Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2022/2023
Sessione di Laurea Aprile 2023

Progetto e implementazione di una mobile app cross-platform per l'inventario di cibi e bevande

Relatori:
Giovanni Malnati

Candidati:
Pierluigi Fimiano

Alla mia famiglia e a tutte le persone che mi sono state vicine

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Lo scopo di questa tesi è analizzare e progettare un sistema software per la gestione e il tracciamento di prodotti alimentari distribuiti da un ente centrale alle diverse sedi dislocate sul territorio. Il progetto è stato realizzato grazie alla collaborazione di diversi sviluppatori che hanno contribuito alla progettazione di un'applicazione per dispositivi *mobile*, un sito web e di diversi servizi REST, tramite i quali vengono rese possibili le diverse funzioni. Il progetto è stato commissionato da un'azienda al fine di automatizzare la creazione di inventari per il tracciamento dei prodotti nei magazzini e per calcolare le perdite dovute a furti o smarrimenti. L'idea alla base è stata quella di soddisfare i diversi requisiti, ma al contempo, di realizzare un prodotto enterprise adattabile a diversi clienti e disponibile come servizio.
La gestione di piccole/medie aziende è un processo assai comune oggigiorno e molte realtà attualmente continuano ad affidarsi a mezzi cartacei, difficili da mantenere nel tempo. Questo rende il problema molto complicato e causa una perdita di efficienza e produttività. L'obiettivo di Optima è proprio quella di ottimizzare e automatizzare, tramite sistemi informatici e soluzioni hardware e software, questi processi, incrementando così la produttività delle aziende.
In particolare, il mio contributo alla realizzazione del sistema software è stato quello di prendere parte alla progettazione dell'applicazione mobile disponibile per sistemi operativi Android e iOS.
Nei capitoli successivi verranno analizzati i diversi requisiti raccolti durante la fase iniziale del progetto e le diverse tecnologie e scelte progettuali intraprese durante la fase di sviluppo. In particolare, nei primi due capitoli verranno analizzati il contesto, i requisiti e il problema. Nel terzo e quarto capitolo si scenderà nel dettaglio, descrivendo le tecnologie, le librerie e le scelte tecniche, argomentandole e comparandole alle diverse soluzioni e tecnologie disponibili. Nell'ultimo capitolo, oltre alle conclusioni, verranno descritti possibili miglioramenti e funzionalità future che permetteranno di realizzare un prodotto maturo e pronto per la commercializzazione.

Capitolo 1

Il progetto e l'analisi dei requisiti

Dopo il commissionamento del progetto da parte del cliente, la prima fase di progettazione è stata quella di raccogliere i diversi requisiti che l'applicazione dovrà rendere disponibile agli utenti. Durante la raccolta dei requisiti, si è cercato in una prima stesura di definire le entità convolte nel sistema (sedi, stanze, prodotti e ricette) e i diversi ruoli (amministratore, utente specializzato e manager). Una volta definite le entità ed i ruoli coinvolti sono stati identificati i vari casi d'uso e si è cercato di capire su quale piattaforma li si volesse rendere disponibili.

1.1 La struttura dell'azienda e il problema del tracciamento

L'azienda committente opera nel settore della vendita di alimenti e bevande. Essa è divisa in varie sedi distribuite sul territorio che si occupano di interfacciarsi con i clienti e vendere diversi tipi di cibo, preconfezionato e non, e diversi tipi di bevande. Ogni sede è divisa in stanze nelle quali vengono gestite e immagazzinate le scorte di prodotti.
Ogni sede effettua un inventario giornaliero o settimanale dei prodotti di cui ha bisogno per la realizzazione dei cibi e delle bevande offerte dal menù. Durante questa fase, viene anche tenuta traccia delle scorte residue di prodotti. Gli inventari possono essere di due tipi: inventari standard, nei quali vengono calcolate le quantità residue dei prodotti e inventari delle eccezioni, nei quali viene tenuta traccia dei prodotti persi, come bottiglie rotte o cibi scaduti.
Questo processo viene effettuato attualmente a mano da personale specializzato che si occupa di controllare i prodotti e le scorte residue stanza per

stanza, riportando e mantenendo i dati su carta. Si tratta di un processo molto dispendioso che impiega il personale per diverso tempo e, oltretutto, non essendo automatizzato, è molto comune che vengano commessi errori che comportano perdite dell'efficienza e della produttività. Le diverse sedi non si occupano solamente della vendita di prodotti primi, ma anche di bevande e cibi preparati seguendo ricette che richiedono l'uso di diversi prodotti e bevande. Ogni sede stima la quantità di cibi e bevande di cui ha bisogno settimanalmente e, sulla base delle stime, vengono effettuati gli ordini. Considerando che spesso i prodotti venduti sono preparati usando altri prodotti, il tracciamento delle riserve e delle quantità può diventare un meccanismo molto complicato.
Quello che si vuole migliorare è proprio il processo di tracciamento, automatizzandolo e quindi incrementandone la produttività e l'efficienza. L'azienda è anche interessata ai dati dei prodotti che vengono persi durante la consegna, la gestione, la preparazione e la distribuzione, al fine di ottimizzare i guadagni e diminuire le perdite.
Questo tipo di problema è molto comune nelle aziende che gestiscono prodotti. Infatti, molte realtà come Amazon, eBay, Zalando e altre grandi multinazionali, hanno un sistema di gestione interna realizzato *ad hoc* o si affidano a soluzioni pronte offerte dal mercato. Le aziende più piccole, invece, non possedendo il budget per acquistare soluzioni *ad hoc*, continuano a gestire i processi manualmente. L'obiettivo di questo prodotto è proprio quello di offrire una soluzione alternativa ad aziende medio/piccole, contenendone i costi ed offrendo un prodotto di qualità.

1.2 Le entità ed i ruoli

Dall'analisi dell'organizzazione dell'azienda e dei processi aziendali che si vogliono automatizzare e migliorare, sono state individuate le entità ed i ruoli che ne fanno parte e che andranno poi a caratterizzare il sistema. L'organizzazione dell'azienda può essere vista in maniera gerarchica. Le entità individuate e rappresentate nel sistema sono:

> **Sedi:** compongono l'azienda e sono distribuite sul territorio. Si occupano della composizione, vendita e distribuzione dei prodotti al cliente. La gestione di ogni sede è garantita dalla presenza di personale specializzato che si occupa delle diverse fasi di elaborazione dei prodotti. Ogni sede immagazzina, ordina e traccia i prodotti di cui ha bisogno, oltre a registrare nuove ricette o nuovi prodotti. Ogni sede è suddivisa a sua volta in stanze.

Stanze: sono la suddivisione logica e fisica di una sede. In ogni stanza vengono gestiti prodotti differenti. La suddivisione in stanze rende più semplice e intuitivo il tracciamento dei prodotti e delle scorte.

Prodotti: rappresentano i cibi e le bevande distribuite alle varie sedi. Possono essere venduti direttamente o usati nella preparazione di ricette. Ogni sede potrebbe registrare anche nuovi prodotti non utilizzati che automaticamente diventano ordinabili anche da altre sedi.

Ricette: rappresentano la preparazione di un nuovo cibo o bevanda composta da diversi prodotti. Anche le ricette potrebbero essere condivise da varie sedi.

Inventari: vengono effettuati giornalmente o settimanalmente dal personale delle verie sedi. Servono a tracciare la giacenza e le scorte dei vari prodotti di ogni sede. Ogni inventario viene diviso per stanza e per ognuna di esse vengono registrati i prodotti residui. Esistono anche gli inventari delle eccezioni che permettono di tracciare i prodotti persi.

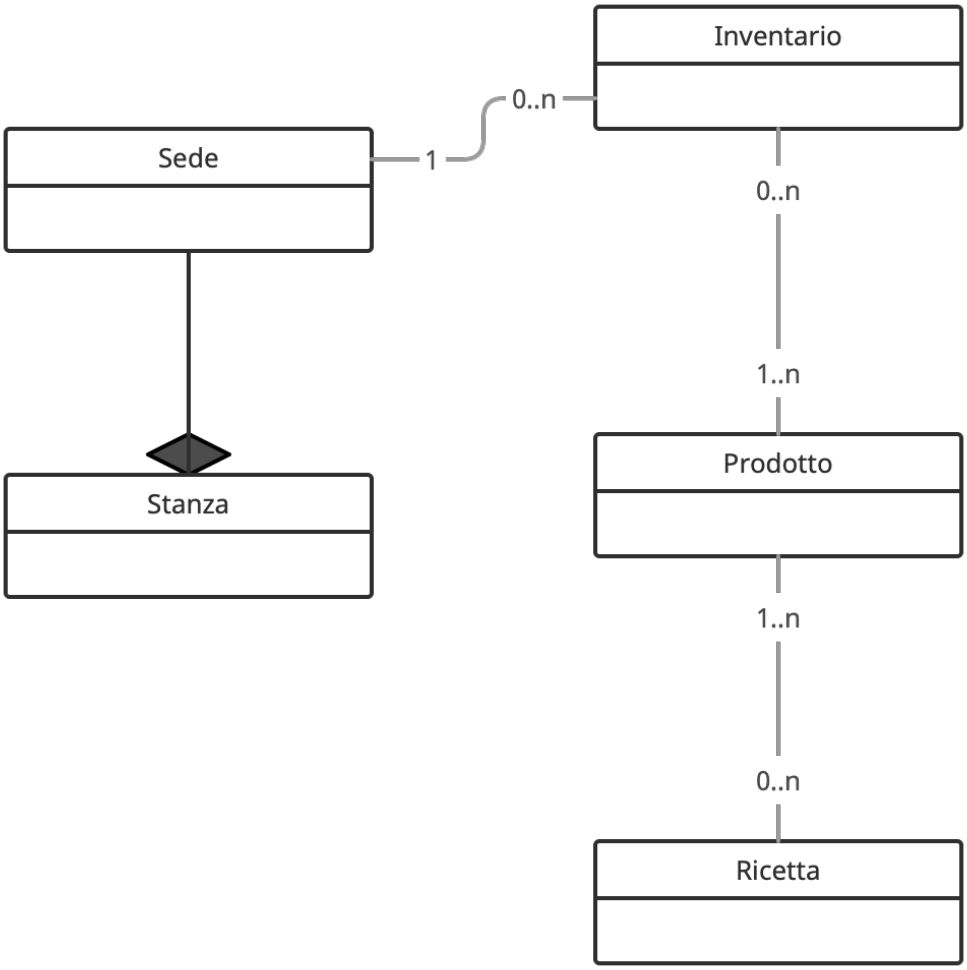

Figura 1.1: Entità del sistema

Il precedente diagramma delle classi definisce le entità del sistema e rispecchia anche la loro disposizione nel database.
Oltre alle entità descritte precedentemente, si vogliono rappresentare nel sistema anche i diversi ruoli coinvolti nel processo. Si può notare come ogni ruolo sia autorizzato ad effettuare una cerchia ristretta di funzionalità. In particolare i ruoli identificati nel sistema sono tre:

> **Amministratore:** si tratta del gestore dell'azienda. Egli ha accesso all'intero sistema, ma in particolare è interessato al tracciamento delle perdite causate dalla distribuzione e lavorazione dei prodotti nelle diverse sedi. Le funzionalità principali solo quelle di creare nuove sedi,

di configurarle e suddividerle in stanze e di effettuare l'estrazione dei dati relativi alle perdite. Alcune funzionalità, come la cancellazione dei prodotti e delle ricette, sono state abilitate solo per amministratore.

Manager: può disporre delle stesse funzionalità dell'amministratore, con la differenza che può accedere solo alle sedi di sua pertinenza. Le sedi di cui si occupa un manager sono definite dall'amministratore in fase di configurazione del sistema.

Utente specializzato: si occupa degli inventari e ha accesso alla loro creazione e gestione. Può registrare nuove ricette e nuovi prodotti e si occupa di tracciare le nuove consegne.

