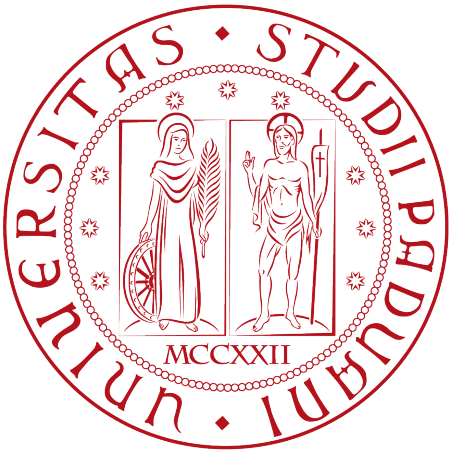
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Sviluppo di un'applicazione per il controllo di un distributore automatico

Tesi di laurea triennale

Relatore

Prof. Tullio Vardanega

Laureando

Giacomo Romano

Anno Accademico 2021-2022

Giacomo Romano: *Sviluppo di un'applicazione per il controllo di un distributore automatico,* Tesi di laurea triennale, © Luglio 2022.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di *stage*, della durata di circa trecento ore, del laureando Giacomo Romano presso l'azienda Moku S.r.l. tra maggio e giugno 2022. L'obiettivo di tale attività era progettare e sviluppare un'applicazione *mobile* per acquistare da *vending machine*. Ho sviluppato un'applicazione *mobile* sia per Android sia per iOS utilizzando un solo codice di sviluppo, attraverso il *framework* Flutter, che permette di fare esattamente questo.

Convenzioni tipografiche

Riguardo la stesura del testo, relativamente al documento ho adottato le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati, li ho definiti nel glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario ho utilizzato la seguente nomenclatura: *parola*$_{\text{g}}$;
- i termini in lingua diversa dall'italiano li ho scritti in *corsivo*.

Ringraziamenti

Ringrazio innanzitutto il Professore Tullio Vardanega, relatore di questa tesi di laurea, per avermi guidato con disponibilità e professionalità nella stesura del lavoro.

Desidero ringraziare con affetto la mia famiglia per aver sempre creduto in me ed essermi stata vicina in ogni momento.

Vorrei inoltre ringraziare il mio tutor aziendale Simone Pozzobon e l'azienda Moku per avermi permesso di crescere professionalmente durante questi mesi di stage.

Padova, Luglio 2022 Giacomo Romano

Indice

1 Contesto aziendale **1**
- 1.1 Profilo aziendale . . . 1
- 1.2 Prodotti e servizi . . . 1
  - 1.2.1 Trasformazione digitale . . . 1
  - 1.2.2 *Software* su misura . . . 2
  - 1.2.3 *Startup* . . . 3
- 1.3 Organizzazione aziendale . . . 3
- 1.4 Processi aziendali . . . 4
  - 1.4.1 Modello di ciclo di vita del *software* . . . 4
  - 1.4.2 Strumenti a supporto dei processi . . . 5
    - 1.4.2.1 Jira . . . 5
    - 1.4.2.2 Bitbucket . . . 6
    - 1.4.2.3 Slack . . . 6
    - 1.4.2.4 Calendario Google . . . 7
    - 1.4.2.5 Passbolt . . . 7
    - 1.4.2.6 Figma . . . 7
- 1.5 Propensione all'innovazione . . . 8

2 Lo *stage* nella strategia aziendale **9**
- 2.1 Gli *stage* per l'azienda . . . 9
- 2.2 Obiettivi del progetto di *stage* . . . 9
- 2.3 Vincoli del progetto di *stage* . . . 10
  - 2.3.1 Tecnologie . . . 10
    - 2.3.1.1 Flutter . . . 10
    - 2.3.1.2 GraphQL . . . 11
    - 2.3.1.3 Stripe . . . 11
    - 2.3.1.4 Android Studio . . . 12
  - 2.3.2 Tempistiche . . . 12
  - 2.3.3 Internazionalizzazione . . . 13
- 2.4 Scelta dello *stage* . . . 13

3 Il progetto di *stage* **15**
- 3.1 Stato iniziale del progetto . . . 15
- 3.2 Metodo di lavoro . . . 15
  - 3.2.1 Interazioni con il *tutor* aziendale e con il *team* del progetto . . . 15
  - 3.2.2 Revisioni di progresso . . . 16
  - 3.2.3 Incontri di allineamento . . . 16
  - 3.2.4 Uso di diagrammi . . . 16

3.2.5 Realizzazione . . . 17
3.3 Attività svolte . . . 18
3.3.1 Studio delle tecnologie . . . 18
3.3.2 Analisi dei requisiti . . . 18
3.3.2.1 Casi d'uso . . . 18
3.3.2.2 Requisiti . . . 22
3.3.3 Progettazione . . . 24
3.3.4 Codifica . . . 28
3.3.5 Verifica e validazione . . . 29
3.4 Risultati raggiunti . . . 30
3.4.1 Copertura di requisiti e di *testing* . . . 30
3.4.2 Quantità e qualità dei prodotti . . . 30

4 Visione retrospettiva **33**
4.1 Soddisfacimento obiettivi . . . 33
4.1.1 Obiettivi dell'azienda . . . 33
4.1.2 Obiettivi personali . . . 34
4.2 Maturazione professionale . . . 34
4.2.1 Conoscenze . . . 34
4.2.2 Abilità . . . 35
4.3 Distanza tra università e lavoro . . . 35
4.4 Conclusioni . . . 35

Acronimi e abbreviazioni **37**

Glossario **39**

Bibliografia **41**

Elenco delle figure

1.1 Interfaccia di Base 2
1.2 Interfaccia di XPlam 2
1.3 Interfaccia di Assibro 3
1.4 Rappresentazione grafica della metodologia Scrum 5
1.5 *Dashboard* per la gestione di Scrum in Jira 6
1.6 Rappresentazione grafica della metodologia Scrum 7

2.1 Esempio di *query* GraphQL 11
2.2 Interfaccia *built-in* di pagamento con Stripe 12

3.1 Incontri di allineamento 16
3.2 Esempio di diagramma dei casi d'uso 17
3.3 Metodo di lavoro nella realizzazione 17
3.4 Tecnologie utizzate 18
3.5 *Use case* - Gerarchia degli attori 18
3.6 *Use Case* - Casi d'uso principali 19
3.7 *Use Case* 1 - Autenticazione 20
3.8 Use Case 3 - Visualizzazione prodotti per categoria 20
3.9 Use Case 3.1 - Visualizzazione singolo prodotto 21
3.10 Distribuzione requisiti in percentuale per priorità 23
3.11 Distribuzione requisiti in percentuale per tipologia 23
3.12 Diagramma dei *package* architettura 25
3.13 Funzionamento di BloC (*Business Logic Components*) 26
3.14 Soluzione adottata per rendere efficiente il *fetch* dei dati 27
3.15 Metodo *handleFetch* della classe *FetchDataBloc* 28
3.16 Funzionalità prodotte 30

Elenco delle tabelle

1.1 Organizzazione aziendale 4

2.1 Pianificazione ore *stage* 13
2.2 Obiettivi personali nello svolgimento dello *stage* 14

3.1 Descrizione attori degli *use case* 19
3.2 Descrizione dei *layer* dell'architettura dell'applicazione 24
3.3 Metriche di qualità 31

4.1 Soddisfacimento obiettivi aziendali primari per il progetto 33
4.2 Soddisfacimento obiettivi aziendali secondari per il progetto 34

Capitolo 1

Contesto aziendale

1.1 Profilo aziendale

Moku S.r.l. è una startup nata nel 2013, fondata su un progetto *software* omonimo all'azienda, che ora opera producendo *software* su commissione. Lavora a stretto contatto con il cliente, sia esso stesso una *startup* o una grande impresa, definendo requisiti, *user experience* ed opportunità di *business*. Utilizza metodi agili e tecnologie recenti per produrre *software* di qualità, facile da mantenere ed estendere. Si è poi evoluta diventando società di consulenza IT, con l'intento primario di trasformare idee ambiziose in prodotti *software*.
Dal 2013 Moku ha sviluppato più di 60 progetti con oltre 40 clienti, tra piccole, medie e grandi aziende che operano in vari settori: biomedicina, *fashion*, educazione, assicurazioni, finanza, elettrodomestici, manufattura, automazione, ecc.

1.2 Prodotti e servizi

I prodotti e servizi offerti da Moku sono: trasformazione digitale, *software* su misura e *startup*, che illustro brevemente di seguito.

1.2.1 Trasformazione digitale

Con la trasformazione digitale, il cliente è guidato nel percorso di innovazione, identificando un piano operativo e strategico per migliorare l'efficacia e l'efficienza dei suoi processi aziendali, abbracciando le migliori soluzioni digitali e sviluppando le integrazioni che mancano.

Un esempio di progetto di trasformazione digitale è Base: un *software middleware* integrato capace di aggregare e gestire in modo indipendente flussi informativi e comunicativi tra laboratori e strutture ospedaliere collegate. Si tratta di un'applicazione *web* modulare installata in un laboratorio centrale automatizzato, che permette di raccogliere prenotazioni di esami batteriologici, elaborare i campioni attraverso dei *robot* con *machine learning*, tracciare i risultati e restituirli, una volta validati, agli ospedali di competenza.

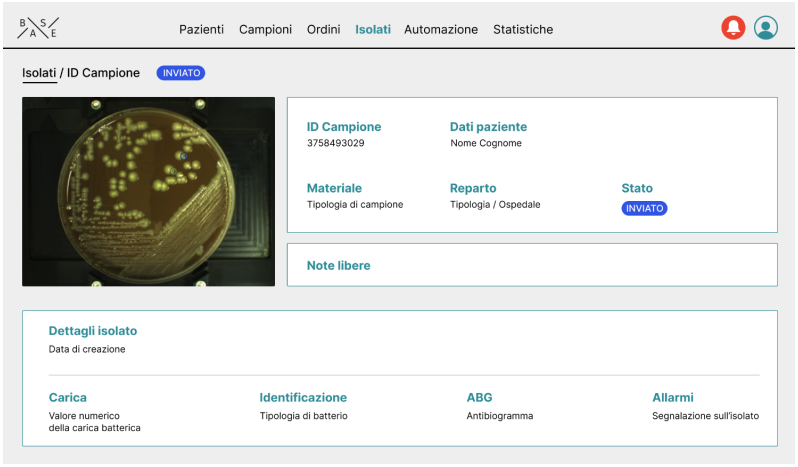


Figura 1.1: Interfaccia di Base

Source: *https://www.moku.io/it/progetti/base*

1.2.2 *Software* su misura

Con un *software* su misura, è possibile progettare, realizzazione soluzioni *software* (*web* e *mobile*) e *hardware* (IoT, *Machine Learning* e domotica) su misura; progetti digitali integrati sia *business-to-business* che *business-to-consumer*.

Un esempio di progetto di trasformazione digitale è XPlam: un *software* personalizzato per la configurazione di prodotti tecnici altamente complessi.

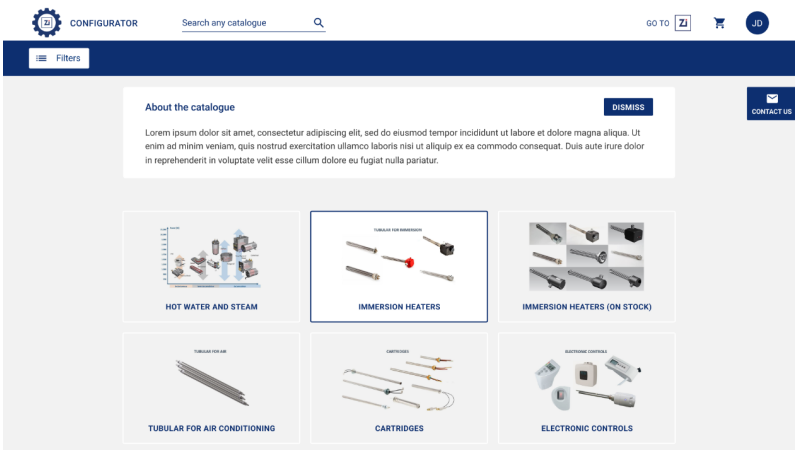


Figura 1.2: Interfaccia di XPlam

Source: *https://www.moku.io/it/progetti/xplam*

1.2.3 *Startup*

Attraverso una *Startup*, è possibile realizzare idee imprenditoriali in ambito digitale mettendo a sistema competenze, conoscenze ed un *team* dedicato, fino allo sviluppo e alla creazione dell'iniziativa.

