UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in ingegneria informatica e dell'automazione

Sviluppo e progettazione di una Web App per la gestione di campagne partecipative di monitoraggio ambientale

Design and development of a web app for the management of participatory environmental monitoring campaigns

Relatore:

Prof. Emanuele Storti

Tesi di Laurea di:

Ludovico Nollino 1093732

A.A. 2022 /2023

INDICE

INDICE DELLE FIGURE E DELLE TABELLE

INTRODUZIONE

Il fine principale di questo elaborato è quello di presentare la progettazione e lo sviluppo di un'applicazione web, che rappresenta uno dei risultati del progetto "Y4DCS - Youth 4 Digital Citizen Science", eseguito ai fini del tirocinio curricolare.

Lo sviluppo di tale web app, prodotta per l'ente ecologista Legambiente Marche, rappresenta, però, solo una parte del lavoro che si è svolto per questa iniziativa: il prodotto principale è costituito, in realtà, da un' applicazione destinata ai normali utenti mobile che consente, tramite lo smartphone e gli strumenti software messi a disposizione, la segnalazione di rifiuti in determinate zone del territorio e la possibilità di aderire alle campagne di pulizia di specifiche aree indette da Legambiente. La web app discussa in questo elaborato, invece, ha come fini lo svolgimento delle operazioni in maniera contemporanea all'applicazione mobile e l'ente ecologista stesso nei seguenti ambiti:

- Gestione delle varie segnalazioni e dei vari utenti mobile;
- Creazione delle varie iniziative di smaltimento rifiuti;
- Controllo dell'andamento delle campagne e delle segnalazioni tramite strumenti per effettuare statistiche ben precise.

La finalità di questo lavoro è, dunque, non solo sensibilizzare i normali utenti al tema dell'inquinamento facendoli partecipare in maniera attiva alle varie iniziative, ma anche aiutare l'associazione Legambiente stessa a svolgere in maniera ancora più ottimale le proprie attività, fornendo degli strumenti software accessibili, intuitivi ed in costante aggiornamento. Ed è per questo motivo e per la grande quantità di dati da gestire prevista che si è optato direttamente per lo sviluppo web.

Nel primo capitolo di questo elaborato, quindi, sarà presentata brevemente l'applicazione mobile destinata ai normali utenti e successivamente verrà fornita un'analisi dei requisiti che la web app deve avere.

Il secondo verterà, invece, interamente sulla fase di progettazione, cioè quella in cui si valuta come procedere all'implementazione. Verranno presentati gli strumenti utilizzati, saranno discussi i casi

d’uso generati dall’analisi dei requisiti e verranno presentate la struttura dati presente nel database e l’architettura scelta per il progetto.

Infine, nel terzo capitolo si analizzerà la vera e propria implementazione, entrando nel dettaglio di come sono state sviluppate alcune delle varie funzionalità più importanti. Verranno evidenziate, inoltre, alcune problematiche incontrate durante lo sviluppo e la loro soluzione. Saranno, poi, presenti degli esempi di utilizzo della web app con dati di prova utilizzati durante lo sviluppo per effettuare dei test.

Nelle conclusioni dell’elaborato verranno, poi, presentati anche alcuni possibili miglioramenti, riguardanti versioni future della web app.

Capitolo 1

Analisi dei requisiti

In questo capitolo sarà mostrata un'analisi preliminare dello sviluppo vero e proprio dell'applicazione web. In particolare, verrà prima fornita una breve panoramica dell'app mobile utilizzata dagli utenti per le segnalazioni dei rifiuti e per aderire alle campagne, in modo tale da capire che tipi di informazioni confluiranno nel database. Successivamente si passerà ad analizzare gli obiettivi funzionali e i requisiti tecnici che la web app dovrà avere.

1.1 Panoramica app mobile

Il primo passo di questa analisi è quella di fornire una breve presentazione dell'applicazione mobile, con la quale la web app dovrà lavorare simultaneamente. Come già accennato sopra, lo scopo di questo strumento è quello di permettere agli utenti di effettuare segnalazioni di rifiuti presenti non solo nelle aree verdi e nelle spiagge del territorio marchigiano, ma anche lungo le strade, fotografandoli con il proprio smartphone. Infatti, l'idea è quella di permettere all'utente base di fare il monitoraggio dei rifiuti anche per poco tempo, durante lo svolgimento di normali attività quotidiane. Per incentivare questo, inoltre, l'app è stata dotata anche di una funzione di gamification: ogni segnalazione assegna un punteggio all'utente in base al rifiuto che individua e smaltisce negli appositi bidoni per la raccolta differenziata. Oltre a questa importante funzionalità, l'app consente agli utilizzatori di aderire a campagne di pulizia indette da Legambiente e di tenere sotto controllo l'andamento delle segnalazioni degli altri utenti, visualizzandole sulla mappa.

Dato l'alto numero di dati prodotti dall'applicazione e per poter effettuare dei controlli più approfonditi sulle informazioni prodotte dagli utenti mobile per rispettare le regole di privacy, si è optato per lo sviluppo della web app amministrativa discussa in questo elaborato. Come mostrato nel diagramma dei sistemi, il punto di raccordo tra i due strumenti software è il database "Cloud Firestore" della piattaforma "Firebase", offerta da Google, che sarà presentata in seguito.

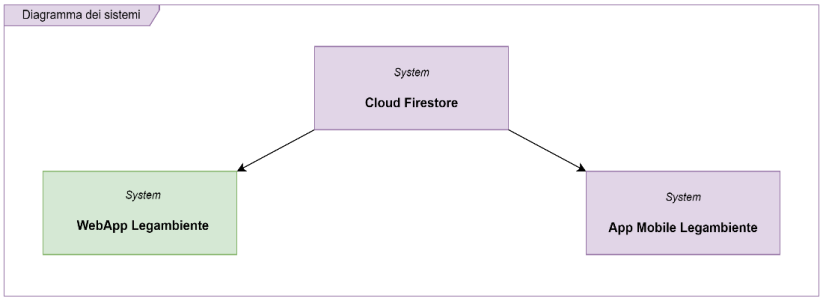

Figura 1.1 Diagramma dei sistemi

1.2 Requisiti

Obiettivo di questo secondo step è valutare e identificare i requisiti fondamentali che la web app dovrà soddisfare. Questo passo è essenziale per ottenere una chiara visione di come procedere nella fase di progettazione e sviluppo, nonché per stabilire i criteri di completamento del lavoro. L'analisi dei requisiti comporta la creazione di una sintesi delle funzionalità che saranno implementate e delle specifiche tecniche che il software dovrà rispettare, riassumendo in modo conciso il prodotto finale concettualizzato.

La suddivisione dello studio dei requisiti avverrà in due categorie principali: requisiti funzionali e requisiti non funzionali. I requisiti funzionali rappresentano gli obiettivi veri e propri relativi al funzionamento della web app, ossia le funzioni disponibili per l'utente finale. In altre parole, sono i requisiti "visibili" ed essenziali per il corretto funzionamento dell'applicazione.

D'altra parte, i requisiti non funzionali riguardano le caratteristiche tecniche dello sviluppo della web app, cioè come le funzionalità devono essere implementate. Questi requisiti includono aspetti come le prestazioni, la sicurezza, l'usabilità e altre caratteristiche che definiscono la qualità complessiva del prodotto. La comprensione chiara e accurata di entrambe queste categorie di requisiti è cruciale per garantire il successo del progetto di sviluppo della web app.

Di seguito è fornita una suddivisione delle macro-aree individuate per avere un'accurata visione dei requisiti:

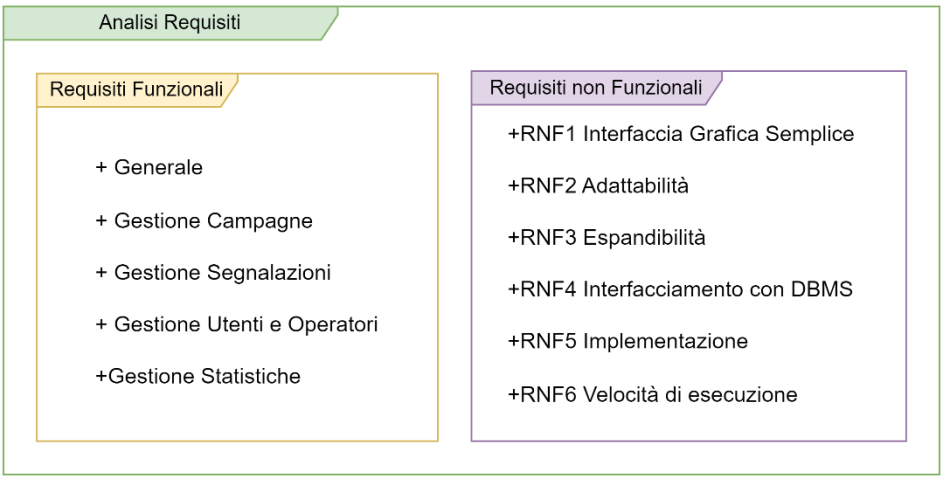

Figura 1.2 Analisi dei requisiti

Inoltre, è opportuno fornire anche un breve glossario dei termini utilizzati frequentemente nell'analisi e nella progettazione:

Termine	Descrizione	Tipo
Utente mobile	Soggetto iscritto all'app mobile in quanto segnalatore di rifiuti	Tecnico
Rifiuto	Oggetto da smaltire che si trova in ambiente	Business
Campagna	Evento di monitoraggio e/o raccolta rifiuti in una determinata area, indetta da soggetti autorizzati quali operatori	Business
Area	Zona fisica delimitata nel quale è possibile segnalare rifiuti in accordo alla campagna durante la sua durata	Business
Database	Struttura atta a contenere dati	Tecnico
Segnalazione	Processo mediante il quale un utente utilizza l'applicazione mobile per informare il sistema della presenza di un rifiuto	Business
Operatore	Soggetto che presidia e controlla le campagne durante il loro svolgimento	Business
Amministratore	Unico soggetto che è autorizzato all'utilizzo della web app	Tecnico

Tabella 1.1 Glossario termini

1.2.1 Requisiti funzionali

I requisiti fondamentali che la web app dovrà avere sono sintetizzati nella figura 1.3 sottostante:

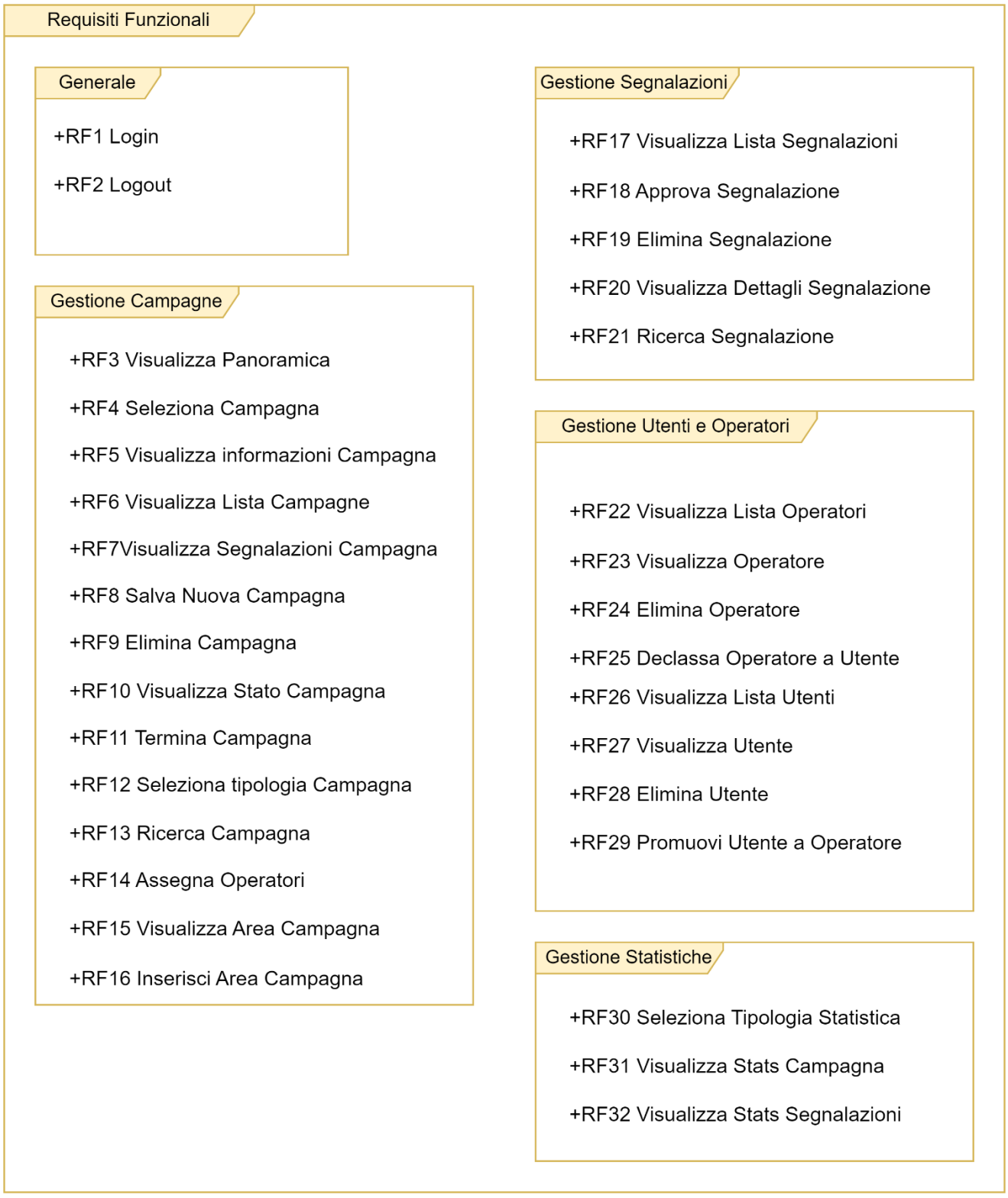

Figura 1.3 Requisiti funzionali

Di seguito è fornita una tabella contenente la descrizione di ogni requisito:

Requisito	***Descrizione***
RF1 Login	Il sistema dovrà permettere l'accesso gestendo la procedura di autenticazione
RF2 Logout	Il sistema dovrà gestire l'uscita
RF3 Visualizza Panoramica	Il sistema dovrà permettere una visualizzazione della panoramica delle attività in svolgimento sul territorio
RF4 Seleziona Campagna	Il sistema dovrà permettere la selezione di una campagna
RF5 Visualizza Informazioni Campagna	Il sistema dovrà mostrare le informazioni di una campagna selezionata
RF6 Visualizza Lista Campagne	Il sistema dovrà visualizzare la lista delle campagne sia in corso che terminate
RF7 Visualizza Segnalazioni Campagna	Il sistema dovrà consentire la visualizzazione delle segnalazioni effettuate in una specifica campagna
RF8 Salva Nuova Campagna	Il sistema dovrà permettere l'inserimento e il salvataggio di una nuova campagna di monitoraggio rifiuti
RF9 Elimina Campagna	Il sistema dovrà permettere l'eliminazione di una campagna salvata e terminata
RF10 Visualizza Stato Campagna	Il sistema dovrà mostrare lo stato di una campagna (in corso o terminata)
RF11 Termina Campagna	Il sistema dovrà permettere di far terminare una campagna in corso
RF12 Seleziona tipologia Campagna	Il sistema dovrà consentire all'amministratore di far scegliere la tipologia della campagna durante l'inserimento
RF13 Ricerca Campagna	Il sistema dovrà permettere la ricerca di una determinata campagna all'interno della lista
RF14 Assegna Operatori	Il sistema dovrà consentire all'amministratore di assegnare operatori durante il processo di inserimento di una nuova campagna
RF15 Visualizza Area Campagna	Il sistema dovrà mostrare l'area di svolgimento di una campagna sulla mappa del territorio
RF16 Inserisci Area Campagna	Il sistema dovrà permettere l'inserimento dell'area di svolgimento sulla mappa

RF17 Visualizza Lista Segnalazioni	Il sistema dovrà mostrare la lista delle segnalazioni effettuate dagli utenti mobile
RF18 Approva Segnalazione	Il sistema dovrà consentire all'amministratore di approvare una segnalazione
RF19 Elimina Segnalazione	Il sistema dovrà consentire all'amministratore di eliminare una segnalazione
RF20 Visualizza Dettagli Segnalazione	Il sistema dovrà mostrare i dettagli di una specifica segnalazione scelta dall'amministratore
RF21 Ricerca Segnalazione	Il sistema dovrà consentire la ricerca di una segnalazione all'interno di una lista
RF22 Visualizza Lista Operatori	Il sistema dovrà mostrare la lista degli operatori che presidieranno alle campagne
RF23 Visualizza Operatore	Il sistema dovrà mostrare le informazioni di un operatore
RF24 Elimina Operatore	Il sistema dovrà consentire l'eliminazione di un operatore
RF25 Declassa Operatore a Utente	Il sistema dovrà permettere il declassamento di un operatore ad utente mobile
RF26 Visualizza Lista Utenti	Il sistema dovrà mostrare la lista degli utenti che utilizzano l'app mobile
RF27 Visualizza Utente	Il sistema dovrà mostrare le informazioni di un utente mobile
RF28 Elimina Utente	Il sistema dovrà consentire l'eliminazione di un utente mobile
RF29 Promuovi Utente a Operatore	Il sistema dovrà permettere la promozione di un utente mobile ad operatore
RF30 Seleziona Tipologia Statistica	Il sistema dovrà permettere all'amministratore di scegliere il tipo di statistica da visualizzare
RF31 Visualizza Stats Campagna	Il sistema dovrà mostrare le statistiche di una determinata campagna scelta dall'amministratore
RF32 Visualizza Stats Segnalazioni	Il sistema dovrà mostrare le statistiche delle segnalazioni effettuate dagli utenti mobile

Tabella 1.2 Requisiti funzionali

1.2.2 Requisiti non funzionali

In questa sezione, saranno esaminati i requisiti non funzionali, ossia i fattori da considerare durante la fase di sviluppo che influenzeranno l'implementazione del software e i criteri che il software dovrà rispettare. Di seguito è fornita una tabella contenente la descrizione di ogni requisito non funzionale:

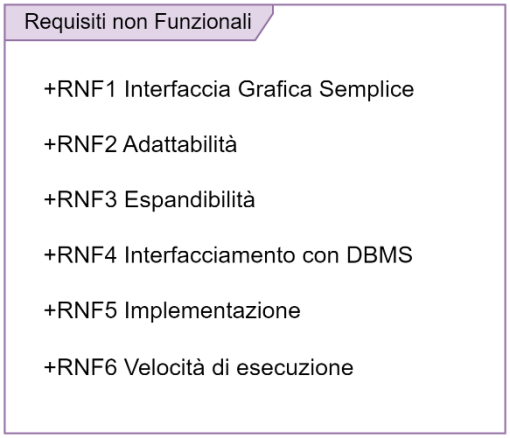

Figura 1.4 Requisiti non funzionali

Requisito	**Descrizione**
RNF1 Interfaccia Grafica Semplice	Il sistema dovrà essere dotato di un'interfaccia grafica semplice e intuitiva per garantire un utilizzo comodo ed efficiente
RNF2 Adattabilità	Nel processo di sviluppo, verranno considerate possibili evoluzioni future, sia per quanto riguarda la struttura dei dati e il suo contenuto, sia per l'applicazione stessa
RNF3 Espandibilità	Nel contesto dello sviluppo, verrà considerata anche la possibilità di futuri sviluppi della web app in termini di funzionalità, e il software sarà progettato con l'obiettivo di rendere agevole la sua espansione in un secondo momento
RNF4 Interfacciamento con DBMS	La web app sarà configurata per interagire con un sistema di gestione di database al fine di archiviare tutte le informazioni necessarie

RNF5 Implementazione	La web app sarà basata sui linguaggi di programmazione web HTML, CSS e Javascript. Inoltre, sarà presente la sintassi JSX, specifica di ReactJS illustrata più avanti
RNF6 Velocità di esecuzione	La web app dovrà risultare veloce nel suo funzionamento, in modo da garantire un buon utilizzo e un rapido aggiornamento delle informazioni provenienti dal database

Tabella 1.3 Requisiti non funzionali

Capitolo 2

Progettazione

Terminata la fase di analisi dei requisiti, si procede alla progettazione, durante la quale viene studiato il piano di sviluppo della web app. In questa sezione, verranno esaminate le prime fasi dell'implementazione. In particolare, saranno prima analizzati i casi d'uso con i relativi diagrammi e poi presentati gli strumenti selezionati per il processo di sviluppo, inclusi editor, linguaggi di programmazione, librerie e browser. Successivamente, si approfondirà la struttura del database, con particolare attenzione alle informazioni più frequentemente utilizzate dall'applicazione web e alla loro organizzazione. Infine, verrà illustrata l'architettura del progetto, presentando i file utilizzati e le cartelle di supporto create.

