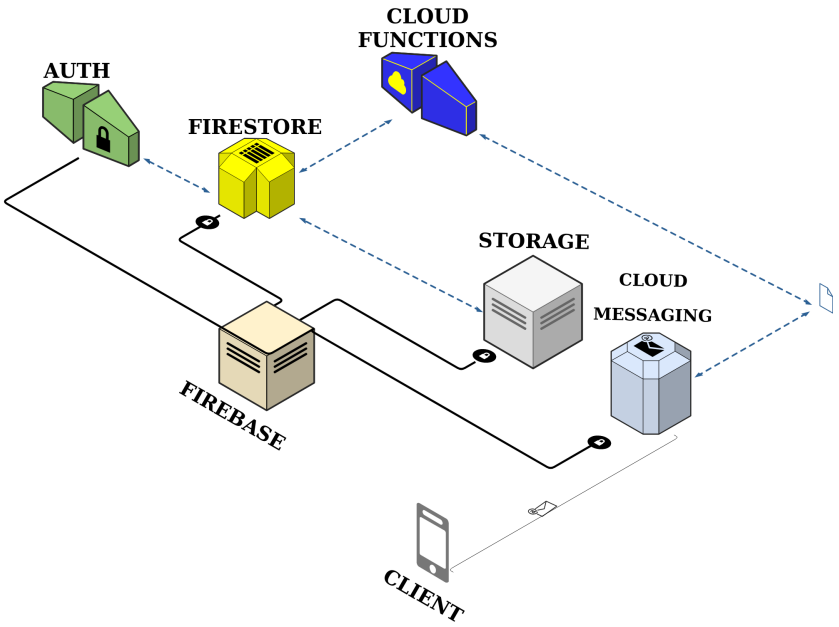

Figura 3.1: Architettura del server

3.1.1 Autenticazione

I client connessi a Firebase, hanno la possibilità di registrarsi attraverso email e social login (Google,Faceook,Twitter). Una volta effettuata la registrazione, FirebaseAuth assegnerà un identificativo univoco al nuovo client, memorizzando nei suoi server le informazioni basilari, quali: ID, nome, data di creazione, ultimo accesso, email e provider (Email, Google, Faceook, Twitter).

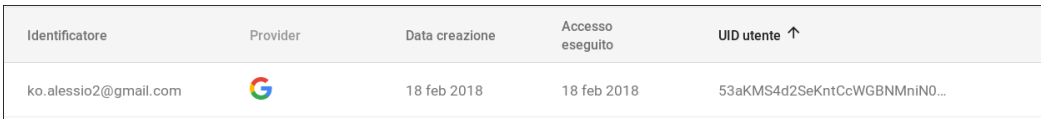

Figura 3.2: Schermata del pannello di controllo di FirebaseAuth User

Le informazioni salvate all'interno del servizio Firebase Auth sono differenti da quelle del database Firestore, di conseguenza per salvare ulteriori informazioni riguardanti l'utente, ogni volta che un client si registra con FirebaseAuth, verrà creato anche il suo rispettivo record all'interno del database Firestore.
Le informazioni aggiuntive salvate all'interno del database sono le seguenti:

- **ID:** identificativo univoco assegnato dal servizio FirebaseAuth.
- **Name:** nome dell'utente, inserito in fase di registrazione.
- **Email:** email univoca di registrazione dell'utente.
- **PhotoUrl:** avatar opzionale dell'utente salvato su Firebase Storage.
- **GroupID:** gruppo di appartenenza univoco
- **TokenFCM:** stringa utilizzata per interagire con il servizio FCM.

3.1.2 Firestore Database

La tipologia di database utilizzato è "Document Based", un database orientato agli oggetti dove i dati, a differenza dei database relazionali non vengono memorizzati in tabelle con campi uniformi per ogni record, ma ogni record è memorizzato sottoforma di documento che possiede determinate caratteristiche e può anche cambiare struttura nel tempo. Ogni documento appartiene ad una collezione, ed è memorizzato in un file JSON.
Il database dell'applicazione è organizzato in due sole collezioni principali, la collezione degli utenti e la collezione dei gruppi.
La struttura di un utente è la seguente:

Dato	**Tipo**	**Descrizione**
ID	Int	Identificativo univoco dell'utente
Name	String	Nome dell'utente, inserito in fase di registrazione
Email	String	Email univoca di registrazione dell'utente
PhotoUrl	String	URL dell'mmagine opzionale dell'utente
GroupID	Int	Gruppo di appartenenza univoco
TokenFCM	String	Stringa utilizzata per interagire con il servizio FCM

Tabella 3.1: Strutura collezione User

L'identificativo ID assegnato all'utente è lo stesso dell'identificativo assegnato da Firebase Auth in modo da avere una diretta associazione tra il servizio Firestore e Firebase Auth.
Il nome, l'email e l'avatar dell'utente vengono settati attraverso le informazioni recuperate quando si esegue la registrazione tramite social login o tramite email, successivamente viene viene settato il groupID, nel momento in cui l'utente seleziona il gruppo a cui partecipare oppure sceglie di crearne uno nuovo.
Infine il TokenFCM è generato dall'SDK del servizio Cloud Messaging.

L'altra collezione principale presente nel database, è la collezione che contiene i gruppi:

Dato	Tipo	Descrizione
ID	Int	Identificativo univoco del gruppo
Name	String	Nome del gruppo
Date	Date	Data di creazione del gruppo
TokenFCM	String	Token utilizzato per inviare messaggi al gruppo
Users	Map	Oggeto che contiene la lista degli utenti del gruppo
Chat	Collection	Riferimento alla collezione Chat
Todolist	Collection	Riferimento alla collezione Todolist
Expense	Collection	Riferimento alla collezione Expense

Tabella 3.2: Strutture collezione Group

L'identificativo del gruppo, la data di creazione e il nome e gli utenti appartenenti al gruppo, vengono settati al momento della creazione del gruppo, in particolare subito dopo la registrazione viene generato il "tokenFCM" del gruppo facendo richiesta al server FCM indicando i token di ogni dispositivo appartenente al gruppo. Quando un utente entra all'interno di un gruppo, viene fatta richiesta al server FCM per aggiungere il token del nuovo dispositivo al gruppo, inoltre la stringa rappresentante il tokenFCM del gruppo, pur subendo cambiamenti interni, in base ai dispositivi associati ad esso, rimane invariato.
All'interno della collezione "Group" ci sono quattro subcollezioni che rappresentano le quattro funzionalità principali dell'applicazione:

- **TodoList:** Contiene le mansioni del gruppo.
- **Expense:** Contiene le spese del gruppo.
- **Event:** Contiene gli eventi del gruppo.
- **Chat:** Contiene i messaggi del gruppo.

All'interno della collezione "Todolist" sono presenti le mansioni da completare condivise nel gruppo, ogni mansione ha la seguente struttura:

Dato	**Tipo**	**Descrizione**
ID	Int	Identificativo univoco della mansione
Name	String	Nome della mansione
Date	Date	Data di creazione della mansione
Description	String	Descrizione della mansione
Priority	Int	Priorità della mansione indicata con un numero da 1 a 3
Members	Map	Oggetto che contiene la lista degli utenti a cui è rivolta la mansione
Completed_User	String	Riferimento contenente l'UID dell'utente ha completato la mansione
CreatedBy	String	Riferimento contenente l'UID dell'utente ha creato la mansione
Status	Boolean	Booleano che indica lo stato della mansione

Tabella 3.3: Struttura collezione Todolist

L'identificativo di ogni mansione è assegnato dal Database Firestore, la priorità invece viene indicata sottoforma di intero, più alto è il valore dell'interno più la priorità sarà alta, e può variare da un minimo di 1 ad un massimo di 3.
Il campo "status" indica lo stato della mansione se impostato a "false", significa che la mansione deve essere completata, altrimenti se impostato a "true" significa che la mansione è stata completata, questo campo viene utilizzato dalle query per dividere le mansioni completate da quelle non completate.
Gli identificativi degli utenti che possono visualizzare la mansione da svolgere sono contenuti all'interno dal campo "members", quando un utente completa una mansione, il suo identificativo viene inserito nel campo Completed_User, e lo stato della mansione viene impostato a "true".

La seconda subcollezione contiene le spese del gruppo, la sua struttura è la seguente:

Dato	Tipo	Descrizione
ID	Int	Identificativo univoco della spesa
Name	String	Nome della spesa
Date	Date	Data di scadenza della spesa
Description	String	Descrizione della spesa
Tipology	String	Tipologia della spesa
Members	Map	Oggetto che contiene la lista degli utenti a cui è rivolta la spesa
Amount	Double	Valore globale che indica l'ammontare totale della spesa effettuata
CreatedBy	String	Riferimento contenente l'UID dell'utente ha creato la spesa
Status	Boolean	Booleano che indica lo stato della spesa

Tabella 3.4: Struttura collezione Spesa

L'ammontare della spesa indicata nel database è totale, quando la spesa dovrà essere divisa tra tutti gli utenti all'interno dell'oggetto "members", il client dovrà dividere l'ammontare totale per il numero di partecipanti alla spesa in modo da ricavare la quota parziale per ciascun utente.
Lo stato della spesa è globale, e può assumere valore vero o falso, se tutti gli utenti hanno pagato la loro quota, il valore di "status" verrà settato a "true" ovvero indicherà che la spesa è stata pagata da tutti.
La struttura dell'oggetto "members" è una Map chiave-valore, come chiave viene utilizzato l'identificativo UID dell'utente, mentre come valore un booleano che identifica lo stato della quota dell'utente.
Quando un membro del gruppo paga la sua quota il client effettua un controllo per vedere se la spesa è stata pagata da tutti o solo dall'utente.
Se la spesa è stata pagata solo dall'utente verrà impostato come valore della

chiave dell'utente all'interno di "members" il valore "true" indicando quindi che l'utente ha pagato la spesa, altrimenti il valore della sua chiave sarà "false".
La tipologia è una stringa che permette di indicare la categoria a cui appartiene la spesa, e viene gestita lato client, con stringhe di valori predefiniti.

La struttura di un documento riguardante un evento invece è il seguente:

Dato	**Tipo**	**Descrizione**
ID	Int	Identificativo univoco dell'evento
Name	String	Nome dell'evento
Description	String	Descrizione dell'evento
Date	Date	Data di creazione dell'evento
RecurringType	String	Stringa che indica il tipo di ricorrenza (Null, Dayly, Weekly, Monthly, Yearly)
Members	Map	Oggetto che contiene la lista degli utenti a cui è rivolto l'evento
CreatedBy	String	Riferimento contenente l'UID dell'utente ha creato l'evento

Tabella 3.5: Struttura collezione Event

Il tipo di ricorrenza di un evento può assumere anche il valore "Null", in questo caso l'evento non sarà definito ricorrente, ma avrà una sola occorrenza stabilita dalla data inserita dall'utente.
Il campo "members" contiene invece gli utenti a cui è rivolto l'evento.