Un esempio di progetto di *software* su misura è Assibro: una piattaforma multicanale per la gestione di polizze assicurative in un ambiente unico.

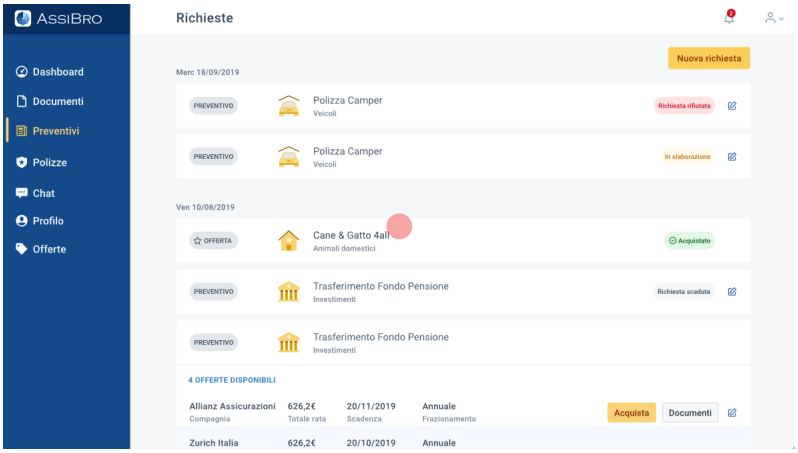


Figura 1.3: Interfaccia di Assibro

Source: *https://www.moku.io/it/progetti/assibro*

1.3 Organizzazione aziendale

La sede operativa di Moku è situata in via Schiavonia 3, Treviso (TV), e il suo organico, in continua crescita, conta venti membri, con competenze di tipo tecnico e amministrativo:

Figura professionale	**Competenze**
CEO (*Chief Executive Officer*)	È l'amministratore delegato dell'azienda. Ne definisce le scelte strategiche, dal modello di *business* all'approccio al mercato. Segue direttamente le relazioni con i clienti chiave e sviluppa le relazioni con investitori e *partner*.
CTO (*Chief Technology Officer*)	Il suo ruolo è quello di monitorare le nuove tecnologie e valutarne il loro potenziale applicato ai prodotti e servizi; ma anche quello di supervisionare i progetti di ricerca per assicurare che portino valore aggiunto alla società.

CFO (*Chief Financial Officer*)	Il suo ruolo principale è la gestione dei rischi finanziari dell'azienda. Si occupa inoltre delle pianificazioni finanziarie e dell'archiviazione della documentazione.
Analista	È colui che si occupa, nel processo di creazione del *software*, dello studio di dominio applicativo e specifica dei requisiti. Il suo operato è poi utilizzato nelle successive fasi di progettazione e sviluppo del *software*.
UX/UI *Designer*	È colui che si occupa di progettare l'interfaccia grafica (UI) e la relativa esperienza utente (UX), sulla base dei requisiti di funzionalità richiesti dal cliente e dei flussi operativi dell'applicazione (*web* o *mobile*) che deve essere sviluppata.
Sviluppatore	È colui che si occupa dell'effettiva codifica di quanto ricavato in fase di analisi e progettazione utilizzando linguaggi di programmazione e *framework*.

Tabella 1.1: Organizzazione aziendale

1.4 Processi aziendali

1.4.1 Modello di ciclo di vita del *software*

Moku, come modello di ciclo di vita del *software*, adotta la metodologia Scrum. Scrum è un *framework* agile di sviluppo del *software*, iterativo ed incrementale, concepito per gestire progetti e prodotti *software* o applicazioni di sviluppo, che è stato creato e sviluppato da Ken Schwaber e Jeff Sutherland.
La scelta, da parte di Moku, di adottare questo modello è determinata dal fatto che, lavorando con tecnologie all'avanguardia e con progetti molto complessi, è difficile pianificare ogni cosa in anticipo, e si predilige la capacità di reagire rapidamente a cambiamenti.
Lo scopo principale di questo modello è concentrarsi sull'ottenimento di *software* funzionante dando meno priorità ad aspetti non essenziali, per quanto importanti, come ad esempio una documentazione completa.
La metodologia Scrum si basa su tre principi:

1. Trasparenza: gli aspetti significativi del processo devono essere visibili ai responsabili del lavoro. La trasparenza richiede che quegli aspetti siano definiti da uno *standard* comune in modo tale che gli osservatori condividano una comune comprensione di ciò che viene visto.

2. Ispezione: chi utilizza Scrum deve ispezionare frequentemente i progressi realizzati verso il conseguimento degli obiettivi prestabiliti, individuando in tal modo precocemente eventuali difformità rispetto a quanto si intende realizzare. La

frequenza delle ispezioni non deve essere tale da determinare un'interruzione del lavoro in corso. Le ispezioni devono essere eseguite diligentemente e da ispettori qualificati.

3. Adattamento: se chi ispeziona verifica che uno o più aspetti del processo di produzione sono al di fuori dei limiti accettabili e che il prodotto finale non potrà essere accettato, deve intervenire sul processo stesso o sul materiale prodotto dalla lavorazione. L'intervento deve essere portato a termine il più rapidamente possibile per ridurre al minimo l'ulteriore scarto rispetto agli obiettivi prestabiliti.

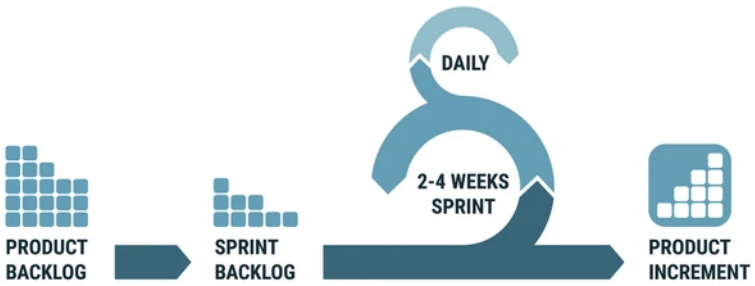


Figura 1.4: Rappresentazione grafica della metodologia Scrum

Source: *https://www.shutterstock.com/search/scrum*

Il *framework* Scrum prevede di dividere il progetto in blocchi rapidi di lavoro, chiamati *Sprint*, creando un incremento del *software* al termine di ciascuno di essi. Ogni *sprint*, che dura da 1/2 a 4 settimane, è preceduto da una riunione di pianificazione in cui vengono definiti gli obiettivi e stimati i tempi; questi obiettivi non possono essere cambiati durante lo *sprint*, ma soltanto alla successiva riunione di pianificazione. Nel corso di ogni *sprint*, il *team* crea porzioni complete di un prodotto. L'insieme delle funzionalità che vengono inserite in un determinato *sprint* provengono dal *product backlog*, che è una lista ordinata di requisiti. I requisiti del *product backlog* che vengono scelti come obiettivo per un determinato *sprint* costituiscono lo *sprint backlog*. Moku utilizza il *framework* Scrum con *sprint* generalmente lunghi da 10 a 15 giorni; alla fine di ciascuno *sprint*, viene valutato il progresso raggiunto e pianificato il successivo, stabilendo quali requisiti andranno a costituire lo *sprint backlog*.

1.4.2 Strumenti a supporto dei processi

L'azienda, per poter svolgere il proprio lavoro al meglio, sia a livello organizzativo che nelle fasi di produzione delle soluzioni, utilizza svariati applicativi. Alcuni di questi sono *open-source* e gratuiti, per altri l'azienda ha investito acquistando licenze e *account business*. Di seguito le tecnologie da me viste e usate, tra tutte quelle comunemente utilizzate dall'azienda e impiegate alla base di un qualunque progetto:

1.4.2.1 Jira

Per il supporto alla metodologia Scrum, Moku fa uso di Jira.
Jira è un software di *issue tracking*, che fornisce inoltre funzioni di *bug tracking* e *project management*.

La *feature* principale che contraddistingue Jira dagli altri software di *issue tracking*, e che ha spinto Moku a preferirlo, è la sua grande flessibilità, che gli permette di spaziare oltre le normali *issue* e di tracciare nuove richieste di *feature* da parte del cliente, *task* legati all'amministrazione di sistema, e soprattutto permette di incorporare il *framework* Scrum e di tracciare gli *sprint*, e i rispettivi *backlog* e *task*.
Ciascun *task* può essere assegnato ad uno o più membri del *team*, che può aggiornarne il progresso e registrare le ore di lavoro spese su quel *task*.
Tutte le operazioni descritte sono effettuabili mediante una comoda interfaccia grafica, con un frequente utilizzo dell'intuitivo *drag-and-drop* per la gestione dei vari *task*.

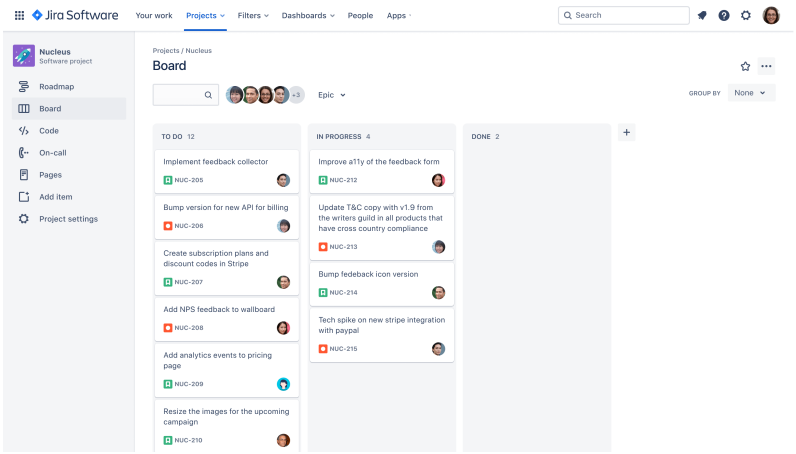


Figura 1.5: *Dashboard* per la gestione di Scrum in Jira

Source: *https://www.atlassian.com/it/software/jira/templates/kanban*

1.4.2.2 Bitbucket

Per la gestione del versionamento, Moku utilizza *repository* Git situati principalmente sulla piattaforma BitBucket.
La scelta è dovuta al fatto che BitBucket fornisce gratuitamente *repository* Git con la possibilità di renderli privati, cosa ad esempio non possibile su GitHub, se non a pagamento.
Una funzionalità molto utile offerta da Bitbucket è l'integrazione di Jira, grazie alla quale è possibile collegare le *issue* di Jira con i *commit* di Bitbucket, inserendo semplicemente il nome della *issue* nella descrizione del *commit*. In questo modo, è possibile vedere esattamente quanti e quali *commit* sono stati fatti su una certa *issue*.

1.4.2.3 Slack

Per la comunicazione interna, Moku utilizza Slack.
Slack è un *software* che rientra nella categoria degli strumenti di collaborazione aziendale utilizzato per inviare messaggi in modo istantaneo ai membri del *team*.
È possibile inviare messaggi sia a singole persone sia a gruppi di persone, dove in generale un gruppo di persone equivale ad un *team* di lavoro su un certo progetto.

