Dipartimento di Scienze e Innovazione Tecnologica

Corso di Laurea in Informatica

Relazione per la prova finale

FIX-IT

Stream Processing su un sistema event-driven per la gestione di disservizi pubblici

Tutore interno:
Dott. Davide Cerotti

Candidato:
Manuel Pagliuca
20025185

Anno Accademico 2019/2020

Ringraziamenti

Prima di procedere con la trattazione intendo dedicare una pagina per i ringraziamenti, visto che il risultato di questo elaborato è stato reso possibile anche grazie all'impegno di diverse persone, che sarebbe sbagliato non ringraziare. In primis al Dott. Davide Cerotti che mi ha dato l'opportunità di lavorare su questo progetto molto interessante, fornendo indispensabili consigli ed aiutando ad organizzare il lavoro nella maniera più chiara possibile. Un particolare ringraziamento va ai miei amici più stretti, i quali mi supportano da sempre, che hanno sempre spinto affinché io riuscissi a realizzare i tanti obiettivi della mia vita, siete stati di immenso aiuto. Un grazie soprattutto a coloro che si sono dimostrati sempre pronti ad ascoltarmi e ad accompagnarmi in questo lungo percorso universitario, che oggi posso dire di avere portato a termine anche per il merito del loro supporto. Nel particolare alla mia ragazza, che da quando l'ho conosciuta si è sempre preoccupata per me, sostenendomi in qualsiasi decisione io prendessi. Un sentito ringraziamento va anche al mio collega di progetto nonché compagno di corso Nicolas Palermo, con cui ho lavorato e condiviso fatiche fin dalle prime fasi di progettazione dell'applicazione (Ottobre 2019). Ritengo doveroso ringraziare tutte quelle persone che rendono disponibile online materiale su cui studiare e imparare, mettendoci molta fatica per delle persone che probabilmente non vedranno mai nel corso della loro vita, senza di loro la stesura di questa tesi sarebbe stata molto più ardua. Un ultimo ringraziamento ma non per importanza, va alla mia famiglia, mi avete sempre aiutato e sostenuto in qualsiasi momento. Ringrazio i miei genitori, avete sempre fatto di tutto e di più per me e continuate a farlo, se sono riuscito a scrivere questa tesi è per merito vostro che me ne avete dato la possibilità, non esistono parole per ringraziarvi di tutto quello che avete fatto e continuate a fare ogni giorno per me.

Grazie a tutti
Manuel Pagliuca

Indice

Elenco delle figure

Capitolo 1

Introduzione

Nell'ultimo decennio si è verificata una crescita esponenziale dei dati e sempre più aziende sono interessate all'argomento Big Data [29] ed in particolare della loro trasformazione in Fast Data. Si passa quindi da Big Data di prima generazione che memorizzavano dati su un real-time database, ad un nuovo approccio dove le informazioni vengono analizzate ed elaborate all'istante. Quindi il dominio della velocità si espande e non riguarda più la fase di raccolta dati ma anche il processamento di essi. L'avvento dei Fast Data sta causando un'importante transizione dei processi di tipo batch allo streaming di dati in tempo reale [30]. Ora più che mai si ha la necessità di mantenere un'interazione sostenuta con un sistema che sia distribuito su differenti dispositivi (PC, mobile, tablet, ...), tutti questi dispositivi hanno un modo differente di interagire con il sistema ed è opportuno creare un ambiente che non porti alla generazione di conflitti interni. Questa tesi ha lo scopo di analizzare le varie fasi avvenute nello sviluppo di un sistema event-driven utilizzando le tecnologie precedente descritte, ed in particolare concentrandosi su quella che implementerà lo **stream processing**, ovvero Apache Kafka [24], al giungere della parte conclusiva verranno discussi i vantaggi e svantaggi da essa comportati.

1.1 Scopo del lavoro

Obbiettivo finale è stata la costruzione di un sistema con architettura **EDA** (Event-Driven Architecture) per la gestione delle segnalazioni di disservizi. Quest'ultimo permette a gli utenti registrati di poter segnalare disservizi pubblici da dispositivi mobile che verranno gestiti in remoto dai membri del personale amministrativo. Tale sistema permette una veloce localizzazione dei disservizi pubblici da parte del cittadino, rendendo quest'ultimo un membro attivo all'interno della piattaforma. Inoltre, il sistema si offre anche come strumento di monitoraggio della località, dando possibilità a gli uti-

lizzatori di accertarsi che non siano presenti dei pericoli in prossimità della loro posizione.

Ulteriore obiettivo dello studio guidato è stato quello di sperimentare un'integrazione della piattaforma Apache Kafka su un sistema event-driven, in modo da poter realizzare elaborazioni su flussi di dati mediante lo **stream processing**. L'offerta di studio guidato è stata proposta dal Prof. Cerotti al sottoscritto ed al compagno di corso Nicolas Palermo, permettendoci di svolgere un lavoro di gruppo e suddividendoci i compiti da svolgere. La direzione intrapresa per la costruzione del sistema è stata di realizzare due applicativi rispettivamente per un utente utilizzatore (quindi in grado di effettuare soltanto delle segnalazioni) e per un utente che le gestisca (quindi con più privilegi del precedente).

1.2 Struttura del documento

La tesi inizialmente presenta gli *strumenti* utilizzati nel capitolo 2, ogni sezione corrisponde ad uno degli strumenti utilizzati.

Dopo il capitolo inerente alla strumentazione, il successivo capitolo 3 tratta la prima fase del ciclo di sviluppo del software, dove a seguito di una fase di *analisi dei requisiti*, seguirà un elenco dei requisiti funzionali e non funzionali coinvolti nel sistema.

Il capitolo 4 riguarda la *progettazione* del sistema, dove verrà illustrata l'architettura del sistema (sezione 4.1) descrivendone il funzionamento delle componenti e delle interfacce.

L'*implementazione* dei requisiti funzionali descritti nel capitolo 3 verrà discussa nel capitolo 5, dove vengono mostrate alcune implementazioni prodotte dal sottoscritto all'interno del progetto.

Il progetto termina con delle note conclusive a riguardo di **Apache Kafka**[24], ricavate dal completamento di questo lungo studio guidato.

Capitolo 2

Strumenti utilizzati

2.1 Apache Kafka

Apache Kafka[24] è una cross-platform open-source per stream processing distribuita che permette di pubblicare, sottoscrivere, archiviare ed elaborare flussi di record in tempo reale. È stata sviluppata dalla **Apache Software Foundation** (originariamente da *Linkedin*) il progetto ha come obiettivo quello di fornire una piattaforma unificata, con un alto throughput e bassa latenza per la gestione di feeds di dati in tempo reale.

2.1.1 Broker, Producer e Consumer

Lo stile architetturale di Kafka è il *Publisher-Subscribe*, questo pattern viene utilizzato per la comunicazione asincrona fra diversi processi. Il pattern prevede tre attori principali, il **broker** che è il tramite degli altri due, il **producer**, che può soltanto pubblicare verso il broker ed in fine il **consumer** che può sottoscriversi (*subscribe*) al broker e leggere (*consume*) i messaggi da quest'ultimo. Apache Kafka viene eseguito come un **cluster** di uno o più server che possono espandersi a molteplici datacenters o cloud regions. Alcuni di questi server sono propio broker, mentre altri si occupano di integrare Kafka con elementi esterni (per esempio *Kafka Connect* permette di effetture un continuo import/export verso piattaforme esterne). Un **Kafka Cluster** è altamente **scalabile** e **fault-tolerant**, se uno dei suoi servers cade o si guasta, gli altri server saranno in grado di sostituirlo prendendo il suo carico di lavoro, in modo da assicurarsi il continuo delle operazioni senza causare una perdita di dati. In **figura 2.1** è presente uno schema semplificato di un singolo **broker** all'interno del cluster che riceve i messaggi dai producers e li veicola verso i consumers (i quali sono iscritti al broker).

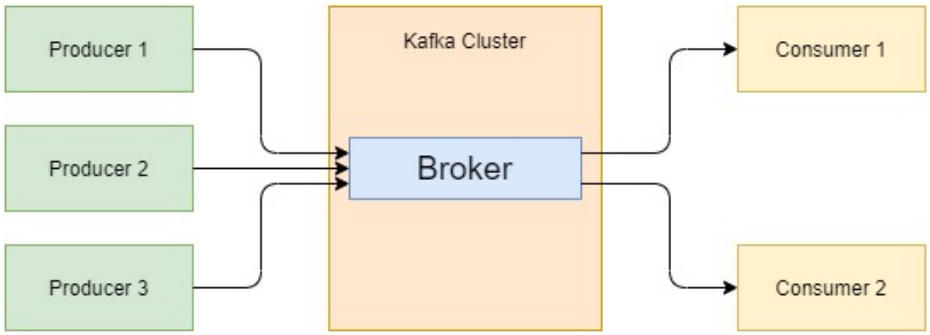

Figura 2.1: Architettura Apache Kafka, singolo broker

2.1.2 Topic

I messaggi che transitano all'interno della piattaforma di Apache Kafka vengono chiamati con differenti nomi tra cui : **record**, **eventi** o semplicemente **messaggi**. La struttura dei record è tendenzialmente composta da tre campi la **chiave**, il **valore** ed il **timestamp**. Questi messaggi vengono memorizzati all'interno dei **topic**, possiamo rappresentare i topic per analogia, immaginandoli come come una cartella all'interno di un filesystem, dove gli eventi sono i file all'interno di quella cartella. I topic sono *multi-subscriber* e *multi-producers*, ovvero, possono avere differenti istanze (anche nessuna) di producers e consumers connesse ad essi. I messaggi all'interno del topic possono essere letti, ed a differenza di altri sistemi i messaggi **non** vengono eliminati dal topic dopo essere stati letti (*consumed*). Sarà compito di chi imposta le configurazioni del topic dire per quanto tempo trattenere i messaggi all'interno del topic, nonostante ciò, la performance di Kafka rimane costante indipendentemente dalla quantità di messaggi all'interno del topic. I topic vengono **partizionati**, ovvero che il singolo topic viene distribuito su un numero arbitrario di partizioni. Questo partizionamento distribuito dei dati è molto importante per la **scalabilità**, permette a differenti clients di leggere e scrivere dai brokers nello stesso momento. Quando un nuovo record viene scritto su un topic, quello che accade è che viene accodato ad una delle partizioni. I record con la stessa chiave vengono scritti nella stessa partizione, ed è Kafka a garantire che l'ordine con cui verranno letti i messaggi di un topic (da parte di un consumer) combaci con lo stesso ordine con cui sono stati scritti.

2.1.3 Kafka Streams

Kafka Streams è una libreria client [13] per il processamento e l'analisi dei dati memorizzati all'interno di Kafka (nello schema generale dell'architettura è rappresentata dalla freccia ricorsiva nella **figura 4.1**). Nonché un motore di stream processing distribuito che permette la costruzione di pipeline di

dati in real-time e applicazioni in streaming. La libreria permette di effettuare delle operazioni di arricchimento e filtraggio dei dati, o più in generale di elaborazione su flussi di dati in tempo reale. Un applicazione di stream processing è una applicazione che utilizza la libreria **Kafka Streams API**, essa definisce la logica computazionale tra una o più *processor topologies* che rappresentano un grafo di **stream processors** (i nodi) connessi da **stream** (gli archi).
Un nodo rappresenta un passo di processamento (quindi uno step di elaborazione e modifica) del dato ricevuto in input dai nodi procedenti di upstream processors per poi passarlo ai nodi di *downstream*.
Esistono due stream processor speciali all'interno della topologia:

- il **source processor**, il quale non ha nessun upstream processor e produce un input stream per la topologia prelevando le informazioni da un topic.
- il **sink processor**, il quale non ha nessun downstream processor e veicola le informazioni ricevute dagli upstream processor ad un topic in uscita.

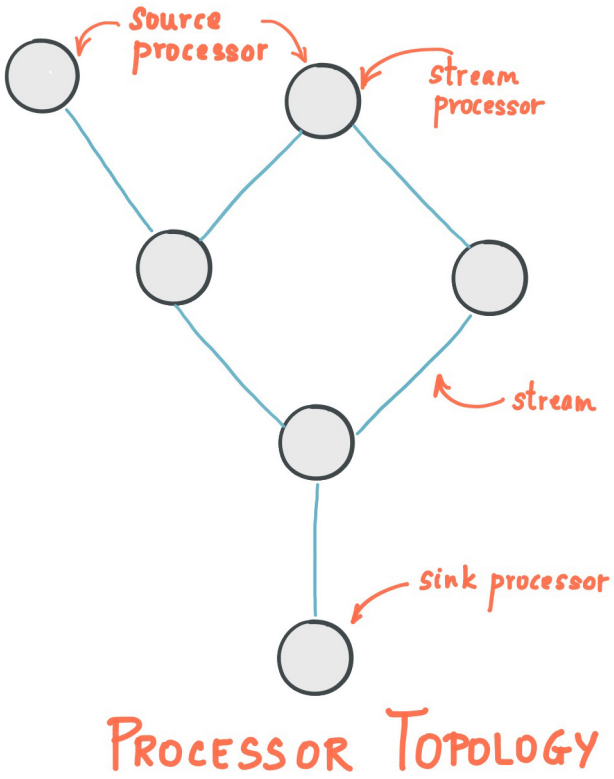

Figura 2.2: Architettura di una processor topology

Source: *https://kafka.apache.org/26/documentation/streams/core-concepts*

Dualità KStream-KTable

Un concetto importante di Kafka Streams è quello riguardante la possibilità di convertire istantaneamente un oggetto **KStream** in un oggetto **KTable** [4], rispettivamente rappresentano un **changelog di records** ed una **tabella di records**. Durante l'implementazione di uno stream processing, solitamente si ha bisogno sia di **stream** che di **database**, questi due oggetti sono essenzialmente due rappresentazioni differenti della stessa cosa.

Possiamo rispettivamente vederli come :

- Uno **stream** può essere considerato come un *changelog* di una tabella, dove ogni record rappresenta un cambio di stato della tabella. Questo fa di uno stream una tabella "*camuffata*", basterà ripetere i record del changelog dall'inizio alla fine per ricostruire facilmente una tabella.

- Una **tabella** può essere considerata uno *snapshot* (o checkpoint), dell'ultimo valore **per ogni** chiave all'interno di uno stream. Una tabella non è altro che uno stream "*cammufato*", e può essere reso uno stream iterando su ogni entry della tabella.