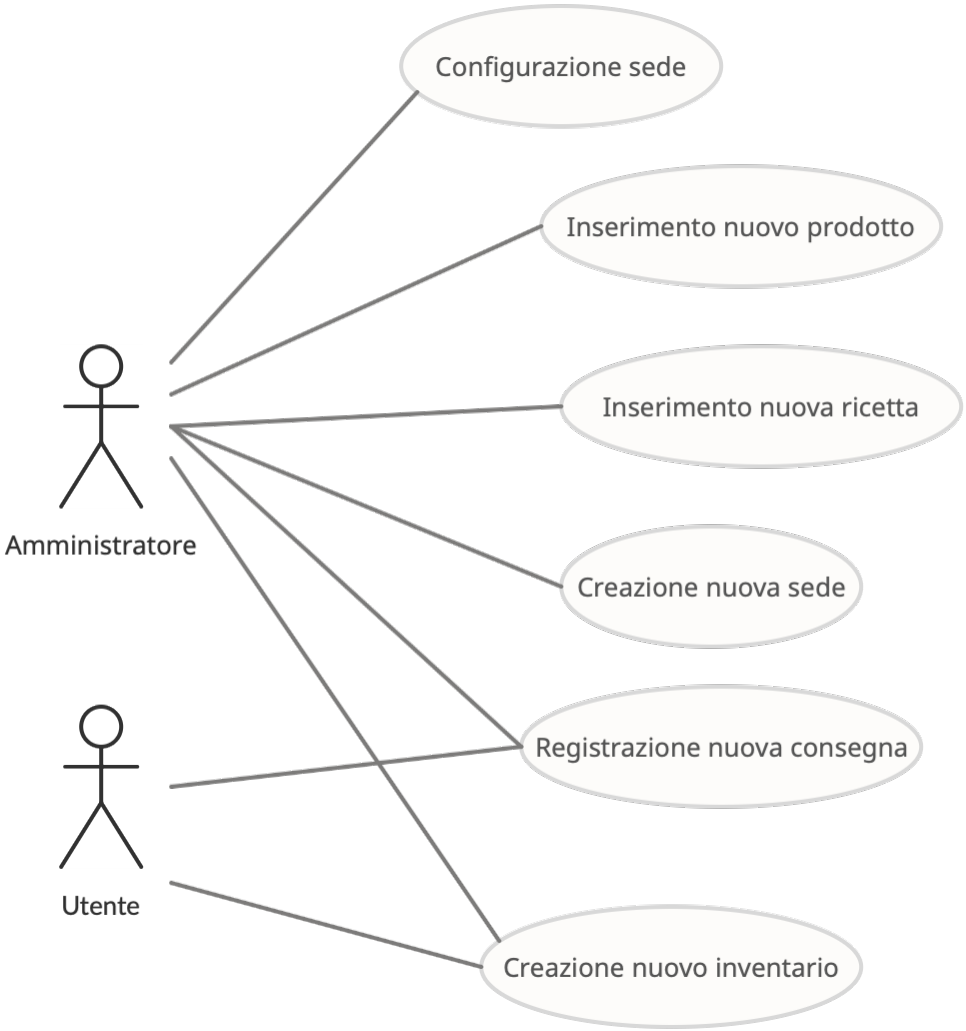

Figura 1.2: Casi d'uso

Il diagramma dei casi d'uso rappresentato in figura, evidenzia le differenti funzionalità alle quali sono interessati gli utenti. Come si può notare, le funzionalità disponibili per l'amministratore sono differenti da quelle alle quali può accedere un utente specializzato. Infatti, l'utente specializzato ha a disposizione un sottinsieme delle funzionalità del sistema. Il ruolo del manager non è stato rappresentato nel diagramma in figura poichè le funzionalità a sua disposizione sono simili a quelle dell'amministratore, ma applicabili ad una cerchia ridotta di sedi.

1.3 I casi d'uso e le funzionalità

Sulla base delle entità e dei ruoli che compongono il processo, sono anche state definite le varie funzionalità e i casi d'uso del sistema. Di seguito verranno analizzate le funzionalità principali suddividendole per ruolo. Le funzionalità disponibili all'amministratore e al manager sono:

Creazione di una nuova sede: nella fase di configurazione, l'amministratore può aggiungere l'elenco delle sedi nel sistema e renderle disponibili e utilizzabili agli utenti.

Configurazione di una sede: ogni sede può essere configurata modificando dati come nome, indirizzo, VAT. In ogni sede possono anche essere aggiunte e rimosse le stanze che la compongono.

Cancellazione prodotto: l'amministratore può eliminare i prodotti dal sistema. Una volta che un prodotto è stato eliminato, non è più visibile ed utilizzabile nella creazione degli inventari o delle ricette.

Cancellazione ricetta: l'amministratore può anche eliminare le ricette che, come per i prodotti, non saranno più utilizzabili e visibili nel sistema.

Le funzionalità disponibili anche agli utenti specializzati invece sono:

Creazione nuovo inventario: è l'operazione effettuata per registrare la quantità di prodotti residui nelle diverse sedi.

Registrazione nuova consegna: effettuate ogni volta che viene consegnata della merce ad una sede. La consegna può essere composta di diversi prodotti in diverse quantità.

Inserimento nuovo prodotto: l'amministratore può inserire nuovi prodotti nel sistema. Dopo l'inserimento di un nuovo prodotto esso sarà visibile e utilizzabile da tutti gli utenti nella creazione di inventari e ricette.

Inserimento nuova ricetta: come per l'inserimento di un nuovo prodotto, l'amministratore può anche creare nuove ricette che diventeranno disponibili e utilizzabili.

Dopo aver definito le funzionalità, si è anche scelto di suddividere il sistema in più applicazioni che permetteranno di accedere al sistema sia da web che da dispositivi *mobile*. L'applicazione web è stata nominata **backoffice** ed

è accessibile solamente all'amministratore. Alcune funzionalità sono state suddivise e rese disponibili solo sul backoffice o solo sull'applicazione *mobile* mentre alcune di esse sono disponibili su entrambe le piattaforme.
Le principali funzionalità disponibili sull'applicazione sono:

- creazione nuovo inventario
- registrazione nuova consegna
- configurazione di una sede
- inserimento nuova ricetta
- inserimento nuovo prodotto

mentre quelle disponibili sul backoffice sono:

- creazione nuova sede
- configurazione sede
- cancellazione prodotto
- cancellazione ricetta

Il backoffice ha principalmente funzionalità di gestione e configurazione del sistema e perciò è stato reso disponibile solo all'amministatore.

Capitolo 2

Il progetto Optima

Per la realizzazione del progetto sono stati definiti diversi componenti software e hardware che interagiscono e comunicano attraverso la rete. Strettamente necessaria è la presenza di un server al quale le applicazioni software possano accedere per recuperare i dati o effettuare modifiche su di essi.
Si è scelto poi di realizzare un'applicazione *mobile* ed un'applicazione web per l'accesso ai dati ed alle funzionalità da diversi dispositivi. Oltre a queste componenti, nel sistema di Optima entrano in gioco altri dispositivi *embadded* come la bilancia bluetooth e il lettore di *barcode*.
Tutti questi dispositivi hardware e applicazioni software comunicano attraverso diversi canali e con diversi protocolli e permettono al sistema di scambiare dati e di realizzare le funzionalità.

2.1 Il server e le API

Il sistema è composto da un server REST sviluppato *ad hoc*. Sul mercato tecnologico esistono diverse soluzioni per sviluppare un server, che vanno dalla progettazione e la realizzazione totalmente personalizzata, all'adozione di soluzioni già pronte disponibili tramite servizi *cloud*.
Per realizzare una soluzione *ad hoc*, bisogna confrontarsi con diverse problematiche come:

- l'acquisto di un dominio DNS pubblico
- l'acquisto di un hardware prestante
- l'accesso ad una rete internet veloce e sempre disponibile

e altre complicazioni, come il consumo di energia e i possibili guasti. Questo tipo di soluzione è molto più complicata da gestire ed implementare rispetto all'utilizzo di una piattaforma *cloud*. La scelta per Optima, infatti, è

ricaduta sull'utilizzo di una piattaforma *cloud* che ha permesso di evitare diverse complicazioni. Le aziende come Amazon, Google e Microsoft offrono a pagamento servizi *cloud* che permettono di rendere le proprie applicazioni accessibili tramite reti pubbliche. Questo tipo di soluzione offre la possibilità di integrare un database con uno spazio dedicato, ma anche di gestire in maniera automatica i DNS e la sicurezza. Esse offrono un ambiente robusto e semplice da utilizzare, ma si tratta di soluzioni costose dal punto di vista economico. Per quanto riguarda Optima, si è scelto di adottare la piattaforma *cloud* di Google che ha permesso di realizzare un server robusto e performante, capace di adattarsi a diverse esigenze e di essere migliorato facilmente in futuri sviluppi.

2.2 L'architettura REST

REST o Representational State Transfer è un'architettura per sistemi distribuiti basata su HTTP (HyperText Transfer Protocol). Il termine fu introdotto nel 2000 da Roy Fielding nella sua tesi di dottorato e negli anni è diventata una delle architetture più diffuse per la progettazione di sistemi distribuiti.
Si tratta di un protocollo stateless che non prevede il concetto di sessione. Questo implica che il server e il sistema non tengano traccia delle applicazioni che lo utilizzano e non mantengano dati di sessione tra due accessi differenti. L'architettura REST supporta le operazioni CRUD (Create, Read, Update e Delete) per l'accesso, la modifica e l'aggiornamento dei dati. Queste operazioni sono disponibili tramite chiamate specifiche e vengono identificate da determinati comandi quali:

GET: è l'operazione che permette di ottenere i dati. Può essere usata per ottenere un singolo dato o una lista.

POST: questa operazione permette l'aggiunta di un nuovo dato/entità o di una lista di dati.

PUT: rappresenta l'operazione di aggiornamento dei dati. Alcune volte viene sostituita dalla POST.

DELETE: permette la cancellazione di una lista di oggetti o di una singola entità.

PATCH, OPTIONS, ecc...: sono operazioni che vengono utilizzate per casi particolari e sono state introdotte in seguito.

Ogni risorsa disponibile sul server può supportare un sottoinsime di queste operazioni che permettono alle applicazioni di interagire con esse. Questo insieme di operazioni, che viene chiamato contratto server-client, è definito solitamente da una documentazione dettagliata del server.
In un server REST, ogni risorsa, la quale può essere un dato, un immagine o qualunque cosa di interesse, è identificata in maniera univoca da un URL. Un URL è una stringa che identifica la risorsa e che deve essere utilizzata in combinazione con l'operazione che si vuole attuare.
Oltre all'URL e all'operazione che si vuole effettuare, le richieste ad un server REST contengono anche un *header* e un *body*. L'header è un insieme di coppie-chiave valore che permette di definire diversi parametri utili al server ed al client. Il body invece rappresenta i dati trasmessi dal client al server. L'architettura REST, essendo basata su HTTP, utilizza messaggi di testo per lo scambio dei dati. Il formato del body invece può essere definito dal progettista del sistema, ma deve essere un formato trasferibile come messaggio di testo.
Un esempio di chiamata REST è il seguente:

```
POST v1/user

HOST: www.host.come
Content-type: application/json
Application-type: application/json

{
    "nome": "Nome"
    "cognome": "Cognome"
}
```

Si può notare che nell'esempio viene mandato un file di testo in cui possiamo identificare l'azione POST, l'url composto da HOST + v1/user, alcuni headers e un body che rappresenta il dato stesso.

2.3 Il formato JSON

Tra i diversi formati disponibili per la rappresentazione dei dati, è stato scelto JSON. JSON o (Javascritp Object Notation) è il formato utilizzato per la rappresentazione degli oggetti nel linguaggio Javascript.
Con l'avvento del web, Javascript ha giocato un ruolo fondamentale nello sviluppo di applicazioni e JSON è diventando uno dei formati più utilizzati per la rappresentazione dei dati. JSON è un formato di testo completamente

indipendente dal linguaggio di programmazione e quindi può essere utilizzato da varie applicazioni o sistemi.
Negli anni JSON è stato standardizzato diverse volte e sono state rilasciate varie versioni con piccole varianti sulla rappresentazione dei dati.
Il seguente è un esempio di rappresentazione JSON di un utente di un sistema.

```
{
    "name: "Name",
    "surname": "Surname",
    "age": 18,
    "city": Turin,
    "identityCard": {
        "id": CA456BG,
        "emissionDate": "18/11/2021"
    },
    "languages": ["Italian", "Spanish"]
}
```

Ogni oggetto JSON inizia con il carattere { e termina con il carattere }.
Un oggetto è un insieme di coppie chiave-valore in cui la chiave deve essere una stringa semplice mentre il valore può essere un tipo primitivo come true/false, stringa e numero o un oggetto complesso come un altro JSON.
JSON permette anche di rappresentare le liste che vengono dichiarate con il carattere [e terminano con].