L'ultima collezione utilizzata rappresenta i messaggi inviati all'interno della chat:

Dato	Tipo	Descrizione
ID	Int	Identificativo univoco del messaggio
Message	String	Stringa che rappresenta il contenuto del messaggio da inviare hline
Timestamp	Date	Data che indica il momento in cui è stato inviato il messaggio
SenderUID	String	Riferimento contenente l'UID dell'utente ha inviato il messaggio

Tabella 3.6: Struttura collezione Chat

La struttura è molto semplice e contiene il corpo del messaggio e la data in cui esso è stato inviato. É presenti anche il campo "SenderUID" che indica l'UID dell'utente che ha inviato il messaggio, tutti gli altri membri appartenenti al gruppo saranno automaticamente impostati come destinatari, non bisogna quindi indicare i membri del gruppo poichè la collezione chat essendo una subcollezione di groups, conosce già i componenti del gruppo. Il valore senderUID viene anche utilizzato per impedire al server FCM di inviare una notifica se l'UID dell'utente che ha inviato il messaggio è lo stesso del nuovo messaggio inserito nel database, da un qualsiasi utente del gruppo.

3.1.3 Storage

Gli utenti hanno la possibilità di salvare e personalizzare i propri avatar e gestire l'immagine del gruppo a cui appartengono, la memorizzazione di queste immagini viene fornita dal servizio Firebase Storage. Il servizio offre la possibilità di caricare online qualsiasi tipo di file sia attraverso l'SDK sia attraverso il pannello di controllo di Firebase.
Ogni file presente sullo storage contiene il nome, la dimensione, il tipo e l'ultima modifica del file, oltre a queste informazioni sono presenti due link: il riferimento del file sullo storage e un URL pubblico che consente il download del file.

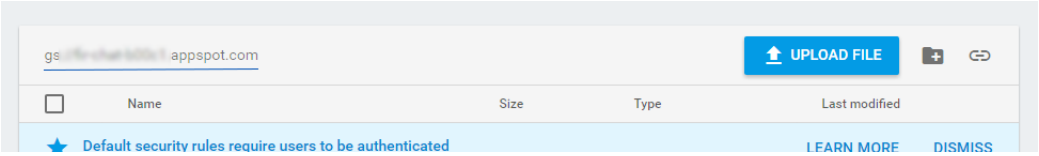

Figura 3.3: Pannello di controllo del servizio Firebase Storage

Il riferimento viene utilizzato dall'SDK per avere un identificativo del file all'interno dello storage, questo identificativo viene utilizzato dai client per scaricare il file applicando restrizioni di visibilità, il link pubblico invece permette a chi lo possiede di visualizzare e scaricare il file (per motivi di sicurezza questo link può essere rigenerato).
L'organizzazione dei file presente all'interno dello storage è gerarchica, per separare le immagini degli utenti con le immagini dei gruppi, ogni immagine viene salvata nella rispettiva cartella. Gli avatar vengono salvati all'interno della cartella "UserAvatar" e come nome identificativo hanno l'ID dell'utente assegnato da Firestore, le immagini dei gruppi invece vengono salvate nella cartella "GroupAvatar" e utilizzano come identificativo l'ID del gruppo. Assegnando quindi come identificativo l'id dell'utente di Firestore, e l'ID del gruppo è possibile fare riferimento a tutte le immagini sia attraverso il pannello di controllo di Firebase, sia utilizzando l'SDK.

3.1.4 Cloud Functions

Le funzioni di Cloud Functions vengono utilizzate per inviare notifiche ai dispositivi in base a eventi scaturiti all'interno del database.

- **onGroupUserAdded:** Funzione che notifica i membri di un gruppo quando viene aggiunto un nuovo utente.
- **onGroupExpenseAdded:** Funzione che notifica i membri di un gruppo quando viene aggiunta una nuova spesa.
- **onGroupTodoCompleted:** Funzione che notifica i membri di un gruppo quando viene completata una mansione.
- **onGroupEventAdded:** Funzione che notifica i membri di un gruppo quando viene aggiunto un nuovo evento.
- **onGroupTodoAdded:** Funzione che notifica i membri di un gruppo quando viene aggiunta una nuova mansione.
- **onGroupMessageAdded:** Funzione che notifica i membri di un gruppo quando viene inviato un messaggio nella chat.

3.1.5 Cloud Messaging

Le notifiche vengono gestite dal server Cloud Messaging, che si occupa della memorizzazione, invio e ricezioni delle notifiche fra client e server.
La ricezione dei messaggi da parte di Cloud Messaging è possibile solo se si utilizza l'apposito SDK e si ottiene un token, necessario per inviare un messaggio ad un dispositivo specifico, il token di registrazione viene generato automaticamente dall'SDK all'avvio dell'applicazione.
L'SDK consente di visualizzare i messaggi ricevuti dal servizio e gestire la generazione e il cambiamento del token. Il token di registrazione può cambiare solo quando l'applicazione viene reinstallata, disinstallata o quando vengono eliminate le cache e i dati dal dispositivo.

I passi necessari per connette e ricevere i messaggi dal server FCM sono i seguenti:

- Client attraverso una richiesta al server FCM, richiede un token
- Il server FCM, se la richiesta è valida, restituisce un token di registrazione
- Il client invia al database Firestore il token di registrazione
- il client da questo momento, con il token, può inviare o ricevere messaggi connettendosi al server FCM

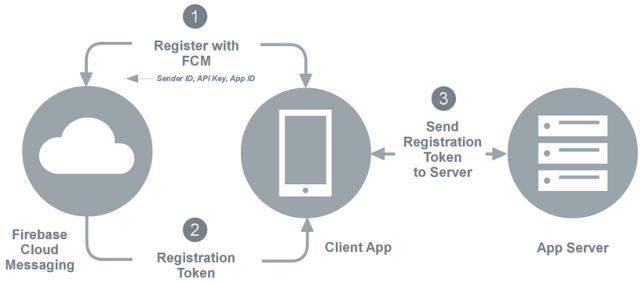

Figura 3.4: Architettura Token FCM

3.1.6 Firestore Rules

Le regole di sicurezza Cloud Firestore Firestore Security Rules, consentono di definire norme di sicurezza per il controllo dell'accesso e la convalida dei dati.
La struttura del database è stata progettata per facilitare la scrittura delle regole di accesso, infatti esistono solo due collezioni principali, quella degli utenti e quella dei gruppi. Le regole di accesso dei dati sono state definite in modo tale che ogni membro del gruppo possa vedere solo i dati appartenenti al suo gruppo, in questo modo ogni mansione da svolgere, spesa, evento e messaggi all'interno della chat, sono protetti attraverso un controllo client e server.

3.2 Model View Presenter

La parte client è stata realizzata utilizzando il pattern MVP.
Il MVP (Model View Presenter) è un pattern architetturale utilizzato per l'organizzazione strutturale di un progetto, in modo da trarne vantaggio in termini di prestazioni, leggibilità e modularità del codice.
La sua caratteristica principale è quella di separare il livello di presentazione dalla logica, in modo che tutto ciò che riguarda l'interazione dell'utente sia separato da come vengono rappresentati i dati.
Il pattern MVP deriva dal pattern MVC (Model View Controller), che ha 3 concetti base, che lo definiscono:

1. **Model:** Il modello dei dati da visualizzare.
2. **View:** L'interfaccia utente che visualizza i dati.
3. **Controller:** Controlla l'interazione tra Model e View.

La principale differenza tra i due pattern risiede nel Presenter che gestisce la logica tra la View e il Model. Come il pattern MVC, anche il pattern MVP permette di rendere le View indipendenti dalla gestione dei dati, dividendo la logica dell' applicazione in tre livelli distinti, livelli che possono essere testati separatamente.

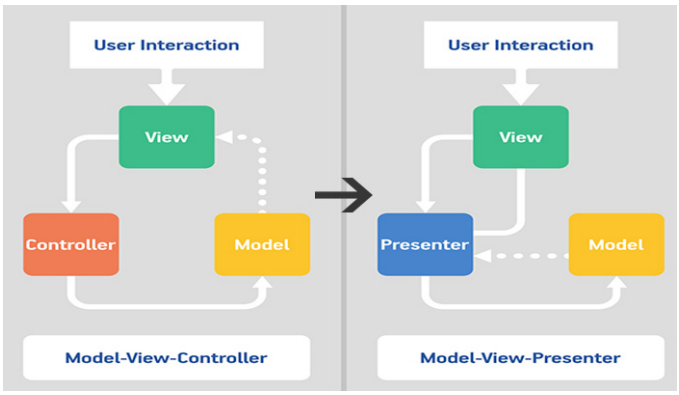

Figura 3.5: Confrontro tra il pattern MVC e MVP

La possibilità di poter testare i livelli separatamente è una delle caratteristiche del MVP, la sua struttur è la seguente:

1. **Model:** Il modello è un'interfaccia che definisce i dati da visualizzare.
2. **View:** La View è un'interfaccia passiva che visualizza i dati (il modello) e instrada i comandi utente (eventi) al Presenter per agire su tali dati.
3. **Presenter:** Il Presenter agisce sul modello e sulla vista. Recupera i dati dai repository (il modello) e li formatta per la visualizzazione nella vista.

3.2.1 Model

Il Model è un'interfaccia dedicata all'acceso dei dati di un'applicazione, si occupa quindi di fornire un'astrazione del modello dei dati presenti nel database.
Il Model oltre a contenere la struttura dei dati da visualizzare si occupa anche di fornire una buona astrazione del database, modificando, aggiungendo e separando alcuni dei dati, in modo da renderene l'accesso e la visualizzazione più semplice per gli altri due componenti del pattern (View, Presenter).
Un esempio potrebbe essere il seguente: Il database contiene una tabella con due tipi di dato:

1. **Nome**: String
2. **DataDiNascita**: Date

Quando il programma riceverà i dati dal database in un qualsiasi formato(Map, Json, Array) il Model selezionerà i dati in base alla definizione data dal programmatore trasformando il risultato del database in un oggetto. Questo oggetto oltre a conservare le due informazioni ricevute dal database (Nome, DataDiNascita) potrà contenere anche informazioni aggiuntive inserite dal Model per facilitare l'uso e la manipolazione degli altri due componenti. In questo caso il modello potrebbe creare il nuovo campo "età" facendo

una semplice sottrazione fra due date, quella attuale e la data di nascita dell'utente.