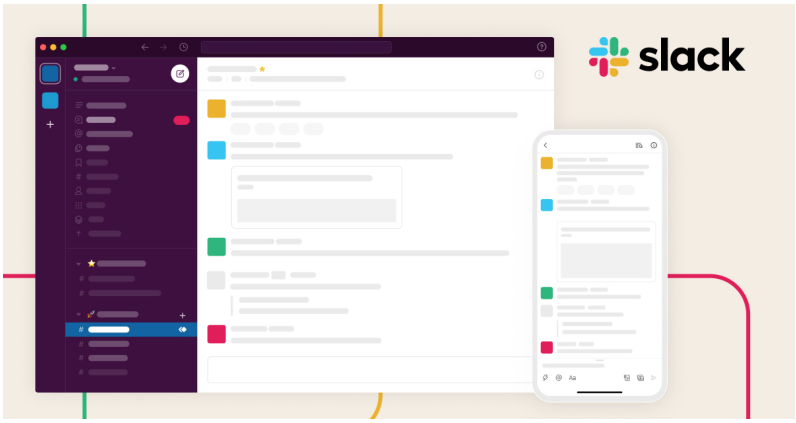


Figura 1.6: Rappresentazione grafica della metodologia Scrum

Source: *https://slack.com/intl/it-it/features*

1.4.2.4 Calendario Google

Moku utilizza tre calendari Google: uno per tenere conto dei permessi, uno per gestire i turni di *smartworking* e uno per segnare gli impegni rilevanti su determinati progetti, come può essere, ad esempio, una *call* con il cliente, piuttosto che un incontro di allineamento tra i membri del gruppo di un progetto, o una revisione di progresso, ecc.

1.4.2.5 Passbolt

Moku utilizza Passbolt come *password manager*, che serve a tenere in sicurezza la *password* dei vari *account* di applicativi *software* utilizzati dall'azienda nei diversi progetti di sviluppo. Passbolt risulta molto utile in quanto permette di centralizzare la gestione delle *password* e di renderle accessibili da tutti i dipendenti contemporaneamente.

1.4.2.6 Figma

Per la progettazione del *design* UX/UI e la prototipazione, Moku utilizza Figma.
Figma è un *editor* di grafica vettoriale e uno strumento di prototipazione attraverso il quale è possibile progettare il *design* grafico di un'applicazione *web* o *mobile*.
L'utilizzo di Figma per la definizione dell'interfaccia grafica ha numerevoli vantaggi:

- facilita il lavoro dello sviluppatore, che con pochi *click* ricava dal *design* prodotto: colori, distanze tra componenti, icone utilizzate, interazioni tra componenti;
- offre l'integrazione di Slack;
- è uno strumento collaborativo in tempo reale;
- è uno strumento *web-based*, quindi non è necessario scaricare alcun *software* e ogni modifica è auto-salvata nel *cloud*;
- è semplice da imparare e da utilizzare;
- è gratuito.

1.5 Propensione all'innovazione

L'azienda, per offrire sempre le soluzioni migliori sia per prestazioni sia per funzionalità, deve essere aggiornata sulle nuove tecnologie emergenti che meglio soddisfano sia bisogni nuovi sia bisogni meno attuali.
L'innovazione è un obiettivo primario per Moku, e lo dimostrano i tipi di prodotti e servizi che offre, in particolare la trasformazione digitale e le *startup*. L'azienda è quindi fortemente interessata all'innovazione, sia a livello di tecnologie innovative, che possano rendere più semplice, efficiente ed efficace lo sviluppo *software* rispetto a com'è utilizzando gli strumenti già conosciuti e utilizzati; sia a livello di progetti innovativi, che possano portare alla luce nuove idee e quindi nuove opportunità di crescita, sia professionale sia economica.
La propensione che Moku ha verso l'innovazione è anche determinata dal forte interesse per i giovani stagisti neo-laureati nei corsi di studi di Informatica e Ingegneria Informatica; infatti, Moku, seleziona di continuo stagisti, circa 2-3 alla volta, con l'intenzione di assumerli come dipendenti dopo lo stage, nel caso in cui il loro lavoro confermasse le aspettative dell'azienda.

Capitolo 2

Lo *stage* nella strategia aziendale

2.1 Gli *stage* per l'azienda

Gli *stage* per Moku significano un'importante opportunità di trovare nuovi "talenti". Infatti, per poter fare uno *stage* con Moku è necessario essere valutati positivamente in un *test* di sviluppo che consiste in 5 domande a cui bisogna rispondere con del codice in qualsiasi linguaggio di programmazione, dove il codice proposto dovrebbe risolvere il problema richiesto nel modo più efficiente possibile. Questo *test* è svolto su una piattaforma che permette a chi revisiona i risultati di un candidato, di visualizzare per ogni domanda un video della durata di tempo che il candidato ha impiegato per rispondere a quella domanda e in questo modo è possibile vedere e capire con quale procedimento viene scritto il codice e quindi il ragionamento che il candidato sta facendo, oltre a vedere il modo in cui il codice viene scritto. È un *test* a cui Moku tiene particolarmente in quanto molto indicativo di quanto un candidato possa o meno essere interessante per l'azienda.
Quando Moku seleziona degli stagisti, lo fa con l'obiettivo di assumerli come dipendenti, una volta terminato lo *stage*, nel caso in cui lo stagista confermasse, con il suo operato, la sua bravura rispetto a quello che Moku aveva notato in lui attraverso il *test* di sviluppo.
In particolare, Moku, ha interesse nei giovani laureati, sia con laurea triennale sia con laurea magistrale, specialmente nei corsi di studi di Informatica e Ingegneria Informatica. Questo perchè oltre ad avere sicuramente delle buone conoscenze di base in materia informatica, i neo-laureati sono abituati ad imparare rapidamente nuove tecnologie e a ideare soluzioni a problemi, qualità che Moku ricerca nei suoi dipendenti.

2.2 Obiettivi del progetto di *stage*

Moku S.r.l. sta sviluppando per conto di un'importante realtà del territorio, un'ecosistema per la gestione di *vending machine smart*. L'obiettivo del progetto di *stage* è sviluppare un'applicazione *mobile* che consenta ad utenti di *coworking*, uffici, hotel o altro di accedere ai servizi di *vending*. Per la prima versione dell'applicazione verranno integrate le macchine del caffè Cimbali e le vending machine Magex. Nel futuro l'obiettivo sarà quello di integrare anche altre tipologie di *vending machine*, in modo

da allargare l'interesse, l'influenza e l'applicabilità di tale applicazione a più clienti nel mercato, cioè a più aziende.
Gli obiettivi primari dello stage per questo progetto erano:

1. implementazione del *login*, con possibilità di reimpostazione della *password* ("*password* dimenticata");
2. implementazione della lista dei prodotti suddivisi in categorie in base alla *location* selezionata (una *vending machine* offre prodotti di una certa categoria);
3. implementazione del carrello;
4. implementazione della ricarica del borsellino tramite Stripe;
5. implementazione del flusso di acquisto di un ordine;
6. implementazione dello storico degli ordini.

Mentre gli obiettivi secondari erano:

1. autoselezione della *location* all'apertura dell'applicazione in base alla vicinanza geografica con le *vending machine* disponibili sul territorio nel raggio di 1 chilometro;
2. implementazione della lista dei preferiti;
3. possibilità di associare l'*account* Nexudus;
4. possibilità di modificare i propri dati di fatturazione;
5. possibilità di cambiare la *password* una volta che si utilizza l'applicazione da utente autenticato;
6. implementazione dello storico delle transazioni (sia delle ricariche sia degli acquisti).

2.3 Vincoli del progetto di *stage*

2.3.1 Tecnologie

2.3.1.1 Flutter

Per lo sviluppo dell'applicazione *mobile* doveva essere utilizzato Flutter.
Flutter è un *framework open-source* creato da Google per lo sviluppo *cross-platform* di applicazioni per dispositivi *mobile*. Con Flutter, è quindi possibile, a partire da un unico codice sorgente, effettuare il *deploy* dell'applicazione risultante sia su iOS che su Android. Recentemente il *team* di Flutter ha aggiunto il supporto anche per le piattaforme *web* e *desktop*.
Moku ha scelto Flutter per tutti i suoi vantaggi, in particolare perchè è multi-piattaforma, quindi è possibile avere la stessa interfaccia su diversi tipi di dispositivi e con un unico codice sorgente, riducendo il tempo di sviluppo e quindi i costi di sviluppo; e perchè ha *performance* equiparabili a quelle di applicazioni sviluppate in modo nativo, specialmente con le ultime versioni, che lo stanno sempre più ottimizzando.

2.3.1.2 GraphQL

Per il *back-end*, Moku ha scelto GraphQL.
Non era compito mio occuparmi del *back-end*, ma sapere la tecnologia che utilizzava era necessario in quanto questo determinava il modo in cui si ricevevano e inviavano i dati, da e verso il *server*.
GraphQL è un linguaggio di interrogazione e manipolazione dei dati *open-source* che fornisce un approccio per sviluppare *web* Application Program Interface (API). Il vantaggio principale di GraphQL è che permette ai *client* di definire la struttura dei dati richiesti, evitando così la restituzione di grandi moli di dati in eccesso rispetto a quello che davvero serve.
Per esempio, se volessi fare una query per ottenere gli ordini e di ogni ordine volessi sapere l'id, il prezzo e i suoi prodotti, e di ogni prodotto volessi sapere: id, nome, prezzo e quantità; mi basterebbe scrivere la seguente *query*:

```
{
  orders {
    id
    price
    products {
      id
      name
      price
      quantity
    }
  }
}
```

Figura 2.1: Esempio di *query* GraphQL

GraphQL mi permette di dire esattamente quali dati mi interessano, e quindi riceverò solo quelli che ho richiesto. Nell'esempio, qualsiasi altra informazione di un ordine o di un prodotto (come potrebbe essere la data in cui è stato fatto l'ordine) non verrebbe ritornata dal *server*.
Le richieste che più comunemente vengono fatte con GraphQL sono le *query* e le *mutation*. Le prime sono l'equivalente di una richiesta di lettura, mentre le *mutation* vengono utilizzate per tutte le altre operazioni: creazione, modifica, cancellazione.
Moku ha scelto GraphQL in quanto strumento innovativo e per la sua versatilità ed efficienza nel recuperare esattamente i dati richiesti, e nessun altro.

2.3.1.3 Stripe

Per effettuare acquisti *in-app*, in particolare per ricaricare il borsellino virtuale con il quale si pagano gli ordini, Moku ha deciso di utilizzare Stripe.
Stripe è un'infrastruttura *software* che permette a privati e aziende di inviare e ricevere pagamenti via Internet in modo sicuro. La sua integrazione con Flutter offre un'interfaccia nativa già pronta, testata e sicura, per effettuare pagamenti *in-app*.
Stripe offre anche la possibilità di effettuare il pagamento tramite Google Pay o tramite Apple Pay.

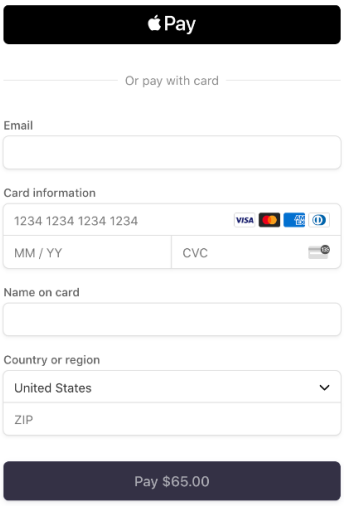


Figura 2.2: Interfaccia *built-in* di pagamento con Stripe

Source: *https://docs.page/flutter-stripe/flutter_stripe/sheet*

Moku ha scelto di utilizzare Stripe per effettuare pagamenti *in-app* in quanto oltre ad essere sicura, la sua integrazione in Flutter è molto semplice, veloce e ben documentata, e offre addirittura un'interfaccia già pronta per l'inserimento dei dati della carta o di altri metodi di pagamento. Il che significa avere la possibilità di effettuare acquisti *in-app* in modo sicuro e con il minimo sforzo possibile.