2.4 L'architettura del sistema

In questa sezione discuteremo più in dettaglio dell'architettura e dei dispositivi che fanno parte del sistema. Come già anticipato, il server REST è un'applicazione in esecuzione su un dispositivo hardware. Esso comunica attraverso la rete con il database il quale è in esecuzione come applicazione a sé. Il database può essere in esecuzione sullo stesso dispositivo dell'applicazione server o su un dispositivo diverso e, in entrambe le soluzioni, le due applicazioni comunicano attraverso un'interfaccia di rete.
Il backoffice, invece, è un applicazione web in esecuzione su un pc o laptop in possesso dell'utente che la usa. Essa comunica con il server attraverso la rete con il protocollo HTTP. Il codice eseguibile del Backoffice si trova su un server web e viene scaricato dal browser client nel momento in cui ci si collega all'url del sito. Da quel momento in poi, trattandosi di una single page application, l'applicazione continua la sua esecuzione sul dispositivo client e comunica con il server attraverso le REST API.

L'applicazione *mobile* viene eseguita su smartphone iOS e smartphone Android. Anche l'applicazione *mobile* comunica con il server attraverso la rete utilizzando il protocollo HTTP e si avvale delle API REST esposte dal server per effettuare le varie operazioni e recuperare i dati. Oltre a comunicare con il server, l'applicazione comunica anche con il barcode reader e la bilancia. Il barcode reader viene collegato al dispositivo *mobile* attraverso l'interfaccia USB che gli permette di essere visto dal sistema operativo come dispositivo di IO direttamente collegato. Per la bilancia invece la comunicazione avviene attraverso il bluetooth. La bilancia è un dispositivo elettronico in grado di eseguire dei programmi e su di essa è in esecuzione un software (firmware) sviluppato dall'azienda produttrice che le permette di eseguire le operazioni e di esporre i dati e le funzionalità sull'interfaccia bluetooth. L'applicazione *mobile* comunica attraverso l'interfaccia bluetooth con il firmware della bilancia. I dettagli sui protocolli di comunicazione sono oscurati dal software proprietario e l'applicazione utilizza l'SDK per effettuare le operazioni.

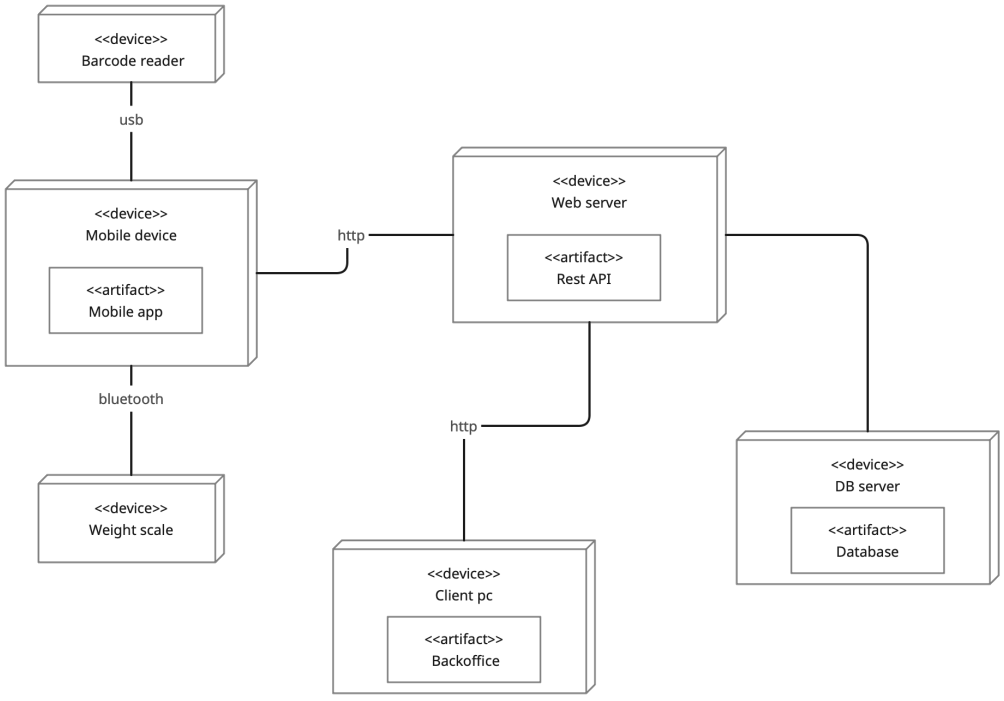

Figura 2.1: Deployment diagram

Il deployment diagram visibile in figura mette in risalto le componenti del sistema e le varie interfacce sulle quali scambiano dei dati. Nel diagramma il

database è stato rappresentato in esecuzione su un dispositivo differente dal server, ma tale rappresentazione non cambia l'architettura del sistema. Non è stato rappresentato nel diagramma il web server che espone l'url dal quale viene scaricato il codice del backoffice.

Capitolo 3

Sviluppo e progettazione applicazione mobile

Le applicazioni mobili sono software progettati per essere eseguiti su dispositivi mobili come smartphone, smartwatch e tablet.
Tali dispositivi, solitamente, hanno a disposizione risorse hardware limitate come RAM, CPU e memoria di massa e consumi energetici ridotti, dovuti al fatto che, in generale, la loro alimentazione avviene tramite batterie. Questo implica tecniche di progettazione del software differenti dalle classiche applicazioni per PC, inoltre, i sistemi operativi *mobile*, come Android e iOS, costringono lo sviluppatore a rispettare determinati vincoli di sicurezza, consumo e accesso ai dati.
Oltre ai vincoli hardware dovuti alle caratteristiche del sistema, anche la progettazione dell'interfaccia grafica è differente. Esistono dispositivi con schermi di forma, dimensione e risoluzione differente e l'interfaccia grafica delle applicazioni software deve essere progettata per potersi adattare ai differenti formati disponibili sul mercato.
Altro aspetto importante da valutare durante la progettazione e il design di un applicativo *mobile* è l'esperienza utente e la fluidità con cui il software reagisce alle interazioni dell'utente. Le applicazioni di successo sono dotate solitamente di interfacce grafiche intuitive, reattive e accattivanti che fanno sì che l'utente interagisca in maniera efficace con il sistema.
In generale, nei sistemi *mobile*, il disegno e l'interazione utente vengono gestite da un singolo thread chiamato main thread. Il main thread deve occuparsi di disegnare ogni frame dell'applicazione e di reagire al contempo agli input esterni. Questo fa sì che non si possano eseguire operazioni pesanti come lettura di dati dall'IO o CPU intensive tasks nel main thread al fine di non generare rallentamenti nell'interfaccia utente. I sistemi *mobile* hanno definito API per semplificare la programmazione asincrona per l'esecuzione di questo

tipo di operazioni.
L'applicazione Optima è stata progettata per poter essere compilata ed eseguita sia su sistema operativo Android che su sistema operativo iOS. Le caratteristiche dei due sistemi operativi sono differenti ed esistono diverse tecnologie e linguaggi per poter progettare e sviluppare applicazioni su di essi.

3.1 Android

Il sistema operativo Android è nato come sistema open source basato sul sistema operativo Linux che è stato adattato per l'utilizzo su hardware con risorse limitate.
Tutte le applicazioni Android vengono create a partire da un processo Linux creato dal sistema operativo. Gli unici processi che vengono eseguiti con i diritti dell'utente principale sono quelli del sistema operativo mentre tutti gli altri, compresi i processi delle applicazioni, vengono eseguiti con i diritti dello sviluppatore che ha pubblicato l'applicazione sul Google Play Store. Questo implica che ogni applicazione al suo avvio abbia permessi limitati garantendo una maggiore sicurezza al sistema e all'utente. Per richiedere permessi come l'accesso alle periferiche di IO o l'accesso alla rete, le applicazioni devono dichiarali esplicitamente all'interno del Manifest. Il Manifest è un file XML all'interno delle applicazioni che viene letto dal sistema operativo e contiene diverse informazioni sull'applicazione, l'autore e i permessi richiesti.
All'avvio del processo applicazione, il sistema operativo crea un thread principale, detto main thread, che si occupa di instanziare tutti gli oggetti e di gestire l'interfaccia grafica e gli eventi principali. Il main thread possiede una coda di eventi detta MessageQueue tramite la quale è possibile notificare nuove operazioni. La coda viene riempita sia dagli eventi generati dall'utente sia da eventi generati dal codice. Essendo la natura stessa della coda sequenziale, il blocco di un task nella coda causerebbe ritardi nell'esecuzione degli altri task. Per questo motivo non è possibile eseguire operazioni pesanti nel main thread che causerebbero lo stop dell'applicazione da parte del sistema.
Alla sua creazione, il main thread analizza il Manifest e da esso estrae le informazioni per generare l'oggetto software Application e per creare l'Activity principale. L'oggetto Applicazione è opzionale e, se non specificato esplicitamente, il sistema operativo provvede a generarne uno di default.
Le Activity invece sono alcuni dei componenti principali di un'applicazione Android. Esse rappresentano gli oggetti in carico di gestire l'interfaccia grafica e l'interazione con l'utente. Le Activity, come i Service, i ContentProvider e i BroadcastReceiver, posseggono un ciclo di vita interamente gestito dal si-

stema operativo. Il ciclo di vita di un'activity viene gestito dagli sviluppatori tramite callback. Il principale ciclo di vita di un activity è:

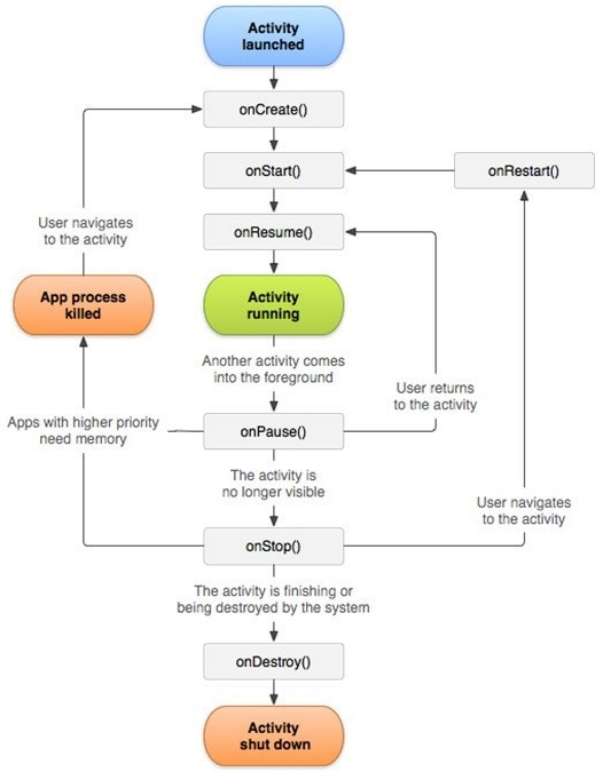

Figura 3.1: Ciclo di vita di un'activity

Come si può notare, alcuni stati di un'activity si possono ripetere nel tempo. Ad esempio, se l'applicazione viene mandata in background vengono chiamate le callback onPause(), onStop() e quando ritorna in foreground le callback onStart(), onResume(). Nel caso di rotazione dello schermo, invece, l'activity viene distrutta e ricreata dal sistema. Questo comportamento rende difficile il salvataggio dello stato dell'applicazione che deve essere gestito dallo sviluppatore tramite l'accesso al Bundle, salvando lo stato temporaneamente su una partizione riservata della memoria di massa.
Oltre alla gestione dello stato, l'activity si occupa di creare l'interfaccia grafica che può essere definita tramite file XML (eXtensible Markup Language) e creata nell'onCreate().
Tutti i processi Android vengono eseguiti all'interno di una versione custom della JVM (Java Virtual Machine) che ha permesso a Java di diventare il linguaggio principale per lo sviluppo di applicazione Android. Con l'avvento di

Kotlin, nuovo linguaggio interoperabile con Java, la popolarità di Java è diminuita e sempre più sviluppatori adottano Kotlin come linguaggio principale per le loro applicazioni.

