Università degli Studi di Ferrara

UNIVERSITÀ DEGLI STUDI DI FERRARA

Corso di Laurea in Informatica

Progettazione e sviluppo di un'applicazione mobile per monitorare e ripristinare dispositivi edge

Relatore:
Prof. Carlo Giannelli

Laureando:
Matteo Macrì

Correlatore:
Dott. Ing. Tommaso Berlose

Anno Accademico 2022/2023

Indice

Introduzione

Nell'epoca dell'Industria 4.0, siamo testimoni di una rivoluzione senza precedenti nel mondo della produzione e dell'automatizzazione rendendo il sistema industriale più efficiente, sostenibile e remunerativo. Una delle chiavi di questa evoluzione è stata resa possibile dalla digitalizzazione e dalla diffusione delle applicazioni mobile che svolgono il compito di mantenere interconnesso l'intera rete industriale. La facilità di portare il cellulare in tasca può essere un buon vantaggio rispetto ad alcuni metodi utilizzati nell'industria.

All'interno di questo contesto le applicazioni mobile stanno fornendo mezzi essenziali per aumentare la produttività e le prestazioni delle fabbriche, le ragioni sono chiare: le app sono facili da usare, convenienti e disponibili ovunque in un formato user-friendly e consentono una comunicazione con l'IoT (Internet of Things) producendo un ambiente agile, produttivo e realmente collaborativo tra uomo e macchina. Tuttavia, anche se questa trasformazione porta numerosi vantaggi, va notato che molte aziende si imbattono in ostacoli e non possono permettersi di accedere all'Industria 4.0 per vari motivi: tra le sfide più impegnative vi è la difficoltà di sostituire le macchine esistenti con quelle di nuova generazione.

A tale scopo l'azienda Fancy Pixel ha dedicato risorse e competenze per sviluppare il FancyBox. Un dispositivo di edge computing che viene collocato in prossimità delle macchine industriali, progettato con l'obiettivo di integrare un sistema di monitoraggio e di controllo remoto delle macchine industriali permettendole di abilitarle all'Industria 4.0. Grazie al FancyBox, l'azienda punta a migliorare l'efficienza e la connettività nell'ambito industriale rendendo il sistema più intelligente e interconnesso.

Con la seguente tesi si propone di progettare e realizzare un'applicazione mobile, denominata FancyBox Companion App. Questa applicazione ha l'obiettivo di offrire agli utenti un efficace strumento per il monitoraggio delle prestazioni e di ripristinare le configurazioni di networking del FancyBox qualora l'utente non riesca più a comunicare con esso.

Nel Capitolo 1, saranno presentati i concetti correlati all'Industria 4.0, gli obiettivi di tesi seguiti dalla descrizione del FancyBox e concludendo con l'analisi dei requisiti del progetto. Nel Capitolo 2, verrà fornita una spiegazione di come funzionano i framework e le librerie utilizzate per lo sviluppo dell'applicazione mobile cross-platform. Infine, nel Capitolo 3, mette a luce l'implementazione del codice sviluppato per le diverse schermate che compongono l'applicazione.

Capitolo 1

Smart Machines

1.1 Industria 4.0

L'industria 4.0 sta rivoluzionando il modo in cui le aziende producono, migliorano e distribuiscono i loro prodotti. I produttori delle macchine stanno iniziando ad applicare nuove tecnologie su di esse come: sensori avanzati, software integrato e macchine collegate alla rete consentono di migliorare il processo decisionale, tramite la visibilità in tempo reale dei dati raccolti che, oltretutto, attraverso adeguate soluzioni potrebbero aiutare a ridurre al minimo i tempi di fermo macchina.

L'utilizzo di queste tecnologie è in grado di apportare importanti miglioramenti nei processi produttivi come, per esempio, l'automatizzazione industriale o l'inserimento di strumenti per la manutenzione predittiva. L'utilizzo di dispositivi IoT all'interno di una fabbrica può portare a una maggiore produttività e a una migliore qualità. Con un investimento minimo, il personale per il controllo qualità può monitorare i processi manifatturieri da qualsiasi luogo si trovi grazie a una piattaforma cloud che permette di raccogliere, archiviare e analizzare i dati provenienti dalla macchine e quindi di essere sempre accessibili in qualsiasi momento del bisogno. Tutti questi strumenti possono essere applicati a tutti i tipi di aziende, sono:

1. *Internet of Things (IoT)*: è una componente chiave nelle fabbriche intelligenti. Grazie alla meccanizzazione e la connettività è possibile raccogliere, analizzare e scambiare un'alta mole di dati. Questi possono essere recuperati tramite i sensori o altri dispositivi con i quali è possibile equipaggiare le macchine.

2. *Cloud Computing*: permette di ridurre i costi iniziali delle piccole e medie imprese, così da poter dimensionare correttamente le loro esigenze e adattarle alla crescita del business. Inoltre è possibile processare in modo più efficiente ed economicamente conveniente la quantità elevata di dati memorizzati e analizzati.

3. *AI e Machine Learning*: possono offrire la prevedibilità e l'automazione di operazioni e processi aziendali. Permettendo alle aziende di avvantaggiarsi del volume di informazioni generate non solo dai reparti della fabbrica, ma anche attraverso le divisioni aziendali e persino dai partner e da terze parti.

4. *Edge Computing*: richiedere operazioni produttive in tempo reale significa che alcune analisi dei dati debbano essere fatte da dispositivi il più possibili vicini alla macchina, cioè laddove i dati vengono creati. Questo minimizza il tempo di latenza tra il momento in cui i dati vengono prodotti e il momento in cui è richiesta una risposta. Il tempo impiegato per inviare i dati al cloud aziendale e poi di nuovo al reparto della fabbrica

può essere eccessivo e dipendere dall'affidabilità della rete. L'utilizzo dell'edge computing significa mantenere i dati vicino alla loro sorgente, riducendo allo stesso tempo i rischi per la sicurezza.

5. *Cybersecurity:* la connettività delle apparecchiature operative della fabbrica, che abilitano i processi produttivi, espongono anche percorsi di ingresso ad attacchi e malware dannosi. Nella trasformazione digitale verso l'Industria 4.0 è fondamentale considerare l'approccio alla sicurezza informatica che comprende tali apparecchiature.

6. *Digital Twin*: La trasformazione digitale offerta dall'Industria 4.0 ha permesso ai produttori di creare dei `gemelli digitali` che sono repliche virtuali di processi, linee produttive. Per creare un gemello digitale occorre estrarre i dati dai sensori IoT, dispositivi, PLC e altri oggetti connessi a Internet. I produttori possono usare questi gemelli per aumentare la produttività, migliorare il flusso di lavoro e progettare nuovi prodotti.

Le macchine che implementano queste tecnologie vengono dette Smart Machines. Una Smart Machine è una macchina più connessa, più flessibile, più efficiente, più sicura e in grado di adattarsi ai requisiti di Industria 4.0.

Una Smart machine è in grado di comunicare tra macchine (M2M), ma anche di comunicare tra macchine e persone o macchine e oggetti. Queste comunicazioni consentono l'acquisizione di dati da un elevato numero di dispositivi e applicazioni contemporaneamente, dove, come detto in precedenza, questi dati spesso vengono salvati in Cloud.

1.2 FancyBox

Il FancyBox, sviluppato dall'azienda Fancy Pixel srl, è un dispositivo di edge computing, progettato per rendere compatibili molti macchinari con i requisiti dell'Industria 4.0. Questo dispositivo è realizzato in diverse varianti, una di queste è basata sulla single board computer (SBC) Raspberry Pi 4, che integra al suo interno una variegata componentistica hardware, come per esempio una CPU (Central Process Unit), un'unità di elaborazione grafica (GPU), RAM (Random Access Memory) e diverse porte di connettività che permettono di interfacciarsi con diverse tipologie di hardware.

Questo dispositivo si basa su un sistema operativo custom ma, data la presenza di una versione basata su Raspberry 4, è possibile installare diverse tipologie di sistema operativo già disponibili, come per esempio Raspberry Pi OS o Ubuntu. Questo ci permette di scegliere il sistema operativo che più ci aggrada o semplicemente in base alle esigenze o funzionalità date da uno specifico OS.

L'architettura del FancyBox è sviluppata a 'moduli' (*Figura 1.1*) un modulo software può essere considerato come un'unità logica distinta dentro un sistema software che svolge o mette

a disposizione ad altri moduli specifiche funzioni. Un modulo può essere progettato per eseguire compiti come: la gestione dei dati, l'interazione con l'utente, l'esecuzione di processi o altro ancora e può essere aggiunto o rimosso in modo indipendente da altri moduli.

Il FancyBox è composto da diversi componenti, alcuni dei quali sono:

1. *UI*: si occupa di gestire l'interfaccia utente del dispositivo. Il suo scopo è fornire una modalità interattiva per gli utenti per interagire con il FancyBox e accedere alle sue funzionalità: visualizzazione delle informazioni, input dell'utente, navigazione e controllo, configurazione e personalizzazione, segnalazione e notifiche.

2. *Core*: rappresenta il nucleo del dispositivo, fa partire i processi singoli e gestisce le operazioni del FancyBox: gestione delle comunicazioni dei dispositivi connessi, elaborazione dei dati, controllo e supervisione, gestione degli eventi.

3. *HTTP Server*: è responsabile di fornire un'interfaccia di comunicazione basata sul protocollo HTTP (Hypertext Transfer Procolo). Questo modulo consente al FancyBox di ricevere richieste e inviare risposte attraverso la rete utilizzando protocolli HTTP.

4. *Broker MQTT*: è un server che serve per la comunicazione. Svolge il ruolo centrale nel protocollo di messaggistica MQTT (Message Queuing Telemetry Transport). il MQTT è un protocollo di messaggistica leggero e basato su un modello di pubblicazione e sottoscrizione (publish and subscribe), utilizzato per il trasferimento di dati tra dispositivi con restrizioni di risorse. Il Broker funge da intermediario tra dispositivi pubblicati (publisher) e i dispositivi sottoscrittori (subscribers), dove il suo ruolo principale è consentire la comunicazione tra dispositivi, inoltrando i messaggi dai publisher ai subscribers interessati.

5. *Networking*: si occupa di gestire le comunicazioni di rete permettendo di associare, mantenere e terminare connessioni di rete, consentendo lo scambio dei dati tra i vari dispositivi. Le sue funzionalità includono: configurazione della rete, connessione di rete, protocolli di comunicazione, gestione degli errori di rete e sicurezza di rete.

6. *Bluetooth*: consente la comunicazione wireless tra dispositivi quando si trovano nelle vicinanze l'uno dell'altro. Il Bluetooth è uno standard di comunicazione senza fili a corto raggio utilizzato per connettere dispositivi come smartphone, tablet, pc, e molti altri dispositivi che supportano il bluetooth. Le sue funzionalità includono: scoperta dei dispositivi, connessione tra dispositivi, scambio di dati, sicurezza.

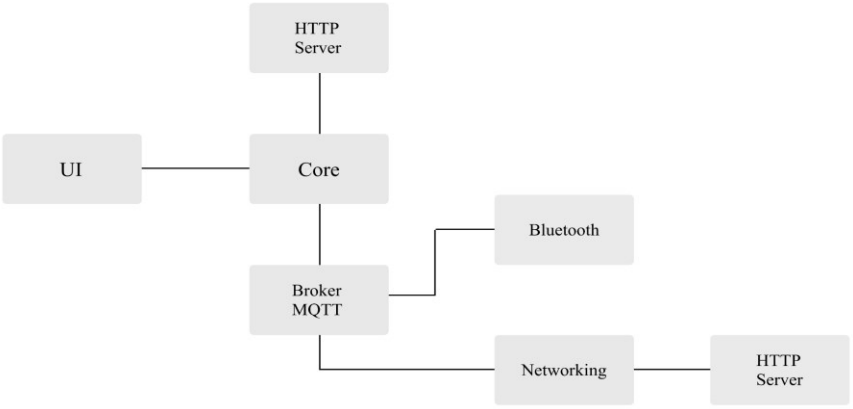

Figura 1.1: Architettura interna del FancyBox.

