Politecnico di Torino

Ingegneria Informatica

A.a. 2021/2022

Sessione di laurea Luglio 2022

Guided Cooking: migliorare l'interazione quotidiana con un forno connesso

Relatori:

Luigi De Russis

Fabio Frairia

Luca Squillaci

Candidato:

Marco La Gala

Ringraziamenti

Desidero, innanzitutto, ringraziare tutti coloro che mi hanno aiutato nella realizzazione del progetto e nella stesura di questa tesi di laurea. Ringrazio il mio relatore, il Professore Luigi De Russis, per la sua competenza, disponibilità e rapidità nell'aiutarmi durante il percorso di tesi.

Ringrazio, inoltre, l'azienda Enhancers che mi ha permesso di realizzare questo lavoro di tesi e soprattutto Luca, Debora e Max, i quali, fin dai primi giorni, mi hanno accolto come se fossi parte della loro famiglia.

Vorrei, infine, ringraziare la mia famiglia, che ha sempre creduto in me e mi ha sempre sostenuto, sia moralmente che economicamente, durante questo percorso di studi.

Indice

Elenco delle figure VI

1 Introduzione 1
1.1 Contesto . . . 1
1.2 Obiettivo . . . 3
1.3 Struttura del documento . . . 4

2 Tecnologie utilizzate 5
2.1 Sviluppo Mobile . . . 5
2.1.1 Tipologie di applicazioni . . . 5
2.1.2 React Native . . . 7
2.2 Pattern Architetturali . . . 9
2.2.1 Design UI: Atomic Design . . . 9
2.2.2 Gestione dello stato: useState, Redux, RxJS . . . 11

3 Metodologia adottata 15
3.1 Architettura generale del progetto . . . 15
3.2 Interfaccia utente . . . 17
3.3 Interazione con il back-end . . . 19

4 Interfaccia utente 21
4.1 Progettazione . . . 21
4.1.1 Analisi dei mock-up e del flusso di esecuzione . . . 21
4.1.2 Scomporre l'interfaccia utente in componenti . . . 23
4.2 Implementazione . . . 29
4.2.1 Implementazione di un componente riutilizzabile . . . 29
4.3 Miglioramenti e validazione . . . 33
4.3.1 Ottimizzare le performance di un componente . . . 33
4.3.2 Validazione dell'interfaccia nei casi limite . . . 35

5 Interazione con il back-end e il forno IoT 37
5.1 Progettazione 37
5.2 Implementazione dello scambio dei dati con il back-end 39
5.2.1 Chiamate REST API 39
5.2.2 Memorizzazione dei dati in locale 41
5.2.3 Implementazione dell'offline mode 43
5.3 Controllare il dispositivo IoT 44
5.3.1 Invio dei programmi dall'app IoT 44
5.3.2 Invio dei programmi dall'app mobile 45

6 Risultati ottenuti 47
6.1 Valutazione del team di Q&A testing 47
6.1.1 Metriche di valutazione 49
6.1.2 Analisi dei risultati ottenuti 51

7 Conclusioni 53
7.1 Implementazioni future 55

A Chiamate API 56

B GetGuidedCooking Epica 59

Bibliografia 63

Elenco delle figure

2.1 Tipi di applicazioni mobile - Native, Ibride e Web app. 6
2.2 Tipi di componenti che è possibile utilizzare per creare interfacce utente 8
2.3 Esempio di scomposizione seguendo il pattern Atomic design. 10
2.4 Esempio di albero dei componenti con lo stato condiviso. 12

3.1 Architettura 16
3.2 Forno della casa produttrice 16
3.3 Esempio di board di Miro. 18

4.1 Board Miro con il flusso del *Guided Cooking* da implementare nell'app IoT. 23
4.2 Flusso di esecuzione da implementare nell'applicazione mobile per selezionare dal menu il *Guided Cooking*. 24
4.3 Flusso di esecuzione da implementare nell'applicazione mobile per scegliere la categoria e la sotto-categoria. 25
4.4 Flusso di esecuzione da implementare nell'applicazione mobile per modificare, schedulare e avviare il programma selezionato. 26
4.5 Esempio di UI scomposta in una gerarchia di componenti. 27
4.6 Scomposizione dello stepper del Guided Cooking. 28

6.1 Esempio di un ticket aperto dal team di testing. 50
6.2 Bug aperti dal team dopo il testing del *Guided Cooking*. 51
6.3 Risultati dei test effettuati. 52

Capitolo 1

Introduzione

1.1 Contesto

Nella società odierna la tecnologia gioca un ruolo importante; molte delle azioni che svolgiamo quotidianamente sarebbero completamente diverse senza di essa. Si può notare, infatti, che dai primi anni del XXI secolo è passata da essere impiegata solo in ambiente aziendale o istituzionale ad essere utilizzata nelle case delle persone.

La velocità di adozione e accettazione delle nuove tecnologie è di anno in anno sempre più veloce. Infatti è diventato molto comune avere dispositivi al polso che monitorano continuamente parametri vitali durante l'arco della giornata, mentre, fino a pochi anni prima, gli *smartwatch* erano stati aspramente criticati perché considerati troppo invasivi.

Questo trend continua imperterrito e uno dei campi in forte crescita è quello dello *Smart Home* che consiste nel rendere "intelligenti" i dispositivi presenti nelle abitazioni come elettrodomestici, lampadine e videocamere di sorveglianza. Più precisamente con *Smart Home* o "casa intelligente" indichiamo la possibilità di controllare e monitorare in automatico oppure da remoto impianti o dispositivi di un'abitazione [1].

Questo campo può essere considerato come l'evoluzione tecnologica della domotica perché, se quest'ultima prevede che i dispositivi siano collegati fisicamente tra di loro, nella prima la comunicazione avviene utilizzando esclusivamente Internet. Inoltre, lo *Smart Home* fonda le sue radici su un concetto più ampio chiamato *Internet of Things* (IoT), il quale fornisce ad oggetti comuni l'accesso ad Internet e, quindi, aumenta le funzionalità che è possibile implementare su di essi, come la possibilità di essere monitorati e controllati a distanza.

Anche grandi aziende come Apple o Google hanno iniziato ad aggiungere queste funzionalità ai loro sistemi operativi e assistenti virtuali come *HomePod* e *Google Home* per fornire all'utente la possibilità di controllare i loro dispositivi intelligenti

in ogni momento della giornata con qualunque device. Apple, ad esempio, riserva molto spazio per mostrare le funzionalità di *Smart Home* durante l'evento annuale di presentazione delle nuove versioni dei sistemi operativi.

Tutto questo interesse da parte di grandi e medie aziende è dovuto al fatto che gli utenti, essendo abituati ad avere dispositivi tecnologici in tutti gli aspetti della loro vita, li hanno introdotti anche nelle abitazioni.

Come emerge dal rapporto "Osservatori Digital Innovation della School of Management del Politecnico di Milano", il mercato in Italia è in forte crescita con un incremento del 29% nel 2021 rispetto al 2020 ed ha raggiunto una quota di 650 milioni di euro. [2]. Notizia ancora più interessante se si considera che la crescita sarebbe stata del 45% se non fosse per la carenza di semiconduttori e materie prime che ha caratterizzato l'anno precedente.

Il principale motivo che ha accelerato l'adozione da parte degli utenti è la possibilità di monitorare e lanciare comandi da remoto. La comodità di accendere i riscaldamenti mezz'ora prima di arrivare a casa, oppure di avviare il robot aspirapolvere mentre si è a lavoro è innegabile. Dalle statistiche, infatti, emerge che il 21% dei dispositivi venduti riguarda il mercato degli elettrodomestici, seguito da smart speaker (20%) e soluzioni per la sicurezza (19%) come telecamere di videosorveglianza [2].

L'esigenza delle aziende che sviluppano sistemi per la "casa intelligente" è quella di realizzare applicazioni che permettano in modo semplice, intuitivo e veloce di monitorare e comandare a distanza i vari dispositivi. La caratteristica più importante che permette di creare un'infrastruttura di successo è l'ecosistema dei prodotti. L'utente, infatti, vuole poter monitorare più dispositivi possibili con una sola applicazione e, ad oggi, non esiste uno standard ampiamente utilizzato che permetta di controllarli indipendentemente dalla loro marca. Per questo motivo, gli utenti sono molto restii ad integrare all'interno della loro abitazione prodotti di aziende diverse, perché comporterebbe non poter più utilizzare una sola applicazione per gestire tutto l'ecosistema.

Da questo importante aspetto derivano due principali effetti collaterali che si devono considerare quando si sviluppa un'architettura del genere. Il primo è che gli utenti devono essere continuamente motivati ad acquistare altri dispositivi in modo da rendere più difficile e, di conseguenza, meno probabile la conversione ad una azienda concorrente. Di solito viene attuato tramite l'aggiunta costante di nuove funzionalità e il supporto a nuove categorie di mercato per coprire a 360 gradi l'abitazione contenendo il più possibile i costi. Per questo motivo il secondo effetto collaterale è che l'applicativo risulta estremamente complesso e ricco di funzionalità comportando ingenti investimenti per evitare un calo delle prestazioni e per garantire un'elevata affidabilità sia dell'applicazione stessa sia dei dispositivi IoT che gestisce in modo da mantenere alto il grado di soddisfazione

degli utilizzatori.

Il lavoro della tesi è stato svolto all'interno di una piattaforma IoT di un noto produttore di elettrodomestici. Da qualche anno, infatti, l'azienda ha incentivato e aumentato gli investimenti nel campo *Smart Home* e ha deciso di raccogliere in un'unica applicazione tutti i suoi dispositivi intelligenti. In questo modo l'utente avrà la possibilità di controllarli e monitorarli con semplicità e potrà avere una visione d'insieme sui dispositivi attualmente in funzione e sui rispettivi consumi energetici.

1.2 Obiettivo

A seguito di analisi di mercato periodiche, l'azienda ha riscontrato una problematica di usabilità in alcuni elettrodomestici, in particolare quelli della tipologia "forno". Questi dispositivi, infatti, hanno moltissime configurazioni perché devono supportare la cottura di pietanze complesse, ma nell'utilizzo giornaliero, come pranzi veloci o cene in famiglia, vengono utilizzati principalmente per pasti semplici e rapidi.

Nei casi sopracitati l'utente utilizzerà solo una cerchia ristretta di funzionalità e si è notato che configurare il forno per questa tipologia di pasti risulta essere troppo complesso e dispendioso in termini di tempo rispetto ai comuni elettrodomestici sul mercato. Tale problematica, infatti, comporta un abbassamento del grado di soddisfazione degli utenti che andrà a compromettere i loro ulteriori acquisti verso altri dispositivi della stessa casa produttrice.

Il seguente lavoro di tesi ha come obiettivo quello di semplificare e velocizzare l'interazione degli utenti con i forni della casa produttrice attraverso la definizione e l'implementazione di una nuova modalità di utilizzo dell'elettrodomestico, chiamata *Guided Cooking* (o *cottura guidata*, in italiano).

La soluzione mira a risolvere i problemi riscontati e dovrà essere sviluppata direttamente sia sul forno IoT sia sull'applicazione mobile che gestisce tutti i dispositivi intelligenti dell'azienda, in modo da consentire il suo utilizzo non solo in presenza, ma anche da remoto e in maniera uniforme.

Il *Guided Cooking* permetterà di impostare tutti i parametri della cottura in modo automatico in base alla pietanza scelta. Gli utenti, infatti, dopo aver selezionato una macro-categoria come *Carne*, *Pesce*, *Antipasti*, etc., e una relativa sotto-categoria, otterranno un programma suggerito, cioè un insieme di impostazioni come temperatura, preriscaldamento del forno, tempo di cottura, accessori da utilizzare, etc., che consentiranno loro un'ottima riuscita del piatto.

In questo modo l'utilizzatore non dovrà configurare le impostazioni manualmente, perché avrà una configurazione sulla base della pietanza scelta, potendo così sia

velocizzare la procedura, sia evitare possibili errori che potrebbero rovinare il piatto, come la scelta di una temperatura troppa alta o il programma di cottura sbagliato.

Dato l'obiettivo appena descritto e il contesto evidenziato nella Sezione 1.1, si auspica che la nuova modalità di utilizzo possa migliorare l'esperienza utente e velocizzare l'utilizzo giornaliero dell'elettrodomestico. I clienti, infatti, oltre a risparmiare tempo, otterranno la migliore configurazione per la tipologia della pietanza selezionata permettendo, quindi, anche a quelli meno esperti di avere una cottura perfetta.

Inoltre, come obiettivi secondari, si dovranno cercare soluzioni ed accortezze che permettano alla nuova modalità di utilizzo di non inficiare troppo sulle prestazioni già precarie dell'applicazione e l'interfaccia utente dovrà essere semplice e facile da usare a causa dell'utenza di riferimento, la quale non è omogenea ed è caratterizzata da diverse fasce d'età e nazionalità.

1.3 Struttura del documento

In questo primo capitolo ***Introduzione*** si è parlato del contesto generale e degli obiettivi del lavoro di tesi, concentrandosi sull'inquadramento della situazione attuale dello *Smart Home* nel nostro Paese.

Nel secondo capitolo ***Tecnologie utilizzate***, si effettuerà una descrizione generale delle tecnologie e dei pattern utilizzati come *Atomic Design*, *React Native*, *Redux* e *RxJS* che permetteranno di porre le basi agli argomenti trattati nei capitoli successivi.

Nel terzo capitolo ***Metodologia adottata***, si descriverà l'architettura generale del progetto e l'approccio utilizzato per raggiungere gli obiettivi prefissati, soprattutto in relazione alle performance e alla compatibilità di dispositivi e lingue diverse.

Nel quarto capitolo ***Interfaccia utente*** e nel quinto ***Interazione con il back-end e il forno IoT*** si descriverà l'implementazione delle funzionalità partendo dall'interfaccia utente, e quindi dalla creazione e modifica di componenti React condivisi, passando per lo scambio dei dati con il back-end tramite chiamate REST, fino ad arrivare a trattare l'invio dei programmi al dispositivo IoT.

Nel sesto capitolo ***Risultati ottenuti*** si andranno ad analizzare i risultati ottenuti in relazione agli obiettivi iniziali, tramite metriche oggettive prodotte dal team di Q&A Testing durante il testing dell'applicazione sui vari dispositivi.

Infine, nel settimo capitolo ***Conclusioni*** si riassumerà il lavoro svolto trattando, inoltre, le possibili migliorie ed implementazioni future.

Capitolo 2

Tecnologie utilizzate

Il seguente capitolo ha lo scopo di introdurre e spiegare alcuni concetti generali necessari per la comprensione di quelli successivi. Nella Sezione 2.1 si illustrerà lo stato dell'arte dello sviluppo mobile concentrandosi sulle varie tipologie di applicazioni e sul framework React Native utilizzato nel lavoro di tesi. Nella Sezione 2.2, invece, si andranno a descrivere i pattern utilizzati per la progettazione dell'interfaccia utente modulare e riutilizzabile e per la gestione e memorizzazione delle informazioni di stato.

2.1 Sviluppo Mobile

Le alternative che oggigiorno uno sviluppatore ha per la realizzazione di un'applicazione mobile sono molteplici e possono essere suddivise in tre macro-categorie: *app native*, *app ibride* e *web app* (Figura 2.1). I fattori principali che permettono di scegliere tra una delle tre sono l'esigenza aziendale, il target di riferimento ma, soprattutto, le risorse economiche e il tempo a disposizione. Per questo motivo non sempre una soluzione è migliore delle altre, ma è importante conoscere tutte le possibili alternative, con i loro vantaggi e svantaggi, per effettuare una scelta consapevole.

2.1.1 Tipologie di applicazioni

Le **app native** sono una tipologia di applicazioni sviluppate esclusivamente per una determinata piattaforma e, per realizzarle, si utilizzano i tool e i linguaggi ufficiali forniti dai creatori dei sistemi operativi. Per esempio, per iOS sono sviluppate in Swift utilizzando XCode, mentre per Android in Kotlin con Android Studio. Di conseguenza, visto che sono specifiche per il sistema operativo di riferimento, esse si adattano perfettamente alla piattaforma per la quale sono state create e possono

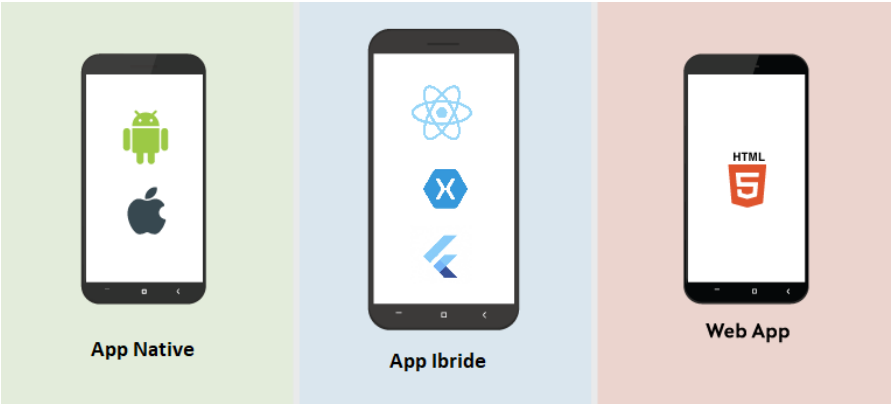

Figura 2.1: Tipi di applicazioni mobile - Native, Ibride e Web app.

sfruttare moltissime funzionalità del dispositivo, come la fotocamera, il GPS, la localizzazione o le notifiche push, le quali permettono di inviare avvisi agli utenti, per esempio a fini promozionali.

Inoltre, l'app risulta più leggera e veloce permettendo all'utente un'esperienza migliore; può anche essere utilizzata in modalità offline, a differenza delle web app, e, quindi, non necessita di una connessione ad Internet costante.

Lo svantaggio dell'essere legati ad una piattaforma specifica è che gli sviluppatori devono programmare l'app indipendentemente per ognuna delle piattaforme. Questo comporta un notevole aumento dei costi e del team di persone necessarie per svilupparla.

Le **web app**, invece, essendo applicazioni simili ad un sito web e sviluppate con i comuni linguaggi di programmazione come *HTML*, *CSS* e *JavaScript*, sono indipendenti dal tipo di sistema operativo. Queste tipologie di app sono caricate da un web server ed eseguite direttamente sul browser e, quindi, non hanno bisogno di nessuna installazione e non andranno ad inficiare la memoria del telefono. Inoltre, hanno il vantaggio che le modifiche al codice non devono essere approvate perché non sono pubblicate sugli store ufficiali del sistema operativo come Apple store o play store. D'altro canto, invece, può risultare difficile incrementare il numero di utenti che la utilizzano non avendo la possibilità di essere sponsorizzata.

L'anello mancante tra le due tipologie di app appena descritte sono le **app ibride**, chiamate anche multipiattaforma. Rispetto alle app native sono più veloci da realizzare e meno dispendiose in termini economici; inoltre, viene prodotta una sola versione, indipendentemente dal numero di piattaforme sulle quali si desidera essere presenti. Infine, le app ibride rappresentano ottime soluzioni web-based, che si integrano perfettamente con l'hardware e il software dei vari dispositivi e,

soprattutto per i costi ridotti, sono l'alternativa migliore per chi vuole affacciarsi al mondo delle app, ma non ha grandi budget da investire.