3.1.1 Kotlin

Kotlin è un linguaggio ad oggetti sviluppato da JetBrains. La caratteristica di essere interoperabile con Java e di essere in grado di generare bytecode eseguibile su JVM gli ha permesso di sostituire Java come linguaggio principale nello sviluppo di applicazioni Android. Si tratta di un linguaggio fortemente e staticamente tipizzato e ad oggetti che introduce importati novità rispetto a Java.
Kotlin, a differenza delle versioni precedenti di Java, supporta il concetto di *type inference* che permette al compilatore di identificare i tipi delle variabili dall'espressione di assegnazione, evitanto allo sviluppatore di doverli dichiarare esplicitamente.
Altra importante novità introdotta dal linguaggio è il concetto di programmazione funzionale. Questo permette di trattare le funzioni come oggetti e quindi di usarle come parametri o tipi di ritorno in altre funzioni. Il concetto di programmazione funzionale permette di elaborare i dati con una sintassi intuitiva e concisa.
Kotlin ha anche ridefinito la gerarchia dei tipi rispetto al Java introducendo il concetto di *null-safety*. Ogni tipo in Kotlin eredita dal tipo Any (rappresenta il tipo Object di Java), ma a cui non possono essere assegnati valori *null*. Per dichiarare un tipo opzionale, bisogna aggiungere alla dichiarazione del tipo il carattere ? che permette, ad esempio, al tipo Any? di contenere valori null. Questo importante concetto permette al compilatore di eseguire dei controlli in fase di compilazione che evitano errori ed eccezioni (come NullPointerException) in fase di esecuzione del programma.
Oltre ad altre novità come lambda function, inline function e sealed class, Kotlin ha introdotto un nuovo concetto di programmazione asincrona, le Coroutines, che semplifica di molto la gestione delle applicazioni Android. Le Coroutines sono supportate dalla versione 1.1 e permettono la gestione di task asincroni in maniera semplificata, introducendo anche il concetto di cancellazione. Una Coroutine può essere vista come un light-weight thread gestito come una macchina a stati. Per dichiarare una coroutine bisogna per prima cosa definire lo scope della coroutine stessa. Uno scope può essere visto come un oggetto capace di creare una coroutine ed è composto da un ExceptionHandler, un Job e un Dispatcher. L'ExceptionHandler si occupa di gestire le eccezioni che possono essere generate nell'esecuzione, il Job tiene traccia di tutte le coroutines generate e permette di attuare il meccanismo

della cancellazione mentre il Dispatcher definisce il contesto di esecuzione della coroutine stessa. Per creare una nuova coroutine, il CoroutineScope mette a disposizione due differenti metodi:

async() che crea una coroutine che termina l'esecuzione con un risultato. Per recuperare il risultato si può chiamare sull'oggetto Deferred <T> restituito dalla funzione async() il metodo await(). Il pattern è molto simile al pattern Future-Promise, ma con diverse migliorie e ottimizzazioni.

launch() crea una coroutine che non ritorna un risultato. L'oggetto restituito dalla funzione launch() è un nuovo Job che è legato solamente alla nuova coroutine. Tramite l'oggetto Job si può cancellare l'esecuzione con il metodo cancel() o aspettare la fine con il metodo join().

Con le Coroutines il linguaggio introduce anche il concetto di suspend function. Una suspend function viene dichiarata anteponendo la keyword suspend e può essere chiamata solamente all'interno di un'altra suspend function o all'interno di una coroutine. La dichiarazione di una suspend function permette al compilatore di intendere che l'esecuzione della funzione è asincrona o bloccante e quindi che il risultato della funzione non sarà disponibile direttamente. Nel caso di una suspend function il compilatore modifica il codice e introduce l'oggetto Continuation che permette al thread di non bloccarsi nell'esecuzione in attesa del risultato, ma di sospendersi, eseguire altre operazioni e poi riprendere l'esecuzione nel momento in cui la funzione ha terminato la sua esecuzione. Questo importante concetto ha permesso di semplificare molto la programmazione asincrona e di dichiarare e scrivere codice asincrono come se fosse codice sequenziale.

3.2 iOS

La prima versione di iOS è stata presentata da Apple nel 2007 con la commercializzazione del primo iPhone. Nel corso degli anni Apple ha pubblicato diversi varianti del sistema operativo, come watchOS, iPadOS e tvOS. iOS è il secondo sistema operativo più diffuso per dispositivo *mobile* dopo Android ed è un sistema operativo proprietario di Apple basato su piattaforma Linux/Unix.
Anche iOS, come Android, ha definito componenti del sistema operativo per la gestione delle risorse limitate disponibili sui sistemi *mobile* e il risparmio energetico. Tutte le applicazioni che vengono eseguite sul sistema operativo

di Apple devono rispettare alcuni requisiti di sicurezza e vengono eseguite in una *sendbox*, la quale isola l'esecuzione dell'applicazione, non permettendole l'accesso alle risorse hardware e software e ai dati dell'utente. Anche su iOS, per poter utilizzare risorse protette come la fotocamera o accedere alle periferiche IO, bisogna dichiarare i permessi all'interno di uno specifico file. All'avvio dell'applicazione, il sistema operativo mostra un messaggio all'utente richiedendo il permesso specifico e dando anche una breve spiegazione del motivo per cui tale permesso viene richiesto.
Quando l'applicazione viene avviata, vengono generati un nuovo processo e un thread che si occuperà di gestire gli eventi utente e il disegno della UI. Il main thread, oltre ad occuparsi di gestire gli oggetti iniziali e la configurazione, si occupa di creare l'oggetto applicazione. Come in Android, anche in iOS l'oggetto applicazione e le schermate grafiche vengono gestite dal sistema operativo e posseggono un ciclo di vita specifico. Il programmatore può controllare lo stato della view tramite l'accesso a diverse callback messe a disposizione e chiamate dal sistema operativo stesso. Il meccanismo di funzionamento del main thread di iOS è molto simile a quello Android e per tale motivo, anche su iOS non è possibile eseguire operazioni pesanti, per non rallentare l'esecuzione dell'applicazione stessa. Anche Apple ha previsto componenti del sistema operativo per supportare la programmazione asincrone di cui discuteremo in seguito.

3.2.1 Swift

Quando iOS è stato rilasciato, il linguaggio principale con cui venivano sviluppate le applicazione era Objective-C. Nel 2014 Apple ha rilasciato Swift, un linguaggio di programmazione orientato agli oggetti e disponibile per iOS, iPadOS, watchOS, macOS, tvOS e Linux, che ha sostituito negli anni Objective-C per lo sviluppo di applicazioni su sistemi Apple.
Si tratta di un linguaggio di programmazione maturo che supporta lambda function, functional programming, null-safety ed è compatibile con Objective-C. Questo permette agli sviluppatori di sviluppare applicazioni ibride che contengono codice misto.
Swift non supporta il concetto di thread, che è stato sostituito delle code asincrone chiamate Dequeue. Per eseguire del codice asincrono bisogna creare una Dequeue specificando nel costrutto il Dispatcher nel quale verrà eseguita la coda. Sulla coda possono essere create delle callback le quali vengono accodate ed eseguite in seguito dal thread che ha in gestione la coda.
Con la versione iOS 13, Apple ha voluto introdurre a supporto della programmazione asincrona il pattern Future-Promise. Dalla versione di iOS 13 sono disponibili in Swift le keyword *async* e *await*. Le funzioni che vengono

dichiarate con la keyword async sono funzioni che vengono eseguite in maniera asincrona e il loro risultato può essere ottenuto anteponendo ad esse la keyword await. La keyword await può essere chiamata solamente all'interno di altre funzioni async.
Apple ha anche rilasciato un sopporto nativo per la programmazione reattiva introducendo il concetto di Stream e AsyncStream. Gli Stream sono oggetti capaci di emettere un flusso di eventi sui quali possono essere applicate operazione anche in maniera asincrona. Prima della versione 13, la programmazione reattiva su iOS non era supportata e l'utilizzo delle code (Dequeue), risultava difficile e di poca comprensione. Questo ha permesso la diffusione nella community degli sviluppatori iOS l'utilizzo della libreria RxSwift.
RxSwift è la versione compatibile e utilizzabile in Swift del progetto ReactiveX, che include il supporto alla programmazione reattiva per diversi linguaggi di programmazione. Gli oggetti principali di Rx sono i Subject, gli Observable e gli Observer. I Subject sono simili agli Observable, ma tramite essi possono essere emessi eventi anche da codice esterno al costruttore. Gli Observer invece sono delle callback registrabili su un Subject o su un Observable semplice e che vengono eseguite nel momento in cui viene emesso un evento. RxSwift, oltre a definire i componenti principali, definisce un insieme di funzioni e operatori che possono essere applicati agli observable per elaborare i dati al loro interno.

3.3 La scelta tecnologica

Scegliere il linguaggio/framework per la realizzazione di un'applicazione *mobile* tra le varie tecnologie esistenti richiede l'analisi di diversi aspetti e caratteristiche. Le possibilità sono varie e possono essere raggruppate in due principali categorie: sviluppo nativo e sviluppo cross-compilabile.
Per sviluppo nativo si intende l'utilizzo di linguaggi e framework dipendenti dalla piattaforma e che permettono di realizzare applicazioni eseguibili esclusivamente su un determinato sistema operativo. Lo sviluppo nativo ha diversi pro a suo favore come il supporto di una vasta community e della documentazione ufficiale di Google o Apple e la possibilità di accedere direttamente alle API di sistema. Questo permette di creare un'applicazione con un'interfaccia grafica in sintonia con il sistema stesso. Lo sviluppo nativo però richiede il progetto e lo sviluppo di due applicazioni differenti e l'utilizzo di tecnologie e linguaggi differenti come Swift/Objective-C per iOS e Kotlin/Java per Android. Questo implica un maggiore effort nello sviluppo e realizzazione dell'applicazione e nella risoluzione dei bugs e quindi anche un *budget* superiore dovuto al maggior numero di progettisti richiesti.

Esistono anche tecniche cross-compilabili come React Native e Flutter che permettono di sviluppare l'applicazione utilizzando un solo linguaggio/framework ed eseguirla su entrambi i sistemi. Questo permette di realizzare un'applicazione con un numero inferiore di sviluppatori e di effort, ma richiede lo studio di nuovi linguaggi e, solitamente, la documentazione è meno sviluppata. Oltretutto, alcune API di sistema e librerie esterne non sono disponibili e quindi tali tecnologie permettono di interfacciarsi al codice nativo.
La tabella seguente riassume le principali caratteristiche distintive dei linguaggi per lo sviluppo *mobile*.

	iOS	Android
Swift/Objective-C	x	
Kotlin/Java		x
React Native	x	x
Flutter	x	x

Tabella 3.1: Confronto tra le tecnologie

Visto che l'applicazione deve essere disponibile sia per sistemi Android che iOS la scelta è ricaduta nell'utilizzo di una tecnologia cross-compilabile. Al fine di scegliere il framework migliore, React Native e Flutter sono stati messi a confronto sotto diversi aspetti e prestazioni.
React Native è un framework sviluppato da Facebook ed è stato ripreso dal pre esistente React utilizzato nello sviluppo Web. La principale caratteristica di React Native è che il codice viene intepretato da un processo software che lo traduce in codice nativo. Questo fa si che le applicazioni sviluppate in React Native risultino differenti tra iOS e Android dato che vengono poi utilizzati dall'interprete i componenti nativi del sistema.
Flutter invece è un framework sviluppato e supportato da Google. La caratteristica principale è che ogni componente è stata ridisegnata e implementata con Flutter. Flutter richiede al sistema operativo un Canvas sul quale poi va a disegnare le componenti grafiche. I grafici successivi mettono a confronto le prestazioni delle varie tecnologie sui due sistemi operativi utilizzando diversi software.