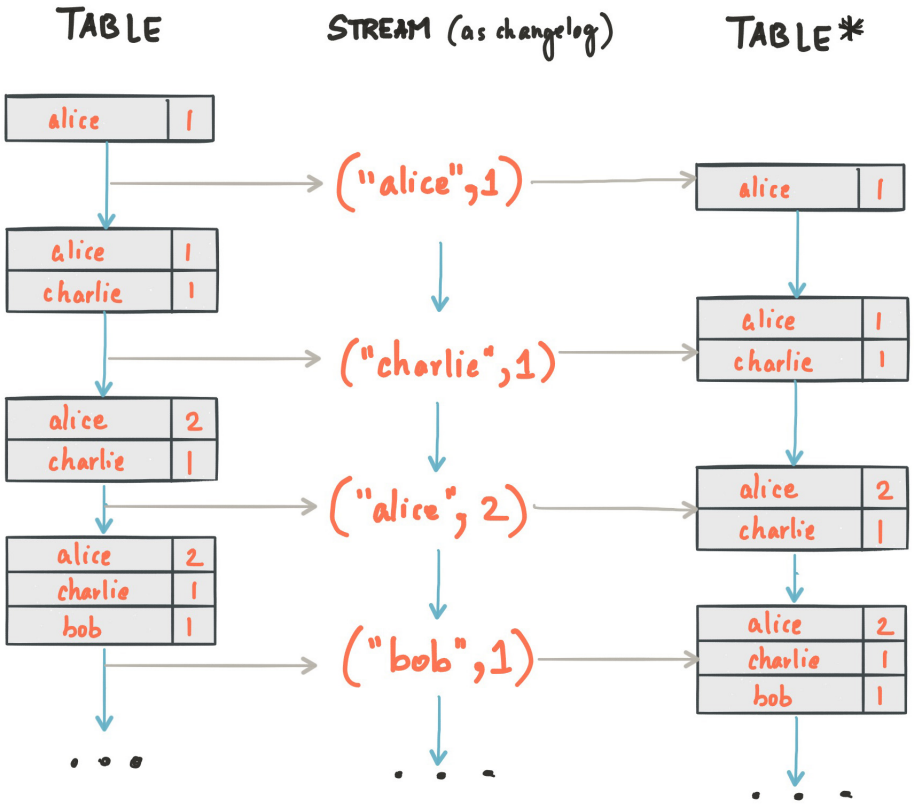

Figura 2.3: Dualità tra tabella e stream

Source: *https://docs.confluent.io/current/streams/concepts.html*

La motivazione che sta alla base di questo meccanismo tutt'altro che banale risiede nella possibilità di effettuare delle operazioni **stateful** sulle **KTable**. Un operazione stateful è un operazione in cui il risultato della trasformazione **dipende** da un informazione esterna, lo **stato**, quindi operazioni come *count()*, *aggregate()*, *reduce()*. L'operazione di conversione da **KStream** in KTable è effettuata tramite una operazione **stateless**, in cui risultato della trasformazione dipende solamente dall'input (e da nessun'altra informazione esterna) *groupBy()*. Viceversa, le **KTable** vengono anch'esse convertite in KStream attraverso un operazione stateless *toStream()*.

2.2 Firebase

È una piattaforma per lo sviluppo di applicazioni mobili attualmente sviluppata da Google [21] (originariamente fu una compagnia indipendente lanciata nel 2011). Questa piattaforma offre svariati prodotti per la gestione della componente web all'interno di un sistema, ne sono stati scelti tre, ma i più fondamentali per il progetto sono i primi due ovvero Firebase **Realtime Database** e Firebase **Authentication**.

2.2.1 Firebase Realtime Database

Il *real time database* permette l'archiviazione e la sincronizzazione dei dati sul database cloud NoSQL. I dati vengono sincronizzati in tempo reale e rimangono disponibili anche quando l'applicazione va offline. I dati sono rappresentati come una coppia **chiave-valore**, vengono archiviati come JSON [9]. Ogni dispositivo mobile appartenente al sistema condivide un istanza con il real time database. Questo permette ad ogni dispositivo presente all'interno della piattaforma di ricevere aggiornamenti per la sincronizzazione in tempi molto brevi, in maniera ben diversa dalla classica richiesta **HTTP** (Hypertext Transfer Protocol).

2.2.2 Authentication

Servizio che permette di autenticare all'interno della propria piattaforma un dispositivo mobile registrato. La peculiarità di questo servizio è la possibilità di creare registrazione e login in semplicità utilizzando delle chiamate estremamente semplici. Durante la generazione del nuovo utente ad esso viene conferito un **UID** (User Identifier) che è un identificativo univoco per l'utente all'interno della piattaforma. Si è presa la decisione di riutilizzare questo identificativo all'interno del real time database per creare una seconda versione (effettuando un *mapping*) degli utenti registrati arricchita con altri dati non essenziali (l'associazione è illustrata nella **figura 2.4**). Questo avviene anche per gli utenti impiegati, seppur la registrazione non avviene via applicazione ma in maniera formale compilando dei moduli cartacei, illustrato nel documento S.R.S. (Specifica dei requisiti Software, **Appendice 6.2**).

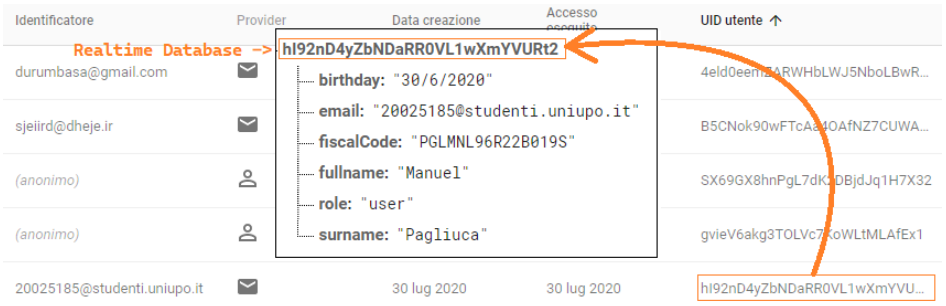

Figura 2.4: Mapping di una entry del servizio di autenticazione sul real time database.

2.2.3 Storage

Firebase Storage è stato utilizzato prevalentemente per l'archiviazione dei dati caricati dagli utenti. In modo che i dati siano a disposizione degli sviluppatori.

Storage è un servizio di **stoccaggio** dati potente ed economico, l'accesso al servizio è molto semplice grazie alle **API** fornite da Firebase. I dati sono organizzati come una directory di un filesystem dove è possibile creare la propria cartella e caricare le immagini.

Il servizio è stato utilizzato per il caricamento delle immagini profilo degli utenti e di alcuni allegati, all'interno del sistema sono state utilizzate solo immagini come dati multimediali, questo per semplicità (ma è prevista un aggiunta di nuovi *formati* degli allegati, come audio e video).

Una volta caricata la entry all'interno di Storage essa sarà provvista di un *link* per accedere alla risorsa stessa. Il link viene utilizzato per arricchire l'istanza utente del real time database con un campo che conterrà l'**URL** (*Uniform Resource Locator*) dell'immagine (illustrato nella **figura 2.5**).

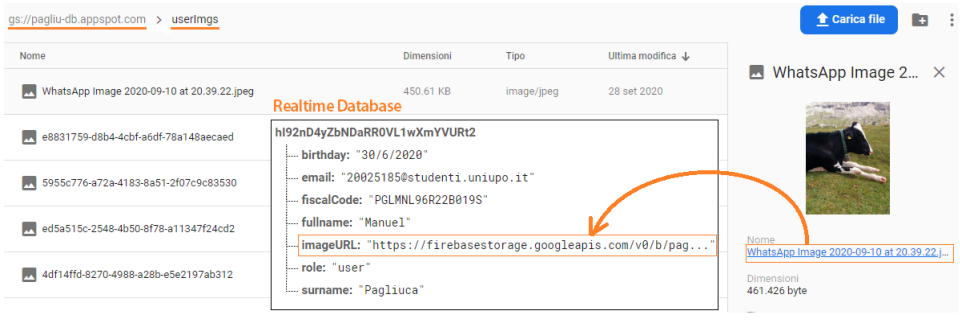

Figura 2.5: Riferimento ad un file su Storage inserito nell'istanza utente del real time database.

2.3 Android Studio

Android Studio[23] è stato sviluppato da **Google** e **JetBrains** e annunciato il 16 Maggio 2013 presso la Google I/O Conference, è basato su **IntelliJ IDEA** (discusso nella sezione successiva), è un **IDE** (Integrated Development Environment) progettato esclusivamente per lo sviluppo di applicazioni Android. Android Studio si è dimostrato un ambiente di sviluppo molto potente che permette all'utente utilizzatore di realizzare qualsiasi task, facilitandogli notevolmente il compito. Questo grazie alle funzionalità come l'auto-completamento o la correzione delle avvertenze (*warnings*) attraverso gli *shortcut*, ma anche grazie ad una solida ed immensa community che sta alla base del software. Per quanto riguarda le motivazioni legate alla scelta del sistema android, verranno discusse successivamente nella sezione 5.1.

2.4 IntelliJ IDEA

Sviluppato e rilasciato dalla **JetBrains** nel 2001 [22], fu uno dei primi *Java IDE* ad offrire una navigazione del codice *avanzata* e capacità di effettuare *refactoring*. Poiché Android Studio è stato costruito sulla base di IntelliJ imparare l'utilizzo di uno dei due software ne facilita notevolmente l'assimilazione dell'altro, considerando che condividono molte caratteristiche.

All'interno dell'IDE è stato integrato uno dei tanti **plugin** disponibili, "*Statistics*" di "*Tomas Topinka*", il quale mostra i dati statistici sui files all'interno del progetto (mostrato in **figura 2.6**).

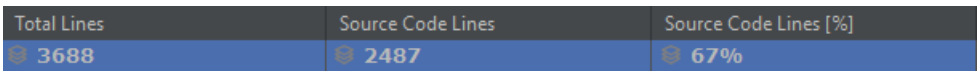

Figura 2.6: Statistiche del progetto sviluppato IntelliJ IDEA.

Source: *Plugin:* Statistics *by Tomas Topinka.*

2.5 Trello

Per la suddivisione e organizzazione dei compiti si è utilizzato il software gestionale **Trello**[28]. Il software permette agli utilizzatori di creare le proprie schede attività con più colonne e scambiare le attività tra di loro, impiegando qualche click del mouse. Le colonne possono essere organizzate a piacere dell'utente, solitamente vengono organizzate come una **Kanban Board** ma si lascia una libera creatività a riguardo. Ogni scheda attività, può contenere all'interno degli ulteriori elementi caratterizzanti come: checklist, scadenze, etichette, allegati o anche rami di **GitHub** (sezione 2.7).

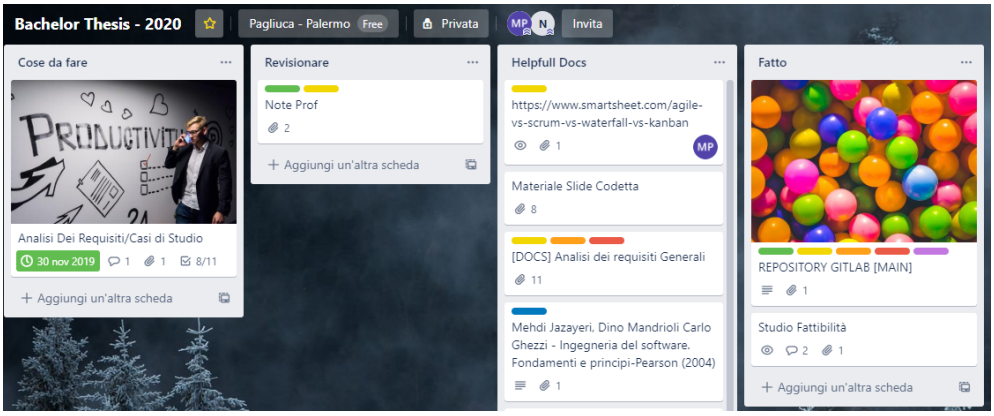

Figura 2.7: La board utilizzata durante la fase di analisi dei requisiti.

Source: *https://trello.com/*

2.6 Visual Paradigm

Visual Paradigm[27] è un software appartenente al dominio **CASE** (Computer-aided software engineering) per la creazione di diagrammi **UML** (Unified Modeling Language). Il software permette di modellare, modificare e creare ben 14 di tipi diversi di diagrammi. Nel nostro caso è stato utilizzato per la creazione del diagramma dei **casi d'uso**.

2.7 Git e GitHub

Per facilitare lo sviluppo del codice dei due applicativi sono stati utilizzati **Git**[2] rinomato **DVCS** (Distributed Version Control System) e la sua implementazione per l'hosting **GitHub**[15]. È stato necessario l'utilizzo di questi strumenti per avere un punto di salvataggio non locale dello stato del progetto (detto *snapshot*). Ci sono molti altri punti a favore di Git per essere preferito rispetto ad altri **VCS**, ne cito solo uno per non perdermi in eccessiveargomentazioni : l'istantaneità delle operazioni. I comandi della shell di Git sono istantanei, compresi i comandi di *branching* e *merging*.

```
Manuel@DESKTOP-R01Q64I MINGW64 ~/Desktop/Github/Uni/FIX-IT-Pagliuca-Employee (master)
$ git push
Enumerating objects: 82, done.
Counting objects: 100% (82/82), done.
Delta compression using up to 8 threads
Compressing objects: 100% (43/43), done.
Writing objects: 100% (45/45), 33.69 KiB | 1.98 MiB/s, done.
Total 45 (delta 28), reused 0 (delta 0)
remote: Resolving deltas: 100% (28/28), completed with 26 local objects.
To https://github.com/20025185/FIX-IT-Pagliuca-Employee.git
   0b4e432..c72333b  master -> master
```

Figura 2.8: Git bash, ultima push.

2.8 Docker

Docker[8] è un progetto *open-source* nato il 13 Maggio del 2013 che permette l'automatizzazione del deployment di applicazioni all'interno di contenitori software. Per i nostri scopi Docker è stato utilizzato per gestire un Kafka Broker in maniera da fornirne anche una **GUI** (Graphical User Interface, **figura 2.9**) per la visualizzazione delle varie componenti, tra cui i topic. Questo è stato possibile grazie alla docker image per lo sviluppo di Kafka[3].

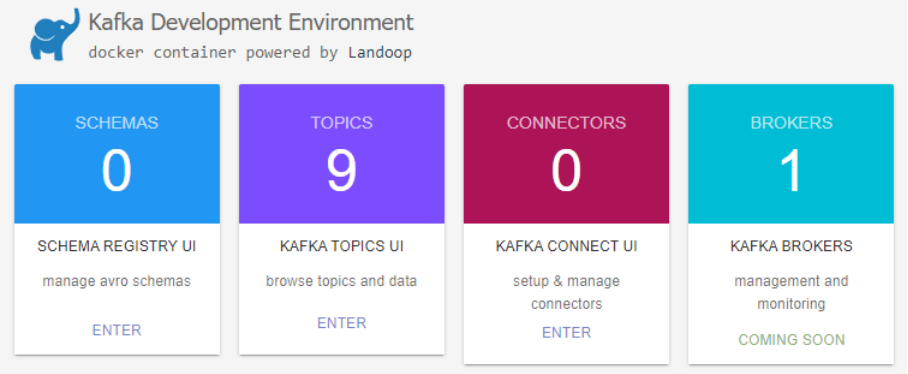

Figura 2.9: GUI di Docker, *127.0.0.1:3030*

2.9 Doxygen

Per la generazione della documentazione del progetto per l'applicativo desktop si è optato per l'utilizzo di **Doxywizard**[32] che non è altro che una **GUI** per la configurazione e l'utilizzo di **Doxygen**[31]. Doxygen è un applicazione per la generazione automatica di documentazione estraendola da i commenti e dalla struttura dati (**figura 2.10**).

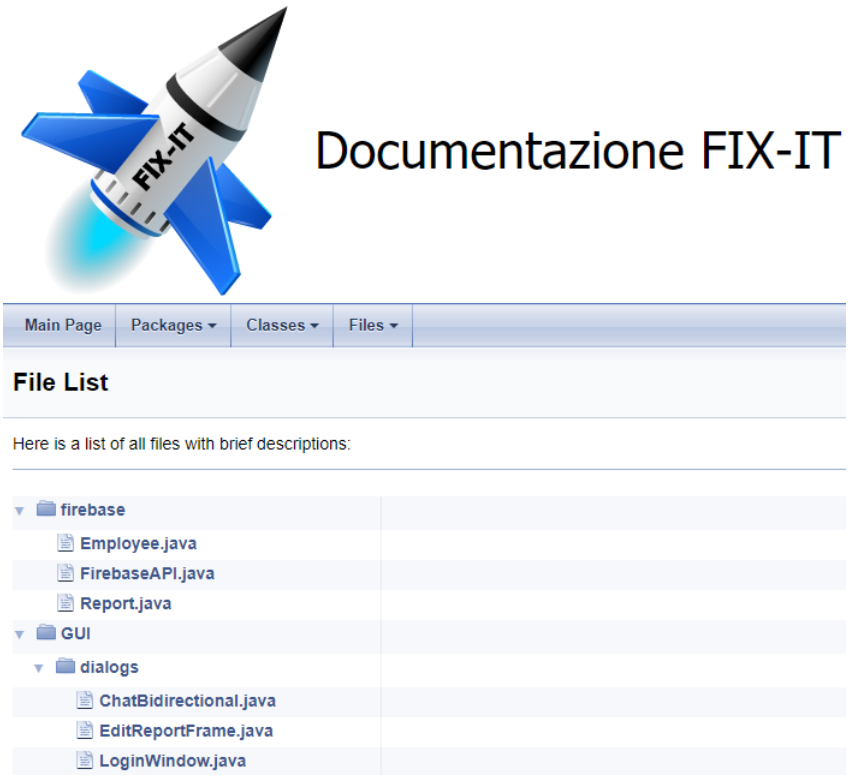

Figura 2.10: Documentazione.