Tuttavia, tutti questi benefici hanno un costo: le performance di queste app sono inferiori a quelle delle app native, dal momento che non sono legate alla piattaforma. Esistono in commercio moltissimi framework che permettono di sviluppare app ibride, come *Xamarin* della Microsoft, *React Native* di Facebook e *Flutter* di Google, i quali forniscono degli strumenti per semplificare lo sviluppo multipiattaforma.

2.1.2 React Native

React Native è un framework *JavaScript* ideato da Facebook nel 2015 che permette di creare applicazioni ibride per iOS e Android. Il vantaggio principale che ha decretato la sua popolarità è che si basa su un linguaggio molto conosciuto dagli sviluppatori e, in più, fonda le sue radici su un framework ampiamente utilizzato per lo sviluppo web chiamato *ReactJS*.

In questo modo è possibile creare applicazioni mobile in *JavaScript* o *TypeScript* con il vantaggio di non dover imparare un altro linguaggio di programmazione e di non dover sviluppare separatamente l'applicazione per ognuno dei sistemi operativi in commercio. Inoltre si ha solo un team di sviluppo e si possono abbassare drasticamente i costi della creazione e del mantenimento dell'applicativo.

Ponendo le sue basi su React, sono presenti gli stessi concetti come la divisione dell'interfaccia utente in componenti e l'utilizzo di particolari funzioni chiamate *hooks* che vengono eseguite in momenti precisi del lifecycle dell'app. La differenza sostanziale è che, invece di utilizzare le API del browser per manipolare il DOM, utilizza le API specifiche della piattaforma mobile.

React Native, infatti, mette a disposizione alcuni componenti chiamati **Core components** come View, *Text*, *Button*, etc, i quali sono alla base della creazione di interfacce utenti. In fase di esecuzione saranno sostituiti dalla loro rispettiva implementazione nativa chiamata **Native components** in base al sistema operativo per cui l'app viene eseguita. Per esempio, $< Text >$ permette di mostrare del testo sullo schermo; questo verrà convertito in $< TextView >$ se siamo in ambiente Android o in $< UITextView >$ se siamo in iOS [3]. Inoltre, è possibile creare componenti nativi (**Your Native Components**) o utilizzare librerie di terze parti (**Community components**) che consentono di supportare funzionalità non ancora implementate direttamente nel framework come mappe, caroselli, gesture, etc. (Figura 2.2).

La filosofia di React, che sarà approfondita in dettaglio nella Sezione 4.1.2, si basa proprio sul fatto che un'interfaccia utente può essere scomposta in componenti custom sempre più piccoli che utilizzano a loro volta o altri componenti o i *core*

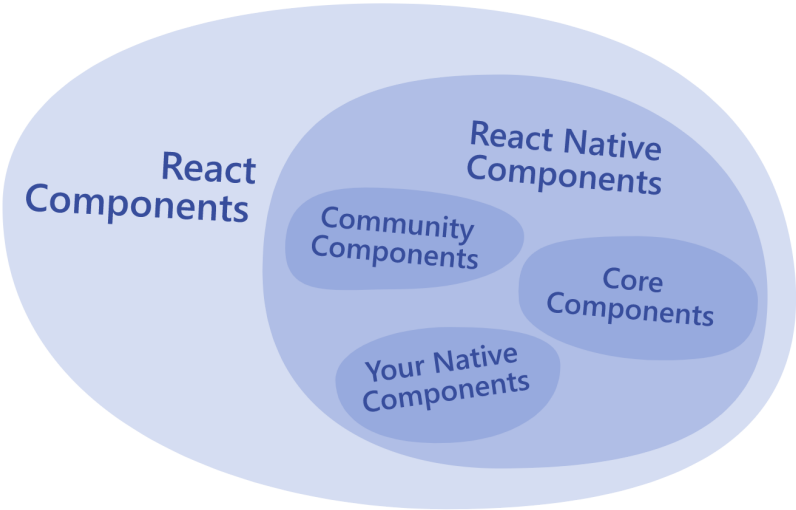

Figura 2.2: Tipi di componenti che è possibile utilizzare per creare interfacce utente

components. In questo modo è possibile garantire importanti proprietà del codice come *encapsulation*, *reusability* e *composability*.

Come si può vedere nel Listato 2.1, dove viene mostrato un componente che visualizza un messaggio di benvenuto all'utente, questo non è altro che una funzione TypeScript che ritorna un elemento della UI attraverso una particolare sintassi chiamata *JSX*. Questa permette di incorporare la logica dei componenti con l'interfaccia utente. Infatti, nell'esempio mostrato, il componente ritorna un oggetto *Text* che racchiude una stringa con una variabile chiamata *name* passata come argomento al componente. La variabile, che viene chiamata con il nome di *props* (proprietà) in React, sarà sostituita a runtime con il valore effettivo, ossia il nome dell'utente che ha effettuato l'accesso.

```
1 const WelcomeMessage = ({name}: {name: string}) => {
2   return (
3     <Text>Bentornato, {name}!</Text>
4   );
5 }
```

Listato 2.1: React: creazione di un componente con una proprietà.

2.2 Pattern Architetturali

2.2.1 Design pattern: Atomic Design

Atomic Design è un pattern architetturale creato da Brad Frost nel 2013 ed è molto utilizzato per progettare e mantenere l'interfaccia utente di grandi applicazioni [4]. Aiuta, infatti, gli sviluppatori e i designer a creare ed implementare un *design system* consistente tra le varie schermate, riutilizzabile e facilmente mantenibile nel tempo attraverso un approccio metodologico.

Prende il nome dal mondo della fisica e, in particolare, dal comportamento degli atomi in natura. Gli elementi atomici, infatti, sono considerati indivisibili e si combinano tra di loro per formare molecole che a loro volta possono raggrupparsi per creare organismi più complessi, i quali alla fine formano la materia di tutto l'universo.

Portando la metafora dal mondo naturale al mondo informatico e, in particolare, a quello della progettazione di interfacce utente, possiamo notare una certa somiglianza. Infatti, queste ultime possono essere considerate come una composizione di elementi più piccoli chiamati componenti. Il vantaggio di questo pattern è proprio legato al fatto che permette di pensare alle interfacce come un insieme di componenti che, in base alla loro grandezza, vengono definiti come atomi, molecole, organismi, template e pagine.

Come è possibile vedere dalla Figura 2.3 che mostra come applicare il pattern ad un post di Instagram, gli **atomi** sono componenti indivisibili, ossia che a loro volta non possono essere scomposti in ulteriori sotto-componenti, e sono identificati come gli elementi base che costituiscono le interfacce. Si può, infatti, notare in questa sezione elementi come pulsanti, titoli, sottotitoli, icone, campi di input, etc. Questi componenti base hanno, al loro interno, degli stati che ne modificano l'aspetto estetico come la grandezza del font, il colore, il nome dell'icona. Per esempio, un bottone può avere degli stati come "disabilitato" quando non potrà essere premuto, "over" quando il mouse dell'utente è posizionato sopra di esso, "loading" quando l'utente ha lanciato l'azione associata e questa non è ancora terminata. Per ognuno di questi, il componente avrà uno stile diverso.

Le **molecole**, invece, sono composizioni di uno o più atomi, come per esempio la *top bar* o la *bottom bar* in Figura 2.3 e sono i primi componenti con un'utilità vera e propria. Risultano, di conseguenza, più complessi perché, a differenza degli atomi i quali non hanno né azioni né funzioni, iniziano ad avere al loro interno anche la logica che ne permette un corretto funzionamento. Ad esempio, la funzione che verrà eseguita al click di un bottone non sarà implementata a livello dell'atomo, ma ad un livello superiore, come quello della molecola, in modo da permettere di

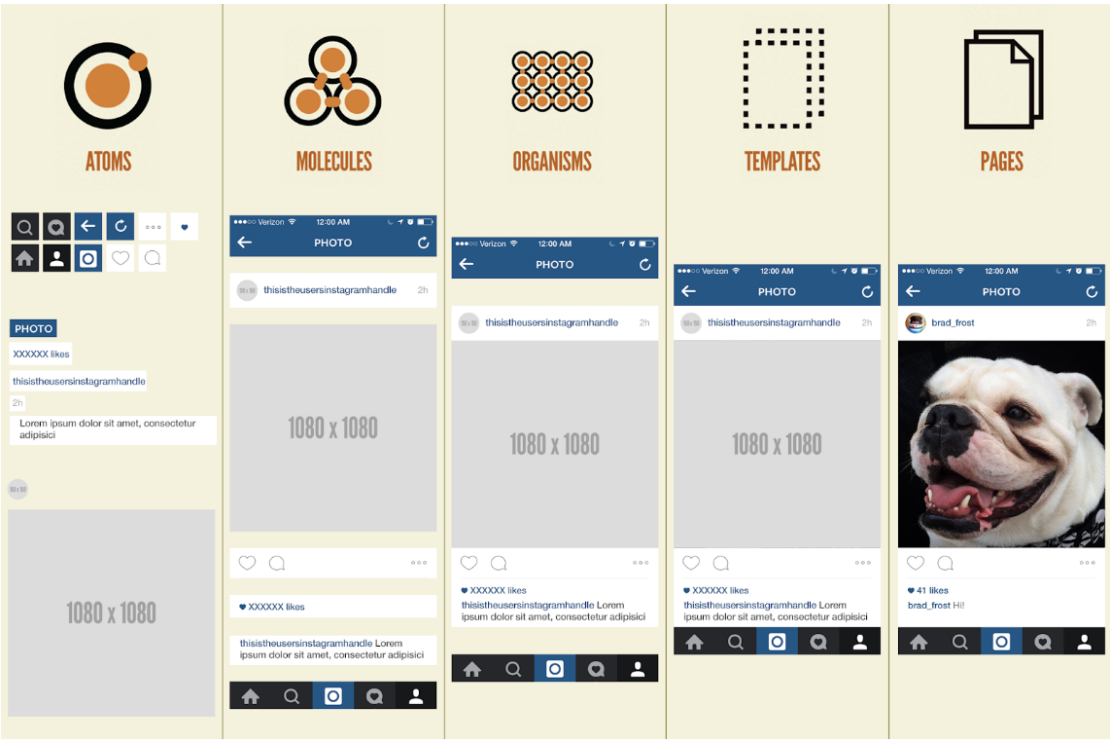

Figura 2.3: Esempio di scomposizione seguendo il pattern Atomic design.

riutilizzare il componente atomo anche in altri contesti.

Più molecole formano gli **organismi**. A questo livello troviamo componenti che rappresentano intere sezioni dell'interfaccia come *footer*, un post di un blog, *header*, etc. Questi, però, devo essere ancora indipendenti e riutilizzabili in contesti diversi.

Con i template e con le pagine, invece, non andiamo più a creare o aggregare componenti per crearne di più complessi, ma iniziamo ad impostare la struttura ed il contesto dell'interfaccia. Il **template**, infatti, permette di creare delle relazioni tra più organismi attraverso la loro posizione nella schermata. Per esempio, è possibile definire un template generico della pagina del singolo post definendo che la pagina sarà composta da due organismi, uno in alto e uno in basso, che rappresentano rispettivamente l'*header* e la *bottom bar*, mentre il restante dello spazio sarà implementato come una scrollview con all'interno il componente del post.

Con le **pagine**, infine, si specializzano i template con i dati reali e quindi passeremo al template tutti i dati dinamici ritornati dalla chiamata API e questo livello, a differenza di tutti gli altri, avrà componenti specializzati. Le pagine, quindi,

possono essere definite come le istanze dei template e saranno connesse tra di loro per poter permettere ad un utente di navigare nel sito.

In conclusione, con questo pattern, è possibile modificare le pagine ed i template utilizzando semplicemente organismi diversi o cambiando la posizione di quelli esistenti senza preoccuparsi del funzionamento degli organismi in sé. Questo ci permette di dividere da una parte lo sviluppo dei componenti e l'implementazione della loro logica, dall'altra lo sviluppo e la progettazione della interfaccia come insieme di organismi. Inoltre, in modo semplice e veloce, è possibile creare diversi layout con lo stesso insieme di organismi, in modo da validarli con un insieme di utenti e capire quello che risulta più efficace.

2.2.2 Gestione dello stato: useState, Redux, RxJS

Lo stato è un'altra componente importante da considerare perché rappresenta il contesto di un'applicazione, ossia tutte le informazioni utili che ne permettono il suo corretto funzionamento.

Se pensiamo ad un app di ricette, per esempio, si può ipotizzare che all'interno del suo contesto siano presenti, come minimo, la lista delle ricette con gli ingredienti e gli step necessari per attuarle.

React ha una sua filosofia per gestire lo stato. Come trattato nella Sezione 2.1.2, il framework è incentrato sulla scomposizione dell'interfaccia in componenti e sulla riutilizzabilità degli stessi. Per questo motivo, lo stato non viene memorizzato in un punto specifico dell'app, come per esempio la *root*, ma è frammentato perché risiede all'interno dei singoli componenti.

Ogni componente, infatti, si occuperà di renderizzare la sua interfaccia grafica e, se necessario, di memorizzare e gestire le informazioni di stato di cui necessita. La libreria mette a disposizione un *hook* specifico chiamato *useState*, il quale ritorna un array con all'interno un *getter* e un *setter* che saranno utilizzati rispettivamente per leggere e aggiornare lo stato. In questo modo si vuole sottolineare che è il componente stesso ad essere l'unico responsabile dello stato che ha inizializzato e, a loro volta, si occuperà di tenerlo aggiornato correttamente.

Da queste premesse seguono due principali considerazioni. La prima è che lo stato all'interno di un componente non risulta di default accessibile agli altri sia in scrittura che in lettura, a meno che non sarà il componente stesso a renderlo disponibile ai suoi figli passandolo come proprietà. Di conseguenza, se si deve condividerlo con componenti che non sono uno figlio dell'altro, è necessario risalire la gerarchia fino a trovare un antenato in comune e posizionare lo stato su di esso. Questo procedimento viene definito nella documentazione con l'espressione *state lift up*, ossia sollevare lo stato posizionandolo più in alto nella gerarchia.

Quindi, se si vuole evitare di ristrutturare il codice causando una perdita di risorse e di tempo, è necessario progettare l'applicazione e il suo stato in anticipo. Per fare ciò si consiglia di rappresentare la sua struttura come un albero di componenti. Come è possibile vedere dalla Figura 2.4, troviamo in cima il componente *root*, il quale sarà eseguito all'avvio dell'app e al livello sottostante sono presenti tanti rami quante sono le pagine che abbiamo definito, le quali a sua volta sono una gerarchia di componenti.

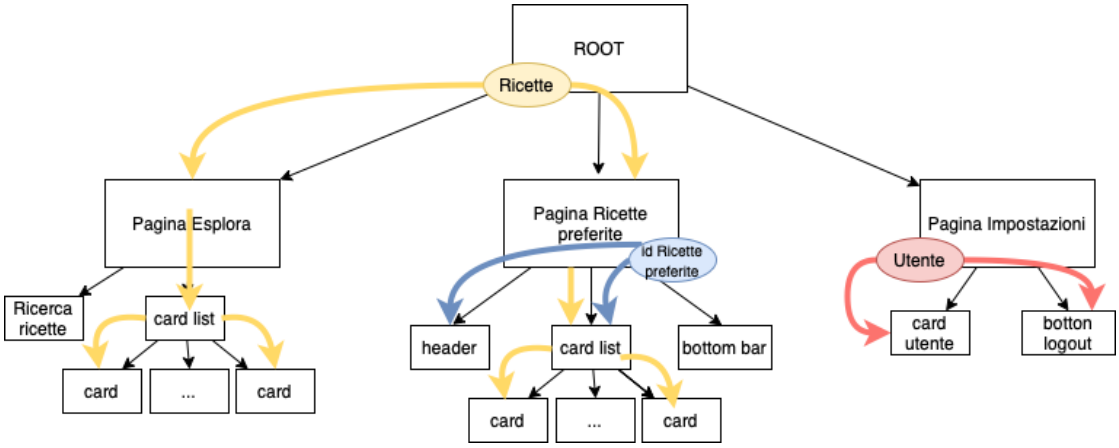

Figura 2.4: Esempio di albero dei componenti con lo stato condiviso.

In questo modo siamo in grado di elencare i componenti di cui abbiamo bisogno ed evidenziare la loro gerarchia. Inoltre, si potrà posizionare lo stato all'interno dell'albero rappresentando dove esso risiede e, tramite delle frecce, evidenziare quali dei componenti figli ne hanno accesso. Risulta molto evidente se uno stato è stato inserito nella posizione sbagliata ed è possibile trovare facilmente quella corretta identificando il componente padre in comune. Ciò non comporterà nessun costo perché siamo in una fase precedente allo sviluppo.

Questa metodologia, inoltre, risolve un altro problema presente nella tipologia di applicazioni dove non si ha uno stato centrale, che è quello di averlo talmente frammentato da non aver più chiaro quali informazioni contiene. Il diagramma ci permette di vederle a colpo d'occhio ed è possibile individuare l'esistenza di stati duplicati, i quali devono essere evitati perché introducono nell'app bug a causa della difficoltà di tenerli aggiornati e sincronizzati tra di loro.

La seconda considerazione da effettuare è che in grandi applicazioni l'albero può diventare molto complesso e profondo e, di conseguenza, la maggior parte delle proprietà dei componenti in alto nella gerarchia sono state inserite esclusivamente per poter esser passate ai componenti sottostanti. Questo aspetto viene chiamato

metaforicamente *props drilling* e comporta il dover scrivere molto codice e, soprattutto, aumenta la complessità dei componenti senza l'aggiunta di nessun beneficio.

I due problemi evidenziati in precedenza aumentano esponenzialmente con la grandezza dell'applicazione. Per questo motivo, si consiglia di utilizzare una libreria che ne faciliti la gestione dello stato come Redux. Quest'ultima, infatti, permette di memorizzare lo stato globale dell'app in un unico oggetto e risolve il problema del *props drilling* evidenziato in precedenza permettendo ad ogni componente, qualunque sia la sua profondità nell'albero, di chiedere e modificare direttamente lo stato senza che questo venga passato come proprietà dai componenti più in alto.
Lo svantaggio dell'introduzione di questa libreria è quello di dover scrivere molto più codice per modificare lo stato dell'app ma, a sua volta, ci permette di avere una gestione più controllata e chiara di quali sono le informazioni memorizzate al suo interno. I creatori di Redux, da qualche anno, consigliano inoltre di utilizzare insieme alla libreria anche un toolkit chiamato Redux toolkit, il quale fornisce un'astrazione maggiore e permette di utilizzare la libreria principale con estrema facilità.

Inizialmente è necessario creare uno *store*, il quale non è altro che il contenitore dello stato dell'app. Lo stato al suo interno può avere moltissime informazioni anche non omogenee tra di loro ed è consigliabile dividerle e categorizzarle in *slice*, i quali, essendo indipendenti tra di loro, permettono una gestione compartizzata dello stato. Ritornando all'esempio iniziale si potrebbe creare uno slice con le informazioni degli utenti (*userSlice*) e un altro con le ricette (*recipesSlice*) come mostrato nel Listato 2.2. Uno stato può essere creato utilizzando la funzione *createSlice* ed avrà un nome che lo identifica univocamente, uno stato iniziale che sarà quello all'avvio dell'app e un reducers il quale elencherà le azioni che è possibile fare sullo slice e, per ogni azione, descriverà come deve essere modificato lo stato. Per esempio, si ha un'azione chiamata *markFavorite*, la quale permette di aggiungere la ricetta alla lista di quelle preferite dall'utente.

```
1  const store = configureStore({
2    reducer: {
3      user: userSlice,
4      recipes: recipesSlice,
5    },
6  })
7
8  export const recipesSlice = createSlice({
9    name: 'recipes',
10   initialState: {
11     recipes: [],
12     favorites: []
13   },
```

```
14   reducers: {
15     markFavorite: (state, action) => {
16       state.favorites = {
17         ...state.favorites,
18         ...action.payload,
19       };
20     }  } })
```

Listato 2.2: Redux: esempio d'uso di *createStore* e *createSlice*.