Il FancyBox, come anticipato, permette di abilitare ai requisiti di Industria 4.0 le macchine che non presentano questa funzionalità nativamente, integrando un sistema di monitoraggio e di controllo remoto. Il FancyBox integra al proprio interno anche la funzionalità di Gateway, cioè permette di collegare reti con protocolli diversi, e allo stesso modo, segmentare le reti in modo tale da garantire la sicurezza alle macchine ad esso collegate. Inoltre, permette la raccolta di dati da fonti eterogenee e tramite diverse tipologie di protocolli e la possibilità di esporre i dati raccolti tramite un formato unico, come per esempio MTConnect e OPC UA. Il primo protocollo citato, MT Connect, è un protocollo di comunicazione utilizzato per il monitoraggio delle macchine. Fornisce uno standard per la raccolta e l'organizzazione dei dati, consentendo la trasmissione di informazioni sullo stato della macchine, i dati dei sensori e altre metriche rilevanti. Dall'altra parte, OPC UA, è un framework di comunicazione industriale che consente lo scambio di dati e informazioni tra dispositivi, sistemi e applicazioni nel contesto dell'automazione industriale. Questo consente la comunicazione tra dispositivi e software eterogenei. In sintesi, MTConnect si concentra sulla raccolta e la trasmissione dei dati delle macchine, OPC UA è uno standard per la comunicazione e lo scambio di informazioni, così da essere facilmente integrati con i sistemi aziendali. Infine, il FancyBox, permette la visualizzazione dei dati in tempo reale grazie all'applicazione web raggiungibile da remoto.

Per poter visualizzare e configurare le funzionalità messe a disposizione dal FancyBox è necessario accedere alla sua interfaccia web, tramite la stessa rete o VPN (Virtual Private Network) se l'utilizzatore si collega da una rete esterna. L'interfaccia web è di default raggiungibile sulla porta 3000 del dispositivo, l'utente ha due possibilità di raggiungerlo: tramite connessione Ethernet o tramite connessione Wireless. Nel caso l'utente voglia collegarsi tramite l'hotspot che il FancyBox mette a disposizione, il nome del Wi-Fi a cui deve collegarsi avrà il nome dell'SSID (Service Set Identifier) del FancyBox: FBX-[Seriale dispositivo] e visitando la pagina *http://fbx-[Seriale dispositivo].local:3000* si avrà accesso alla Dashboard del FancyBox, *Figura 1.2*.

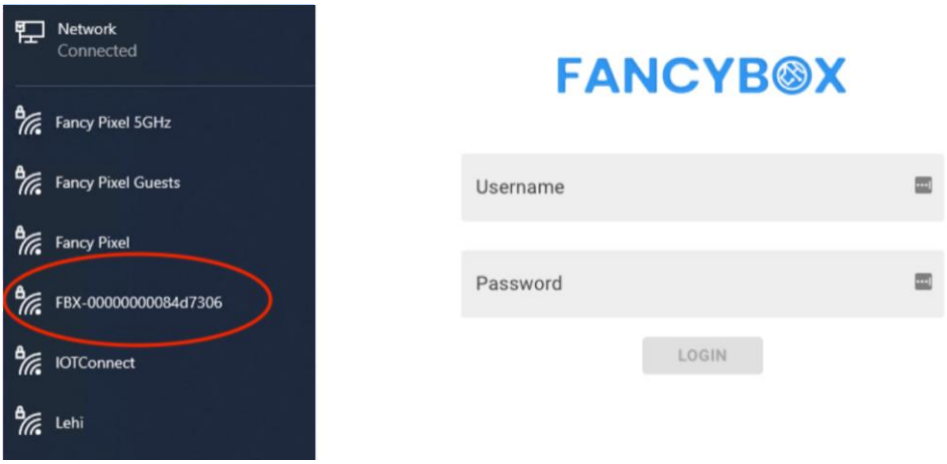

Figura 1.2: Accesso al FancyBox tramite hotspot WiFi

Se l'utente vuole un accesso tramite connessione Ethernet, bisogna collegare il cavo di rete, sotto la quale anche il FancyBox è collegato, e visitando la pagina *http://fbx-[Seriale dispositivo].local:3000* sarà possibile accedere al dispositivo.

La schermata iniziale della Dashboard, *Figura 1.3,* mostra una panoramica dello stato del dispositivo mostrando il carico di CPU, memoria e l'occupazione del disco. Abbiamo altre informazioni che vengono mostrate a schermo come: l'area del *Quick View* dove è possibile trovare le informazioni più specifiche riguardanti il dispositivo, utili a chi ha bisogno di dettagli tecnici. Un'altra parte fondamentale è quella del *Networking,* area che mette a disposizioni informazioni sulla connessione di rete o alla gestione delle interfacce di rete.

All'interno dell'interfaccia troviamo altre sezioni dedicate ad altre tipologie di informazione del dispositivo, come per esempio:

1. *License*: mostra le licenze attive sul dispositivo e le loro eventuali scadenze.
2. *Modules*: permette di gestire i componenti software aggiuntivi installati su un FancyBox.
3. *Administrator*: espone le funzionalità del sistema che permettono di gestire il dispositivo, come: aggiornamento del software, password amministratore, cronologia degli eventi, riavvio del dispositivo, ripristino impostazioni di fabbrica del dispositivo.

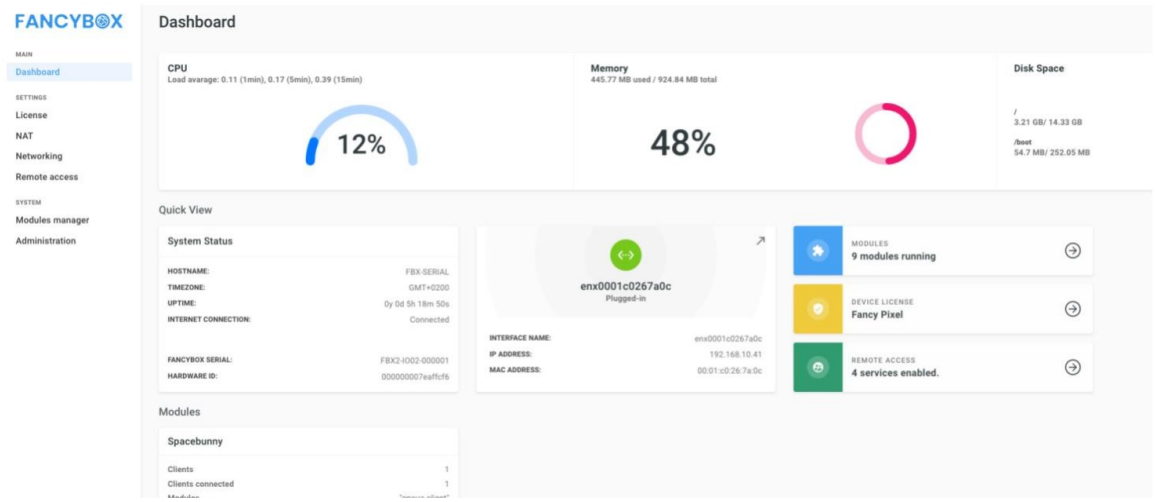

Figura 1.3: Dashboard del FancyBox

1.3 Obiettivo

L'obiettivo di questa tesi è quella di realizzare un applicativo mobile cross-platform che possa affiancare l'interfaccia web, in modo da portare migliore praticità, flessibilità e velocità nella visualizzazione delle informazioni del FancyBox, ma soprattutto, avere la possibilità di ripristinare la configurazione relativa al Networking qualora il modulo di Networking precedentemente citato, andasse fuori uso a causa di una mal configurazione delle impostazioni di rete da parte dell'utente.

1.3.1 Analisi dei requisiti dell'applicazione mobile

La fase di analisi dei requisiti è una parte fondamentale nello sviluppo di un software, ha lo scopo di capire al meglio il problema che si sta cercando di risolvere. Quando si inizia a pensare a come sviluppare un software si parte chiedendosi a chi è rivolto e a cosa deve implementare l'applicativo, partendo da concetti generali per poi andare nello specifico cercando nei migliori dei modi di scomporre il problema in altri meno complessi (denominata anche progettazione top-down). Per questa tesi l'applicazione mobile sviluppata provvederà due requisiti funzionali: **Monitoraggio del FancyBox e Recovery del FancyBox**; ulteriori funzionalità future come il ripristino totale del dispositivo o la modifica di impostazioni del FancyBox potranno essere esaminate e implementate da Fancy Pixel se necessarie al prodotto.

Il *Monitoraggio del FancyBox* permetterà ai clienti di visualizzare le informazioni del FancyBox, presentando due pagine possibili:

- *Dashboard Page*: fornisce all'utente informazioni di maggiore importanza che riguardano il dispositivo tra cui:
    - Stato connessione FancyBox: per capire se il dispositivo è raggiungibile.
    - Utilizzo della CPU, Memoria RAM e dello Spazio d'archivio.

- *Detail Page*: invece mette a disposizione due carte:
    - *System Status*: mostra le informazioni riguardanti i dettagli hardware del FancyBox tra cui *hostname, timezone, uptime, internet connection, FancyBox serial, hardware ID.*
    - *Network*: mostra *il nome dell'interfaccia* e *l'IP address*.

Per garantire sicurezza nell'accesso del FancyBox in modo tale che non ci siano intrusi, si è previsto un sistema di *Login* di credenziali prima di accedere alle funzionalità fornite dal dispositivo, questa pagina prevede che l'utente che voglia collegarsi al FancyBox debba immettere *username* e *password*.

Effettuato il login, l'utente si ritroverà nella *Home Page* dell'applicazione che mostra una lista di dispositivi attualmente registrati sull'applicazione, dove viene richiesto all'utente se vuole vedere solo quei FancyBox registrati sotto la rete a cui lo smartphone è attualmente connesso oppure di mostrare la lista intera dei dispositivi; se l'utente acconsente a questa funzionalità sarà per lui più semplice e veloce trovare i Fancybox desiderati. Una volta che viene selezionato il dispositivo da monitorare l'applicazione si muoverà nella *Dashboard Page* mostrando le informazioni riguardante al FancyBox con la possibilità di muoversi anche sulla *Detail Page* oppure di tornare alla *Home Page.*

Nel caso in cui poi l'utente vorrà fare il Logout di un dispositivo, potrà premere sul pulsante apposito dove verrà trasferito alla *Home Page*.

La seconda funzionalità è il *Recovery del FancyBox*. Supponiamo che il cliente del FancyBox abbia messo configurazioni sbagliate di indirizzo IP nei settings del dispositivo, questo potrebbe portare ad un `clash` di indirizzi IP, cioè un conflitto che si verifica quando due o più dispositivi sotto la stessa rete locale cercano di utilizzare lo stesso indirizzo, quindi sarebbe impossibile contattare una specifico dispositivo perché avrebbe lo stesso nome di un'altro dispositivo e per questo l'unico mezzo disponibile per poter comunicare con il FancyBox desiderato sarebbe con l'utilizzo del Bluetooth.
La funzionalità del ripristino del FancyBox è aiutare a contattare il dispositivo con cui si vuole comunicare. Per usufruire di questa funzionalità l'utente ha bisogno di accedere alla sezione *Bluetooth* tramite apposito simbolo predisposto su schermo. Una volta cliccato l'utente verrà portato nella *Scanning Page* dove potrà scannerizzare i dispositivi nelle vicinanze che possono fare uso di una connessione Bluetooth Low Energy e scegliere il dispositivo a cui si vuole connettere.
Prima di eseguire il ripristino delle impostazioni di rete, l'utente ha bisogno di eseguire il login per poter accedere allo scambio di dati con il FancyBox, questo per lo stesso motivo di sicurezza del Monitoraggio del FancyBox.
Infine nell'ultima schermata, la *Recovery Page*, verranno ripristinate le impostazioni relative al Networking e verrà attivato l'hotspot del FancyBox che permetterà all'utente di collegarsi subito al dispositivo tramite WiFi.

1.3.2 Analisi dei requisiti lato dispositivo edge

Per permettere lo scambio di dati tramite Bluetooth abbiamo bisogno di un servizio BLE fornito dal modulo Bluetooth del FancyBox in modo tale che se ci sono delle problematiche con il modulo del Networking, siamo sempre in grado di comunicare con il dispositivo.
Sul lato FancyBox quindi abbiamo dei requisiti funzionali per il modulo *Bluetooth*, infatti questo modulo avrà il ruolo di *Periferica*, cioè un dispositivo che offre servizi BLE e che può essere rilevato e connesso da altri dispositivi chiamati *Centrali*, come lo smartphone, dispositivo che usufruisce del servizio.
Quello che deve fare il dispositivo edge nel modulo Bluetooth che abbiamo implementato è:

1. Ricevere i dati tramite BLE dallo smartphone.
2. Estrarre i dati ricevuti.
3. Eseguire la chiamata HTTP e attendere la risposta.
4. Inviare la risposta allo smartphone.