2.1 Casi d'uso

In questa sezione, verranno esaminati i casi d'uso della web app, i quali derivano direttamente dall'analisi dei requisiti funzionali precedentemente condotta. Questi casi d'uso delineano scenari specifici e interazioni cruciali tra utenti e applicazione. Ciascuno di essi fornisce un quadro dettagliato delle modalità con cui gli utenti interagiranno con la web app per soddisfare le funzionalità identificate nei requisiti. In questo contesto, verranno esplorati i casi d'uso al fine di acquisire una comprensione approfondita di come la web app sarà utilizzata nella pratica, grazie anche ai loro rispettivi diagrammi.

L'analisi sarà suddivisa secondo le macro-aree individuate nella sezione dei requisiti funzionali: per ogni ambito verrà presentato prima il diagramma e poi descritti i relativi casi d'uso. In ogni scenario l'attore è denominato come amministratore ed è l'unico utente previsto per l'utilizzo di questa web app.

2.1.1 Generale

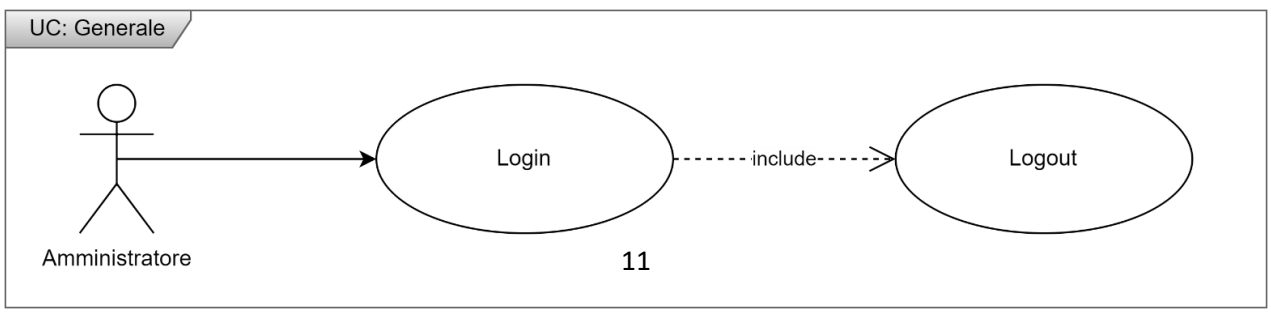

Figura 2.1 Diagramma casi d'uso per l'accesso e l'uscita dal sistema

Di seguito è fornita la descrizione dei casi d'uso di questo scenario:

Login
Questo caso d'uso si verifica qualora l'attore voglia effettuare il login per utilizzare la web app.
Attori: Amministratore
Pre-condizioni: l'attore non ha effettuato login
Post-Condizioni: l'attore ha effettuato login e può utilizzare la web app
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole iniziare ad usare la web app; 2. Il sistema visualizza la schermata di login; 3. L'attore fornisce le proprie credenziali; 4. L'attore avvia la procedura di login; 5. Il sistema approva il login.
Sequenza alternativa: Comincia al punto 4 5. Le credenziali dell'amministratore sono errate o inesistenti; 6. Il sistema non approva il login

Tabella 2.1 Descrizione caso d'uso: Login

Logout
Questo caso d'uso si verifica qualora l'attore voglia uscire dalla web app e terminare il suo utilizzo.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e può usare la web app
Post-Condizioni: l'attore ha effettuato logout e non può utilizzare la web app
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole smettere di usare la web app; 2. L'attore avvia la procedura di logout; 3. Il sistema approva il logout.
Sequenza alternativa: Nessuna

Tabella 2.2 Descrizione caso d'uso: Logout

2.1.2 Gestione campagne

Figura 2.2 Diagramma casi d'uso gestione campagne

Di seguito sono fornite delle tabelle contenenti le descrizioni dei casi d'uso individuati in questo scenario:

Visualizza Panoramica
Questo caso d'uso si verifica qualora l'attore voglia visualizzare la panoramica delle attività in svolgimento sul territorio.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login
Post-Condizioni: l'attore visualizza la mappa contenente una panoramica delle attività
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare la panoramica delle attività; 2. L'attore seleziona la sezione di visualizzazione della panoramica; 3. La web app carica la mappa contenente la panoramica delle attività; 4. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.3 Descrizione caso d'uso: Visualizza Panoramica

Seleziona Campagna
Questo caso d'uso si verifica qualora l'attore voglia selezionare una campagna sia dalla lista delle campagne sia dalla mappa contenente la panoramica delle attività.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login
Post-Condizioni: l'attore può selezionare una campagna
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole selezionare una campagna; 2. L'attore seleziona la sezione di visualizzazione della panoramica o la lista delle campagne; 3. La web app carica il contenuto della sezione scelta; 4. La web app visualizza a schermo il contenuto; 5. L'attore sceglie la campagna da selezionare.
Sequenza alternativa: Nessuna

Tabella 2.4 Descrizione caso d'uso: Seleziona Campagna

Visualizza Segnalazioni Campagna
Questo caso d'uso si verifica qualora l'attore voglia visualizzare le segnalazioni effettuate in una determinata campagna selezionata.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login ed ha selezionato una campagna
Post-Condizioni: l'attore visualizza le segnalazioni effettuate nella campagna
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare le segnalazioni effettuate in una campagna; 2. Il sistema visualizza a schermo le campagne; 3. L'attore seleziona la campagna di cui vuole visualizzare le segnalazioni; 4. La web app carica il contenuto della campagna e le segnalazioni; 5. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.5 Descrizione caso d'uso: Visualizza Segnalazioni Campagna

Visualizza Campagna
Questo caso d'uso si verifica qualora l'attore voglia visualizzare una determinata campagna.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login ed ha selezionato una campagna
Post-Condizioni: l'attore visualizza la campagna
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare una campagna; 2. Il sistema visualizza a schermo le campagne; 3. L'attore seleziona la campagna che vuole visualizzare; 4. La web app carica il contenuto della campagna; 5. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.6 Descrizione caso d'uso: Visualizza Campagna

Termina Campagna
Questo caso d'uso si verifica qualora l'attore voglia terminare una determinata campagna.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, la campagna esiste a sistema, è in corso ed è visualizzata
Post-Condizioni: lo stato della campagna è passato da "in corso" a "terminata"
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole terminare una campagna; 2. Il sistema visualizza a schermo la campagna da terminare; 3. L'attore avvia la procedura per terminare la campagna; 3.1 **If** l'attore conferma di voler terminare la campagna: il sistema aggiorna lo stato della campagna; 4. La web app visualizza a schermo la campagna con lo stato aggiornato.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla il processo per terminare la campagna 4. Si torna al punto 2.

Tabella 2.7 Descrizione caso d'uso: Termina Campagna

Elimina Campagna
Questo caso d'uso si verifica qualora l'attore voglia eliminare una determinata campagna.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, la campagna esiste a sistema ed è visualizzata con stato "terminata"
Post-Condizioni: la campagna è stata eliminata e non esiste più a sistema
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole eliminare una campagna; 2. Il sistema visualizza a schermo la campagna da eliminare; 3. L'attore avvia la procedura di eliminazione; 3.1 **If** l'attore conferma l'eliminazione: il sistema elimina la campagna.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla l'eliminazione della campagna 4. Si torna al punto 2.

Tabella 2.8 Descrizione caso d'uso: Elimina Campagna

Visualizza Lista Campagne
Questo caso d'uso si verifica qualora l'attore voglia visualizzare la lista delle campagne.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e le campagne esistono a sistema
Post-Condizioni: l'attore visualizza la lista delle campagne
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare la lista delle campagne; 2. L'attore seleziona la sezione che contiene la lista delle campagne; 3. La web app carica il contenuto della lista; 4. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.9 Descrizione caso d'uso: Visualizza Lista Campagne

Ricerca Campagna
Questo caso d'uso si verifica qualora l'attore voglia ricercare una campagna all'interno della lista.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, le campagne esistono a sistema e viene visualizzata la lista delle campagne
Post-Condizioni: l'attore può ricercare la campagna e visualizzarla
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole ricercare una campagna nella lista delle campagne; 2. Il sistema visualizza a schermo la lista delle campagne e la barra di ricerca; 3. L'attore inserisce il nome della campagna all'interno della barra di ricerca; 4. La web app legge i caratteri inseriti dall'amministratore; 5. La web app carica l'item della lista contenenti i caratteri inseriti; 6. La web app visualizza a schermo la campagna ricercata.
Sequenza alternativa: Nessuna

Tabella 2.10 Descrizione caso d'uso: Ricerca Campagna

Inserisci Nuova Campagna
Questo caso d'uso si verifica qualora l'attore voglia inserire una nuova campagna.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login
Post-Condizioni: la nuova campagna esiste a sistema
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole inserire una nuova campagna; 2. L'attore seleziona la sezione contenente la form di inserimento della nuova campagna; 3. La web app carica la form; 4. La web app visualizza a schermo la form; 5. L'amministratore inserisce i dati richiesti all'interno dei rispettivi campi; 6. L'attore seleziona salva la nuova campagna; 6.1 **If** l'amministratore conferma l'inserimento: la web app salva la nuova campagna;
Sequenza alternativa: Inizia al punto 6.1. 6.2 **Else**: il sistema annulla l'inserimento della nuova campagna 7. Si torna al punto 4.

Tabella 2.11 Descrizione caso d'uso: Inserisci Nuova Campagna

Visualizza Area Campagna
Questo caso d'uso si verifica qualora l'attore voglia visualizzare l'area di svolgimento della campagna sulla mappa del territorio.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, la campagna esiste a sistema, l'attore è nella sezione della panoramica delle attività
Post-Condizioni: l'attore visualizza l'area della campagna
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare l'area di una campagna; 2. Il sistema visualizza a schermo la mappa del territorio; 3. Il sistema mostra a schermo le aree delle campagne; 4. L'attore sceglie la campagna della quale vuole vedere l'area cliccando sul marker;
Sequenza alternativa: Nessuna

Tabella 2.12 Descrizione caso d'uso: Visualizza Area Campagna

2.1.3 Gestione Segnalazioni

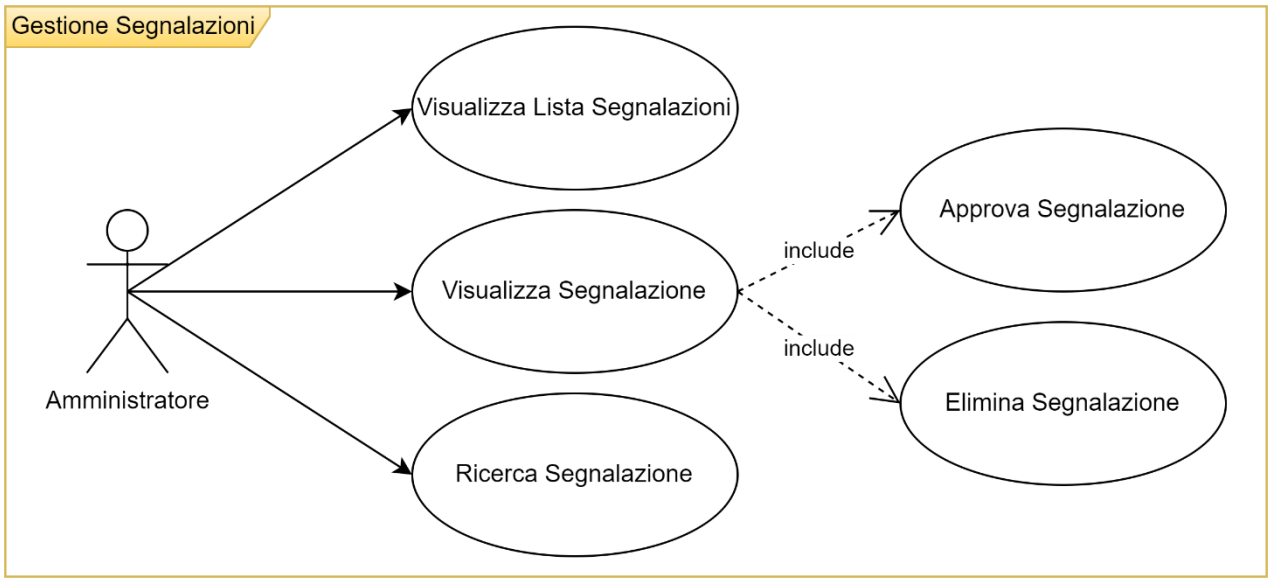

Figura 2.3 Diagramma casi d'uso gestione segnalazioni

Di seguito sono fornite delle tabelle contenenti le descrizioni dei casi d'uso di questo scenario:

Visualizza Lista Segnalazioni
Questo caso d'uso si verifica qualora l'attore voglia visualizzare la lista delle segnalazioni.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e le segnalazioni esistono a sistema
Post-Condizioni: l'attore visualizza la lista delle segnalazioni
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare la lista delle segnalazioni; 2. L'attore seleziona la sezione che contiene la lista delle segnalazioni; 3. La web app carica il contenuto della lista; 4. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.13 Descrizione caso d'uso: Visualizza Lista Segnalazioni

Ricerca Segnalazione
Questo caso d'uso si verifica qualora l'attore voglia ricercare una segnalazione all'interno della lista.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, le segnalazioni esistono a sistema e viene visualizzata la lista delle segnalazioni
Post-Condizioni: l'attore può ricercare la segnalazione e visualizzarla
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole ricercare una segnalazione nella lista delle segnalazioni; 2. Il sistema visualizza a schermo la lista delle segnalazioni e la barra di ricerca; 3. L'attore inserisce il nome della segnalazione all'interno della barra di ricerca; 4. La web app legge i caratteri inseriti dall'amministratore; 5. La web app carica l'item della lista contenenti i caratteri inseriti; 6. La web app visualizza a schermo la segnalazione ricercata.
Sequenza alternativa: Nessuna

Tabella 2.14 Descrizione caso d'uso: Ricerca Segnalazioni

Visualizza Segnalazione
Questo caso d'uso si verifica qualora l'attore voglia visualizzare una determinata segnalazione.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login ed ha selezionato una segnalazione
Post-Condizioni: l'attore visualizza la segnalazione
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare una segnalazione specifica; 2. Il sistema visualizza a schermo le segnalazioni; 3. L'attore seleziona la segnalazione che vuole visualizzare; 4. La web app carica il contenuto della segnalazione; 5. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.15 Descrizione caso d'uso: Visualizza Segnalazione

Approva Segnalazione
Questo caso d'uso si verifica qualora l'attore voglia approvare una segnalazione.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, la segnalazione esiste a sistema ed è visualizzata
Post-Condizioni: la segnalazione è approvata
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole approvare una segnalazione; 2. Il sistema visualizza a schermo la segnalazione da approvare; 3. L'attore avvia la procedura per approvare la segnalazione; 3.1 **If** l'attore conferma di voler approvare la segnalazione: il sistema aggiorna lo stato della segnalazione in "approvata"; 4. La web app visualizza a schermo la segnalazione con lo stato aggiornato.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla il processo per approvare la segnalazione 4. Si torna al punto 2.

Tabella 2.16 Descrizione caso d'uso: Approva Segnalazione

Elimina Segnalazione
Questo caso d'uso si verifica qualora l'attore voglia eliminare una segnalazione.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, la segnalazione esiste a sistema ed è visualizzata
Post-Condizioni: la segnalazione è eliminata
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole eliminare una segnalazione; 2. Il sistema visualizza a schermo la segnalazione da eliminare; 3. L'attore avvia la procedura per eliminare la segnalazione; 3.1 **If** l'attore conferma di voler eliminare la segnalazione: il sistema elimina la segnalazione; 4. La web app visualizza a schermo la lista delle segnalazioni aggiornata.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla il processo per eliminare la segnalazione 4. Si torna al punto 2.