Una volta configurati lo stato e gli slice all'interno del componente è possibile direttamente richiedere una porzione dello stato tramite un *hook* chiamato *useSelector* oppure eseguire un'azione che ne modificherà lo stato tramite *useDispatch* come mostrato nel Listato 2.3. Come si può vedere non è possibile modificare lo stato direttamente, ma è necessario effettuare il dispatch dell'azione e sarà poi la libreria che si occuperà di aggiornarlo.

```
1 const favoriteRecipes = useSelector(
2         state => state.favorites.value
3 )
4
5 const dispatch = useDispatch()
6 dispatch(markFavorite(recipesId)
```

Listato 2.3: Redux: esempio d'uso di *useSelector* e *useDispatch*.

L'unico svantaggio dell'utilizzo di Redux è che, per garantire una gestione ottimale ed efficiente dello stato, le azioni devono essere funzioni pure, ossia prive di *side effect*, le quali descrivono come aggiornarlo partendo dallo stato iniziale. In alcuni casi, però, è necessario avere codice più complesso come, per esempio, chiamate API al back-end per recuperare alcune informazioni oppure avere sempre i dati sincronizzati con il back-end tramite dei socket.

Ciò non è possibile in Redux e per questo motivo viene utilizzata un'altra libreria chiamata **RxJS**, la quale permette di lavorare con gli stream dei dati in modo reactive. Può essere usata per moltissimi scopi, ma in questo lavoro viene utilizzata principalmente per creare *observable* sui dati richiesti al back-end.

In tal modo è possibile collegare la funzione che trasforma i dati e crea l'observable chiamata epica all'azione Redux, la quale farà una chiamata API e resterà in ascolto su un socket in modo da avere sempre i dati sincronizzati con quelli del back-end. Il componente, invece, non avrà conoscenza di questo aspetto e lancerà l'azione Redux come se fosse una semplice azione ma, al contempo, è come se avesse un collegamento diretto al back-end ottenendo sempre i dati aggiornati. Un esempio di utilizzo di RxJS collegato ad un'azione Redux verrà mostrato nella Sezione 3.3.

Capitolo 3

Metodologia adottata

3.1 Architettura generale del progetto

L'architettura generale del progetto, visibile in Figura 3.1, è molto articolata e complessa. Le componenti principali su cui si è svolto il mio lavoro sono cinque.

La prima, chiamata ***IoT app***, è un'applicazione React Native la quale permette all'utente di gestire il forno IoT. Questo dispositivo, infatti, essendo sprovvisto di bottoni fisici ad eccezione di quello d'accensione, può essere utilizzato esclusivamente tramite l'app presente nell'elettrodomestico. L'utente, quindi, potrà utilizzarla per eseguire cotture manuali tramite la configurazione dei parametri oppure può eseguire cotture guidate attraverso l'utilizzo di funzionalità come il ricettario oppure il *Guided Cooking*. L'app, inoltre, permette di essere utilizzata anche per l'intrattenimento avendo un browser integrato e la possibilità di utilizzare applicazioni come il meteo o il player musicale.

La seconda componente è la ***libreria IoT*** che si interfaccia a basso livello con l'hardware del dispositivo fisico ed espone dei metodi che permettono di controllare il forno, come per esempio quello per impostare una determinata temperatura oppure per modificare il tempo di cottura.

I seguenti metodi saranno utilizzati dall'IoT app quando dovrà controllare il dispositivo. Con questa architettura si crea un'astrazione perché l'IoT app non saprà qual è l'implementazione effettiva dei metodi che espone la libreria IoT, ma solo la definizione dei loro prototipi. In questo modo, quando si dovrà supportare un nuovo elettrodomestico, non servirà aggiornare l'IoT app, ma basterà sviluppare una nuova versione della seguente libreria modificando il codice secondo le nuove specifiche hardware del dispositivo avendo l'accortezza di lasciare inalterati i metodi che espone.

La terza componente dell'architettura è ***la mobile app*** utilizzata come centro di controllo per tutti i dispositivi IoT della casa produttrice tra cui il forno in questione.

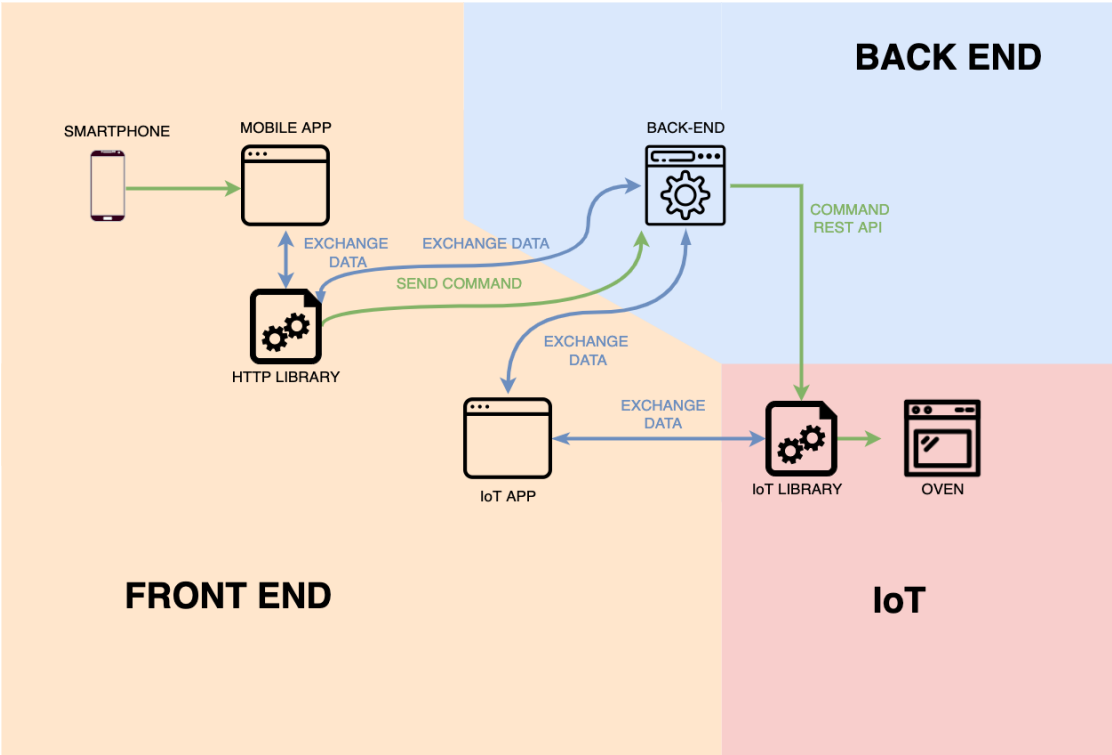

Figura 3.1: Architettura

Figura 3.2: Forno della casa produttrice

L'app permette all'utente di aggiungere tutti i suoi "dispositivi intelligenti" e di controllarli da remoto. Per questo motivo si deve avere la possibilità di comunicare con l'applicazione del forno, sia per chiedere dei dati da visualizzare in app, come lo stato del forno, la temperatura, ecc., sia per lanciare programmi di cottura o ricette.

La comunicazione tra le due applicazioni non avviene direttamente, ma consiste in alcuni step intermedi con il ***back-end***. Questa quarta componente, infatti, oltre a fornire i dati come le ricette o le informazioni dell'utente, permette anche di comunicare con il forno in remoto. L'utente, infatti, dall'applicazione mobile potrà visualizzare lo stato del dispositivo, ossia se è accesso o spento, e l'eventuale programma o cottura che sta eseguendo. Inoltre, potrà anche lanciare comandi ed avviare il forno da remoto.

Questa comunicazione però non avviene direttamente ma tramite un componente chiamato ***http library***, sempre per lo stesso motivo spiegato in precedenza ossia per creare un livello di astrazione ulteriore e separare l'app mobile dalle funzionalità per la comunicazione con il back-end. La libreria esporrà dei metodi e si occuperà di raccogliere e formattare correttamente le informazioni da inviare al back-end. Essendo un'applicazione che gestisce informazioni sensibili e private ogni chiamata al back-end deve essere autenticata e criptata e la libreria si occuperà anche di questo aspetto.

In conclusione, il seguente lavoro di tesi è incentrato sulla sezione front-end (arancione) dovendo sviluppare il *Guided Cooking* sia sull'app mobile che su quella IoT. Inoltre, si utilizzeranno i metodi esposti dalla *http library* e dalla *libreria IoT* per comunicare, rispettivamente, con il back-end (celeste) e con il forno IoT (rosso) visibile in Figura 3.1.

3.2 Interfaccia utente

La progettazione e l'implementazione dell'interfaccia utente dell'app mobile e di quella IoT è stata effettuata utilizzando inizialmente i *mock object*, i quali sono degli oggetti simulati che riproducono il comportamento di quelli reali in modo controllato. In questo modo, il programmatore potrà concentrarsi esclusivamente sulla UI senza considerare ed implementare lo scambio di dati con il back-end.

Essendo un'applicazione composta da moltissime schermate è necessario che la nuova funzionalità rispecchi lo stile e l'interfaccia utente delle altre. Questo crea un'idea di omogeneità nell'app e, inoltre, semplifica l'interazione dell'utente. Per questo motivo l'azienda ha un team di designer che realizza i mockup delle schermate e le valida con il cliente finale prima di fornirle al team di sviluppatori.

Il team di design, per progettare l'interfaccia utente e validarla sia con il cliente che con lo sviluppatore, utilizza il software *Miro*, con il quale è possibile creare

delle pagine bianche chiamate *board* e descrivere il flusso di navigazione tra le varie schermate (Figura 3.3).

Grazie all'utilizzo di questo tool è possibile rappresentare e descrivere in un'unica *board* le specifiche delle funzionalità da implementare fornendo agli sviluppatori un'idea abbastanza fedele su quello che è richiesto da realizzare. Inoltre, è possibile validare questi diagrammi con i manager dell'azienda permettendo di raccogliere i primi feedback ed effettuare le dovute correzioni prima di passare all'implementazione effettiva.

Una volta definite le schermate da realizzare e le relazioni tra di esse, anche ascoltando i feedback dello sviluppatore che andrà ad implementare le funzionalità, utilizzano un tool chiamato *Zeplin*, il quale permette di avere dei riferimenti stilistici per ogni elemento che compone la pagina, come per esempio il tipo di font, la grandezza del carattere, il colore, ecc. Lo sviluppatore, di conseguenza, avrà una linea guida per lo stile in modo da poter replicare il mock-up perfettamente e concentrarsi sullo sviluppo.

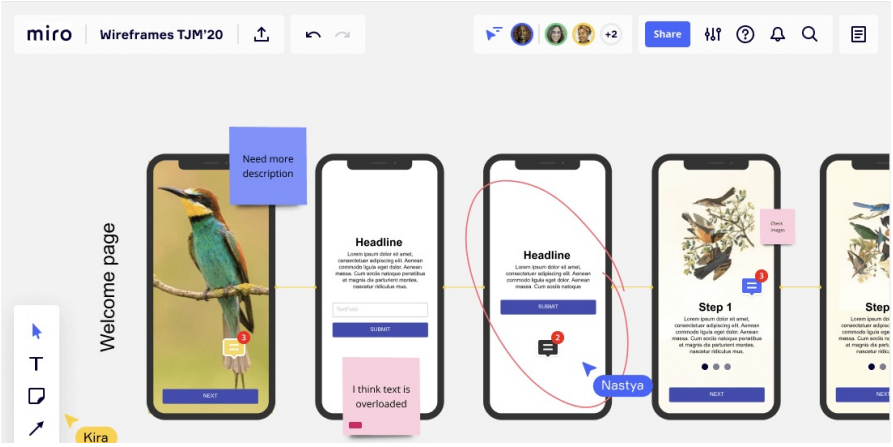

Figura 3.3: Esempio di board di Miro.

Quando si deve inserire una nuova funzionalità in un applicativo pre-esistente è necessario effettuare, prima di procedere alla progettazione ed implementazione effettiva, una fase preliminare di analisi che consiste in tre step.

Il primo consiste nell'avere chiaro il flusso di esecuzione sia dell'applicazione IoT sia dell'applicazione mobile, in modo da evidenziare le similitudini tra i due flussi da implementare, capire le relazioni tra le schermate ed iniziare ad ipotizzare quale sia la soluzione migliore per soddisfare gli obiettivi prefissati.

Lo step successivo è stato quello di testare le chiamate al back-end e controllare la correttezza dei dati che restituiscono. Può capitare, infatti, che i dati mostrati nel mockup siano incompleti oppure siano obsoleti perché modificati dal cliente in una fase successiva. Per questo motivo prima di iniziare lo sviluppo, è sempre necessario

controllare che il *Miro* e i dati del back-end siano allineati e, nel caso non lo siano, si deve andare ad interpellare le due parti per effettuare un allineamento. Questo evita di avere problemi nelle fasi successive di progettazione ed implementazione e permette anche di stimare in modo preciso il tempo necessario per realizzare questa funzionalità.

Il terzo step effettuato è stato quello di scomporre la UI in componenti seguendo la filosofia di React e capire quale scelta adottare per ognuno di essi. Infatti, un componente può già esistere nel progetto e, quindi, non sono necessarie una progettazione ed un'implementazione effettiva, ma solo richiamarlo con le proprietà che ha bisogno per funzionare correttamente. Oppure, potrebbe essere presente all'interno dell'app un componente molto simile che è stato progettato ed utilizzato per un'altra funzionalità ed è necessario valutare e capire se è possibile modificarlo leggermente per adattarlo al caso in questione, avendo l'accortezza di non modificare il funzionamento per i precedenti flussi in cui questo componente viene utilizzato.

In ultimo, se non è possibile adottare una delle due strade precedenti, si deve creare il componente da zero. Questo deve essere implementato cercando di renderlo il più generico possibile in modo da aumentare la probabilità che possa essere utilizzato anche in funzionalità future.

Le considerazioni e le analisi descritte in precedenza sono necessarie e importanti da effettuare perché in applicazioni così grandi è di vitale importanza tenere sotto controllo la quantità di componenti create, i quali corrispondono al numero dei file del progetto. Ogni file, infatti, comporta un aumento di peso sull'applicativo finale e si hanno vincoli molto stringenti sia dettati dagli store su cui pubblicare l'applicazione sia dalla velocità di connessione degli utenti in paesi emergenti. Inoltre, sempre a causa delle dimensioni ingenti dell'app, si devono effettuate delle scelte di programmazione volte a rendere la funzionalità più efficiente ed ottimizzata in modo da non inficiare sulle prestazioni complessive dell'applicativo.

Dopo aver effettuato l'implementazione è consigliato effettuare un testing scrupoloso sulla UI implementata per verificare la sua correttezza su dispositivi di grandezza e sistemi operativi diversi (iOS, Android e HarmonyOS).

3.3 Interazione con il back-end

La seconda fase di questo lavoro consiste nell'implementazione delle chiamate al back-end per ottenere i dati delle categorie, delle sotto-categorie e del programma selezionato. Per effettuarle è necessario interfacciarsi con l'*http library* o con l'*IoT library* rispettivamente se stiamo sviluppando la funzionalità sulla mobile app o sulla IoT app. Una volta implementata e verificata la correttezza dei dati ottenuti, è possibile sostituire i *mock object* usati nella fase precedente con i dati reali.

Inoltre, bisogna considerare che essendo dati dinamici, il cliente finale nel tempo potrebbe aggiungere o rimuovere categorie e sotto-categorie. Di conseguenza, bisogna porre un'attenzione particolare ai casi limite, ossia casi che si discostano dal comportamento standard, come quello dell'assenza di informazioni come per esempio una foto mancante oppure quello di una sotto-categoria con moltissimi programmi. Questi casi possono portare sia a errori di visualizzazione della schermata sia a *crash* a runtime dell'app. Un'ulteriore considerazione riguarda la gestione degli errori di rete come la perdita momentanea della connessione nel dispositivo oppure problemi al back-end, i quali vanno gestiti opportunamente come ad esempio effettuare un *retry* dopo qualche secondo e, nel caso ancora continuano ad esserci problemi, mostrare un errore all'utente.

Un altro aspetto da considerare è che il forno, a differenza dell'app, può essere anche usato in assenza di connessione WI-FI e, di conseguenza, è necessario progettare la funzionalità in modo tale che il *Guided Cooking* funzioni anche offline. Si è deciso quindi di memorizzare una porzione dei dati all'interno dell'app da caricare nel caso in cui ci troviamo nella seguente condizione.

Terza ed ultima fase della progettazione è stata quella di progettare ed implementare l'invio effettivo del programma selezionato al forno attraverso l'utilizzo delle due librerie citate in precedenza.

Questa fase non può essere effettuata e testata con gli emulatori, essendo impossibile da simulare l'hardware dell'elettrodomestico e quindi si è dovuto utilizzare sia il forno che lo smartphone reale per testare la correttezza delle funzionalità implementate; in particolare, che il programma selezionato venga lanciato correttamente con i tutti i parametri di configurazione selezionati dall'utente.

In ultimo, si è generato l'APK delle due app e queste sono state consegnate e testate insieme al team di testing, i quali hanno avuto la possibilità di fare controlli e test ancora più rigorosi sia sull'interfaccia come test su lingue e dispositivi di grandezza diversi per vedere sia sull'interazione con il dispositivo IoT. Il team, dopo un'attenta analisi, ha segnalato dei bug e si è proceduto apportando le dovute correzioni.

Capitolo 4

Interfaccia utente

In questo capitolo viene descritta la prima fase del progetto di tesi: la progettazione e l'implementazione dell'interfaccia utente. Dopo un'iniziale analisi del flusso d'esecuzione affrontato nella Sezione 4.1.1, si andrà a scomporre l'interfaccia in componenti React cercando di riutilizzare il più possibile quelli già esistenti (Sezione 4.1.2) e si tratterà in dettaglio l'implementazione della funzionalità del *Guided Cooking* (Sezione 4.2). In ultimo, nella Sezione 4.3, si migliorerà l'interfaccia creata precedentemente effettuando le necessarie ottimizzazioni e verificando la sua correttezza anche nei casi limite, come l'assenza della connessione oppure la presenza di dati mancanti.

4.1 Progettazione

4.1.1 Analisi dei mock-up e del flusso di esecuzione

La board *Miro*, visibile in Figura 4.1, rappresenta il flusso di esecuzione delineato dall'area design della funzionalità *Guided Cooking* per l'applicazione IoT installata nel forno. Come è possibile vedere nella prima schermata (in alto a sinistra), la funzionalità sarà accessibile dall'*homepage* del forno attraverso un'apposita sezione. Questa, infatti, è costituita da due parti, una in cui si può cucinare manualmente impostando i vari parametri come se fosse un comune elettrodomestico, l'altra in cui è possibile utilizzare tutte le funzionalità accessorie inserite nell'applicazione tra cui la cottura guidata.