La libreria di riferimento per questa implementazione sarà *Bleno*, di cui viene parlato nel Capitolo 2.

Dopo aver dato un background generale sui vari concetti che fanno parte di questa tesi, l'Industria 4.0, il FancyBox e cosa dobbiamo aspettarci dall'applicazione mobile, possiamo andare più in profondità parlando più specificamente dell'applicativo e delle tecnologie che sono state usate.

Capitolo 2

Tecnologie

2.1 TypeScript

Il linguaggio di programmazione che è stato utilizzato per lo sviluppo dell'applicazione è TypeScript. Questa tecnologia è stata proprio scelta per motivi che la portano ad essere il linguaggio migliore per questa app, infatti, possiamo usare un unico codebase di codice per le varie piattaforme e oltretutto l'azienda aveva molta familiarità con TypeScript che porta ad essere più efficienti nel caso in cui si voglia implementare nuove funzionalità.

TypeScript è un superset di JavaScript con tipizzazione statica sviluppato da Microsoft nel 2012. É stato creato per risolvere alcune carenze di JavaScript, come la mancanza di tipizzazione statica, che possono aiutare gli sviluppatori a rilevare errori prima dell'esecuzione del software, rende più adatto allo sviluppo di app su larga scala fornendo una migliore organizzazione e manutenzione del codice e per lo più TypeScript è compatibile con qualsiasi ambiente JavaScript.

2.2 React Native

Prima di introdurre React Native dobbiamo parlare di React. React è una libreria JavaScript che ci permette di creare interfacce utenti riutilizzabili, infatti usa un approccio basato sul riutilizzo di componenti che consentono di frammentare la *UI (User Interface)* in parti più facili da gestire, sono come delle funzioni in TypeScript che accettano input e ritornano componenti grafici dinamici che descrivono cosa dovrebbe apparire. Quando React viene affiancato a TypeScript riusciamo ad ottenere una combinazione che ci permette di sviluppare applicazioni web più robuste, andando a creare una struttura gerarchica di componenti, (*Figura 2.1*). React Native, dall'altra parte, è un framework per lo sviluppo di applicazioni mobile che permette di utilizzare TypeScript e React per creare applicazioni native sia per iOS che per Android. Quando si utilizza TypeScript e React Native, riusciamo a beneficiare di tutte le funzionalità offerte da React, ma con l'obiettivo di sviluppare su mobile.

Quando si crea un'applicazione React Native è possibile anche farlo tramite Expo. Expo è un framework che estende React Native fornendo un set di strumenti, servizi e librerie aggiuntive che semplificano il processo di sviluppo. Un grande beneficio arriva dall'utilizzo di Expo Application Service, un servizio Cloud che automatizza il processo di build e di rilascio dell'eseguibile su App Store e Google Play Store, permettendo al team di sviluppo di risparmiare tempo. Poi tramite un app Expo Go i developer possono eseguire e fare debug

dell'app direttamente su smartphone o tramite simulatore su pc, tutto ciò senza utilizzare software di sviluppo native come Xcode per iOS e Android Studio per Android. Noi per lo sviluppo della nostra applicazione abbiamo infatti utilizzato React Native affiancato a Expo.

React Native per processare un codice universale JavaScript in uno nativo per il sistema operativo compila in JavaScript ma dopo di che va a convertire le componenti in native utilizzando API per interagire direttamente con il sistema operativo sottostante e moduli specifici della piattaforma. L'elemento principale dell'architettura di React Native è il *Bridge*, ponte (*Figura 2.2*). Questo sistema utilizza la libreria React per renderizzare l'applicazione su un dispositivo, il bridge permette di trasformare il codice JavaScript quindi anche TypeScript in componenti nativi e viceversa. Riceve la chiamata in codice JavaScript e sfrutta le API (Kotlin, Swift), che consentono il rendering nativo dell'app. Il processo non influisce sull'esperienza utente perchè queste chiamate sono asincrone e avvengono separatamente dal thread principale, quindi anche quando abbiamo una ricomposizione di una componente, esempio quando l'utente preme su un pulsante, l'applicazione non si blocca.

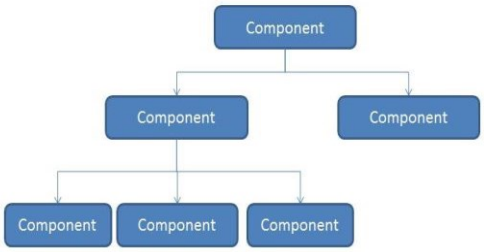

Figura 2.1: Struttura DOM delle componenti

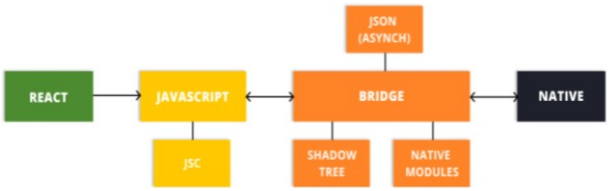

Figura 2.2: Architettura React Native

Quando si parla di componenti non possiamo non parlare dello 'stato di un componente', informazioni o dati specifici di cui un componente tiene traccia e che possono cambiare nel corso del tempo, influenzando l'aspetto o il comportamento dell'applicazione.

Nelle applicazioni vengono utilizzati gli 'observable' per gestire lo stato o le informazioni dinamiche, strumento utile per 'osservare' i cambiamenti in modo reattivo.
In React gli strumenti disponibili sono gli 'hooks'. Senza gli hooks non si possono variare gli stati del Virtual DOM (Document Object Model), il documento che organizza l'interfaccia in una struttura ad albero per manipolare dinamicamente il contenuto della schermata permettendo esperienze con l'utente interattive e dinamiche (*Figura 2.1*), perché non è capace di registrare le modifiche apportate.
React mette a disposizione tanti tipi di hooks che variano in base a ciò che vogliamo fare con lo stato di un componente e l'hook che ci permette di tenere traccia del suo stato è il *useState(valore)* che si preoccuperà di osservare e aggiornare a video il componente con il suo stato più recente.

Vediamo un esempio di come funziona l'hook *useState*:

Con un bottone e un testo nella nostra schermata dell'app, vogliamo che quando clicchiamo il bottone vada a cambiare il testo da `default` a `Hello`, quindi avremmo una funzione

changeTitle che quando premiamo sul bottone, viene azionata e quello che fa e andare a cambiare il testo del *title* in `Hello`.

```
const HomePage = () => {                    // componente personalizzato HomePage
  let title = 'default' ;                   // variabile per il titolo

  const changeTitle = () => {               // funzione che cambia il titolo
     title = 'Hello' ;
  }

  return (                                  // cosa ritorna questo componente, cioè il suo
                                            // aspetto, qua abbiamo:
     <View>                                 // una layer padre per contenere tutti i sotto
                                            //  componenti
       <Text> {title} </Text>               // componente testo
       <Pressable onPress={ ()=>{changeTitle} } >       // componente bottone con testo
         <Text> Change Title </Text>
       </Pressable>
     </View>
  );
}
```

Il codice funziona correttamente, ma a schermo vedremo ancora la scritta `default`, questo perché la variabile testo è una variabile `non reattiva` cioè React non si preoccupa di ricomporre il componente del testo ogni qualvolta che il suo valore cambia e quindi il testo che vedremo rimarrà sempre `default`.
Per farlo cambiare, dobbiamo tenere traccia del cambiamento di stato del componente utilizzando *useState* dove porterà al cambiamento del testo in `Hello`. Dunque osserva lo stato del componente, se il suo valore cambia, ricomponi il componente con il nuovo dato, e il nuovo codice diventerebbe:

```
const HomePage = () => {
  // let title = 'default' ;      vecchio codice
  // andiamo a inizializzare un hook in questo modo
  // [nomeVariabile, setNomeVariabile] = hook di React in questo caso useState
  const [title, setTitle] = useState<string>('default');

  /* const changeTitle = () => {   non più necessario perchè abbiamo la funzione setTitle
      title = 'Hello';
  } */

  return (
     <View>
       <Text> {title} </Text>
       // chiamiamo direttamente la funzione che abbiamo dato alla variabile, setTitle
       // per farli cambiare lo stato, e quindi cambiare titolo poi React
       // ricompone il componente mostrando 'Hello'
       <Pressable onPress={ ()=>{setTitle('Hello')} } >
```

```
        <Text> Change Title </Text>
      </Pressable>
    </View>
  );
}
```

2.3 Redux Toolkit

Insieme a TypeScript e React Native viene affiancata anche un'altra tecnologia che è Redux Toolkit. Nello sviluppo di applicazioni più o meno grandi la best practice è sempre dividere concetti differenti, in questo caso vogliamo andare a dividere la UI dall'architettura del sistema, in modo che sia più facile da mantenere.

L'applicazione che abbiamo sviluppato contiene dati che vanno salvati, esempio la lista di FancyBox, in qualche punto dell'app in modo tale che non vengono cancellati ogni volta che il software viene avviato. Redux è un framework che si integra perfettamente con già le due tecnologie che stiamo usando, ed è un contenitore di stato, cioè fornisce un modo semplice per centralizzare lo stato e la logica di un'applicazione sia web che mobile. Qualora avessimo bisogno che la pressione di un bottone vada a salvare qualcosa, il componente che viene cliccato va ad aggiornare lo stato attraverso le *actions* (azioni) che descrivono ciò che è successo nell'applicazione. Quando viene emessa un'azione, redux la invia a una funzione chiamata *reducer* (riduttore) che aggiorna lo stato dell'applicazione in base alla specifica azione. Una volta che lo stato è aggiornato, redux invia questa informazione a tutte le parti sia a livello di logica che a livello di componenti che utilizzano lo stato consentendo di aggiornare la UI in modo coerente (*Figura 2.3*).

Redux Toolkit introduce anche il concetto di *slice*, un approccio consigliato per organizzare e gestire lo stato nelle applicazioni, dove andiamo ad `estrarre` una porzione specifica di stato globale dell'applicazione per creare uno stato locale più piccolo e focalizzato. Quando si fa lo *slice*, si va a creare un modulo separato che gestisce solo quella porzione di stato e si vanno ad aggiungere i *reducers* e le *actions* relative. Riprendendo l'esempio precedente della lista di FancyBox, andiamo a creare uno *slice* il cui compito è solo di aggiornare lo stato della lista, tramite *reducers* che avviano le funzioni di *aggiunta, rimozione o la modifica* di FancyBox, che vedremo nel capitolo 3 di questa tesi.