Tabella 2.17 Descrizione caso d'uso: Elimina Segnalazione

2.1.4 Gestione Utenti e Operatori

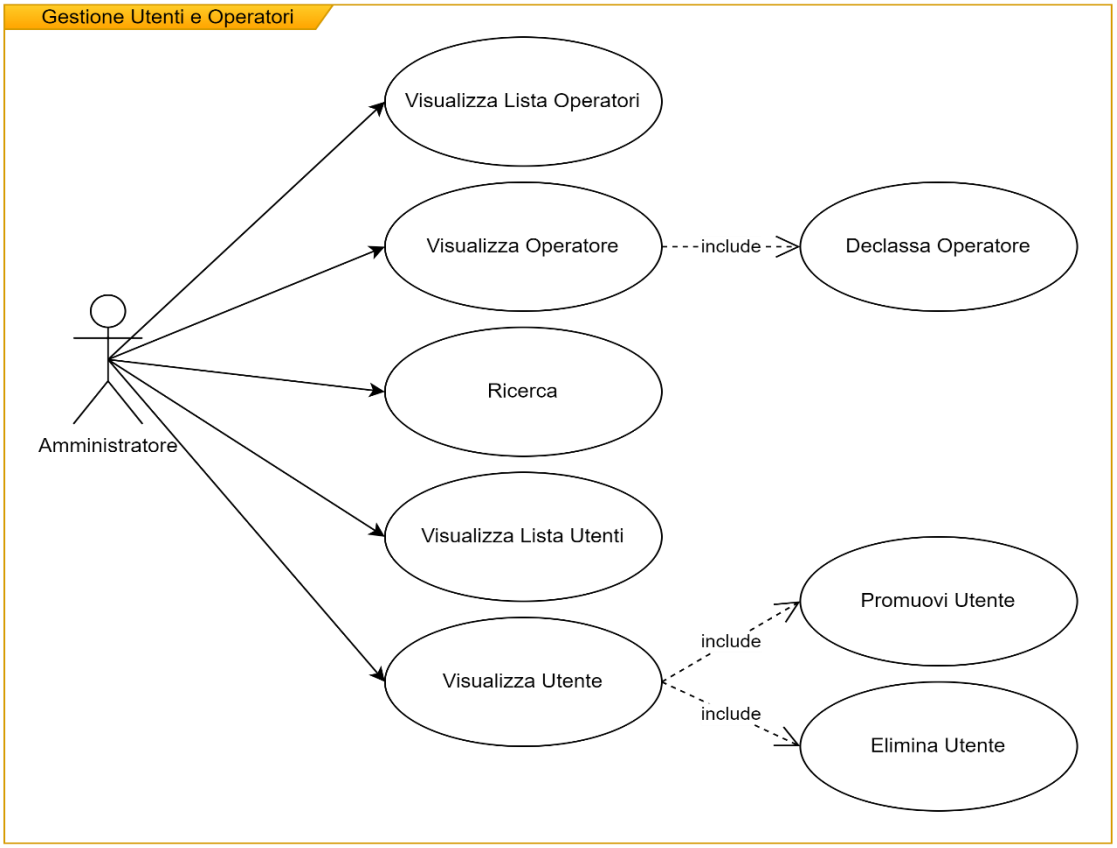

Figura 2.4 Diagramma casi d'uso Gestione Utenti e Operatori

Di seguito sono fornite delle tabelle contenenti le descrizioni dei casi d'uso di questo scenario:

Visualizza Lista Operatori
Questo caso d'uso si verifica qualora l'attore voglia visualizzare la lista degli operatori.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e gli operatori esistono a sistema
Post-Condizioni: l'attore visualizza la lista degli operatori
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare la lista degli operatori; 2. L'attore seleziona la sezione che contiene la lista degli operatori; 3. La web app carica il contenuto della lista; 4. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.18 Descrizione caso d'uso: Visualizza Lista Operatori

Visualizza Operatore
Questo caso d'uso si verifica qualora l'attore voglia visualizzare un operatore.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e l'operatore esiste a sistema
Post-Condizioni: l'attore visualizza le informazioni dell'operatore
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare le informazioni di un operatore; 2. Il sistema legge le informazioni dell'operatore; 3. Il sistema visualizza a schermo le informazioni dell'operatore.
Sequenza alternativa: Nessuna

Tabella 2.19 Descrizione caso d'uso: Visualizza Operatore

Declassa Operatore
Questo caso d'uso si verifica qualora l'attore voglia declassare un operatore e farlo tornare ad utente mobile base.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, l'operatore esiste a sistema ed è visualizzato
Post-Condizioni: l'operatore è stato declassato e non è più presente nella lista degli operatori
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole declassare un operatore; 2. Il sistema visualizza a schermo l'operatore da declassare ad utente; 3. L'attore avvia la procedura per declassare l'operatore; 3.1 **If** l'attore conferma di voler declassare l'operatore: il sistema declassa l'operatore; 4. La web app visualizza a schermo la lista degli operatori aggiornata.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla il processo per declassare l'operatore 4. Si torna al punto 2.

Tabella 2.20 Descrizione caso d'uso: Declassa Operatore

Ricerca
Questo caso d'uso si verifica qualora l'attore voglia ricercare un operatore o un utente all'interno delle rispettive liste.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, gli operatori e gli utenti esistono a sistema e sono visualizzati nelle rispettive liste
Post-Condizioni: l'attore può ricercare l'operatore o l'utente e visualizzarlo
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole ricercare un operatore o un utente all'interno delle rispettive liste; 2. Il sistema visualizza a schermo la lista e la barra di ricerca; 3. L'attore inserisce il nome dell'operatore o dell'utente all'interno della barra di ricerca; 4. La web app legge i caratteri inseriti dall'amministratore; 5. La web app carica l'item della lista contenenti i caratteri inseriti; 6. La web app visualizza a schermo l'operatore o l'utente ricercato ricercata.
Sequenza alternativa: Nessuna

Tabella 2.21 Descrizione caso d'uso: Ricerca

Visualizza Lista Utenti
Questo caso d'uso si verifica qualora l'attore voglia visualizzare la lista degli operatori.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e gli operatori esistono a sistema
Post-Condizioni: l'attore visualizza la lista degli operatori
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare la lista degli operatori; 2. L'attore seleziona la sezione che contiene la lista degli operatori; 3. La web app carica il contenuto della lista; 4. La web app visualizza a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.22 Descrizione caso d'uso: Visualizza Lista Utenti

Visualizza Utente
Questo caso d'uso si verifica qualora l'attore voglia visualizzare un utente.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login e l'utente mobile esiste a sistema
Post-Condizioni: l'attore visualizza le informazioni dell'utente
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare le informazioni di un utente; 2. Il sistema legge le informazioni dell'utente; 3. Il sistema visualizza a schermo le informazioni dell'utente.
Sequenza alternativa: Nessuna

Tabella 2.23 Descrizione caso d'uso: Visualizza Utente

Promuovi Utente
Questo caso d'uso si verifica qualora l'attore voglia promuovere l'utente ad operatore.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, l'utente mobile esiste a sistema ed è visualizzato
Post-Condizioni: l'utente mobile è stato promosso ad operatore e non è più presente nella lista degli utenti
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole promuovere un utente; 2. Il sistema visualizza a schermo l'utente da promuovere; 3. L'attore avvia la procedura di promozione; 3.1 **If** l'attore conferma di voler promuovere l'utente: il sistema promuove l'utente; 4. La web app visualizza a schermo la lista degli utenti aggiornata.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla il processo di promozione 4. Si torna al punto 2.

Tabella 2.24 Descrizione caso d'uso: Promuovi Utente

Elimina Utente
Questo caso d'uso si verifica qualora l'attore voglia eliminare un utente.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, l'utente mobile esiste a sistema ed è visualizzato
Post-Condizioni: l'utente mobile è stato eliminato
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole eliminare un utente; 2. Il sistema visualizza a schermo l'utente da eliminare; 3. L'attore avvia la procedura di eliminazione; 3.1 **If** l'attore conferma di voler eliminare l'utente: il sistema elimina l'utente; 4. La web app visualizza a schermo la lista degli utente aggiornata.
Sequenza alternativa: Inizia al punto 3.1. 3.2 **Else**: il sistema annulla il processo di eliminazione 4. Si torna al punto 2.

Tabella 2.25 Descrizione caso d'uso: Elimina Utente

2.1.5 Gestione statistiche

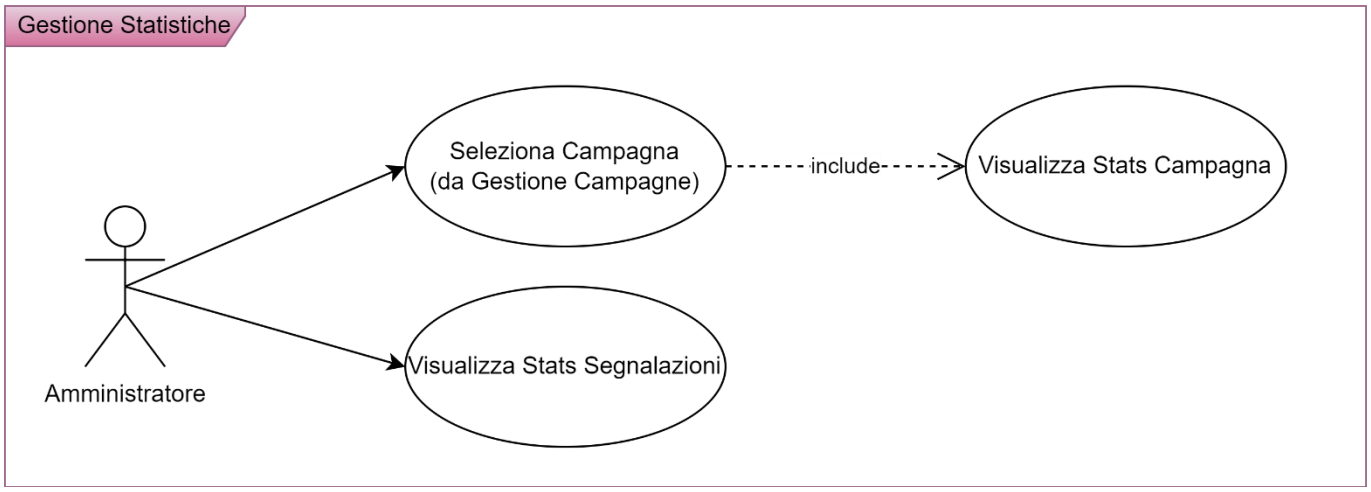

Figura 2.5 Diagramma Casi d'uso Gestione Statistiche

A pagina seguente sono fornite delle tabelle contenenti le descrizioni dei casi d'uso di questo scenario.

Visualizza Stats Campagna
Questo caso d'uso si verifica qualora l'attore voglia visualizzare le statistiche relative ad una campagna selezionata.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, la campagna esiste a sistema ed è selezionata
Post-Condizioni: l'attore visualizza le statistiche della campagna
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare le statistiche di una campagna; 2. L'attore seleziona la sezione che contiene le statistiche sulle campagne; 3. L'attore seleziona la campagna di cui vuole visionare le statistiche; 4. La web app carica le statistiche inerenti alla campagna selezionata; 5. La web app mostra a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.26 Descrizione caso d'uso: Visualizza Stats Campagna

Visualizza Stats Segnalazioni
Questo caso d'uso si verifica qualora l'attore voglia visualizzare le statistiche alle segnalazioni.
Attori: Amministratore
Pre-condizioni: l'attore ha effettuato login, le segnalazioni esistono a sistema
Post-Condizioni: l'attore visualizza le statistiche sulle segnalazioni
Sequenza degli eventi: 1. Il caso d'uso comincia quando l'attore vuole visualizzare le statistiche sulle segnalazioni; 2. L'attore seleziona la sezione che contiene le statistiche sulle segnalazioni; 3. La web app carica le statistiche inerenti alle segnalazioni; 4. La web app mostra a schermo il contenuto.
Sequenza alternativa: Nessuna

Tabella 2.27 Descrizione caso d'uso: Visualizza Stats Segnalazioni

2.2 Strumenti utilizzati

Conclusa l'analisi dei casi d'uso, che ha apportato maggiore chiarezza ai risvolti pratici che avrà l'applicazione web, il passo successivo è stato quello di individuare quali strumenti sono necessari allo sviluppo. La scelta di queste risorse è stata fondamentale per garantire l'efficacia e la completezza dell'applicazione.

Tra gli strumenti e le tecnologie utilizzate, si possono identificare i seguenti:

- **Ambiente di Sviluppo Integrato (IDE) - Visual Studio Code [1]**: *Visual Studio Code* è stato l'IDE principale, fornendo un'interfaccia intuitiva e potenti funzionalità per la scrittura del codice.

- **Linguaggi Fondamentali – HTML [2], CSS [3] e JavaScript [4]**: Questi linguaggi costituiscono il cuore della web app, consentendo la creazione di pagine web interattive, la formattazione dei contenuti e l'implementazione di funzionalità dinamiche.

- **Servizio di Backend – Firebase [5]**: *Firebase* è stato impiegato per gestire il backend dell'applicazione, inclusi il database in tempo reale, l'autenticazione degli utenti e la gestione dei file. Sono stati impiegati rispettivamente *Cloud Firestore*, *Firebase Authentication* e *Firebase Storage*.

- **Libreria – ReactJS [6]**: *ReactJS* ha svolto un ruolo chiave nella costruzione dell'interfaccia utente, semplificando la gestione dei componenti e l'aggiornamento dinamico delle pagine web.

- **Mappatura – OpenStreetMap [7] con Libreria Leaflet [8]**: L'integrazione di *OpenStreetMap* è stata realizzata attraverso la libreria *Leaflet*, che ha consentito di visualizzare mappe interattive all'interno dell'app.

- **Bootstrap [9]**: *Bootstrap* è stato utilizzato per la creazione di componenti grafici, garantendo uno stile coerente e una risposta adeguata ai dispositivi utilizzati dagli utenti.

- **Grafici Interattivi - Chart.js [10]**: È stata integrata la libreria *Chart.js* per generare grafici interattivi all'interno delle sezioni di statistiche, offrendo un'interpretazione visuale dei dati.

- **Browser - Google Chrome**: *Google Chrome* è stato il browser di riferimento durante lo sviluppo e i test per garantire la compatibilità e le prestazioni ottimali dell'app. Nel progetto presentato, in particolare, è stato utilizzato sia come browser sia come web server locale, per testare l'applicazione durante lo sviluppo.

- **Gestione delle Versioni - GitHub**: *GitHub* è stato utilizzato come piattaforma di controllo delle versioni per tenere traccia delle modifiche apportate al codice sorgente e mantenere l'integrità del progetto.

La selezione oculata di queste tecnologie e strumenti ha contribuito in modo significativo al successo complessivo della web app, migliorando l'efficienza dello sviluppo e garantendo la soddisfazione dei requisiti delineati. Nelle sezioni a seguire, verranno esaminati approfonditamente gli strumenti più importanti presentati sopra.

2.2.1 HTML, CSS e Javascript

HTML, acronimo di *HyperText Markup Language*, è un linguaggio di markup utilizzato per la creazione di pagine web che non include variabili, strutture dati o funzioni. Esso si basa sull'uso di tag di formattazione, ognuno dei quali rappresenta un ruolo specifico per il contenuto associato. L'insieme di questi tag, eventualmente concatenati, costituisce lo scheletro della pagina web. Il codice HTML viene interpretato dal browser, che genera la pagina visibile in base ai contenuti e ai tag specificati.

Sintatticamente, ogni elemento HTML è composto da un tag di apertura e uno di chiusura, con il contenuto situato tra di essi. Il tag di apertura è identificato da codice racchiuso tra parentesi angolari, mentre il tag di chiusura ha uno slash aggiunto dopo l'apertura della parentesi angolare.

Esistono anche tag a chiusura implicita, come <img>, che vengono utilizzati per inserire immagini direttamente nel punto desiderato della pagina, senza la necessità di un tag di chiusura.

È possibile specificare attributi all'interno dei tag, tra cui l'ID per identificare un elemento o regole di stile per definire il layout dell'elemento. Le regole di stile possono essere raggruppate in un file separato, solitamente denominato CSS, per una gestione più efficiente e la possibilità di riutilizzo su più elementi.

Il CSS (Cascading Style Sheets) è un linguaggio per definire le regole di formattazione degli elementi HTML, consentendo la descrizione di layout e stili per gli elementi. Le regole CSS possono essere applicate a elementi HTML utilizzando id univoci o classi condivise da più elementi. In caso di conflitto tra regole diverse, il CSS segue una gerarchia predefinita: le regole *inline* hanno la massima priorità, seguite da quelle associate all'id e poi da quelle associate alla classe o al tipo di tag.

HTML e CSS strutturano le pagine web ma generano contenuto statico. JavaScript, un linguaggio di programmazione, consente di interagire con l'HTML e modificarne dinamicamente il contenuto, creando, eliminando o modificando elementi. JavaScript deriva dalla sintassi del linguaggio C e può essere utilizzato per lavorare in diversi contesti, ma è ampiamente impiegato per modificare dinamicamente pagine HTML in risposta a eventi specifici, come clic del mouse.

2.2.2 Firebase

Firebase rappresenta una potente piattaforma per lo sviluppo di applicazioni mobili e web, suddivisa in quattro categorie principali: *Creazione*, *Rilascio* e *Monitoraggio*, *Analisi* e *Coinvolgimento*. Nel contesto dello sviluppo della web app oggetto di questo elaborato, sono stati impiegati tre dei suoi componenti appartenenti alla categoria "Creazione": *Cloud Firestore* per il database, *Firebase Storage* per l'archiviazione dei file e *Firebase Authentication* per la gestione dell'autenticazione degli utenti.

Cloud Firestore è un database *NoSQL* che si distingue per la sua struttura flessibile. La sua organizzazione si basa su "raccolte", ciascuna delle quali contiene documenti distinti, ognuno contrassegnato da un id univoco. All'interno di ciascun documento è possibile definire campi con dati di vario tipo, consentendo una modellazione dei dati dinamica e senza rigidezza schematica. Ciò significa che è possibile adattare il database alle esigenze specifiche dell'applicazione senza dover rispettare uno schema fisso. Nelle sezioni successive sarà approfondita la struttura dati utilizzata in questo progetto.

Firebase Storage offre una soluzione affidabile per l'archiviazione e il recupero di file di qualsiasi formato. È possibile organizzare i file in cartelle, semplificando la gestione dei dati multimediali come immagini, video o documenti. Questo servizio si integra in modo trasparente con altri componenti Firebase, facilitando l'accesso ai file memorizzati.

Firebase Authentication semplifica la gestione degli utenti autenticati. Fornisce un sistema di registrazione e accesso sicuro, consentendo agli utenti di autenticarsi e accedere alle risorse dell'applicazione in modo controllato. Questo componente è particolarmente utile per garantire la sicurezza e la privacy dei dati degli utenti.

2.2.3 ReactJS

ReactJS è una libreria *JavaScript* che ha ottenuto un notevole successo per la sua capacità di creare interfacce utente dinamiche e reattive per le applicazioni web.