Una volta che l'utente ha selezionato la seguente funzionalità, ha la possibilità di scegliere la categoria e la sotto-categoria in base alla pietanza che deve cucinare. Per esempio, se vorrà preparare un piatto di pesce come il carpaccio, dovrà selezionare la categoria "Pesce" e come sotto-categoria quella che più si avvicina alla pietanza da cucinare; in questo caso "Carpaccio di pesce al forno".

L'utente avrà, comunque, la possibilità di cambiare la categoria scelta in precedenza cliccando sul pulsante "More" e, di conseguenza, le sotto-categorie saranno aggiornate sulla base della nuova categoria scelta.

Una volta che l'utente sarà soddisfatto del programma selezionato potrà procedere con la cottura cliccando il pulsante "Continue". Questo permetterà di tornare alla schermata principale del forno con tutti i parametri per la cottura della pietanza impostati correttamente. In ultimo, se l'utente ritiene opportuno, potrà modificarli ulteriormente prima di lanciare effettivamente la cottura tramite il pulsante "Start".

La seguente descrizione rappresenta il flusso d'esecuzione del *Guided Cooking* sul forno. Come è possibile vedere dalla Figura 4.1 è prevista l'implementazione di uno *shortcut* che permette di velocizzare ulteriormente il processo di selezione. L'utente, infatti, potrà scegliere la categoria direttamente dalla sezione in *homepage* attraverso un carosello. Quest'azione permetterà di saltare completamente la selezione della categoria vista in precedenza nella seconda schermata da sinistra.

È importante notare che le categorie, le sotto-categorie e i programmi non sono informazioni statiche, ma è necessario richiederli al back-end quando il forno è online. Inoltre si deve prendere in considerazione che la funzionalità deve essere disponibile, almeno in parte, anche offline in quanto l'utente deve comunque avere la possibilità di utilizzare l'elettrodomestico senza connessione ad internet. Le strategie adottate per soddisfare questo requisito saranno trattate approfonditamente nel Capitolo 5.

La funzionalità del *Guided Cooking* è presente anche all'interno dell'applicazione mobile. Come è possibile vedere in Figura 4.2, sarà accessibile tramite un menu posto nella *bottom bar* dell'applicazione. All'interno dovrà essere inserita un'opzione chiamata "Guided cook without probe" che permetterà di accedere alla nuova funzionalità da sviluppare. Alla prima apertura della nuova sezione sarà presente una modale che spiegherà all'utente il suo funzionamento. Una volta chiusa, l'utilizzatore avrà la possibilità di scegliere le categorie tramite una lista di card. A sua volta, come visto in precedenza, troverà una lista di sotto-categorie tra cui scegliere (Figura 4.3).

Se finora le funzionalità erano molto simili a quelle viste precedentemente per l'applicazione IoT con l'unica differenza del layout dell'interfaccia utente, si può vedere in Figura 4.4 che il flusso per lanciare il programma è completamente diverso. L'utente, infatti, avrà una schermata che riassumerà tutte le impostazioni del programma selezionato e tramite il pulsante "Start" potrà avviarlo. Con lo stesso principio dell'app IoT, cliccando sulla card o sul pulsante "Details", l'utente potrà modificare liberamente i parametri avviando una versione personalizzata del programma. Inoltre, nel caso in cui il preriscaldo del forno (*preheating*) è OFF, avrà anche la possibilità di schedulare nel tempo il suo avvio. L'utente, in tutti e tre i casi, avvierà il programma selezionato da remoto.

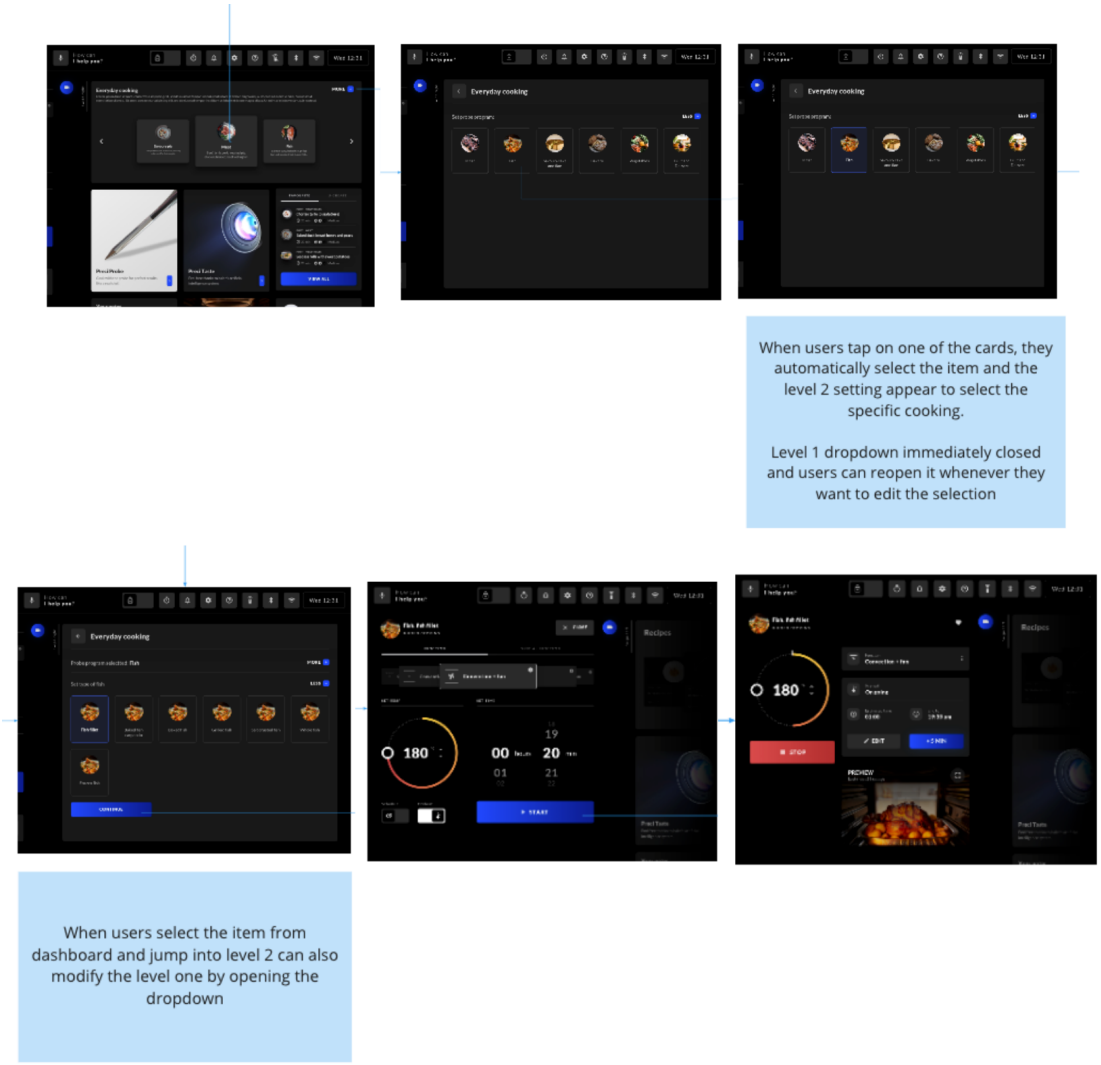

Figura 4.1: Board Miro con il flusso del *Guided Cooking* da implementare nell'app IoT.

4.1.2 Scomporre l'interfaccia utente in componenti

Thinking in React è l'idea principale da considerare quando si sviluppa un'applicazione in React o React Native. Viene, infatti, definito nella documentazione come [5]:

> Una delle caratteristiche migliori di React è il modo in cui ti permette di pensare alle applicazioni mentre le sviluppi.

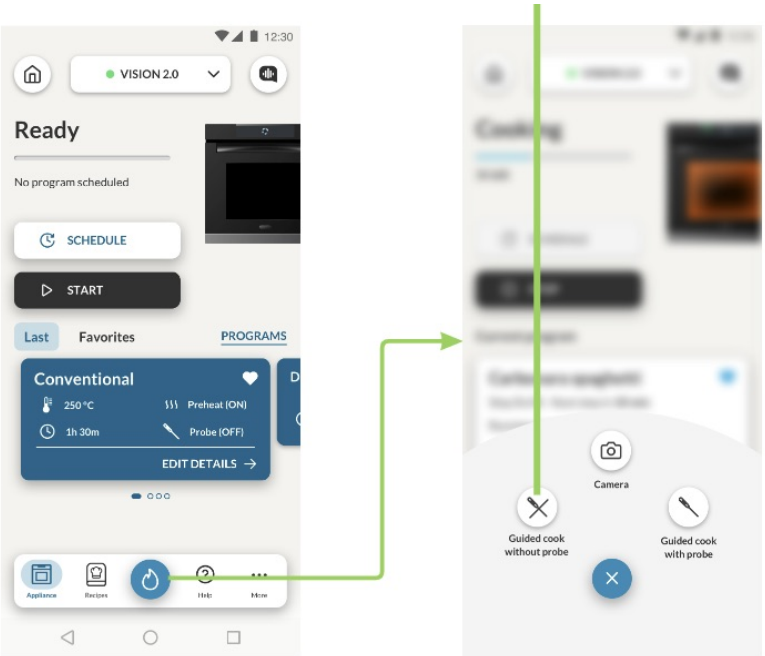

Figura 4.2: Flusso di esecuzione da implementare nell'applicazione mobile per selezionare dal menu il *Guided Cooking*.

Con questo concetto, gli ingegneri di Facebook pongono l'attenzione sul fatto che, quando si inizia a sviluppare un'applicazione o una nuova funzionalità, bisogna innanzitutto pensare all'architettura seguendo la filosofia con cui è stato ideato e progettato il framework. In questo modo si possono porre delle basi solide per la progettazione ed implementazione ottenendo un codice più mantenibile e corretto.

Il primo passo consigliato è quello di scomporre la UI in una gerarchia di componenti. Questo concetto è alla base della filosofia React e permette, se applicato correttamente, di avere componenti altamente generici e riutilizzabili. É consigliato, infatti, scomporre l'interfaccia in componenti e sotto-componenti. In genere, seguendo i concetti e le buone norme di *clean code* e la riutilizzabilità del codice, consigliano che un componente debba occuparsi di un compito soltanto seguendo il *principio di singola responsabilità*. Di conseguenza, se quest'ultimo svolge più di un compito, vuol dire che può essere scomposto ulteriormente in sotto-componenti.

In Figura 4.5 viene mostrata l'analisi effettuata su una delle schermate da implementare tenendo in considerazione i concetti appena trattati. Per primo, si può scomporre la UI in due porzioni: un Header e un Body. L'*Header*, a sua volta, può essere considerato formato da un titolo, un *ActionButton* e uno *Stepper*. Anche il titolo potrebbe essere considerato come una componente, ma nel nostro

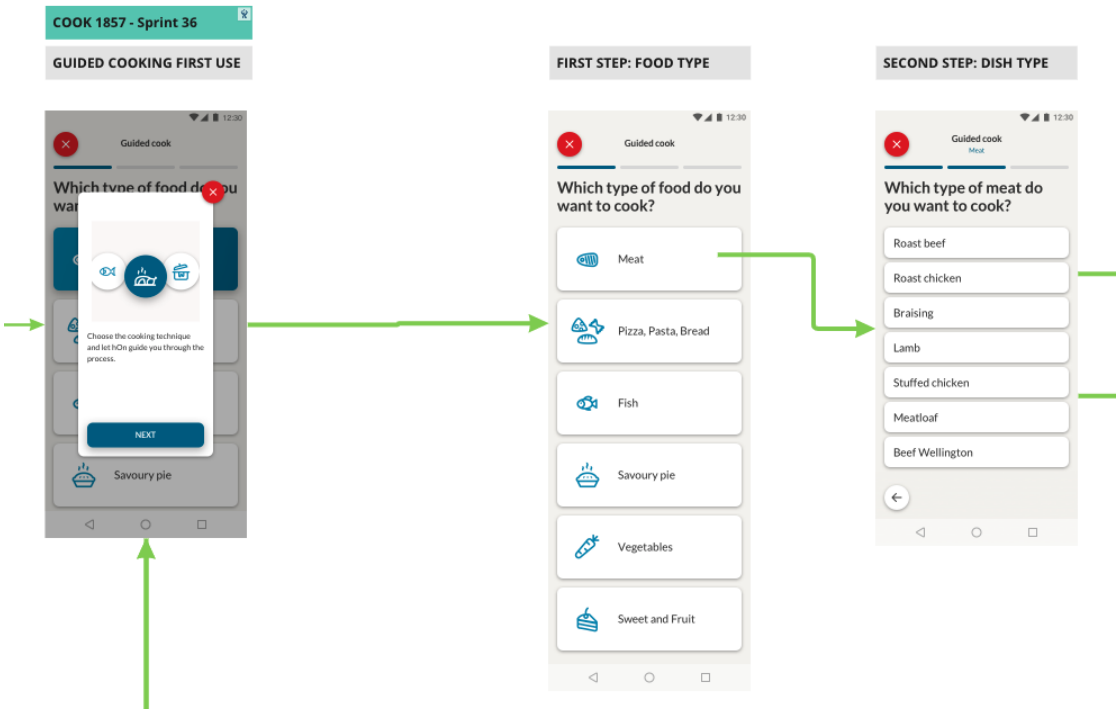

Figura 4.3: Flusso di esecuzione da implementare nell'applicazione mobile per scegliere la categoria e la sotto-categoria.

caso ci limiteremo ad inserirlo all'interno dell'*Header* per non complicare troppo l'architettura.

Anche il *Body* può essere visto formato da un titolo e da una lista di elementi e dove quest'ultima non è altro che un insieme di componenti *Card*. In questo caso è evidente il vantaggio che otteniamo con l'approccio a componenti; è necessario esclusivamente crearne uno che rappresenta la singola card e poi renderizzarlo più volte all'interno della lista quanti sono i suoi elementi. Facendo così, se si cambia il layout del componente *Card*, tutti gli elementi della lista risulteranno modificati.

La Card, infine, potrebbe essere scomposta in due sotto-componenti uno chiamato *CardIcon*, che mostrerà l'icona della card, e l'altro *CardBody* che conterrà il nome della categoria o sotto-categoria. In quest'ottica è possibile notare il concetto di riutilizzabilità; infatti se avremo bisogno all'interno dell'applicazione di una card composta da un'icona e da un contenuto, potremo riutilizzare il componente appena creato.

La Figura 4.6 rappresenta la schermata con il grafico ad albero trattato nella Sezione 2.2.2. Attraverso questa rappresentazione è possibile vedere la gerarchia di

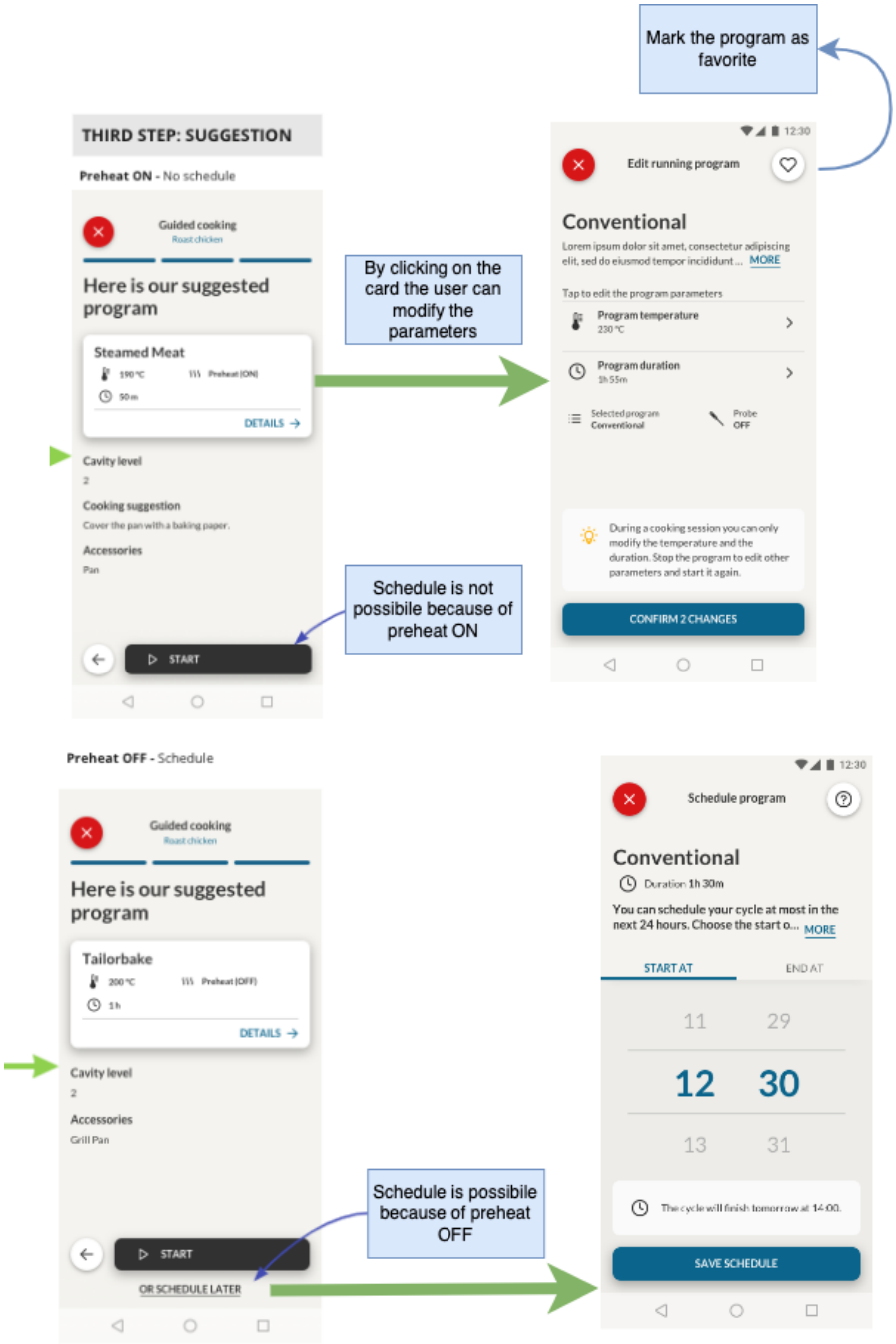

Figura 4.4: Flusso di esecuzione da implementare nell'applicazione mobile per modificare, schedulare e avviare il programma selezionato.