2.3.1.4 Android Studio

Per lo sviluppo dell'applicazione, Moku ha deciso che avrei dovuto utilizzare Android Studio come Integrated Development Environment (IDE)$_g$.
Android Studio è un IDE per lo sviluppo per la piattaforma Android. È stato annunciato il 16 maggio 2013 in occasione della conferenza Google I/O tenuta dal *Product Manager* Google, Katherine Chou. Disponibile gratuitamente e basato sul *software* di JetBrains IntelliJ IDEA, Android Studio è stato progettato specificamente per lo sviluppo di applicazioni Android. È disponibile il *download* su Windows, Mac OS X e Linu e sostituisce gli *Android Development Tools* (ADT) di Eclipse, diventando l'IDE primario di Google per lo sviluppo nativo di applicazioni Android.
Nonostante sia pensato primariamente per sviluppare applicazioni Android, supporta molto bene anche lo sviluppo in Flutter, attraverso l'installazione del relativo *Plugin*$_g$.
Moku ha scelto di utilizzare Android Studio per la realizzazione dell'applicazione in quanto è l'IDE che già utilizzavano per lo sviluppo nativo in Android, e, oltre a trovarsi bene, hanno colto la possibilità, offerta da Android Studio, di sviluppare anche in Flutter.

2.3.2 Tempistiche

Ho iniziato lo stage il 27 aprile 2022 e l'ho terminato il 24 giugno 2022, per un totale di trecento ore. Il carico di lavoro giornaliero è sempre stato di otto ore, in giornate lavorative che iniziavano alle 9 del mattino e finivano alle 18 di sera, con

l'ora dalle 13 alle 14 libera per il pranzo. Le trecento ore sono state preventivamente ripartite/pianificate così come ho descritto nella seguente tabella:

Attività	**Ore di lavoro**
Comprensione sistema e obiettivi	20
Analisi dei requisiti	20
Progettazione	60
Studio e setup ambiente di sviluppo	20
Implementazione	150
Test e validazione	30
Documentazione	20

Tabella 2.1: Pianificazione ore *stage*

2.3.3 Internazionalizzazione

Anche se inizialmente l'applicazione verrà rilasciata solo in Italia, è da prevedere anche il suo rilascio in Unione Europea (e in futuro magari anche oltre l'Unione Europea); di conseguenza deve essere già predisposta per l'internazionalizzazione.
In particolare, la lingua che l'applicazione deve utilizzare è la stessa lingua utilizzata dal sistema operativo, e se la lingua che utilizza non è ancora gestita dall'applicazione, viene usata una lingua di *fallback*, che per adesso è la lingua italiana.

2.4 Scelta dello *stage*

Ho avuto la possibilità di venire a conoscenza dell'azienda Moku, e dei progetti di *stage* che essa offriva, attraverso l'evento STAGE-IT.
STAGE-IT è l'evento promosso da Assindustria Venetocentro in collaborazione con l'Università di Padova per favorire l'incontro tra aziende, con progetti innovativi in ambito IT, e studenti dei corsi di laurea in Informatica, Ingegneria informatica e Statistica. Aziende di tutti i settori, dalle manifatturiere alle *software house*, hanno l'opportunità di presentarsi e descrivere agli studenti i progetti in cui mettere alla prova le proprie competenze valutandone così attitudini e talenti. Noi studenti abbiamo l'opportunità di entrare in contatto con le realtà aziendali e presentare i nostri profili per partecipare a concrete proposte formative.
Tra diverse opportunità di *stage* che mi si erano presentate, ho scelto proprio Moku per le seguenti ragioni principali:

- Al primo colloquio che ho fatto con loro, quello durante l'evento STAGE-IT, i due capi che erano presenti mi sono sembrati molto simpatici e alla mano, inoltre mi sono sembrati piuttosto giovani, cosa che mi ha fatto intuire che probabilmente anche i loro dipendenti lo fossero, e lavorare in un ambiente giovane era un aspetto che vedevo come molto positivo, segno, tra l'altro, di propensione all'innovazione.

- Ridotta distanza geografica rispetto a casa.
- Uno dei progetti di *stage* proposti, quello a cui mi sono candidato e che mi è stato assegnato dal mio *tutor* aziendale, richiedeva l'utilizzo del *framework* Flutter, e questo mi interessava particolarmente in quanto quando ho avuto la possibilità di usarlo nel progetto di Ingegneria del *Software*, mi ero reso conto di quanto questo strumento fosse innovativo e potente, e quindi mi piaceva l'idea di approfondire la sua conoscenza.

Gli obiettivi personali che mi ero prefissato prima di iniziare lo *stage* erano:

Obiettivo	**Motivo**
Approfondire la mia conoscenza del *framework* Flutter	Sia dal punto di vista di come lavora "sotto", sia dal punto di vista delle *best practices* da utilizzare nello sviluppo, specialmente a livello di struttura dei componenti grafici delle schermate e le relazioni tra di essi, cercando di massimizzarne l'efficienza. Prima di questo *stage*, avevo utilizzato Flutter per la prima volta solo qualche mese prima, in occasione del progetto di Ingegneria del *Software* e di conseguenza, avevo ancora poca esperienza e tante cose non le avevo ancora capite.
Imparare nuove tecnologie	Per ampliare i miei orizzonti, con lo scopo sia di imparare strumenti che mi permettessero di fare cose che non avevo mai provato a fare o nemmeno pensato di fare, sia di imparare strumenti che mi permettessero di fare quello che già sapevo fare ma farlo in modo più semplice ed efficiente.
Scoprire il mondo del lavoro	Sia per avere un'idea più chiara di quello che mi aspetta nella vita futura, sia per rendermi conto del rapporto tra le conoscenze e capacità che posseggo, e l'applicabilità di queste nel mondo reale: un mondo non più didattico o universitario, ma fatto di clienti che vogliono determinati prodotti e io devo essere grado di realizzarli nel modo più efficiente possibile. Una prima esperienza lavorativa come questa ti fa rendere conto della tua preparazione effettiva e quindi di quello che puoi effettivamente offrire nel mondo del lavoro.

Tabella 2.2: Obiettivi personali nello svolgimento dello *stage*

Capitolo 3

Il progetto di *stage*

3.1 Stato iniziale del progetto

Non ho continuato lo sviluppo o fatto *restyling* di un'applicazione già esistente, ma ho iniziato un progetto "da zero". In particolare, il cliente disponeva già di un'applicazione che offriva le stesse funzionalità dell'applicazione che avrei dovuto sviluppare io, ma non era più soddisfatto di quella soluzione, in quanto

- alcune cose non funzionavano bene;
- molte cose di *user interface* (UI) non andavano più bene;
- molte cose di *user experience* (UX) non andavano più bene.

L'applicazione in questione è stata completamente cestinata da Moku e quindi non sono partito da quello che c'era già per risolvere i problemi che erano emersi, ma ho proprio dovuto iniziare un nuovo progetto.

3.2 Metodo di lavoro

3.2.1 Interazioni con il *tutor* aziendale e con il *team* del progetto

Le interazioni con il *tutor* aziendale e con gli altri membri del *team* del progetto sono state praticamente all'ordine del giorno, o parlando di persona oppure attraverso Slack, specialmente nei giorni in cui ho lavorato da remoto. Con il *tutor* aziendale le interazioni sono state tante probabilmente perchè oltre ad essere il mio *tutor* aziendale era anche *project manager* del progetto che stavo sviluppando; di conseguenza tutti i dubbi che mi sorgevano riguardo alle funzionalità dell'applicazione, li esponevo a lui. A lui chiedevo anche il permesso di introdurre alcune piccole funzionalità marginali che non erano state previste esplicitamente, ma che sviluppando l'applicazione, mi sembravano utili dal punto di vista della *user experience*.
Le interazioni sono state tante anche con gli altri membri del *team* di questo progetto, in particolare con il UX/UI *designer*, a cui rivolgevo alcuni dubbi e proposte riguardo le scelte progettuali dell'interfaccia grafica e dell'esperienza utente, e con lo sviluppatore del *back-end*, a cui rivolgevo la richiesta di aggiungere alcuni dati che era necessario ricevere da parte di alcune *query* che ancora non ritornavano quegli specifici dati.

3.2.2 Revisioni di progresso

Le revisioni di progresso avvenivano circa una volta ogni due settimane e consistevano nel mostrare l'applicazione sviluppata fino a quel momento, al *tutor* aziendale, attraverso una sorta di *showcase*, discutendo degli avanzamenti, di ciò che mancava da fare e di eventuali problemi o dubbi riguardanti l'applicazione in generale o la *user experience*.

3.2.3 Incontri di allineamento

Durante lo *stage*, il *team* del mio progetto, composto da me (*mobile developer*), chi faceva *back-end*, chi si occupava del *design* UX/UI e il *project manager*, ha tenuto alcuni incontri di allineamento. Durante questi incontri, ognuno esponeva sinteticamente:

- cosa aveva fatto
- cosa gli mancava da fare
- problemi o dubbi emersi nel periodo appena terminato

rispetto al proprio lavoro.
In base a quanto veniva fuori e in base anche alle priorità di alcuni membri del *team* che erano impegnati su più progetti, il *project manager* decideva quali sarebbero state le prossime funzionalità da sviluppare nel breve periodo, che non necessariamente dovevano essere diverse rispetto a quanto era già stato deciso in precedenza.

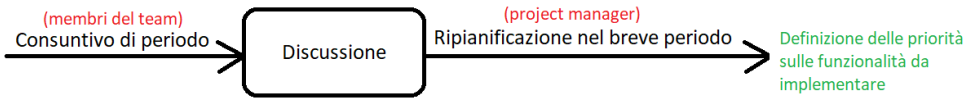


Figura 3.1: Incontri di allineamento

Questi incontri li ho trovati molto utili in quanto di facevano capire concretamente quale fosse lo stato complessivo del progetto e la priorità da dare allo sviluppo di alcune funzionalità rispetto ad altre.

3.2.4 Uso di diagrammi

Nella seconda settimana, cioè quella in cui ho svolto l'attività di analisi dei requisiti, ho utilizzato i diagrammi dei casi d'uso per comprendere meglio gli scenari principali e alternativi delle funzionalità dell'applicazione.
I diagrammi dei casi d'uso sono diagrammi Unified Modeling Language $(UML)_g$ comportamentali, dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.
Nel seguente diagramma di esempio, si descrivono due funzionalità di un sistema di *e-commerce*, cioè la visualizzazione dei prodotti e l'aggiunta al carrello di un prodotto; l'attore è colui che può utilizzare quelle funzionalità e nell'esempio è un generico utente.

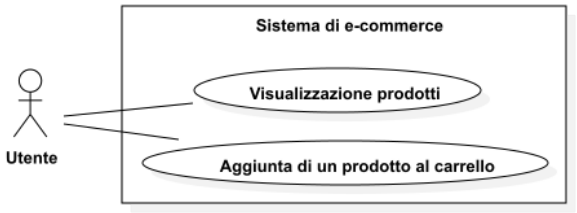


Figura 3.2: Esempio di diagramma dei casi d'uso

Utilizzare questi diagrammi nella fase di analisi del problema mi è stato molto utile in quanto mi ha permesso di sviluppare un'idea più chiara del problema in generale, ma anche dei flussi principali e alternativi dell'applicazione. Mi ha inoltre permesso una più rapida comprensione delle funzionalità che il sistema, cioè l'applicazione, avrebbe dovuto offrire.

3.2.5 Realizzazione

Il metodo di lavoro che ho adottato durante la realizzazione è il seguente: per ogni funzionalità su cui volevo o dovevo iniziare a lavorare, creavo una *issue* su Jira, se già non esistesse, e la spostavo nella colonna "*in progress*" oltre che assegnarla a me; inoltre in molti casi mi è stato utile suddividere una *issue* in vari *sub-task* (i quali vengono trattati da Jira come *issue* a tutti gli effetti), per entrare più nello specifico della parte della funzionalità che stavo sviluppando, specialmente nel caso di funzionalità molto complesse o articolate. Attraverso codice, in Flutter, implementavo quella funzionalità, facendo riferimento al *design* progettato su Figma, sia a livello di interfaccia grafica (UI) sia a livello di esperienza utente (UX). Terminata la realizzazione di piccole parti che compongono l'intera funzionalità, eseguivo un *commit* sul *repository* Git del progetto (sulla piattaforma BitBucket), e nella descrizione del commit inserivo l'identificativo della *issue* a cui stavo lavorando. In questo modo su Jira, è possibile vedere tutti i *commit* fatti su una certa *issue*. Terminata l'implementazione di una *issue*, spostavo tale *issue* sulla colonna "*done*", ad indicare che l'avevo implementata. Infine, sempre su Jira, andavo a segnare le ore della giornata che avevo speso per lavorare ad una determinata *issue*.