Innanzitutto, *React* si basa su un concetto fondamentale: i componenti. In pratica, un componente *React* è una piccola parte dell'interfaccia utente che è responsabile di un aspetto specifico o di una funzionalità dell'applicazione. Questi componenti possono essere progettati per essere riutilizzabili, il che semplifica la creazione e la manutenzione del codice. Un altro punto di forza di *React* è il concetto di *Virtual DOM* (DOM Virtuale): invece di aggiornare direttamente il DOM del browser quando cambiano i dati, *React* crea una copia virtuale del DOM in memoria e confronta questa copia con il DOM reale. Questo approccio consente di minimizzare le operazioni costose di aggiornamento del DOM, rendendo l'applicazione più efficiente e reattiva. Inoltre, un aspetto particolare di questa libreria è l'utilizzo del *JSX* (JavaScript XML), una sintassi che permette di definire la struttura dell'interfaccia utente all'interno del codice *JavaScript*, rendendo più chiara la definizione delle componenti e facilitando la creazione di interfacce utente complesse.

La gestione dello stato e delle proprietà delle componenti è un altro aspetto chiave di *React* e consente di creare componenti dinamiche e interattive. Lo stato è utilizzato per memorizzare dati che possono cambiare nel tempo, mentre le proprietà (*props*) vengono utilizzate per passare dati tra componenti. Quest'ultime sono dotate anche di un ciclo di vita ben definito, su cui eseguire operazioni specifiche in momenti precisi. Inoltre, l'introduzione di funzioni speciali chiamate *hook* in versioni recenti di *React* ha semplificato notevolmente la gestione dello stato e del ciclo di vita delle componenti funzionali: queste, infatti, possono essere riutilizzabili e complesse. Tra gli *hook* più importanti, utilizzati nello sviluppo della web app e che verranno presentati più avanti, è possibile trovare**:**

- *useState*, per gestire lo stato del componente, tenendo traccia delle variabili di stato per permettere di reagire a eventuali modifiche di tali variabili;

- *useEffect*, per la gestione del ciclo di vita del componente e permettere a quest'ultimo di rispondere alle modifiche di stato e alle azioni dell'utente in modo reattivo;

- *useContext*, che consente ai componenti di accedere ai dati all'interno di un contesto creato utilizzando *createContext*. Quest'ultima è una funzione utilizzata per creare un contesto, che può essere utilizzato per la condivisione delle proprietà dati senza doverle passare manualmente;

- *useRef*, per creare e gestire riferimenti a elementi del DOM o a valori mutabili all'interno di componenti funzionali. È stato utilizzato soprattutto per mantenere riferimenti a valori che non devono causare il rendering del componente quando cambiano;

Infine, *React* è spesso utilizzato in combinazione con librerie di *routing* come *React Router DOM* per gestire la navigazione tra le diverse pagine di un'applicazione web, semplificando la gestione delle rotte dell'applicazione. L'hook utilizzato, infatti, è stato *useNavigate*. Inoltre, supporta il Server-Side Rendering (SSR), una tecnica che prevede la creazione e l'invio dell'HTML di una pagina direttamente dal server al browser del cliente. Questo migliora le prestazioni iniziali, favorisce l'indicizzazione da parte dei motori di ricerca e aumenta l'accessibilità. In seguito, il JavaScript client prende il controllo per rendere l'applicazione interattiva. L'uso del SSR è vantaggioso in determinate situazioni, ma richiede una maggiore complessità nell'implementazione.

Sfruttando, quindi, le potenzialità di questa libreria lo sviluppo della web app è risultato molto semplice.

2.2.4 Openstreetmap con libreria Leaflet

OpenStreetMap (OSM) è un progetto collaborativo globale che mira a creare una mappa dettagliata e aggiornata del mondo, basata su dati geografici aperti e contributi da parte degli utenti. Tale strumento è stato utilizzato in combinazione con Leaflet, una libreria JavaScript open-source ampiamente utilizzata per la visualizzazione di mappe basate su dati geografici, molto facile da integrare in progetti web. Ciò ha permesso di rendere interattiva la mappa, consentendo l'inserimento di marker personalizzati, pop-up informativi, sistema di zoom personalizzato e possibilità di disegnare poligoni.

Grazie, inoltre, all'utilizzo delle estensioni di questa libreria è stato possibile l'utilizzo del servizio di geocodifica offerto da *Nominatim-geocode*.

2.2.6 Bootstrap

Bootstrap è un framework open-source molto utilizzato per lo sviluppo di siti applicazioni web a livello front-end. È stato creato da Twitter e fornisce una raccolta di strumenti, componenti e stili CSS predefiniti per semplificare la progettazione e lo sviluppo web. La scelta di questo strumento è stata dettata per varie ragioni:

- Responsività del design, per adattare la web app a dispositivi di diverse dimensioni;
- Utilizzo di componenti HTML e CSS predefiniti, che ha permesso un notevole risparmio delle tempistiche. Nel progetto sono state utilizzate componenti come bottoni, barre di ricerca, componenti modali per i pop-up di conferma, form, sistemi di griglia e di creazione del menù;
- Utilizzo di temi personalizzati per l'aspetto e lo stile dell'interfaccia grafica.

2.2.7 ChartJS

Chartjs è una libreria JavaScript open source molto utilizzata per la creazione di grafici interattivi e dinamici all'interno di pagine web. È una libreria flessibile e potente che consente agli sviluppatori di visualizzare dati e informazioni in modo visivamente accattivante. Offre una vasta gamma di grafici interattivi tra cui scegliere, in base alle proprie esigenze: nel caso di questo progetto, sono stati utilizzati solo grafici a barre, più chiari ed utili per il confronto dei dati.

2.3 Progettazione dei dati

Come si intuisce dall'analisi svolta, la progettazione dei dati è stata un punto cardine di tutto lo sviluppo, in quanto si è dovuta creare una struttura in comune tra app mobile e web prontamente organizzata per gestire le esigenze di entrambi i prodotti software. In questa sezione verrà discussa approfonditamente le composizioni del *Cloud Firestore* e del *Firebase Storage* adottate.

2.3.1 Cloud Firestore

Il *Cloud Firestore Database* è composto in totale da tre raccolte principali:

- *campagne*: contiene le informazioni relative alle campagne. È qui che si inseriranno le nuove campagne dalla web app;
- *segnalazion*i: contiene le segnalazioni con le rispettive descrizioni di ogni rifiuto;
- *users*: contiene le informazioni riguardanti utenti mobile e operatori;

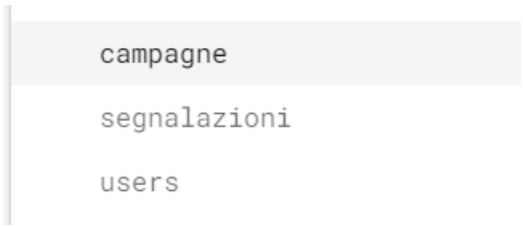

Figura 2.6 Raccolte

La raccolta *campagne* è composta da una serie di documenti che rappresentano le singole campagne, ciascuna identificata da un ID univoco. All'interno di ogni documento troviamo i seguenti campi:

- *Titolo (stringa)*: contiene il titolo o il nome della campagna;
- *Tipo (stringa)*: indica il tipo di campagna;
- *Inizio (timestamp)*: indica la data e l'ora di inizio della campagna;
- *Fine (timestamp)*: indica la data e l'ora di termine della campagna;
- *Immagine (stringa)*: contiene l'URL dell'immagine associata alla campagna archiviata su Firebase Storage;
- *In Corso (booleano)*: valore booleano che indica se la campagna è attualmente in corso o meno. Viene utilizzato come flag e può avere due valori: "*true*" se la campagna è in stato di svolgimento, "*false*" se è terminata o deve ancora cominciare;
- *Info (stringa)*: contiene informazioni descrittive sulla campagna;
- *NumPartecipanti (numero)*: numero di partecipanti attuali alla campagna;

- *Operatori (array)*: array di stringhe che rappresenta gli operatori associati alla campagna, indicati con il loro ID univoco;

- *Partecipanti (array)*: array di stringhe che rappresenta i partecipanti alla campagna, indicati con il loro ID univoco;

- *Posizione (array)*: array di oggetti mappa che rappresenta le coordinate geografiche di punti di interesse associati alla campagna. Ogni oggetto mappa ha due campi: *latitude (latitudine)* e *longitude (longitudine).*

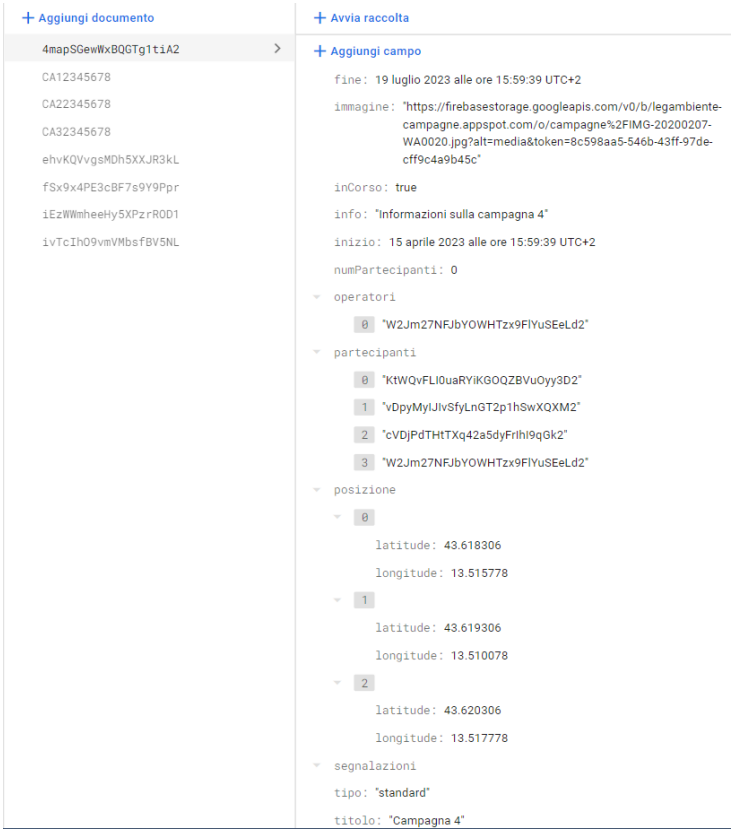

Figura 2.7 Esempio di documento 'campagna'

Per quanto riguarda la raccolta *segnalazioni*, ogni documento rappresenta una segnalazione effettuata da un utente mobile, ed è costituito dai seguenti campi:

- *dataeora(timestamp)*: contiene la data e l'orario della segnalazione, cioè, registra il momento in cui è stata effettuata la segnalazione;

- *idCampagna(stringa)*: stringa che rappresenta un identificativo unico associato alla campagna a cui appartiene la segnalazione;

- *idUtente(stringa)*: stringa che rappresenta l'identificativo unico dell'utente che ha effettuato la segnalazione;
- *immagine(stringa)*: stringa che contiene un URL che punta all'immagine associata alla segnalazione archiviata nel Firebase Storage;
- *luogo(mappa)*: mappa che rappresenta il luogo associato alla segnalazione. È composto da due campi: *latitude (latitudine)* e *longitude (longitudine)*, che indicano le coordinate geografiche del luogo;
- *note(stringa)*: stringa che può contenere note o descrizioni aggiuntive relative alla segnalazione;
- *quantità(numero)*: numero intero che rappresenta la quantità o il numero associato alla segnalazione;
- *tipo(stringa)*: stringa che indica il tipo di segnalazione;
- *validazione(stringa)*: stringa che indica lo stato di validazione della segnalazione. È utilizzato come flag e può avere tre valori: “si” (segnalazione approvata), “in corso” (segnalazione da validare) e “no” (segnalazione non approvata);

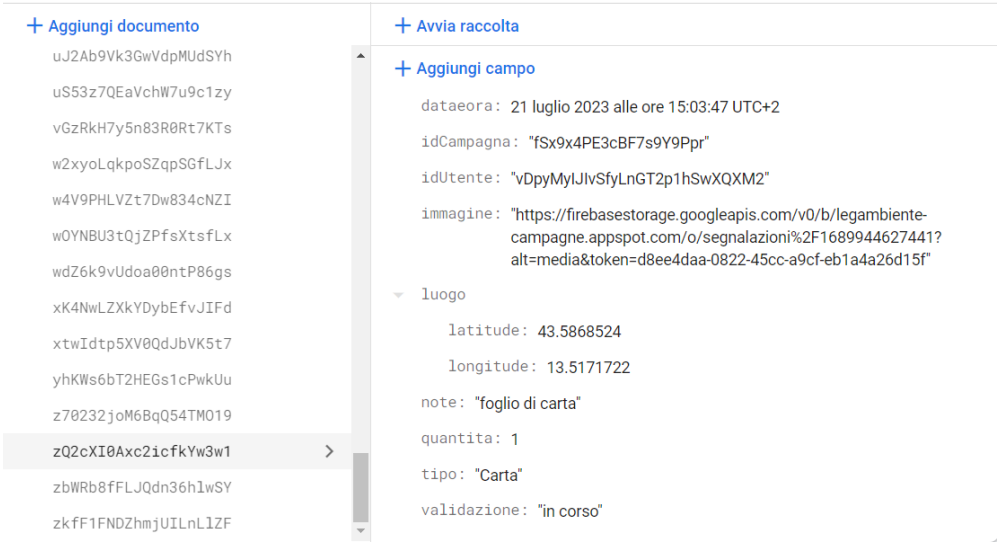

Figura 2.8 Esempio documento segnalazione

Infine, la raccolta *users* è rappresentata da tutti gli utenti mobile e gli operatori, ognuno dei quali è rappresentato da un documento, in cui troviamo i seguenti campi:

- *name(stringa)*: contiene il nome dell'utente;
- *dob(stringa)*: stringa che rappresenta la data di nascita dell'utente;
- *email(stringa)*: stringa che rappresenta l'email dell'utente;
- *isOperator(booleano)*: valore booleano che indica se l'utente è operatore o meno. È utilizzato come flag e può avere due valori: "true" (l'utente è un operatore) o "false" (l'utente non è un operatore).

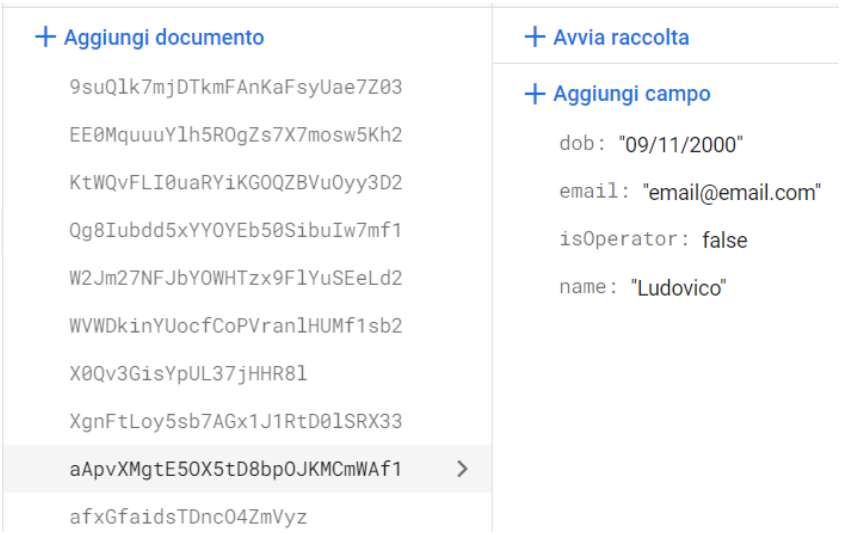

Figura 2.9 Esempio di documento utente

2.3.2 Storage

Il prodotto Firebase Storage appare più semplice essendo utilizzato solo per l'archiviazione delle immagini delle segnalazioni e delle campagne. Si precisa che attualmente sono presenti altre due raccolte per le immagini inerenti agli articoli e alle immagini del profilo degli utenti che non sono state utilizzate durante lo sviluppo. Pertanto, verranno illustrate solo le raccolte presenti nella figura sottostante:

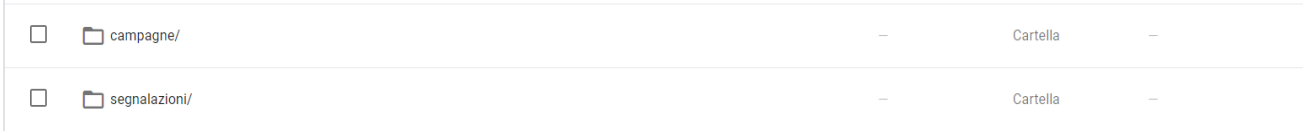

Figura 2.10 Cartelle di archiviazione utilizzate nel firebase storage

Nella cartella *campagne/* si trovano tutte le immagini inerenti alle campagne rappresentanti, ad esempio, il luogo di svolgimento o il logo di Legambiente, mentre nella cartella *segnalazioni/* sono presenti le foto dei rifiuti inviate dagli utenti tramite l'applicazione mobile.

2.4 Struttura del progetto

Il passo finale della fase di progettazione è quello di individuare la struttura che dovrà avere l'applicazione web che meglio rispecchia le esigenze. A tal proposito, si è optato per l'adozione di una struttura basata sul classico design pattern *Model-View-Controller* [11], arricchita da altri *package* di supporto. Tale pattern architetturale, utilizzato per una vasta gamma di applicazioni software, è stato riadattato al contesto dello sviluppo web conferendo alle sue componenti i seguenti ruoli:

- *Model*: contiene i metodi di accesso ai dati;
- *View*: si occupa di visualizzare i dati all'utente e gestisce l'interazione fra quest'ultimo e l'infrastruttura sottostante;
- *Controller*: riceve i comandi dell'utente attraverso il View e reagisce eseguendo delle operazioni che possono interessare il Model e che portano generalmente ad un cambiamento di stato del View.

Nel rispetto di quanto osservato, la struttura del progetto è la seguente:

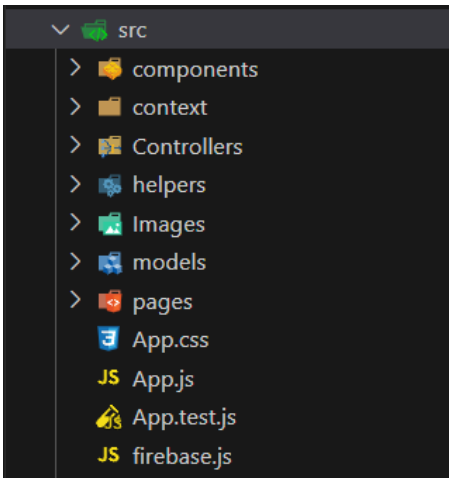

Figura 2.11 Struttura principale del progetto

Di seguito sono puntualizzati tutti i package mostrati nella figura:

- *Components*: contiene alcune delle componenti grafiche, con i rispettivi file css, utilizzati per comporre le views sotto la cartella pages;
- *Context*: contiene tutti i processi di autenticazione dell'utente, il login e il logout;
- *Controllers*: contiene le classi deputate a gestire la logica dell'applicazione, compresi i metodi per il recupero dei dati dal Cloud Firestore Database e la loro manipolazione;
- *Helpers*: contiene la classe per la gestione delle immagini della web app;
- *Images*: contiene le immagini di default come il logo di Legambiente e immagini di profilo neutre per operatori e utenti;
- *Models*: contiene che descrivono le proprietà degli oggetti rappresentati nell'applicazione;
- *Pages*: contiene le viste principali;
- *App.css*: file che racchiude alcuni stili css adottati;
- *App.js*: file principe dell'applicazione che contiene il punto iniziale in cui React viene utilizzato per costruire e renderizzare l'interfaccia utente dell'applicazione. Sono presenti le varie componenti create e le rotte dell'applicazione;
- *App.test.js*: file associato al testing dell'applicazione (non utilizzato);
- *Firebase.js*: contiene l'inizializzazione e la configurazione dei prodotti *Firebase* utilizzati.