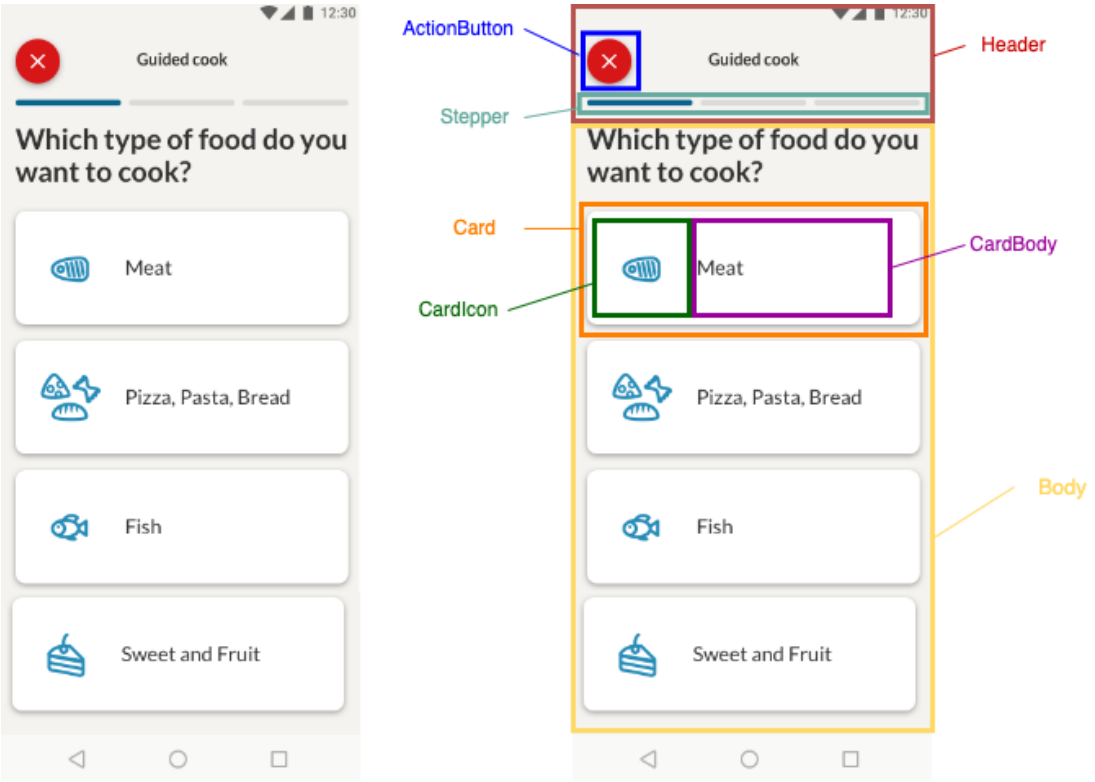

Figura 4.5: Esempio di UI scomposta in una gerarchia di componenti.

componenti ed evidenziare le proprietà e gli stati che necessitano. In particolare si può notare che l'*Header* ha bisogno di conoscere solo in quale dei tre step l'utente si trova, ma non necessita delle informazioni sulle categorie o sotto-categorie che possono essere inserite direttamente all'interno del *Body*.

Guardando il layout dei tre step visibile in Figura 4.3 e Figura 4.4, inoltre, è possibile notare che i tre hanno una UI molto simile e si potrebbe pensare di creare un'unica pagina che racchiuda tutti e tre gli step senza ricorrere a tre pagine separate. Questa soluzione comporta come svantaggio quello di dover scrivere molto codice per gestire il cambio di step, rendendo visibili solo gli elementi che fanno parte di quello corrente e nascondendo gli altri, ma se si suddivide bene il codice in componenti che risiedono in file diversi e se si applicano buone norme di programmazione, si riescono ad attenuare gli svantaggi.

Viceversa separarli in pagine differenti ha il vantaggio di avere una gestione del cambio step più semplice, ma, allo stesso tempo, introduce molto codice ridondante essendo sia il layout che la logica delle tre pagine molto simili. Si è deciso, quindi, di seguire la prima alternativa. Di conseguenza il componente pagina avrà come stato lo step corrente e questo sarà passato sia all'*Header* che dovrà visualizzare

correttamente lo *Stepper*, sia al *Body* che dovrà recuperare i dati e mostrarli nell'interfaccia correttamente.

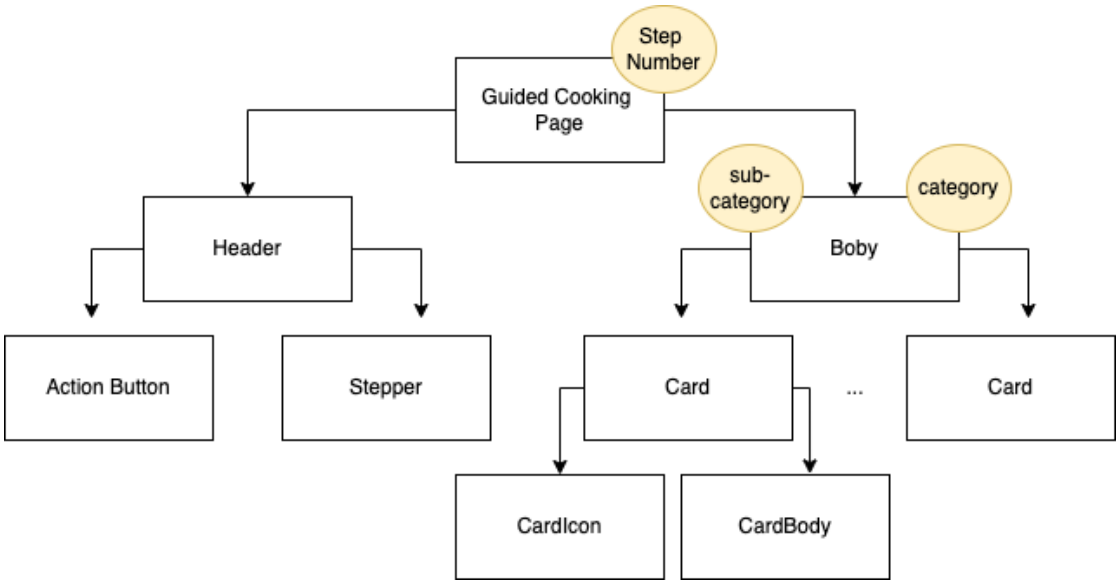

Figura 4.6: Scomposizione dello stepper del Guided Cooking.

Inoltre, si deve porre molta attenzione alla progettazione del layout considerando che l'applicazione è multilingue. Ogni lingua ha, infatti, le sue peculiarità. Il problema principale è che in alcune, come per esempio il tedesco o il cinese, la traduzione del testo può diventare molto lunga portando il layout a rompersi. Di conseguenza, bisogna considera che un titolo o un testo di un bottone potrebbero diventare troppo lunghi per essere contenuti in un'unica riga e perciò si devono adottare alcune soluzioni. Una di queste è quella di dare una larghezza massima al testo in modo che vada a capo forzatamente se la supera, oppure, nel caso in cui non è possibile avere più di una riga, si tronca la frase inserendo dei puntini alla fine.

Le stesse considerazioni possono essere effettuate anche per il flusso dell'app IoT (Figura 4.1). Per i motivi illustrati in precedenza e anche per il fatto che in questo caso i tre step sono rappresentati come sezioni collapsable, si è scelto di adottare la stessa strategia discussa per l'app mobile.

Ulteriore considerazione da effettuare è che i dati delle categorie e delle sotto-categorie non sono statici, ma dovranno essere recuperati tramite un'API. Di conseguenza, bisogna considerare che sui dispositivi con prestazioni non elevate e con connessioni lente, è necessario predisporre un caricamento per evitare che l'utente veda inizialmente la lista vuota. Per questo caso le scelte sono molteplici e si differenziano su dove inserire un *loader*. É possibile posizionarlo direttamente sul

componente pagina e questo comporta per l'utente una schermata bianca con all'interno solo un loader di caricamento. Queste soluzioni si adottano esclusivamente se tutta l'interfaccia cambia completamente a seconda della risposta dell'API. Nel nostro caso, invece, si è a conoscenza che l'unico elemento della UI a cambiare, a seconda della risposta dell'API, è la lista delle card e, quindi, si è scelto di mostrare per intero l'interfaccia e un loader all'interno della lista.

Infine, si è deciso di apportare un'ulteriore modifica per migliorare l'usabilità dell'app. Nei dispositivi più veloci la chiamata API avviene in pochi millisecondi e, quindi, si avrà il comportamento spiacevole di avere il loader che appare e scompare nella pagina solo per pochissimo tempo. Questo comportamento tende a rendere ancora più evidente all'utente il cambio repentino del layout. Si è deciso, quindi, di non visualizzare il loader se la risposta al back-end arriva prima di 500 millisecondi.

4.2 Implementazione

L'analisi e la progettazione effettuate nel Capitolo 4.1 sono state molto utili per capire quali componenti fossero necessari da implementare e per discutere e scegliere quale fosse la tecnica di programmazione più adeguata alla casistica in questione. La fase successiva è quella dell'implementazione e, in particolare, vengono descritti due componenti che hanno richiesto di adottare strategie particolari per fornire una migliore esperienza utente.

4.2.1 Implementazione di un componente riutilizzabile

Il primo componente è mostrato nel Listato 4.1 e rappresenta la *Card* delle categorie e sotto-categorie descritta nella sezione precedente. Si è scelto di mostrare questo particolare componente perché ha richiesto numerose sfide di implementazione.

Innanzitutto si è deciso di denominarlo il più generico possibile in modo da poter essere utilizzato nelle situazioni in cui c'è la necessità di implementare una card con un titolo ed una descrizione. Per permettere di utilizzare lo stesso file sia sull'app mobile che su quella IoT, si è introdotta anche una proprietà chiamata *horizontal*, la quale consente di creare due varianti di layout; orizzontale quando il titolo e la descrizione sono al lato destro dell'immagine, verticale quando sono posizionate sotto l'immagine. In questo modo, la logica del componente è la stessa in tutte e due le app, mentre lo stile sarà opportunamente modificato per rispecchiare i mock-up descritti nella Sezione 4.1.1.

In secondo luogo si sono implementate le traduzioni utilizzando la funzione già sviluppata internamente chiamata *ht*, la quale, data la chiave della stringa, permette di recuperare la traduzione associata nella lingua impostata dall'utente. Le chiavi, che saranno contenute nelle proprietà *title* e *description*, sono state recuperate

dal back-end e la funzione *ht* le utilizzerà per richiedere al servizio traduzioni di un'azienda terzi la stringa nella lingua corrente.

Per quanto riguarda l'implementazione dell'immagine, invece, si è deciso di utilizzare il componente *Image* fornito dalla libreria di *React Native*, il quale permette una gestione nativa dell'immagine a seconda della piattaforma per cui sarà buildata l'applicazione. Il problema principale di questo componente è legato al fatto che non effettua nessun controllo sull'immagine che gli viene passata nella proprietà *source*. Per esempio, se è un'immagine inesistente, come una stringa vuota oppure un link corrotto, questo non renderizzerà nessuna immagine all'interno. Inoltre, se gli viene passato un *path* locale scorretto, avremo un errore a *runtime* che causerà il *crash* dell'applicazione.

Come già affermato in precedenza, è molto importante implementare delle soluzioni che permettano di evitare i seguenti problemi anche alla luce del fatto che l'azienda potrebbe modificare nel tempo le categorie e le sotto-categorie, aggiungendo o rimuovendo le immagini, ed è molto probabile che in alcune situazioni ci siano degli errori o delle risorse mancanti. Questo in nessun modo deve lesionare l'esperienza utente.

Perciò si è deciso di inserire un *LoadingScheletron* che fornisce all'utente un'animazione per rappresentare il caricamento dell'immagine, ossia quando la variabile *loading* è a true. Il componente *Image*, intanto, inizierà a scaricare e a caricare nella schermata l'immagine. Quando il procedimento è terminato, sarà richiamata la funzione inline associata alla proprietà *onLoadEnd*, la quale imposterà la variabile *loading* a false permettendo di visualizzare l'immagine reale. Nel caso in cui si verificassero dei problemi, come l'URL dell'immagine non corretto, sarà richiamata la funzione inline associata alla proprietà *onError*, la quale imposterà, come in precedenza, la variabile *loading* a false, ma allo stesso tempo caricherà nell'immagine un placeholder mostrando così all'utente comunque un'immagine.

```
1  const CardImageTitleDescription = ({
2    title,
3    description,
4    image,
5    onPress,
6    horizontal = true
7  }: CardImageTitleDescriptionProps) => {
8    let placeholder = 'asset:/images/categories/icons-gallery.png';
9
10   let imagePath = `${ENV.imageDomainURL}/images/categories/${image
      }.jpg`;
11
12   const [imageToLoad, setImageToLoad] = useState(imagePath ??
      placeholder)
13   const [loading, setLoading] = useState(true)
14
```

```
15    return (
16      <TouchableOpacity style={styles.containerCard} onPress={
      onPress}>
17        <View>
18          <View style={[styles.imageContainer, horizontal ? styles.
      horizontalImg : styles.verticalImg] }>
19            {loading && <LoadingScheletron/>}
20
21            <Image
22                style={[styles.image, {display: loading ? 'none' : '
      '}]}
23                source={{
24                  uri: imageToLoad,
25                }}
26                onLoadEnd={() => {
27                  setLoading(false)
28                }}
29                onError={() => { setLoading(false); setImageToLoad(
      placeholder; })}
30              />
31          </View>
32          <View style={[styles.contentContainer, horizontal ? styles
      .horizontalCont : styles.verticalCont]}>
33            <Text style={styles.title}>{ht(title)}</Text>
34            <Text style={styles.description}>{ht(description)}</Text
      >
35          </View>
36        </View>
37      </TouchableOpacity>
38    );
39  };s
```

Listato 4.1: CardImageTitleDescription: Implementazione del componente *card* per le categorie e per le sotto-categorie.

Il secondo componente che si vuole trattare prende il nome di *CollapsableSection* ed è mostrato nel Listato 4.2. É utilizzato quando si ha molto contenuto nella pagina perché permette di racchiudere, tradotto letteralmente "collassare", intere sezioni all'interno di un header. Per esempio, viene utilizzando nella sezione FAQ dei siti web perché permette di nascondere tutte le risposte e visualizzare solo una lista di domande. Se poi l'utente sarà interessato ad una di queste, potrà cliccare l'header ed il componente mostrerà il suo contenuto con un'animazione di apertura.

Nel dominio del lavoro di tesi viene utilizzato per implementare gli step della scelta delle categorie e delle sotto-categorie all'interno dell'app IoT. In questo modo, anche se l'utente avrà in un'unica pagina tutte le categorie e le sotto-categorie che rappresentano un numero ingente di elementi, non sarà sopraffatto dalla quantità di card presente nella pagina grazie al componente *CollapsableSection*. Infatti,

quest'ultimo, nasconderà tutti gli elementi all'interno di un titolo e l'utente cliccando sul pulsante "More" potrà espanderlo e visualizzare gli elementi che contiene.

Per lo sviluppo è stato utilizzato un componente già esistente nel progetto chiamato *SubtitlePagesMoreButton*, il quale implementa l'header descritto in precedenza. Si è utilizzato, inoltre, il concetto di *children* in React, la quale è una proprietà "speciale" passata automaticamente ad ogni componente dalla libreria. All'interno sono presenti tutti gli elementi compresi tra il tag di apertura e quello di chiusura di *CollapsableSection*. In questo modo, il componente è stato reso più generale possibile ed è agnostico su cosa conterrà, occupandosi solo di espandere o comprimere una determinata sezione.

Per permettere un'esperienza utente migliore è stata aggiunta un'animazione all'apertura e chiusura del componente utilizzando una libreria implementata internamente, la quale fornisce un componente chiamato *AnimatedSection*. Specificando la durata (300 ms) e l'animazione da utilizzare *LayoutAnimation.easeInEaseOut*, è stato possibile permettere un'apertura e chiusura più fluida e animata della sezione.

```
1  const CollapsableSection: React.FC<CollapseSectionProps> = ({
2    title,
3    htObject,
4    isOpen,
5    containerStyle,
6    textStyle,
7    setIsOpen,
8    children,
9  }) => {
10   return (
11     <View style={[styles.containerCollapse, containerStyle]}>
12       <SubtitlePagesMoreButton
13         titleText={title}
14         rightText={
15           isOpen ? 'ht('LESS) : ht('MORE')
16         }
17         iconName={isOpen ? 'vision-minus-sign' : 'vision-plus-sign
     '}
18         textStyle={[styles.sectionTitle, textStyle]}
19         onMorePress={() => setIsOpen(!isOpen)}
20       />
21        <AnimatedSection
22           mode={LayoutAnimation.easeInEaseOut}
23           duration: {300}
24         >
25           {isOpen && children}
26       </AnimatedSection>
27     </View>
28   );
29 };
```

Listato 4.2: CollapsableSection: Implementazione del componente.

4.3 Miglioramenti e validazione

Le fasi di progettazione ed implementazione ci permettono di realizzare un componente funzionante e, a prima vista, completo e si potrebbe pensare che siano esaustive. In realtà, è necessario effettuare un'ulteriore fase che mira a rendere il componente da un lato più ottimizzato e performante, dall'altro corretto anche nelle casistiche più rare come quella della momentanea assenza di connessione.

4.3.1 Ottimizzare le performance di un componente

Per ottimizzare un componente non si ha una soluzione univoca, ma, di solito, una serie di soluzioni che dipendono molto sia dall'ambiente di sviluppo sia dal framework utilizzato.

In generale, la prima osservazione da effettuare riguarda la dimensione dell'applicazione. Anche se i telefoni cellulari diventano di anno in anno sempre più performanti, bisogna considerare che la maggior parte del mercato utilizza dispositivi di fascia bassa caratterizzati da un processore con bassa potenza computazionale e da internet a basse velocità. Di conseguenza, è sconsigliato non superare la dimensione di circa 125 MB, in quanto comporta una pessima esperienza utente sia per scaricare ed installare sia per avviare l'applicazione.

Il primo passo per ridurre il tempo di apertura è ridurre i file di cui necessita per avviarsi. Di conseguenza è importante prima di tutto avere meno file possibili nel progetto e ciò avviene solo se i componenti vengono implementati altamente riutilizzabili. Inoltre, è importante ridurre al minimo gli import dei componenti che non necessitiamo alla partenza dell'app. In generale, si va ad utilizzare una tecnica chiamata *dynamic import* che consiste nell'importare il componente solo esclusivamente quando questo viene utilizzato. In questo modo alla partenza dell'app verranno importati solo i componenti necessari per quella schermata; gli altri saranno caricati esclusivamente se richiesti dall'utente.

Anche nei componenti sviluppati per il seguente lavoro di tesi è stato necessario utilizzare la tecnica del *dynamic import*. Il componente pagina del *Guided Cooking,* infatti, utilizza una libreria esterna per inviare i comandi al back-end, la quale risulta essere pesante, ma viene utilizzata esclusivamente nell'ultimo step e solo se l'utente avvia il programma selezionato. In questo modo, la schermata per la selezione delle categorie risulterà avviarsi più velocemente dovendo solo caricare i componenti grafici.

Il secondo passaggio che si può effettuare per rendere un componente più ottimizzato è quello della *memorizzazione*, in inglese *memoization*, la quale consiste nel memorizzare in *cache* il risultato di espressioni computazionalmente pesanti e fornire il risultato calcolato quando si verificano nuovamente gli stessi input.

Esistono due *hook* che permettono la *memoization* e prendono il nome di *useMemo* e *useCallback* e, in grandi applicazioni, è molto importante utilizzarli perché riducono drasticamente la computazione necessaria per eseguire l'applicazione. La differenza tra i due *hook* è che il primo permette di effettuare la *memorizzazione* di un'espressione costante, ossia priva di parametri, mentre con il secondo è possibile effettuare questa operazione con funzioni che necessitano di parametri. Nel Listato 4.3 viene mostrato un esempio di utilizzo di *useMemo* e *useCallback*. Come è possibile notare i due *hook* necessitano di un array di dipendenze formato da tutte le variabili e funzioni esterne che vengono richiamate all'interno della porzione del codice in cui effettuare *memorizzazione*. In questo modo, anche se abbiamo un componente che viene renderizzato più volte, perché le proprietà al suo interno cambiano rapidamente, l'espressione racchiusa dall'*hook* verrà ricalcolata solo esclusivamente se cambiano una o più dipendenze.

```
1 const memoizedResult = useMemo(() => {
2   return expensiveFunction(propA, propB);
3 }, [propA, propB]);
4
5 const memoizedCallback = useCallback(() => {
6     addSomething((t) => [...t, "New thing"]);
7   }, [todos]);
```

Listato 4.3: Esempio d'uso di *useMemo* e *useCallback*.