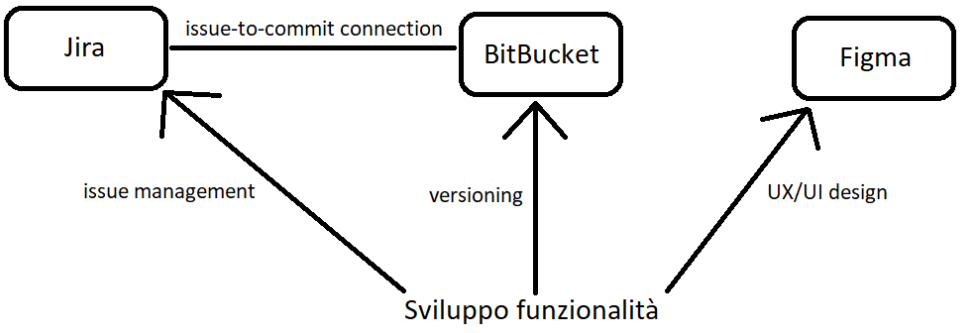


Figura 3.3: Metodo di lavoro nella realizzazione

3.3 Attività svolte

3.3.1 Studio delle tecnologie

La prima settimana di lavoro l'ho impiegata nella comprensione degli obiettivi del progetto e nello studio delle tecnologie che avrei dovuto utilizzare nello sviluppo dell'applicazione e per conseguire tali obiettivi. Le tecnologie sono sostanzialemente 4:

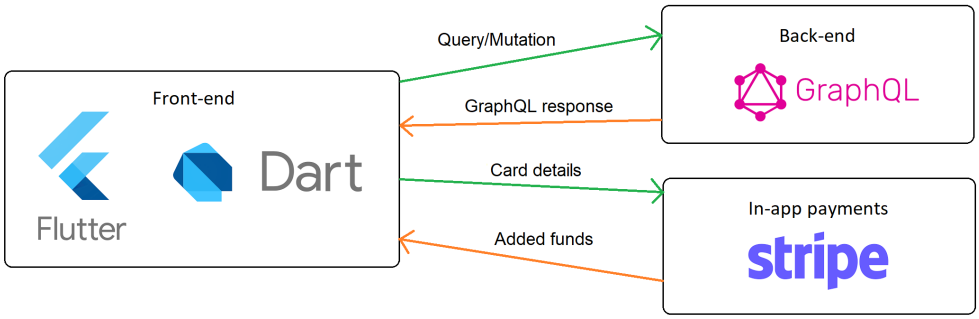


Figura 3.4: Tecnologie utizzate

Dart, che a differenza delle altre tecnologie non l'ho ancora introdotto in questa tesi, è il linguaggio di programmazione su cui si basa Flutter, è stato sviluppato da Google con l'intento di risolvere i problemi di JavaScript (che, si dice, non possano essere risolti con l'evoluzione del linguaggio) offrendo al tempo stesso migliori prestazioni, la possibilità di sviluppare più facilmente strumenti utili alla gestione di progetti di grandi dimensioni e migliori funzionalità legate alla sicurezza.
Lo studio di tali tecnologie è avvenuto principalmente attraverso la lettura della documentazione ufficiale di ognuna di essi, con anche qualche prova pratica per verificare di averne capito i concetti.

3.3.2 Analisi dei requisiti

3.3.2.1 Casi d'uso

In questa sezione verranno analizzati i casi d'uso delle funzionalità principali. Gli attori del sistema sono tre:

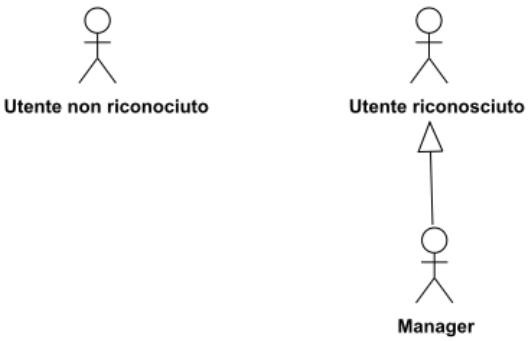


Figura 3.5: *Use case* - Gerarchia degli attori

e sono descritti come di seguito:

Attore	**Descrizione**
Utente non riconosciuto	Identifica un utente che non ha ancora effettuato il *login* nell'applicazione.
Utente riconosciuto	Identifica un utente che ha effettuato il *login* nell'applicazione.
Manager	Identifica un utente riconosciuto che ha la possibilità di scegliere il metodo di pagamento con il quale pagare un certo ordine, cioè ha la possibilità di scegliere se usare il proprio borsellino virtuale o se mettere la spesa in conto alla sua azienda.

Tabella 3.1: Descrizione attori degli *use case*

Per l'identificazione dei casi d'uso, ho scelto la seguente notazione:

UC[Identificatore]

dove l'**identificatore** è un codice numerico univoco.
I casi d'uso principali del sistema sono i seguenti:

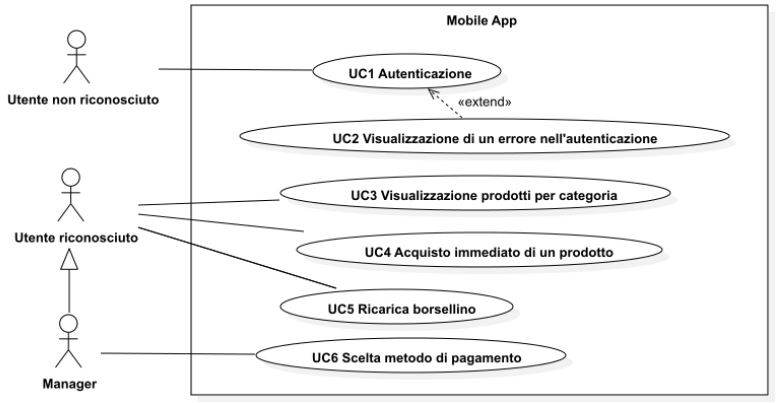


Figura 3.6: *Use Case* - Casi d'uso principali

Ce ne sono molti altri che però non riporto per motivi di sintesi.

UC1: Autenticazione

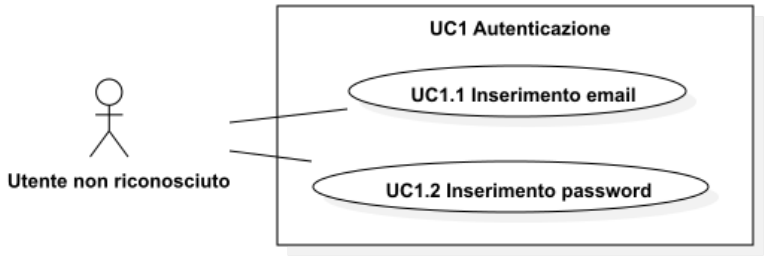


Figura 3.7: *Use Case* 1 - Autenticazione

Attori Principali: Utente non riconosciuto.

Precondizioni: l'utente si trova nella videata di *login*, contenente il *form* per l'inserimento della *email* e della *password*.

Descrizione: l'utente inserisce l'*email* e la *password* e clicca sul bottone per effettuare il *login*.

Postcondizioni: se le credenziali sono corrette, l'utente diventa un utente riconosciuto, in quanto ha effettuato il *login*; se invece le credenziali non corrispondono ad alcun utente registrato, un messaggio di errore localizzato (cioè tradotto nella lingua utilizzata) informa l'utente di tale situazione.

Scenario Alternativo: nella richiesta di *login*, potrebbe accadere un errore (o un errore dovuto alla connessione ad Internet assente o un errore dovuto a problemi lato *back-end*), in questo caso viene visualizzato un messaggio di errore localizzato che informa di tale situazione (UC2).

UC2: Visualizzazione di un errore nell'autenticazione

Attori Principali: Utente non riconosciuto.

Precondizioni: l'utente ha provato ad effettuare il *login* ma un errore è stato sollevato nell'elaborazione della richiesta.

Descrizione: i casi in cui può essere sollevato un errore nell'autenticazione sono sostanzialmente due: o il dispositivo *mobile* dell'utente non è connesso ad internet, oppure c'è qualche problema lato *server*.

Postcondizioni: tramite un messaggio di errore localizzato, l'utente prende conoscenza della particolare situazione di errore che si è verificata.

UC3: Visualizzazione prodotti per categoria

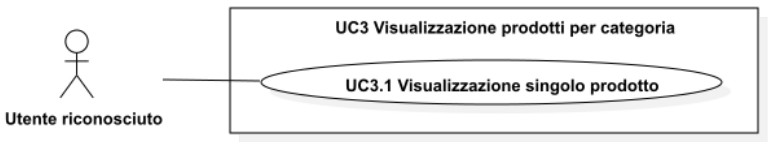


Figura 3.8: Use Case 3 - Visualizzazione prodotti per categoria

Attori Principali: Utente riconosciuto.

Precondizioni: l'utente è nella home, in cui può visualizzare i prodotti presenti nella *location* selezionata, divisi per categorie, dove una categoria corrisponde ad una specifica *vending machine*.

Descrizione: l'utente visualizza i prodotti offerti.

Postcondizioni: l'utente ha visualizzato i prodotti offerti.

UC3.1: Visualizzazione singolo prodotto

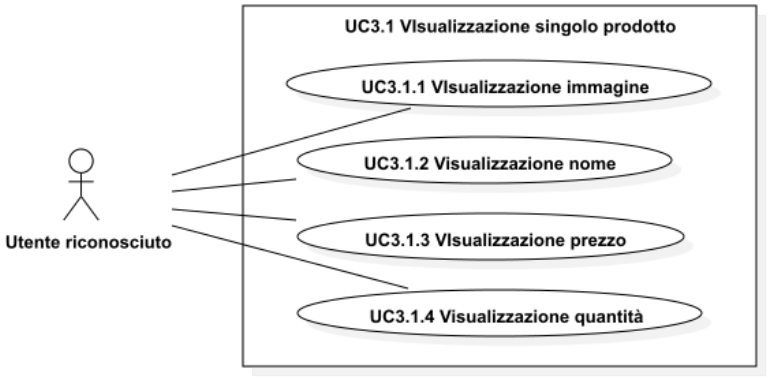


Figura 3.9: Use Case 3.1 - Visualizzazione singolo prodotto

Attori Principali: Utente riconosciuto.

Precondizioni: l'utente è nella *home*, e tra tutti i prodotti offerti, si sofferma a visualizzarne uno solo.

Descrizione: l'utente visualizza l'immagine, il nome, il prezzo e la quantità del singolo prodotto; la quantità è presente solo per i prodotti che non sono caffè.

Postcondizioni: l'utente ha visualizzato un singolo prodotto tra quelli offerti.

UC4: Acquisto immediato di un prodotto

Attori Principali: Utente riconosciuto.

Precondizioni: l'utente si trova nella pagina di dettaglio di un prodotto e preme sul bottone dell'acquisto immediato.

Descrizione: l'acquisto immediato si differenzia da un acquisto classico in quanto non c'è la necessità di passare per il carrello; questa possibilità è offerta con l'obiettivo di migliorare la *user experience* nel caso in cui si volesse acquistare un solo prodotto, diminuendo il numero di *click* necessario per farlo.

Postcondizioni: se l'utente ha abbastanza fondi nel borsellino virtuale, viene effettuato il pagamento e se tutto va a buon fine viene visualizzato il *qr code* dell'ordine, con il quale l'utente può procedere con l'erogazione del prodotto; se invece l'utente non ha abbastanza fondi oppure se durante il pagamento accade un errore di qualsiasi tipo, viene informato di tale situazione attraverso un messaggio di errore localizzato.

UC5: Ricarica borsellino

Attori Principali: Utente riconosciuto.

Precondizioni: l'utente è nella videata del borsellino e dopo aver scelto un importo di ricarica, clicca sul bottone per effetuare il pagamento *in-app* tramite carta, Google Pay, Apple Pay, ecc.