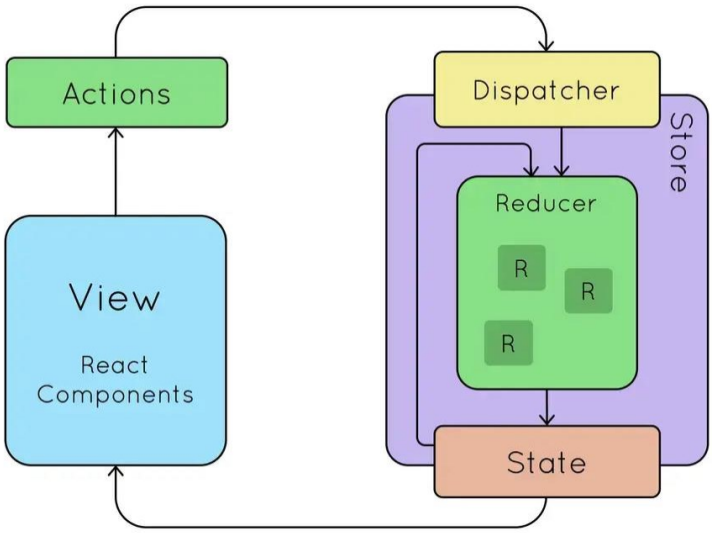

Figura 2.3: Funzionamento RTK

2.4 Redux Toolkit Query

La nostra applicazione oltre che a lavorare a stretto contatto con lo stato generale, utilizza anche chiamate HTTP, per eseguire le chiamate API fornite dal FancyBox. Nella famiglia delle tecnologie di Redux Toolkit è presente anche Redux Toolkit Query (RTK Query) una libreria aggiuntiva fornita da Redux Toolkit per semplificare la gestione delle richieste HTTP e lo stato dei dati, permettendo di eliminare il bisogno di scrivere manualmente la logica di recupero dei dati e della loro memorizzazione nella cache dei dati, evitando codice ridondante. Per capire meglio vediamo un pò di codice esemplificativo:

```
default function App() {
  // inizializza un hook che usa useState
  const [data, setData] = useState();
  const [isLoading, setIsLoading] = useState<boolean>(false);
  const [isError, setIsError] = useState<boolean>(false);

  // hook di React che permette quando il componente App viene
  // 'montata' esegue il codice al suo interno,
  // dove esegue una chiamata http ad un api per prendere i dati del FancyBox
  // Se la risposta esiste la converte in
  // formato json e lo inserisce dentro alla variabile data.
  useEffect( () => {
    setIsLoading(true);
    fetch('http://192.168.10.102:3000/api/stats');
      .then((res) => res.json())
      .then((data) => {
        setData(data.message);
        setIsLoading(false);
      })
      .catch((error) => {       // eseguito se chiamata ha avuto problemi
```

```
        setIsError(true);
        setIsLoading(false);
      }
  }, []) ;
}
```

il codice appena presentato è come normalmente le chiamate HTTP vengono fatte in TypeScript, anche se in questo viene chiamata automaticamente quando questa componente viene composta e usiamo la variabile *isError* un booleano che ci dice se ci sono stati errori nell'esecuzione della chiamata e usiamo *isLoading* un booleano per capire se stiamo ancora aspettando una risposta dalla chiamata. Come si può vedere dobbiamo gestire tutto noi gli eventi della chiamata è questo può portare a scrivere codice boilerplate come il *setIsLoading*. Quello che ci permette di fare RTK Query è quello di creare *Mutazioni* utilizzate per andare cambiare dati cioè sono come le chiamate POST e le *Query* che sono usate per accedere ai dati, come le GET. Vediamo un esempio:

```
const api = createApi({
  reducerPath: 'core',
  baseQuery: fetchBaseQuery({
    baseUrl: 'http://192.168.10.102:3000'  // base url
  }),
  endpoints: (build) => ({
    login: build.mutation<IUserData, ILoginInput>({        // mutazione
      query: (body) => ({ url: '/api/session', method: 'POST', body })
    }),
    stats: build.query<IApiStatsResponse, undefined>({      // query
      query: (params) => ({ url: '/api/stats', method: 'GET', params })
   })
});
```

In questo codice esemplificativo usato per capire il concetto troviamo: *createApi* genera un *reducer* per gestire lo stato relativo alle chiamate API , questo *reducer* viene automaticamente montato nello store di Redux, in più genera un *middleware* che deve essere messo sempre nello store di Redux che permettono di fare il polling, il caching e altre feature di RTK Query. *createApi* che prende come parametri:

1. *reducerPath*: specifica il percorso all'interno dell'albero di stato in cui il reducer generato sarà montato, in modo tale che quando abbiamo bisogno di chiamare un endpoint possiamo farlo tramite *store.nomeReducerPath.useNomeQuery*.
2. *baseQuery*: dove usa *fetchBaseQuery* che funziona similmente alla chiamata *fetch* normale.
3. *endpoints*: qua è dove scriviamo gli endpoint che la nostra app utilizza, dove dato che utilizziamo TypeScript possiamo specificare *<ilDatoCheCiAspettiamo, ilDatoCheDiamoComeInput>*.

Quello che poi Redux Toolkit Query farà in modo nascosto sarà crearci gli hook per facilitare l'utilizzo delle chiamate API definite negli endpoint, ma non solo crea anche le funzioni per controllare errori *isError* e controllare lo stato di caricamento tramite *isLoading.*

2.5 Libreria di terze parti

Per poter realizzare la comunicazione tramite Bluetooth Low Energy (BLE) abbiamo utilizzato librerie esterne open-source, e queste sono *react-native-ble-plx* e *bleno*. Prima di vedere queste librerie dobbiamo introdurre cos'è il BLE. Il Bluetooth Low Energy è una versione migliorata della tecnologia Bluetooth, progettata per trasferire dati a basso livelli di energia. Questa tecnologia è ottimizzato per consumare meno energia rispetto alla versione standard Bluetooth, consentendo una maggiore durata della batteria per i dispositivi. Le caratteristiche del BLE sono:

1. *Connessione diretta:* supporta connessioni dirette tra dispositivi, consentendo la trasmissione di dati in modo affidabile tra dispositivi accoppiati.
2. *Scambio di dati:* consente la trasmissione di piccoli pacchetti di dati, consentendo il trasferimento di informazioni come dati di sensori.
3. *Modalità stand-by:* consente ai dispositivi di risparmiare energia quando non sono attivamente impegnati nella trasmissione o ricezione dei dati.

Il BLE può essere utilizzato in due modi: essere un *Central* 'centrale' o essere un *Peripheral* 'periferica', in base a quello che vogliamo implementare e un dispositivo può comunque essere entrambi i ruoli, (*Figura 2.4*). Un *Central* device non è altro che un dispositivo che avvia lo scan per trovare altri dispositivi BLE ed eseguire la connessione con essi, esempio come lo smartphone. Invece i *Peripheral* device sono quei dispositivi che 'pubblicizzano' un servizio e si rendono visibili agli altri dispositivi, esempio il FancyBox. Dopo che si è eseguita una connessione tra i due dispositivi le procedure per la comunicazione dei dati tra i due è descritto dal *Generic Attribute Profile* (GATT) che organizza e definisce i dati che si possono scambiare, dove ci permette di andare a modificare il MTU Maximum transmission unit, che definisce quanti pacchetti si possono inviare al massimo nella comunicazione, normalmente è settato su 20 byte. GATT definisce due ruoli:

1. *Server GATT*: il dispositivo espone un insieme di servizi e caratteristiche che possono essere letti, scritti o monitorati.
2. *Client GATT:* il dispositivo che legge, scrive o monitora le caratteristiche dei servizi offerti dal server.

Queste caratteristiche sono dati che rappresentano informazioni specifiche all'interno di un dispositivo BLE. Ogni caratteristica è definita da una serie di attributi che ne specificano il tipo di dato, le proprietà e il comportamento, questi sono:

1. *UUID:* Universal Unique Identifier, un identificatore univoco che identifica una caratteristica all'interno di un servizio.
2. *Valore:* il dato effettivo, che può essere numero, stringa, o qualsiasi altro tipo.

3. *Proprietà*: definiscono il comportamento della caratteristica. Possono includere la lettura, scrittura, la notifica (cioè invia il valore al central device ogni qualvolta il dato che osserviamo cambia) e altre funzionalità.
4. *Permessi:* le autorizzazioni specificano quali operazioni possono essere eseguite sulla caratteristica da parte del client GATT.

I servizi invece sono collezioni di caratteristiche. Anche qua ai servizi è possibile dare un UUID univoco che lo identifica, ad esempio un servizio di monitoraggio del battito cardiaco potrebbe includere caratteristiche per leggere il battito cardiaco attuale, impostare allarmi, monitorare livello batteria del dispositivo e altro.

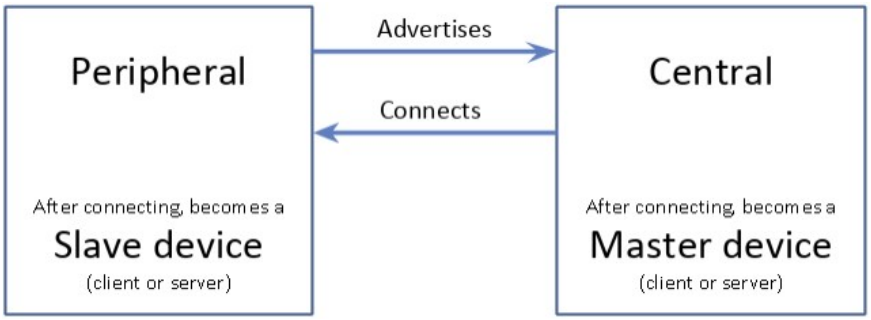

Figura 2.4: Central and peripheral

2.5.1 React-native-ble-plx

Questa libreria fornisce API per React Native per la comunicazione con dispositivi dotati di BLE nelle applicazioni mobile. Ci permette di creare client GATT quindi un *Central*:

1. *Scansione delle periferiche BLE:* effettua scansioni delle periferiche BLE disponibili nelle vicinanze.
2. *Connessioni a periferiche BLE:* una volta identificata una periferica BLE di interesse, puoi stabilire una connessione con essa.
3. *Lettura e scrittura di caratteristiche:* leggiamo e scriviamo dati nelle caratteristiche della periferica.
4. *Notifiche di caratteristiche:* possiamo registrarci a un servizio per ricevere notifiche di quando una caratteristica cambia o viene aggiornata. Ciò permette di rimanere aggiornato sui cambiamenti di stato o sui dati inviati dalla periferica senza dover effettuare ripetute richieste di lettura.

2.5.2 Bleno

Per il server GATT, invece utilizziamo *Bleno* una libreria Node.js che consente di creare un dispositivo periferico. Questa libreria fornisce API per la creazione e la gestione di periferiche BLE personalizzate, le principali funzioni sono:

1. *Creazione di un dispositivo BLE:* crea un dispositivo periferico su un dispositivo che supporta il BLE. Ci permette di definire il nome del dispositivo che verrà visualizzato alla scansione e l'UUID del servizio e delle caratteristiche che offrirà.
2. *Creazione di servizi e caratteristiche:* possiamo definire servizi e caratteristiche BLE, con i relativi UUID, i dati che conterranno, le varie proprietà di lettura, scrittura e notifica.
3. *Gestione delle connessioni:* puoi gestire gli eventi di connessione e disconnessione e interagire con i client durante la connessione
4. *Annuncio del dispositivo:* impostare l'annuncio del dispositivo cioè renderlo visibile ad altri dispositivi BLE nelle vicinanze, specificando il nome, la potenza di trasmissione e altro.

Capitolo 3

Progettazione e Sviluppo

3.1 Gestione dei permessi

La Gestione dei permessi nelle applicazione Android e iOS sono essenziali per garantire la sicurezza e la privacy degli utenti. I permessi permettono agli utenti di controllare quali informazioni e risorse l'app può utilizzare e vengono richiesti per vari motivi, come:

1. Protezione della privacy: consentendo agli utenti di decidere quali dati e risorse del dispositivo desiderano condividere con un'app.
2. Sicurezza del dispositivo: aiutando a prevenire abusi da parte di app dannose o malware. Consentendo solo le autorizzazioni necessarie, gli utenti possono ridurre il rischio di esporre il proprio dispositivo a minacce potenziali.
3. Controllo dell'utilizzo delle risorse: alcuni permessi riguardano l'accesso alle risorse del dispositivo, come fotocamera, microfono o la posizione del GPS. Consentire l'accesso solo alle app di fiducia può aiutare a prevenire l'utilizzo non autorizzato delle risorse del dispositivo.

Nell'applicazione che abbiamo sviluppato abbiamo bisogno di richiedere i permessi, perché abbiamo sviluppato funzionalità che vanno a utilizzare la localizzazione del dispositivo e l'accesso alla rete wifi in cui si trova. I permessi nelle applicazioni native in Android vanno messe nel file *AndroidManifest.xml* un file in cui vengono fornite le informazioni fondamentali sull'applicazione al sistema operativo Android, dove esempio troviamo il nome del pacchetto, la versione, l'icona dell'applicazione e altro. Nello stesso modo anche iOS ha un file simile chiamato *Info.plist*, ma dato che noi utilizziamo Expo che ci va a semplificare la struttura dei file per iOS e Android non abbiamo accesso a i loro manifest, per questo andiamo a mettere i permessi che abbiamo bisogno direttamente nel manifest dell'applicazione React Native nel file chiamato *app.json*.
Nella nostra applicazione chiediamo solo due tipi di permessi:

1. Permesso di *Localizzazione:* per accedere alle info del WiFi, utili se l'utente vuole filtrare la lista dei suoi FancyBox per rete locale.
2. Permesso di *Scansione* dei dispositivi vicini per il BLE.