In fase di generazione della documentazione è stato integrato l'utilizzo di **Graphviz**[25] (*Graph Visualization Software*) il quale è un programma per la generazione di grafi, esso verrà utilizzato da Doxygen per la generazione di diagrammi come la gerarchia delle classi (**figura 2.11**).

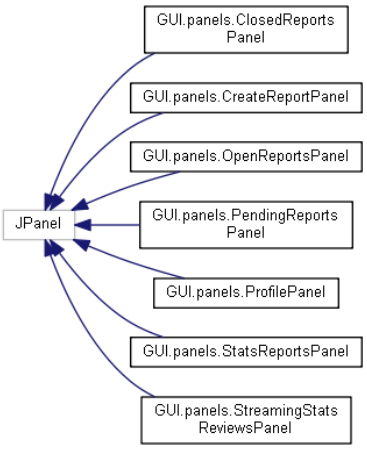

Figura 2.11: Gerarchia dei vari JPanels.

Capitolo 3

Analisi dei requisiti

La prima tappa fondamentale dello studio guidato (nonché del ciclo di sviluppo) è stata ovviamente l'analisi dei requisiti, preceduta da una sessione di **brainstorming** che ha generato un insieme di idee abbozzate grossolanamente riguardanti le funzionalità del sistema. Successivamente l'insieme è stato raffinato scartando quelle poco pertinenti ai fini del progetto, rimanendo con un blocco di idee cardine che hanno conferito una forma ben definita al sistema. Dalle idee rimaste sono stati definiti i **requisiti funzionali** e **non funzionali** che sono presenti nel documento di S.R.S. (**Appendice 6.2**), la cui stesura è iniziata il 7/11/2019 e terminata il 29/11/2019.

3.1 Requisiti funzionali

I requisiti funzionali sono stati pensati in modo da permettere un utilizzo intuitivo dell'applicativo da perte degli utenti, ma rispettando i paletti impostati dai **requisiti non funzionali**.

Essi rappresentano le funzionalità **cardine** pensate durante la fase di brainstorming, le quali danno forma all'intero sistema guidato a eventi. Sono presenti sia i requisiti funzionali per l'applicativo mobile che quelli per l'applicativo desktop. La definizione dei requisiti funzionali elencati in Tabella 3.1 ha fornito un punto da cui partire per lo sviluppo dell'intero progetto.

Requisito	*Titolo*
RF00	Registrazione utenti
RF01	Login utenti
RF02	Visualizzazione satellitare delle segnalazioni
RF03	Dati testuali
RF04	Invio segnalazione a gli impiegati
RF04.1	Controllare stato della propria segnalazione (utente)
RF04.1.1	Chat bidirezionale Utente-Impiegato
RF04.1.2	Riapertura segnalazione
RF04.1.3	Recensione segnalazione
RF04.1.3.1	Statistiche recensioni - Stream Processing
RF05	Statistiche delle segnalazioni nella zona (utente)
RF06	Login impiegati
RF07	Statistiche (impiegati gestionali)
RF08	Handling delle segnalazioni
RF08.1	Fornire feedback all'utente (impiegato)
RF09	Caricamento della segnalazione (impiegato)

Tabella 3.1: Requisiti funzionali.

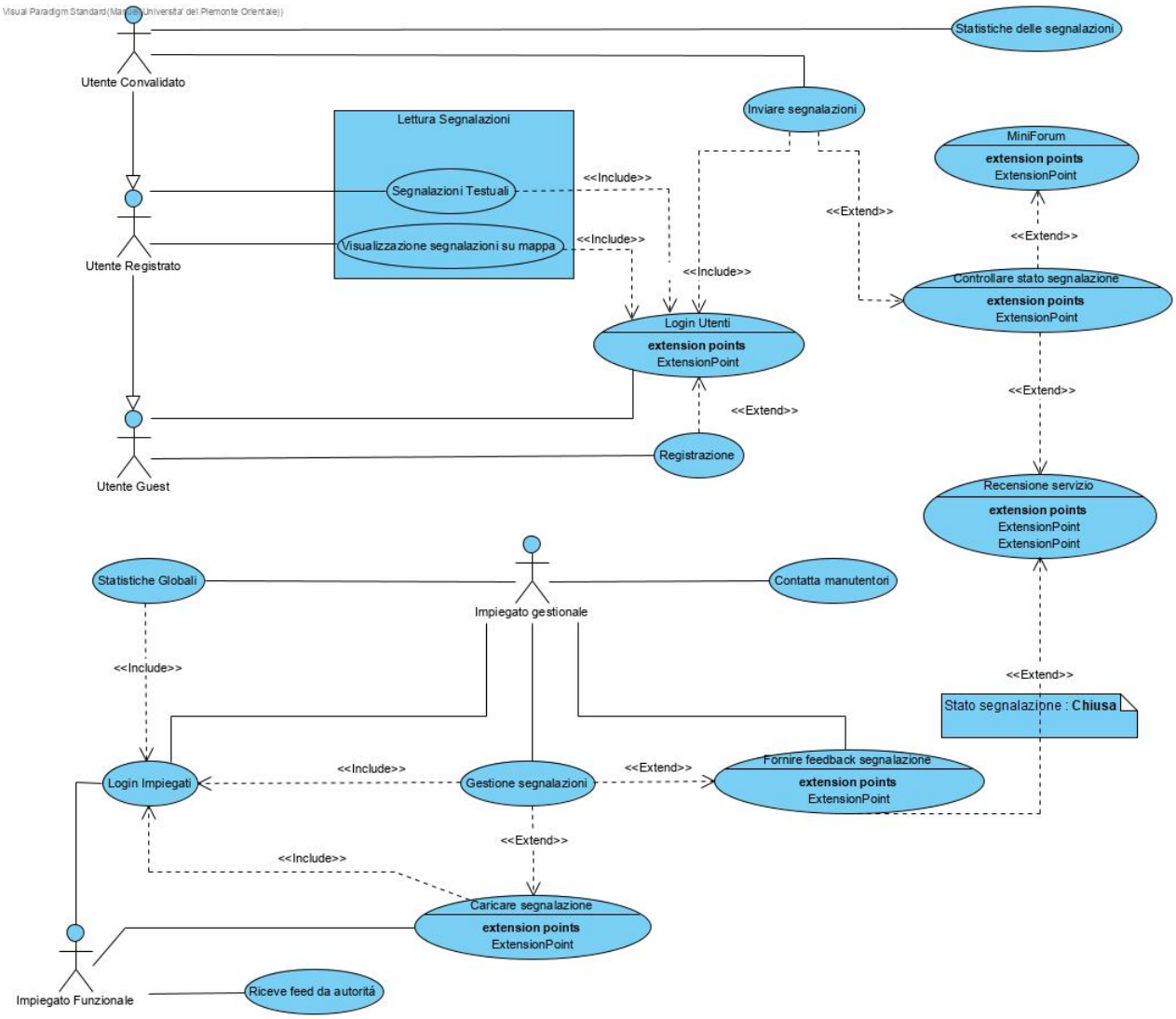

Figura 3.1: Diagramma dei casi d'uso (**U.C.D.** - *Use Case Diagram*).

Il diagramma dei casi d'uso illustrato nella **figura 3.1**, rappresenta i **requisiti funzionali** con un ellisse, mentre per indicare una dipendenza verso un altro requisito viene utilizzata una freccia tratteggiata con apposta la scritta "*<include>*".

La freccia è uscente dai requisiti **dipendenti**, ed è entrante nei requisiti **indipendenti**. Per esempio per utilizzare "*Segnalazioni testuali*" è necessario che *Login utenti* sia soddisfatto poiché il primo requisito dipende dal secondo.

Alcuni requisiti vengono specificati come estensione di altri requisiti già presenti e generici, per esempio "*Controllare stato segnalazione*" estende (o sblocca) "*Recensione servizio*", tale requisito permette di effettuare una recensione inerente al servizio ricevuto.

Gli attori protagonisti sono raffigurati con degli omini stilizzati, essi rap-

presentano le diverse tipologie di utente ed impiegato. Ognuno di questi ha diverse viste e diversi poteri all'interno del sistema.

Per esempio un utente *guest* potrà visualizzare soltanto la finestra di login e la schermata di registrazione, mentre un utente *convalidato* (e loggato, specificato dalla *<include>*) avrà accesso ai requisiti *inviare segnalazioni* e *statistiche delle segnalazioni.*

La differenza sostanziale tra le due tipologie di impiegato risiede nel fatto, che l'impiegato **funzionale** è un impiegato in grado di ricevere dei **feed** da autorità ufficiali e caricarli mediante lo stesso strumento (*carica segnalazione*) dell'**impiegato gestionale**, quest'ultimo ha anche altre compiti riguardanti la gestione del sistema e la comunicazione con gli utenti segnalatori.

3.2 Status e priorità delle segnalazioni

Il ciclo di vita di una segnalazione è stato gestito attraverso un sistema di differenti stati, rappresentati da delle stringhe testuali.
Sono presenti tre **status** differenti per ciascuna segnalazione :

- **Pending** (o *Attesa*), viene impostata automaticamente dal sistema quando una segnalazione viene mandata da un utente.
- **Aperta**, viene impostata manualmente da un impiegato dopo che ha preso visione di una segnalazione in stato di Attesa. Oppure, nel caso in cui sia avvenuta una richiesta di riapertura di una segnalazione precedentemente chiusa.
- **Chiusa**, viene impostata manualmente da un impiegato dopo che una segnalazione è stata risolta.

Le segnalazioni, oltre ad avere l'etichetta per indicare lo status, godono anche di un'altra etichetta che ne descrive la **priorità**.

Lo status indica la condizione (una delle tre possibili) attuale della segnalazione, mentre la priorità indica l'**ordine** di rilevanza delle segnalazione. La priorità delle segnalazioni, viene espressa tramite una delle tre *stringhe* rappresentanti valori numerici :

- 0 $\Longrightarrow$ *Bassa priorità*
- 1 $\Longrightarrow$ *Media priorità*
- 2 $\Longrightarrow$ *Alta priorità*

3.3 Requisiti non funzionali

La scelta dei requisiti non funzionali è stata effettuata ponendosi come obiettivo la realizzazione di un sistema che soddisfi i criteri di **usabilità**, **scalabilità** e **manutenibilità** [14]. In modo da permettere ad un maggior numero di utilizzatori di poter venire in contatto con il prodotto, e di permettere a gli sviluppatori di mettere mano sul progetto senza perdere troppo tempo a cercare la porzione di codice interessata.

La *manutenibilità* è stata ottenuta suddividendo il progetto in moduli, e commentando tutte le funzionalità che esso dispone. In maniera che sia possibile generare la documentazione che sarà il manuale di riferimento sviluppatori (gli strumenti per la generazione della documentazione sono trattati nella sezione 2.9).

Requisito	*Descrizione*
RNF00	L'interfaccia utente sarà grafica e testuale.
RNF01	L'interfaccia utente sarà **"user-friendly"**, la chiarezza e semplicità permetteranno un utilizzo intuitivo dell'applicazione anche agli utenti appena registrati.
RNF02	La visualizzazione dell'interfaccia, e quindi l'accesso ai servizi annessi dipenderà dalla tipologia dell'utente. Tipologie: • Utente guest (può visualizzare soltanto la finestra di login) • Utente registrato (utente registrato all'interno del sistema ma non gode di pieni poteri, può accedere ad un numero limitato di funzionalità. Ovvero *Dati testuali* e *Visualizzazione satellitare delle segnalazioni*) • Utente convalidato (In seguito ad un attivazione nel sistema, l'utente registrato sarà in grado di accedere a tutti gli strumenti offerti dal sistema)
RNF03	L'applicazione utilizzerà una porzione di banda diversa in base alla richiesta effettuata dall'utente.
RNF04	L'applicazione dovrà parallelizzare le molteplici richieste effettuate dagli utenti.
RNF05	L'applicazione mobile sarà sviluppata su Android in modo di venire in contatto con un pubblico maggiore.

Tabella 3.2: Requisiti non funzionali.

3.4 Suddivisione del lavoro

Per la suddivisione del lavoro di gruppo, abbiamo deciso di adottare una strategia proposta dal Prof. Cerotti, nella quale i vari requisiti funzionali sono stati spartiti tra me ed il mio collega in modo da ottenere un carico di lavoro proporzionato per entrambi, ed anche di poter lavorare nel campo di maggiore interesse, nel mio caso è stato lo **stream processing**.

Visto la dimensione del gruppo molto ristretta e contando l'emergenza sanitaria in corso, risultava eccessivo l'utilizzo di una metodologia **Agile** nel sua interezza. Si è quindi optato per l'utilizzo dei concetti cardine del *manifesto Agile* [1] in combinazione con l'elevata *flessibilità* che un gruppo dal personale ridotto può offrire.

Fondamentale fin dalle prime fasi di analisi è stato l'utilizzo di *Trello* (spiegato nel dettaglio alla sezione 2.5) per la gestione e organizzazione del lavoro [26].

Prima di procedere con la costruzione dell'intera applicazione desktop, si è optato per terminare l'implementazione dei requisiti funzionali inerenti all'applicazione Android (sezione 5.1), poiché i restanti requisiti dell'applicazione mobile interagiscono con l'applicazione remota. I requisiti funzionali di cui mi sono occupato personalmente sono evidenziati in grassetto nella **tabella 3.1**.

Capitolo 4

Progettazione

4.1 Panoramica dell'architettura

L'architettura dell'intero sistema viene esposta nel diagramma di flusso informativo in **figura 4.1**.
L'architettura è composta dai seguenti elementi:

- uno o più **dispositivi mobile android** dai quali gli utenti possono inviare segnalazioni o ricevere aggiornamenti.
- la piattaforma **Firebase** nella quale vengono memorizzate le segnalazioni all'interno **real time database** garantendone la persistenza dei dati.
- l'**applicativo desktop** che fornisce una GUI attraverso la quale l'impiegato potrà in maniera amministrativa gestire le segnalazioni del sistema.
- Infine il **server** Apache Kafka fa da intermediario nella comunicazione tra i dispositivi mobili degli utenti e l'applicazione desktop dell'impiegato, processando in **real time** determinate tipologie di segnalazioni.

L'interazione tra i dispositivi mobile ed il *broker* avviene mediante l'utilizzo delle **REST API**(Representational State Transfer API, il loro utilizzo viene illustrato nella sezione 4.2) rilasciate da Apache Kafka . L'applicazione desktop fornisce una **GUI** (*Graphical User Interface*) su cui l'impiegato possa interagire per effettuare operazioni di amministrazione all'interno del sistema. Quest'ultima è collegata con la piattaforma di Firebase tramite il Firebase **Admin SDK**[16] (*Software Development Kit*) per eseguire operazioni privilegiate, e con le REST API (di Firebase Authentication) per effettuare l'operazione di login nel sistema. Il database di Firebase si occupa principalmente di gestire tutti i dati che danno forma al sistema, ovvero : *utenti*, *impiegati* e *segnalazioni*.