Descrizione: la ricarica può essere fatta scegliendo arbitrariamente l'importo, oppure scegliendolo tra alcuni premi fissi (10 euro, 15 euro, 20 euro, 30 euro); in ogni caso, l'importo minimo di ricarica è di 10 euro.

Postcondizioni: se il pagamento va a buon fine, il denaro viene aggiunto al borsellino virtuale e sarà quindi spendibile all'interno dell'applicazione; se invece il pagamento non va a buon fine, per qualsiasi ragione, l'utente verrà informato, tramite un messaggio localizzato, della situazione di errore che si è verificata.

UC6: Scelta metodo di pagamento

Attori Principali: *Manager*.

Precondizioni: per avere la possibilità di scegliere il metodo di pagamento, l'utente deve essere un *Manager*, e il fatto che un utente sia o meno un *Manager* è un informazione che viene reperita dall'account Nexudus dell'utente, quindi l'utente deve aver associato, all'interno dell'applicazione, il suo *account* Nexudus.

Descrizione: l'utente *Manager* può scegliere se pagare un ordine con il proprio borsellino virtuale, come qualsiasi altro utente, oppure se mettere la spesa in conto alla sua azienda.

Postcondizioni: l'utente *Manager* ha scelto il metodo di pagamento.

3.3.2.2 Requisiti

Durante le prime settimane di stage, oltre a prendere famigliarità con le nuove tecnologie, ho avuto modo di studiare tutti i requisiti del sistema, attraverso l'interazione con il *tutor* aziendale, *project manager* di questo progetto, che dopo aver parlato con il cliente, mi riportava, a voce, le funzionalità richieste. Ho scelto la seguente notazione per identificare i requisiti:

R[Tipo][Priorità][Identificatore]

- **Tipo**: indica se il requisito è di tipo
  - **F**: funzionale (aggiunge funzionalità al prodotto);
  - **Q**: qualitativo (relativo alla qualità del prodotto finale);
  - **V**: di vincolo (regole e norme che devono essere seguite).
- **Priorità**: indica l'importanza di un requisito
  - **O**: irrinunciabile;
  - **D**: desiderabil (non strettamente necessario ma dal valore aggiunto riconoscibile);
  - **Z**: opzionale (relativamente utile, che apporta un piccolo valore in più).

- **Identificatore**: codice numerico per identificare e disambiguare i requisiti.

Al termine dell'analisi, ho individuato 31 requisiti.

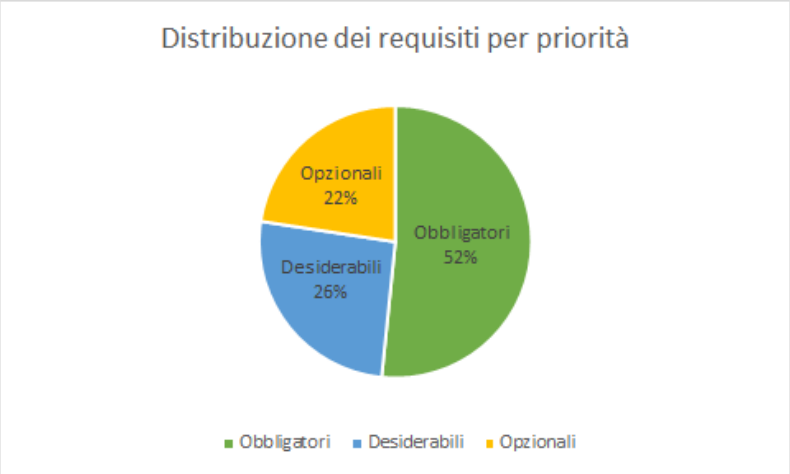


Figura 3.10: Distribuzione requisiti in percentuale per priorità

Distribuiti per priorità, i requisiti sono 16 obbligatori, 8 desiderabili e 7 opzionali. In particolare, molti requisiti tra quelli desiderabili e opzionali sono emersi durante lo svolgimento del progetto, in occasione delle interazioni tra il cliente e il *project manager*, e tra me e il *project manager* (che era anche il mio *tutor* aziendale).

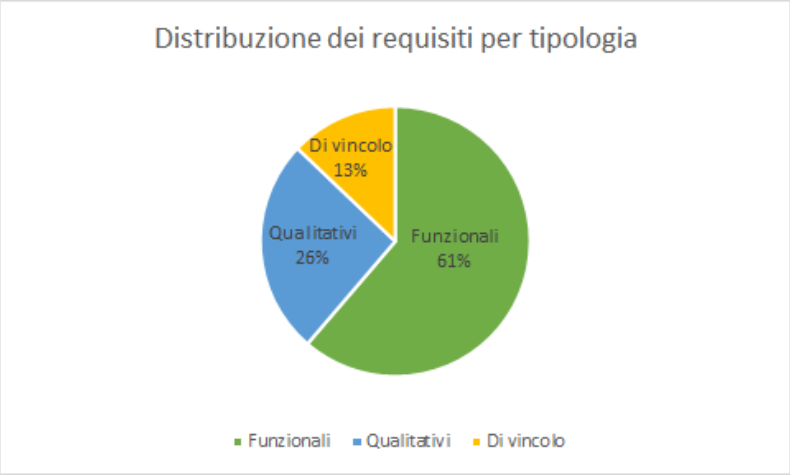


Figura 3.11: Distribuzione requisiti in percentuale per tipologia

Distribuiti per tipologia, i requisiti sono 19 funzionali, 8 qualitativi e 4 di vincolo. I requisiti di vincolo rappresentano le tecnologie a cui il mio progetto di stage era vincolato, quelli di qualità riguardano invece la qualità progettuale e quindi del sistema.

3.3.3 Progettazione

Nell'attività di progettazione ho ragionato su come implementare l'applicazione nel modo più efficiente possibile, attraverso la scelta dell'architettura più adatta e l'uso di *best practices*.
L'architettura che ho scelto di adottare è BLoC. BLoC (*Business Logic Components*) assomiglia molto al MVVM (*Model View ViewModel*), in cui BLoC sostituisce il *ViewModel*. Bloc è una libreria per la gestione dello stato, ma il suo utilizzo permette anche una netta suddivisione dell'applicazione in tre layer: *presentation layer*, *business layer* e *data layer*.

Layer	**Descrizione**
Presentation layer	Il *presentation layer* è costituito da componenti grafici e la sua responsabilità è di capire come ricostruirli in base agli stati emessi delle classi BLoC che utilizza. Inoltre gestisce l'*input* utente (interazione dell'utente con la UI) e gli stati del ciclo di vita dell'applicazione.
Business layer	Il *business layer* è costituito dalle classi BLoC: generalmente si ha una classe BLoC per videata, o per *feature*, o per componente grafico indipendente (cioè stesso componente riutilizzato in più videate dell'applicazione). La responsabilità del *business layer* è di rispondere agli eventi sollevati dal *presentation layer*, emettendo nuovi stati. Questo *layer* dipende da uno o più *repository* del *data layer*, dai quali recupera i dati necessari per costruire il nuovo stato dell'applicazione.
Data layer	La responsabilità del *data layer* è di recuperare/manipolare una o più sorgenti di dati. Questo *layer* costituisce il livello più basso dell'applicazione e si occupa dell'interazione con *database*, richieste di rete e altre sorgenti asincrone di dati. È composto da un certo numero di *repository* e di *data provider*. La responsabilità di un *data provider* è di provvedere dati grezzi. Un *data provider* dovrebbe essere generico e versatile, e generalmente espone una API per l'esecuzione di operazioni sulla sorgente dati che gestisce/rappresenta (ad esempio, un *data provider* che permette l'accesso ad un *database*, potrebbe esporre dei metodi per le operazioni CRUD_g). Un *repository* contiene i riferimenti a uno o più *data provider*. Oltre ad esporre esattamente i metodi esposti da ogni singolo *data provider* che usa, un *repository* potrebbe esporre un metodo che interagisce con più *data provider* ed eseguire trasformazioni sui dati prima di restituire il risultato al *business layer*.

Tabella 3.2: Descrizione dei *layer* dell'architettura dell'applicazione

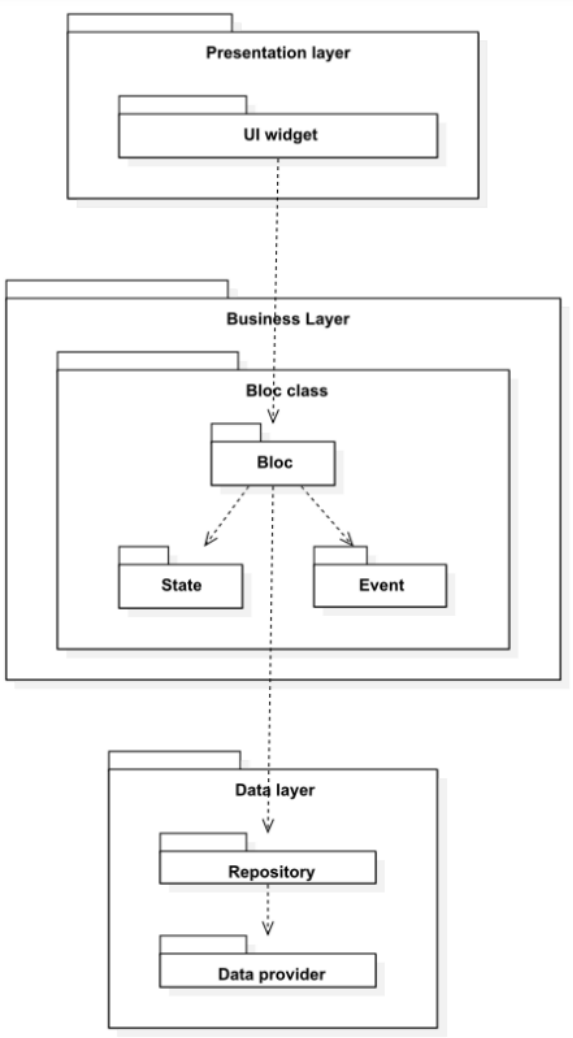


Figura 3.12: Diagramma dei *package* architettura

Il motivo principale per cui ho avuto la necessità di utilizzare una libreria per gestire lo stato dell'applicazione è perchè, di *default*, nella gestione dello stato in Flutter, *presentation layer* e *business layer* sono uniti, non c'è una netta separazione, e questo non permette la facile manutenibilità di un prodotto.
Tra le numerose alternative di librerie per la gestione dello stato di un'applicazione realizzata in Flutter, ho scelto BLoC perchè:

- dispone di un'eccellente documentazione;
- è la più utilizzata per sviluppo con Flutter e quindi in rete sono presenti un grande numero di tutorial ed esempi;
- è risultata, in generale, migliore, nella comparazione con altre librerie.

In particolare, i vantaggi di BLoC sono:

- sviluppo facile e veloce;

- netta separazione tra *presentation layer* e *business layer*;
- facile da testare;
- efficienza;
- codice facilmente riusabile.