In entrambi i casi le righe di codice per richiedere il permesso è uguale:

```
let { status } = await Location.requestForegroundPermissionAsync();
if (status != 'granted') return;     // controlla se accettato
await Location.getCurrentPositionAsync({});   // aspetta dati sulla posizione
```

cambierà solo in quale funzione verrà implementata.
La riga di codice '*Location.requestForegroundPermissionAsync()'* è quello utilizzato per richiedere i permessi, il resto sono le operazioni aggiuntive che facciamo se l'utente ha accettato o meno i permessi, esempio:

```
React.useEffect(() => {
  // creiamo una funzione per prendere le info del wifi a cui siamo collegati
  const findNetworkInfo = async () => {
    try{
      // le seguenti 3 righe di codice sono per richiedere i permessi
      let { status } = await Location.requestForegroundPermissionAsync();
      if (status != 'granted') return;     // controlla se accettato
      await Location.getCurrentPositionAsync({});   // aspetta dati sulla posizione
      // prendo le informazioni della connessione attuale
      const state = await NetInfo.fetch();
      // se è tipo wifi, ritorna il suo ssid
      if (state.type === 'wifi'){
        return state.details?.ssid;
      } else {
        return undefined;
      }
    } catch (err) {
     console.log(err);
    }
  };
  ...
  ...
}, []);
```

3.2 Progettazione e Sviluppo delle attività

Quando si va a sviluppare un software è importante scegliere un'organizzazione architetturale del software identificando i diversi componenti in base al ruolo che essi devono ricoprire e definendo le interazioni tra gli stessi all'interno del sistema, in modo da suddividere il codice in moduli del progetto in modo efficiente, chiaro e pulito. L'applicazione *FancyBox Companion App* usa il pattern architetturale MVC, *Model-View-Controller,* che prevede la suddivisione della struttura software in tre macro componenti:

1. *Model*: rappresenta i dati dell'applicazione e la logica che si occupa di manipolarli in lettura e scrittura, questa parte viene affidata a Redux Toolkit e Redux Toolkit Query.
2. *View:* definisce tutto ciò che compone l'interfaccia grafica verso l'utente per presentare i dati del modello, dato dalle componenti della UI.
3. *Controller*: implementa la logica di controllo dell'applicazione e svolge la funzione di intermediario tra la vista e il modello, traduce i comandi ricevuti, notifica la vista nel caso di aggiornamenti ai dati nel modello, dunque le varie schermate dell'app.

L'obiettivo principale del MVC è quello di disaccoppiare i dati del sistema dalla loro rappresentazione, così da renderli riutilizzabili in diversi contesti.

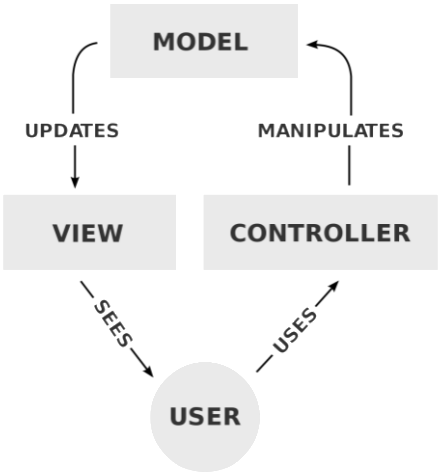

Figura 3.1: pattern MVC

Un'altra cosa fondamentale quando si sviluppa un software è anche capire da subito quello che sarà lo user-flow dell'utente, cioè il percorso che l'utente può seguire attraverso l'applicativo. In React Native per poter permettere la navigazione delle schermate è possibile utilizzare la libreria *react-navigation*, che ci permette di definire uno *Stack* di navigazione, cioè una struttura dati utilizzata per gestire la navigazione tra pagine, che verrà implementato nel file iniziale di avvio dell'app cioè nel *App.js*.

```
function AppNavigator(props: AppNavigatorProps){
  // definizione dello Stack per la navigazione delle pagine
  return(
  // crea uno Stack navigator con al suo interno le pagine che compongono la nostra
  //App
    <Stack.Navigator
       initialRouteName = {props.isAppFirstLaunch ? AppRoutes.OnboardingPage:
         AppRoutes.EndpointPage} > // inizializza come prima pagina
       <Stack.Screen name={AppRoutes.OnboardingPage}
         component={OnBoardingPageScreen} options={{headerShown: false}} />
       <Stack.Screen name={AppRoutes.EndpointPage}
         component={EndpointListPageScreen} options={{headerShown: false}} />
       <Stack.Screen name={AppRoutes.DetailPage}
         component={DetailNavigator} options={{headerShown: false}} />
       <Stack.Screen name={AppRoutes.BluetoothPage}
         component={BluetoothNavigator} options={{headerShown: false}} />
    </Stack.Navigator>
}
```

Nel codice possiamo vedere che lo Stack.Navigator contiene un set di schermate in cui si può navigare, e inizializza come prima pagina o *OnboardingPage* o *EndpointPage*, perchè se

l'utente è la prima volta che apre l'app viene mostrata l'*OnboardingPage* altrimenti l'*EndpointPage*. Le props, cioè le proprietà, delle schermate sono:

1. *name:* indica il nome o l'identificatore univoco della schermata, in modo tale che se abbiamo necessità di chiamare la schermata lo facciamo tramite il suo nome.
2. *component*: specifica il componente da renderizzare in questo caso rappresenta delle pagine
3. *options:* permette di dare opzioni aggiuntive alla schermata, in questo caso togliamo l'header che React Native mette alle schermate, perchè ne vogliamo uno personalizzato.

Un'applicazione potrebbe contenere più Stack di navigazione, che possono essere sia innestati tra loro che separati, e vengono utilizzati per definire una gerarchia di quali sono le pagine raggiungibili da una schermata, contribuendo così a mantenere ordine e coerenza nell'applicazione.

Onboarding Page

La pagine del primo avvio dell'applicazione. L'Onboarding nelle applicazioni mobile è il processo che consente di spiegarne i vantaggi e coinvolgere gli utenti sin dai primi istanti, (*Figura 3.2*). Questo rappresenta la prima impressione che diamo del nostro progetto, questa schermata infatti viene messa come prima pagina che l'utente vede. L'implementazione di questa schermata è molto semplice: a livello di componente UI, quindi di View, abbiamo una lista chiamata FlatList, dove ha vari props, tra quali *data* che prende i dati: titolo, immagine, descrizione da mostrare e *renderItem* definisce come ogni elemento della lista deve essere renderizzato, in questo caso il componente che renderizza è una singola pagina della Onboarding.

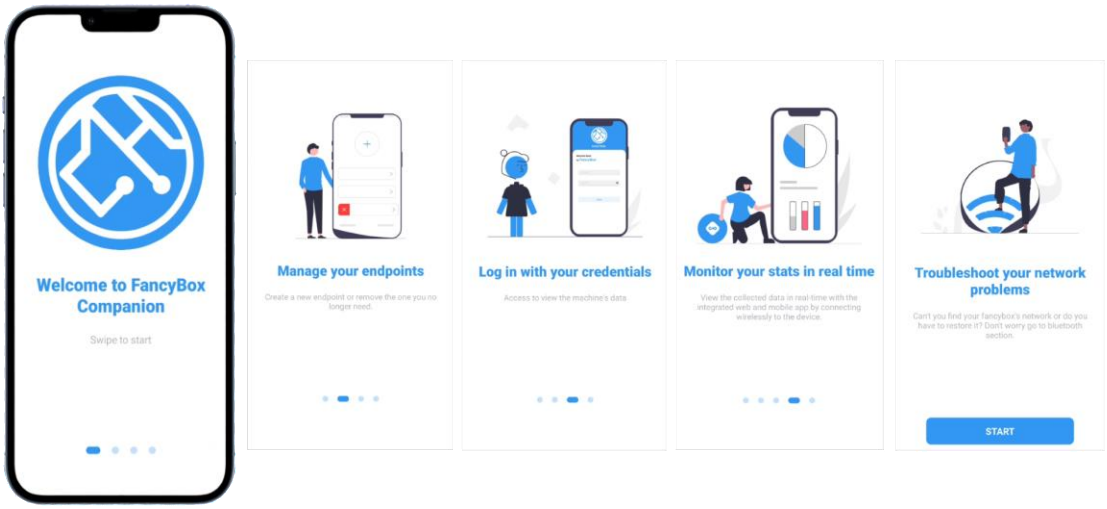

Figura 3.2: Onboarding Page

Per poter visualizzare solo una volta (al primo avvio dell'app) questa schermata, abbiamo utilizzato la libreria AsyncStorage di React Native.
Questa libreria viene utilizzata per memorizzare dati in modo asincrono sul dispositivo dell'utente, generalmente utilizzato per conservare piccole quantità di dati locali, senza perderli dopo la chiusura dell'applicazione. I dati vengono memorizzati in formato chiave-valore, e ci sono tre operazioni principali: *setItem, getItem* e *removeItem.*

1. *setItem('chiave', 'valore')*: viene utilizzato per memorizzare nella memoria locale un valore associato alla sua chiave di accesso.
2. *getItem('chiave')*: ci permette di prelevare il dato dalla memoria che è stato associato alla *chiave*.
3. *removeItem('chiave')*: utilizzato per rimuovere il dato dalla memoria associato alla chiave.

Quando l'utente preme sul pulsante 'Start' dell'ultima pagina dell'Onboarding, l'applicazione va a salvare nella memoria del dispositivo tramite *setItem* un dato con chiave: *is_app_first_launch* e valore: *false*. Ogni volta che l'applicazione viene aperta viene eseguito, con *getItem*, un controllo di questa variabile: se esiste allora sappiamo che ha valore *false* e quindi l'app ha già eseguito il suo primo lancio, se invece non esiste vuol dire che l'applicazione è stata avviata per la prima volta.
Infine esegue una navigazione verso la *Home Page* dove l'utente può scegliere se monitorare il FancyBox o ripristinarlo.

3.2.1 Monitoraggio del dispositivo edge

Quanto detto in precedenza, il FancyBox mette a disposizione delle API per permetterci di fare richieste HTTP. Per effettuare le richieste al FancyBox abbiamo bisogno di un token di accesso. Il *token* è una stringa di dati che serve per autenticare l'identità di un utente o quando si accede ad una risorsa.
Le richieste che faremo avranno bisogno del *token* di accesso dell'utente, che verrà assegnato dal FancyBox ogni volta effettuato il login. Dopo di che questo token lo salveremo nello *store* e ogni volta che facciamo delle richieste al FancyBox, includiamo il token nella chiamata.

EndpointList Page

Questa pagina rappresenta la nostra *Home Page*, la prima schermata che l'utente vede ogni volta che apre l'applicazione (*Figura 3.3*). La pagina permette di:

1. Visualizzare una lista di FancyBox attualmente registrati sul dispositivo con la possibilità di mostrare solo quelli sotto la rete locale corrente.
   Per quest'ultimo abbiamo bisogno dei permessi di accesso alla localizzazione per accedere al nome della rete locale. Se l'utente non ha ancora dato una risposta effettiva

tra 'Sì' e ‘No’ data dal dialogo di selezione (*Figura 3.3*), l’applicazione andrà a mostrare questo Alert ogni volta che l’app viene avviata, invece se approvata, l’applicazione chiederà di attivare il GPS dello smartphone.

2. Aggiungere un nuovo FancyBox premendo sul pulsante ‘+’: dove viene chiesto di inserire indirizzo IP del dispositivo edge da monitorare.
3. Rimuovere un FancyBox: eseguendo un'azione di ‘swipe’ da sinistra verso destra.
4. Cercare il FancyBox tramite barra di ricerca, tramite il nome del dispositivo o tramite suo indirizzo IP.

La schermata permette anche di riconoscere quali FancyBox hanno ancora il token di accesso valido, riconoscibile tramite l’icona vicino al nome del dispositivo illuminata di colore blu, se invece l’icona è di colore grigio significa che l’utente se vuole monitorare il FancyBox ha bisogno di eseguire l’accesso tramite login.

Infine nella parte alta dello schermo, specificamente nell’header, troviamo un bottone con il simbolo del Bluetooth che ci porterà nella sezione dedicata al recovery del dispositivo.