1. **Nome**: String
2. **DataDiNascita**: Date
3. **Età**: Int

3.2.2 View

La View è un'interfaccia che definisce cosa deve implementare il Presenter, affinchè possa interagire con l'interfaccia utente.
La View interagisce con il Presenter per visualizzare i dati e notifica al Presenter le azioni che compie l'utente nell'interfaccia.
La View può essere implementata da un Activity, un Fragment, o un widget Android, che contengono ProgressBar, TextView, RecyclerView o altri elementi che necessitano di essere aggiornati in base a qualche azione dell'utente o cambiamento nel server.
Gli aggiornamenti della View possono essere gestiti in due diversi modi:

- **Passive View**
- **Supervising Controller**

Nella Passive View, il Presenter aggiorna la vista per applicare i cambiamenti del modello, in questa modalità l'interazione con il Model è gestita esclusivamente dal Presenter, la vista quindi ha un comportamento "passivo" e non è a conoscenza dei cambiamenti nel Model.
Ad esempio, se si dispone di un modulo username/password e di un pulsante "Invia", non si scrive la logica di validazione all' interno della View ma all'interno del Presenter. La View infatti dovrebbe solo contenere il nome utente e la password e inviarli al Presenter.
Nel Supervising Controller, la vista interagisce direttamente con il Model per eseguire semplici operazioni di binding dei dati, senza l'intervento del Presenter.

Il Presenter aggiorna il Model, e gestisce cambiamenti sulla View solo nei casi più complessi, ad esempio l'aggiornamento del colore di un bordo in base alle modifiche effettuate su un dato del Model, poichè la modifica non prevede una corrispondenza diretta tra la View e il Model. Entrambe le modalità facilitano il testing in Android poichè le loro implementazioni riducono al minimo la quantità di logica implementata nella View.

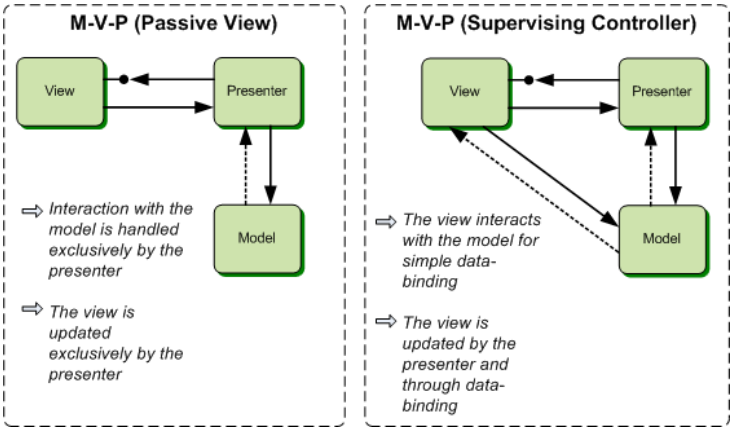

Figura 3.6: MVP View types

3.2.3 Presenter

Il Presenter è il mediatore tra il Model e la View e si occupa di recuperare i dati dal Model, formattarli e passarli alla View, ma a differenza del pattern MVC, decide anche cosa succede quando si interagisce con la View reagendo alle interazioni dell'utente.
Il Presenter per facilitare il testing deve cercare di non dipendere minimamente da Android, ma contenere solo metodi e dipendente Java, senza l'utilizzo del "Context" ad esempio.
Come detto in precedenza il Presenter deve dipendere dall' interfaccia View e non direttamente dall'Activity o Fragment, in questo modo si tengono separati il Presenter e l'Activity rispettando la D dei principi SOLID: "Dipendi dalle astensioni. Non dipendere dalle concrezioni", questo permetterà inoltre di scrivere i test per il Presenter senza l'utilizzo di un emulatore Android.

3.3 Reactive Programming

La programmazione reattiva o Reactive Programming è un paradigma di programmazione che monitora e gestisce flussi di dati statici o dinamici e la propagazione dei flussi nel tempo, ciò significa che attraverso dei costrutti messi a disposizione da librerie che implementano il concetto di programmazione reattiva sarà possibile sviluppare applicazioni basate sugli eventi.
Il concetto fondamentale utilizzato in questo paradigma è la possibilità di creare flussi di qualsiasi tipo sia facendo riferimento al click dell'utente sull'interfaccia grafica sia a variabili, input utente, proprietà, cache, strutture dati e richieste HTTP.
Un flusso è una sequenza di eventi ordinati nel tempo che può emettere tre risposte differenti: un valore (di qualche tipo), un errore o un segnale per indicare che il flusso è terminato. Questi tre eventi vengono gestiti in modo asincrono, definendo una funzione che verrà eseguita quando verrà emesso un valore, un'altra funzione quando verà emesso un errore e un'altra funzione quando verrà completato il flusso.
I due componenti principali sono quindi "Observables" ovvero colui che emette un flusso di dati e "Observers", ovvero colui che monitora il flusso e rimane in ascolto di nuovi valori.

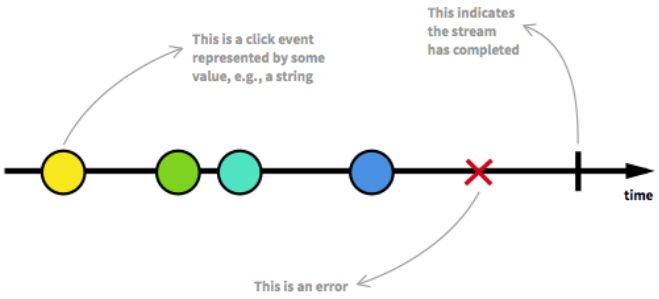

Figura 3.7: Reactive Programming Esempio 1

- **onNext**: Metodo che viene richiamato quando un Observable emette un nuovo elemento nel corso del flusso.
- **OnComplete:** Metodo che indica che il flusso è giunto al termine e quindi non verrà emesso nessun nuovo elemento.
- **onError** Metodo che viene chiamato in presenza di errori all'interno del flusso, questo metodo interrompe il normale percorso del flusso.

Nello sviluppo dell'applicazione è stata utilizzata la libreria RxJava per utilizzare il paradigma di programmazione reattiva, e sfruttare la logica e i benefici della programmazione asincrona.
In particolare la libreria è stata utilizzata per la gestione dei flussi di dati durante le richieste effettuate al database Firestore.

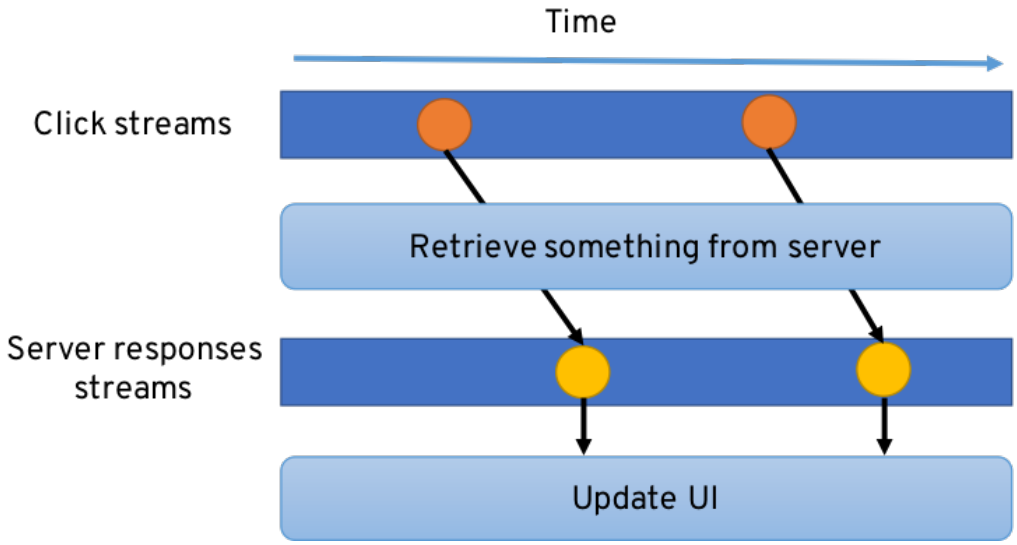

Figura 3.8: Reactive Programming Esempio 2

Quando un utente si interfacciava con la pagina di una funzionalità veniva effettuata la richiesta al server e aggiornata l'interfaccia con i nuovi dati.

Capitolo 4

Sviluppo Client

Inizialmente per sviluppare l'applicazione, è stato preso in considerazione il linguaggio Java, successivamente dopo aver testato le funzionalità di Java fu preso in considerazione Kotlin come linguaggio di programmazione per Android, in alternativa a Java, in modo tale da ricavare un confronto fra i due linguaggi.
In questo capitolo vengono illustrate e commentate le scelte implementative, l'organizzazione e la struttura utilizzata per organizzare i file in base al loro utilizzo. L'applicazione è stata realizzata e sviluppata seguendo il pattern MVP, alcuni elementi di programmazione reattiva, cercando di rendere lo sviluppo di tutte le componenti dell'applicazione modulari e facili da gestire.

4.1 Struttura

Il Client è stato strutturato in modo da contenere in packages differenti i componenti principali dell'applicazione.

- **Pages:** Contiene quattro packages corrispondenti alle 4 funzionalità dell'app.
- **Activities:** Contiene le Actvity utilizzate dall'applicazione.
- **Models:** Contiene i modelli, utili per il pattern MVP
- **Repositories:** Contiene classi che facilitano la gestione e le richieste con il database.
- **Services:** Contiene due servizi per la gestione delle notifiche.
- **Utils:** Contiene classi utili per la gestione della cache, PreferenceShared e componenti view personalizzati.

4.1.1 Pages

L'implementazione delle funzionalità principali dell'applicazione sono contenute all'interno del package "Pages", ogni pagina rappresentante una funzionalità rispetta una logica comune per gestire l'interazione con l'utente e l'aggiornamento dei dati.
La logica utilizzata si basa sul pattern MVP, di conseguenza ogni volta che bisognerà mostrare dei dati, verranno implementati i seguenti componenti:

- **Adapter:** Estensione della classe "RecyclerView.Adapter" che contiene la lista elementi da visualizzare
- **Presenter:** Componente che ha il compito di richiedere al Repository i dati da visualizzare
- **View:** Interfaccia grafica della pagina con cui l'utente può interagire

- **Model:** Modello astratto da rappresentare nella View
- **Repository:** Classe che ha il compito di inviare richieste al Database, o di restituire le query necessarie per la richiesta al database

Le quattro pagine verranno illustrate e commentate nel dettaglio nell'apposita sezione del capitolo quattro.

4.1.2 Activities

Le Activity utilizzate dall'applicazione sono sette:

- **BaseActivity:** gestisce il funzionamento dei menù (Menu delle impostazioni, Menù delle funzionalità), e si occupa di mostrare le quattro pagine principali
- **IntroActivity:** mostra delle pagine di introduzione all'applicazione, priama di poter effetuare, l'accesso o la registrazione
- **ProfileActivity:** mostra le informazioni personali dell'utente, e i campi che può modificare
- **GroupActivity:** mostra le informazioni del gruppo
- **JoinGroupActivity:** gestisce l'accesso ad un nuovo gruppo
- **NewGroupActivity:** gestisce la creazione di un nuovo gruppo
- **AuthActivity:** gestisce la logica dell'accesso e della registrazione

Gestione gruppo

La creazione e l'accesso ad un gruppo vengono gestite dalle activity "JoinGroupActivity" e "NewGroupActivity".
Effettuata la registrazione o l'accesso all'applicazione, l'utente avrà la possibilità di entrare a far parte di un gruppo o crearne uno nuovo, con l'eccezione che ogni utente può fare parte di un solo gruppo.