I dati non devono per forza passare da entrambe le piattaforme per raggiungere il destinatario. Le segnalazioni vengono memorizzate ed organizzate sul real time database (tecnologia per i Big Data, sottosezione 2.2.1) esso è una delle componenti fondamentali su cui si basa l'intero sistema.

Il server di Apache Kafka (tecnologia "Fast Data", descritte nell'introduzione 1) si occupa anch'esso di organizzare le segnalazioni ricevute dai dispositivi mobili, ma queste ultime al posto di essere posizionate all'interno di un database vengono posizionate all'interno dei **topic** (sezione 2.1.2), le segnalazioni presenti nei topic vengono ricevute dai dispositivi e passate all'applicazione remota da un componente, il **broker** (il funzionamento della piattaforma viene descritto ampiamente nella sezione 2.1.1).

Fondamentali nella realizzazione del sistema e di grande aiuto per l'accesso alle piattaforme esterne sono state le REST API (descritte nella sezione successiva 4.2) che hanno reso estremamente più leggero l'utilizzo delle risorse per l'applicativo Android ed hanno assicurato un accesso alle funzionalità del sistema anche nelle situazioni dove un operazione banale non era così semplice da implementare. Parte dell'implementazione delle REST API verrà descritta nella sottosezione 5.2.1, mentre verranno discusse in maniera astratta nella sezione successive.

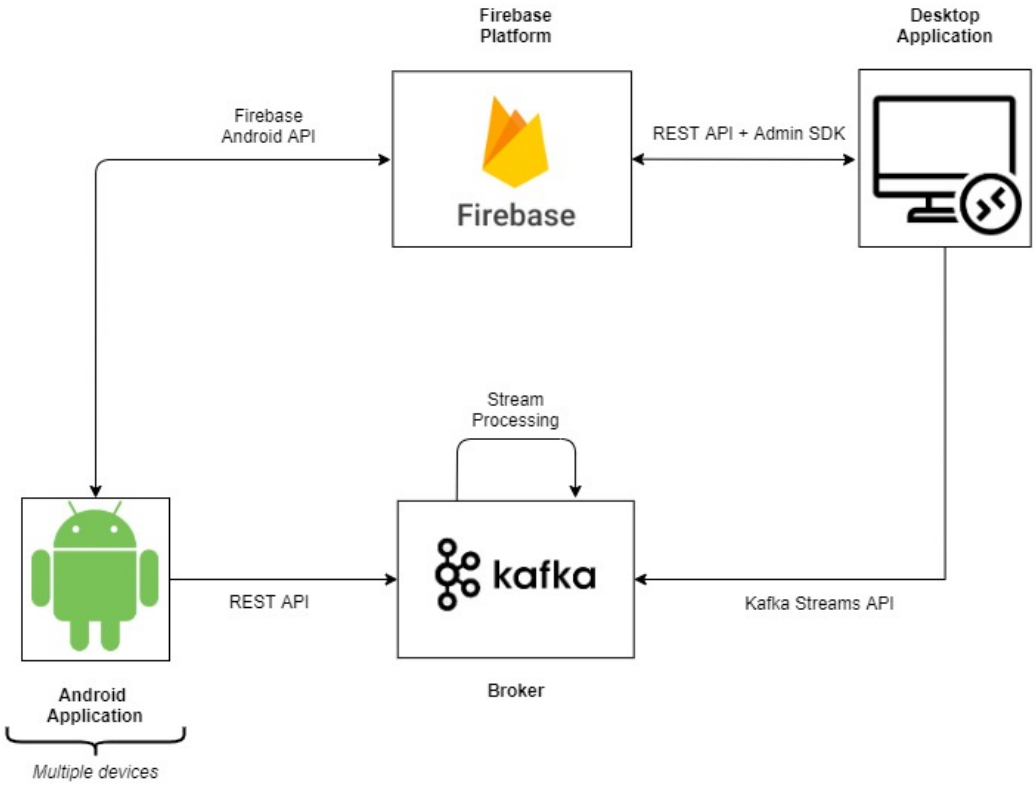

Figura 4.1: Architettura FIX-IT, flusso di informazioni.

Apache Kafka viene utilizzato per trattare determinate segnalazioni che necessitano di elaborazione dei dati in realtime. I topic utilizzati per effettuare l'elaborazioni sono tre : *input-ratings*, *fav-issues-filter* e *count-fav-issues*.

Il primo topic ha lo scopo di prelevare le *recensioni* inviate dai dispositivi mobile degli utenti.

Successivamente le recensioni vengono *rielaborate* e *selezionate* in base ad un criterio imposto nel sistema, spostando quel che ne rimane all'interno del secondo topic (*fav-issues-filter*).

Infine viene effettuata un'ulteriore elaborazione e salvati nell'ultimo topic *count-fav-issues* (le seguenti operazioni vengono descritte dettagliatamente nella sottosezione 5.2.1).

Firebase è anch'esso un servizio realtime ma non offre nessuna possibilità riguardo l'elaborazione dei flussi di dati, per questo viene utilizzato Apache Kafka che a suo modo (descritto nella sezione 2.1) fa da *tramite* tra utente (colui che effettua la segnalazione) ed impiegato (colui che la gestisce) di *determinate segnalazioni* che necessitano di elaborazioni in tempo reale.

Avere due piattaforme differenti dove poter veicolare le stesse informazioni aiuta ad aumentare l'**affidabilità** dell'intero sistema (*per esempio*, se dovessi perdere i dati presenti su Kafka li potrò recuperare dal database di Firebase) e la **persistenza** dei dati (*per esempio*, se il mio applicativo mobile dovesse andare incontro ad un crash mentre sto effettuando delle operazioni delicate, i dati saranno salvati in qualche maniera su Firebase).

In maniera astratta possiamo dire che all'interno dell'architettura del sistema il ruolo dei *producers* e *consumers* viene ricoperto rispettivamente dai dispositivi mobile e dalle applicazioni impiegato. Anche se in realtà essi sono integrati all'interno delle applicazioni. Le applicazioni Android grazie all'ausilio delle **REST API**, invieranno le informazioni (messaggi, eventi o record) come oggetto **JSON** all'interno di una richiesta **HTTP** al **broker** di Apache Kafka (il server).

Il server di Kafka è eseguito su un elaboratore e smisterà i messaggi all'interno dei topic corretti. Il **consumer**, è presente all'interno dell'applicazione desktop degli impiegati tramite le **Consumers API**. Il server di Kafka potrà effettuare delle operazioni sui flussi di dati che transitano tra producer e consumer (argomento trattato nella sottosezione 2.1.3).

Visto che i dispositivi android lavorano sue due piattaforme differenti, essi smisteranno le segnalazioni in maniera **consistente** nelle due locazioni differenti. Questo permette di consolidare l'**integrità dei dati** in caso di *modifiche* o *perdite* accidentali delle segnalazioni.

Su Firebase Database sarà presente la segnalazione stoccata ed è utilizzata come riferimento all'ultima segnalazione fatta. Mentre su Apache Kafka passerà la medesima segnalazione, ma la piattaforma **utilizza** la segnalazione per effettuare elaborazione dei dati. Questo permette di alzare i livelli di **sicurezza** del sistema garantendone una certa **integrità** dei dati, que-

sto perché nel caso in cui si dovesse perdere una segnalazione esiste sempre un'altra locazione da cui reperirla.

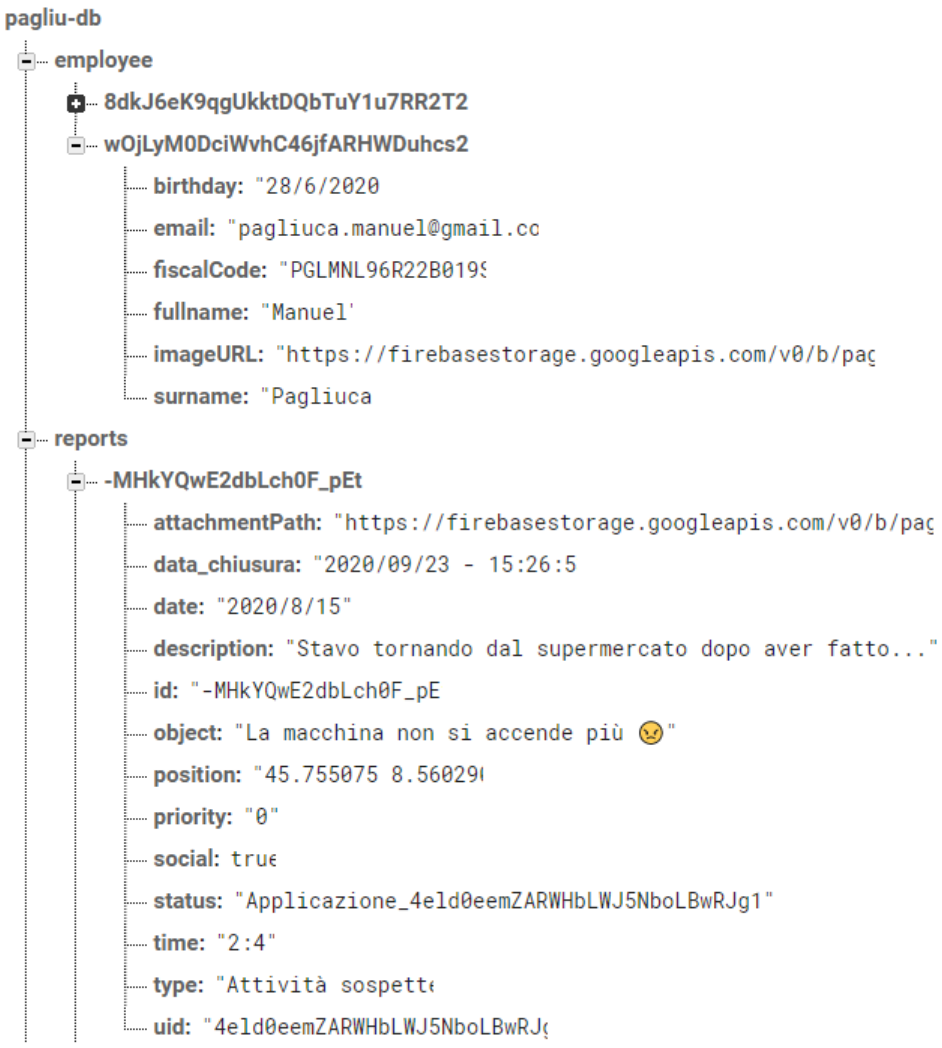

Figura 4.2: Organizzazione interna dei dati nel real time database (account impiegati e reports).

Le informazioni all'interno del database sono organizzate in tre grandi sezioni, la *prima* con nome "*employee*" nel quale sono presenti tutti gli **UID** (User Identifier, vengono generati in maniera casuale ed univoca al momento della registrazione) appartenenti a gli impiegati. Ogni UID contiene le informazioni inerenti all'utente impiegato registrato (in maniera non convenzionale, poiché impiegato) sulla piattaforma.

La *seconda sezione*, tecnicamente simile alla prima, si chiama "*users*" e racchiude tutti gli utenti registrati sulla piattaforma, gli utenti anche in questo caso sono **indicizzati** attraverso un UID. Le informazioni racchiuse in un nodo rappresentante un utente sono : nome, cognome, data di nascita, immagine del profilo (tramite un **link** a Firebase Storage), codice fiscale, email e ruolo (posizione all'interno della piattaforma).

L'*ultima sezione*, nonché la più complessa risulta essere quella relativa alle segnalazioni, dove i singoli nodi sono rappresentanti delle segnalazioni all'interno del sistema, la quali sono rese univoche da un opportuna chiave (generata tramite le API di Google).

Le informazioni contenute all'interno di un nodo rappresentante di una segnalazione sono numerose, nella **figura 4.2** sotto al nodo "*reports*" è possibile visualizzare come sono state rappresentate le informazioni all'interno di Firebase.

4.2 Interfacce

L'architettura utilizza a sua volta dei *moduli* per interfacciarsi con le piattaforme esterne al sistema come Firebase e Apache Kafka.

Per l'interazione tra dispositivi android e la piattaforma di Firebase sono state utilizzate tre API : **Firebase Auth API**[17] per permettere l'autenticazione all'interno della piattaforma Firebase, **Firebase Database API**[18] per la gestione del real time database e **Firebase Storage API**[20] che offre una locazione per lo stoccaggio dei dati.

Mentre per effettuare il **collegamento tra dispositivo android ed il broker Kafka** si è dovuto ricorrere all'utilizzo delle **REST API**. Queste ultime sono state utilizzate anche per il collegamento tra l'applicazione desktop e la componente Firebase Authentication.

Le REST API, acronimo di *Representational State Transfer* (*Application Programming Interface*), definite nella tesi di dottorato di Roy Fielding [10], sono uno stile architetturale per sistemi distribuiti. L'obiettivo delle architetture REST è quello di rappresentare una trasmissione di dati da parte di un sistema soltanto mediante il protocollo **HTTP**. Per un corretto funzionamento la struttura del messaggio prevede un determinato **URL** che si

riferisca ad una determinata risorsa, ed un metodo fra quelli previsti dal protocollo (GET, POST, PUT, ...).
Come annunciato precedentemente all'interno del sistema le REST API sono state utilizzate per effettuare due collegamenti, nel dettaglio :

1. Invio delle segnalazioni dal dispositivo mobile sul **topic** "*input-ratings*" (le operazioni sono descritte nella sottosezione 5.2.1 inerente al **RF04.1.3.1**). L'API utilizzata si chiama **REST Proxy API v2**[6], essa permette di inviare messaggi effettuando richieste POST a dei topic specifici.

2. Effettuare il login dell'applicazione desktop mediante la componente Firebase Authentication. L'API in questione si chiama **Firebase Auth REST API**[19], effettuando una richiesta POST si riceverà in risposta un *payload* contenente alcune informazioni sensibili dell'utente (i dettagli sono discussi nella sottosezione 5.2.3).

Per quanto riguarda la connessione tra l'applicazione desktop ed il real time database di Firebase si sono utilizzate le **Firebase Admin SDK API** [16]. Un particolare pacchetto di API che permette di operare come amministratore della piattaforma, permettendo l'utilizzo di funzioni privilegiate che le API "*normali*" di Firebase non dispongono.

Infine, per effettuare la lettura dei records di un topic da parte dall'applicativo desktop, è necessaria la presenza di un *consumer*. Per questo, sono state utilizzate **Kafka Consumer API**[11].

In relazione alla gestione dei flussi di dati che attraversano il broker, sono state coinvolte delle API apposite, le **Kafka Streams API**[13]. La teoria a riguardo è stata discussa nella sottosezione 2.1.3, l'implementazione verrà affrontata nella sottosezione 5.2.1 del capitolo implementativo.

Capitolo 5

Implementazione del sistema

5.1 Applicazione Android

L'applicazione mobile è un applicazione Android, la scelta è stata intrapresa basandosi in gran parte sulla sua distribuzione open-source, ma anche sul potente IDE quale è **Android Studio**.

L'attività iniziale dell'applicazione prevede una schermata di login dove l'utente registrato può inserire email e password (**figura 5.1**), una volta data conferma il sistema farà accedere l'utente alla schermata del profilo. Nel caso l'utente non fosse registrato dovrà compilare un form di registrazione con le sue credenziali (per ulteriori informazioni visionare Appendice 6.2).