Capitolo 3

Implementazione

In questo capitolo verrà illustrata la fase di implementazione della web app, mostrando le funzionalità più importanti e le soluzioni adottate per la loro realizzazione, con specifici riferimenti al codice sorgente. Successivamente, verranno presentati degli esempi di esecuzione del software.

3.1 Logica dell'app e soluzioni adottate

In questa sezione verranno discussi alcuni dei caratteri implementativi più importanti del progetto. Si inizierà descrivendo il file App.js, che contiene tutte le rotte dell'applicazione. Seguiranno, poi, il menù di navigazione e la fase di login dell'amministratore, fondamentale per l'utilizzo di tutto l'applicativo, e poi si proseguirà con la visualizzazione della panoramica delle attività, concepita come una sorta di homepage dell'applicazione. Verranno poi discusse le gestioni:

- delle campagne, evidenziando in particolar modo le fasi di terminazione di una campagna e di inserimento di una nuova campagna, punto cardine del progetto;
- delle segnalazioni, evidenziando in particolar modo la loro validazione;
- degli utenti e degli operatori;
- delle statistiche sulle segnalazioni e su una determinata campagna selezionata dall'amministratore.

3.1.1 App.js

Questo file rappresenta il componente principale di tutta l'applicazione. Qui, vengono importati vari moduli da *React* e da altre librerie. Questi moduli includono: *React Router* per la gestione delle rotte dell'applicazione, un componente *ProtectedRoute*, un contesto di autenticazione, e diverse pagine e componenti personalizzati utilizzati nell'applicazione.

Il componente App è la radice dell'applicazione *React* ed è definito come una funzione. Questa funzione restituisce gli elementi JSX che costituiscono l'interfaccia utente dell'applicazione.

Come mostrato in figura 3.1, l'app utilizza *<Router>* per gestire le rotte e la navigazione all'interno dell'applicazione, definite utilizzando *<Routes>* e *<Route>,* e ciascuna rotta è associata a un componente specifico. La maggior parte di esse sono protette e richiedono l'autenticazione

Figura 3.1 Rotte applicazione

dell'utente per accedervi e questo è gestito tramite il componente *ProtectedRoute*. Il contesto di autenticazione è avvolto attorno alle *route*, consentendo l'accesso alle informazioni sull'utente autenticato. Alla fine, vengono visualizzati un componente *Footer* e una *Navbar* che forniscono una struttura di base per la navigazione all'interno dell'applicazione.

3.1.2 Login

Per gestire la fase di login è stato inizializzato il prodotto *Firebase Authentication* nel file *firebase.js*. In seguito, nel package *Context*, è stato creato un file *AuthContext.jsx* in cui sono contenute tutte le importazioni e tutti i metodi per gestire il processo di autenticazione. Come mostrato nella figura 3.2, è stato creato uno *UserContext* per la condivisione dei dati sull'utente autenticato tra i componenti dell'applicazione.

```
const UserContext = createContext()

export const AuthContextProvider = ({ children }) => {
  const [user, setUser] = useState({});


  const login = (email, password) =>  {
   return signInWithEmailAndPassword(auth, email, password)
  }

  const logout = () => {
    return signOut(auth)
  }

  useEffect(() => {
    const unsubscribe = onAuthStateChanged(auth, (currentUser) => {
      console.log(currentUser);
      setUser(currentUser);
    });
    return () => {
      unsubscribe();
    };
  }, []);

  return (
    <UserContext.Provider value={{ user, logout, login }}>
      {children}
    </UserContext.Provider>
  );
};

export const UserAuth = () => {
  return useContext(UserContext);
};
```

Figura 3.2 Codice AuthContext

Il provider di questo componente è *AuthContextProvider*, che accetta un componente figlio '*children*': qui viene sfruttato '*user*' per tenere traccia dell'utente autenticato, e sono state create le funzioni di login e logout che ritornano rispettivamente i metodi importati da *firebase/auth,* denominati *signInWithEmailAndPassword* e *signOut*. Viene, poi, utilizzato *useEffect* per sottoscrivere un ascoltatore di cambiamenti nello stato di autenticazione dell'utente. Quando l'autenticazione cambia, *UserContext* viene aggiornato con le informazioni sull'utente corrente. Infine, per permettere agli altri componenti di accedere alle informazioni e alle funzioni di accesso/disconnessione tramite il contesto, viene definita la funzione *UserAuth*.

3.1.3 Menù e visualizzazione panoramica

Per il menù di navigazione è stato creato un componente *Navbar* nel package *Components*. Qui sono stati importati l'hook *useNavigate* di *React* e vari componenti bootstrap utilizzati per la struttura di questo componente, tra cui *Nav* per creare una lista di elementi di navigazione, *NavDropdown* per creare elementi di elenco a discesa e *OffCanvas* per la comparsa laterale. La barra di navigazione è resa dinamica tramite un *map* che viene utilizzato per renderizzare il menu solo quando *expand* è *false*, utile per gestire l'Offcanvas. Infatti, quando viene cliccato sul *NavbarToggle* del menù questo valore diventa *true* e il menù si apre lateralmente.

```
let navigate=useNavigate();
return (
    <>
      {[false].map((expand) => (
        <Navbar key={expand} style={{ backgroundColor: '#e5be01' }} expand={expand} className="mb-1">
            <Container fluid>
             <Navbar.Toggle aria-controls={`offcanvasNavbar-expand-${expand}`} />
              <img src={Legambiente_logo} alt="Logo" width="80" height="50" style={{position:"absolute", left:100}} />
                <Navbar.Offcanvas id={`offcanvasNavbar-expand-${expand}`}
                aria-labelledby={`offcanvasNavbarLabel-expand-${expand}`} placement="start">
```

Figura 3.3 Gestione offcanvas barra di navigazione

Quest'ultimo è diviso in due sezioni: *header*, dove è contenuto il logo di Legambiente, e *body*, in cui sono presenti i *NavDropdown* con le opzioni di navigazione. Anche qui è presente *expand* per gestire l'apertura dei singoli menù a discesa.

Come si evince dalla figura 3.3, viene utilizzata la variabile "navigate" per la navigazione: essa è inizializzata come *let* in quanto il suo valore può cambiare nel corso del tempo a seconda del nome della rotta passatole dall'hook *useNavigate*.

```
<Offcanvas.Header closeButton>
  <Offcanvas.Title id={`offcanvasNavbarLabel-expand-${expand}`}> <img src={Legambiente_logo} alt="Logo" width="60" height="40"/></Offcanvas.Title>
</Offcanvas.Header>
 <Offcanvas.Body>
     <Nav className="justify-content-end flex-grow-1 pe-3">

      <NavDropdown title="Panoramica Attività" id={`offcanvasNavbarDropdown-expand-${expand}`}>
        <NavDropdown.Item href="#action1" onClick={() => navigate('/')}>Visualizza Mappa</NavDropdown.Item>
        <NavDropdown.Divider />
        <NavDropdown.Item href="#action2" onClick={() => navigate('/newCampagna')}>Nuova Campagna</NavDropdown.Item>
        <NavDropdown.Divider />
        <NavDropdown.Item href="#action3" onClick={() => navigate('/listacampagne')}>Lista Campagne</NavDropdown.Item>
      </NavDropdown>
```

Figura 3.4 Header e Body dell'Offcanvas

La pagina di visualizzazione della mappa rappresentante la panoramica delle attività è data dalla rotta *"/"* ed è la prima ad essere visualizzata dopo il login. Per la costruzione di questa *view* è stata sfruttata la libreria *Leaflet*, in combinazione ai componenti *React* e ai controller creati per la gestione delle segnalazioni e delle campagne, all'interno di un componente denominato *MyMapComponent*. In esso abbiamo l'inizializzazione e la personalizzazione della mappa, dei marker e dei pop-up presenti ma, prima di fare ciò, sono state effettuate delle operazioni preliminari in uno *useEffect* per estrapolare le informazioni necessarie da associare a questi elementi. Nella fattispecie sono presenti due metodi importanti:

- *unsubscribeSegnalazioni*, che ottiene i dati dal metodo *getSegnalazioniDaApprovare* presente nel *SegnalazioneController*, ordina le segnalazioni in base alla data e all'ora. Quest'ultime vengono poi elaborate per aggiungere un numero unico a ciascuna e per estrarre le coordinate di luogo e altre informazioni;

```
const unsubscribeSegnalazioni = SegnalazioneController.getSegnalazioniDaApprovare((segnalazioni) => {
  const segnalazioniOrdinate = segnalazioni.sort((a, b) => new Date(a.dataOra) - new Date(b.dataOra));
  // Assegna un numero unico a ciascuna segnalazione
  const segnalazioniConNumeri = segnalazioniOrdinate.map((segnalazione, index) => ({...segnalazione, numero: index + 1,}));
  // Estrae solo le coordinate luogo dalla lista delle segnalazioni e altre informazioni utili a popolare il popup
  const infoSegnalazione = segnalazioniConNumeri.map((segnalazione) => ({
    latitude: segnalazione.luogo.latitude,
    longitude: segnalazione.luogo.longitude,
    id: segnalazione.id,
    tipo: segnalazione.tipo,
    dataOra: segnalazione.dataOra,
    numero: segnalazione.numero,
  }));
  setCoordinateSegnalazione(infoSegnalazione);
});
```

Figura 3.5 Codice metodo unsubscribeSegnalazioni

- *unsubscribeCampagne*, che ottiene i dati dal metodo *getCampagne*presente nel *CampagnaController*, calcola il punto medio tra le coordinate della campagna. Quest'ultime sono anche utilizzate per definire il perimetro dell'area geografica.

```
const unsubscribeCampagne = CampagnaController.getCampagne((campagne) => {
  // Estrae il punto medio di ciascuna campagna e crea un array di coordinate
  const coordinateCampagneArray = campagne.map((campagna) => {
    const campagnaCoordinates = campagna.posizione.map((coordinate) => [coordinate.latitude, coordinate.longitude]);
    // Calcola il punto medio delle coordinate
    const sumLatitude = campagnaCoordinates.reduce((sum, point) => sum + point[0], 0);
    const sumLongitude = campagnaCoordinates.reduce((sum, point) => sum + point[1], 0);
    const averageLatitude = sumLatitude / campagnaCoordinates.length;
    const averageLongitude = sumLongitude / campagnaCoordinates.length;
    return {
      latitude: averageLatitude,
      longitude: averageLongitude,
      id: campagna.id,
      nome: campagna.titolo,
      stato: campagna.inCorso,
      periodo: `da ${campagna.inizio.toLocaleString()} a ${campagna.fine.toLocaleString()}`,
      campagnaCoordinates
    };
  });
  setCoordinateCampagne(coordinateCampagneArray);
});
```

Figura 3.6 Codice metodo unsubscribeCampagne

Una volta recuperate ed elaborate le informazioni e personalizzato lo stile dei marker e dei popup, bisogna associare i dati a questi elementi. Per le segnalazioni, come si evince dalla figura 3.7, si utilizza il componente *MarkerClusterGroup* di *react-leaflet* per raggruppare i marker delle segnalazioni. A loro volta tali componenti marker sono posizionati sulla mappa tramite l'attributo position che va a leggere le coordinate in *infoSegnalazione*.

```
<MarkerClusterGroup iconCreateFunction={createClusterCustomIcon}>
  {coordinateSegnalazione.map((infoSegnalazione, index) => (
    <Marker key={index} position={[infoSegnalazione.latitude, infoSegnalazione.longitude]} icon={SegnalazioneIcon}>
      <Popup>
        <Link to={`/segnalazione/${infoSegnalazione.id}`} className="navlink">
          <p>Numero Segnalazione: {infoSegnalazione.numero}</p>
        </Link>
        <p>Tipo: {infoSegnalazione.tipo}</p>
      </Popup>
    </Marker>
  ))}
</MarkerClusterGroup>
```

Figura 3.7 Codice per l'organizzazione delle segnalazioni sulla mappa

Per le campagne, invece, non è stato necessario il raggruppamento; pertanto, è stato utilizzato solo il componente marker che, tramite l'attributo *position*, va a leggere le coordinate del punto medio della campagna in *infoCampagna*. Qui, è stato anche utilizzato il componente *polygon* di *react-*

leaflet per la rappresentazione dell'area di svolgimento: tramite l'attributo positions va a leggere l'array di coordinate recuperato nel metodo *unsubscribeCampagne*.

```
{coordinateCampagne.map((infoCampagna, index) =>infoCampagna.stato && (
    <Marker key={index} position={[infoCampagna.latitude, infoCampagna.longitude]} icon={CampagnaIcon}>
      <Polygon key={index}
        positions={infoCampagna.campagnaCoordinates} // Utilizziamo l'array di coordinate per definire il perimetro del poligono
        color="blue" fillColor="lightblue" fillOpacity={0.5}>
      </Polygon>
      <Popup>
          <Link to={`/campagna/${infoCampagna.id}`} className="navlink">
            <p>Nome Campagna: {infoCampagna.nome}</p>
          </Link>
          <p>Periodo: {infoCampagna.periodo}</p>
          <p>Stato: {infoCampagna.stato ? "In Corso" : "Terminata"}</p>
      </Popup>
    </Marker>
))}
```

Figura 3.8 Codice per l'organizzazione delle campagne sulla mappa

In seguito, il componente *MyMapComponent* è richiamato nella *view* di visualizzazione della mappa.

3.1.4 Gestione delle campagne

Per la gestione delle campagne è utile, più che soffermarsi sulla costruzione della lista e in che modo viene visualizzata, a come viene manipolata la singola campagna e quali sono le informazioni più importanti recuperate. Nella fattispecie è analizzato il processo di terminazione di una campagna, eseguibile sia dalla lista campagne che dalla scheda della campagna. Per la visualizzazione di quest'ultima, il metodo principale creato all'interno del controller è il seguente:

```
//preleva una campagna con uno specifico id
 static getCampagnaById(campagnaId, callback) {
   const campagnaRef = doc(db, "campagne", campagnaId);
   getDoc(campagnaRef).then((doc) => {
       if (doc.exists()) {
         const data = doc.data();
         const campagna = new Campagna(doc.id, data);
         callback(campagna);
       } else {console.log("Nessuna campagna trovata con l'ID fornito.");}
     }).catch((error) => {console.error("Errore durante il recupero della campagna:", error);});
 }
```

Figura 3.9 Codice metodo getCampagnaById

Qui, viene passato un parametro *campagnaId*, che rappresenta l'ID univoco della campagna che si desidera recuperare e creato un riferimento al documento all'interno *collection* "campagne" tramite l'oggetto *doc*. Successivamente viene chiamato il metodo *getDoc* importato da *firestore*, precedentemente inizializzato e configurato nel file firebase.js, per il recupero dei dati associato al documento. Attraverso, poi la funzione di *callback* fornita come secondo argomento del metodo, viene verificata l'esistenza del documento e, se esiste, viene creato un nuovo oggetto "*Campagna*". Inoltre, tale metodo è analogo nel suo funzionamento a *getCampagne*, menzionato nella sezione 3.1.3, con la differenza che in quest'ultimo vengono recuperate tutte le campagne nella raccolta e

utilizzato per la costruzione della view "*Lista Campagne*". Quest'ultima è costruita con una serie di componenti denominate card, di dimensione fissa, in cui sono contenute informazioni come titolo, luogo, periodo di svolgimento e l'immagine.

Recuperata la campagna bisogna ora visualizzarla nell'apposita scheda, pertanto viene richiamato nel componente il metodo appena illustrato. I dati sono manipolati e organizzati in vari modi e i più importanti sono:

- Posizione: questo dato è manipolato grazie a due metodi che sfruttano la libreria *nominatim-geocoder,* rappresentati nella figura 3.10:

```
useEffect(() => {
  const geocoder = new NominatimGeocoder();
  const calcolaPuntoMedio = () => {
    if (!campagna || !campagna.posizione || campagna.posizione.length === 0) {return null;}
    const numPoints = campagna.posizione.length;
    let latSum = 0;
    let lonSum = 0;
    for (const point of campagna.posizione) {
      latSum += point.latitude;
      lonSum += point.longitude;
    }
    const latMedia = latSum / numPoints;
    const lonMedia = lonSum / numPoints;
    return { lat: latMedia, lon: lonMedia };
  };

  const ottieniLuogoMedio = async () => {
    const puntoMedio = calcolaPuntoMedio();
    if (puntoMedio) {
      try {
        const result = await geocoder.reverse({ lat: puntoMedio.lat, lon: puntoMedio.lon }, { language: "it" });
        const indirizzo = result.display_name || "N/D";
        setLuogoMedio(indirizzo);
      } catch (error) {console.error("Errore nella conversione delle coordinate:", error);
                       setLuogoMedio("Non disponibile");}
    } else {console.log("Punto medio non valido");
            setLuogoMedio("Non disponibile");}
  };
  if (campagna && !luogoMedio) {ottieniLuogoMedio();}
}, [campagna, luogoMedio]);
```

Figura 3.10 Metodi calcolaPuntoMedio e ottieniLuogoMedio

Posti all'interno di uno *useEffect*, viene prima creato un oggetto *geocoder* di tipo *NominatimGeocoder*, utilizzato per la geocodifica inversa, cioè per convertire le coordinate geografiche (latitudine e longitudine) in un indirizzo comprensibile.

La funzione *calcolaPuntoMedio* calcola il punto medio delle coordinate di una campagna. Prima verifica se esiste una campagna e se contiene coordinate (nel campo campagna.posizione). Se non ci sono coordinate valide, restituisce *null*, altrimenti, somma le latitudini e le longitudini di tutte le coordinate e calcola le medie.

La funzione *ottieniLuogoMedio* è asincrona e utilizza il risultato della funzione *calcolaPuntoMedio* per ottenere l'indirizzo leggibile del luogo medio delle coordinate.