L'utente che sceglie di entrare a far parte di un nuovo gruppo deve aver precedentemente ricevuto il codice invito da un membro appartenente ad un gruppo esistente. Una volta ricevuto il codice di invito, il nuovo utente dovrà inserire il codice e confermare di entrare a far parte del gruppo, se conferma verranno aggiornati i membri del gruppo e il gruppo di appartenenza dell'utente e gli altri membri del gruppo invece riceveranno una notifica.
L'utente che sceglie invece di creare un nuovo gruppo deve indicare il nome del gruppo e un immagine opzionale da associarci, in seguito alla creazione potrà invitare altri utenti ad iscriversi al suo gruppo tramite un codice invito.

Accesso e Registrazione

Al primo avvio dell'applicazione, verranno mostrare delle pagine scorrevoli "IntroActivity" che illustreranno le caratteristiche principali con il quale può interagire l'utente, successivamente dopo una breve introduzione verrà mostrata la pagina di login, che permette di effettuare la registrazione e l'accesso attraverso un solo pulsante, senza differenziare se un utente sia già registrato o meno.
Quando l'utente cliccherà sul pulsane "accedi", l'applicazione automaticamente controllerà se l'utente si era già precedentemente registrato o deve effettuare la registrazione.
Se l'utente richiede di registrarsi, l'interfaccia e la logica di registrazione vengono controllate dalla libreria FirebaseUI, l'accesso invece viene gestito manualmente all'interno dell'activity "AuthActivity".
Ogni utente è univoco e non può creare account diversi utilizzando la stessa email, inoltre utilizzando la libreria FirebaseUI si hanno a disposizione l'integrazione con SmartLock, e l'account linking. L'account linking consiste nel collegare account che utilizzano la stessa email, se si effettua, ad esempio l'accesso attraverso uno dei social supportati, e l'email di registrazione del social è già presente nei server di Firebase-Auth, verrà effettuato un collegamento degli account automatico (Account Linking), fra gli account che utilizzano la stessa email.

Quando un utente registrato, effettua l'accesso, viene controllato se è presente il record all'interno del database Firestore, in caso contrario viene fatta richiesta di aggiungere il nuovo utente al database Firestore, utilizzando come ID, l'identificativo fornito dal servizio Firebase-Auth. Una volta effettuato l'accesso per evitare ulteriori richieste al Database vengono anche salvate le informazioni dell'utente e gli identificativi dei membri appartenenti al gruppo nelle "Shared Preferences" di Android.
L'accesso e la registrazione possono essere effettuati utilizzando i social più diffusi o attraverso la semplice registrazione via email e password.
I social disponibili sono: Google, Facebook, Twitter

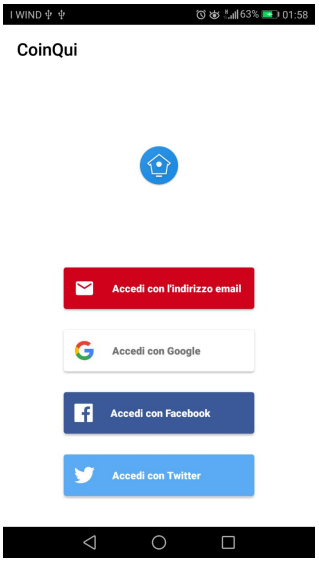

Figura 4.1: Schermata applicazione Login

Se l'utente sceglierà di registrarsi attraverso l'utilizzo di un email, gli verranno richiesti l'email di registrazione, un nominativo (Nome,Cognome) e una password, per effettuare il login invece verranno richiesti solo l'email e la password.
Alternativamente se l'utente seleziona il metodo di registrazione attraverso un social, comparirà a schermo una finestra che chiederà all'utente registrato al social di consentire l'utilizzo dell'email e del nome dell'utente da parte dell'applicazione, una volta ricevuta l'autorizzazione, le volte successive verrà

effettuato un login automatico senza richiedere permessi aggiuntivi.
Un utente che ha dimenticato la propria password può richiederne una nuova inserendo l'email di registrazione, in seguito dopo pochi secondi riceverà via email un avviso per reimpostare la password e un link che permetterà di reimpostare la password.

Menù

L'applicazione offre due menù differenti, il menù delle funzionalità e il menù delle impostazioni, gestite dall'activity "BaseActivity".
Il menù delle funzionalità si trova nella parte inferiore dello schermo, mentre per accedere al menù delle impostazioni, bisognerà cliccare la relativa icona del menù presente nella toolbar.

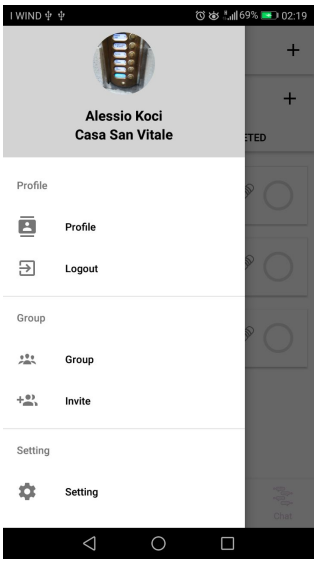

Figura 4.2: Schermata applicazione Menù

Interagendo con il menù delle impostazioni laterale si potranno gestire le informazioni personali dell'utente e gestire le informazioni del suo gruppo.
Nella pagina riguardante il profilo "ProfileActivity" dell'utente sarà possibile

visualizzare le informazioni personali come il nome, l'email e il gruppo a cui esso appartiene, queste informazioni sono modificabili in qualsiasi momento. Nella pagina riguardante il gruppo "GroupActivity" invece sarà possibile visualizzare le informazioni principali riguardanti il gruppo a cui l'utente appartiene, come: il nome, l'immagine e gli utente che appartengono al gruppo. Le informazioni modificabili in questa pagina sono l'immagine del gruppo, il nome del gruppo e la possibilità di invitare altre persone ad unirsi al gruppo tramite invito, (verrà inviato all'utente un codice di invito che dovrà inserire al momento della registrazione).

4.1.3 Repository

All'interno del package Repositories sono presenti le classi necessarie per interagire con il database Firestore.
Sono presenti due tipi di classi:

- **Query:** Sono classi che contengono le query necessarie per interagire con il database, ogni classe conterrà le query relativa alle sue funzionalità, la funzione Todolist dell'applicazione ad esempio avrà una classe TodoListQuery.
- **Repository:** Sono classi che restituiscono un oggetto Single, Observable o Completable in base al tipo di richiesta effettuata. Ogni Repository richiamerà la relativa classe Query per ricevere l'istruzione necessaria per aggiungere modificare o richiedere dei dati al database.

Un esempio di una Repository è il seguente:

Listing 4.1: Esempio RepositoryAggiunta elemento Todolist

```
fun getTodoItems(): Single<List<TodoItem>> {
      return Single.create<List<TodoItem>> { emitter ->
          queryTodo.getTodoItems().addSnapshotListener({
              querySnapshot, exception ->
```

```
                if (exception != null)
                    emitter.onError(Throwable(exception))
                else {
                    emitter.onSuccess(TodoItem.mapping(querySnapshot))
                }
            })
        }
    }
```

Ogni Repository non conosce la query necessaria per effettuare la richiesta al database ma utilizzando la rispettiva classe che restituisce la query necessaria per richiedere al database la lista degli elementi all'interno della todolist.

4.1.4 Utils

Il package Utils contiene classi di supporto per azioni comuni che vengono svolte dai componenti dell'applicazione.
In particolare le principali sono:

- **PreferenceUtils:** Classe utilizzata per gestire le SharedPreferences.
- **ImageUtils:** Classe che interagendo con la libreria Glide, aggiunge funzionalità alla gestione delle immagini.
- **RxFirestore:** Estensioni effettuate su alcuni oggetti dell'SDK Firestore per implementare il pattern Observable.
- **UserSpinner:** Componente View che estende. AppCompatSpinner, creato per mostrare e selezionare graficamente gli utenti attraverso uno spinner.
- **FirestoreCMUtils:** Classe che gestire i Token utilizzati da Cloud Messaging.

La classe PreferenceUtils memorizza le informazioni riguardanti gli utenti apparteneti al gruppo, le informazioni personali dell'utente consentendo di

modificarne i valori e cancellare tutti i dati e le cache dell'applicazione.
. La classe ImageUtils, è stata realizzata per estendere alcune funzionalità della libreria Glide, utilizzata per visualizzare le immagini, in particolare è stato implementato un metodo per visualizzare un immagine attraverso un link URL. La libreria opportunamente modificata consentiva di scarichere l'immagine del link, memorizzarla nella cache e renderla visibile all'utente attraverso l'interfaccia.
Il file RxFirestore contiene le funzioni necessarie per implementare dei metodi aggiuntivi agli oggetti DocumentReference, CollectionReference messi a disposizione dall'SDK di Firebase per fare riferimento ad una collezione o documento all'interno del database. I metodi aggiuntivi consentono di applicare la programmazione reattiva alle chiamate eseguite per richiedere e modificare dati all'interno del database Firestore. UserSpinner è un widget creato appositamente per mostrare una finestra di dialogo contenente la lista del nome degli utenti appartenenti al gruppo, e una checkbox per ogni utente, offrendo quindi la possibilità di selezionare uno o più utenti durante la creazione di una nuova mansione, spesa o evento.
Infine la classe FirestoreCMUtils gestisce i token e i gruppi associati ad un token di Cloud Messaging, la classe in particolare viene utilizzata per generare un token necessario per connettersi ai server FCM, creare un nuovo gruppo di utenti indicando i token di ogni dispositivo, e per ultimo gestire l'aggiunta di un nuovo utente ad un gruppo di utenti preesistenti.