Tabella 3.2: Confronto delle prestazioni

Dai grafici possiamo notare che le prestazioni di Flutter non sono molto distanti da quelle di un'applicazione nativa, mentre quelle di React Native sono molto più lente. Questo è dovuto al diverso processo di rendering dei componenti grafici. Mentre React Native fa da interprete tra il codice e i componenti nativi, il codice Flutter disegna direttamente su un canvas di sistema e questo permette di aumentare di molto le prestazioni.
La scelta quindi è ricaduta su Flutter, scrivendo poi parte del codice in linguaggi nativi quando le API non erano disponibili.

3.4 Flutter e il linguaggio Dart

Flutter è un framework sviluppato e supportato da Google che utilizza il linguaggio Dart e permette di sviluppare software *mobile* e web.
Il linguaggio Dart è stato sviluppato da Google e presentato nel 2011 con lo scopo di sostituire JavaScript nello sviluppo web. Successivamente Google ha rilasciato Dart native che permette a Dart, non solo di essere compilato e tradotto in JavaScript, ma di essere compilato ed eseguito su dispositivi mobili. Si tratta di un linguaggio tipizzato e ad oggetti, progettato per semplificare lo sviluppo di interfacce grafiche e per migliorare l'efficienza in tempo di esecuzione.

Essendo un linguaggio open source è supportato da una vasta community di sviluppatori e di librerie di terze parti.

3.4.1 I Widgets

I componenti principali di un'applicazione Flutter sono i widgets. Un widget è un oggetto software che estende le classi StatefullWidget o StatelessWidget e rappresenta un componente grafico. Al fine di rendere i widgets riutilizzabili in diverse parti dell'applicazione, è possibile definire delle proprietà da passare attraverso il costruttore e che permettono di cambiare le caratteristiche del widget.
In generale esistono due tipologie di widgets:

Stateless widget

Si tratta di componenti grafici che non posseggono uno stato interno e possono essere controllati solo tramite le proprietà definite dal costruttore. Ogni stateless widget deve estendere la classe StatelessWidget definita nel framework e deve implementare il metodo Widget build(BuildContext context). Tramite questo metodo viene definita l'interfaccia grafica e la struttura del componente software.

```
class GreenFrog extends StatelessWidget {
    const GreenFrog({ super.key });

    @override
    Widget build(BuildContext context) {
        return Container(
            color: const Color(0xFF2DBD3A),
        );
    }
}
```

Stateful widget

A differenza degli stateless widgets, questi componenti grafici posseggono uno stato interno che mantengono nel tempo. Per definire uno stateful widget bisogna estendere la classe StatefulWidget e definire una classe stato che estende State<T extends StatefulWidget>.
La classe principale si occupa solo di generare lo stato del widget, mentre la classe State permette di definire i componenti, il design e lo stato stesso del widget. Al fine di aggiornare uno stato bisogna chiamare la funzione

setState{} che informa il framework che lo stato è cambiato e che è richiesta una ricomposizione.
Le applicazioni Flutter vengono generate a partire dal metodo main che si occupa di creare il widget principale dell'app. Ogni volta che una proprietà o lo stato di un widget cambia, il framework calcola la differenza tra il frame precedente e il nuovo, chiamando le funzione build e aggiorna la UI.

```
class YellowBird extends StatefulWidget {
    const YellowBird({ super.key });

    @override
    State<YellowBird> createState() =>
        _YellowBirdState();
}

class _YellowBirdState extends State<YellowBird> {
    @override
    Widget build(BuildContext context) {
        return Container(
            color: const Color(0xFFFFE306),
        );
    }
}
```

3.4.2 Programmazione asincrona e reattiva

Le applicazioni Flutter vengono eseguite in un thread principale detto *main thread*. Il main thread si occupa di generare i componenti grafici e software dell'applicazione, di disegnarli e di reagire agli input dell'utente. Il funzionamento del main thread è gestito tramite una coda di eventi. Ogni nuovo evento generato del sistema o dall'utente viene inserito nella coda e eseguito in seguito dal main thread.
Un evento può rappresentare diverse azioni generate dall'utente come il click di un bottone o lo swipe sull'interfaccia grafica, ma anche eventi generati a livello software come cambiare lo stato di uno stateful widget.
Essendo il main thread l'unico thread a gestire il disegno dell'interfaccia grafica e la reazione agli eventi utente e di sistema, un eventuale task bloccante eseguito nel main thread farebbe perdere reattività e fluidità all'applicazione stessa causando un'esperienza utente non adeguata. Al fine di evitare tali operazioni, Flutter supporta diversi pattern e tecniche per la programmazio-

ne asincrona e reattiva allo scopo di eseguire operazioni pesanti al di fuori del main thread e di notificare il main thread al termine di esse.

Il pattern Future-Promise

Dart supporta nativamente il pattern Future-Promise tramite le keyword **async** e **await**. Ogni funzione che viene dichiarata tramite la keyword async ha come tipo di ritorno un oggetto Future<T> che rappresenta il risultato di un'operazione asincrona. Una funzione asincrona può a sua volta invocare altre funzioni asincrone e attendere il risultato di tali operazioni anteponendo all'oggetto Future<T> la keyword await. Questo permette di bloccare l'esecuzione della funzione attuale e attendere il risultato T della funzione asincrona invocata. Data la natura bloccante della keyword await, essa può essere invocata solamente all'interno di funzione async.

```
Future<String> createOrderMessage() async {
    var order = await fetchUserOrder();
    return 'Your order is: \$order';
}

Future<String> fetchUserOrder() =>
    // Imagine that this function is
    // more complex and slow.
    Future.delayed(
        const Duration(seconds: 2),
        () => 'Large Latte',
    );

Future<void> main() async {
    print('Fetching user order...');
    print(await createOrderMessage());
}
```

Il codice precedente è un esempio di utilizzo delle keyword async await e dell'oggetto Future<T>, al fine di effettuare operazioni lunghe senza bloccare il main thread.

Gli Stream

Altra caratteristica supportata dal linguaggio è la programmazione reattiva e quindi la scrittura di codice reattivo basato su eventi e flussi di dati.

La programmazione reattiva è un'evoluzione del pattern Observable permettendo di eseguire il codice su threads differenti. Un Observable è un oggetto software capace di emettere eventi che poi vengono consumati da un oggetto Observer appositamente registrato. In Flutter, il supporto alla programmazione reattiva è implementato tramite gli Stream<T>. Uno stream è un oggetto capace di emettere in maniera asincrona un flusso di dati. Per reagire agli eventi generati da uno stream, possiamo chiamare la funzione subscribe che ci permette di registrare una callback che viene eseguita ad ogni nuovo evento. Gli Stream possono essere generati tramiti appositi costruttori della classe Stream o attraverso la keyword **async***, mentre si possono emettere eventi tramite le keywords **yield** o **yield***.

```
Future<int> sumStream(Stream<int> stream) async {
    var sum = 0;
    await for (final value in stream) {
        sum += value;
    }
    return sum;
}

Stream<int> countStream(int to) async* {
    for (int i = 1; i <= to; i++) {
        yield i;
    }
}

void main() async {
    var stream = countStream(10);
    var sum = await sumStream(stream);
    print(sum); // 55
}
```

Il codice precedente è un esempio di creazione e consumo di uno Stream in Dart.
La programmazione reattiva è uno strumento importante per la progettazione e implementazione di interfacce grafiche poichè ogni evento come interazione utente o eventi di sistema possono essere rappresentati tramite Streams.

3.4.3 La struttura di un progetto Flutter

Dopo aver installato il framework Flutter, un nuovo progetto può essere generato tramite il comando **flutter create projectname**. Questo comando genera una cartella con all'interno diversi file e packages. Le principali cartelle in un progetto Flutter native sono:

lib: all'interno della quale è contenuto il codice Dart/Flutter che rappresenta l'applicazione software.

android: è un progetto Android nativo all'interno del quale può essere sviluppato codice nativo Android.

ios: è il codice nativo iOS che può essere utilizzato per generare codice iOS nativo.

L'installazione di Flutter aggiunge anche un comando per le gestione delle dipendenze **flutter pub** che permette di installare, aggiornare e rimuovere librerie esterne e dipendenze. Il file di configurazione al quale fa riferimento flutter pub è il file **pubspec.yalm** all'interno del quale vengono inserite tutte le dipendenze e la versioni dell'app.

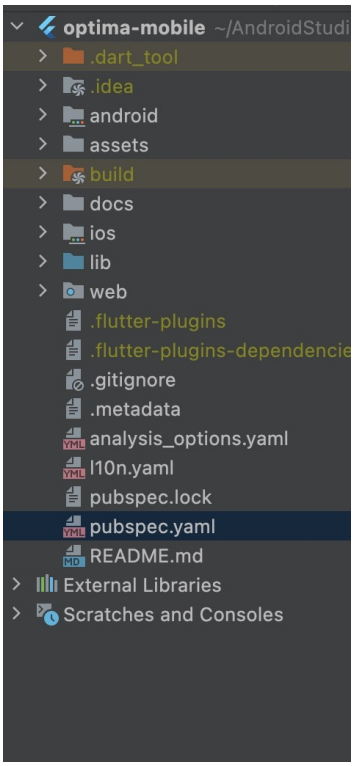

Figura 3.2: Un esempio di progetto Flutter

Le applicazioni vengono generate tutte a partire dalla funzione main che si occupa di creare il widget principale, gli oggetti iniziali e le configurazione. In generale, un'applicazione Flutter è un insieme di stateless e stateful widgets, visualizzati sull'interfaccia grafica.
Il widget principale di un'applicazione è solitamente uno stateless widget creato a partire da un AppWidget presente nel framework. Tale widget, oltre a definire le varie proprietà dell'app, permette anche di definire una lista di routes rappresentate da una mappa. Ogni route definisce un nuovo screen nell'applicazione e la key ad essa associata permette al navigation controller di gestire quale screen/route è attualmente visibile o sulla quale si vuole navigare.
Altro aspetto importate è la gestione delle stringhe. Esse vengono definite tramite un file Json presente per ogni lingua e contenente per ogni stringa una coppia chiave-valore. Flutter genera automaticamente un oggetto **AppLocalizations** che permette di recuperare la traduzione di una stringa tramite la key associata.

3.5 Git

Per gestire il codice sorgente dell'applicazione è stato utilizzato un repository remoto su Bitbucket e Git. Git è un SVC (software version control), ovvero uno strumento software per la gestione del codice sorgente e per tracciare le modifiche apportate ad esso nel tempo. L'utilizzo di Git ha permesso di facilitare l'accesso al codice, il tracciamento delle modifiche e anche la collaborazione di diversi sviluppatori nella fase di sviluppo.
Git è un software open source disponibile e installabile su Windows, Linux e MacOS e mette a disposizione dello sviluppatore diversi comandi eseguibili attraverso una shell o attraverso diverse applicazioni compatibili che permettono di accedere a Git attraverso un'interfaccia grafica. Nel momento in cui si abilita Git su un progetto (attraverso il comando git init) viene creata una repository locale sulla macchina dello sviluppatore che rappresenta un nuovo progetto in Git. Alla creazione della repository viene anche generato il primo *branch* che è un puntatore all'interno di una repository git ad una versione specifica del codice sorgente anche detto commit.
Ogni nuova modifica ad un file, per essere salvata, deve essere aggiunta ad un commit. Ogni commit contiene le modifiche che sono state effettuate al codice rispetto al commit precedente, un identificativo univoco detto *hash*, un messaggio inserito dallo sviluppatore come commento delle modifiche e una lista di informazioni quali nome, cognome e email dello sviluppatore che l'ha creato. Attraverso il meccanismo dei branch è possibile generare commit che

hanno come parent uno stesso commit. Questo permette di generare dei rami indipendenti nella repository e quindi di poter lavorare parallelamente sullo stesso codice evitando di creare conflitti. Una volta terminate le modifiche, esse possono essere riportate sul branch principale attraverso l'operazione di *merge*.