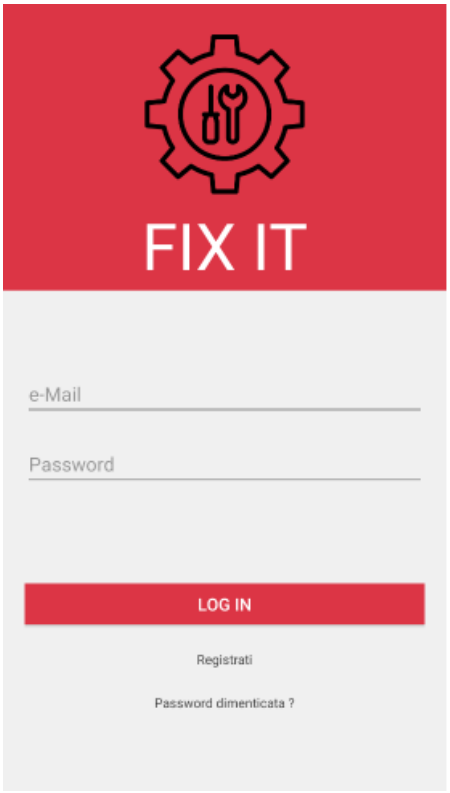

Figura 5.1: Schermata di login.

Source: *Android Studio*

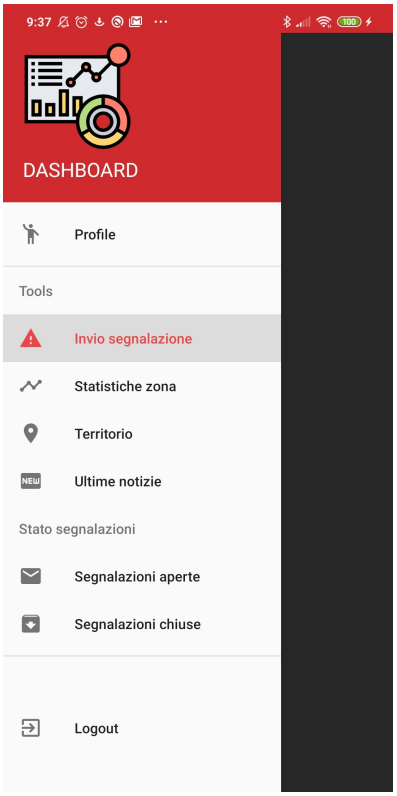

Figura 5.2: Dashboard.

Source: *Xiaomi Redmi Note 7*

Considerando il caso in cui l'utente abbia superato la schermata di accesso con successo, si visualizzerà la schermata del profilo utente, la cui mostrerà i dati dell'utente (prelevati dal real time database) con annessa l'immagine profilo.

Dalla schermata di profilo è possibile accedere alla **Dashboard** facendo *slide* da sinistra verso destra. Nella Dashboard sono presenti le opzioni elencate in **figura 5.2**. Ognuna di queste opzioni corrisponde ad uno strumento sul quale è stato sviluppato un requisito funzionale. Nelle prossime sezioni il funzionamento e l'implementazione di tali requisiti verrà esaminato nel dettaglio.

5.1.1 Implementazione della mappa

Il requisito funzionale **RF02** intende fornire una rappresentazione grafica ed interattiva delle segnalazioni nelle località.

La schermata dispone di una *Searchview*, ovvero una casella di input testuale nella quale è possibile cercare il luogo desiderato, e di un frame per la visualizzazione della mappa. Il sistema preleverà le segnalazioni dal real

time database e le caricherà all'interno di un array, successivamente itererà sull'array prelevando la latitudine e longitudine delle segnalazioni e la loro priorità, in base a queste informazioni verranno generati dei marker sulla mappa (utilizzando le API di *Googlemap*).

Per l'implementazione di questo requisito sono stati utilizzate tre tipologie di marker (corrispondenti alla priorità che una segnalazione può assumere descritte nella sezione 3.2).

- **Verde** $\Longrightarrow$ *Bassa priorità*
- **Giallo** $\Longrightarrow$ *Media priorità*
- **Rosso** $\Longrightarrow$ *Alta priorità*

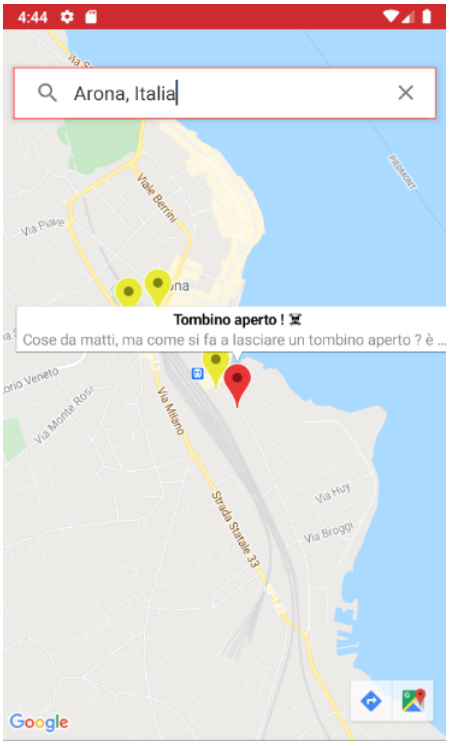

Figura 5.3: Visualizzazione satellitare delle segnalazioni.

Source: *Screenshot Pixel 2 API 28, emulato su Android Studio*

Al momento di generazione del marker vengono caricati su di essi anche il *titolo* e *oggetto* della segnalazione che rappresentano. Quindi, all'avvio del tool, prima di ricevere un qualsiasi input da parte dell'utente i vari **marker** risulteranno già presenti sulla mappa.

Durante l'atto di *Submit* della location (contenuta nella Searchview), si procederà per la ricerca del luogo digitato, e soltanto in caso affermativo la camera verrà posizionata sul luogo.

Per realizzare ciò si è utilizzato un oggetto di tipo *Geocoder* che restituisce una lista di indirizzi (ridotta con ad un singolo elemento) data una stringa contenente la posizione cercata.

Un altra caratteristica interessante, se l'utente dovesse uscire dal tool, allora il tool stesso salverà l'ultima posizione ricercata (quest'ultima caratteristica **non è persistente**, se si dovesse riavviare l'applicazione la posizione iniziale sarebbe ancora quella di *default*) come favorita, in maniera che alla riapertura della mappa la *camera* si posizioni sull'ultimo luogo cercato nell'istanza (del tool) precedente.

5.1.2 Dati testuali

Sempre dalla dashboard è possibile accedere alla sezione "**Ultime notizie**" la quale contiene una visualizzazione delle segnalazioni come dati testuali, le segnalazioni stesse vengono mostrate in ordine cronologico (quelle più in alto sono le più recenti, vedere **figura 5.4**).

Per la realizzazione di tale requisito si è utilizzato un *RecyclerView*, il quale è una versione più avanzata delle classiche *ListView*.

Questo **widget** permette di rappresentare una collezione di oggetti come lista, ogni oggetto è una vista dell'interfaccia RecyclerView, ed è un oggetto di tipo *ViewHolder*.

Ogni ViewHolder è in carica di rappresentare un singolo oggetto, mentre è compito dell'interfaccia RecyclerView di gestire dinamicamente la quantità di ViewHolder da mostrare sullo schermo.

Figura 5.4: Visualizzazione delle segnalazioni come dati testuali.

Source: *Screenshot Pixel 2 API 28, emulato su Android Studio*

5.1.3 Recensione segnalazioni

Una volta che la segnalazione viene risolta il suo **status** viene cambiato in "**Chiusa**" (le informazioni su gli status per la gestione dei record verranno poi trattati nella sezione 5.3.1). Le segnalazioni chiuse sono raggiungibili dal tool "**Segnalazioni Chiuse**" presente sulla dashboard, questo' ultimo contiene una RecyclerView dove tutti gli oggetti ViewHolder rappresentano le segnalazioni chiuse effettuate dall'utente.
Ogni uno di questi ViewHolder offre due sotto servizi (utilizzabili interagendo con due bottoni) :

1. Dare una recensione del servizio (illustrato nella **figura 5.5**, questo sotto servizio sarà poi utilizzato come *input* per lo stream processing, discussa nella sottosezione 5.2.1)

2. Effettuare una richiesta di riapertura della segnalazione (trattata nella sezione 5.3.2)

Il primo sotto servizio, (che sarebbe il **RF04.1.3**) è composto da :

- *RatingBar* che offre all'utente la possibilità di esprimere un voto da 0 a 5 (con uno stepsize di 0.5)
- Due *TextView*, una per porre la domanda di come ci si è trovati con il servizio, ed una per notificare che la votazione è andata a buon fine
- Un *Button*, per inviare il voto attraverso una funzione di Callback

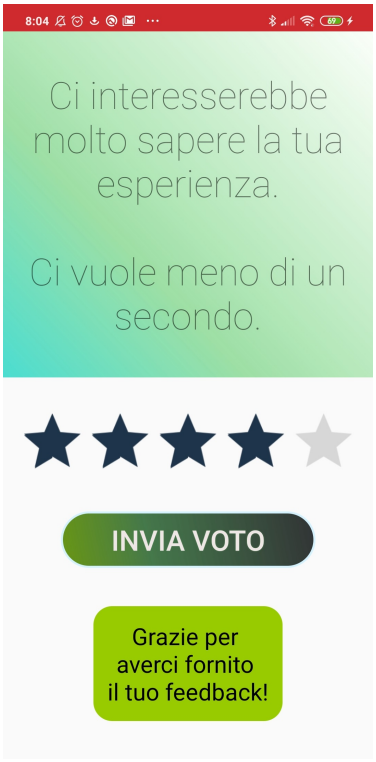

Figura 5.5: Recensione di una segnalazione chiusa.

Source: *Screenshot preso da Xiaomi Redmi Note 7*

Alla chiamata della funzione di Callback per merito del Listener del bottone, il valore contenuto nella *RatingBar* viene convertito in stringa ed **aggiunto** come un nuovo campo all'istanza rappresentante la segnalazione ormai chiusa.

5.2 Applicazione desktop

Il frame iniziale dell'applicazione desktop segue il passo dell'applicazione mobile, ma in questa occasione l'operazione di login risulterà possibile solo per gli utenti che ricoprono il ruolo di impiegato.

Inoltre, non sarà presente nessuna finestra per effettuare una registrazione nel sistema come utente impiegato (spiegato nel documento S.R.S. **Appendice 6.2**), l'applicazione ha come punto d'ingresso soltanto la finestra di login.

Una volta superata con successo la fase di login, il programma presenta un nuovo frame contenente una barra degli strumenti per il monitoring e di amministrazione degli impiegati. La prima schermata mostrata è quella del profilo impiegato, contenente informazioni a riguardo dell'utente.

Gli strumenti disponibili sulla barra :

1. **Pending list**, visualizzare le segnalazioni in stato di attesa
2. **Aperte**, visualizzare le segnalazioni aperte
3. **Chiuse**, visualizzare le segnalazioni chiuse
4. **Statistiche - Recensioni in streaming**, visualizzazione delle statistiche riguardanti le recensioni in real-time.
5. **Statistiche - Segnalazioni**, visualizzazione delle statistiche per le segnalazioni (in generale)
6. **Invia segnalazione**, permette di creare una segnalazione sul database, potendo impostare direttamente le informazioni sensibili

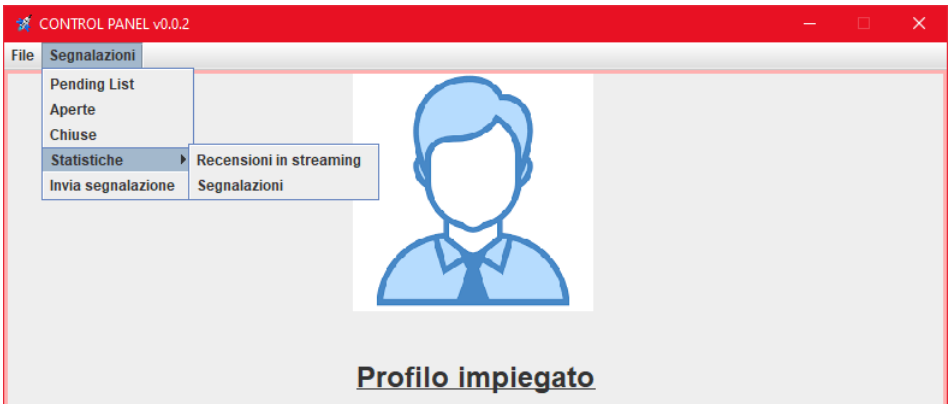

Figura 5.6: Barra degli strumenti dell'applicazione desktop.

5.2.1 Realizzazione dello stream processing

L'obiettivo del requisito **RF04.1.3.1** consiste nel ricevere in ingresso tutte le recensioni delle segnalazioni chiuse (sezione 5.1.3) che hanno una ricevuto una valutazione ≥ 4.0, smistarle in base alla loro **tipologia** e poi contare

quante sono per ogni gruppo. Questa serie di operazioni viene effettuata in real time, attraverso l'elaborazione di flussi di dati tramite Kafka Streams (sezione 2.1.3).

Il requisito ha una prima parte di invio delle segnalazioni su un **topic** che viene effettuata da mobile, mentre la parte di stream processing viene gestita dal client remoto per l'impiegato.
Alla pressione del bottone per inviare il voto, il dispositivo mobile invierà l'**intera** segnalazione all'interno di un topic chiamato *input-ratings* come JSON (**figura 5.7**). Tale JSON avrà come chiave l'identificatore della segnalazione (che viene assegnato dal sistema) e come valore l'intera segnalazione (tutti i campi, eccetto l'identificatore che viene spostato sulla chiave).

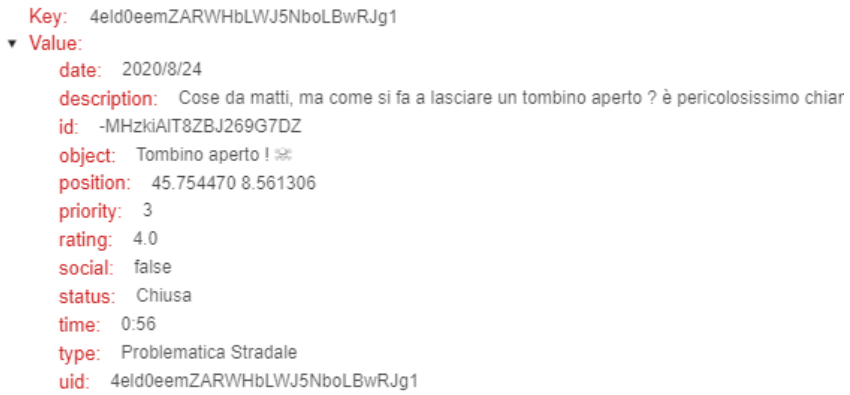

Figura 5.7: JSON di una segnalazione nel topic *input-ratings*.

Per realizzare l'invio di questa segnalazione non è consigliabile utilizzare le **Producer API**[12] di Apache Kafka sopra un dispositivo Android, poichè, oltre alla difficoltà nel fare funzionare la libreria (ideata per applicazioni desktop) sopra Android Studio, l'aggiunta di un thread per la realizzazione di un producer renderebbe più pesante l'intero applicativo. Come soluzione sono state utilizzate le **REST Proxy API v2**[6] le quali hanno permesso alle applicazioni mobile di svolgere l'attività dei *producers* utilizzando delle semplici HTTP request.

Per realizzare ciò si è dovuto ricorrere alla libreria **Retrofit**, (**type-safe HTTP client**) che permette di convertire interfacce Java in HTTP API (e viceversa).

Si è realizzata una piccola interfaccia utilizzando due annotazioni di Retrofit (ovvero le stringhe che iniziano con '@'), la prima annotazione permette di eseguire la "**Header Manipulation**", ovvero permettere la configurazione di header statici, mentre la seconda è una "**Request Body**", che ci permette di specificare che tipo di oggetto utilizzare nel corpo della nostra

richiesta HTTP.

Header Manipulation:

- **Content-Type**, indica le proprietà dei dati. In questo caso i dati vengono mandati come una stringa *Base64* codificata. La parte "v2" significa che verranno utilizzate le API di versione 2, "json" sta a significare il formato di serializzazione.
- **Accept**, è una ulteriore specifica (*non* obbligatoria), serve per descrivere nella maniera più dettagliata possibile il formato e la versione dell'informazione.