Utilizza l'oggetto *geocoder* per fare una richiesta di geocodifica inversa, fornendo le coordinate del punto medio. Se la richiesta ha successo, l'indirizzo risultante viene impostato utilizzando *setLuogoMedio,* impostato inizialmente come stato per contenere il nome del luogo medio. Il metodo si conclude con un controllo condizionale che verifica se esiste una campagna e se *luogoMedio* non è già definito. Se entrambe le condizioni sono vere, viene chiamata la funzione *ottieniLuogoMedio*. Questo controllo si assicura che l'elaborazione venga eseguita solo quando una campagna è disponibile e quando il luogo medio non è ancora stato calcolato.

- Nomi operatori e nomi partecipanti: è stato necessario manipolare i dati inerenti agli utenti e agli operatori che partecipano alla campagna in quanto venivano mostrati gli ID di questi ultimi. Pertanto, tale processo raffigurato nella figura 3.11, è stato suddiviso nelle seguenti fasi (descritte solo per gli operatori, per i partecipanti è analogo):

```
useEffect(() => {
  // Ottieni i nomi degli operatori
  if (campagna && campagna.operatori && campagna.operatori.length > 0) {
    const operatoriPromises = campagna.operatori.map((operatoreId) => UtenteController.getNomeUtenteById(operatoreId));
    Promise.all(operatoriPromises).then((nomiOperatori) => {setNomiOperatori(nomiOperatori);});}
  // Ottieni i nomi dei partecipanti
  if (campagna && campagna.partecipanti && campagna.partecipanti.length > 0) {
    const partecipantiPromises = campagna.partecipanti.map((partecipanteId) =>UtenteController.getNomeUtenteById(partecipanteId));
    Promise.all(partecipantiPromises).then((nomiPartecipanti) => {setNomiPartecipanti(nomiPartecipanti);});}
}, [campagna]);
```

Figura 3.11 Processo per ottenere i nomi degli operatori e degli utenti

Viene verificata l'esistenza della campagna e se ha operatori associati. Questo controllo è essenziale, per evitare richieste effimere al database.

Se la condizione sopra è soddisfatta, viene creato un elenco di promesse. Ciascuna di esse corrisponde all'ottenimento del nome di un operatore specifico. Si utilizza un metodo di *getNomeUtenteById(operatoreId)* di UtenteController, che funziona in modo analogo al metodo visto nel punto precedente, per effettuare queste richieste, consentendo di ottenere i nomi in modo asincrono. Viene, poi, sfruttato *Promise.all(operatoriPromises)* per attendere il completamento di tutte le richieste. Una volta che tutte le promesse sono state risolte con successo, si ottiene un elenco completo dei nomi degli operatori, che verranno poi memorizzati nello stato *setNomiOperatori* inizializzato in precedenza per renderizzare o elaborare i nomi degli operatori nell'interfaccia utente.

Recuperate le informazioni necessarie inerenti alla campagna, è importante vedere come funziona la terminazione di questa. L'idea è stata quella di far effettuare un'operazione di scrittura sul database che cambia il campo "*inCorso*", presentato nella sezione 2.3.1, da true a false. Il metodo ideato, denominato *terminaCampagna,* si trova sempre nel *CampagneController* ed è il seguente:

```
//cambia il flag inCorso da true a false terminando difatto la campagna
static async terminaCampagna(campagna) {
  const campagnaRef = doc(db, "campagne", campagna.id);
  try {
    const docSnap = await getDoc(campagnaRef);
    if (docSnap.exists()) {
      await updateDoc(campagnaRef, { inCorso: false });
      console.log("Campagna terminata con successo");
    } else {console.log("Il documento campagna non esiste");}
  } catch (error) {console.error("Errore durante la terminazione della campagna:", error);
    throw error;}
}
```

Figura 3.12 Metodo terminaCampagna

Il funzionamento è simile ai metodi visti prima, con la differenza che qui viene effettuata un'operazione di aggiornamento del documento tramite la funzione *updateDoc*, di tipo *await* perché il metodo è inizializzato come asincrono.

Il metodo descritto sopra è richiamato nel componente *SchedaCampagna* tramite la funzione *handleTerminaCampagna* (sempre di tipo *async*), che a sua volta viene richiamata nella finestra di dialogo modale per confermare l'operazione.

3.1.5 Inserimento di una nuova campagna

Questa importante funzionalità è gestita principalmente dal componente *CampagnaForm*, in cui viene costruita la *form* di inserimento di una nuova campagna, con le rispettive regole di validazione dei campi. Questo componente pone le sue basi sul metodo *aggiungiNuovaCampagna(CampagnaData)* nel *CampagnaController*, illustrato nella figura 3.13, che accetta un argomento *CampagnaData* e lo aggiunge alla *collection* tramite la funzione *addDoc* di tipo *await.*

```
//aggiunge una nuova campagna alla collection
 static async aggiungiNuovaCampagna(campagnaData) {
   try {
     const campagneRef = collection(db, "campagne");
     const docRef = await addDoc(campagneRef, campagnaData);
     console.log("Nuova campagna aggiunta con successo!", docRef.id);
     return docRef; // Ritorna il riferimento al nuovo documento campagna
   } catch (error) {console.error("Errore durante l'aggiunta della campagna:", error);
     throw error;}
 }
```

Figura 3.13 Metodo aggiungiNuovaCampagna

L’argomento proviene proprio dal componente *CampagnaForm* dove corrisponde l’oggetto *nuovaCampagna,* creato dopo il completamento della form. I passi più importanti seguiti per la creazione del suddetto componente sono illustrati nel seguente elenco:

- Tramite l’hook *useState* vengono definiti gli stati dei campi che il nuovo documento dovrà avere, della finestra di dialogo e dei messaggi di errore. Inoltre, viene definito un array *tipologieCampagna* con le opzioni di tipologia di campagna disponibili;

Figura 3.14 Stati dei campi della form

- All'interno di un blocco *useEffect*, viene chiamato il metodo *getOperatori* di *UtenteController* per ottenere un elenco di operatori e popolare lo stato. Viene creato un array *options* per popolare un componente *Typeahead* con le opzioni degli operatori per selezionarne più di uno contemporaneamente. Sono presenti, poi, i metodi *handlePoligonoChange*, che gestisce i cambiamenti nelle coordinate del poligono selezionato sulla mappa, e *handleSubmit*, richiamato quando l'utente invia il *form*.

  All'interno del metodo *handleConfirm*, vengono effettuate diverse verifiche:

  1. Vengono controllati i campi obbligatori e se uno qualsiasi di questi campi è vuoto o non valido, viene aggiunto un messaggio di errore allo stato *errorMessages*. Sulla data viene inoltre controllato se quella di fine sia successiva a quella di inizio;

2. Se il form è valido, i dati vengono convertiti e una nuova campagna viene creata come oggetto. Viene, poi chiamato il metodo *aggiungiNuovaCampagna*, illustrato in precedenza, per passare come argomento la campagna creata;

```
if (Object.keys(errors).length === 0) {
  const dataInizioTimestamp = convertiInTimestamp(inizio);
  const dataFineTimestamp = convertiInTimestamp(fine);
  const dataOdiernaTimestamp = Math.floor(Date.now() / 1000); // Timestamp della data odierna

  const nuovaCampagna = {
    titolo,
    inizio: Timestamp.fromMillis(dataInizioTimestamp * 1000), // Converti la data inizio in timestamp Firebase
    fine: Timestamp.fromMillis(dataFineTimestamp * 1000), // Converti la data fine in timestamp Firebase
    operatori: selectedOperatori.map((operatore) => operatore.value),
    info,
    tipo,
    posizione: poligono.map(point => ({ latitude: point[0], longitude: point[1] })),
    partecipanti: [],
    segnalazioni: [],
    inCorso: dataInizioTimestamp <= dataOdiernaTimestamp, // Imposta il flag inCorso
  };
  try {
    const docRef = await CampagnaController.aggiungiNuovaCampagna(nuovaCampagna);
    console.log("Nuova campagna aggiunta con successo!", docRef.id);
```

Figura 3.15 Creazione oggetto nuovaCampagna

3. Se l'aggiunta della campagna ha successo, vengono azzerati i campi del *form* e viene chiamata la funzione *onCampagnaAvviata* passata come proprietà, richiamata nella view *NewCampagna* per mostrare all'amministratore il messaggio di successo.

```
const handleCampagnaAvviata = () => {
  setSuccessMessageVisible(true);
};
return (
  <div>
    <h3>Nuova Campagna</h3>
    {successMessageVisible && (
      <div className="alert alert-success mt-3">
        La nuova campagna è stata salvata con successo!
      </div>
    )}
    <CampagnaForm onCampagnaAvviata={handleCampagnaAvviata} />
  </div>
);
```

Figura 3.16 Passaggio proprietà onCampagnaAvviata

- Infine, è stata creata la grafica della *form* utilizzando varie componenti bootstrap. In questa fase è stata posta particolare attenzione a come tracciare l'area di svolgimento della campagna direttamente sulla mappa. Per fare ciò, è stato creato un altro componente denominato *ContenitoreMappaForm,* che accetta l'oggetto *onPoligonoChange*. Quest'ultimo viene passato come proprietà per essere richiamato all'interno del componente della *form* dedicato all'inserimento dell'area. Tale oggetto, infatti, rappresenta una funzione *callback* da chiamare quando un nuovo poligono viene creato o modificato. Per tenere traccia dei poligoni disegnati sulla mappa, viene definito uno stato per l'array *mapLayers,* utilizzando l'hook *useState*. Quando viene creato un nuovo poligono, invece, viene chiamata la funzione *handleCreate*, che estrae le informazioni, lo aggiunge a *mapLayers* e chiama la funzione *onPoligonoChange* per notificare il cambiamento.

```
const handleCreate = (e) => {
  const { layerType, layer } = e;
  if (layerType === "polygon") {
    const { _leaflet_id } = layer;
    const newPolygon = { id: _leaflet_id, latlngs: layer.getLatLngs()[0] };
    setMapLayers((prevLayers) => [...prevLayers, newPolygon]);

    const coordinates = newPolygon.latlngs.map((latlng) => [
      latlng.lat,
      latlng.lng,
    ]);
    onPoligonoChange(coordinates);
  }
};
```

Figura 3.17 Funzione handleCreate

Il disegno e la modifica del poligono avvengono tramite il componente *<FeatureGroup>*, che consente di gestire i poligoni disegnati e li rende modificabili. Esso avvolge a sua volta *<EditControl>* fornito da *react-leaflet-draw* che gestisce il disegno, la modifica e l'eliminazione. Successivamente, il codice mappa i poligoni esistenti da *mapLayers* a componenti *<Polygon>* che vengono visualizzati sulla mappa.

```
<FeatureGroup>
  <EditControl position="topright"
    onCreated={handleCreate}
    onEdited={handleEdit}
    onDeleted={handleDelete}
    draw={{
      rectangle: false,
      polyline: false,
      circle: false,
      circlemarker: false,
      marker: false,
    }}
  />
</FeatureGroup>

{mapLayers.map((polygon) => (
  <Polygon key={polygon.id}
    positions={polygon.latlngs}
    color="blue"
  />
))}
```

Figura 3.18 Componenti FeatureGroup, EditGroup e Polygon

Alla fine, viene renderizzata la mappa, i poligoni, il controllo di disegno e il *layer* delle mappe.

3.1.6 Approvazione ed eliminazione delle segnalazioni

La gestione delle segnalazioni avviene all'interno delle view *SegnalazioniApprovate*, *SegnalazioniDaApprovare* e *SchedaSegnalazione*. Prendendo in esame, ad esempio, la seconda vista, in cui si possono eseguire entrambi i processi, essa è costruita a livello grafico con una serie di card a dimensione fissa, contenenti informazioni come l'immagine, il tipo di rifiuto, il numero della segnalazione, la data il luogo e il nome del segnalatore, e i bottoni "*Approva*" ed "*Elimina*". Ogni card è gestita dal componente grafico *CardSegnalazione*. In questo contesto, oltre ad analizzare come avvengono approvazione ed eliminazione, è utile illustrare come è stata risolta la problematica di mostrare il bottone "Approva" solo nel caso in cui la segnalazione è da approvare. Per farlo, è stata passata al componente *CardSegnalazione* la proprietà *isDaApprovare*, utilizzata come condizione per la creazione del bottone, come mostrato in figura 3.19.

```
{isDaApprovare && (<> <button className="button-approva-segnalazione"
style={{ borderRadius: '5px', padding: '5px 10px', backgroundColor: 'rgba(46,139,87, 0.7)' }}
onClick={onApprova}>Approva</button></> )}
```

Figura 3.19 Proprietà isDaApprovare in cardSegnalazione

Se è *true* il bottone viene visualizzato e, nel caso della view *SegnalazioniDaApprovare*, questa proprietà è passata come tale quando viene richiamato il componente:

```
<CardSegnalazione key={segnalazione.id}
segnalazione={segnalazione} numero={filteredSegnalazioni.length - index - offset}
 isDaApprovare={true}
```

Figura 3.20 Passaggio proprietà isDaApprovare alla view

Analizzando in dettaglio il processo di approvazione nella vista in esame, questo viene attuato grazie alla funzione *handleApprovaSegnalazione* che richiama, a sua volta, il metodo *ApprovaSegnalazione* definito nel controller delle segnalazioni, che prende come argomento un oggetto segnalazione.

```
//funzione che permette il cambio del flag validazione da in corso a si
 static async approvaSegnalazione(segnalazione) {
  const segnalazioneRef = doc(db, "segnalazioni", segnalazione.id);
  try {
    const docSnap = await getDoc(segnalazioneRef);
    if (docSnap.exists()) {
      await updateDoc(segnalazioneRef, { validazione: "si" });
      console.log("Segnalazione approvata con successo");
    } else {console.log("Il documento segnalazione non esiste");}
  } catch (error) {console.error(error);}
}
```

Figura 3.21 Funzione approvaSegnalazione

Come si evince dalla figura 3.21, viene eseguita un'operazione di scrittura sul documento specifico della segnalazione da approvare che cambia il campo "validazione" da "in corso" a "si" tramite la funzione *updateDoc*. Il tutto viene svolto all'interno di un blocco *try-catch* per intercettare eventuali errori, e viene prima verificato se il documento esiste per evitare richieste effimere al database.

Il processo di eliminazione ha la stessa logica di quello appena descritto, con la differenza che nel metodo che la gestisce all'interno del controller viene cambiato il campo "validazione" da "in corso" a "no". Inoltre, bisogna sottolineare che l'eliminazione può avvenire in qualsiasi vista inerente alle segnalazioni all'interno dell'app.

3.1.7 Gestione Operatori e Utenti mobile

La logica che gestisce gli operatori e gli utenti è la stessa vista per le segnalazioni, con la differenza che qui vengono eseguiti i processi di declassamento dell'operatore e promozione dell'utente. L'implementazione è molto simile a quella vista nella sezione precedente, ma è importante evidenziare che sono gestiti dallo stesso controller, denominato *UtenteController*. Questo perché, come descritto nella sezione 2.3.1, si trovano all'interno della stessa *collection*.

È interessante, però, analizzare un tipo di informazione mostrata all'interno del componente *CardOperatore*, ottenuta da una manipolazione dei dati all'interno di *UtenteController*: lo stato. Tale dato indica, semplicemente, se un operatore è impegnato in una campagna oppure è disponibile. Per fare ciò, il componente *CardOperatore* è stato prima dotato di una classe CSS personalizzata in un elemento basato sul valore della proprietà *isDisponibile* dell'oggetto operatore, accettato, a sua volta, dal componente stesso come *prop*. Se *isDisponibile* è *true*, la classe sarà 'disponibile', altrimenti sarà 'non-disponibile'. Questo permette di applicare stili CSS diversi in base allo stato di disponibilità dell'operatore all'interno della "card" che rappresenta l'operatore.

```
const CardOperatore = ({ operatore, onDeclassa }) => {
  const handleDeclassaClick = () => { onDeclassa(operatore);};
    return (
      <div className={`card ${operatore.isDisponibile ? 'disponibile' : 'non-disponibile'}`}>
        <div className="card-body">
          <div id="logo" className="image">
            <img src={operatoreImg} alt={`Immagine non disponibile`} style={{ width: "70%", marginRight: "5px", borderRadius: "5px" }}/>
          </div>
          <h5 className="card-title">{operatore.name}</h5>
          <p className="card-text">Data di nascita: {operatore.dob}</p>
          <p className="card-text">e-mail: {operatore.email}</p>
          <p className="card-text">Stato: {operatore.isDisponibile ? "disponibile" : "occupato"}</p>
```

Figura 3.22 Proprietà isDisponibile dell'operatore

Lo stato è recuperato all'interno del metodo statico *getOperatori*, che accetta una funzione di *callback* come argomento. Tale *callback* sarà chiamata una volta che i dati degli operatori sono stati recuperati e processati. Questa funzione può essere divisa sostanzialmente in tre parti:

1. Recupero dei dati degli operatori: si definisce un riferimento alla *collection "users"* all'interno del *Cloud Firestore*. Quindi, si costruisce una query per ottenere solo gli utenti che sono contrassegnati come operatori (*isOperator* è *true*). Successivamente, viene chiamato *onSnapshot* su questa query per ascoltare eventuali modifiche nella raccolta "utenti". Questo è il momento in cui inizia il recupero dei dati degli operatori, e una volta concluso, si esegue una mappatura su ciascun documento all'interno dello *snapshot* restituito dalla query. Per ciascun operatore, viene creato un oggetto *Utente* basato sui dati del documento e lo aggiunge a un array chiamato operatori;

2. Recupero e calcolo delle campagne attive: successivamente, vengono recuperate le campagne attualmente in corso. Dopo aver ottenuto i dati, questi vengono elaborati. Per ogni campagna attiva, vengono raccolti gli operatori associati a quella campagna all'interno dell'array *campagneAttive*.