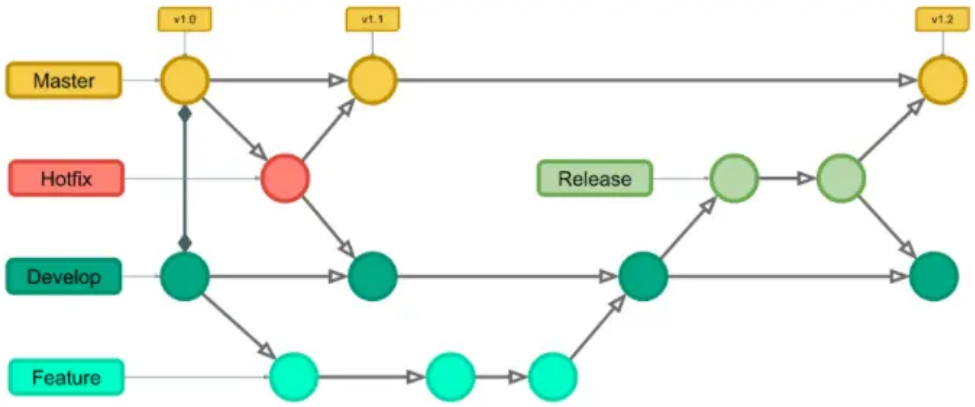

Figura 3.3: Git flow

La figura precedente rappresenta il Git flow adottato durante lo sviluppo del software. Il Git flow rappresenta la strategia seguita dagli sviluppatori al fine di garantire lo sviluppo parallelo e la correttezza del codice. Il codice principale è stato mantenuto sul branch master dal quale vengono anche generate le release dell'applicazione. Gli sviluppi invece vengono salvati temporaneamente su develop e, quando si decide di includerli nella successiva release, attraverso l'operazione di merge vengono portati su master. Per garantire la correttezza del codice, ogni sviluppatore prima di iniziare un nuovo sviluppo, crea un nuovo branch a partire da develop. Alla fine degli sviluppi, viene fatta un review del codice e vengono eseguiti i test per garantire la correttezza. Se i test e la review vengono approvati, il branch e tutte le modifiche che contiene vengono portati su develop. Esistono anche altri tipi di branch, come gli hotfix, che però vengono utilizzati solamente in situazioni particolari, come la soluzione di un bug in produzione che deve essere rilasciata velocemente.

3.6 L'architettura di Optima

Nel progetto e realizzazione di un'applicazione *mobile*, gioca un ruolo molto importante anche la scelta di un'architettura software adeguata. Dopo l'analisi e il confronto di diverse architetture, è stato scelto di adottare per Optima il pattern architetturale MVC o Model-View-Controller. MVC è un pattern architetturale che permette di separare i componenti grafici e la logica per rappresentarli/aggiornarli dai dati e dalla logica di business. I componenti principali del pattern MVC sono:

View: rappresenta l'interfaccia grafica utente, compresi componenti grafici come bottoni, icone, menu e caselle di testo, ma anche temi, colori e fonts. All'interno della view può anche essere contenuta della logica che interessa l'aggiornamento della UI (come il cambio di un colore), ma che non interferisce con processi di business. Gli eventi che generano o includono processi di business vengono affidati dalla view al Controller di cui la view mantiene un riferimento.

Controller: è l'oggetto che la view utilizza per richiamare o eseguire processi di business o per aggionare i dati. Solitamente il Controller espone un'interfaccia alla View che risulta in un contratto/serie di operazioni che la View può effettuare. Il controller può anche essere utilizzato per esporre i dati e i modelli alla view, evitando che essa vi acceda direttamente.

Model: rappresenta i dati sui quali l'applicazione opera. Si tratta di classi/oggetti o interfacce per la comunicazione di rete o con un database. Il Model può essere un oggetto osservabile reattivo al quale la view si sottoscrive per riceverne gli aggiornamenti. Esistono diverse operazioni che la View può effettuare sul Model, come le operazioni CRUD (Create, Read, Update e Delete). Tali operazioni sono esposte alla view tramite il Controller in modo che non si abbia un riferimento diretto e che si separi la logica di business da quella grafica.

Il diagramma seguente rappresenta le relazioni che esistono tra Model, View e Controller.

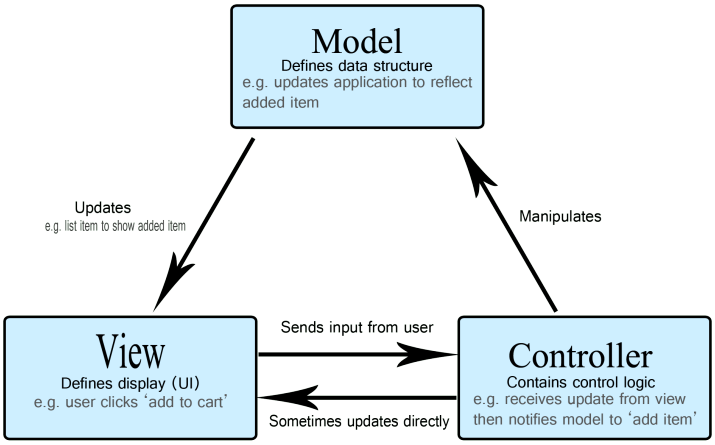

Figura 3.4: Rappresentazione del pattern MVC

Nell'applicazione Optima, ogni screen/route possiede un controller associato che espone alla view le funzionalità e le operazioni che è possibile utilizzare ed effettuare sui dati. Al fine di riutilizzare la logica comune, esistono dei controller che non sono associati a view specifiche, ma che vengono utilizzati da altri controller o combinati con altri. Un esempio di controllers non associati alla view sono il **NetworkController** e il **NavigationController**. Questi due oggetti software vengono utilizzati da altri controller rispettivamente per comunicare attraverso la rete e ottenere i dati dal server e per navigare nello screen/route successivo o precedente.

3.7 L'integrazione con il codice nativo

Oltre a comunicare con il backend per scambiare informazioni e salvare permanentemente i dati, l'applicazione è anche in grado di comunicare con due dispositivi embadded: la bilancia bluetooth e il lettore di barcode.
Il lettore di barcode è un dispositivo elettronico che permette di leggere un barcode e di visualizzarlo direttamente nella textview apposita, in un form dell'applicazione. L'integrazione di questo dispositivo non ha richiesto particolari sviluppi essendo compatibile con il protocollo di IO del sistema. Il dispositivo può essere collegato allo smartphone e dopo la lettura del barcode tramite il sensore apposito, simula la digitazione dei tasti sulla tastiera facendo sì che essi vengano visualizzati in una qualsiasi casella di testo come

se fossero stati inseriti manualmente dall'utente.
L'integrazione della bilancia ha richiesto invece sviluppi di codice nativo e l'adozione di un'architettura apposita. Essa comunica con l'applicazione tramite protocollo bluetooth e permette di misurare il peso dei prodotti. Si tratta di una bilancia bluetooth con la quale si può comunicare attraverso un SDK sviluppato dall'azienda produttrice. L'SDK è disponibile in nativo sia per sistemi Android che iOS, ma non è disponibile e compatibile con Flutter/Dart. Questo ha richiesto lo sviluppo e l'integrazione di codice nativo con il quale il codice Flutter potesse comunicare.
La gestione della bilancia è affidata all'oggetto **KitchenScaleController**. Questo oggetto software, oltre a comunicare con la bilancia attraverso l'integrazione del codice nativo, utilizza altri controller, come LifecycleController, BluetoothController e PermissionController, per gestire tutti gli eventi del sistema e dell'interfaccia bluetooth.
Per utilizzare il bluetooth sui dispositivi *mobile* è richiesta l'accettazione di alcuni permessi da parte dell'utente. La logica per il controllo e la richiesta dei permessi è gestita all'interno del PermissionController. Esso regisce attivamente ai cambiamenti dei permessi d'interesse per accedere al bluetooth, notificando l'utente e informandolo nel caso in cui uno di essi dovesse mancare. Il numero e il tipo di permessi richiesti dipende dal sistema operativo (iOS o Android) e dalla versione.
Il LifecycleController invece è incaricato di gestire gli eventi di background/foreground dell'applicazione. Quando l'applicazione va in background, la comunicazione con il bluetooth e lo scambio di dati con la bilancia viene interrotto. Al ritorno dell'applicazione in foreground la comunicazione viene ripresa automaticamente e l'utente nota solamente un piccolo ritardo. Questo tipo di logica è stata implementata per garantire un consumo energetico basso, essendo la comunicazione tramite bluetooth energeticamente dispendiosa.
Il codice seguente permette di visualizzare come i diversi Stream di eventi vengano combinati nel KitchenScaleController e mappati su stati facilmente interpretabili e gestibili nell'interfaccia grafica.

```
class KitchenScaleController {

    Stream<KitchenScaleState> get state => ...
        .switchMap((state) async* {
    if (
        state.item1 == BluetoothState.unavailable
        ) {
        yield KitchenScaleState
```

```
            .bluetoothNotAvailable;
      } else if (state.item4.isDenied) {
          yield KitchenScaleState
              .permissionsDenied;
      } else if (!state.item4.isGranted) {
          yield KitchenScaleState
              .permissionPermanentlyDenied;
      } else {
          if (state.item3) {
          if (state.item1 != BluetoothState.on) {
              yield KitchenScaleState
                  .bluetoothDisabled;
          } else if (
              state.item2 == LocationState.disabled
              ) {
              yield KitchenScaleState
                  .locationDisabled;
          } else {
              yield* _result.map((event) {
                  if (event.isLoading) {
                      return KitchenScaleState
                          .loading;
                  } else if (event.data == null) {
                      return KitchenScaleState
                          .error;
                  } else {
                  return KitchenScaleStateSuccess(
                      convertToOz(event.data!));
                  }
              });
          }
          }
      }
      })
      .startWith(KitchenScaleState.loading)
      .distinct();
  }
```

L'integrazione con il codice nativo è supportata da Flutter tramite l'utilizzo dei Channel o EventChannel. Il framework prevede di utilizzare un Channel nel codice Flutter e un Channel nel codice Kotlin/Java per Android e

Objective-C/Swift per iOS. I Channel definiti in Flutter e quelli definiti in Android o iOS devono condividere lo stesso idetificativo (si tratta di una stringa) e possono comunicare inviandosi oggetti serializzabili come stringhe, boolean o numeri. Il seguente codice è un esempio di creazione di un channel in Flutter.

```
final _eventChannel =
    const EventChannel("weightStream");

Stream<Result> get _result => _eventChannel
    .receiveBroadcastStream()
        .map((event) => Result.fromJson(
            jsonDecode(event)
                as Map<String, dynamic>
        ))
    .distinct();
```

Il metodo **receiveBroadcastStream()** permette di convertire un Channel in uno Stream. Si può notare come si utilizzi una rappresentazione JSON per la serializzazione dei dati trasmessi dal codice nativo al codice Flutter. Lo stesso channel deve essere definito nel codice nativo.

```
class MainActivity : FlutterActivity() {

    companion object {
        private const val CHANNEL = "weightStream"
    }

    override fun onCreate(
        savedInstanceState: Bundle?
    ) {
        super.onCreate(savedInstanceState)

        val flutterEngine = flutterEngine ?: return
    val getDataFromFirstKitchenDeviceDiscovered =
        (applicationContext as FlutterApplication)
        .getDataFromFirstKitchenDeviceDiscovered
    val moshi =
    (applicationContext as FlutterApplication)
    .moshi

    lifecycleScope.launch {
        eventFlow(
```

```
                flutterEngine.dartExecutor
                .binaryMessenger,
                CHANNEL
            ).collectLatest { ... }
        }
        }
    }
```