4.1.5 Modelli

I modelli sono stati realizzati utilizzando le data class offerte da Kotlin, in questo modo non è stato necessario implementare i metodi get e set come quando si devono creare dei modelli in Java. L'unica aggiunta effettuata è stata la creazione di due nuovi metodi chiamati "mapping" che permettono di convertire i tipi DocumentSnapshot e QuerySnapshot restituiti dal'SDK Firebase come risposta ad una richiesta effettuata al database Firestore in modelli utilizzabili come oggetti all'interno dell'applicazione, mappando quindi ogni

valore contenuto negli SnapShot all'interno dei singoli valori del modello.
I modelli vengono utilizzati nell'applicazione per rispettare il pattern MPV garantendo una buona interazione dei dati con l'interfaccia grafica, e soprattutto per avere una migliore gestione dei tipi durante la chiamata e la restituzione di valori associati ad una funzione.
Un esempio di un modello utilizzato è il seguente:

Listing 4.2: Aggiunta elemento Todolist

```
data class GroupItem(
        var id: String = "",
        var name: String? = null,
        var creation_date: Date = Date(),
        var logo: String? = null,
        var users: Map<String, Boolean> = emptyMap(),
        var tokenFCM: String? = null
)
```

Il seguente modello rappresenta un documento all'interno della collezione contenente i gruppi. Attraverso il costrutto data class, e inizializzando le variabili con la keyword "var" Kotlin in automatico in fase di compilazione aggiungerà i metodo necessari per settare e ricevere i valori dalla classe GroupItem.
L'id di ogni gruppo è impostato come stringa vuota, e verrà settato al momento dell'aggiunta di un nuovo gruppo, come il nome il logo e la lista degli utenti.
Il valore "data" è assegnato di default in base alla data di creazione del gruppo, mentre il tokenFCM è generato subito dopo la creazione del gruppo, attraverso il servizio FCM.

4.1.6 Servizi

I servizi utilizzati dall'applicazione sono stati creati per interagire con il servizio Cloud Messaging di Firebase.
I servizi sono due:

- **FirebaseMessagingService**
- **FirebaseInstanceIdService**

Il primo: FirebaseMessagingIdService gestisce i token necessari per utilizzare Cloud Messaging, in particolare gestisce l'aggiornamento del token del dispositivo, inviando una richiesta di aggiornamento al Database Firestore, qualora il token sia aggiornato o eliminato.
Il secondo servizio invece: FirebaseInstanceService gestisce i messaggi ricevuto dal server di Cloud Messaging, implementando il metodo "onMessageReceived" sarà possibile innanzitutto capire il tipo di messaggio ricevuto da FCM (messaggio di notifica o messaggio contenente dati), successivamente in base al tipo di messaggio ricevuto vengono effettuate modifiche o mostrate le adeguate notifiche.
Oltre ad aggiungere la libreria che si occupa di interagire con il servizio FCM, è stato necessario indicare all'interno del AndroidManifest del progetto, che le due classi create sono dei servizi:

Listing 4.3: Android Manifest - I servizi FCM

```
<service android:name=".Services.FirestoreEventFunctionService">
          <intent-filter>
              <action
                  android:name="com.google.firebase.MESSAGING_EVENT"
                  />
          </intent-filter>
      </service>
      <service
          android:name=".Services.FirestoreEventFunctionnstanceIDService">
          <intent-filter>
```

```
            <action
                android:name="com.google.firebase.INSTANCE_ID_EVENT"
                />
        </intent-filter>
    </service>
```

In particolare quando viene ricevuto il messaggio di aggiunta di un nuovo membro nel gruppo, oltre ad inviare la notifica per avvisare i dispositivi, vengono anche aggiornate le PreferenceShared che contiene localmente una copia delle informazioni riguardanti il gruppo senza dover contattare ripetutamente il database.

Listing 4.4: Metodo onMessageReceived

```
override fun onMessageReceived(remoteMessage: RemoteMessage ? ) {
...
  val msgData = remoteMessage.data.toProperties()
  val loggedUser = PreferenceUtils(context =
      applicationContext).getUserUID()
  if (remoteMessage.data.isNotEmpty()) {
   val msgData = remoteMessage.data.toProperties()
   val loggedUser = PreferenceUtils(context =
       applicationContext).getUserUID()
   if (msgData.getProperty("sender") != loggedUser) {
    when(msgData.getProperty("type")) {
      ...
     "todo" -> {
      sendNotification(messageTitle = R.String.new_todo_item,
          messageBody = msgData.getProperty("name"))
     }
     ...
    }
```

```
   }
  }
}
```

La funzione "onMessageReceived" viene richiamata quando il server FCM segnala al dispositivo la ricezione di un nuovo messaggio. Successivamente, verrà controllato se l'utente, che ha compiuto il cambiamento all'interno del database, aggiungendo o modificando un elemento, è lo stesso utente loggato all'interno dell'applicazione, in caso affermativo non verrà inviata nessuna notifica al dispositivo, in caso contrario verrà mostrata una notifica, avvisando l'utente del nuovo cambiamento all'interno dell'applicazione.

4.2 Funzionalità

L'applicazione aiuta la gestione di attività e problemi riscontrati durante una convivenza fra due o più persone, in ambito lavorativo o fra studenti fuori-sede.
Le funzionalità principali consentono ai membri di un gruppo di gestire una lista di mansioni comuni da svolgere, gestire e dividere le spese, organizzare eventi periodici e/o ricorrenti e confrontarsi utilizzando la chat di messaggistica istantanea. L'applicazione avendo funzionalità molto generiche lascia il completo utilizzo di esse all'utente finale, permettendogli di gestire le varie funzionalità come meglio crede. Un esempio potrebbe essere la gestione degli eventi: alcuni studenti fuori sede potrebbero creare eventi per organizzare i turni di pulizia all'interno della casa, assegnando eventi ricorrenti a coinquilini specifici, in ambito lavorativo invece, i membri del gruppo potrebbero utilizzare la funzione di gestione degli eventi per organizzarsi il lavoro o creare incontri aziendali.
L'utente dopo aver effettuato l'accesso potrà interagire con le funzionalità dell'applicazione selezionando l'icona della relativa funzionalità dal menù. La struttura e l'organizzazione dei file delle quattro funzionalità dell'app è la stessa, inoltre il pattern utilizzato per la gestione fra le varie componenti del

progetto per interagire con i dati e l'interfaccia utente è MVP (Model View Presenter).
Esiste una Activity principale chiamato BaseActivity che gestisce il funzionamento dei menù (Menu delle impostazioni, Menù delle funzionalità), e si occupa di mostrare le quattro pagine principali, realizzate estendendo la class Fragment.
I quattro Fragment sono quindi:

- **FragmentTodo:** Pagina per visualizzare e interfacciarsi con le mansioni del gruppo
- **FragmentSpese:** Pagina per visualizzare e interfacciarsi con le spese del gruppo
- **FragmentEventi:** Pagina per visualizzaere e gestire gli eventi del gruppo
- **FragmentChat:** Pagina per accedere alla chat di messaggistica istantanea del gruppo

I fragment vengono cambiati e gestiti dall'Activity principale, che in base all'interazione con il menu delle funzionalità, si interscambiano.

Listing 4.5: Aggiornamento fragment del BaseActivity

```
supportFragmentManager.beginTransaction().replace(R.id.activity_content,
    TodoFragment()).commit()
```

Quando si seleziona una della pagine, il controllo dell'interfaccia passa al relativo Fragment, che in base alla funzionalità mostrerà e permetterà di agire sull'interfaccia.

4.2.1 Todolist

Selezionando dal menù dell'applicazione l'icona della "Todolist", l'utente visualizzerà l'interfaccia dedicata per interagire con le funzionalità della todolist, quali: visualizzare le mansioni da svolgere, visualizzare le mansioni già svolte, aggiungere, modificare o eliminare una mansione.
L'interfaccia per visualizzare le mansioni è composta da due sezioni, la sezione delle mansioni da completare, in primo piano e le mansioni già completate in un'apposita sezione.
Le mansioni sono composte da un nome obbligatorio, una data di scadenza, una priorità ed i membri del gruppo a cui è rivolta la mansione.
Ogni utente visualizza la mansione comprensiva di nome e data, la priorità invece viene indicata con un bordo colorato in base all'importanza della mansione, le altre informazioni possono essere viste, cliccando sulla mansione.

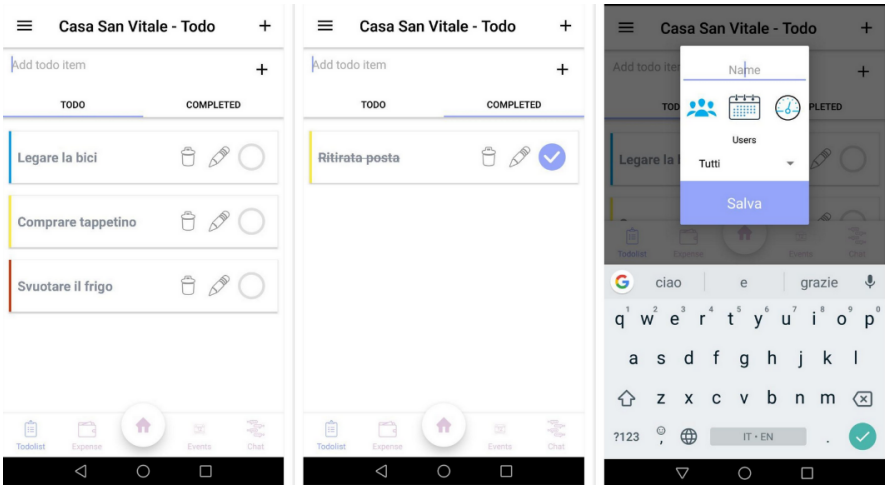

Figura 4.3: Schermata applicazione Todolist

Gli utenti possono vedere sia le mansioni create dagli altri membri del gruppo sia le loro mansioni, in base alle restrizioni di visibilità assegnate durante la creazione, l'unica limitazione imposta riguarda le funzionalità di modifica ed eliminazione, che sono consentite solamente all'utente che ha creato la mansione, gli altri utenti invece potranno completare la mansione

marcandola.
Le mansioni che vengono marcate e completate vengono spostate automaticamente nell'elenco delle mansioni completate e ogni utente avrà la possibilità di portare nuovamente una mansione non completata nella sezione delle mansioni da completare, senza dover aggiungerne un'ulteriore. Quando si sposta una mansione dalla sezione "mansioni completate" alla sezione "mansioni da completare" il nome, la visibilità e la priorità della mansione rimarranno inalterate, mentre la data di scadenza sarà impostata al giorno in cui è avvenuto il cambiamento.
L'aggiunta di una nuova mansione viene effettuata attraverso due modalità differenti: la prima rapida, la seconda personalizzata.
La modalità rapida si trova nella parte superiore dello schermo, sottostante alla toolbar, in questa modalità l'utente può aggiungere un nuovo elemento indicando solamente il nome ed in automatico l'applicazione setterà i campi opzionali, impostando la data di scadenza alla data in cui è stato aggiunto l'elemento, la priorità di medio livello e la visibilità a tutti i membri del gruppo.
La modalità personalizzata invece permette di inserire tutte le informazioni possibili per una mansione, questa modalità di aggiunta compare se l'utente clicca la relativa icona presente nella toolbar della pagina "Todolist". Una volta cliccata l'icona si aprirà una finestra con un testo da completare corrispondente al nome della mansione e tre icone: l'icona di una data, l'icona di un gruppo e l'icona della priorità, che se cliccate consentono all'utente di inserire le informazioni opzionali.