Request Body:

- Si utilizzerà un metodo per eseguire una richiesta POST che pubblicherà l'oggetto sul topic "*input-ratings*".

```
1  public interface KafkaAPI {
2  @Headers({"Host: com.example.fix_it_pagliu",
3          "Content-Type: application/vnd.kafka.json.v2+json",
4          "Accept: application/vnd.kafka.v2+json,
               application/vnd.kafka+json, application/json"
5  })
6
7  @POST("topics/input-ratings")
8  Call<Void> createPost(@Body KafkaRecords kafkaRecords);
9  }
```

Interfaccia che espone la richiesta POST.

Si utilizzerà l'istruzione "*kafkaAPI = retrofit.create(KafkaAPI.class)*" permetterà all'interfaccia implementata di generare un JSON, l'oggetto *retrofit* è un oggetto appartenente alla libreria di Retrofit che viene istanziato dando al suo costruttore l'indirizzo del broker di Kafka.

Successivamente, viene popolata un lista di Records che verrà caricata in un altro oggetto di tipo *KafkaRecords* il quale è composto da un solo campo, una **List** di *KafkaSingleRecord*.
La classe è provvista di due annotazioni @SerializedName("records") ed @Expose, la prima serve per rappresentare il campo *recordsList* con la chiave "records", la seconda serve per indicare che il membro verrà esposto per la serializzazione e deserializzazione JSON.

```
1  public class KafkaRecords {
2      @SerializedName("records")
```

```
3       @Expose
4       private List<Records> recordsList = null;
5
6       public void setRecordsList(List<Records> recordsList) {
7           this.recordsList = recordsList;
8       }
9   }
```

Struttura di un KafkaRecords.

Un simile procedimento è ripetuto anche per l'oggetto *KafkaSingleRecord* che è composto dai membri *key* e *report*, rispettivamente di tipo String e Report (rappresentazione della segnalazione presente sul database).

Anch'essi vengono rappresentati ed esposti con le annotazioni @SerializedName ed @Expose, è da tenere presente che quest'oggetto sarà incapsulato all'interno dell'oggetto di tipo KafkaRecords (sostanzialmente è una lista di *KafkaSingleRecord*).

```
1   public class KafkaSingleRecord {
2       @SerializedName("key")
3       @Expose
4       private String key;
5
6       @SerializedName("value")
7       @Expose
8       private Report report;
9
10      public String getKey() {
11          return key;
12      }
13
14      public void setKey(String key) {
15          this.key = key;
16      }
17
18      public Report getReport() {
19          return report;
20      }
21
22      public void setValue(Report report) {
23          this.report = report;
24      }
25  }
```

Struttura di un KafkaSingleRecord.

Alla pressione del pulsante di invio della recensione, verrano caricate le informazioni riguardante la valutazione all'interno di un oggetto *Report* (precaricato con le informazioni della segnalazione).

L'oggetto Report viene mappato sull'oggetto *KafkaSingleReport* che viene a sua volta inserito nell'oggetto lista *KafkaRecords*.

Viene chiamato il metodo "*createPost(kafkaRecords)*" che si occupa di creare la richiesta HTTP POST, restituendo un oggetto di tipo *Call* (il quale è in grado di inviare una richiesta ad un webserver e ricevere una risposta). Sarà il metodo *enqueue()* ad effettuare l'invio della richiesta in maniera **asincrona** e di notificare una risposta alla funzione di Callback.

```
1  dbr.child("rating").setValue(ratingBar.getRating());
2  reportToSend.setRating(String.valueOf(ratingBar.getRating()));
3
4  KafkaRecords kafkaRecords = new KafkaRecords();
5  ArrayList<Record> recordArrayList = new ArrayList<>();
6
7  Record record = new Record();
8
9  record.setKey(uidKey);
10 record.setValue(reportToSend);
11 recordArrayList.add(record);
12
13 kafkaRecords.setRecordList(recordArrayList);
14
15 Call<Void> call = kafkaAPI.createPost(kafkaRecords);
16
17 call.enqueue(new Callback<Void>() {
18     @Override
19     public void onResponse(Call<Void> call, Response<Void>
           response) {
20         Log.d(TAG, response.toString());
21     }
22
23     @Override
24     public void onFailure(Call<Void> call, Throwable t) {
25         Log.d(TAG, t.toString());
26     }
27 });
```

Istruzione eseguite durante l'invio della recensione.

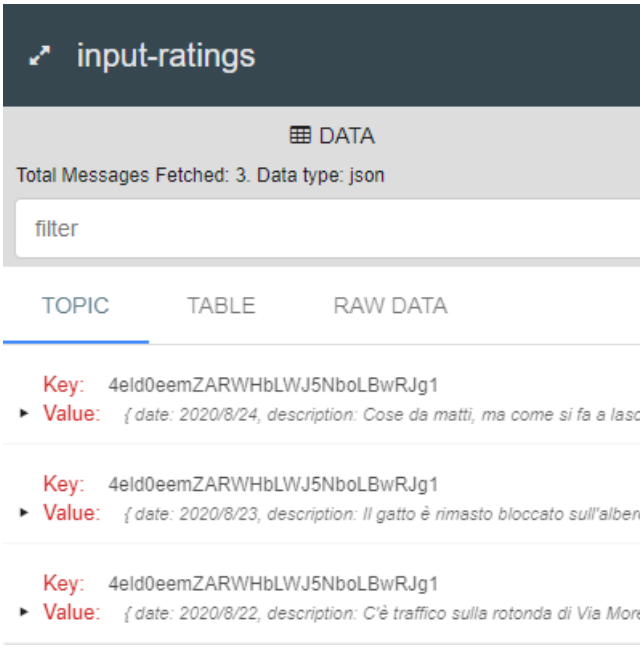

Figura 5.8: Il Topic *input-ratings* dopo l'invio di tre recensioni.

Spostandoci sul lato impiegato, prima di effettuare la parte di analisi statistica occorre che venga effettuato lo **stream processing** per la raccolta dei dati.

In **figura 5.9** è possibile visualizzare le due **sotto-topologie** (descritte nella sottosezione 2.1.3) di processamento con cui vengono analizzati i dati. Essendo delle topologie dirette, senza diramazioni particolari, possiamo paragonarle ad una catena di montaggio (dove ogni **nodo di processamento** è una fase della catena di montaggio).

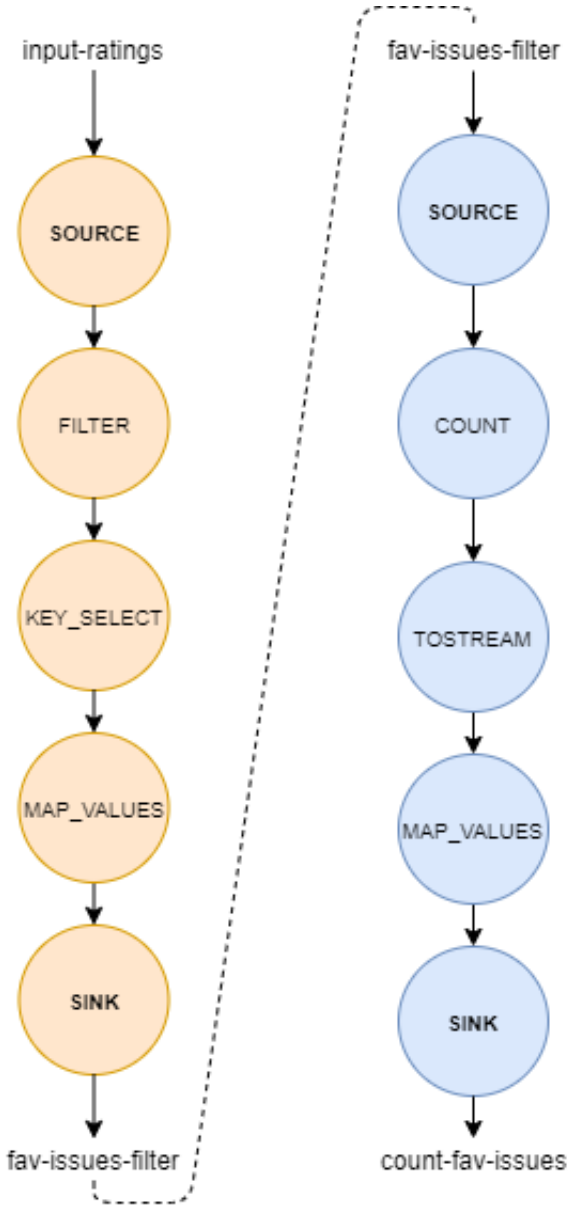

Figura 5.9: Schema di processamento delle due **sotto-topologie** interne allo stream.

Lo **stream processing** viene gestito tramite una classe *FixItStream* la quale è in grado di gestire diversi stream, ma per rimanere essenziali tratterò solo del metodo che si occupa del requisito interessato. Ogni linea di codice del metodo corrisponde ad un punto dell'elenco e ad un nodo di processamento della **figura 5.9**.

Il metodo *input_ ratingsStreming()* (**figura 5.10**) :

1. Si mette in ascolto del topic *input-ratings* restituendo un KStream di nome *inputStream* (**linea 2**).

2. Viene eseguita un operazione di **filtraggio** delle recensioni utilizzando la funzione di Callback *checkRating(value)*, nella quale vengono fatte passare soltanto le segnalazioni con un rating ≥ 4.0 (**linea 5**).

3. Viene eseguito un mappaggio delle **chiavi** di ogni record utilizzando la funzione di Callback *retrieveIssueType()*, nella quale viene recuperato un **tag** della tipologia inerente alla segnalazione. Per un ulteriore sicurezza sulla correttezza del tag vengono rimossi gli spazi e rese minuscole tutte le lettere (**linea 6**).

4. Viene eseguito un mappaggio dei **valori**, inserendo il valore della rispettiva recensione presente nel record (**linea 7**).

5. Le operazioni dalla 2 alla 3 vengono eseguite in serie restituendo al termine un nuovo KStream di nome *favStream* (contenente il changelog delle recensioni favorevoli).

6. *favStream* viene *materializzato* nel topic *fav-issues-filter* (questo perchè è un topic che potrebbe tornare utile per altre elaborazioni di stream), 5.11 (**linea 9**).

7. Ci si mette in ascolto del topic *fav-issues-filter* con un nuovo oggetto KStream, *favCountTopics* (**linea 11**).

8. Viene effettuato un raggruppamento per chiave (che al punto 3 sono state rimappate), quello che si ottiene è un oggetto di tipo KGrouped-Stream (rappresenta un gruppo di stream)(**linea 14**).

9. Si effettua il conteggio dei record con la stessa chiave restituendo una KTable (si utilizza anche uno **StateStore**, esso è un motore di archiviazione per la gestione dello stato mantenuto dallo **stream processor**) (**linea 15**).

10. La KTable ottenuta al passo precedente viene convertita in una KStream tramite il metodo *toStream()* (**linea 16**).

11. Si effettua un mappaggio dei valori nei record, questa volta è semplicemente *conversione* del valore nella coppia *chiave-valore* in una stringa (questo perchè il metodo *.count()* per default restituisce una KTable che ha come valore una variabile di tipo Long, ma noi vogliamo produrre record che siano descritto da una coppia di stringhe) (**linea 17**).

12. Lo stream viene materializzato nel topic *count-fav-issues* (**linea 18**).

Notare : il punto *5* **non** è presente nella **figura 5.9** perché è un operazione che avviene prettamente sul codice. Mentre il punto *6* è considerabile all'interno della medesima figura come la **linea tratteggiata**, la quale sta a significare che il risultato del processamento della prima topologia termina nel "*fav-issues-filter*" e quello stesso contenuto viene poi utilizzato come input della seconda tipologia.

```
1  private void input_ratingsStreaming() {
2      KStream<String, String> inputStream =
           builder.stream("input-ratings");
3
4      KStream<String, String> favStream = inputStream
5              .filter((key, value) -> checkRating(value))
6              .selectKey((key, value) ->
                   retrieveIssueType(value).replaceAll(" ",
                   "").toLowerCase())
7              .mapValues((key, value) -> getRating(value));
8
9      favStream.to("fav-issues-filter",
           Produced.with(Serdes.String(), Serdes.String()));
10
11     KStream<String, String> favCountTopics =
           builder.stream("fav-issues-filter");
12
13     favCountTopics
14             .groupByKey()
15             .count(Materialized.as("count-store"))
16             .toStream()
17             .mapValues((key, value) -> Long.toString(value))
18             .to("count-fav-issues", Produced.with(Serdes.String(),
                   Serdes.String()));
19
20     final Topology topology = builder.build();
21     System.out.println(topology.describe());
22
23     KafkaStreams streams = new KafkaStreams(topology, config);
24
25     streams.start();
26     streams.localThreadsMetadata().forEach(data ->
           System.out.println());
27     Runtime.getRuntime().addShutdownHook(new
           Thread(streams::close));
28 }
```

Figura 5.10: Porzione di codice dello stream processing.

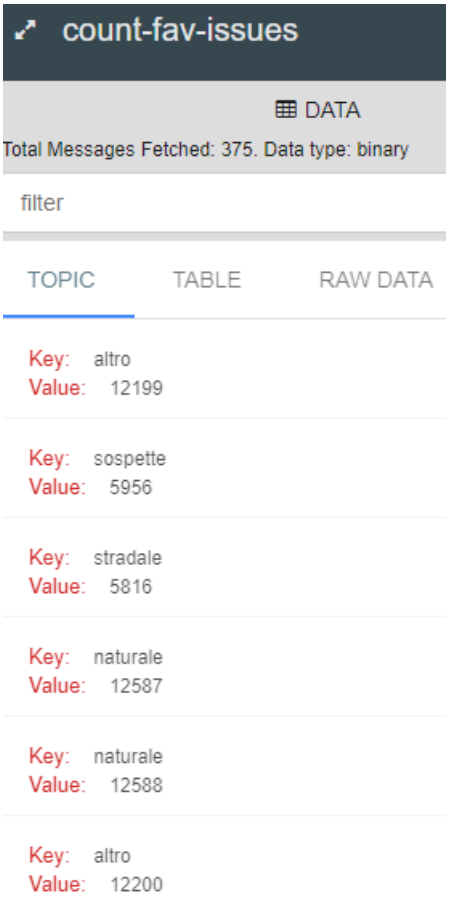

Figura 5.11: Contenuto del topic *count-fav-issues* dopo l'esecuzione di una simulazione.

Le fasi del processamento terminano alla linea 18, dopo questo lo stream viene avviato con l'utilizzo del comando *streams.start()* (viene aggiunto anche uno **ShutDownHook** che richiama il metodo *close()* della libreria di KafkaStrams).
Una volta realizzato lo **stream processing** si può accedere con estrema facilità ai dati elaborati leggendo dal topic "*count-fav-issues*". Per il completamento del requisito, i dati vengono inseriti all'interno di un grafico aggiornato dinamicamente, il quale restituisce una visualizzazione grafica dei dati in real-time.