Inoltre, un altro aspetto da considerare è quello di evitare di utilizzare le funzioni inline perché queste saranno instanziate ed eseguite nuovamente ad ogni render del componente. Questo perché le funzioni in JavaScript o TypeScript sono oggetti e ogni istanza è diversa dall'altra, di conseguenza questo problema può rallentare le performance. Si consiglia, quindi, di convertire la funzione da quella anonima ad una con un nome ben definito. In questo modo, non verrà ricreata ad ogni re-render del componente.

Il *lazy loading* è una tecnica comune di ottimizzazione delle prestazioni e viene principalmente utilizzata quando si hanno nel progetto moltissime immagini e consiste nel fornire inizialmente all'utente una versione sfocata dell'immagine, la quale è a risoluzione molto bassa e può essere scaricata velocemente, per poi successivamente fornire la versione ad alta risoluzione. In questo modo, è possibile dare priorità ad altre informazioni dinamiche come i testi. L'utente, quindi, non avrà un'esperienza utente spiacevole a confronto di quella che avrebbe avuto vedendo l'interfaccia caricarsi molto lentamente a causa della grandezza delle immagini.

Si può riassumere l'intera iterazione come un processo costituito da tre fasi:

1. Carica il contenuto prima di iniziare il caricamento degli assets, tra cui le immagini.
2. Procede a caricare l'immagine a bassa risoluzione nascondendola da un effetto sfocato ed effettua la richiesta per scaricare la risorsa ad alta risoluzione.
3. Una volta che l'immagine originale è disponibile, verrà mostrata all'utente sostituendo quella sfocata.

Infine, si è effettuata un'ulteriore ottimizzazione per le immagini. In alcune casistiche, come quella della lista di categorie e sotto-categorie, è necessario caricare un elenco di oggetti che contengono molte immagini. Queste influiranno sulle performance, perché impiegheranno molto a scaricarsi e visualizzarsi. Inoltre, la maggior parte di queste non sarà visibile perché il dispositivo, in base alla grandezza dello schermo, potrà visualizzare al più sei elementi della lista. Quindi non c'è assolutamente bisogno che l'app scarichi ed inizi a caricare tutti gli elementi della lista, ma necessita solamente di quelli che risultano visibili nella schermata. In questo modo, riusciamo a dare priorità a quelle che saranno immediatamente visibili e forniremo un'esperienza utente fluida e veloce. Le restanti immagini saranno caricate esclusivamente se l'utente scorrerà la pagina.

4.3.2 Validazione dell'interfaccia nei casi limite

Come già spiegato in precedenza, tutte le informazioni presenti nel *Guided Cooking* sono dinamiche e possono cambiare radicalmente nel tempo. L'azienda, infatti, attraverso una piattaforma esterna, può aggiungere o rimuovere sia nuove categorie che sotto-categorie. Per questo motivo è stato necessario effettuare una validazione dei componenti sviluppati, soprattutto nei casi che si discostano dal percorso standard, ossia quello descritto nei mock-up mostrati nella Sezione 4.1.1.

Innanzitutto è stata validata l'interfaccia provando ad aggiungere moltissime categorie e sottocategorie, cercando di capire se sia l'esperienza utente che le performance dalle pagine risentissero dalla mole ingente di elementi. Fortunatamente, grazie alle tecniche di ottimizzazioni e al *lazy loading* delle immagini trattate nella Sezione precedente, non si sono riscontrate criticità.

Inoltre, sono stati implementati degli *empty state*, ossia degli elementi che informano l'utente tramite un'immagine e una piccola spiegazione che la lista in questione risulta vuota. In questo modo abbiamo prevenuto la possibilità che l'azienda, per errore, lasci una categoria sprovvista di sotto-categorie.

Logicamente in situazioni come queste si è introdotto un logger, il quale permette di inviare un log dell'errore evidenziando la problematica riscontrata in modo da avvisare l'azienda della categoria con le sotto-categorie mancanti.

Un'ulteriore validazione è stata eseguita provando a visualizzare componenti con informazioni mancanti focalizzandoci principalmente su URL mancanti o corrotti.

Si è cercato di validare se l'interfaccia mantenesse il layout corretto anche in mancanza di alcuni dei suoi elementi e se venissero fornite sufficienti spiegazioni all'utente per la mancanza degli stessi.

Si è implementato, anche, un logger nel caso in cui le informazioni mancanti fossero di vitale importanza per il corretto funzionamento del *Guided Cooking*, come la temperatura, il tempo di cottura etc. Senza di esse, infatti, l'utente avrà un programma non completo e non potrà avviarlo.

Infine, si sono effettuati test per validare l'interfaccia su dispositivi di grandezza e lingue diverse. È molto importante effettuare questi test perché in alcune lingue, come il tedesco, le traduzioni possono diventare molto lunghe e si devono adottare delle tecniche affinché il layout non si modifichi. Per esempio, la maggior parte delle volte si progetta il testo in modo che vada a capo forzatamente se supera una certa larghezza, oppure, nel caso in cui non è possibile avere più di una riga, si tronca la frase inserendo dei puntini alla fine.

Capitolo 5

Interazione con il back-end e il forno IoT

La seconda fase del seguente lavoro di tesi consiste nella progettazione ed implementazione della comunicazione con il back-end sia per recuperare le informazioni sulle categorie, sulle sotto-categorie e sui programmi, sia per inviare il programma selezionato al forno connesso.

5.1 Progettazione

Se nel Capitolo 4.1 si è parlato dell'importanza della progettazione per permettere di ottenere un'interfaccia fluida, completa ed esente da errori in modo da fornire all'utente la miglior esperienza possibile, in questo capitolo tratteremo, invece, la progettazione dello scambio dei dati con il back-end. Anche se, a prima vista, non risulta direttamente visibile all'utente finale, una progettazione errata potrebbe causare problemi molto più gravi di quelli visti precedentemente.

Il primo aspetto da considerare è legato alla sicurezza. Infatti non possiamo e non dobbiamo considerare i nostri utenti finali come fidati. Si deve, quindi, tenere a mente che ogni comunicazione con il back-end deve essere sicura in modo da proteggerci, perché le applicazioni saranno scaricate ed utilizzate direttamente sui dispositivi personali degli utenti. Quindi, un utilizzatore malevole potrebbe analizzare l'applicazione scaricata al fine di ottenere indicazioni sulla sua architettura e potrebbe falsificare la sua identità per avere informazioni riservate dal back-end.

Per proteggerci da queste casistiche ogni comunicazione con il back-end deve essere autorizzata ed autenticata. Sebbene i due concetti sembrino simili, essi svolgono compiti differenti, ma ugualmente essenziali per la protezione di applicazioni e dati.

L'autenticazione verifica che l'utente, che vuole accedere ad una determinata risorsa, sia veramente chi dichiara di essere. Di solito, ciò avviene chiedendo all'utente una sfida che può essere risolta fornendo alcune informazioni che solo l'utente, che dichiara di essere, conosce. Per esempio, una sfida può consistere nel fornire la password di un determinato username oppure verificare l'identità attraverso l'utilizzo di sensori biometrici come l'impronta digitale o la scansione del viso.

Sia l'applicazione mobile che quella IoT del seguente lavoro di tesi implementano una procedura di login, la quale chiede all'utente di fornire una email e una password per verificare la sua identità. Se la sfida viene risolta correttamente, il back-end fornirà all'utente un *token*, il quale può essere utilizzato successivamente per effettuare chiamate autenticate. In questo modo, non sarà necessario ad ogni chiamata richiedere all'utente di risolvere la sfida attraverso la procedura di login, ma basterà fornire il *token* ricevuto precedentemente.

L'autorizzazione, invece, consiste nel definire una politica di controllo degli accessi, ossia specificare i privilegi di accesso che si necessitano per richiedere determinate informazioni. Esistono differenti strategie per implementare il controllo degli accessi, ma quella più usata è di racchiudere utenti omogenei all'interno di un determinato gruppo ed assegnare a quest'ultimo dei privilegi. In questo modo, è possibile, per esempio, racchiudere tutti gli utilizzatori dell'applicazione all'interno del gruppo *user* e si può specificare a quali risorse questo gruppo può accedere. Allo stesso modo è possibile racchiudere tutti gli amministratori di sistema all'interno del gruppo *admin*, i quali potranno accedere a una più grande varietà di risorse.

Nel caso in questione il controllo degli accessi non può essere fatto lato client, ossia non si può controllare nelle due applicazioni se l'utente ha i requisiti per effettuare una determinata chiamata, perché un attaccante potrebbe far *reverse engineering* dell'app e modificare il codice rimuovendo il controllo. Per questo motivo, l'autorizzazione viene verificata esclusivamente nel back-end e, attraverso il *token* che fornirà l'utente, si risalirà alla sua identità e si verificherà se ha i privilegi per accedere alla risorsa richiesta.

Nel *Guided Cooking* i dati richiesti riguardano le categorie, le sotto-categorie e il programma selezionato. Si è deciso di progettare una sola chiamata che, in base al parametro passato, consentisse di recuperare una delle tre informazioni desiderate. Inoltre, per permettere futuri sviluppi della seguente funzionalità, si è deciso di introdurre ulteriori due parametri alla chiamata per indicare la tipologia di elettrodomestico e il modello. In questo modo, sarà possibile differenziare le informazioni in base alla tipologia del modello, permettendo così di fornire all'utente solo programmi che possono effettivamente essere usati su quell'elettrodomestico. Inoltre, si avrà già tutta l'architettura sviluppata nel caso in cui l'azienda decidesse di espandere la stessa funzionalità anche in ulteriori elettrodomestici come per esempio, microonde o piani ad induzione.

Un altro aspetto che si è considerato durante la fase di progettazione è stato quello di trovare una soluzione che permettesse l'utilizzo della funzionalità nell'app IoT anche in assenza di connessione. Il forno, infatti, a differenza dell'app che necessità di un collegamento ad internet costante per essere utilizzata, può anche non essere sempre connesso alla rete WI-FI. Per questo motivo, è stato necessario considerare varie soluzioni affinché si potesse utilizzare il *Guided Cooking* offline. L'alternativa migliore tra quelle considerate è stata quella di memorizzare all'interno dell'app un sottoinsieme di categorie, sotto-categorie e programmi in modo da caricarli se la richiesta al back-end non andasse a buon fine. É stata decisa questa soluzione perché indirettamente ci permette di garantire il funzionamento del *Guided Cooking* anche nelle casistiche in cui il servizio è offline perché in manutenzione oppure perché si sono verificati degli errori non previsti.

5.2 Implementazione dello scambio dei dati con il back-end

5.2.1 Chiamate REST API

Le *API*, per esteso *Application Programming Interfaces*, permettono a due programmi di interagire tra di loro attraverso un linguaggio in comune. Tramite una chiamata API, è possibile, quindi, mandare un messaggio al server chiedendo una determinata risorsa. Il server sarà in ascolto su determinati *API endpoint*, i quali rappresentano i sui punti di contatto con l'esterno e, dopo aver validato la formattazione della richiesta e i permessi necessari per evaderla, la processerà raccogliendo le informazioni e inviando la risposta con i dati richiesti o con un messaggio di errore.

Esistono differenti tipologie di API e per implementare le funzionalità del *Guided Cooking* si è scelto di utilizzare le *REST API*. Questa categoria di API non è un protocollo o uno standard, ma rappresenta un insieme di vincoli architetturali. Il server, prima di tutto, deve identificare le informazioni che vuole esporre all'esterno e dividerle in gruppi omogenei. Ad ogni gruppo sarà assegnato un identificatore che nel paradigma REST viene chiamato *risorsa*.

Successivamente il server esporrà degli endpoint su questa risorsa, i quali permettono di richiedere, modificare o aggiornare le informazioni della risorsa. Tipicamente l'endpoint viene rappresentato attraverso un URL, simile agli indirizzi dei siti web, che descrive chiaramente cosa è possibile ottenere se richiamato.

Nel caso del lavoro di tesi le informazioni delle categorie, delle sotto-categorie e dei programmi selezionati sono state inserite all'interno della risorsa che prende come identificatore *recipe*. Per accedere alle informazioni specifiche della funzionalità da

implementare bisogna, però, richiamare la risorsa seguita dal numero di versione e dalla funzionalità richiesta: */recipe/v1/guided-cooking*.

Le REST API, inoltre, sono implementate seguendo il protocollo HTTP, il quale permette attraverso un *Metodo* di indicare al server quello che è possibile fare sulla risorsa. Per esempio, il metodo GET consente di richiedere la risorsa, mentre il POST viene utilizzato per inviare dei dati.

Nel Listato 5.1, viene mostrata la prima chiamata API da implementare, attraverso la quale si ottiene l'elenco delle categorie. Come è possibile notare, è stata progettata con il metodo POST e non con il GET, in quanto, dovendo essere una chiamata autenticata, si ha il bisogno di passare dei parametri nel body. Il metodo GET, infatti, non permette questa possibilità. I parametri in questione sono *applianceType*, il quale indica il tipo di elettrodomestico, con OV si indica il forno, *models* per il modello e *token* per l'autenticazione dell'utente.

```
1 POST //api-iot.example.com/recipe/v1/guided-cooking HTTP/1.1
2 Content-Type:application/json
3 Accept:application/json
4
5 {
6     "applianceType": "OV",
7     "models": "tad_steam",
8     "token": "EAX123AAAABBBBBCCCC"
9 }
```

Listato 5.1: Chiamata API per recuperare le categorie.

Per ottenere le sotto-categorie, invece, è necessario effettuare la stessa chiamata trattata in precedenza aggiungendo un ulteriore parametro chiamato *category*. Ad esempio inserendo *"category":"FISH"* otterremo tutte le sotto-categorie della categoria *Fish*. A sua volta, per ottenere il programma selezionato si dovrà inserire anche il parametro *subcategories*. É possibile vedere in dettaglio le tre chiamate API in Appendice A.

Nel Listato 5.2 viene mostrata la risposta ottenuta chiedendo il programma con la categoria *Fish* e con la sotto-categoria *Baked Fish Cartoccio*. Le informazioni principali ottenute sono il codice univo del programma *prCode_cata_cdm*, la modalità di cottura *programName*, la temperatura *temperature* ed il tempo necessario *programTime*.

```
1 {
```

```
2       "payload": {
3           "program":  {
4               "programName":"PROGRAMS.OV.MULTI_LEVEL",
5               "prCode_cata_cdm": "23",
6               "prCode_pyro_cdm": "23",
7               "temperature":200,
8               "programTime": 25,
9               "models":"tft_base;tft_steam;tad_base;tad_steam;ft_20;
    vision_20;preco;",
10              "preheating":true,
11              "suggestions": {
12                  "suggestion_level": 2,
13                  "suggestion_accessory":"OV.GUIDED_ACCESSORY.PAN"
14              }
15          }
16      },
17      "authInfo":{...}
18  }
```

Listato 5.2: Risposta della chiamata API per ottenere il programma con la categoria *Fish* e la sotto-categoria *Baked Fish Cartoccio*.

5.2.2 Memorizzazione dei dati in locale

Le chiamate API descritte nella sezione precedente saranno utilizzate all'interno dell'epica RxJS chiamata *getGuidedCooking* visibile nel Listato 5.3. Questa è una funzione che verrà eseguita quando si richiamerà l'azione Redux *GET_GUIDED_COOKING*. La particolarità dell'utilizzare la libreria RxJS è che mette a disposizione molti operatori che permettono di scrivere in modo dichiarativo codice asincrono che di per sé è molo complesso e difficile da gestire.

Il primo operatore che si nota nel listato è *pipe*, il quale, proprio come dice il suo nome che in italiano vuol dire *tubo*, è una funzione pura che prende un *observable* in input, attua le trasformazioni descritte all'interno, e ne ritorna un altro in output. In questo modo è possibile creare un *observable* con i dati ritornati dal back-end rimanendo in ascolto di eventuali cambiamenti. L'applicazione, quindi, avrà sempre le informazioni aggiornate e non avrà bisogno di effettuare molteplici chiamate API nel tempo.

Si creerà, quindi, una *pipe* che risponderà esclusivamente se sarà lanciata un'azione del tipo *GET_GUIDED_ COOKING* e, per prima cosa, andrà a recuperare lo stato più aggiornato attraverso l'operatore *withLatestFrom* e inizierà ad eseguire un blocco di codice asincrono attraverso l'operatore *switchMap*. Quest'ultimo prenderà la richiesta all'interno del payload dell'azione attraverso *action?.payload* e la passerà

come parametro alla funzione *getGuidedCookingData* del *CookingMicroservice* che effettuerà l'effettiva chiamata API.

Essendo quest'ultima asincrona, si dovrà aspettare che la risposta del back-end arrivi e, per fare ciò, si andrà a creare un'ulteriore *pipe* associata alla chiamata API, la quale avrà in input *response* che rappresenta la risposta dell'API. Dopo aver estratto dal payload le categorie, le sotto-categorie e il programma selezionato si andrà ad eseguire la funzione *storeGuidedCooking* che memorizzerà all'interno dello stato di Redux le informazioni precedentemente manipolate.

```
1  const getGuidedCooking: Epic = (action$, state$) => {
2    return action$.pipe(
3      ofType(RecipesActionType.GET_GUIDED_COOKING),
4      withLatestFrom(state$),
5      switchMap(([action, state]) =>
6        defer(() => {
7            const requestGuidedCooking = action?.payload as
     GuidedCookingRequest;
8
9            return CookingMicroservice.getGuidedCookingData(
10                   requestGuidedCooking
11               );
12       })
13       .pipe(
14         switchMap((response: any) => {
15             const payload = response.data.payload
16             const categories = payload.guidedCookingCategories;
17             const subcategories = payload.Subcategories;
18             const program = payload.guidedCookingProgram;
19
20             // Manipolazioni della risposta
21             const guidedCookingCategories = [...]
22
23             return of(storeGuidedCooking(guidedCookingCategories))
     ;
24         }),
25         catchError((_err) => {
26             [...] // Gestione degli errori di rete
27         }),
28       ),
29     ),
30   );
31 };
```

Listato 5.3: GetGuidedCooking: Epica per richiedere al back-end le categorie, le sotto-categorie e il programma selezionato.

In Appendice B viene mostrato il codice completo dell'epica.

5.2.3 Implementazione dell'offline mode

Come trattato nella Sezione 5.1, il *Guided Cooking* nel forno deve funzionare anche offline e, di conseguenza, si è deciso di implementare la seguente specifica nel *catch* dell'epica *getGuidedCooking*. In questo modo, se quest'ultima non andasse a buon fine sia per motivi di assenza di connessione sia perché il server è momentaneamente irraggiungibile, si finirà nel *catch* e si avrà la possibilità di gestire l'errore.