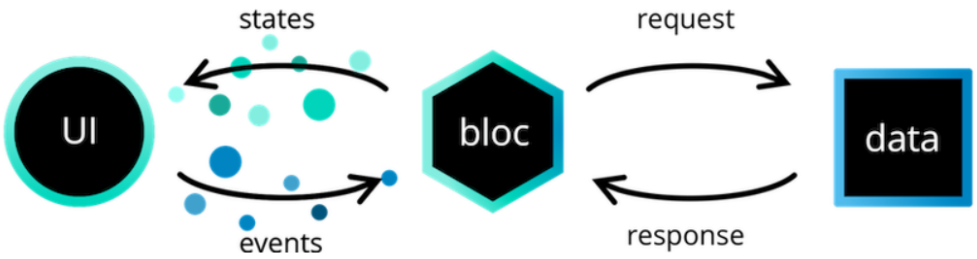


Figura 3.13: Funzionamento di BloC (*Business Logic Components*)

Source: *https://bloclibrary.dev/#/architecture*

Il flusso classico di funzionamento è il seguente: l'utente interagisce con l'interfaccia grafica, questa reagisce all'interazione attraverso l'aggiunta di un evento allo Stream degli eventi di una classe BLoC, l'evento viene ascoltato dal BLoC, il quale utilizza il *data layer* per recuperare i dati necessari a costruire il nuovo stato; ottenuti i dati, costruisce il nuovo stato e lo emette attraverso lo Stream degli stati; tutti i componenti grafici che sono in ascolto di quello stato, lo ricevono e possono aggiornarsi di conseguenza, utilizzando i dati presenti all'interno dello stato ricevuto.
Ogni classe BLoC implementa il *design pattern* Observer attraverso gli Stream di Dart, uno *stream* per gli stati e uno per gli eventi.
L'utilizzo degli *stream* spiega perchè nel diagramma dei *package* dell'architettura, il *business layer* non abbia un riferimento al *presentation layer*: una classe BLoC emette stati, ma non sa se ci sono componenti grafici in ascolto di quegli stati e quali siano questi componenti, nel caso ci fossero. Questo aumenta il grado di disaccoppiamento tra le classi.
Il *business layer* si interfaccia esclusivamente con i *repository* e mai direttamente con i *data provider*. I *data provider* sono nascosti all'interno dei *repository* e quindi chi usa un certo *repository* non è a conoscenza di quali *data provider* il *repository* fa uso. Di conseguenza il *business layer* usa i metodi esposti da un *repository* senza poter sapere, e senza avere interesse nel sapere, la fonte da cui i dati ritornati provengono.

Oltre che nella scelta dell'architettura del prodotto, mi sono concentrato molto anche sulle *best practices* da adottare, sia riguardo all'implementazione dell'interfaccia grafica e quindi dei suoi componenti e la loro interazione, sia riguardo l'implementazione delle operazioni che l'interazione con l'interfaccia grafica comportava. In generale, l'obiettivo che ho perseguito, e a cui ho ambito, era di ottenere le migliori *performance* con la migliore efficienza, cioè con la migliore gestione delle risorse.
Di seguito intendo fare due esempi pratici che possano far comprendere meglio questo intento.
Per quanto riguarda le *best practices* da seguire nella realizzazione dell'iterfaccia grafica,

l'idea è che quando una videata deve aggiornarsi perchè cambiano i dati a cui fa riferimento, si debbano aggiornare solamente i componenti che necessitano di aggiornamento e per i quali i dati di riferimento sono diversi rispetto allo stato precedente (in quanto i dati potrebbero aggiornarsi ma essere uguali a quelli già presenti). Prevenire inutili ricostruzioni dei componenti grafici è sinonimo di migliori prestazioni in quanto la ricostruzione di un componente ha un costo. *Best practices* sempre riguardo le prestazioni e l'efficienza sono state utilizzate anche nell'implementazione di liste e nella gestione in memoria delle immagini dei prodotti, oltre che in altri contesti dell'applicazione.

Per quanto riguarda invece massimizzare le *performance* e l'efficienza delle operazioni che l'interazione con l'interfaccia grafica comporta, era presente il seguente problema: quasi ogni interazione con l'interfaccia grafica comporta il dover effettuare una richiesta al *back-end*, ma le richieste dovrebbero essere fatte tutte allo stesso modo, cioè la fase di "richiesta in corso" dovrebbe mostrare un componente grafico di caricamento/attesa al centro dello schermo; il fallimento della richiesta, per qualsiasi motivo, deve essere gestito visualizzando un *dialog* con uno o più messaggi di errore; mentre in caso di successo viene ritornato il risultato che però è dello stesso tipo per ogni richiesta. Quindi il comportamento rispetto all'esecuzione/elaborazione di una richiesta doveva essere lo stesso per ogni richiesta; il problema è che le richieste vengono fatte da tutte le schermate e ogni videata ha la sua classe BLoC che se ne occupa. Un altro problema, è l'alto numero di dipendenze sul *repository* che espone i metodi per fare le richieste, che ho chiamato *DataRepository*: ogni classe BLoC avrebbe dovuto averne un riferimento.
Dunque come gestire questo problema evitando di ripetere lo stesso codice di gestione di una richiesta per ogni classe BLoC?
La soluzione che mi è venuta in mente è quindi quella di avere una classe BLoC centralizzata che si occupa solo ed esclusivamente del *fetch* dei dati e della gestione del risultato della richiesta. In questo modo: il *fetch* e il modo in cui viene gestito il risultato è scritto una volta per tutte, e quindi è uguale ad ogni richiesta fatta, inoltre, tutte le classi BLoC ora hanno solo un riferimento a questo BLoC centralizzato, che ho chiamato *FetchDataBloc*, e non più al *data repository*, che ora è riferito esclusivamente dal *FetchDataBloc*.

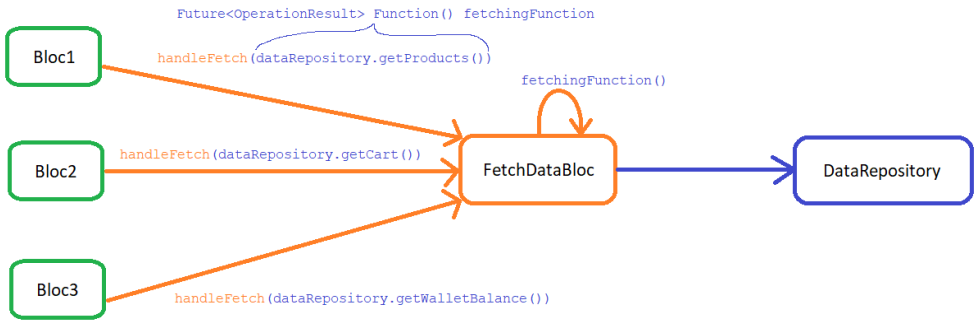


Figura 3.14: Soluzione adottata per rendere efficiente il *fetch* dei dati

Come è possibile notare da questa immagine, i vari BLoC delle schermate, si interfacciano ad un BLoC comune, centrale, per poter effettuare una qualsiasi richiesta. In particolare, il *FetchDataBloc* ha un riferimento al *DataRepository*, il quale espone le varie operazioni consentite. Per eseguire una di queste operazioni, una qualunque

classe BLoC deve invocare:

fetchDataBloc.handleFetch(fetchDataBloc.dataRepository.getProducts())

Dove al posto di "*getProducts*()" è possibile scegliere qualunque metodo esposto dal *DataRepository*.
In questo modo, passo come parametro al metodo "*handleFetch*" la funzione che effettivamente esegue la richiesta che mi interessa, e che ho recuperato attraverso il riferimento a *DataRepository*, esposto da *FetchDataBloc*.

```
Future<Map<String, dynamic>?> handleFetch(
    Future<OperationResult> Function() fetchingFunction,
    {bool showProgress = true}) async {
  add(FetchDataLoading(showProgress: showProgress));
  OperationResult result = await fetchingFunction();
  if (result.errorMessages.isNotEmpty) {
    add(FetchDataFailed(
        errorMessages: result.errorMessages,
        errorCodes: result.errorCodes,
        networkError: result.networkError));
  } else {
    add(const FetchDataSucceeded());
  }
  return result.data;
}
```

Figura 3.15: Metodo *handleFetch* della classe *FetchDataBloc*

Grazie a questa soluzione, la gestione dell'elaborazione della richiesta è fatta, decisa, una sola volta per tutte le richieste possibili nel presente e nel futuro, ed è racchiusa nel metodo "*fetchData*(...)" della classe *FetchDataBloc*. Questo permette di gestire l'attesa per l'elaborazione di una richiesta allo stesso modo per tutte le richieste, così come il successo e anche il fallimento. È da notare inoltre che questa funzione può essere usata per fare il *fetch* di qualunque richiesta, purchè sia una funzione che ritorni un futuro di tipo "*OperationResult*", che è una classe che ho definito io con l'obiettivo di unificare il tipo di ritorno di ogni richiesta, qualunque sia la fonte da cui i dati di quella richiesta provengono.

Questa soluzione, come la soluzione di tante altre problematiche di ottimizzazione emerse durante lo sviluppo, è stata pensata da me con l'intento di perseguire: efficienza, *performance*, riusabilità di codice, manutenibilità.

3.3.4 Codifica

L'attività di codifica è consistita nella realizzazione delle soluzioni che avevo ideato durante l'attività di progettazione. Quindi, per ogni funzionalità che dovevo implementare, prima veniva una fase di progettazione in cui pensavo alla soluzione e poi una fase di codifica in cui andavo effettivamente a realizzare tale soluzione.
A livello di tempo, sicuramente ho dedicato complessivamente più tempo alla codifica rispetto che alla progettazione e a tutte le altre attività, però sono state tante anche le ore di progettazione, in quanto non è stato, per me, sempre immediato trovare la

soluzione migliore per ogni problema: a volte ho impiegato anche una mattinata intera o anche una giornata di lavoro intera solo a pensare alla soluzione migliore per un certo problema.
La fase di codifica, per le prime funzionalità, ha richiesto più tempo in quanto ho dovuto studiare le *best practices* anche della codifica e ho dovuto approfondire la mia conoscenza di Flutter e Dart; dopodichè, andando avanti con le altre funzionalità ho preso sempre più la mano, a tal punto che in alcuni casi implementare una certa funzionalità poteva richiedere meno tempo o ugual tempo rispetto che progettarne la soluzione.

Esempi di *best practices* utilizzate nella codifica sono: organizzazione del codice in file e cartelle secondo un certo criterio, per massimizzare la manutenibilità; utilizzo della *keyword const* su tutti i componenti grafici non variabili, in modo da prevenirne inutili ricostruzioni; utilizzo di codice asincrono ove possibile per migliorare l'efficacia; naming conventions per variabili, classi, file, cartelle; definizione di componenti grafici indipendenti nel caso in cui lo stesso componente venisse utilizzato in più schermate, in modo da rendere il codice più leggibile, riutilizzabile e manutenibile; suddivisione del *layout* di una videata in più file, uno per ogni singolo componente grafico di quella videata, questo aumenta in modo considerevole la leggibilità e la manutenibilità del codice (soprattutto nel caso di *layout* molto complessi, tenere tutto in un solo file diventa improponibile), ecc.

3.3.5 Verifica e validazione

Per la verifica del codice ho utilizzato strumenti di analisi statica, nello specifico:

- controllo delle classi realmente usate tra quelle importate, con Android Studio;
- controllo e segnalazione di eventuali variabili non usate o metodi privati non utilizzati, con Android Studio;
- *Inspection* e *Walkthrough* manuale.

Oltre a questo, nonostante non abbia prodotto dei *test* a livello di codice, la più importante verifica che ho fatto durante lo sviluppo di ogni funzionalità, è testare l'applicazione per quella particolare funzionalità, in modo da trovare eventuali malfunzionamenti. Ogni singola funzionalità, oltre che l'applicazione nella sua interezza, l'ho testata a fondo mediante utilizzo pratico dell'applicazione, andando a provare anche tutti i possibili casi limite.
Questo l'ho fatto eseguendo l'applicazione sul mio telefono mentre era connesso al *computer*. Una funzionalità molto utile di Android Studio, che mi ha permesso di riprovare l'applicazione subito dopo una modifica e in brevissimo tempo (nemmeno un secondo), è l'*hot reload*, grazie alla quale basta modificare il codice e salvare per aggiornare l'applicazione rispetto alle modifiche fatte, senza doverla eseguire nuovamente. Questo strumento ha migliorato molto la velocità di *debugging* dell'applicazione.
In generale ho eseguito l'applicazione in modalità *debug* ma in alcuni casi l'ho eseguita in modalità *profile* (modalità intermedia tra *debug* e *release*), che è una modalità che permette di avere prestazioni simili a quelle reali (*release*) e in questo modo è possibile verificarne le prestazioni in determinati scenari (ad esempio per verificare le prestazioni dello *scroll* delle liste di prodotti).