- **Priorità:** la priorità di una mansione dispone di 3 opzioni:"Bassa priorità", "Alta priorità" e "Media priorità".

- **Visibilità:** la visibilità di una mansione si potrà indicare selezionando da una lista gli utenti del gruppo.

- **Data:** la data viene impostata, attraverso un calendario, indicando il giorno di scadenza della mansione.

Il fragment TodolistPage, si occupa di gestire l'aggiunta rapida di un nuovo elemento e la logica delle due Tab "Da completare" "Completato".
L'aggiunta di un elemento con la modalità rapida, inserendo solamente il nome della mansione, viene svolta, inizializzando un nuovo elemento di tipo Todolist, contenente il nome inserito dall'utente e impostando i valori di default della mansione.
Successivamente viene inizilizzata la Repository che ha il compito di gestire l'interazione con il database per la lista delle mansioni, attraverso un approccio asincrono, il fragment attenderà la risposta dal Database e in caso negativo mostrerà un errore.

Listing 4.6: Aggiunta elemento Todolist

```
val todoItem = TodoItem(name = itemName, date = Date(), members =
    members, created_by = userUID)
todoRepo.add(todoitem =
    todoItem).observeOn(AndroidSchedulers.mainThread())
    .subscribeOn(Schedulers.io()).doOnComplete {
    todolist_editext_newitem.setText("")
    Toast.makeText(context, "todoitem successfully created!",
        Toast.LENGTH_SHORT).show()
}.doOnError {
            Toast.makeText(context, "error!",
                Toast.LENGTH_SHORT).show()
        }.subscribe()
```

Il componente TabLayout che mostra le due Tab richiede l'utilizzo di un Adapter che estende la class FragmentStatePagerAdapter, in questo modo quando un utente interagirà con una delle due Tab, l'Adapter si occuperà di istanziare il Fragment corrispondente per visualizzare la lista delle mansioni completate o non completate.
Dato che per visualizzare gli elementi completati e da completare, teoricamente sarebbero servite due Fragment con parti di codice molto simile, si è scelto di realizzarne solamente uno, che in base al valore di un parametro

chiamato “type”, passato nella creazione dell’istanza del fragment, visualizzerà elementi differenti.

Listing 4.7: FragmentTodo.kt

```
companion object {
      fun newInstance(type: Int): TodoFragment {
          val fragment = TodoFragment()
          fragment.type = type
          return fragment
      }
  }
```

4.2.2 Spese

La seconda funzionalità principale dell’applicazione è la gestione delle spese condivise, per accedere a questa funzionalità l’utente dovrà cliccare l’icona di un portafoglio dal relativo menù delle funzionalità.
L’interfaccia che si presenta all’utente è molto simile all’interfaccia della gestione delle mansioni: è presente la visualizzazione globale delle spese da pagare e pagate e la possibilità di aggiungerne modificarne o cancellarne una.
Ogni utente appartenente al gruppo visualizzerà tutte le spese non completate e quelle già completate e in qualsiasi momento potrà marcare una spesa, segnandola come pagata.
La visualizzazione dei una singola spesa comprende di nome, la data di scadenza, l’icona della categoria a cui è associata la spesa, e la quota parziale che dovrà pagare l’utente. Cliccando su una spesa apparirà una finestra di dialogo che mostrerà il resoconto totale della spesa con la lista degli utenti che hanno pagato la loro quota e la lista degli utenti che ancora devono pagarla. Marcando una spesa l’utente segnerà di aver pagato la sua quota e di conseguenza la spesa, per quell’utente, verrà spostate automaticamente

nell'elenco delle spese pagate.

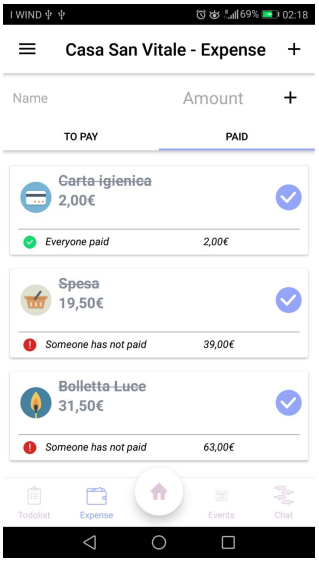

Figura 4.4: Schermata applicazione Spese

Le funzionalità di modifica ed eliminazione di una spesa sono consentite solamente all'utente che ha creato la spesa, gli altri utenti invece potranno solamente indicare di aver pagato la quota, marcandola.
Le modalità di aggiunta di una nuova spesa sono due, la modalità rapida e la modalità personalizzata.
La modalità rapida è accessibile attraverso l'interfaccia principale, nella parte superiore dello schermo infatti sono presenti due caselle di testo e un pulsante.Questa modalità permette di indicare solamente i parametri obbligatori: il nome e l'ammontare globale, alternativamente se l'utente vuole specificare anche altre informazioni, dovrà utilizzare il relativo pulsante per accedere alla finestra con tutti i campi opzionali per la creazione di una spesa.
La modalità di aggiunta personalizzata invece prevede un interfaccia con una casella di testo corrispondente al nome della spesa, e un'altra casella di testo corrispondente all'ammontare globale, sottostante alle caselle ci saranno tre icone: l'icona di un calendario, l'icona di un file, l'icona di un gruppo e

l'icona di un etichetta, che se cliccate consentiranno all'utente di inserire le informazioni opzionali.

- **Descrizione:** Casella di testo per indicare una descrizone della spesa
- **Visibilità:** Scelta multipla fra gli utenti del gruppo per indicare a chi è rivolta la spesa.
- **Data:** Calendario per indicare il giorno di scadenza della spesa.
- **Categoria:** Scelta multipla per indicare il la tipologia di spesa effettuata

Le categorie selezionabili sono preimpostate dall'applicazione e sono le seguenti:

- Bolletta Gas
- Bolletta Acqua
- Bolletta Luce
- Bolletta Telefono
- Cibo
- Pulizie
- Casa
- Strumenti
- Spesa generica

L'implementazione di questa funzionalità utilizza sempre il pattern MPV e la logica e simile a quella della funzionalità TodoList.

4.2.3 Eventi

Un'altra funzionalità è la gestione degli eventi, che permetterà ai membri del gruppo di creare degli eventi indicando una data, e qualora l'evento fosse ricorrente indicando la ricorrenza dell'evento.
L'interfaccia della pagina dedicata agli eventi è molto semplice, sulla parte superiore della toolbar è presente un icona che permette di aggiungere un nuovo evento, sottostante ad essa sono presenti tutti gli eventi sottoforma di lista.
Quando un utente clicca sull'icona per inserire un nuovo evento, verrà mostrata una finestra di dialogo, dove l'utente dovrà indicare il nome dell'evento, un eventuale descrizione, i partecipanti all'evento, e la ricorrenza dell'evento. La ricorrenza dell'evento può essere di cinque tipi differenti:

- Non ripetere
- Giornaliera
- Settimanale
- Mensile
- Annuale

Una volta selezionata la ricorrenza verrà richiesta la data di riferimento dell'evento.
Gli utenti hanno la possibilità di aggiungere all'interno del gruppo eventi classici ed eventi ricorrenti. La logica utilizzata per visualizzare ed aggiungere elementi all'interno del database è la stessa utilizzata per le precedenti funzionalità, è quindi presente un fragment principale, una view, un presenter che richiede al database firestore i dati da visualizzare, e un adapter.
Durante l'aggiunta di un nuovo elemento è stata usata la libreria SublimePicker che offre l'interefacia di un calendario personalizzabile, con cui far interagire l'utente per selezionare una data e impostare una ricorrenza. Il Widget SublimePicker necessita di un inizializzazione con il passaggio di un

oggetto “SublimeOptions” per poter funzionare.

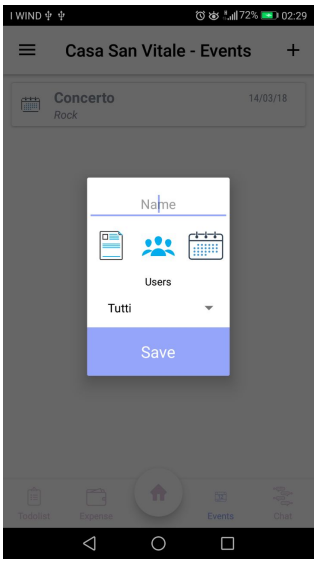

Figura 4.5: Schermata applicazione Eventi

Listing 4.8: Configurazione del Widget SublimePicker

```
..
val options = SublimeOptions()
var displayOptions = 0
displayOptions = displayOptions or
    SublimeOptions.ACTIVATE_DATE_PICKER
displayOptions = displayOptions or
    SublimeOptions.ACTIVATE_RECURRENCE_PICKER
options.pickerToShow = SublimeOptions.Picker.REPEAT_OPTION_PICKER
options.setDisplayOptions(displayOptions)
options.setCanPickDateRange(false)
..
```

Una volta indicati i parametri di default del Widget sarà possibile renderlo disponibile graficamente all’utente. Nella visualizzazione degli eventi per distingue se un evento è ricorrente è stato ricavato un nuovo dato non presente

nel database, che consiste nel differenziare eventi ricorrenti da eventi che non hanno nessuna ricorrenza.

Listing 4.9: Linea di codice del modello Event

```
var is_recurring =
    result.getString("recurring_type").isNullOrBlank().not()
```

Gli eventi ricorrenti hanno assegnato il valore di ricorrenza (Dayly,Weekly,Monthly,Yearly) all'interno della variabile "recurring'_type", di conseguenza effettuando un controllo sul contenuto della variabile è possibile creare la variabile booleana "is'_Recurring" che differenzia i due tipi di eventi

4.2.4 Chat

L'interfaccia della sezione chat è simile ad altre applicazioni di messaggistica istantanea e consente di visualizzare tutti i messaggi inviati dall'utente e ricevuti dagli altri membri del gruppo.
I messaggi inviati dall'utente saranno contrassegnati con un colore blu e si troveranno nella parte destra dello schermo, mentre i messaggi ricevuto dagli altri membri del gruppo si troveranno nella parte sinistra dello schermo con un colore differente e informazioni aggiuntive come il nome e l'avatar dell'utente che ha inviato il messaggio. Nella parte inferiore dello schermo è presente una casella di testo e un pulsante consentendo all'utente di scrivere e inviare un nuovo messaggio che sarà spedito in tempo reale a tutti i membri del gruppo, infatti una volta inviati, i messaggi appariranno come notifica a tutti i dispositivi online.
Quando un utente non dispone di una connessione internet, il messaggio verrà conservato e l'utente verrà notificato appena si connetterà ad internet.