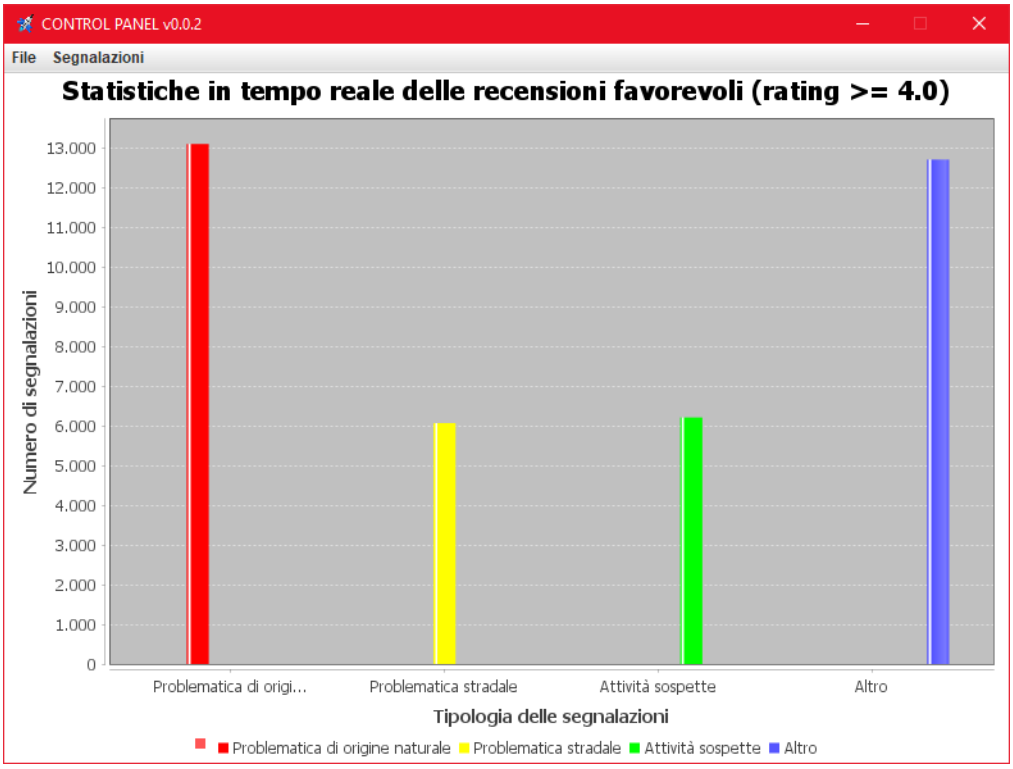

Figura 5.12: Grafico in *real-time* dei dati elaborati dallo stream-processing (topic "*count-fav-issues*").

5.2.2 Statistiche per gli impiegati

Secondo il **RF07**, gli impiegati gestionali dovranno avere a loro disposizione una visualizzazione statistica (senza *stream-processing*) delle segnalazioni riguardanti sia le attività che avvengono sulla piattaforma ma anche quelle inerenti ai lati più interni dell'azienda o ente che utillizzerà l'applicativo (per esempio la gestione del budget).

I grafici proposti a gli utenti utilizzatori sono i seguenti :

- *Tutte le segnalazioni*, mostra il numero delle segnalazioni presenti sulla piattaforma in una determinata giornata.
- *Bassa priorità*, mostra il numero delle segnalazioni con bassa priorità presenti sulla piattaforma in una determinata giornata.
- *Medie priorità*, mostra il numero delle segnalazioni con media priorità presenti sulla piattaforma in una determinata giornata.
- *Alta priorità*, mostra il numero delle segnalazioni con alta priorità presenti sulla piattaforma in una determinata giornata.

- *Budget*, mostra le variazioni che ha subito il budget aziendale.
- *Costo segnalazioni*, mostra il costo per la riparazione delle segnalazioni
- *Utenti attivi*, mostra la **tipologia** di utenza presente sulla piattaforma mediante un grafico a torta.

Figura 5.13: Menu a tendina per la selezione del grafico da visualizzare.

La scelta del grafico avviene mediante una *JComboBox* (**figura 5.13**), alla selezione i grafici vengono soltanto scambiati quindi si utilizza sempre lo stesso *JPanel*.

Le tipologie di grafici utilizzate sono due, grafici a **linee** e **torta**. Nella **figura 5.14** viene mostrato il grafico a linee rappresentante le transazioni (per esempio costi per riparare segnalazioni, costi amministrativi, *ecc...*) effettuate dall'ente che ha installato l'applicativo.

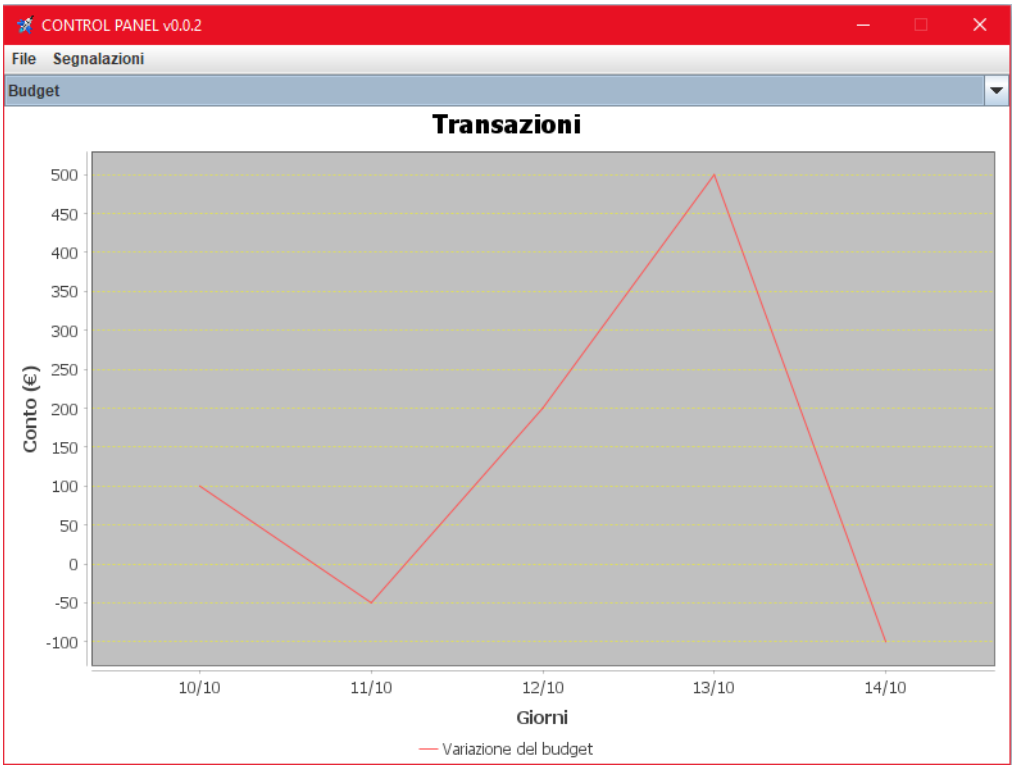

Figura 5.14: Grafico che mostra le variazioni del budget aziendale.

5.2.3 Login per gli impiegati

Per la realizzazione del **RF06** si è pensato ad un modo per aggirare il problema della restrizione di Firebase a gli applicativi mobile, la soluzione utilizzata consiste nell'impiego delle **Firebase Auth REST API**[19]. In questa maniera si è resa possibile l'autenticazione su un servizio ideato per mobile con una richiesta HTTP POST.
La schermata di Login presenta un *JTextField* e un *JPasswordField* per l'inserimento di email e password. Le informazioni presenti all'interno del campo vengono prelevate e combinate per creare una richiesta HTTP che abbia un **Body Payload** composto da tre campi :

- *email*, una stringa rappresentante l'email.
- *password*, una stringa rappresentante la password.
- *returnSecureToken*, un booleano che abilita la restituzione nel **Response Payload** del ID e refresh del Token.

Una volta effettuata la richiesta, si riceverà in risposta (all'interno di un *Json*) un **Response Payload** contenente diverse informazioni, ma quelle rilevanti sono contenute nei campi email, localId, idToken.

- *email*, indirizzo email dell'utente autenticato.
- *localId*, l'**uid** dell'utente autenticato.
- *idToken*, un Firebase Auth ID Token per l'utente autenticato.

Ma questo non basta per assicurare che sia un impiegato ad essere autenticato sul sistema, perché chiunque potrebbe essere registrato sulla piattaforma Firebase Auth (anche gli utenti normali), è necessario un ulteriore controllo.

Questo verrà effettuato attraverso il metodo :

```
1 private Employee retrieveOtherInfo(String email, String uid, String
      token);
```

Il metodo prende in ingresso i dati ricevuti dal ResponseBody, ed attraverso un *addValueListener(...)* si andrà a verificare sul real time data base sul nodo *employee* la presenza di un figlio che contenga i determinati campi.

In caso di risposta affermativa si restituirà un oggetto *Employee* contenente le informazioni presenti sul database, altrimenti si restituirà un oggetto nullo (*null object*).

Questo in maniera che il metodo chiamante possa aprire il pannello di controllo dell'applicazione nel caso di login riuscito (quindi verificando $employee \neq null$) oppure di mostrare un *JDialog* di errore durante il tentativo di Login ($employee = null$).

5.3 Interazione fra le due applicazioni

5.3.1 Gestione segnalazioni

Il **RF08** è uno dei più importanti, specifica che è compito dell'impiegato amministrare le priorità (ed eventualmente anche altre informazioni) delle segnalazioni, l'utente al momento di creazione della segnalazione genera (inconsapevolmente) una priorità *temporanea* della stessa (il ciclo di vita delle segnalazioni è trattato nella sezione 3.2).

Sarà l'impiegato, dalla parte remota del sistema a decidere se effettivamente la priorità temporanea sia effettivamente una priorità valida, altrimenti utilizzerà gli strumenti a propria disposizione per modificarla ed aggiustarla.

Esempio 5.3.1. Un utente effettua una nuova segnalazione con l'apposito strumento della dashboard, l'utente imposta come tipologia della segnalazione "Altro", il sistema implicitamente imposta un valore di priorità temporaneo pari ad 1.

L'impiegato riceve sull'applicazione remota una segnalazione in stato di *Attesa*, controllando la segnalazione si accorge che il fatto segnalato è ben più grave di ciò che è rappresentato dalla priorità temporanea, allora l'impiegato ne modifica il valore alzando la priorità a 2.

5.3.2 Riapertura segnalazione

Il requisito **RF04.1.2** è molto sintetico. Dal lato Android è presente una semplice schermata (**figura 5.15**) che presenta una una *TextArea* ed un bottone, le quali permettono all'utente (al quale la segnalazione è stata chiusa) di effettuare una richiesta di riapertura a patto di inserire una motivazione opportuna (sarà compito dell'impiegato a verificare che sia tale).

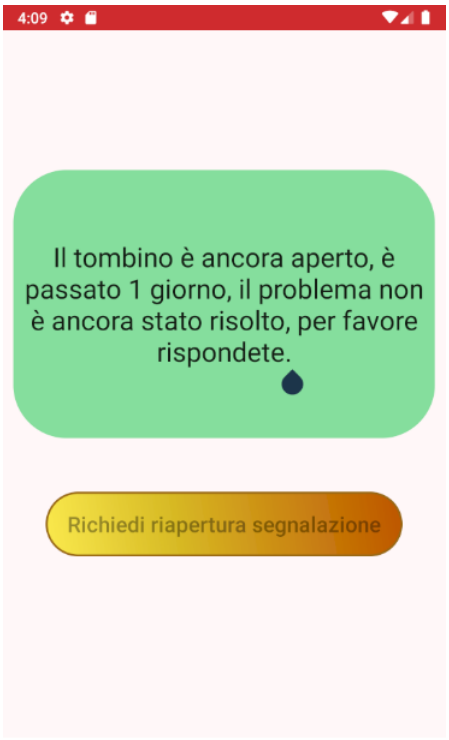

Figura 5.15: Richiesta di riapertura di una segnalazione.

Source: *Android Studio*

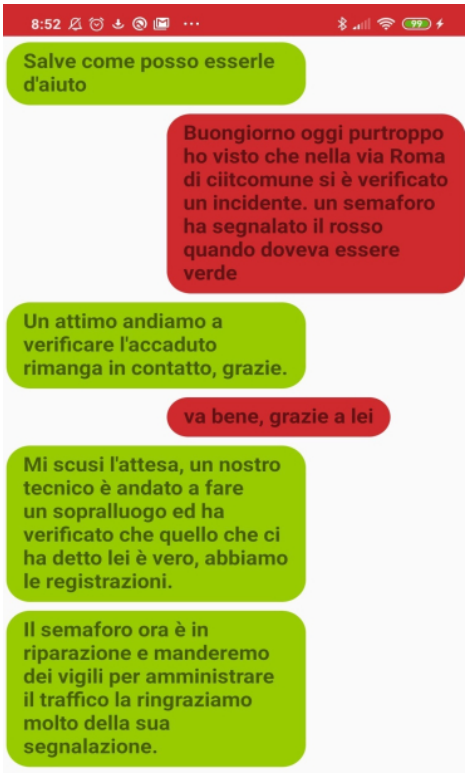

Figura 5.16: Chat Utente-Impiegato.

Source: *Xiaomi Redmi Note 7*

La richiesta effettuata dall'utente verrà mostrata sull'applicazione remota per l'impiegato (**figura 5.17**), questa richiesta sarà visualizzabile utilizzando l'apposito strumento di visualizzazione delle **segnalazioni chiuse**.

Lo strumento di visualizzazione delle segnalazioni chiuse mostrerà tra le varie informazioni della segnalazione selezionata, l'eventuale richiesta di

riapertura, sarà compito dell'impiegato decidere se riaprire o meno la segnalazione.

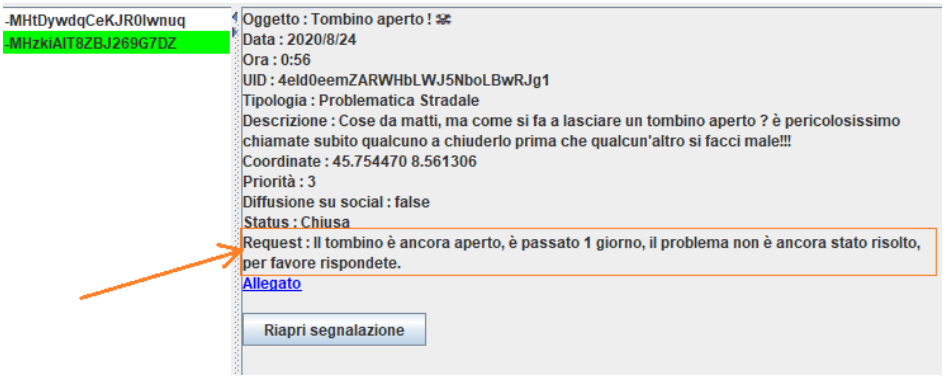

Figura 5.17: Dove viene visualizzata la richiesta di riapertura dopo il suo invio.

5.3.3 Chat bidirezionale Utente-Impiegato

Il **RF04.1.1** consiste in una chat (o *mini-forum*) bidirezionale in real time tra l'utente e l'impiegato. Per la realizzazione della chat si sono presi in considerazione due approcci : l'utilizzo dei **Kafka Connector**[5] con Firebase oppure solamente il real time database di Firebase. A seguito di uno studio di fattibilità dell due opzioni, sono emerse delle problematiche con l'integrazione dei Kafka Connectors di Firebase su Docker, a seguito di ciò si è optato per l'implementazione solo su Firebase. Si può accedere a tale strumento soltanto dopo che la propria segnalazione è stata mossa da stato di *Pending* a stato di *Attesa*.

La chat consiste in un insieme di coppie chiave-valore caricate sopra il real time database, per ogniuna delle due applicazioni esiste un interfaccia differente della stessa chat.

La **chiave** sul database rappresenta l'indice del messaggio nella discussione, e viene utilizzato per mantenere un integrità dei messaggi nella fase di caricamento sulle interfacce. Il **valore** contiene il messaggio stesso. L'applicazione Android sfrutta anche questa volta un *RecyclerView*, dove ogni oggetto *ViewHolder* è un messaggio (**figura 5.16**), i messaggi vengono scaricati dal database (vedere il **codice 5.3.3**).

```java
private void getMessagesFromFirebase() {
    Query query = databaseReference;

    query.addListenerForSingleValueEvent(new
        ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot snapshot)
            {
            ClearAll();
            for (int i = 1; i <= snapshot.getChildrenCount();
                i++) {
                Messages message = new Messages();
                message.setMsg(Objects.requireNonNull(snapshot.child("msg_"
                    + i).getValue()).toString());
                messageList.add(message);
            }
            messageAdapter = new
                MessageAdapter(getApplicationContext(),
                messageList, userFullname);
            recyclerView.setAdapter(messageAdapter);
            messageAdapter.notifyDataSetChanged();
        }

        @Override
        public void onCancelled(@NonNull DatabaseError error) {

        }
    });
}
```

Codice 5.2.3, Java, Recupero dei messaggi appartenenti ad una discussione Utente-Impiegato dal Realtime Database di Firebase.