3.4 Risultati raggiunti

Le funzionalità che ho prodotto in questi due mesi di *stage* si possono riassumere con la seguente immagine:

Figura 3.16: Funzionalità prodotte

3.4.1 Copertura di requisiti e di *testing*

Il lavoro che ho svolto ha coperto il 100% dei requisiti, quindi sia quelli obbligatori, sia quelli desiderabili sia quelli opzionali. Di conseguenza, gli obiettivi aziendali per il progetto (sezione 2.2) li ho tutti soddisfatti, sia quelli primari sia quelli secondari.
Ogni funzionalità dell'applicazione, oltre che l'applicazione nella sua interezza, è stata da me testata a fondo secondo i metodi di verifica descritti nella sezione 3.3.5, pertanto, anche qui, la copertura è del 100%.

3.4.2 Quantità e qualità dei prodotti

In merito alla quantità di prodotti, in questi due mesi di *stage*, ho prodotto 6317 linee di codice sorgente (SLOC).
Per quanto riguarda invece la qualità dei prodotti che ho sviluppato, quelle di seguito sono alcune metriche comunemente utilizzate per valutare la qualità del codice, applicate ai singoli metodi e alle singole funzioni:

Metrica	Valore massimo	Valore ottenuto
Complessità ciclomatica	20	10
Linee di codice sorgente	70	60
Numero di parametri	4	2
Massima nidificazione	5	3

Tabella 3.3: Metriche di qualità

(i valori scritti nella colonna dei valori ottenuti corrispondono, per ogni metrica, al massimo tra i valori ottenuti calcolando quella metrica su tutti i metodi e su tutte le funzioni)

Dove:

- **complessità ciclomatica**: misura il numero di cammini linearmente indipendenti attraverso il grafo di controllo di flusso in un metodo o in una funzione;
- **linee di codice sorgente**: misura il numero di linee di codice di un metodo o in una funzione, senza tener conto di linee vuote o commenti;
- **numero di parametri**: misura il numero di parametri di un metodo o in una funzione;
- **massima nidificazione**: misura il massimo livello di nidificazione di blocchi di codice presenti in un metodo o in una funzione.

Capitolo 4

Visione retrospettiva

4.1 Soddisfacimento obiettivi

4.1.1 Obiettivi dell'azienda

Di seguito intendo riportare gli obiettivi aziendali primari e secondari (paragrafo 2.2) con il loro grado di soddisfacimento (**soddisfatto**/**non soddisfatto**).

Obiettivi primari	**Soddisfacimento**
Login, con possibilità di reimpostazione della *password*	Soddisfatto
Lista dei prodotti suddivisi in categorie in base alla *location* selezionata	Soddisfatto
Carrello	Soddisfatto
Ricarica del borsellino tramite Stripe	Soddisfatto
Acquisto di un ordine (uno o più prodotti)	Soddisfatto
Storico degli ordini	Soddisfatto

Tabella 4.1: Soddisfacimento obiettivi aziendali primari per il progetto

Obiettivi secondari	**Soddisfacimento**
Autoselezione della *location* all'apertura dell'applicazione in base alla vicinanza geografica con le *vending machine* disponibili sul territorio	Soddisfatto
Preferiti	Soddisfatto
Associazione *account* Nexudus	Soddisfatto

Modifica dei dati dati di fatturazione	Soddisfatto
Modifica della *password*	Soddisfatto
Storico delle transazioni (sia delle ricariche sia degli acquisti)	Soddisfatto

Tabella 4.2: Soddisfacimento obiettivi aziendali secondari per il progetto

Come è possibile vedere, ho soddisfatto il 100% degli obiettivi prefissati dall'azienda per il progetto assegnatomi.

4.1.2 Obiettivi personali

Gli obiettivi personali che mi ero prefissato prima di iniziare lo *stage* (paragrafo 2.4), li ho tutti soddisfatti con alto grado di soddisfacimento. In particolare: ho approfondito molto la mia conoscenza del *framework* Flutter e ora sono molto più cosciente di come lavori "sotto", come usarlo al meglio e quanto sia potente e utile come strumento di sviluppo, soprattutto per diminuire i costi e i tempi di realizzazione. Ho imparato nuove tecnologie, in particolare GraphQL e Stripe. Di GraphQL ho compreso l'intento e ho capito in che modo si fanno i vari tipi di richiesta e come interpretarne le risposte. Con Stripe ho potuto apprendere uno strumento che permette di integrare la funzionalità di pagamento all'interno di un'applicazione. Infine ho scoperto il mondo del lavoro e cosa significhi lavorare, non solo rispetto al lavoro in sè, in quanto scrivere codice mi era già capitato in altre occasioni, ma mi ha fatto capire il lavoro di *developer* nel suo insieme, e cioè: la gestione di un progetto, come si dividono i compiti tra i membri di un *team* di progetto, il fatto che ci siano delle priorità sulle funzionalità in base alle tempistiche per il quale il cliente desidera avere un primo riscontro pratico, il rapporto di comunicazione attiva tra membri di un *team* di progetto per lavorare al meglio accordandosi su scelte di *design* (di ogni tipo), il relazionamento con gli altri dipendenti. Tutto questo mi ha sicuramente fatto crescere sia professionalmente sia personalmente.

4.2 Maturazione professionale

4.2.1 Conoscenze

Lo *stage* mi ha permesso di:

- **Approfondire notevolmente la conoscenza di Flutter**: come organizzare un progetto al meglio per rendere il codice leggibile, riutilizzabile e manutenibile; come implementare l'interfaccia grafica ricercando le *performance*, come separare al meglio il *presentation layer* dal *business layer*; come rendere l'applicazione internazionalizzabile, ecc.
- **Imparare GraphQL**: come effettuare richieste con le *query* e *mutation* esposte da un certo *endpoint*, come interpretare la risposta e quindi eventuali errori.
- **Imparare Stripe**: come integrare la possibilità di effettuare pagamenti *in-app* in modo semplice e sicuro, utilizzando le API esposte da un *endpoint* Stripe per creare e gestire pagamenti.

4.2.2 Abilità

Con lo stage ho potuto ampliare le seguenti abilità:

- **Problem solving**: ho dovuto trovare le soluzioni migliori per ogni tipo di problema che mi si presentava, perseguendo: efficienza, *performance*, manutenibilità, leggibilità e riusabilità di codice. Le mie abilità nel *problem solving* è aumentata per il semplice motivo che mi sono trovato a dover risolvere problemi che non mi erano capitati prima, cioè a me nuovi.

- **Comunicative**: è stato necessario per me comunicare molto con gli altri membri del *team* del progetto su cui ho lavorato, e loro con me, in quanto tutti abbiamo lavorato su parti diverse dello stesso prodotto finale, di conseguenza perchè si possa produrre un risultato ottimo c'è bisogno di un'ottima comunicazione/collaborazione, in particolare, per accordarsi su: scelte di *design*, priorità da dare alla realizzazione delle varie funzionalità, discussione e risoluzione di problematiche relative al progetto, ecc.

4.3 Distanza tra università e lavoro

Rispetto alle competenze richieste dallo *stage* che ho affrontato e dal lavoro in generale, sicuramente c'è un divario rispetto a quanto l'università ha saputo insegnarmi in questi tre anni. Questo perchè tutte le tecnologie che ho utilizzato erano nuove, mai viste nei vari corsi offerti dal corso di studio in Informatica. Ma non ho avuto problemi nell'impararle autonomamente, in quanto credo che l'università prepari proprio a questo: l'obiettivo non è insegnare tutte le tecniche e tecnologie esistenti, ma dare degli *input* specifici ed insegnare le basi in un modo tale che lo studente sviluppi l'abilità di apprendere rapidamente una nuova tecnica o tecnologia in modo autonomo.
Ed è ciò che capita, ad esempio, nel corso di Ingegneria del Software: tutti i progetti offerti dalle aziende richiedono l'utilizzo di tecnologie che sicuramente prima non si conoscevano, o almeno non tutte, ma, giustamente, nessuno ci ha fatto una preparazione sulle tecnologie che avremmo dovuto utilizzare, invece, in modo autonomo, abbiamo fatto un periodo di studio delle tecnologie. Questo ha permesso ad ogni studente che abbia affrontato il progetto del corso di Ingegneria del Software di ampliare la propria abilità di auto-apprendimento, ed è quindi stato formativo, soprattutto in vista dello *stage*, in cui abbiamo replicato la stessa esperienza, ma nel mondo del lavoro vero e proprio.
Per questo credo che ci sia sicuramente un divario tra le conoscenze erogate dall'università e quelle richieste in un qualunque stage, penso sia evidente a chiunque, ma credo che questo divario sia colmato dall'abilità che si sviluppa ad imparare autonomamente le cose.

4.4 Conclusioni

In conclusione, l'esperienza che ho vissuto durante il periodo di *stage* l'ho trovata molto utile e formativa, per le conoscenze apprese e le abilità ampliate.
Penso che lo *stage* sia la perfetta attività conclusiva del percorso di studi della laurea triennale, in quanto mi ha permesso di colmare il divario che il mondo del lavoro ha rispetto all'università.

Ho messo alla prova le mie conoscenze, abilità e capacità di apprendimento, che si sono rivelate molto buone e adatte ad affrontare un'esperienza lavorativa da *mobile developer*.
Grazie a questo progetto di *stage* ho avuto la possibilità di riscoprire il mondo dello sviluppo *mobile*, che avevo scoperto, per la prima volta, negli ultimi due anni di scuola superiore, anni in cui ho imparato le basi dello sviluppo nativo in Android, e in cui avevo già intuito che probabilmente sarebbe stata la mia strada. Questo *stage* mi ha permesso di confermare questa mia percezione.
Il lavoro che ho prodotto mi ha altamente soddisfatto, specialmente in termini di efficienza con il quale l'applicazione gestisce le risorse, di robustezza (in quanto per adesso non ci sono *bug* noti) e di ricerca di *performance* nella gestione dei vari componenti grafici. Anche il mio *tutor* aziendale è rimasto molto soddisfatto del mio lavoro, e mi ha proposto di rimanere a lavorare come dipendente nella figura di *mobile developer*.
Ho deciso di rimanere e quindi di non proseguire, almeno per adesso, la laurea magistrale, perchè oltre a piacermi l'azienda e oltre a trovarmi particolarmente bene con Flutter, penso che Flutter sia il futuro dello sviluppo *mobile*, per i seguenti motivi:

- rapidità di realizzazione e quindi basso costo;
- è multi-piattaforma, quindi con un solo codice sorgente si può fare il *deploy* dell'applicazione su più piattaforme, selezionabili tra; *Android*, *iOS*, *web*, *desktop* (Windows, MacOS, Linux);
- il *team* di Google che si occupa di Flutter rilascia continuamente aggiornamenti, che, oltre a portare nuove funzionalità e oltre ad ampliare il *framework*, migliorano le *performance*, assottigliando sempre di più il divario con lo sviluppo nativo;
- l'utilizzo di Flutter è cresciuto esponenzialmente negli ultimi anni.

Acronimi e abbreviazioni

API Application Program Interface. 11, 24, 34, 39

CRUD Create, Read, Update, and Delete. 24

IDE Integrated Development Environment. 12

UML Unified Modeling Language. 16, 39

Glossario

API in Informatica con il termine *Application Programming Interface API* (interfaccia di programmazione di un'applicazione) si indica un insieme di procedure disponibili al programmatore, di solito raggruppate a formare un *set* di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'*hardware* e il programmatore o tra *software* a basso livello e quello ad alto livello semplificando così il lavoro del programmazione. 37

Inspection si esegue una lettura mirata del codice focalizzando la ricerca su aspetti ben definiti, contenuti nella lista di controllo. 29

Plugin è un programma non autonomo che interagisce con un altro programma per ampliarne o estenderne le funzionalità originarie.. 12

UML in Ingegneria del Software UML, *Unified Modeling Language* (linguaggio di modellazione unificato) è un linguaggio di modellazione e di specifica basato sul paradigma *object-oriented*. L'UML svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione a oggetti. Gran parte della letteratura di settore usa tale linguaggio per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico. 37

Walkthrough una forma di revisione del materiale statica, cioè che non richiede l'esecuzione dello stesso, al fine di rilevare la presenza di difetti ed eseguire una lettura critica del prodotto in esame. La strategia è percorrere il codice simulandone l'esecuzione. 29

Bibliografia

Siti web consultati

Sito di Moku. URL: https://www.moku.io/it.