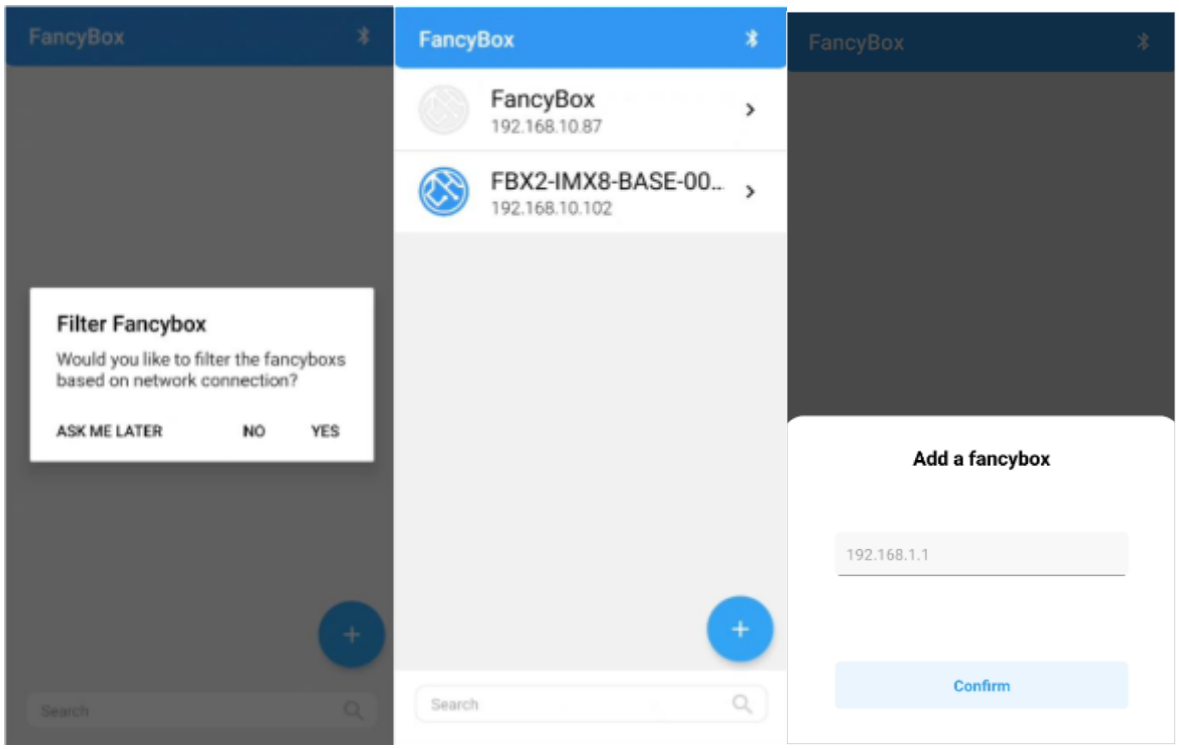

Figura 3.3: Home Page

Per vedere come Redux Toolkit ci aiuti nello store di una FancyBox nella memoria locale del dispositivo vediamo il codice usato per aggiungere un dispositivo. Nel file che contiene il componente della pagina *Home Page* troviamo una funzione per l’aggiunta del FancyBox che verrà azionata quando il bottone ‘Confirm’ viene cliccato:

```
const handleConfirmPress = () => {
  // controlla se ho un endpoint in input e il suo formato sia x.x.x.x
  if( !isEmpty(endpoint) && endpointRegExp.test(endpoint) ){
    // invia l'azione allo slice dello store dei fbx
    dispatch(slice.actions.addEndpoint(endpoint));
    setIsVisible(false);     // chiudiamo la bottom card
    setEndpoint('');         // puliamo l'input
  }
}
```

Quello che avviene in questo codice è semplice, abbiamo una funziona che viene chiamata alla pressione del bottone conferma, l'if viene eseguito solo se l'utente ha inserito un endpoint di un FancyBox nell'input di testo e se ha rispettato il formato *x.x.x.x,* cioè almeno 4 numeri separati da un '.', se così allora invia l'azione al reducer che si trova nello slice con payload, cioè con argomento l'endpoint del FancyBox, dopo di che esegue l'azione che ha nel reducer, poi chiude la Card per l'aggiunta di un FancyBox e pulisce l'input di testo. Nello slice troviamo:

```
const slice = createSlice({
  name: "session",                  // nome dello slice
  initialState,                     // stato iniziale
  reducers: {                       // inseriamo i reducer relativi a questo slice
    addEndpoint(state, action: {payload: string}) {
      state.FancyBoxConfigs.push({endpoint: action.payload, user: null,
                          ssidNetwork: state.currentSSIDNetwork});
    },
    ...
}
```

Come notiamo andiamo a creare uno slice di nome *session* tramite *createSlice*, inizializziamo lo stato iniziale che sarà un oggetto che contiene:

1. *isFilterOn = undefined* : per tenere traccia se l'utente ha accettato di visualizzare solo la lista di FancyBox sotto la rete attuale.
2. *currentSSIDNetwork = undefined* : per tenere traccia a quale connessione Wifi è attualmente in uso sul dispositivo
3. *endpointSelected = undefined* : per tenere traccia quale endpoint è stato selezionato
4. *FancyBoxConfigs = [ ]* : array vuoto di oggetti relativi alla configurazione dei FancyBox, dove andranno le configurazioni che l'utente aggiunge tramite *addEndpoint* del codice precedente.

Alla fine abbiamo i *reducers*, dove andiamo a scrivere come si deve comportare l'applicazione quando uno di questi viene azionato. In questo caso vediamo *addEndpoint*, che prende due argomenti:

1. *state* : lo stato corrente dello slice
2. *action* : l'azione invocata per modificare lo stato dello slice, dove a sua volta contiene *payload* cioè le informazioni utili per la modifica dello stato.

Quando viene azionato questo reducer aggiunge una configurazione di un FancyBox nell'array *FancyBoxConfigs* tramite la funzione *push*.
I dati che rappresentano la singola configurazione sono:

1. *ssidNetwork* : una stringa che dice sotto quale Wifi è stato registrato.
2. *endpoint* : l'endpoint del FancyBox.
3. *user* : cioè le informazioni dell'utente: se autenticato, il nome, il ruolo e il suo token.

Login Page

Dopo che abbiamo visto l'aggiunta del dispositivo edge, l'utente per poter monitorarlo ha bisogno di accedere in modo tale che il FancyBox gli assegna un *token* univoco per poter eseguire le azioni che desidera. La schermata serve per fare l'accesso al FancyBox tramite credenziali conosciute dall'utente poi abbiamo il bottone in alto a sinistra che ci permette di ritornare alla lista dei FancyBox (*Figura 3.4*).

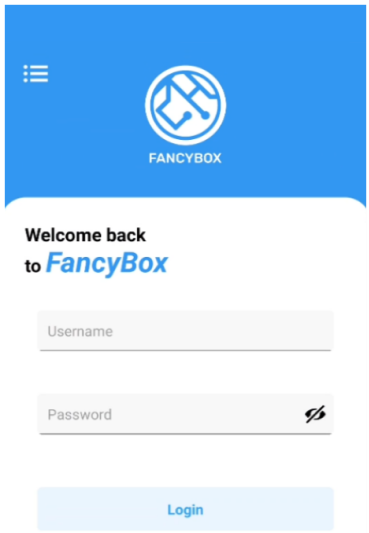

Figura 3.4: Login Page

In questa schermata però utilizziamo il fratello di Redux Toolkit che abbiamo utilizzato nella pagina precedente tramite i reducer, qui usiamo Redux Toolkit Query per fare una richieste HTTP al FancyBox in modo tale che ci dia il *token* da memorizzare nella configurazione attuale del FancyBox dentro l'oggetto *user* che contiene l'attributo *token*.

```
function LoginPageScreen() {
  // ricaviamo l'hook da RTK Query
  const [login, {isLoading}] = actions.core.useLoginMutation();
  ...
  const onLogin = async () => {
    try{
      if( isNotBlank(username) && isNotBlank(password) ) {
        // eseguiamo la chiamata passando come argomenti username e psw
        const res = await login({username, password});
        ...
      }
    } ...
}
```

Il codice che ci interessa è quello di come avviene la chiamata login. Redux Toolkit Query quando va a creare un API, come abbiamo mostrato nel Capitolo 2.4, dove fornisce automaticamente dei hook per eseguire le chiamate HTTP, come *useLoginMutation* nel codice di sopra. Dopo che l'utente avrà messo le credenziali a premuto sul pulsante login, verrà eseguita una funzione in modo asincrona che verifica se i campi degli input sono riempiti e poi esegue la funzione *login* passando come argomenti le credenziali dell'utente. Come si nota andiamo anche ad estrarre dall'hook la variabile *isLoading* in modo tale che se l'utente ha già fatto partire una richiesta di verifica di accesso il bottone di *login* venga bloccato per poi sbloccarsi una volta che abbiamo ricevuto una risposta, se la risposta ha successo positivo allora la prossima schermata a cui verremo reindirizzati sarà la *Dashboard Page*.

Dashboard Page

In questa schermata l'utente riesce a vedere le informazioni relative al carico eseguito dal FancyBox su CPU, RAM e Memoria disponibile (*Figura 3.5*). Partendo dalla parte in alto dello schermo troviamo come prima funzionalità la possibilità di ritornare alla pagina iniziale per la selezione dei FancyBox senza che il dispositivo esegui il logout di accesso invece dalla parte opposta c'è un pulsante per il Logout, dove in questo caso verrà eseguito la disconnessione con il dispositivo e poi rimandati alla *Home Page*. In basso invece abbiamo una barra di navigazione per passare da una schermata all'altra ed è per questo che nel codice fornito nel Capitolo 3.2 nello stack di navigazione lo Stack.Screen di *DetailPage* fa riferimento ad un'altro stack di navigazione, cioè al bottom navigation bar (*Figura 3.5*). In questa schermata la prima chiamata per ricavare le informazioni relative al FancyBox viene eseguita alla costruzione del componente, le successive avvengono tramite un sistema di *polling,* dove ogni cinque secondi viene inviata una chiamata Query, *stats*, sempre definita nel *createApi* nella sezione *endpoint* dove troviamo anche la chiamata del login.

```
const api = createApi({
```

```
  reducerPath: 'core',
  baseQuery: fetchBaseQuery({
    baseUrl: 'http://192.168.10.102:3000'
  }),
  endpoints: (build) => ({
  …
    stats: build.query<Response, undefined>({        // Query
      query: (params) => ({ url: '/api/stats', method: 'GET', params })
  })
});
```

Per poter fornire questi dati ad entrambe le pagine invece di fare due chiamate uguali una sulla pagina di *Dashboard* e di *Detail*, quello che facciamo è creare un *Provider* cioè un componente 'fornitore' che condivide i dati che contiene con tutti i componenti figli.

```
const MainContextProvider= ( {children} : {children: React.ReactNode} )  => {
  const {data} = actions.core.useStatsQuery(undefined, {pollingInterval: 5000});

  return (
    <MainContext.Provider value={{data}}>
      {(data === undefined) ? (
        <MainLoader visible={true} />
      ) : children}
    </MainContext.Provider>
  );
}
```

Creiamo un *Provider* che prende come argomento una props chiamato *children* di tipo React.ReactNode che rappresenta i componenti figlio che saranno inclusi all'interno del MainContext.Provider, in questo caso quello che verrà incluso sara lo Stack di navigazione della bottom navigation bar, in modo tale riusciamo a navigare tra le due pagine. Il *Provider* mostra un loader finchè la prima chiamata per prendere i dati dal FancyBox non ha dato risposta, dopo di che mostra la pagina effettiva *Dashboard* e ogni cinque secondi il *Provider* richiama la richiesta HTTP in modo tale che i dati siano sempre aggiornati.