3. Determinazione della disponibilità degli operatori: si itera attraverso l'array operatori e si determina se ciascun operatore è disponibile, cioè se non è coinvolto in alcuna delle campagne attive, o meno. Questo stato di disponibilità viene aggiunto all'oggetto dell'operatore come *isDisponibile*. Alla fine, viene chiamata la *callback* fornita come argomento e le passa l'array degli operatori con i loro stati di disponibilità.

```
static getOperatori(callback) {
  const utentiRef = collection(db, "users");
  const q = query(utentiRef, where("isOperator", "==", true));
  const operatoriSnapshotUnsubscribe = onSnapshot(q, (snapshot) => {
    const operatori = snapshot.docs.map((doc) => {
      const data = doc.data();
      const operatore = new Utente(doc.id, data);
      return operatore;
    });
    //qui imposto lo stato
    const campagneRef = collection(db, "campagne");
    const campagneQuery = query(campagneRef, where("inCorso", "==", true));
    getDocs(campagneQuery).then((campagneSnapshot) => {
        const campagneAttive = campagneSnapshot.docs.reduce((attive, doc) => {
          const campagnaData = doc.data();
          const operatoriCampagna = campagnaData.operatori || [];
          return [...attive, ...operatoriCampagna];
        }, []);
        operatori.forEach((operatore) => {
          const isDisponibile = !campagneAttive.includes(operatore.id);
          operatore.isDisponibile = isDisponibile;
        });
        callback(operatori);
      }).catch((error) => {console.log("Errore durante il recupero delle campagne attive:", error);});
  });
  return operatoriSnapshotUnsubscribe;
}
```

Figura 3.23 Funzione getOperatori

Inoltre, nel metodo vengono gestiti eventuali errori durante il recupero dei dati tramite *blocchi try-catch*.

Infine, viene restituita una funzione di annullamento (*operatoriSnapshotUnsubscribe*) che può essere utilizzata per interrompere l'ascolto delle modifiche nella raccolta di utenti.

3.1.8 Visualizzazione statistiche

Per le statistiche si è optato per la creazione di due viste, *StatsRifiuti* e *StatsCampagne*, gestite da un unico controller denominato *Statistiche*Controller. A livello grafico, risultano implementati con gli stessi componenti e mostrano gli stessi tipi di dato. Pertanto, in questa sezione, viene illustrato il processo solo per la visualizzazione delle statistiche di una campagna, calcolate solo sulle segnalazioni approvate (non avrebbe senso farle su quelle ancora in fase di accettazione).

Partendo dalla vista *StatsCampagne*, qui vengono importati i componenti grafici Chart dalla libreria *Chart.js*, per la creazione del grafico interattivo, e *Typehead* da *react-bootstrap*, per creare il menù

di selezione della campagna. Vengono, poi, inizializzati gli stati per memorizzare le campagne disponibili (*campagne*), la campagna selezionata (*selectedCampagna*), la stringa di ricerca (*searchTerm*) e i dati delle statistiche della campagna (*statistiche*).

In seguito, si passa alla funzione *renderChart* per l'inizializzazione del grafico, che prende in input l'oggetto *rifiutiPerTipo*, che contiene i dati relativi ai diversi tipi di rifiuti e le rispettive quantità. Vengono, creati gli oggetti: *chartData*, che rappresenta i dati del grafico, *chartOptions*, che definisce le opzioni di configurazione del grafico, e *chartElements*, in cui è memorizzato l'elemento del canvas HTML con l'ID "*chart*". Si effettua una verifica per vedere se esiste già un grafico sul canvas. Se esiste un grafico precedente, viene distrutto. Successivamente, viene creato un nuovo grafico di tipo "*bar*" (a barre), passando come parametri *chartData* e *chartOptions*, e disegnato sul canvas con l'ID "*chart*". I passaggi descritti sono raffigurati nella figura 3.24:

```
const renderChart = useCallback((rifiutiPerTipo) => {
  // Crea un oggetto per i dati del grafico
  const chartData = {
    labels: Object.keys(rifiutiPerTipo),
    datasets: [{ label: 'Quantità di rifiuti', // Etichetta del dataset
                 data: Object.values(rifiutiPerTipo), // Valori del dataset
                 backgroundColor: 'rgba(75, 192, 192, 0.2)', // Colore di sfondo delle barre
                 borderColor: 'rgba(75, 192, 192, 1)', // Colore del bordo delle barre
                 borderWidth: 1 // Spessore del bordo delle barre
               },],
  };
  // Opzioni del grafico
  const chartOptions = {responsive: true, maintainAspectRatio: false, aspectRatio: 2, scales: { y: {beginAtZero: true}}};
  const chartElement = document.getElementById('chart');
  if (chartElement) {
    // Controlla se esiste già un grafico sul canvas
    const existingChart = Chart.getChart(chartElement);
    if (existingChart) {existingChart.destroy();} // Distruggi il grafico esistente
    // Crea un nuovo grafico utilizzando Chart.js
    new Chart(chartElement, {type: 'bar', data: chartData, options: chartOptions});
  }
}, []);
```

Figura 3.24 Metodo renderChart

Successivamente sono utilizzati tre *useEffect* rispettivamente per: recuperare e gestire le campagne recuperate dal *CampagnaController* tramite *handleCampagne*, impostare le statistiche quando una campagna è selezionata, richiamando il metodo *generaStatisticheCampagna* da *StatisticheController*, e chiamare *renderChart* quando ci sono i dati validi inerenti alle statistiche e alla campagna selezionata.

Dopo queste operazioni preliminari viene montato il componente a livello grafico: la pagina mostra una lista di campagne all'utente, consentendo la selezione della campagna tramite un componente *Typeahead*. Se è stata selezionata una campagna e ci sono statistiche valide, viene visualizzato un riepilogo delle statistiche relative a quella campagna. Questo include il numero di segnalazioni totali, una tabella che elenca i tipi di rifiuti e le rispettive quantità e il grafico a barre che

rappresenta le quantità di rifiuti per tipo. Se non è stata selezionata alcuna campagna, viene visualizzato un messaggio che indica all'utente di selezionare una campagna per visualizzare le statistiche.

```
{statistiche && selectedCampagna && selectedCampagna.titolo ?  (
<div>
  <div className="rounded bg-white shadow mx-5 p-5 my-3"
  style={{  display: 'flex', flexDirection: 'column', alignItems: 'center' }}>
<h3>Statistiche per la campagna: {selectedCampagna.titolo}</h3>
<p>Numero di segnalazioni totali: {statistiche.numeroSegnalazioniTotali}</p>
<p>Rifiuti per tipo:</p>
<table style={{ border: '1px solid black', borderCollapse: 'collapse', margin: '20px', width: '400px' }}>
    <thead>
      <tr>
        <th style={{ border: '1px solid black', padding: '8px' }}>Tipo</th>
        <th style={{ border: '1px solid black', padding: '8px' }}>Quantità</th>
      </tr>
    </thead>
    <tbody>
      {Object.entries(statistiche.rifiutiPerTipo).map(([tipo, quantita]) => (
        <tr key={tipo}>
          <td style={{ border: '1px solid black', padding: '8px' }}>{tipo}</td>
          <td style={{ border: '1px solid black', padding: '8px' }}>{quantita}</td>
        </tr>
      ))}
    </tbody>
  </table>
  <div style={{ width: '400px', height: '250px' }}>
  <canvas id="chart" style={{ width: '100%', height: '100%' }}></canvas>
</div>
</div>
</div>
) : ( <p>Seleziona una campagna per visualizzare le statistiche.</p>)}
```

Figura 3.25 Codice per la grafica di StatsCampagne

Come accennato prima, le statistiche vengono generate dal metodo *generaStatisticheCampagna*, che prende come argomento l'ID della campagna selezionata. Vengono prelevate tutte le segnalazioni approvate nella campagna selezionata tramite una Promise e richiamando il metodo *getSegnalazioniApprovate* da *SegnalazioniController*. La *callback* passata a *getSegnalazioniApprovate* è utilizzata per filtrare le segnalazioni relative alla campagna in base all'ID *campagnaId*.

Successivamente si inizializzano le statistiche della campagna in un oggetto *statisticheCampagna*, che ha due chiavi principali:

- *numeroSegnalazioniTotali*: inizializzato con la lunghezza delle segnalazioni approvate nella campagna, rappresentando il numero totale di segnalazioni;
- *rifiutiPerTipo*: un oggetto vuoto che verrà popolato con il conteggio dei rifiuti per ogni tipo.

In seguito, si itera con un *forEach* su ciascuna segnalazione nella campagna e si controlla il tipo di rifiuto, incrementando il conteggio nel relativo campo nell'oggetto *rifiutiPerTipo*. Terminato il calcolo, viene restituito l'oggetto *statisticheCampagna* contenente il numero totale di segnalazioni e il conteggio dei rifiuti per tipo. I passaggi descritti sono rappresentati nella figura 3.26:

```
static async generaStatisticheCampagna(campagnaId) {
  try {
    // preleva tutte le segnalazioni approvate nella campagna selezionata
    const segnalazioniApprovateCampagna = await new Promise((resolve) => {
      SegnalazioniController.getSegnalazioniApprovate((segnalazioni) => {
        const segnalazioniCampagna = segnalazioni.filter((segnalazione) => {return segnalazione.idCampagna === campagnaId;});
        resolve(segnalazioniCampagna);
      });
    });
    // Calcola le statistiche sulla campagna
    const statisticheCampagna = {numeroSegnalazioniTotali: segnalazioniApprovateCampagna.length,rifiutiPerTipo: {}};
    // Calcola le statistiche sui rifiuti per tipo per la campagna
    segnalazioniApprovateCampagna.forEach((segnalazione) => {
      if (!statisticheCampagna.rifiutiPerTipo[segnalazione.tipo]) {statisticheCampagna.rifiutiPerTipo[segnalazione.tipo] = 1;}
      else {statisticheCampagna.rifiutiPerTipo[segnalazione.tipo] += 1;}
    });
    // Restituisci le statistiche calcolate per la campagna
    return statisticheCampagna;
  } catch (error) {console.error('Errore durante il calcolo delle statistiche per la campagna:', error);
    throw error;}
}
```

Figura 3.26 Metodo generaStatisticheCampagna

3.2 Screenshot della web app

In questa sezione verranno riportate le schermate presenti nella web app. Per ognuna di esse saranno descritte le funzionalità che offrono e le azioni che l'utente può fare.

3.2.1 Login

Figura 3.27 Schermata Login

All'avvio della web app, l'utente visualizza la schermata di login in cui deve inserire, nei rispettivi campi, le credenziali richieste per l'accesso.

3.2.2 Panoramica attività e menù

Una volta effettuato il processo di autenticazione, viene mostrata all'utente la pagina della panoramica delle attività ancora in svolgimento su territorio. La schermata contiene una mappa con due tipi di marker: la bandierina rappresenta una campagna, alla quale è annessa la sua area di svolgimento indicata dal poligono, mentre il sacchetto dei rifiuti rappresenta una segnalazione. Cliccando su uno dei marker, è possibile visualizzare un pop-up informativo contenente alcune informazioni e un link che porta l'utente alla scheda di una campagna o di una segnalazione.

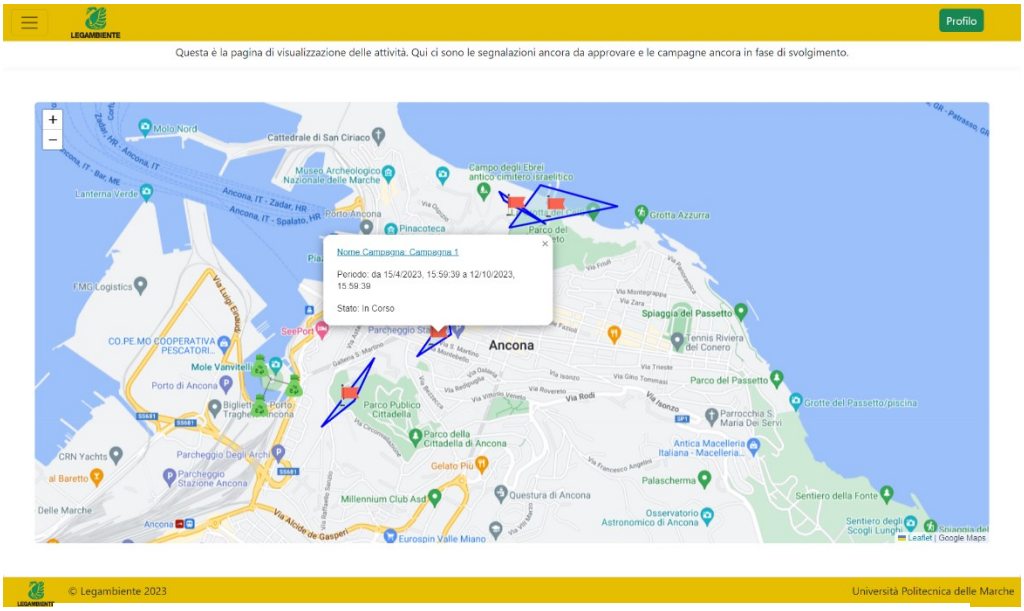

Figura 3.28 Schermata visualizzazione mappa

L'utente può spostarsi tra le pagine grazie ad un menù laterale a scorrimento da sinistra, contenente le varie sezioni mostrate nella figura 3.29:

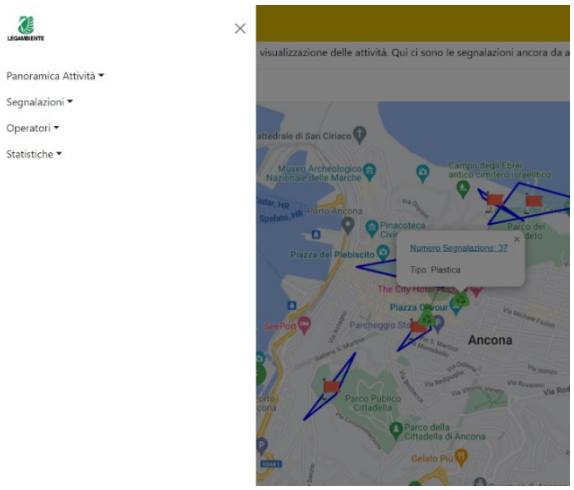

Figura 3.29 Menù

Tale menù è organizzato in sezioni, rappresentate da dei menù a tendina con le seguenti opzioni di navigazione:

- nella parte "Panoramica Attività", l'utente può spostarsi tra le schermate di visualizzazione della mappa, inserimento di una nuova campagna e la lista delle campagne;
- nella parte "Segnalazioni", l'utente può spostarsi tra le due liste di segnalazioni approvate e da approvare;

- nella parte “Operatori”, l’utente può spostarsi tra le due liste di operatori, per visualizzarli e declassarli, e di utenti, per visualizzarli e promuoverli;
- nella parte “Statistiche”, l’utente può spostarsi tra la schermata di visualizzazione delle analisi sui rifiuti segnalati in generale e quella di una specifica campagna.

È importante evidenziare che questo menù è raggiungibile da qualsiasi parte dell’applicazione.

3.2.3 Inserimento nuova campagna

In questa pagina è presente la *form* che l’utente deve compilare per inserire una nuova campagna. Sono obbligatori tutti i campi, incluso il tracciamento dell’area di svolgimento sulla mappa.

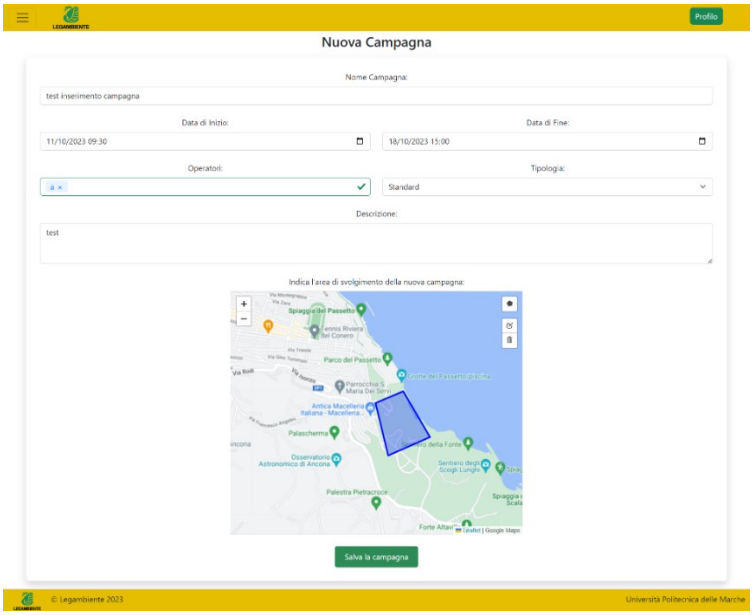

Figura 3.30 Form inserimento nuova campagna

Cliccando sul bottone “Salva la campagna”, verrà mostrato il pop-up di conferma dell’operazione.

3.2.4 Lista campagne e scheda campagna

Nella schermata “Lista campagne” è possibile visualizzare tutte le campagne, ricercarle, terminarle ed eliminarle. Per quanto riguarda queste due ultime azioni, sono mostrati, in entrambi i casi, dei pop-up di conferma.

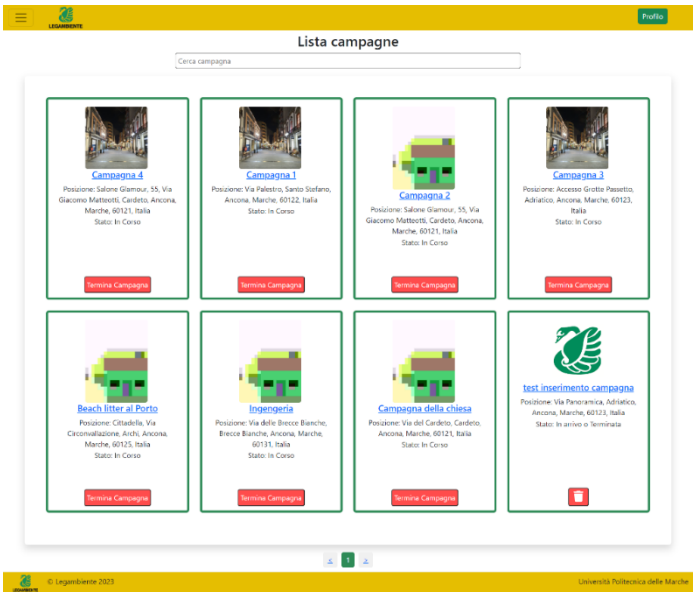

Figura 3.31 Schermata lista campagne

Cliccando su un qualsiasi card, è possibile raggiungere la scheda della campagna scelta, nella quale saranno visualizzate tutte le informazioni necessarie, incluso l’elenco dei partecipanti tramite un pop-up apribile al click del bottone “Visualizza”. Le segnalazioni sono poste al di sotto della scheda e possono essere filtrate tra quelle approvate e quelle da approvare.