Nel Listato 5.4 si mostra la gestione del *catch* della funzione *getGuidedCooking*. In particolare, si va a recuperare un file JSON all'interno della cartella *localStorage* che contiene un sottoinsieme di categorie, sotto-categorie e programmi memorizzati localmente in formato JSON all'interno dell'applicazione.

Successivamente, grazie all'operatore RxJS chiamato *concat*, verranno eseguite tre funzioni in cascata. Per prima cosa si andrà a richiamare *storeGuidedCooking* che memorizzerà all'interno dello stato di Redux le informazioni recuperate, poi si notificherà all'utente, attraverso un messaggio che apparirà in app, che i dati presenti sono solo un sotto-insieme perché si trova in modalità offline attraverso la funzione*notifyUserOfflineMode*.

Infine, si chiamerà *loggerCauseError*, il quale prenderà in input l'errore che ha scaturito il *catch* e, se questo è differente da quello che identifica l'assenza di connessione, memorizzerà la causa dell'errore e lo invierà ad un servizio di logger di terze parti. In questo modo, è possibile avere informazioni riguardo l'errore che ha causato il seguente problema e correggerlo per poi rilasciare un aggiornamento dell'app.

```
1  const getGuidedCooking: Epic = (action$, state$) => {
2    return action$.pipe(
3      ofType(RecipesActionType.GET_GUIDED_COOKING),
4      withLatestFrom(state$),
5      switchMap(([action, state]) =>
6        defer(() => {
7            [...] // Codice mostrato nella sezione precedente
8    }).pipe(
9          switchMap((response: any) => {
10           [...] // Codice mostrato nella sezione precedente
11          }),
12         catchError((_err) => {
13           console.warn('Guided cooking epic error: ', _err);
14
15           const guidedCookingCategories = require('../assets/
     localStorage/guided-cooking-categories.local').default;
16
17           return concat(
18                   of(storeGuidedCooking(guidedCookingCategories)),
19                   of(notifyUserOfflineMode()),
20                   of(loggerCauseError(_err))
```

```
                    );
            }),
        ),
      ),
    );
};
```

Listato 5.4: GetGuidedCooking: Implementazione del *catch* per gestire errori di connessione.

In Appendice B viene mostrato il codice completo dell'epica.

5.3 Controllare il dispositivo IoT

Nell'ultima sezione del capitolo affronteremo l'implementazione dell'invio del programma selezionato dall'utente al forno connesso sia dall'app mobile che dall'app IoT.

5.3.1 Invio dei programmi dall'app IoT

L'applicazione IoT comunica con il forno attraverso una libreria sviluppata internamente chiamata *IoT Library* che si interfaccia a basso livello con l'hardware del dispositivo fisico ed espone dei metodi che permettono di controllare il forno, come per esempio quello per impostare una determinata temperatura oppure per modificare il tempo di cottura.

Nel Listato 5.5, viene mostrato l'invio del programma al forno dall'app IoT. Per prima cosa si prepara la UI della dashboard in modo da passare dalla normale visualizzazione a quella di quando si ha un programma in cottura. Poi si avvisa l'utente attraverso un feedback audio informandolo che il forno ha iniziato a cucinare. Nel caso in cui l'utente abbia attivato l'impostazione per avviare il programma in ritardo, si va a calcolare il *delay* in minuti. Infine, si inviano tutti i parametri alla funzione *sendApplianceCommand* che comunicherà con l'IoT library e si tornerà alla dashboard.

```
const startCooking = () => {
    // Trigger UI changes
    props.setIsCookingActive(true);

    // Show audio effect
    props.startAudioEffect(FileName.COOKING_START);

    // Check the duration of the program and calculate the delay
   if scheduled
```

```
9       const duration = this.getDuration();
10      let delay: number = 0;
11      if (props.isScheduled){
12        delay = props.calculateDelay();
13      }
14
15      // Send program to OvenLibrary
16      props.sendApplianceCommand(
17          props.selectedOvenFunction.hOnProgramName,
18          delay,
19          duration,
20          props.program,
21          props.isUsingPreheating,
22          props.isProbeCooking,
23      );
24
25      // Go to dashboard
26      this.props.updateDashboardActiveState(
27          DashboardEntertainmentFlow.DASHBOARD,
28      );
29    };
```

Listato 5.5: StartCooking: Invio di un programma al forno dall'app IoT.

5.3.2 Invio dei programmi dall'app mobile

Nel Listato 5.6 viene mostrata la funzione *SendCommand*, la quale permette di inviare un programma sul forno dall'app mobile. Per farlo, l'app deve comunicare con l'*http library*, la quale esporrà dei metodi e si occuperà di raccogliere e formattare correttamente le informazioni da inviare al back-end.

L'implementazione di questa funzionalità consiste in due parti. Il primo step è creare un oggetto del tipo *SendCommandHttpPayload* che conterrà tutte le informazioni necessarie per avviare il programma selezionato, come il tempo di cottura, la temperatura, il preriscaldo.

Inoltre, essendo un'applicazione che gestisce informazioni sensibili e private, ogni chiamata al back-end deve essere autenticata e criptata e, quindi, si ha la necessità di inserire all'interno dell'oggetto le informazioni sul device dell'utente, il mac address dell'elettrodomestico ed il token di autenticazione. Si andrà, poi, a creare l'oggetto *command* e quest'ultimo sarà passato come parametro alla funzione della libreria *http library* che permetterà effettivamente di inviare il comando al back-end.

Infine, si navigherà versò la dashboard dove l'utente potrà vedere le informazioni del programma appena avviato.

```
1  const sendCommand = () => {
2
3     const sendCommandPayload: SendCommandHttpPayload = {
4        macAddress: activeAppliance.macAddress,
5        timestamp: moment().toISOString(),
6        commandName: CommandName.setParameters,
7        programName: suggestedCycle?.prName,
8        applianceOptions: applianceOptions,
9        device: {
10         mobileId: DeviceInfo.getUniqueId(),
11         mobileOs: Platform.OS,
12         osVersion: String(Platform.Version),
13         appVersion: rootConfig.appVersion,
14         deviceModel: DeviceInfo.getDeviceId(),
15       },
16       attributes: {
17         prStr: ht(suggestedCycle?.prName),
18         channel: SendCommandChannel.MobileApp,
19         origin: SendCommandOrigin.StandardProgram,
20       },
21       parameters: {
22         delayTime: delayTime,
23         onOffStatus: status,
24         prCode: suggestedCycle?.prCode,
25         prPosition: suggestedCycle?.prPosition,
26         prTime: suggestedCycle?.programTime,
27         tempSel: suggestedCycle?.temperature?.toString(),
28         preheatStatus: suggestedCycle?.preheating ? '1' : '0',
29       },
30       applianceType: activeAppliance.applianceTypeName as string,
31       token: userData.token
32     };
33
34     const command = {
35       commandName: CommandName.startProgram,
36       sendCommandPayload,
37       applianceMacAddress: activeAppliance.macAddress,
38     };
39
40     sendCommand(command);
41     navigation?.navigate(Screen.OVHomepage);
42   };
43 }
```

Listato 5.6: SendCommand: Invio di un programma al forno dall'app mobile.

Capitolo 6

Risultati ottenuti

In questo capitolo verrà presentato un'analisi dei risultati ottenuti dai test di Q&A effettuati in collaborazione col team dedicato in azienda. L'obiettivo è quello di testare l'applicativo prima che venga rilasciato in produzione in modo da identificare, categorizzare e correggere eventuali errori. Il rischio, se questi errori non venissero corretti, è quello di andare incontro ad un danno economico e d'immagine.

Il lavoro di tesi comporta l'inserimento del *Guided Cooking* all'interno di un prodotto preesistente ed è importante che le nuove funzionalità aggiunte non vadano ad inficiare sulle performance dell'architettura d'origine. Dovranno, inoltre, permettere alti standard di usabilità e semplicità di utilizzo, essendo l'utenza finale non omogenea.

La modalità *Guided Cooking* ha l'obiettivo di semplificare l'utilizzo dell'elettrodomestico di tipo forno, aumentando così l'interesse dei clienti e l'usabilità del prodotto. Un'analisi sui test effettuati permette di valutare il lavoro svolto e correggerlo in caso di errori riscontrati.

6.1 Valutazione del team di Q&A testing

Nella Sezione 1.2 sono stati presentati gli obiettivi del lavoro di tesi. La nuova modalità di utilizzo sviluppata, chiamata *Guided Cooking*, permetterà un'interazione più semplice ed immediata con gli elettrodomestici, in particolare con quelli della tipologia forno. L'utenza non omogenea, alla quale il prodotto è destinato, rende necessaria un'interfaccia semplice ed immediata nell'utilizzo. Inoltre, la complessità del progetto richiede un'attenzione particolare alle prestazioni.

Prima di rilasciare sul mercato nuove funzionalità, è altamente consigliato valutarle e testarle in modo approfondito. Il rilascio di un prodotto non ottimale potrebbe comportare gravi perdite per l'azienda in termini economici e d'immagine. Le nuove funzionalità introdotte, inoltre, non dovranno inficiare sulla piattaforma

IoT pre-esistente e dovranno portare un valore aggiunto ad un prodotto che già gode dell'attenzione di tanti clienti abituali.

Per questo motivo le aziende hanno, al loro interno, un team dedicato alla valutazione che permetterà di sollevare dubbi e criticità prima del rilascio in produzione. Il metodo di testing utilizzato può essere differenziato in due macro-categorie: manuale e automatico.

Il *test manuale*, proprio come indica il termine, viene effettuato manualmente dagli operatori e ha lo scopo di analizzare attentamente le performance del prodotto. Essendo una procedura lenta, perché richiede molto tempo, e vasta, in quanto esistono una miriade di casi ed aspetti che si possono controllare in un software, è necessario definire in partenza una precisa selezione dei casi da prendere in esame e definire il risultato per cui il test può essere considerato come superato. In questo modo, si può definire in maniera chiara qual è il preciso aspetto da valutare per ciascun test effettuato.

Inoltre è favorevole l'impiego di un vasto gruppo di utilizzatori e una selezione accurata di dispositivi di diverse marche e fasce di prezzo, in modo da permette una valutazione più critica delle prestazioni su diversi tipi di hardware e sull'usabilità del prodotto. Alcune aziende, inoltre, effettuano il *crowdtesting* che consente di far partecipare gruppi di utenti, simili a quelli che utilizzeranno l'applicativo finale, a sessioni di testing permettendo così di avere risultati più efficaci e verosimili a quelli reali.

I *test automatici*, invece, permettono di scrivere codice per valutare la correttezza del software e hanno il vantaggio che, essendo automatici, non hanno bisogno di un operatore che effettui i test manualmente. Per questo motivo non si è più limitati dalla grandezza del team di testing e molte più casistiche ed aree dell'applicativo possono essere considerate. La difficoltà di questa categoria di testing è quella della creazione e gestione dei test in maniera automatica, in quanto bisogna considerare che l'implementazione dei test occupa una buona porzione del tempo a disposizione degli sviluppatori. Inoltre, non tutte le casistiche di testing risultano efficaci se fatte in modo automatico, perché è praticamente impossibile valutare il rapporto di un utente con l'applicazione ed è molto difficile testare la correttezza del layout sui vari dispositivi.

La soluzione migliore, quindi, è quella di avere entrambe le categorie di testing ed utilizzarle in sinergia. A causa dei costi elevati, moltissime aziende effettuano solo i test manuali, sacrificando quelli automatici.

Anche l'azienda in cui si è svolto il seguente lavoro di tesi ha un team di *Q&A testing* che conduce principalmente test di tipo manuale e valuta la correttezza e le prestazioni delle funzionalità sviluppate attraverso l'utilizzo di un vasto range di dispositivi di diversa grandezza, sistema operativo e fascia di prezzo. Per ogni problema trovato, il team apre un ticket descrivendo l'errore e i passi per ricrearlo ed assegna una gravità in modo da avere una priorità su quelli più gravi, come i

crash dell'applicazione o i bug che impediscono il suo corretto funzionamento.

I tipi di test vengono definiti sulla base delle necessità dell'applicazione presa in esame. Nella Sezione 6.1.1 verranno descritti i tipi di test eseguiti e i motivi per cui è importante effettuarli, mentre nella Sezione 6.1.2 verranno descritti i *ticket* ottenuti dal testing del *Guided Cooking* e le modifiche apportate per risolvere i seguenti problemi.

6.1.1 Metriche di valutazione

La valutazione dei test effettuati è utile al fine di migliorare, quando necessario, le funzionalità prima del rilascio in produzione. Quando un test viene superato, non necessita di ulteriore approfondimento, mentre se risulta fallito, verrà aperto un *ticket* che conterrà commenti utili al fine di risolvere il problema riscontrato. Si assegnerà, inoltre, una priorità tra bassa, media o alta, la quale descrive la gravità dell'errore identificato. Al fine di valutare i risultati ottenuti in questo lavoro di tesi verranno presi in considerazione i *ticket* aperti dal Q&A team a seguito dei test effettuati e, sulla base delle priorità assegnate nella Sezione 6.1.2, si discuterà dei risultati ottenuti.

Esistono molteplici tipologie di test che possono essere effettuate su un applicativo. I test da svolgere vengono selezionati dal team considerando vari aspetti del prodotto finale e della funzionalità da testare, come per esempio la complessità del progetto, il target finale, l'utilizzo effettivo della nuova funzionalità, etc. I test effettuati in questo lavoro possono essere suddivisi in quattro tipologie:

- **Test di funzionalità**: questi tipi di test mirano a valutare il corretto funzionamento dell'applicativo. In questo lavoro di tesi sono state aggiunte funzionalità ad un applicativo preesistente ed è necessario essere certi del loro corretto funzionamento in modo da non inficiare sull'architettura preesistente.

- **Test sulla UX e UI**: test che cercano di valutare quanto un'interfaccia sia facile da usare e rispecchi fedelmente quanto pianificato. Nel lavoro presentato, il target degli utenti è molto vasto perché queste categorie di elettrodomestici sono quotidianamente utilizzate da persone di diversa fascia di età e nazionalità, perciò è necessario che l'applicativo sia semplice ed intuitivo anche per utenti non tecnologicamente esperti.

- **Test di compatibilità**: sono test che validano la compatibilità dell'app provandola su dispositivi con hardware e sistema operativo differente. L'app mobile è multi-piattaforma, compatibile con Android versione 7 o superiore e con iOS 13 in avanti. In fase di test saranno valutate solo alcune versioni dei sistemi operativi privilegiando quelle più utilizzate dagli utenti e, in particolare, Android 7, 11 e 13 ed iOS 13, 14 e 15. Recentemente si è dovuta aggiungere

anche la compatibilità con il sistema operativo HarmonyOS. L'applicazione del forno, invece, è compatibile solo con Android, essendo di questa tipologia il dispositivo inserito nell'elettrodomestico.

- **Test di performance**: questo tipo di test è utile per valutare le performance dell'applicazione durante le operazioni più critiche ed è strettamente legato ai testi di compatibilità descritti in precedenza. Per questo motivo nelle sezioni successive saranno considerati e discussi insieme.

La fase di testing prevede molteplici step, in questo documento si riportano solo i report forniti dal Q&A team a seguito dei test eseguiti e le modifiche apportate al fine di correggere i bug identificati.

La Figura 6.1 riporta un esempio di un ticket aperto e, in particolare, descrive un bug sul pulsante per impostare il forno in modalità preriscaldo. La dicitura "[OV]" permette di classificare l'errore in base all'elettrodomestico testato, in questo esempio la sigla è utilizzata per indicare un forno. La possibilità, inoltre, di allegare materiale multimediale come foto o video ad un ticket permette di identificare con estrema precisione l'errore al fine di correggerlo. Nella sezione di destra, si può notare il campo "Dettagli" dove vengono riportate le informazioni utili come chi ha aperto il ticket, la priorità assegnata, ed altri aspetti burocratici. Il "Reporter" è colui del team di Q&A testing che ha identificato il bug, l'"Assegnatario" è colui che si occuperà di risolverlo. Il *ticket* ha, infine, una priorità media a causa del bug che comporta l'impossibilità di usare la cosiddetta funzionalità.

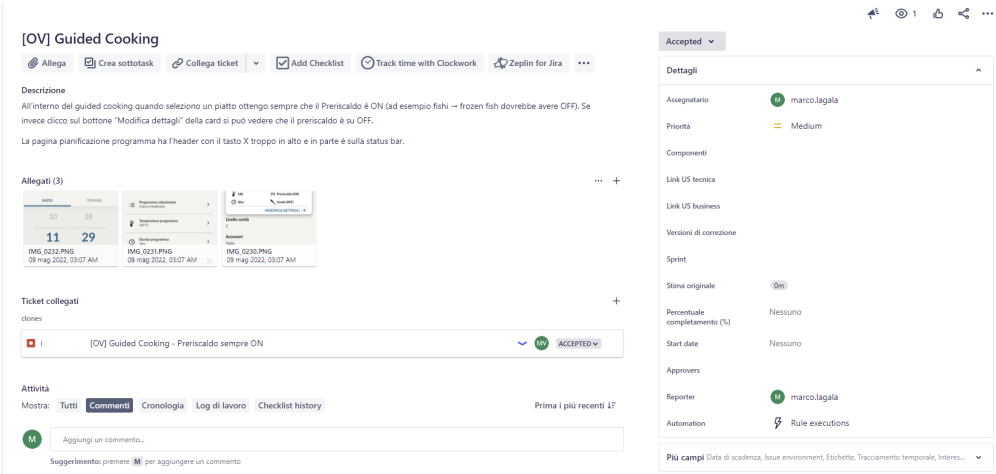

Figura 6.1: Esempio di un ticket aperto dal team di testing.

6.1.2 Analisi dei risultati ottenuti

Come precedentemente descritto, il team di *Q&A testing* effettua i test rispettando le tipologie di interesse selezionate e, se un test risulta fallire, aprirà un ticket spiegando la problematica. A seguito dei test selezionati per il *Guided Cooking*, i ticket aperti sono presenti nella Figura 6.2.

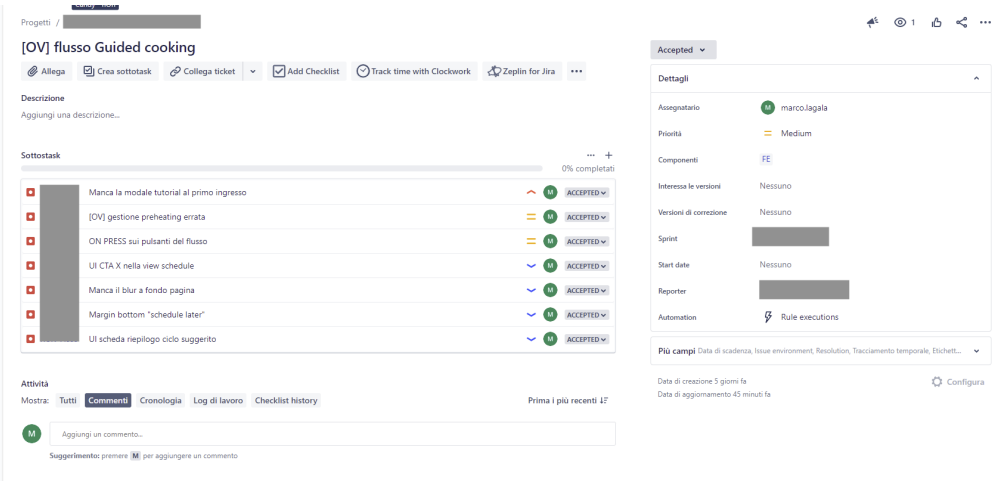

Figura 6.2: Bug aperti dal team dopo il testing del *Guided Cooking*.