La chat come le precedenti pagine, utilizza il pattern MVP, di conseguenza all'interno del fragment verranno instanziati: il Presenter l'Adapter e implementata la View.
La chat per facilitare la visione dei messaggi ricevuti e inviati all'interno del-

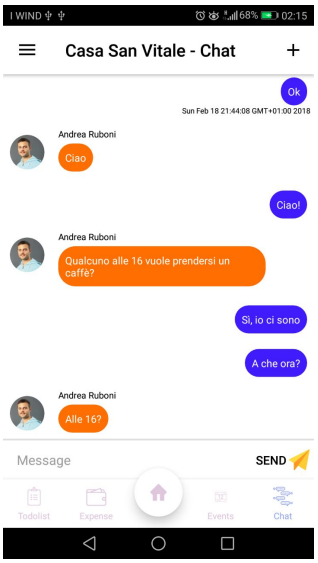

Figura 4.6: Schermata applicazione Chat

l'Adapter prevede due differenti ViewHolder che differenziano il messaggio inviato da quello ricevuto.
La realizzazione di questa distinzione grafica fra i due tipi di messaggio richiede la creazione di una classe astratta ChatHolder che avesse come unico metodo la funzione "bind" e prendesse come parametro il messaggio, in modo tale che i due holder verranno implementati basandosi e estendendo questa classe.
Successivamente sono state sovrascritti i metodi "getItemViewType" e "onCreateViewHolder" del Fragment in modo tale da introdurre il controllo necessario per distinguere i messaggi.
All'interno del metodo getItemViewType, viene controllato il campo "SenderUID" del messaggio, corrispondente all'ID dell'utente che ha inviato il messaggio nel gruppo, questo ID viene poi confrontato con l'ID dell'utente loggato all'interno dell'applicazione in modo da restituire l'identificativo del layout da utilizzare nel ViewHolder.

Listing 4.10: Logica della funzione getItemViewType della Chat

```
when (senderUid == userUid) {
    false -> return R.layout.item_message_recived
    true -> return R.layout.item_message_sent
}
```

Una volta differenziato il tipo attraverso il metodo getItemViewType, è stata sovrascritto il metodo onCreateViewHolder che dovendo restituire un singolo Holder, come valore di ritorno restituisce un tipo ChatHolder (classe astratta implementata dalle due ViewHolder). In questo modo in base al tipo restituito da getItemViewType la classe onCreateViewHolder restituirà l'holder corrispondente al messaggio.

Listing 4.11: Logica della funzione onCreateViewHolder della chat

```
when (viewType) {
          R.layout.item_message_sent -> holder =
              MessageChatSentHolder(itemView = view,
              dateOfLastMessage = lastItem?.timestamp)
          R.layout.item_message_recived -> holder =
              MessageChatRecivedHolder(itemView = view,
              dateOfLastMessage = lastItem?.timestamp)
      }
```

4.3 Supporto

Il progetto è stato realizzato utilizzando librerie open source importante e gestite da Gradle, un programma per l'automazione dello sviluppo.
Gradle è stato progettato per automatizzare il processo di generazione di un progetto, facilitando l'aggiunta di librerie esterne, la compilazione finale del progetto comprese le sue dipendenze e l'aggiunta di test automatizzati.

La lista delle librerie utilizzate nel progetto possono essere divise in quattro categorie:

- Librerie di supporto per Android e Kotlin
- Librerie SDK di Firebase
- Libreria per il supporto della programmazione reattiva
- Librerie per la gestione delle immagini
- Librerie per componenti grafici aggiuntivi

4.3.1 Librerie

Le librerie offerte da Android per il supporto di funzionalità e componenti grafici aggiuntivi sono le seguenti:

Libreria	**Descrizione**
com.android.support:support	Libreria di supporto per Android
com.android.support:appcompat	Supporto per i componenti grafici aggiuntivi
com.android.support:recyclerview	Widget personalizzabile simile al ListView ma più avanzato e flessibile
com.android.support:cardview	Componente per realizzare interfacce Material Design
com.android.support:design	Libreria di supporto per l'interfaccia Android
com.android.support:constraint-layout	Layout personalizzabile
com.google.android:flexbox	Layout personalizzabile Flex per Android
com.android.support:customtabs	Libreria di supporto per il widget Tab
android.arch.lifecycle:extensions	Supporto per Android Components Architecture
com.android.support:vector-drawable	Supporto per immagini svg e vettoriali
android.arch.lifecycle:common-java8	Supporto per la gestone del lifecycle Android
org.jetbrains.kotlin:kotlin-stdlib-jre7	Supporto per il linguaggio Kotlin su Android

Tabella 4.1: Librerie Google del progetto

Le librerie di supporto offerte da Firebase per utilizzare i suoi servizi sono le seguenti:

Libreria	**Descrizione**
com.google.android.gms:play-services-auth	Libreria di supporto per i servizi Google
com.google.firebase:firebase-auth	SDK del servizio di autenticazione di Firebase
com.google.firebase:firebase-firestore	SDK per interfacciarsi con il database Firestore
com.google.firebase:firebase-storage	SDK per usufruire dello storage di Firebase
com.google.firebase:firebase-messaging	SDK del servizio di messaggistica
com.firebase:firebase-jobdispatcher	Supporto per lo scheduling e l'esecuzione in background
com.firebaseui:firebase-ui-auth	Libreria di supporto per il servizio Firebase-Auth
com.firebaseui:firebase-ui-firestore	Libreria di supporto per il servizio Firestore
com.firebaseui:firebase-ui-storage	Libreria di supporto per il servizio Storage
com.facebook.android:facebook-android-sdk	SDK per l'accesso tramite il social Facebook
com.twitter.sdk.android:twitter-core	SDK per l'accesso tramite il social Twitter

Tabella 4.2: Librerie Firebase del progetto

Le librerie che offrono componenti grafici aggiuntivi sono le seguenti:

Libreria	**Descrizione**
com.github.ittianyu: BottomNavigationViewEx	Widget per il menu di navigazione
com.stephentuso:welcome	Libreria per mostrare pagine di presentazione e introduzione iniziali
com.shuhart.moneyedittext: moneyedittext-kotlin	Widget per indicare l'ammontare di una spesa
br.com.simplepass:loading-button-android	Widget che trasforma un Button in unna barra di caricamento
com.github.Mariovc:ImagePicker	Libreria di supporto per selezionare un'immagine dalla galleria del dispositivo
com.github.bumptech.glide:glide	Libreria per la gestione delle immagini con funzionalità di caching
de.hdodenhof:circleimageview	Libreria per visualizzare immagini con un bordo circolare
com.appeaser.sublimepickerlibrary: sublimepickerlibrary	Libreria per la selezione di una data dal calendario personalizzabile

Tabella 4.3: Librerie Firebase del progetto

Infine le librerie di supporto per implementare la programmazione reattiva sono i seguenti:

Libreria	**Descrizione**
io.reactivex.rxjava2:rxjava:	Supporto dei costrutti della programmazione reattiva su Java
io.reactivex.rxjava2:rxkotlin	Supporto per la programmazione reattiva su Kotlin
io.reactivex.rxjava2:rxandroid	Supporto della libreria RxJava2 su Android

Tabella 4.4: Librerie di supporto per il Reactive programming

Capitolo 5

Sviluppo Server

La realizzazione e lo sviluppo della parte server è stato facilitato dal servizio Firebase, che facilitava la creazione del database, la scrittura di regole di sicurezza e l'implementazione di script per il servizio Cloud Functions, la manutenzione dei server invece è stata interamente gestita da Firebase.

5.1 Notifiche

Gli utente appartenenti ad un gruppo riceveranno delle notifiche in tempo reale ogni volta che verrà aggiunto un nuovo contenuto all'interno dell'applicaziond, in particolare quando un utente aggiunge o completata una mansione, un evento o una spesa, oppure viene ricevuto un messaggio dalla chat.
Il servizio che consente di inviare messaggi ai dispositivi è Firebase Cloud Messaging e opera tipicamente attraverso le porte 5228 fino alle 5230, le richieste che vengono effettuate dall'applicazione al server FCM sono:

- Creazione di un nuovo token attraverso l'SDK.
- Creazione di un gruppo comprendente più token.
- Aggiunta di un token all'interno di un gruppo di token.

La richiesta e il processo di generazione di un nuovo token per un nuovo client è automatizzata dall'SDK di FCM. La creazione di un nuovo gruppo di utenti invece è effettuabile attraverso una richiesta HTTP POST ad un API, che restituisce un token, utilizzato come identificativo del gruppo.

Listing 5.1: Creazione di un token per il servizio FCM

```
https://android.googleapis.com/gcm/notification
Content-Type:application/json
Authorization:key=API_KEY
project_id:SENDER_ID
{
   "operation": "create",
   "notification_key_name": "appTesi",
   "registration_ids": ["4", "8", "15", "16", "23", "42"]
}
```

Una volta creato il gruppo è possibile aggiungere altri membri al gruppo, effettuando una richiesta simile alla precedente indicando come tipo di operazione "add" anzichè "create", che indica l'aggiunta di un nuovo token, oltre ad indicare il tipo di operazione la richiesta necessita del token del dispositivo da aggiungere al gruppo.

Listing 5.2: Creazione token FCM

```
https://android.googleapis.com/gcm/notification
Content-Type:application/json
Authorization:key=API_KEY
project_id:SENDER_ID

{
   "operation": "add",
   "notification_key_name": "appTesi",
   "registration_ids": ["7"]
}
```

5.2 Cloud Functions

I cambiamenti all'interno del database Firestore vengono monitorati utilizzando il servizio Cloud Functions che consentiva di scrivere script NodeJS eseguiti in base ai cambiamenti avvenuti nel database.
Gli script vengono salvati all'interno di un unico file, con l'estensione ".js", questo file conterrà tutte le funzioni che il Cloud Functions dovrà monitorare ed eseguire.
Il file principale utilizza due librerie per funzionare: "Firebase-functions" e "firebase-admin", che offrono le funzionalità utilizzabili dalle funzioni per interagire con Cloud Functions e con gli altri servizi Firebase.

Listing 5.3: Librerie utilizzate per interagire con il servizio Cloud Functions

```
// Definisco le librerie
let functions = require('firebase-functions');
let admin = require('firebase-admin');

//Inizializzo la libreria admin
admin.initializeApp(functions.config().firebase);
//Definisco ''firestore'' utilizzando il relativo modulo della
    libreria admin
const firestore = admin.firestore();
```

Ogni singola funzione invece è contrassegnata da un nome, il documento o la collezione del database da monitorare e l'evento ad esso associato.
Le funzioni scritte sono molto simili fra loro poichè monitorano solamente l'aggiunta di un nuovo elemento all'interno di una collezione o il cambiamento di un valore all'interno di un documento.