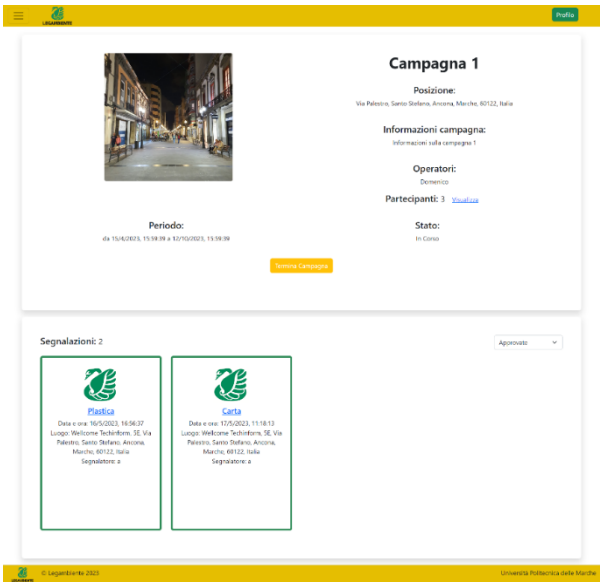

Figura 3.32 Schermata scheda campagna

3.2.5 Segnalazioni approvate, da approvare e scheda segnalazione

In “Segnalazioni approvate” l’utente vede tutte le segnalazioni approvate dagli operatori e può ricercarle, visualizzarne la scheda ed eliminarle, previa conferma dell’operazione tramite pop-up, nel caso in cui sia stata approvata erroneamente.

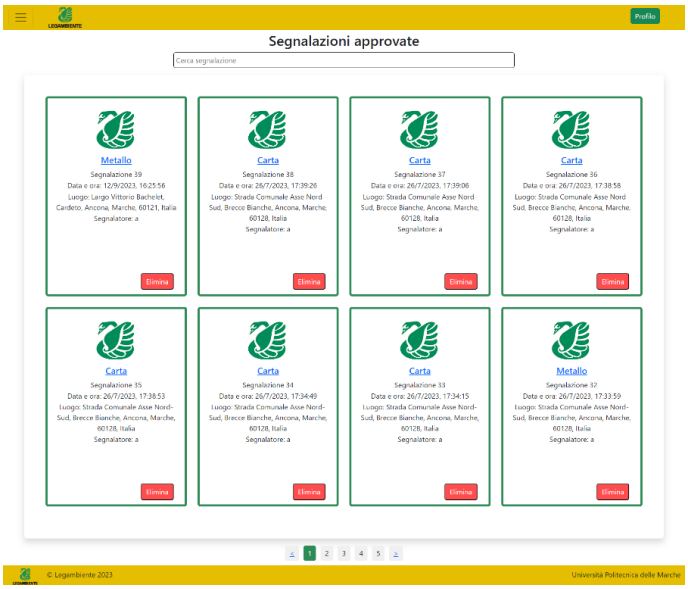

Figura 3.33 Schermata segnalazioni approvate

In “Segnalazioni da approvare” l’utente vede una lista uguale a quella precedentemente descritta, con la differenza che è presente un bottone “Approva” per approvare la segnalazione, previa conferma tramite pop-up.

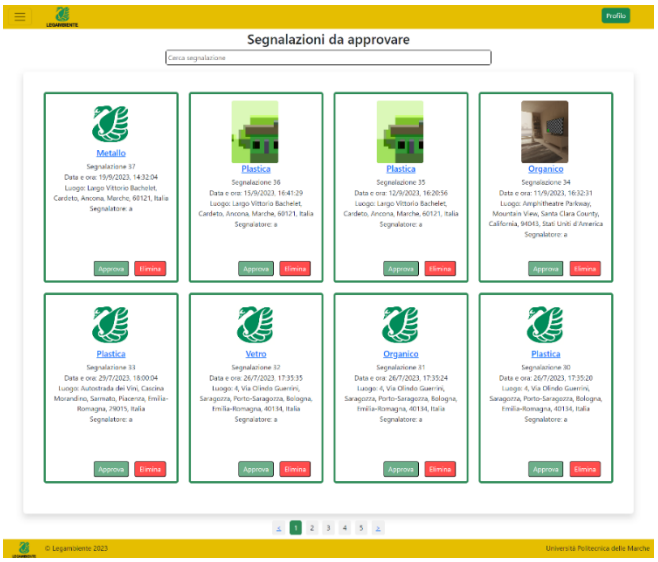

Figura 3.34 Schermata segnalazioni da approvare

Nella scheda di una segnalazione sono presenti tutte le informazioni e sono eseguibili tutte le operazioni viste in precedenza. Da notare che, se tale segnalazione è stata effettuata in una campagna, sarà presente un link di riferimento ad essa.

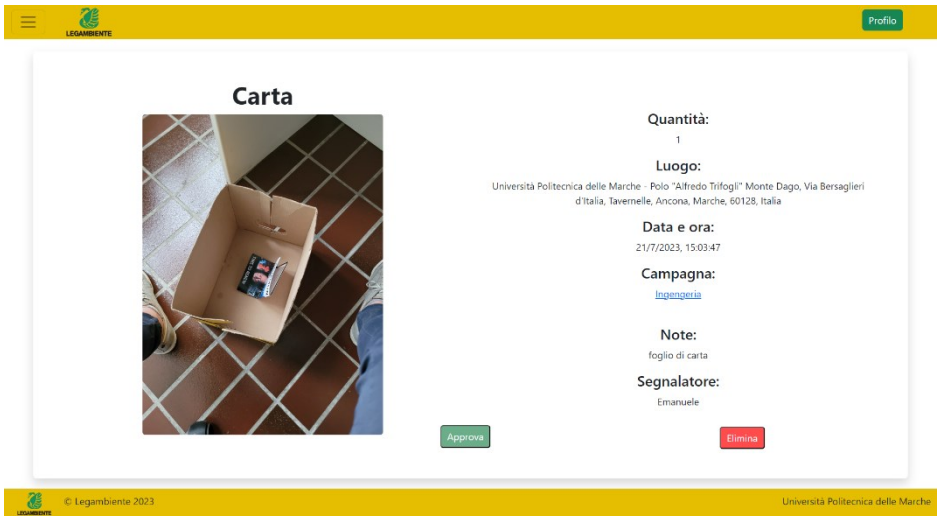

Figura 3.35 Schermata scheda segnalazione

3.2.6 Lista Operatori e Lista Utenti

In "Lista Operatori" è possibile visualizzare tutti gli operatori presenti. Cliccando sul pulsante "Declassa" è possibile regredire l'operatore scelto ad utente mobile base, previa conferma tramite pop-up.

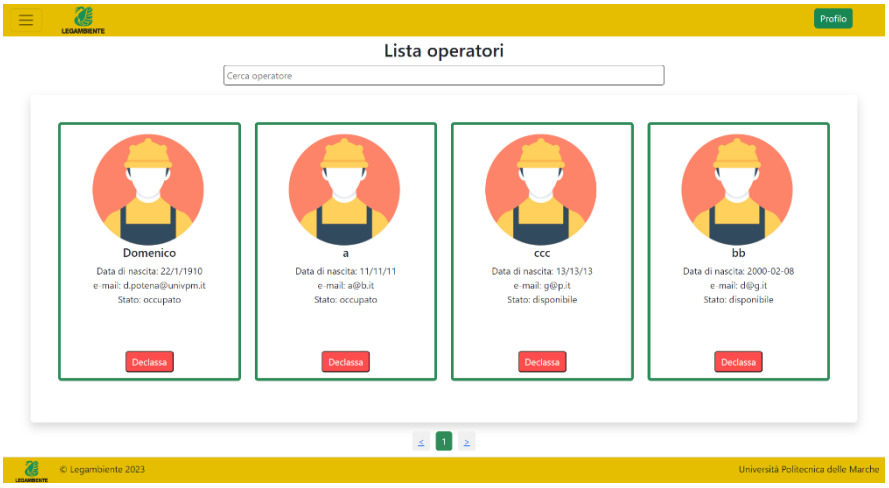

Figura 3.36 Schermata lista operatori

Nella "Lista Utenti" è possibile visualizzare tutte le informazioni degli utenti che si sono iscritti all'app mobile. È possibile promuoverli ad utenti se, ad esempio, sono molto attivi sull'applicazione,

cliccando il bottone “Promuovi”, oppure eliminarli definitivamente in caso di violazione delle regole

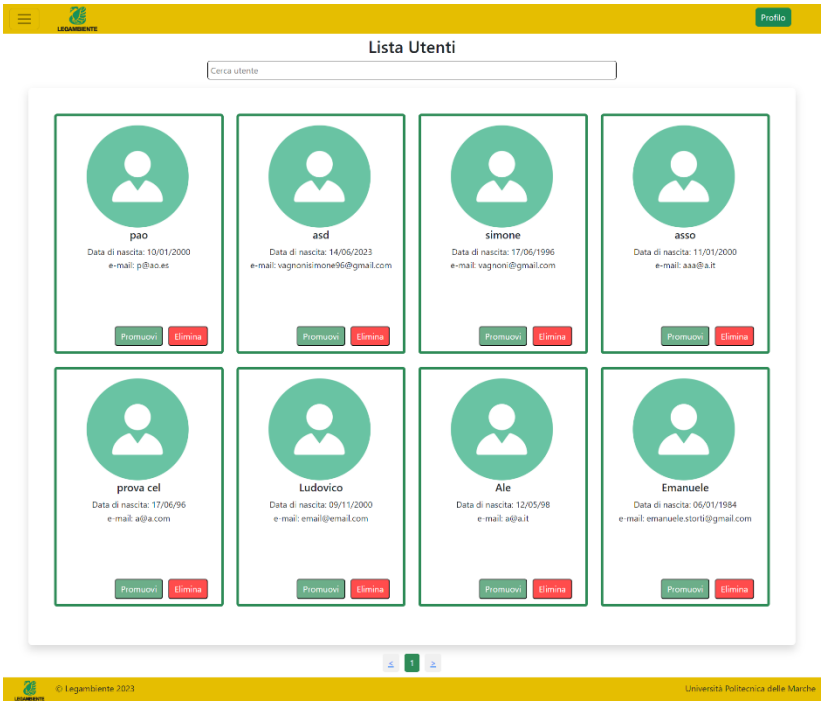

Figura 3.37 Schermata lista utenti

al click del pulsante “Elimina”. Entrambe le operazioni devono essere confermate.

Si noti che, se si vuole eliminare un operatore, bisogna prima declassarlo ad utente e poi eliminarlo.

3.2.7 Statistiche

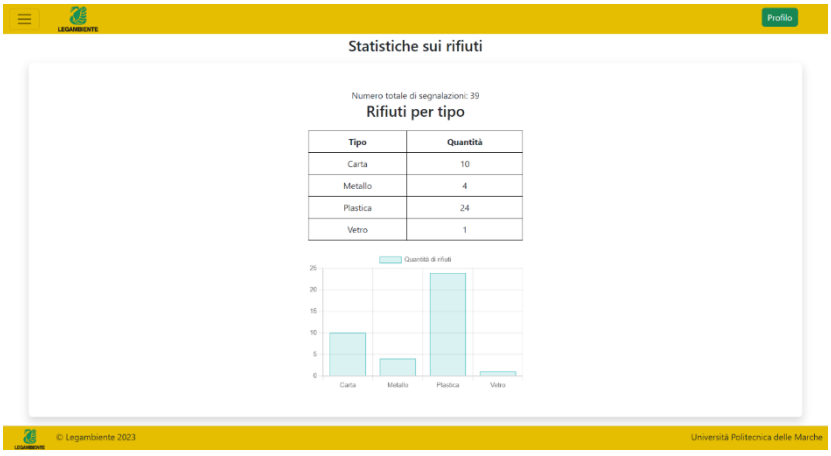

Figura 3.38 Schermata statistiche rifiuti segnalati

Per visualizzare, invece, le statistiche inerenti ad una campagna, questa va prima selezionata:

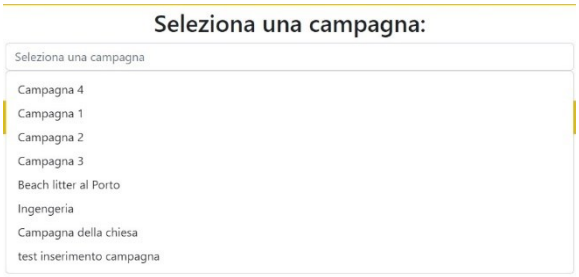

Figura 3.39 Menù di selezione campagna

Selezionando, ad esempio, “Ingegneria” si ottiene la seguente schermata:

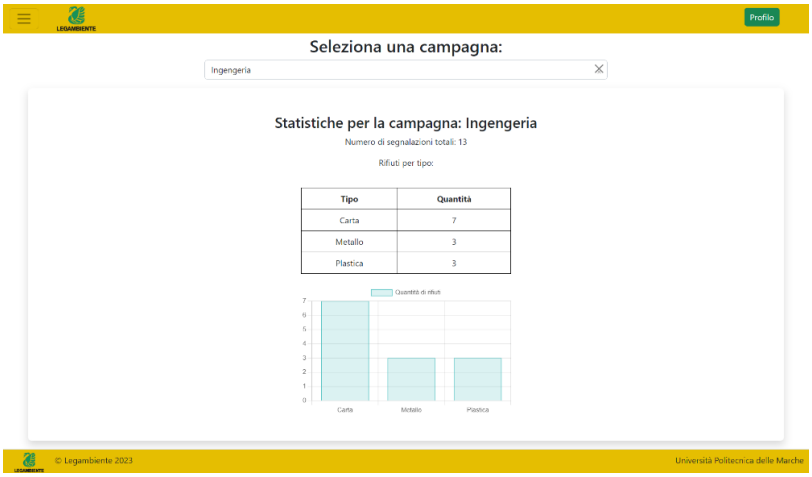

Figura 3.40 Schermata statistiche per una campagna

3.2.8 Profilo e Logout

Questa schermata contiene l’e-mail dell’utente e il bottone “Logout” che, al click, effettua la disconnessione.

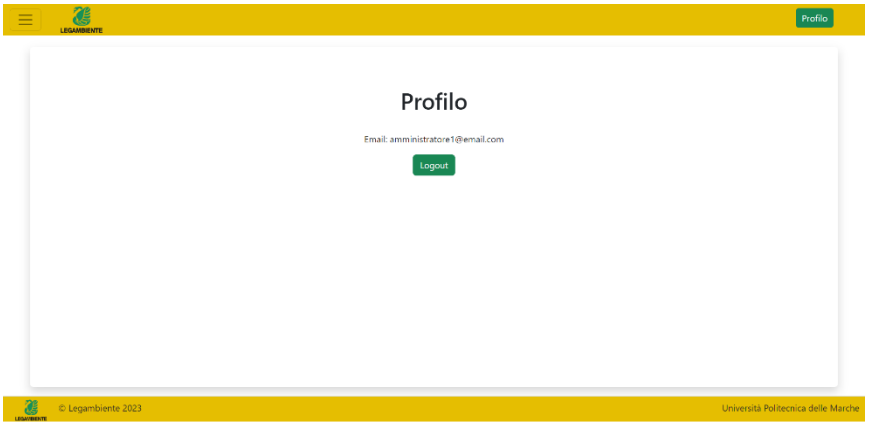

Figura 3.41 Schermata profilo e logout

Conclusioni

In questa tesi sono stati analizzate le fasi che hanno portato alla creazione di una web app gestionale per generare campagne partecipative di monitoraggio ambientale.

Nel primo capitolo, nello specifico, è stata presentata una descrizione dell'applicazione mobile con la quale lavora ed è stata effettuata un'analisi approfondita dei requisiti funzionali e non funzionali.

Nel secondo capitolo si è passati, invece, alla fase di progettazione, in cui sono stati studiati i vari casi d'uso, elencati i principali strumenti utilizzati ed è stata illustrata la struttura scelta per il database e l'architettura del progetto scelta per soddisfare i requisiti.

Infine, nel terzo ed ultimo capitolo è stata spiegata la fase di implementazione e sviluppo, nella quale è stata messa particolare attenzione su come le funzionalità più importanti sono state implementate.

Nel complesso lo sviluppo di questa web app può ritenersi completato, in quanto sono stati soddisfatti i requisiti analizzati nel primo capitolo. È importante, però, sottolineare che il lavoro svolto non si esaurisce con questa tesi, bensì ci sono diverse direzioni da essere esplorate e diversi miglioramenti e sviluppi futuri quali:

- Migliorare interfaccia grafica: nello sviluppo di questa web app è stato dato poco peso alla grafica, pertanto sarebbe utile, ai fini dell'esperienza dell'utente, rendere la grafica più accattivante;
- Ottimizzare la comunicazione con il database: analizzando il numero di richieste di lettura in arrivo al Cloud Firestore sulla console di Firebase, si denota che il suo valore è alto, nonostante siano stati inseriti all'interno del codice numerosi controlli. Pertanto, sarebbe opportuno ottimizzare in maniera ancora più efficiente questo aspetto;
- Aggiungere la possibilità di modificare una campagna già salvata: per garantire lo svolgimento di tutte le operazioni di CRUD all'interno dell'app, è necessario aggiungere la possibilità di modificare una campagna. Un'idea potrebbe essere quella di aggiungere un pulsante di modifica all'interno della scheda che, al suo click, porta l'utente alla form già compilata;

- Aggiungere degli avvertimenti durante il processo di declassamento di un operatore o impedirlo se questo è l'unico ad essere presente in una determinata campagna;
- Aggiornare la lista degli operatori presenti in una campagna se uno di questi viene declassato ad utente;
- Aggiungere all'interno della pagina web "Profilo" la possibilità di modificare le credenziali;
- Aggiungere la possibilità di recuperare una segnalazione eliminata erroneamente: un suggerimento può essere quello di creare una lista denominata "Segnalazioni Eliminate" e inserire in ogni card un bottone "Recupera", facendo tornare il campo "validazione" da "no" a "in corso".

Con questi miglioramenti, la web app risulterà sicuramente più completa e accessibile.

Inoltre, a livello personale, questo progetto è risultato molto istruttivo e funzionale, soprattutto per quanto riguarda lo studio di *ReactJS*. Grazie a questa tecnologia, risultata di facile apprendimento, lo sviluppo della web app è risultato semplice e la risoluzione dei problemi è stata abbastanza veloce ed intuitiva. Questo perché è largamente utilizzata in ambito web e gode di una documentazione precisa e dettagliata. Averla trattata ed incontrata, dunque, rappresenta una crescita personale piuttosto significativa.

Bibliografia

[1] Visual Studio Code: https://code.visualstudio.com/docs.

[2] HTML: https://it.wikipedia.org/wiki/HTML.

[3] CSS: https://www.html.it/guide/guida-css-di-base/.

[4] Javascript: https://www.javascript.com/learn.

[5] Firebase: https://firebase.google.com/.

[6] ReactJS: https://it.legacy.reactjs.org/docs/.

[7] OpenStreetMap: https://osmit.it/.

[8] Leaflet: https://leafletjs.com/.

[9] Bootstrap: https://getbootstrap.com/docs/.

[10] Chart.js: https://www.chartjs.org/docs/.

[11] Design pattern Model-View-Controller: https://www.html.it/pag/18299/il-pattern-mvc/.