In Android, il Channel è stato definito nella MainActivity che è l'unica activity dell'applicazione. Mentre per iOS è stato definito nell'oggetto applicazione.

```
let controller : FlutterViewController =
    window?.rootViewController
        as! FlutterViewController
FlutterEventChannel(
    name: "weightStream",
    binaryMessenger: controller.binaryMessenger
).setStreamHandler(self)
```

Il codice non è molto differente tra Android ed iOS e si distingue solamente per alcune caratteristiche del linguaggio e del sistema operativo.
La connessione alla bilancia avviene appena il channel si connette e si interrompe quando il channel viene chiuso. La creazione e la chiusura del channel vengono gestite automaticamente dallo Stream nel codice Flutter che diventa attivo quando si connette il primo observer e si disattiva quando l'ultimo observer si disconnette.
In generale, la connessione alla bilancia avviene in due principali fasi.
All'attivazione del canale inizia lo scan dei dispositivi bluetooth e quando viene trovato il primo dispositivo compatibile, la scansione viene fermata e viene creata una connessione con il dispositivo. Se la connessione dovesse interrompersi, il processo riparte dalla scansione dei dispositivi bluetooth.
Tutto questo meccanismo è stato implementato con la libreria RxSwift su iOS poichè Swift non supporta la programmazione reattiva che è stata integrata solo dalla version 13 di iOS.

```
func execute() -> Observable<Result> {
    return discoverKitchenDevices.execute().first()
        .asObservable().map {
            device in _Result(device: device)
        }
        .startWith(_Result(isLoading: true))
        .catchAndReturn(_Result(isError: true))
```

```
            .flatMap { _result in
                if let d = _result.device {
                    return self.kitchenScaleRepository
                        .dataObservable(device: d)
                        .map {
                            data in Result(data: data)
                        }
                        .catch {
                            error in self.execute()
                        }
                }
                return Observable.of(_result.toResult())
            }
        }
```

Su Android invece si è scelto di utilizzare le Coroutines e i Flow supportati nativamente da Kotlin senza l'ausilio di una libreria esterna.

```
    operator fun invoke(): Flow<Result> = flow {
        while (true) {
            emit(Result(isLoading = true))

            val device = try {
                discoverKitchenDevices().first()
            } catch (e: Throwable) {
                emit(Result())
                break
            }

            emitAll(kitchenScaleRepository
                .dataFlow(device).map {
                    Result(data = it)
                }.catch {})
        }
    }
```

Le problematiche principali nell'utilizzo della bilancia sono state l'integrazione dell'SDK proprietario e la sincronizzazione degli accessi all'interfaccia bluetooth. Inoltre, a complicare la soluzione, è stato un problema di sincronizzazione interna nell'SDK Java per Android. Al fine di ovviare a tale problema, si è deciso di incapsulare l'SDK in una classe di interfaccia che si occupa di gestire gli accessi al bluetooth. Per sincronizzare l'accesso su Android sono stati utilizzati gli oggetti messi a disposizione dal linguaggio come

Mutex, Lock e Monitor. Questi tipi di sincronizzazione non sono disponibili invece su Swift/iOS per cui sono state utilizzate code di accesso (Dequeue) che permettono di simulare i monitor in Java.

Capitolo 4

Implementazione della soluzione

Sull'applicazione, oltre all'integrazione con la bilancia e il lettore di barcode, sono state implementate diverse funzionalità. L'utilizzo dell'applicazione è stato pensato come un software specializzato nell'eseguire determinate operazioni per semplificare il lavoro giornaliero degli operatori e dei dipendenti. Sono state rese disponibili anche funzionalità per l'amministratore che permettono di configurare alcuni parametri e recuperare i report e i dati di analisi.

4.1 Creazione nuovo inventario

Il principale caso d'uso pensato per l'applicazione è quello della creazione di un nuovo inventario. Il processo di creazione di un nuovo inventario consiste nel tracciamento, per ogni sede, da parte di un operatore, di tutti i prodotti contenuti in ogni stanza. La seguente tabella rappresenta il caso d'uso descritto in maniera dettagliata.

Creazione nuovo inventario	
Precondizione:	L'utente ha effettuato l'accesso
1.	L'utente clicca su aggiungi nuovo inventario
2.	L'utente seleziona una stanza
3.	L'utente clicca su inserisci nuovo prodotto
4.	Vengono riempiti i campi obbligatori
5.	Il prodotto viene salvato
6.	Si riparte dal punto 3 o 2 fino al termine delle stanze
7.	Viene salvato l'inventario

Tabella 4.1: Creazione nuovo inventario

L'inserimento di un nuovo inventario richiede che l'utente abbia effettuato l'accesso e si sia autenticato ed è disponibile sia per utenti amministratore sia per utenti specializzati.

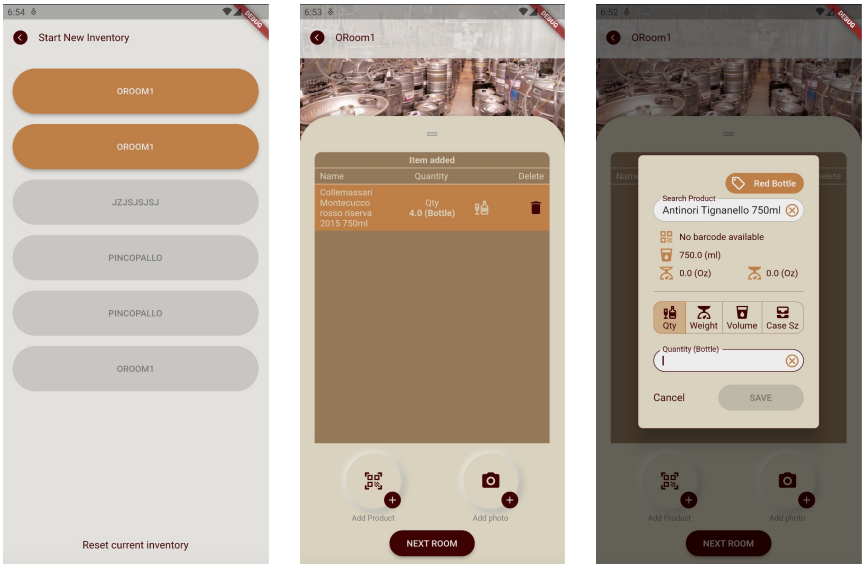

Figura 4.1: Creazione nuovo inventario

Come si può notare dagli screenshots precedenti, la creazione di un nuovo inventario inizia con la selezione di una stanza dalla lista di quelle che compongono la sede preselezionata. La configurazione della sede viene caricata nel database locale al lancio dell'applicazione tramite una chiamata REST al backend che permette di ricevere la lista di sedi disponibili per un determinato utente.
Ogni sede ha configurata una lista di stanze che vengono poi visualizzate. La

creazione di un nuovo inventario è gestita dal **NewFullCountStockController**. Il controller viene utilizzato dal **NewFullCountStock** che rappresenta la view. Un inventario può anche essere sospeso e ripreso in seguito e, al fine di garantire questa possibilità, quando si crea un nuovo inventario viene salvato nel database dell'applicazione un oggetto temporaneo che può essere cancellato in seguito o recuperato.
Il salvataggio e l'accesso al database avvengono tramite chiamate al **DatabaseController**, il quale si occupa di sincronizzare ed esporre al resto dell'applicazione le operazioni sul database. L'aggiunta di nuovi prodotti all'inventario viene effettuata tramite il click sul pulsante "Add product" visualizzato dopo aver selezionato una stanza. Per aggiungere un nuovo prodotto, lo si può selezionare da una lista di prodotti disponibili, anche essa recuperata tramite una chiamata al **NetworkController** che gestisce le richieste di rete. Nel caso in cui il prodotto che si vuole aggiungere non sia disponibile tra quelli precaricati, è possibile crearne uno nuovo tramite un'apposita funzionalità. Tutte le operazioni vengono affettuate sull'inventario temporaneo. Cliccando sul pulsante "salva inventario", che si abilita solo dopo il completamento di tutte le stanze, l'oggetto temporaneo viene mandato al server tramite una chiamata ad un API REST e viene salvato permanentemente nel database.

4.2 Inserimento nuovo prodotto

Altra funzionalità implementata nell'applicazione è quella di inserimento di un nuovo prodotto. Questa operazione è effettuata nel caso in cui in una ricetta o nella creazione di un nuovo inventario, il prodotto che si vuole inserire o registrare non sia presente in quelli caricati nel sistema. L'operazione di solito viene effettuata dall'amministatore attraverso il backoffice, ma si è voluto renderla disponibile anche sull'applicazione perché potrebbe capitare di dover aggiungere un nuovo prodotto da parte di un dipendente di una sede.

Aggiungi nuovo prodotto	
Precondizione:	L'utente ha effettuato l'accesso
1.	L'utente clicca sul pulsante aggiungi nuovo prodotto
2.	L'utente completa i cambi richiesti
3.	L'utente clicca su salva

Tabella 4.2: Aggiungi nuovo prodotto

Nella tabella dei casi d'uso precedente viene descritta l'interazione dell'utente con l'applicazione quando viene aggiunto un nuovo prodotto. L'operazione può essere effettuata sia direttamente dal menù principale dell'applicazione sia durante la compilazione di un nuovo inventario nel caso in cui si inserisca un barcode o un nome prodotto non presente nel database.

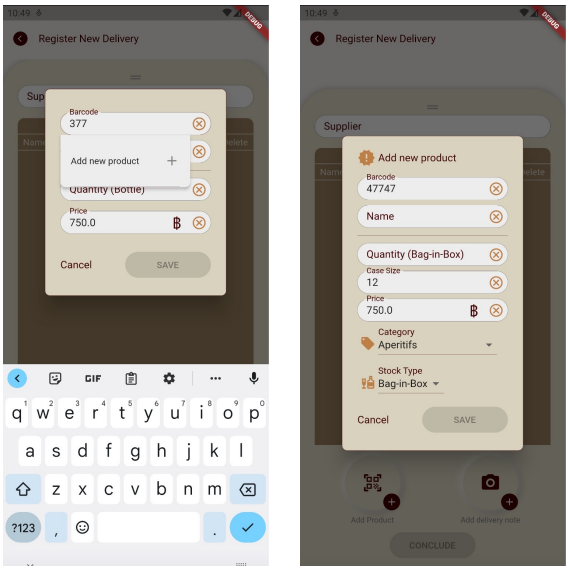

Figura 4.2: Aggiungi nuovo prodotto

Dagli screenshots dell'applicazione si può notare come l'aggiunta di un nuovo prodotto venga fatta tramite un form in un dialog. L'aggiunta di un nuovo prodotto viene gestita tramite alcune callback registrate nel popup che chiamano il codice dei controllers.

4.3 Registrazione nuova consegna

Oltre alla creazione dell'inventario per tracciare i prodotti presenti in ogni sede, si è voluta implementare la possibilità di registrare le consegne giornaliere o settimanali di nuovi prodotti. Questa operazione è accessibile dall'applicazione attraverso un'apposita voce nel menù principale. Durante questa operazione, l'utente registra ogni singolo prodotto ricevuto, può inserire il venditore che si è occupato della merce e deve inserire una foto della bolla di consegna. La nuova consegna registrata sarà consultabile in una sezione dell'applicazione dove vengono elencate in una lista tutte le consegne ricevute.

Registra nuova consegna	
Precondizione:	L'utente ha effettuato l'accesso
1.	L'utente clicca sul pulsante registra nuova consegna
2.	L'utente aggiunge i prodotti e le quantità
3.	L'utente carica una o più foto della bolla
4.	L'utente clicca su salva

Tabella 4.3: Registra nuova consegna

Nel secondo step, in cui si registrano i diversi prodotti, è anche possibile inserire prodotti non registrati nel database ricollegandosi al caso d'uso specifico.