Prendendo in considerazione una funzione è possibile vedere quindi tutti gli elementi e le caratteristiche utilizzate per scrivere le funzioni, un esempio utilizzato per lo sviluppo della gestione della todolist è il seguente:

Listing 5.4: Funzone Cloud Functions

```
exports.onGroupTodoCompleted =
    functions.firestore.document("groups/{groupsID}/todolist/{todoID}")
    .onUpdate(event => {
    const promises = [];
    const todo = event.data.data();
    const groupUIid = event.params.groupsID
    const getGroupTokenFCMPromise =
        firestore.collection("groups").doc(groupUIid).get()
    const prev_status = event.data.previous.data().status
    if (todo.status !== prev_status) {
        ...
    } else {
        return Promise.all(promises);
    }
});
```

In questa porzione di codice è possibile notare la struttura base di una funzione comprendente il nome (onGroupTodoCompleted), il riferimento alla collezione da monitorare (groups/groupsID/todolist/todoID) e l'evento (onUpdate).
Quando un documento all'interno della collezione "todolist" subisce una modifica viene confrontato il valore "status" della mansione precedente alla modifica, se questo valore è cambiato allora verrà inviata una notifica ai membri del gruppo altrimenti la funzione terminerà restituendo un Array vuoto.
Cloud Functions come citato in precedenza, permette di interagire anche con altri servizi Firebase, in questo caso il servizio con cui interagisce è Cloud Messaging, permettendo quindi di inviare una notifica ad uno o più dispositivi.
Un esempio di inivio di una notifica attraverso le Cloud Functions è il seguente:

Listing 5.5: Funzone Cloud Functions

```
const groupTokenFCM = groupResponse.data().tokenFCM
var todotype = (todo.status === 'true') ? COMPLETED_TODO_TYPE :
    NEW_TODO_TYPE;
var payload = {
    "data": {
        "type": todotype, "name": todo.name, "sender":
            todo.created_by,
    }
};
const pushNotificationPromise =
    admin.messaging().sendToDeviceGroup(groupTokenFCM, payload)
return Promise.resolve(pushNotificationPromise){
    .then(function(response) {
        return Promise.all(promises);
    })
    .catch(function(error) {
        console.log("Error sending todo message:", error);
    })
}).catch(function(error) {
    console.log("Error sending todo message:", error);
})
```

5.3 Sicurezza

La sicurezza dei dati presenti sul database è fornita attraverso le Firestore Rules, un servizio integrato all'interno del database che permette di definire regole di accesso e validazione dei dati, attraverso una sintassi propria simile al Javascript.

I dati possono essere visualizzati solamente dagli utenti registrati, di conseguenza è stata creata una funzione ausiliare che verrà richiamata all'in-

terno delle relative regole riguardanti le collezioni che utilizzano il controllo dell'accesso.

Listing 5.6: Firestore Rules Controllo autenticazione

```
function isUserAuthenticated() {
  return request.auth.uid != null;
}
```

Successivamente sono state definite le funzioni per controllare che l'utente faccia parte della mansione spesa o evento interno al gruppo.

Listing 5.7: Firestore Rules Controllo Visibilità

```
function userBelongsToGroup(userId) {
  return resource.data.users[userId] != null;
}
```

Queste funzioni vengono richiamate dalle regole da applicare alle collezioni, una particolarità utile è la possibilità di definire regole all'interno di sub-collezioni che sovascrivono parzialmente o totalmente le regole definite nella collezione esterna. Un esempio è il seguente:

Listing 5.8: Firestore Rules Controllo Visibilità

```
match /groups/{groupsId} {
    allow read: if isUserAuthenticated();
    allow write, update, delete: if isUserAuthenticated() &&
        userBelongsToGroup()

    match /todolist/{todolistId} {
            allow write, update, delete if canUserReadItem()
          ...
    }
  ..
}
```

Conclusioni

Lo stato dell'arte al momento in cui si scrive è un'applicazione in grado di gestire eventi, commissioni da svolgere, spese condivise con la possibilità di usufruire di una chat di messaggistica istantanea.
Attraverso l'utilizzo del database Firebase ed i suoi servizi viene offerta una sincronizzazione in tempo reale dei dati con la possibilità di consultarli anche offline.
L'applicazione pur garantendo l'uso quotidiano delle funzionalità può essere ulteriormente migliorata, aggiungendo funzionalità, rendendola più stabile e riscriverla anche per la piattaforma iOS. Alcune idee in fase di sviluppo che verranno implementante sono le seguenti:

- Chat di messaggistica istantanea anche fra i singoli componenti del gruppo
- Migliore flessibilità e funzionalità nella gestione degli eventi
- Nuovi informazioni inseribili all'interno di una spesa o commissione
- Permettere all'utente di partecipare a più di un gruppo contemporaneamente
- Scrivere l'applicazione per iOS utilizzando il linguaggio Swift

Attualmente il codice dell'applicazione è presente su Gitlab[1], in una repository privata, non appena l'applicazione sarà considerata stabile e con le

[1]http://gitlab.com/

funzionalità desiderate, verrà resa pubblica.

L'applicazione è stata scritta utilizzando il linguaggio di programmazione Kotlin, a differenza del linguaggio Java solitamente utilizzato nello sviluppo di applicazioni mobili Android.

L'utilizzo di Kotlin come linguaggio per lo sviluppo dell'applicazione ha permesso di analizare i nuovi concetti introdotti dal linguaggio e porlo a confronto con Java. Il confronto effettuato fra Java e Kotlin è stato possibile poichè si aveva a disposizione sia una parte del progetto scritta inizialmente in Java, sia la parte del progetto corrispondente in Kotlin. I risultati ricavabili dal confronto effettuato, sono stati significativi soprattutto per quanto riguarda le linee di codice, alcune funzionalità che in Java richiedevano 140 linee di codice in Kotlin la stessa logica veniva implementata in circa 80 linee.

L'utilizzo della memoria e la dimensione dell'APK finale invece rimanevano inviariati,

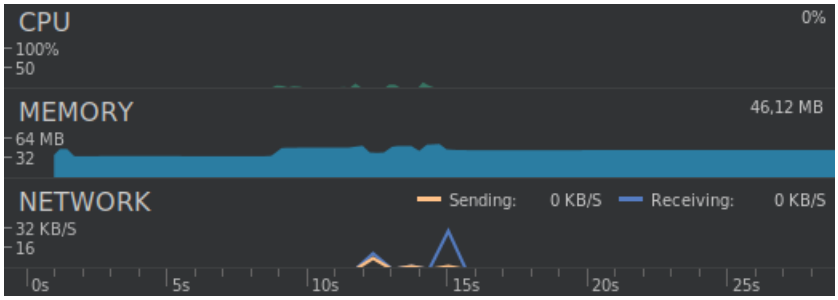

Figura 5.1: Android Studio Memory Profile Java

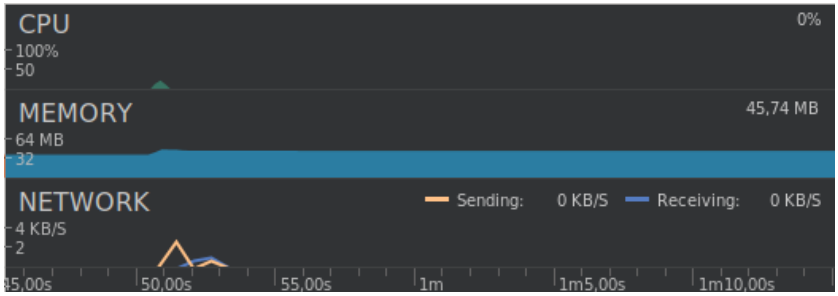

Figura 5.2: Android Studio Memory Profile Kotlin

La tesi, il progetto, lo studio di Kotlin, e la realizzazione di un'infrastruttura server attraverso i servizi Firebase di Google, ha permesso di avere una visione globale della realizzazione di un applicazione in Kotlin, utilizzando tecnologie e tecniche di programmazione recenti.

Bibliografia

[1] Massimo Carli *Android 6. Guida per lo sviluppatore*. Apogeo, 2016.

[2] Stephen Samuel, Stefan Bocutiu *Programming Kotlin*. Packtpub, 2017.

[3] Miloš Vasić *Mastering Android Development with Kotlin*. Packtpub, 2017.

[4] Ronak K. Panchal,Akshay K. Patel *A comparative study: Java Vs kotlin Programming in Android*. International Journal of Innovative Trends in Engineering and Research, 2016.

[5] ReactiveX documentation
`http://reactivex.io/documentation`

[6] Google Android Developer Documentation
`https://developer.android.com`

[7] Google Firebase Documentation
`https://firebase.google.com/docs/`

[8] Google I/O - 2017
`https://events.google.com/io/`

[9] Kotlin Language Documentation
`https://kotlinlang.org/docs/kotlin-docs.pdf`

[10] Oracle Java Documentation
`https://docs.oracle.com/`

[11] Xamarin Android Google services guides
`https://developer.xamarin.com/guides/android/data-and-cloud-services/`

[12] Libreria per la gestione dei servizi Firebase
`https://github.com/firebase/FirebaseUI-Android`

[13] Libreria di supporto per la programmazione reattiva su Java
`https://github.com/ReactiveX/RxJava`

[14] Libreria di supporto per la programmazione reattiva su Kotlin
`https://github.com/ReactiveX/RxAndroid`

[15] Libreria di supporto per la programmazione reattiva su Android
`https://github.com/ReactiveX/RxKotlin`

[16] Libreria di supporto per il Widget del menu di navigazione
`https://github.com/ittianyu/BottomNavigationViewEx`

[17] Libreria di supporto per mostrare pagine di presentazione e introduzione iniziali
`https://github.com/stephentuso/welcome-android`

[18] Libreria di supporto per il widget che consente di indicare l'ammontare di una spesa
`https://github.com/shuhart/MoneyEditText`

[19] Libreria di supporto per il widget che trasforma un Button in una barra di caricamento
`https://github.com/leandroBorgesFerreira/LoadingButtonAndroid`

[20] Libreria di supporto per la gestione delle immagini con funzionalità di caching
`https://github.com/bumptech/glide`

[21] Libreria di supporto per visualizzare immagini con un bordo circolare
`https://github.com/hdodenhof/CircleImageView`

[22] Libreria di supporto per la selezione di una data da un calendario personalizzabile
`https://github.com/vikramkakkar/SublimePicker/`

[23] Libreria di supporto per selezionare un'immagine dalla galleria del dispositivo
`https://github.com/Mariovc/ImagePicker`

[24] Libreria di supporto per l'accesso tramite il social network Facebook
`https://github.com/facebook/facebook-android-sdk`

[25] Libreria di supporto per interagire con i servizi Firebase con NodeJS
`https://github.com/firebase/firebase-admin-node`

[26] Tool a linea di comando per gestire i servizi Firebase
`https://github.com/facebook/facebook-android-sdk`

[27] Libreria di supporto per l'utilizzo del servizio Firebase Cloud Functions
`https://github.com/firebase/firebase-functions`

Ringraziamenti

Un ringraziamento speciale alla mia famiglia, parenti e amici che mi ha sempre sostenuto in questo percorso.