Nel codice si aggiunge un listener per un singolo valore ad un oggetto query, il quale ha come riferimento un nodo che ha come percorso del database, *reports\reportIDentifier\discussion*. Questo percorso ha come figli tutti i messaggi della discussione ordinati per chiave.

I messaggi vengono scaricati ed il loro valore viene incapsulato in un oggetto di tipo *Message* che verrà a sua volta inserito in un *ArrayList* contenente tutti i messaggi dell'intera discussione. Soltanto una volta raggiunto il termine della parte iterativa del codice ci sarà la generazione del *MessageAdapter*, un particolare oggetto che permette di gestire il RecyclerView ed i vari items così da ottenere una chat scorrevole ed **user-friendly**.

Capitolo 6

Conclusione

In conclusione allo studio guidato, nella sezione 6.1 verranno illustrate le considerazioni su Apache Kafka maturate durante l'esperienza con la piattaforma.

6.1 Considerazioni su Apache Kafka

Apache Kafka è una piattaforma con stile architetturale *publisher-subscribe*, essa offre un ottima base per la costruzione di applicazioni distribuite e propone molte caratteristiche interessanti per una web-company. Risulta quindi necessario effettuare una comparazione di esse, e mettere in luce sia i lati positivi che negativi, per permette di avere un quadro completo di quello che è Apache Kafka.

Vantaggi

- **High-throughput**
  Kafka è capace di gestire alte velocità e alti volumi di dati utilizzando un hardware che **non** è in egual misura costoso.
- **Low latency**
  Kafka è in grado di gestire i *messaggi* (o *eventi* o *record*) con una bassa latenza, nell'ordine dei ms.
- **Fault tolerant**
  Kafka è in grado di resistere ai fallimenti dei macchinari interni ad un cluster (grazie alle **ISR**, *In-Sync Replica*).
- **Durability**
  I record di Kafka sono persistenti sul disco e replicati, rendendoli durabili

- **Struttura a coda**
  La maggior parte dei databases odierni utilizza una struttura ad albero, questo conferisce una performance in ricerca di $O(\log n)$. Ma nel caso di Kafka la struttura utilizzata è una coda e nella maggior parte dei casi i dati vengono accodati al sistema, questo rende le letture semplici ottenendo una performance O(1).

- **Real-Time handling**
  Kafka è in grado di gestire pipeline di dati in tempo reale.

- **Scalability**
  Kafka è in grado di gestire grandi quantità di messaggi contemporaneamente.

Svantaggi

- **Assenza di un set completo di strumenti per il monitoraggio**
  Questo vale anche per strumenti di tipo gestionale, può rendere l'approccio da parte degli sviluppatori arduo e tedioso.

- **Non supporta le wildcards built-in**
  Kafka non supporta le wildcards **built-in** per la selezione dei topic, può soltanto effettuare il **matching** esatto con il nome del topic. Tuttavia, è possibile ovviare a questa piccola problematica creando delle funzioni ad hoc per la gestione delle wildcards.

- **Mancanza di alcuni paradigmi per i messaggi**
  Alcuni paradigmi come point-to-point queues, request/reply, ... non sono presenti in Kafka (per il funzionamento di certi casi d'uso).

Per via di queste performance e caratteristiche, in particolare la scalabilità, Apache Kafka viene utilizzato ampiamente nei grandi spazi di dati per via della sua affidabilità nel gestire grandi quantità di dati (viene adottato da compagnie come Netflix, Paypal, Twitter, Spotify, Linkedin, Coursera, ...).

Apache Kafka si dimostra una efficace soluzione per le grandi web-company, ma è altrettanto utilizzabile all'interno di sistemi di scala più piccola, questo perché Kafka rimane lo stesso un centro affidabile dove tutti i messaggi del sistema vengono collezionati e tenuti.

Apache Kafka è attualmente una piattaforma in costante crescita di adozione, per questo motivo sempre più compagnie sono alla ricerca di *computer scientists* e *software engineers* che siano competenti nell'uso di tale strumento, ma questo difficilmente questo accade[7].

6.2 Note finali

Giungendo alla conclusione di questo percorso si può asserire che il progetto sviluppato è stato all'altezza delle sue aspettative e che l'esperienza formativa ricavata ne è di immenso valore, poiché una grande fetta di questo lavoro è stata spesa sulla lettura delle documentazioni e sullo studio dei meccanismi fondamentali.
Lo sviluppo del sistema event-driven iniziato verso il termine del 2019, ha sollevato parecchie sfide durante la fase di progettazione e implementazione, per merito di ciò ritengo opportuno sottolineare la necessità di effettuare un attenta analisi prima di lanciarsi nello sviluppo di un intero sistema.

Strumenti potenti come il real time database di Firebase e lo stream processing di Apache Kafka hanno reso possibile lo sviluppo di un tale sistema EDA con relativa semplicità.

Poiché gli strumenti utilizzati sono tutt'altro che banali, richiedono una sufficiente comprensione del loro funzionamento prima di cimentarsi nell'utilizzo.

Il sistema implementato risulta notevolmente ampliabile e mutabile, questo sta ad evidenziare che la conclusione di questa tesi non pone termine al mondo di possibilità implementabili all'interno del progetto.

Alcune delle idee su possibili estensioni dell'implementazione :

- Spostare l'intero requisito di messaggistica su Apache Kafka, in modo da usufruire dello stream processing per elaborare i singoli messaggi in tempo reale (come censurare delle parole inappropriate).
- Effettuare una *join* fra tutte le segnalazioni (smistate nei topic) e prelevare quelle con le coordinate simili o uguali.
- Introdurre un sistema di punteggi e ricompense per gli utenti segnalatori (così da incentivarne l'utilizzo).
- Creare una community all'interno dell'applicativo mobile del sistema, con successivo ampliamento di chat bidirezionali Utente-Utente ma anche di gruppo.

Le idee per il miglioramento del sistema sono innumerevoli, ed è questo il bello di progettare un proprio sistema, poter dare spazio alle nostre idee e nel tempo vederle materializzarsi.

Appendice A

Documento S.R.S.

Note per il lettore
In questa appendice si vuole mostrare il livello di dettaglio con cui sono stati definiti i requisiti funzionali durante la fase di analisi. Al di sotto di questo paragrafo è riportata una porzione del documento S.R.S., contenente alcuni esempi di requisiti (poiché l'inserimento dell'intero documento di specifica allungherebbe esageratamente la tesi).

3.Requisiti Specifici

RF02: Visualizzazione satellitare delle segnalazioni
Questo servizio può essere utilizzato da gli utenti registrati, permette di visualizzare una mappa satellitare di una località di interesse con le relative informazioni su malfunzionamenti e/o traffico. L'utente può impostare la località manualmente inserendola da tastiera, oppure potrà essere rilevata dal ricevitore GPS del dispositivo. Nel caso, l'utente inserisca una località non esistente verrà mostrato un opportuno messaggio di errore. Una volta inserita la località preferita, questa sarà memorizzata per futuri utilizzi. La mappa satellitare mostrerà la zona interessata dall'utente e gli eventuali malfunzionamenti attraverso dei pin colorati, il colore dei pin ne indica la priorità, cliccando su di esso si verrà riportati al dato testuale relativo.

Colori pin:

- Rosso, priorità alta
- Giallo, priorità media
- Verde, priorità bassa

Visualizzazione mappa

Goal Level: Sea Level

Main Success Scenario:

1. L'utente risulta loggato all'interno dell'applicazione
2. L'utente clicca sulla sezione "Territorio" dalla side-bar a sinistra
3. L'applicazione carica una nuova pagina
4. Il sistema preleva i dati riguardanti le recenti segnalazioni e li carica sulla mappa
5. L'utente può ora spostarsi sulla mappa e cliccare su i pin colorati per leggere i dettagli delle segnalazioni.

Extensions:
3a: Il sistema fallisce nel caricare la pagina "Territorio"
.1: Il sistema mostra l'errore sulla pagina stessa
4a: Il sistema non riesce a prelevare i dati delle segnalazioni dal database
.1: Il sistema mostra all'utente un errore SQL sulla pagina stessa
4b: Il sistema non riesce a caricare i dati delle segnalazioni sulla pagina
.1: Il sistema mostra l'errore sulla pagina stessa

RF04.1.3: Recensione segnalazione
Gli utenti la cui segnalazione è stata conclusa, potranno esprimere il loro gradimento (o non) tramite una valutazione decimale da 0.0 a 5.0, dove il massimo sarà rappresentato graficamente da 5 stelle e il minimo da nessuna, la mezza stella verrà rappresentata quando la mantissa della valutazione prende valore 5.

Recensione segnalazione

Goal Level: Sea Level

Main Success Scenario:

1. L'utente risulta loggato all'interno dell'applicazione
2. L'impiegato ha chiuso lo stato della segnalazione
3. L'utente accede alla pagina di recensione della segnalazione dal bottone "Vota" sulla segnalazione chiuse, all'interno della sezione "Segnalazioni chiuse" accessibile dalla Dashboard.
4. Viene visualizzata la pagina di valutazione della segnalazione, vengono mostrate delle grafiche selezionabili per esprimere la valutazione ed una Textbox **non obbligatoria** per fornire dettagli aggiuntivi ed un bottone "Invia Voto"
5. L'utente preme il bottone "Invia Voto"
6. Il sistema mette invia la valutazione sul database
7. Viene mostrato un messaggio di completamento dell'operazione, la segnalazione chiusa non risulterà più valutabile

Extensions:
3a: Il sistema fallisce nel caricare la pagina "**Recensione segnalazione**"
.1: Il sistema mostra l'errore sulla pagina stessa
4a: Il sistema fallisce nel caricare la pagina di valutazione
.1: Il sistema non mostrerà alcun dettaglio, stato, messaggi del mini-forum ma un messaggio di errore.
4b: Il sistema rileva che non è stato espressa alcuna valutazione
.1: Il sistema non prosegue con l'invio della segnalazione, e manda una notifica all'utente di cliccare su una valutazione per proseguire.

RF05: Statistiche delle segnalazioni nella zona (utente)
L'utente ha la possibilità di visualizzare, mediante un grafico, l'andamento statistico inerente alle segnalazioni globali della località. L'utente può scegliere i dati interessati da visualizzare sul grafico, il grafico sarà organizzato in modo d'avere le date e orari sull'asse delle ascisse e il numero di segnalazioni sull'asse delle ordinate. I tipi di dati interessati sono :

- Tutte le segnalazioni
- Segnalazioni bassa priorità
- Segnalazioni media priorità
- Segnalazioni alta priorità

Statistiche delle segnalazioni nella zona (utente)

Goal Level: Sea Level

Main Success Scenario:

1. L'utente risulta loggato all'interno dell'applicazione
2. L'utente clicca sulla sezione "**Statistiche zona**" dalla Dashboard
3. L'applicazione carica la pagina inerente al servizio
4. Il sistema preleva i dati riguardanti le segnalazioni presenti nella località dal database, viene effettuata una piccola elaborazione delle segnalazioni e successivamente verranno disposte sul grafico secondo la tipologia di default, ovvero visualizzando tutte le segnalazioni
5. L'utente può cliccare i punti sulla funzione del grafico (le intersezioni rappresentano la singola segnalazione) verranno mostrati i dettagli della segnalazione
6. L'utente può cambiare l'ordinamento del grafico selezionando la tipologia di segnalazione che vuole vedere tramite menu a tendina sopra il grafico

Extensions:
3a: Il sistema fallisce nel caricare la pagina "**Statistiche zona**"
.1: Il sistema mostra l'errore sulla pagina stessa
4a: Il sistema non riesce non riesce a prelevare i dati delle segnalazioni dal database
.1: Il sistema mostra all'utente un errore SQL sulla pagina stessa
4b: Il sistema non riesce a caricare i dati delle segnalazioni sul grafico
.1: Il sistema notifica il problema con un messaggio sulla pagina stessa

Bibliografia

[1] ... ..., Kent Beck. **Manifesto per lo Sviluppo Agile di Software**, https://agilemanifesto.org/iso/it/manifesto.html.

[2] Inizialmente Linus Torvalds attualmente Juno Hamano. **Git**, https://git-scm.com/.

[3] Lenses Box. **fast-data-dev / kafka-lenses-dev**, https://github.com/lensesio/fast-data-dev.

[4] Confluent. **Duality of Streams and Tables**, https://docs.confluent.io/current/streams/concepts.html#duality-of-streams-and-tables.

[5] Confluent. **Kafka Connect**, https://docs.confluent.io/platform/current/connect/index.html.

[6] Confluent. **REST Proxy API v2**, https://docs.confluent.io/platform/current/kafka-rest/api.html#crest-api-v2.

[7] Luanne Dauber. **The 2017 Apache Kafka Survey: Streaming Data on the Rise**, https://www.confluent.io/blog/2017-apache-kafka-survey-streaming-data-on-the-rise.

[8] Inc. Docker. **Docker**, https://www.docker.com/.

[9] ECMA-262. **JSON**, https://www.json.org/json-en.html.

[10] Roy Fielding. **Architectural Styles and the Design of Network-based Software Architectures**, https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[11] Apache Software Foundation. **Kafka Consumer API**, https://kafka.apache.org/documentation/#consumerapi.

[12] Apache Software Foundation. **Kafka Producer API**, https://kafka.apache.org/documentation/#producerapi.

[13] Apache Software Foundation. **The Kafka Streams DSL (Domain Specific Language)**, https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. ***Ingegneria del software: fondamenti e principi***. Pearson Education Italia, 2004.

[15] Inc. GitHub. **GitHub**, https://github.com/.

[16] Google. **Firebase Admin SDK**, https://firebase.google.com/docs/admin/setup#java.

[17] Google. **Firebase Authentication API**, https://firebase.google.com/docs/auth/android/start#java.

[18] Google. **Firebase Real Time Database API**, https://firebase.google.com/docs/database/android/start#java.

[19] Google. **Firebase REST API,** https://firebase.google.com/docs/reference/rest/auth.

[20] Google. **Firebase Storage API**, https://firebase.google.com/docs/storage/android/start#java.

[21] Google. **Google Firebase**, https://firebase.google.com/.

[22] JetBrains. **IntelliJ IDEA**, https://www.jetbrains.com/idea/.

[23] Google & JetBrains. **Android Studio**, https://developer.android.com/.

[24] Apache Kafka. **Apache Kafka**, https://kafka.apache.org/.

[25] AT&T Labs and contributors. **Graphviz**, https://graphviz.org/.

[26] Andrew Littlefield. **The Beginner's Guide To Scrum And Agile Project Management**, https://blog.trello.com/beginners-guide-scrum-and-agile-project-management.

[27] Visual Paradigm International Ltd. **Visual Paradigm**, https://www.visual-paradigm.com/.

[28] Atlassian Corporation Plc. **Trello**, https://trello.com/.

[29] Elisabetta Raguseo. **Big data technologies: An empirical investigation on their adoption, benefits and risks for companies**, https://www.sciencedirect.com/science/article/abs/pii/S0268401217300063?via%3Dihub.

[30] Jun Rao. **The Value of Apache Kafka in Big Data Ecosystem**, https://www.confluent.io/blog/the-value-of-apache-kafka-in-big-data-ecosystem/.

[31] Dimitri van Heesch. **Doxygen**, https://www.doxygen.nl/index.html.

[32] Dimitri van Heesch. **Doxywizard**, https://www.doxygen.nl/manual/doxywizard_usage.html.