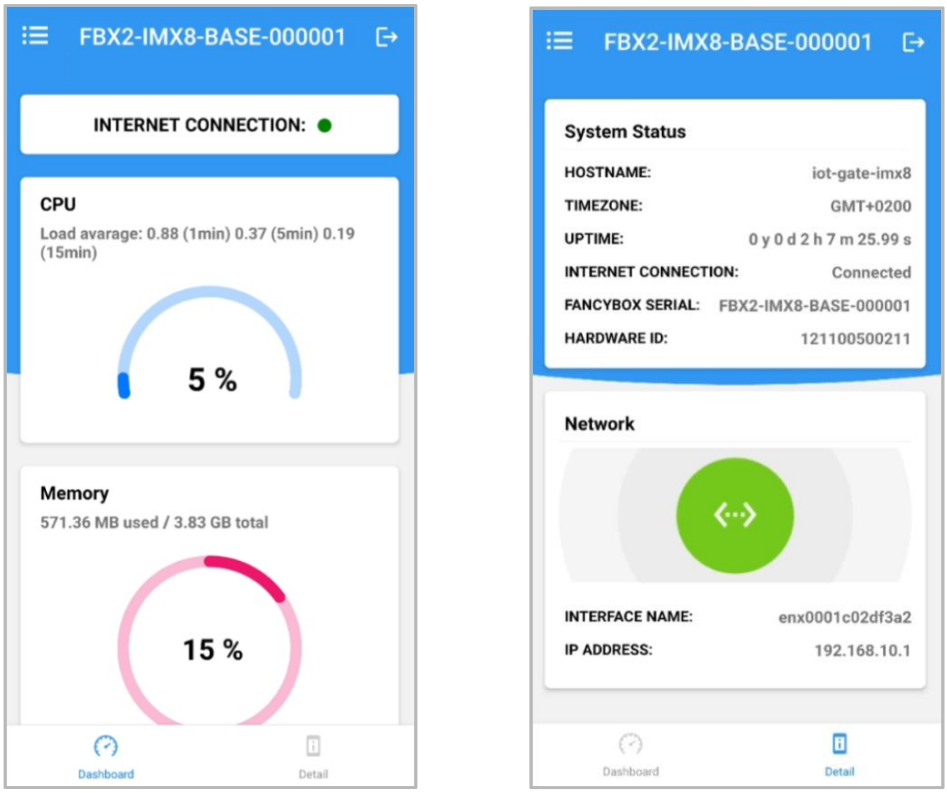

Figura 3.5: Dashboard Page e Detail Page

3.2.2 Ripristino del dispositivo edge

Dopo aver visto le schermate che rappresentano il monitoraggio del FancyBox andiamo a vedere quelle che rappresentano il recovery.
In questa parte dobbiamo precisare che usavamo un FancyBox di test in modo da studiare la comunicazione BLE tra i dispositivi. Questo FancyBox di test veniva utilizzato come *Proxy*, cioè come intermediario tra lo smartphone, il *Central,* e il FancyBox operativo in cui c'erano i dati, cioè il *Peripheral*.
Quello che facevamo era mandare stringhe di testo al FancyBox di test, dove poi era lui a fare le chiamate HTTP al FancyBox 'padre' in modo da recuperare i dati come se fossero dentro direttamente al FancyBox di test e inviarli allo smartphone.
Nella reale implementazione della funzionalità BLE, il codice sviluppato per il FancyBox di test dovrebbe venire implementato nel modulo del *Bluetooth* del FancyBox 'padre', in modo tale che non esegua chiamate HTTP, ma vada a leggere i dati direttamente dagli altri moduli presenti nel dispositivo .

Detto questo l'utente per accedere a questa funzionalità dall'*Home Page* deve premere semplicemente sul bottone del simbolo del Bluetooth che viene mostrato nell'header, poi navigherà sulla prima pagina che è la pagina della scansione dei dispositivi. Come detto in precedenza lo smartphone farà da dispositivo *Central* quindi andremo a fare lo scanning dei dispositivi nelle vicinanze.

Scanning Page

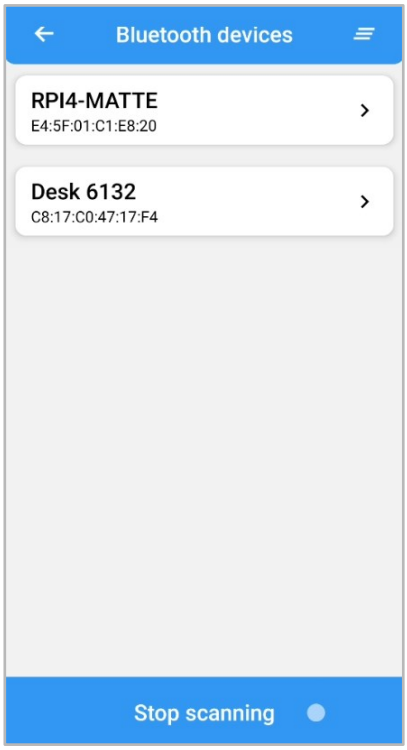

Figura 3.6: ScanningPage

In questa pagina ancora una volta vengono chiesti i permessi di localizzazione per permettere di visualizzare i dispositivi vicini tramite Bluetooth Low Energy, nel caso in cui l'utente precedentemente li avesse rifiutati (*Figura 3.6*). Nell'header troviamo la freccia che ci riporta alla schermata *Home Page* e poi un simbolo di pulizia della lista che va a rimuovere tutti i dispositivi che ha fino al quel momento trovato. Nella parte più bassa troviamo il bottone di avvio della scansione e una volta avviato, per far capire all'utente che il dispositivo sta cercando altri dispositivi nelle vicinanze, viene messo un indicatore chiamato *Pulse* usando la libreria react-native-pulse. Successivamente vengono mostrati i dispositivi che vengono trovati con BLE attivo, come nella figura che troviamo un FancyBox usato per i test con il nome *RPI4-MATTE*.

```
const bleManager = new BleManager();
...
const onScan = () => {
  // mostriamo l'icona Pulse
  setIsScanning(true);
  // avviamo lo scanning
  bleManager.startDeviceScan(null, null, (error, device) => {
    if( error ) …
    // se troviamo un dispositivo
    if( device && device.name ) {
      // andiamo inserire nella lista i device non duplicati
      setAllDevices( prevState: Device[] ) => {
```

```
      if( !isDuplicateDevice(prevState, device) ) {
        return [...prevState, device];
      }
      return prevState;
    }
}
```

Grazie a questo codice una volta che viene eseguito quando l'utente preme su 'Start Scanning' riusciamo a permettere al dispositivo *Central* di trovare i dispositivi nelle vicinanze.

1. *const bleManager = new BleManager() :* viene usato per creare un oggetto di tipo '*bleManager'* che rappresenta il gestore BLE, e verrà usato per interagire con i dispositivi trovati.
2. *bleManager.startDeviceScan():* viene chiamato il metodo per iniziare la scansione dei dispositivi dell'oggetto bleManager, questo accetta tre parametri:
    1. Il primo specifica se vogliamo cercare dispositivi specifici o se conosciamo già il loro UUID
    2. Il secondo specifica che non ci sono opzioni che riguardano in quale modo lo scanning deve avvenire, esempio per un tempo massimo o altro
    3. Nel terzo parametro invece è una funzione callback che viene eseguito ogni volta un dispositivo viene trovato e quello che fa è gestire se ci sono errori, altrimenti verifica che il dispositivo che è stato trovato abbia un nome, se vero allora si aggiunge alla lista tramite *setAllDevice* alla lista dei dispositivi trovati solo se quel dispositivo nella lista non esiste già. Per prendere la lista dei nomi utilizziamo lo *spread operator* che ci permette di creare un nuovo array che contiene tutti gli elementi del precedente stato, *prevState* e aggiunge il dispositivo appena trovato a questa lista, in modo tale che poi nell'UI viene mostrata la lista dei dispositivi trovati.

Per fermare lo scanning bisogna premere su 'Stop Scanning' e questa azionerà la funzione per rimuovere il Pulse e di fermare lo scanning.

Login Page

Successivamente una volta selezionato il FancyBox desiderato, la pagina del login viene montata (*Figura 3.7*) e viene eseguita la connessione tra *Central* e *Peripheral* tramite BLE, in modo tale che i dati che usiamo per le credenziali verranno inviati al FancyBox di test che a sua volta li userà per eseguire una richiesta HTTP come succede nella *Login Page* del Capitolo 3.2.1 e poi ci manda una stringa se siamo autorizzato o meno ad accedere.

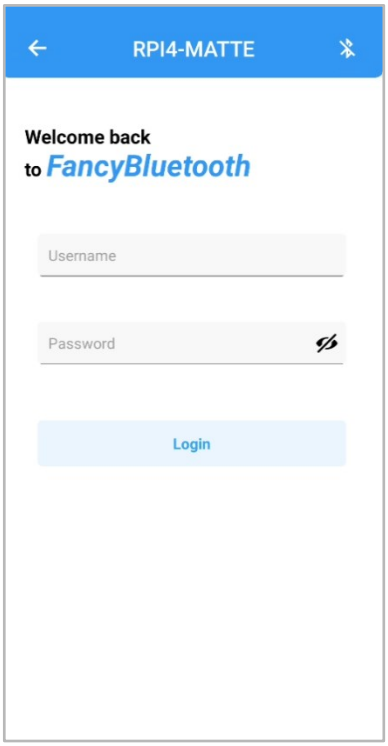

Figura 3.7: Login Bluetooth

Per comprendere come avviene il funzionamento delle comunicazione BLE vediamo del codice per la connessione di un device, di lettura delle informazioni e di scrittura.

```
const connectToDevice = async (deviceId: string) => {
  try{
    const deviceConnetion = await bleManager.connectToDevice( deviceId,
                                                                {requestMTU:256} );
    await deviceConnection.discoverAllServicesAndCharacteristics(); bleManager.stopDeviceScan();
  } catch(err) { ... }
}
```

La funzione *connectToDevice* prende come argomento il dispositivo a cui ci si vuole connettere.
Poi all'interno di un try-catch andiamo ad istanziare *deviceConnection* e tramite il metodo di *bleManager,* cioè *connectToDevice(),* chiederemo una connessione con il dispositivo desiderato indicando come parametri:

1. Il dispositivo con cui ci vogliamo collegare.
2. La grandezza dei messaggi che si possono scambiare. Normalmente questo parametro è di default su 20 bytes, perchè mandare piccole grandezze di informazioni è più affidabile, ma si può aumentare, e noi lo abbiamo fatto in modo tale da semplificare un pò la comunicazione perchè quando andiamo a fare richieste al FancyBox di test andiamo a inviare il *token* dell'utente (50 bytes) e i vari comandi. Perciò per dare un margine di sicurezza, lo abbiamo aumentato a 256 bytes.

Dopo che abbiamo stabilito il collegamento con il *Peripheral* andiamo a fermare la scansione dei dispositivi e possiamo mandargli dei messaggi come in questo caso il messaggio che vogliamo scrivere è quello delle credenziali dell'utente. Il formato dei messaggi che andiamo sono una concatenazione di stringhe: *token* + *azione da eseguire* + *payload* questa stringa poi sul FancyBox viene 'splittata' in modo da prendere il *token* dell'utente, l'azione che deve eseguire che viene usata per identificare tramite uno switch cases nel codice il cosa deve eseguire, e il payload.
Questo tipo di formato e la gestione della stringa è sicuramente migliorabile, ma come detto in precedenza l'obiettivo per ora era solo quello di capire come la comunicazione tra dispositivi BLE avviene.

```
const readCharacteristic = (device?: Device) => {
  device?.readCharacteristicForService( SERVICE_UUID, CHARACTERISTIC_UUID )
    .then((characteristic) => {
       if( characteristic.value )  {
          var response = base64.decode(characteristic.value).split(' ');
          const action = response[0];
          switch(action) {
            case 'SESSION': {
               setIsGranted(response[1]);
               setAccessToken(response[2]);
             }
              ...
              default: break;
           }
     } ...  });
```

Per la lettura di una characteristic andiamo a chiamare il metodo dell'oggetto *device* che viene passato come parametro, e indica a quale device ci siamo collegati. *readCharacteristicForService* prende come argomenti l'UUID del servizio e della caratteristica che vogliamo leggere, questi UUID vengono definiti dall FancyBox di test nel suo codice utilizzando *Bleno*. Una volta che abbiamo il valore della caratteristica lo andiamo a decodificare, perché la comunicazione in BLE avviene in base64, quindi da base64 in testo.
Poi andiamo a 'splittare' la stringa estraendo i singoli dati e infine vengono eseguite i switch case in base all'azione letta.
In questo caso per il login, noi sappiamo che il FancyBox di test ci restituisce un valore booleano, *isGranted*, che poi viene dato come valore alla variabile locale corrispondente tramite *setIsGranted* e sappiamo che restituisce anche il *token*, che ci serve per eseguire le richieste future, che salviamo in una variabile tramite *setAccessToken*.

```
const writeCharacteristic=async( action:string, payload:string | null, token:string,
                                        device?: Device) => {
 const VALUE64 = base64.encode ( `${token ? accessToken:''}$action${payload ? payload:' '} );
 await device?.writeCharacteristicWithResponseForService( SERVICE_UUID,
                                                                  CHARACTERISTIC_UUID, VALUE64);
 readCharacteristic(connectedDevice);
}
```

Quando vogliamo mandare un dato al *Peripheral* usiamo il codice seguente per la scrittura sul

canale di comunicazione BLE. La funzione prende come parametri:

1. *action* : una stringa che serve sempre per differenziare cosa il FancyBox di test deve fare nello switch case.
2. *payload* : i dati che vogliamo inviare, che sappiamo nel caso del login sono *username e password*.
3. *token*: token di accesso.
4. *device* : il device con cui vogliamo comunicare.