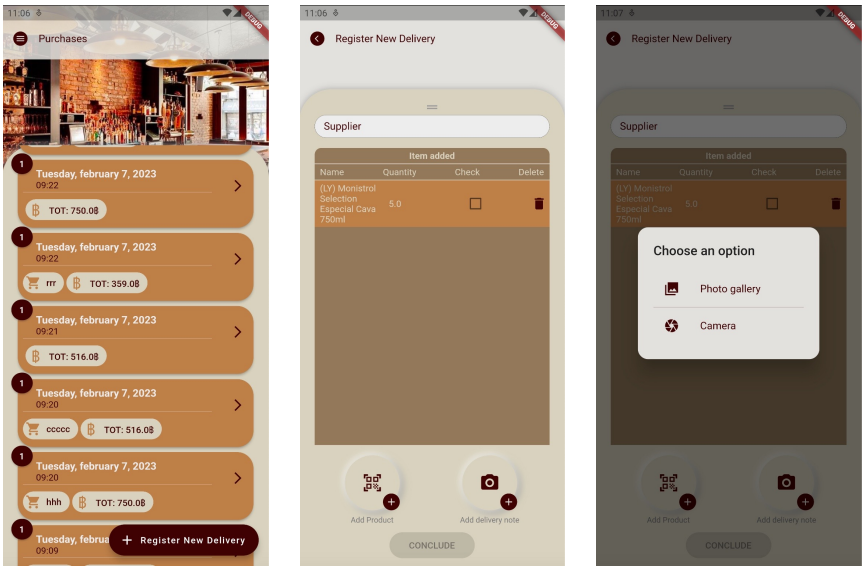

Figura 4.3: Registra nuova consegna

La registrazione di una nuova consegna è gestita dalla view **RegisterNewPurchase** e dal controller **RegisterNewPurchaseController**. I dati temporanei inseriti dall'utente vengono salvati all'interno del controller.
Se l'utente decide di sospendere la registrazione della consegna, i dati vengono temporaneamente salvati nel database. Nel caso in cui la registrazione venga completata, il RegisterNewPurchaseController interagisce con il **PurchaseController** e con il DatabaseController salvando la consegna sia nel database locale sia, tramite una chiamata REST, nel server remoto.

4.4 Configurazione della sede

L'operazione di configurazione della sede è disponibile solamente per gli utenti amministratori. Se l'accesso all'applicazione viene effettuato da un utente non amministratore, la voce nel menù non è visibile e le informazioni sulla dashboard dell'applicazione non sono editabili. La configurazione della sede si divide in due casi d'uso. Accedendo alla dashboard si possono editare nome, VAT, indirizzo e descrizione mentre accedendo alla sezione configura area, oltre a poter visualizzare vari dati, è possibile modificare le stanze.

Cambio dati area	
Precondizione:	L'utente ha effettuato l'accesso come amministratore
1.	L'utente accede alla dashboard e clicca su edit
2.	L'utente modifica i vari campi come: VAT, nome, ecc...
3.	L'utente clicca su salva

Tabella 4.4: Cambio dati area

Se l'utente non è autenticato come amministratore, il pulsante edit nella sezione dashboard è disabilitato. Questo permette all'utente di visualizzare i dati, ma non di modificarli.

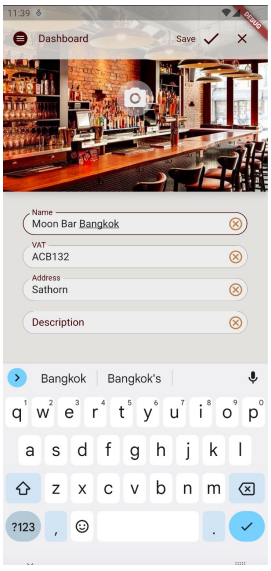

Figura 4.4: Cambio dati area

La gestione di questa sezione è affidata al **DashboardController**. I dati del form vengono salvati su variabili temporanee contenute nel controller.

Una volta che il pulsante salva viene cliccato, il DashboardController crea un oggetto con tutti i dati momentanei e utilizza il NetworkController per mandare i dati al server e il DatabaseController per salvarli in locale.
L'utente amministratore ha anche la possibilità di modificare le stanze di una sede. Per attivare questa funzionalità, l'amministratore ha a disposizione una voce specifica nel menù principale. All'interno di questa sezione sono presenti diverse voci come bevande, ricette e altre funzionalità che potrebbero essere integrate in futuro. Per modificare le stanze, l'utente deve accedere alla sezione Aree.

Configurazione stanze	
Precondizione:	Autenticato come amministratore
1.	L'utente accede alla sezione configurazione stanze
2.	L'utente può aggiungere o eliminare le stanze
3.	L'utente può ordinare le stanze
4.	L'utente clicca salva

Tabella 4.5: Configurazione stanze

Le stanze vengono visualizzate in una lista che è riordinabile tramite la funzionalità drag&drop. Per aggiungere una nuova stanza o eliminarla bisogna cliccare sul pulsante specifico.

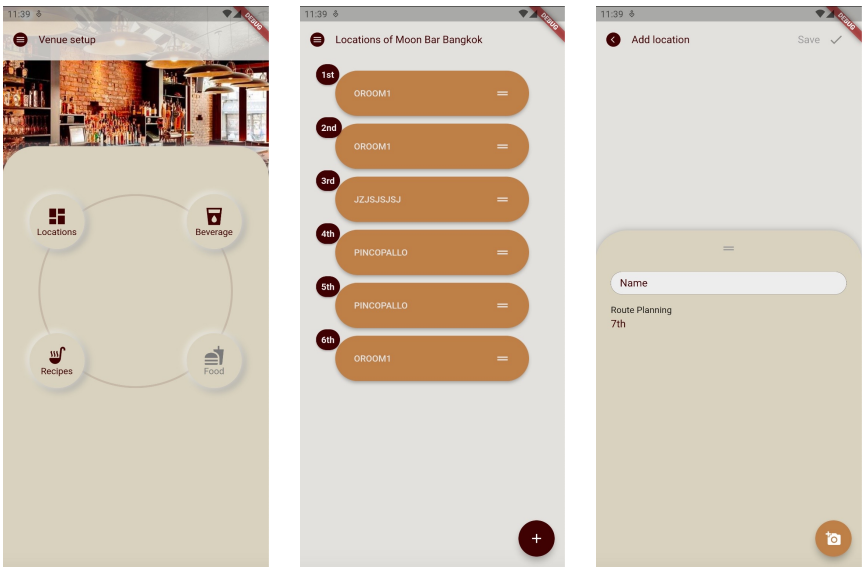

Figura 4.5: Configurazione stanze

La gestione delle stanze è implementata da due controller differenti: **VenueLocationListController** e **VenueLocationController**. Il primo controller si occupa di gestire, visualizzare e ordinare la lista di stanze in una determinata sede. Il secondo controller si occupa invece di aggiungere o eliminare una nuova stanza. L'ordinamento è gestito in real time, ovvero quando si cambia l'ordine della lista viene attivato un listener che modifica la configurazione anche nel database locale.

Capitolo 5

Conclusioni

Il progetto Optima è stato pensato sin dai primi sviluppi per essere un prodotto enterprise. L'idea alla base è che, essendo il problema della creazione di inventari molto comune nelle aziende medio/piccole, si voleva creare un prodotto che fosse possibile adattare a diversi clienti. Il progetto è ancora molto immaturo e molte funzionalità e migliorie potranno essere integrate in futuro al fine di renderlo affidabile e utilizzabile. Alcune funzionalità non sono state ancora implementate ma saranno disponibili in una seconda versione dell'applicazione. Per completezza ne elencheremo i casi d'uso.

5.1 Inventario delle eccezioni

Una delle funzionalità principali che non è stata ancora sviluppata è la creazione di un inventario delle eccezioni. Lo sviluppo di questa funzionalità non è ancora terminato lato server e quindi non è stato possibile completarla lato applicazione. L'inventario delle eccezioni è simile alla creazione di uno standard, ma all'interno di esso vengono registrati tutti i prodotti persi o scaduti. Questo tipo di inventario è molto utile all'azienda per capire le perdite dovute alla gestione dei prodotti o allo scarso consumo di essi. La logica è gia stata in parte implementata lato applicativo e il pulsante è disponibile nella sezione inventari, ma è momentaneamente disabilitato. Il seguente caso d'uso permette di definire l'interazione dell'utente con l'applicazione.

Creazione inventario delle eccezioni	
Precondizione:	L'utente ha effettuato l'accesso
1.	L'utente accede alla sezione inventari
2.	L'utente clicca su inventario delle eccezioni
3.	L'utente selezione una stanza
4.	L'utente inserisce i prodotti e le quantità
5.	L'utente procede alla stanza successiva
6.	L'utente clicca su salva inventario

Tabella 5.1: Creazione inventario delle eccezioni

Come si può notare, il caso d'uso è molto simile a quello della creazione di un nuovo inventario. La logica lato applicazione è gestita dalle stesse classi e oggetti che gestiscono la creazione di un nuovo inventario. L'unica differenza è nel salvataggio per cui vengono utilizzate tabelle del database locale e chiamate REST al server differenti.

5.2 Dati di vendita

Altre funzionalità non ancora sviluppate sono la possibilità da parte dell'amministratore di generare i dati di vendita e le analisi sulle perdite e sui guadagni. Queste funzionalità non sono supportate attualmente né lato server né lato applicazione, ma le voci sono presenti e disabilitate nel menù dell'applicazione. Entrambe le funzionalità sono disponibili solamente agli utenti amministratori e le voci scompaiono dal menù nel caso l'accesso sia effettuato come utente specializzato. Le due funzionalità possono essere descritte da due casi d'uso differenti.

Estrazioni dati delle vendite	
Precondizione:	Autenticato come amministratore
1.	L'utente accede al menù principale
2.	L'utente clicca su dati delle vendite
3.	L'utente può visualizzare i dati
4.	L'utente può esportare i dati

Tabella 5.2: Dati delle vendite

Questa funzionalità è molto utile all'amministratore per recuperare i dati sui prodotti effettivamente venduti fatta eccezione di quelli persi e quelli che sono ancora in giacenza nei magazzini. La seconda funzionalità invece

è l'estrazione di un report più complesso e completo su tutti i prodotti che siano persi, venduti o in giacenza.

Analisi dettagliata	
Precondizione:	Autenticato come amministratore
1.	L'utente accede al menù principale
2.	L'utente clicca su analisi dettagliata
3.	L'utente può visualizzare i dati
4.	L'utente può cambiare la visualizzazione dei dati
5.	L'utente può esportare i dati

Tabella 5.3: Analisi dettagliata

Il caso d'uso è molto simile all'estrazione dei dati delle vendite, ma in questo caso le informazioni riguardano anche prodotti persi o in giacenza. Il caso d'uso prevede anche di visualizzare i dati filtrandoli per tempo o per sede e stanza. In questo modo l'amministratore ha una visione più completa e dettagliata dell'azienda e può intervenire per migliorare i punti in cui si registrano più perdite.
Oltre alle funzionalità elencate, che sono già state previste negli sviluppi futuri, ci sono anche altri aspetti dell'applicazione che potrebbero essere migliorati. Nel caso in cui si voglia gestire il sistema per diversi clienti è presumibile pensare di dover gestire la personalizzazione dell'applicazione come labels, colori e immagini. Anche il server attualmente non è abbastanza maturo da poter gestire clienti multipli e oltretutto la sicurezza dei dati e il MTTF (Mean time to failure) andrebbe migliorato con tecniche RAID o mirroring.

Bibliografia

[1] Android Documentation, https://developer.android.com

[2] iOS Documentation, https://developer.apple.com

[3] Flutter Documentation, https://flutter.dev

[4] Git, https://www.atlassian.com/it/git/tutorials/what-is-version-control

[5] MVC, https://developer.mozilla.org/en-US/docs/Glossary/MVC

[6] Flutter vs Native vs React-Native: Examining performance, https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980

Ringraziamenti

In conclusione a questo lavoro di tesi, colgo l'occasione per ringraziare la mia famiglia per essermi sempre stata vicina e avermi supportato durante i miei studi universitari.
Un ringraziamento speciale va a mia sorella Beatrice che mi ha sempre supportato e aiutato durante il mio percorso di studi.
Ringrazio anche i miei amici per avermi distratto, ma allo stesso tempo supportato e sopportato.
Colgo anche l'occasione di ringraziare in maniera particolare il mio relatore il professore Giovanni Malnati e tutto il team che ha collaborato alla realizzazione di Optima.