I test effettuati ricadono nelle tipologie precedentemente presentate: funzionalità, UX e UI, compatibilità e performance. É importante ricordare che i bug presentati in Figura 6.2 sono riconducibili a test falliti, molti altri test sono stati effettuati e sono stati superati.

Tutti i *test sulle funzionalità* sono stati superati. Questo è un risultato ottimo, indice del fatto che le implementazioni erano corrette e ben integrate nel progetto preesistente.

I *test sulla UX e UI*, invece, hanno riportato qualche errore. Dalla Figura 6.2, si può notare che tutti i ticket aperti appartengono al seguente ambito e sono relativi a piccoli problemi dell'interfaccia utente, come per esempio l'assenza di un effetto *blur* in un testo o un pulsante posizionato troppo in basso. Inoltre, risulta assente la modale del tutorial per il primo utilizzo. Tutti i ticket sono stati risolti e corretti prontamente ed i test falliti sono stati ripetuti, questa volta superandoli con successo. Solo un bug riscontrato risulta avere una priorità alta, mentre due hanno priorità media e quattro bassa.

Considerando, infine, i *test di compatibilità e performance* è noto che i modelli più vecchi potrebbero avere problemi di sovraccarico a causa della loro ridotta potenza computazionale. É, quindi, necessario effettuare test dedicati su questi device, al fine di garantire il loro corretto funzionamento. Il test di compatibilità ha restituito un bug con priorità alta a causa di un *crash* dovuto a versioni datate del

sistema operativo. Il principale motivo era dovuto all'utilizzo di una libreria interna per le animazioni che non supportava la versione datata del sistema operativo. La soluzione al seguente problema è stata quella di non utilizzare la suddetta libreria per quella versione dell'OS risolvendo il *crash* e, inoltre, non sovraccaricando il processore di bassa potenza utilizzando le animazioni.

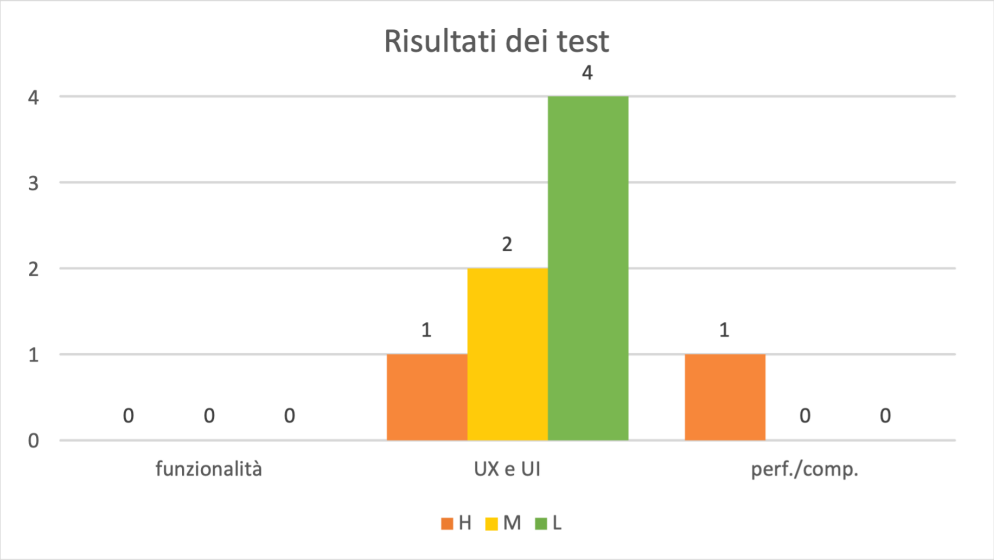

Figura 6.3: Risultati dei test effettuati.

La Figura 6.3 riassume i risultati ottenuti e riporta un istogramma utile a visualizzare la quantità di ticket aperti e le priorità ad essi collegate. Le priorità sono alta (H), media (M) e bassa (L).

Si può affermare, quindi, che i test hanno confermato la corretta implementazione e progettazione della funzionalità *Guided Cooking* sia sull'app IoT che sull'app mobile. Testimoniano, inoltre, che la nuova funzionalità risulta ottimizzata e non causa un peggioramento delle prestazioni nell'architettura preesistente permettendo così di fornire un valore aggiunto e di migliorare l'usabilità del forno connesso.

Capitolo 7

Conclusioni

Lo *Smart Home* è l'evoluzione tecnologica della domotica e consiste nel rendere "intelligenti" i dispositivi presenti nelle abitazioni come gli elettrodomestici, le lampadine e le videocamere di sorveglianza fornendo loro l'accesso ad Internet e potendo così controllarli e monitorarli a distanza.

Le aziende mirano a creare piattaforme centralizzate, con le quali è possibile controllare e monitorare tutti i dispositivi da loro realizzati in modo da fidelizzare il cliente al proprio marchio. Analisi di mercato hanno dimostrato come soddisfare i bisogni di un cliente abituale porta ad un guadagno più sicuro e stabile nel tempo. Per ottenere ciò si deve sempre aggiornare il proprio prodotto ed ampliarne le sue funzionalità, realizzando soluzioni *user friendly* che semplificano e incentivano l'utilizzo.

Il lavoro di tesi presentato in questo documento è stato svolto all'interno di una piattaforma IoT di una nota casa produttrice, che permette di gestire molteplici elettrodomestici presenti nell'abitazione in maniera semplice ed immediata. Nel Capitolo 3 si è descritta l'architettura generale della piattaforma concentrandosi sui vari elementi che la compongono e che permettono di controllare e comunicare da remoto con i vari dispositivi.

L'obiettivo della tesi è stato quello di migliorare l'esperienza utente di un forno connesso. Attraverso studi di mercato da parte dell'azienda si è notato, infatti, un calo di interesse del consumatore nei confronti dell'elettrodomestico che ha portato l'azienda a volerne migliorare l'interazione quotidiana. Le problematiche principali erano dovute al fatto che la configurazione del forno per i pasti giornalieri era lunga e tediosa, soprattutto a causa delle numerose impostazioni da configurare. La soluzione proposta in questo documento si chiama *Guided Cooking* ed ha lo scopo di risolvere i problemi citati in precedenza ed invogliare di nuovo l'utilizzo del forno da parte degli utenti.

La funzionalità è stata inserita sia nell'app mobile, che ha il compito di essere il centro di controllo di tutti i dispositivi dell'utente, sia nell'app IoT installata all'interno del forno connesso. Lascia all'utente, quindi, la possibilità di utilizzarla sia direttamente nell'elettrodomestico, ma anche da remoto attraverso lo smartphone e una connessione ad internet.

La prima parte del lavoro svolto, trattato nel Capitolo 4, si è incentrato sulla progettazione e sull'implementazione dell'interfaccia utente sia nell'app mobile che in quella IoT, cercando soluzioni che ne migliorassero l'esperienza utente e focalizzandosi nel rendere la funzionalità la più fluida ed ottimizzata possibile. Le modifiche apportate, infatti, non devono appesantire ulteriormente l'architettura preesistente, la quale risulta già essere ingente a causa della mole di elettrodomestici presenti nell'ecosistema dell'azienda.

Tutte le categorie, le sotto-categorie ed i programmi, inoltre, sono dinamici e vengono recuperati attraverso chiamate API dal back-end in modo da permettere la loro modifica nel tempo. Per questo motivo si è dedicato gran parte del tempo a progettare e verificare che l'interfaccia utente fosse ben costruita ed esente da errori, anche nella casistica in cui l'azienda dimenticasse di fornire alcune informazioni all'interno del singolo programma. Le due applicazioni, inoltre, hanno un target di utilizzo molto ampio ed è stato necessario un'ingente fase di validazione e testing sia con lingue che con dispositivi differenti.

In secondo luogo, nel Capitolo 5, si è gestita l'interazione con il back-end per lo scambio dei dati e la comunicazione con l'elettrodomestico al fine di avviare il programma scelto. Come discusso precedentemente e mostrato in Figura 3.1, l'app IoT dialoga direttamente con il forno tramite l'*IoT library* permettendo, quindi, la comunicazione esclusivamente all'interno dell'elettrodomestico.

Nel caso dell'app mobile, invece, la selezione dell'utente avviene da smartphone in una sezione dedicata alla *Guided Cooking*. Affinché il comando arrivi al forno, questo deve essere inviato alla libreria IoT per mezzo del back-end, il quale gestisce il protocollo di comunicazione. Per questioni di sicurezza, inoltre, tutti i messaggi scambiati sono autenticati attraverso un *token* che verrà rilasciato durante l'iniziale procedura di login e rinnovato nel tempo. Il back-end comunicherà alla libreria il programma selezionato con le varie importazioni scelte e quest'ultima si occuperà di avviare il forno correttamente.

L'approccio utilizzato, inoltre, è stato quello di progettare la funzionalità del *Guided Cooking* in modo che potesse essere estesa anche ad ulteriori modelli di forno ed elettrodomestici. Ad esempio, potrebbe essere utilizzata anche con i piani ad induzione permettendo all'utente di avere dei programmi preimpostati che ne velocizzano l'utilizzo e ne migliorano l'esperienza utente.

Con la strumentazione e l'aiuto del team di Q&A testing, infine, si è sono attuati dei test sul *Guided Cooking* al fine di valutare la correttezza delle modifiche effettuate. Come trattato nel Capitolo 6, i test di funzionalità sono stati tutti

brillantemente superati, mentre i test di compatibilità e performance hanno portato piccoli problemi soprattutto nei confronti di smartphone più vecchi. Nel caso del modello iPhone 6 si è verificato un crash all'avvio della funzionalità del *Guided Cooking*, dovuto al fatto che una libreria per le animazioni utilizzata non era compatibile con il sistema operativo obsoleto. La soluzione per ovviare al problema è stata quella di non utilizzare la suddetta libreria nel caso dei device più datati. I test di UX e UI, di fondamentale importanza visto lo scopo del lavoro, hanno sollevato piccoli bug di carattere grafico, come per esempio la mancanza di un tutorial iniziale che potrebbe portare ad una prima esperienza negativa nei confronti degli utilizzatori. Dopo le opportune segnalazioni, tutti i bug sollevati sono stati prontamente corretti e il team di Q&A testing ha confermato il corretto superamento dei test precedentemente falliti.

7.1 Implementazioni future

Molte sono le ipotesi che permettono di estendere il lavoro svolto. La più semplice e veloce, in quanto durante la progettazione è stata già tenuta in considerazione, è quella di poter implementare le funzionalità del *Guided Cooking* anche ad ulteriori modelli di forni oppure ad altre categorie di elettrodomestici, come per esempio i piani ad induzione.

Un'ulteriore ipotesi, inoltre, è quella di fornire all'utente la possibilità di modificare le categorie e sotto-categorie nascondendo i programmi non utilizzati e creandone di nuovi, in modo da adattarsi personalmente allo stile di vita dell'utilizzatore. In questo modo è possibile aumentare ancora di più l'esperienza utente e la soddisfazione del cliente.

Al fine di ampliare la funzionalità del *Guided Cooking*, si potrebbe aggiungere la possibilità di scegliere le categorie, le sotto-categorie ed il programma selezionato attraverso la voce, utilizzando i comuni assistenti virtuali come *Alexa*, *Siri* o *Google Home*. Questo permetterebbe un ulteriore aumento dell'esperienza utente e renderebbe la funzionalità accessibile ad una più vasta gamma di utenti, compresi quelli con disabilità.

Appendice A

Chiamate API

```
CHIAMATA API PER RICHIEDERE LE CATEGORIE

POST //api-iot.example.com/recipe/v1/guided-cooking HTTP/1.1
Content-Type:application/json
Accept:application/json

{
    "applianceType": "OV",
    "models": "tad_steam",
    "probe": false
}

Risposta:
{
    "payload": {
        "categories": ["OV.GUIDED_WITHOUT_PROBE.FISH","OV.
    GUIDED_WITHOUT_PROBE.FRUIT_DESSERT","OV.GUIDED_WITHOUT_PROBE.
    MEAT","OV.GUIDED_WITHOUT_PROBE.PASTA_BREAD_PIZZA","OV.
    GUIDED_WITHOUT_PROBE.SAVORY_CAKE_AND_FLAN","OV.
    GUIDED_WITHOUT_PROBE.VEGETABLES"]
    },
    "authInfo":{}
}


CHIAMATA API PER RICHIEDERE LE SOTTO-CATEGORIE

POST //api-iot.example.com/recipe/v1/guided-cooking HTTP/1.1
Content-Type:application/json
Accept:application/json

{
    "applianceType": "OV",
```

```
    "category":"FISH"
    "models": "tad_steam",
    "probe": false
}

Risposta:
{
    "payload": {
        "subcategories":["OV.GUIDED_WITHOUT_PROBE.BAKED_FISH_CARTOCCIO","OV.GUIDED_WITHOUT_PROBE.FISH_FILLET","OV.GUIDED_WITHOUT_PROBE.FROZEN_FISH","OV.GUIDED_WITHOUT_PROBE.GRILLED_FISH","OV.GUIDED_WITHOUT_PROBE.SALT_CRUSTED_FISH","OV.GUIDED_WITHOUT_PROBE.WHOLE_FISH"]
    },
    "authInfo":{}
}


CHIAMATA API PER RICHIEDERE IL PROGRAMMA
POST //api-iot.example.com/recipe/v1/guided-cooking HTTP/1.1
Content-Type:application/json
Accept:application/json

{
    "applianceType": "OV",
    "category":"FISH"
    "subcategories":"BAKED_FISH_CARTOCCIO"
    "models": "tad_steam",
    "probe": false
}

Risposta:
{
    "payload": {
        "program":  {
            "programName":"PROGRAMS.OV.MULTI_LEVEL",
            "prCode_cata_cdm": "23",
            "prCode_pyro_cdm": "23",
            "temperature":200,
            "programTime": 25,
            "models":"tft_base;tft_steam;tad_base;tad_steam;ft_20;vision_20;preco;",
            "preheating":true,
            "suggestions": {
                "suggestion_level": 2,
                "suggestion_accessory":"OV.GUIDED_ACCESSORY.PAN"
            }
        }
    },
```

```
    "authInfo":{}
}
```

code/ch5/APICall.txt

Appendice B

GetGuidedCooking Epica

```
1  const getGuidedCooking: Epic = (action$, state$) => {
2    return action$.pipe(
3      ofType(RecipesActionType.GET_GUIDED_COOKING),
4      withLatestFrom(state$),
5      switchMap(([action, state]) =>
6        defer(() => {
7          // Prendo la richiesta dal payload
8          const requestGuidedCooking = action?.payload as
     GuidedCookingRequest;
9
10         // Chiedo al backend i dati passandogli il payload
11         return CookingMicroservice.getInstance().
     getGuidedCookingData(
12           requestGuidedCooking,
13         );
14       }).pipe(
15         switchMap((response: any) => {
16           const requestGuidedCooking = action?.payload as
     GuidedCookingRequest;
17
18           // Prendiamo la risposta della chiamata API
19           // In base al tipo di chiamata effettuata avremo le
     variabili riempite o no
20           const categories = response.data.payload.
     guidedCookingCategories;
21           const subcategories = response.data.payload.
     guidedCookingSubcategories;
22           const program = response.data.payload.
     guidedCookingProgram;
23
24
25           // Prendiamo i dati che abbiamo da redux e li salviamo
     in un oggetto
```

```
26          // Questo oggetto sarà modificato con le nuove
    informazioni che ha ritornato la chiamata API
27          // Poi verrà risalvato in redux
28          let guidedCookingCategories: GuidedCookingCategory[] =
29            state.guidedCooking.guidedCookingCategories;
30
31          if (!requestGuidedCooking) {
32            // Caso chiamata di primo livello utilizzata per
    prendere le categorie.
33            // Capiamo che siamo nel primo livello perchè questa
    chiamata viene fatta senza payload
34
35
36            // Aggiungiamo le categorie mappandole correttamente
37            guidedCookingCategories = categories.map((category:
    string) => ({
38                title: category,
39                subcategories: [],
40              }));
41
42
43          } else if (!requestGuidedCooking.subcategory) {
44            // Caso chiamata di secondo livello utilizzata per
    prendere le sottocategorie
45
46            // Questa è la categoria di cui abbiamo richiesto le
    sottocategorie
47            const categoryName = requestGuidedCooking.category
48
49            guidedCookingCategories = guidedCookingCategories.map(
50              (category: GuidedCookingCategory) => {
51                // Troviamo la categoria alla quale aggiungere le
    sue sottocategorie
52                if (category.title === categoryName) {
53                  // Aggiungiamo le sottocategorie della chiamata
    API alla categoria
54                  category.subcategories = subcategories.map(
55                    (subcategory: string) => ({
56                      title: subcategory,
57                      program: [],
58                    }),
59                  );
60                }
61                return category;
62              },
63            );
64
65          } else {
```

```
66            // Caso chiamata di terzo livello utilizzata per
    prendere il programma
67
68            // Questa è la categoria di cui abbiamo richiesto le
    sottocategorie
69            const categoryName = requestGuidedCooking.technique
70            const subcategoryName = requestGuidedCooking.
    foodCategory
71
72            guidedCookingCategories = guidedCookingCategories.map(
73              (category: GuidedCookingCategory) => {
74
75                if (category.title === categoryName) {
76                  category.subcategories = category.subcategories.
    map(
77                    (subcategory: GuidedCookingSubCategory) => {
78
79                      if (subcategory.title === subcategoryName) {
80                        subcategory.program = program;
81                      }
82                      return subcategory;
83                    },
84                  );
85                }
86                return category;
87              },
88            );
89          }
90
91          // Lanciamo una azione redux per salvare l'oggetto
    opportunamente modificato
92          return of(storeGuidedCooking(guidedCookingCategories));
93        }),
94        catchError((_err) => {
95          console.warn('Guided cooking epic error: ', _err);
96
97          // Prendo i dati da un json perchè siamo offline oppure
    c'è un problema al backend
98          const data = require('../../../../../assets/localStorage
    /guided-cooking-categories.local')
99            .default;
100
101         return concat(
102                 of(storeGuidedCooking(data.
    guidedCookingCategories)),
103                 of(notifyUserOfflineMode()),
104                 of(loggerCauseError(_err))
105               );
106       }),
```

```
        ),
      ),
    );
};

export default getGuidedCooking;
```

code/ch5/getGuidedCooking.tsx

Bibliografia

[1] *Definizione Treccani di smart house*. 2013. URL: https://www.treccani.it/enciclopedia/smart-house_%28Lessico-del-XXI-Secolo%29/ (cit. a p. 1).

[2] *Corre il mercato della Smart Home: metà degli italiani ha in casa almeno un oggetto smart*. 2013. URL: https://www.osservatori.net/it/ricerche/comunicati-stampa/smart-home-mercato-italiano (cit. a p. 2).

[3] *React Native documentation : Core Components and Native Components*. URL: https://reactnative.dev/docs/intro-react-native-components (cit. a p. 7).

[4] Brad Frost. *Atomic design*. URL: https://bradfrost.com/blog/post/atomic-web-design/ (cit. a p. 9).

[5] *Thinking in React*. URL: https://reactjs.org/docs/thinking-in-react.html (cit. a p. 23).