Successivamente tramite i parametri che sono stati passati, viene creata un'unica stringa che verrà convertita in base64.
Il FancyBox di test riceve questo messaggio e similmente a come viene fatto nel codice della *readCharacteristic*, i dati vengono divisi tramite *split* per poi essere utilizzati per eseguire una chiamata al FancyBox padre.
Usando l'esempio di login, la stringa che viene costruita ha la seguente struttura:

1. *token*: il token alla prima chiamata al FancyBox di test verrà riempita con una stringa vuota, questo perchè il token al momento l'utente non lo ha ancora, invece nelle successive chiamata verrà sostituito con il token associato all'utente.
2. *action*: definiamo il tipo di azione usando la stringa '*SESSION'* che verrà letto dallo switch case sul FancyBox di test eseguendo le azioni specifiche di quando riceve quel tipo di parola. 'SESSION' viene usato per far riferimento all'azione per il login, altri tipi di azioni avranno una stringa che differente che la identifica, esempio 'STATS' fa riferimento all'azione di leggere le informazioni del FancyBox).
3. *payload*: in questo caso contiene le credenziali, cioè *username e password.* Per altri tipi di azioni potrebbe non essere necessario, infatti in quei la stringa sarà vuota.

Dopo mandiamo il messaggio al FancyBox di test specificando il servizio e la caratteristica che vogliamo scrivere e infine leggiamo la risposta.
Il FancyBox ci risponderà con il *token* di accesso per le operazioni future.
Avendo visto che la comunicazione tramite BLE funziona, siamo riusciti a eseguire il login mandando da smartphone i dati in base64 al FancyBox di test, lui li converte in stringa, estrae l'azione da eseguire, la esegue e infine va a modificare la sua characteristic in modo tale che quando lo smartphone poi esegue il *readCharacteristic* va a leggere il la caratteristica leggendo il *token.* Per le altre azioni si prosegue similmente.

Recovery Page

Una volta loggati, l'utente si troverà in questa pagina dove permetterà di eseguire il reset dell'interfaccia Wifi del FancyBox in modo che sarà di nuovo raggiungibile (*Figura 3.8*). L'utente quando premerà sul pulsante, questa componente andrà a chiamare la *writeCharacteristic* passando solamente il *token* e poi l'*action*, chiamata *MODULES* perché va ad attivare il modulo di Networking e invia come *payload* il dispositivo a quale attualmente è connesso. Sul lato FancyBox di test quello che andrà a fare è verificare che il FancyBox padre abbia certi moduli disponibili tra quale quello del Networking, va poi a mettere di default

alcune proprietà e infine attiva l'hotspot del FancyBox padre in modo che sia raggiungibile tramite Wifi, dopo di che termina la connessione Bluetooth con lo smartphone.

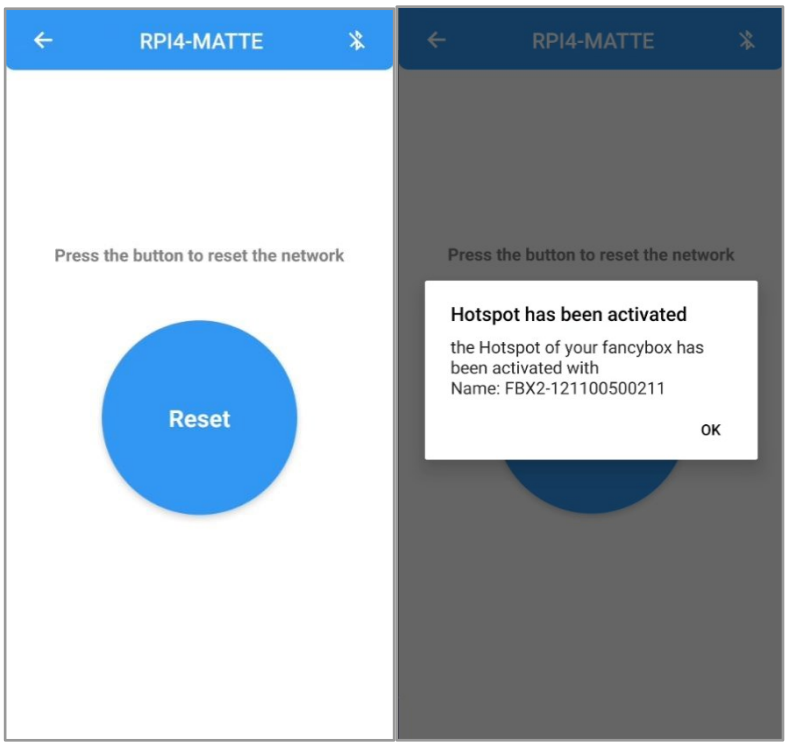

Figura 3.8: Recovery Page

FancyBox

Sul lato del FancyBox di test, viene utilizzata la libreria *Bleno* per la creazione e la gestione di servizi e caratteristiche BLE. Andiamo vedere due *listeners*, che permettono di eseguire codice in base all'evento che si verifica in questo caso, permettono l'avvio e la pubblicazione del servizio e caratteristiche. Per poter utilizzare questi due *listeners* dobbiamo inizializzare in una variabile le funzione che ci permette di utilizzare *Bleno* cosè che possiamo chiamarle tramite un nome di variabile:

const bleno = require('@abandonware/bleno');

```
bleno.on('stateChange', function (state) {
  if ( state === 'poweredOn' )  {
    // name device, service UUID
    bleno.startAdvertising('RPI4-MATTE', ['12ab']);
  } else {
    bleno.stopAdvertising();
  }
});
```

Il *listener* '*on'* va a controllare che quando lo stato del BLE del dispositivo cambia si vada a rendere visibile il dispositivo usando il nome 'RPI4-MATTE' e il UUID del servizio che rende disponibile è il valore '12ab'.

```
bleno.on('advertisingStart', function (error) {
  if (error) {...}
  else {
    bleno.setServices( [
       // Definiamo un servizio con queste proprietà
       new bleno.PrimaryService( {
          // service UUID
          uuid: '12ab',
          characteristics: [
          // Definiamo una caratteristica con queste proprietà
          new bleno.Characteristic ( {
             value: null, // valore della caratteristica
             uuid: '1212', // characteristic UUID
             properties: ['read', 'write'],

          // dentro alle funzioni onReadRequest e
          // dentro onWriteRequest scriviamo poi il codice
          // di cosa il FancyBox deve fare.
          onReadRequest: function (offset, callback) { ... },

          onWriteRequest: async function ( data, offset, withoutResponse, callback ) {
            // dividiamo la stringa
            var data = data.toString("utf-8").split(' ');
            token = data[0];
            action = data[1];
            // creiamo i differenti switch cases
            switch (action) {
               ...
               case ACTION.SESSION:
                  username = data[2];
                  password = data[3];
                  try {
                     const session = await onLogin();
                     this.value = `SESSION ${session.authenticated} ${session.token}`;
                  } catch (err) {...}
                  break;
               ...
            }
            callback( this.RESULT_SUCCESS );  } } ) ] } ) ] );
} } );
```

Ora andiamo controllare quando è sullo stato '*advertisingStart*', con *bleno.setServices* accetta un array di servizi BLE da configurare, noi andiamo a definire un servizio primario tramite *bleno.PrimaryService*, e che richiede alcune proprietà di configurazione tra quali:

1. *uuid:* il nome del servizio che andiamo a pubblicare.
2. *characteristic:* un array di caratteristiche che appartengono a questo servizio.

dentro a questo array andiamo a definire una caratteristica tramite *bleno.Characteristic* che anche questo richiede di fornire delle proprietà tra cui

1. *value*: inizializzato a null, che sarà il valore che verrà letto ogni volta che lo il *Central* esegue il codice *readCharacteristic.*
2. *uuid:* il nome che viene data alla caratteristica.
3. *properties:* proprietà della caratteristica, cioè cosa il *Central* può fare, in questo codice troviamo, che può scrivere e leggere le caratteristiche, ma ci sarebbe anche l'opzione di

notifica, che ogni qualvolta che il *value* cambia stato, viene notificato al *Central*.
sempre all'interno della caratteristica che stiamo definendo, andiamo a dire le istruzioni che deve eseguire quando il *Central* richiede una scrittura o lettura della caratteristica. Quando l'applicazione richiede una lettura, la funzione *onReadRequest* entra in gioco e ritorna semplicemente il valore che attualmente *value* contiene.
Nel caso in cui chiede di scrivere, quindi di mandare informazioni al *Peripheral*, viene eseguita *onWriteRequest*.
Prendiamo come riferimento sempre il login, quello che succedeva sullo smartphone e che dobbiamo mandare le credenziali al FancyBox di test, quei dati venivano messi in una stringa in formato, *token* + *azione* + *payload* (cioè username e password), convertiti in base64 e inviati.
Sul FancyBox la funzione *onWriteRequest* prende questi dati, li converte in formato UTF-8, estrae i dati di *token ed azione* poi nello switch case il codice che viene eseguito è quello nella si basa sulla stringa di action, cioè '*SESSION*': esegui una chiamata HTTP al FancyBox padre come si faceva nella schermata *Login Page,* la differenza qui e che non usiamo RTK Query ma semplicemente le chiamate di Javascript usando la funzione fetch.
Dopo aver ricevuto risposta poi andiamo a mettere come valore del *value* della caratteristica la stringa formata da: *action* + *payload* (username e password) e quando il *Central* legge il valore di *value* del servizio.
Il ragionamento dietro allo scambio di informazioni tra dispositivi BLE è lo stesso solo che i switch cases saranno diversi in base a cosa il dispositivo deve fare, quindi in base alle *actions*.

Conclusioni

In conclusione, il FancyBox Companion è un'applicazione mobile multi-piattaforma, Android e iOS, sviluppata in React Native per affiancare la versione web del software di amministrazione del FancyBox, fornendo la possibilità sia di monitorare che di ripristinare il modulo di Newtorking del FancyBox e può essere inserito all'interno dell'ecosistema progettato dall'azienda Fancy Pixel, aiutando l'azienda a ridurre le richieste di assistenza . Le specifiche che erano state richieste sono state soddisfatte e grazie alla modularità dell'architettura con cui è stato progettato, è possibile espanderla in futuro, aggiungendo nuove funzionalità soprattutto nella parte di recovery grazie al sistema di comunicazione Bluetooth.

Per questo progetto siamo partiti da quello che era un'idea di sviluppo, coltivandola e studiando come si potesse implementare in tutte le sue funzionalità.
Abbiamo iniziato con la creazione dei wireframe per le varie schermate, successivamente arrivando all'implementazione a livello di codice.
Un'altro obiettivo principale era affrontare la sfida più grande: la comunicazione Bluetooth. Avevamo preoccupazioni che non ci fosse un modo per poter cambiare le configurazioni del FancyBox senza una connessione internet, ma alla fine ci siamo riusciti, rendendo l'applicazione un efficace strumento per il ripristino dei dispositivi edge.

Durante questa esperienza il mio bagaglio formativo si è ampliato enormemente, il lavoro svolto mi ha permesso di sviluppare delle abilità trasversali, tra cui la capacità di lavorare in team, collaborare con colleghi di ruoli diversi e mi ha anche fornito competenze tecniche base nel campo di sviluppo di applicazioni mobile cross-platform tramite l'utilizzo di React Native. Infine mi ritengo pienamente soddisfatto del lavoro svolto, ma soprattutto sono felice di come mi abbia migliorato a livello umano.