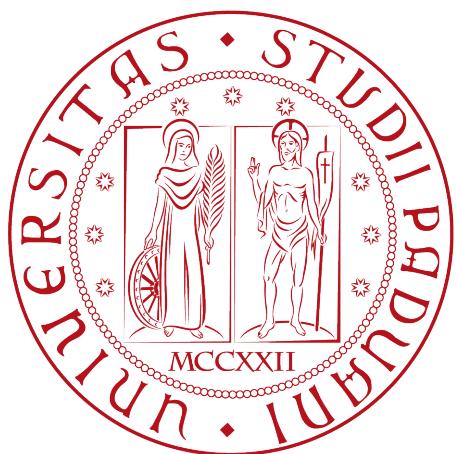


Università degli Studi di Padova
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"
CORSO DI LAUREA TRIENNALE IN INFORMATICA



**Sviluppo di un'app multipiattaforma legata
al mondo wellness con Flutter**

Tesi di laurea

Relatore

Prof. Paolo Baldan

Laureando

Giacomo Sassaro

ANNO ACCADEMICO 2020-2021

Giacomo Sassaro: *Sviluppo di un'app multipiattaforma legata al mondo wellness con Flutter*, Tesi di laurea, © Settembre 2021.

Sommario

Il presente documento è il resoconto della mia attività di tirocinio, della durata di circa trecentoventi ore, presso l'azienda Datasoil S.r.l.

L'obiettivo principale è stato la progettazione e l'implementazione di un'applicazione multipiattaforma legata al mondo del wellness/fitness a supporto della piattaforma di analisi dati realizzata per un prodotto spinoff dell'azienda. Il tutto con un focus sull'usabilità e l'engagement/gamification.

Il prodotto è stato sviluppato utilizzando Dart come linguaggio di programmazione e Flutter come framework. In questo modo è stato possibile scrivere un'unica applicazione che possa funzionare sia su dispositivi Android, sia su dispositivi iOS.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Paolo Baldan, relatore della mia tesi, per l'aiuto che mi ha fornito nella stesura di questo documento.

Un ringraziamento speciale va ad Andrea e Pietro, che sono sempre stati disponibili quando avevo bisogno di chiarimenti, e in generale a tutto il team Datasoil S.r.l. che mi ha permesso di vivere questa esperienza in un ambiente sereno e amichevole.

Desidero ringraziare con affetto la mia famiglia per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Infine, desidero i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Settembre 2021

Giacomo Sassaro

Indice

1	Introduzione	1
1.1	L'azienda	1
1.2	Il progetto e lo stage	2
1.2.1	Descrizione	2
1.2.2	Obiettivi	2
1.2.3	Pianificazione del lavoro	2
1.3	Prodotto ottenuto	3
1.4	Difficoltà incontrate	3
1.5	Struttura della relazione	4
2	Strumenti utilizzati	5
2.1	Strumenti di sviluppo	5
2.1.1	Visual Studio Code	5
2.1.2	Android Studio	5
2.1.3	Docker	6
2.1.4	Dart	6
2.1.5	Flutter	6
2.2	Strumenti organizzativi	7
2.2.1	Slack	7
2.2.2	Jira	7
2.2.3	Github	8
3	Analisi dei requisiti	9
3.1	Requisiti individuati	9
3.1.1	Requisiti app cliente	9
3.1.2	Requisiti app coach	11
3.2	Riepilogo requisiti	13
3.2.1	Riepilogo requisiti app cliente	13
3.2.2	Riepilogo requisiti app coach	13
4	Progettazione	15
4.1	Organizzazione repository e sistema di ticketing	15
4.2	Scelta framework Flutter	16
4.3	Onboarding e primo login	16
4.3.1	Autenticazione tramite auth0	17
4.3.2	Chiamate asincrone	17
4.3.3	Richieste API	18
4.3.4	API di LifestyleSync	20

4.3.5	Stato globale e locale	23
4.3.6	Routing iniziale	23
4.4	Connessione ai device	24
4.5	Dashboard utente	24
4.5.1	Homepage	24
4.5.2	Move	25
4.5.3	Food	25
4.5.4	Health	25
4.5.5	Agenda	25
4.5.6	Chat	25
4.6	Applicazione coach	26
5	Sviluppo	27
5.1	Onboarding e primo login	27
5.1.1	Widget di routing iniziale	27
5.1.2	User model	30
5.1.3	Introduzione all'app e questionario	31
5.2	Connessione ai device	32
5.3	Dashboard utente	33
5.3.1	Models	33
5.3.2	Homepage	35
5.3.3	Move	36
5.3.4	Agenda	37
5.3.5	Chat	38
5.4	Applicazione coach	39
6	Conclusioni	41
6.1	Valutazione dei risultati ottenuti	41
6.2	Sviluppi futuri	41
6.3	Conoscenze acquisite	42
6.4	Utilizzo di Flutter	42
6.5	Valutazione personale	42
Sitografia		47

Elenco delle figure

1.1	Logo Datasoil S.r.l	1
2.1	Logo di Visual Studio Code	5
2.2	Logo di Android Studio	5
2.3	Logo di Docker	6
2.4	Logo di Dart	6
2.5	Logo di Flutter	6
2.6	Logo di Slack	7
2.7	Logo di Jira	8
2.8	Logo di GitHub	8
4.1	Esempio di ticket su Jira	15
4.2	Diagramma di sequenza dell'autenticazione usando auth0	17
4.3	Chiamate sincrone VS asincrone	18
4.4	Diagramma di sequenza di una richiesta API	19
5.1	Prime schermate dell'applicazione	29
5.2	Modello di User	30
5.3	Alcune schermate onboarding	32
5.4	Schermata del profilo cliente con sezione relativa ai profili fitness	33
5.5	Modelli dei dati	35
5.6	Homepage e dettagli eventi	36
5.7	Schermata Move	37
5.8	Homepage e dettagli eventi	38
5.9	Schermate app coach	39

Elenco delle tabelle

3.1	Tabella del tracciamento dei requisiti funzionali	11
3.2	Tabella del tracciamento dei requisiti vincolo	11
3.3	Tabella del tracciamento dei requisiti funzionali	12
3.4	Tabella del tracciamento dei requisiti vincolo	12
3.5	Numero di requisiti per importanza app cliente	13
3.6	Numero di requisiti per tipologia app cliente	13
3.7	Numero di requisiti per importanza app coach	13
3.8	Numero di requisiti per tipologia app coach	13

Elenco dei frammenti di codice

4.1	User provider	19
4.2	JSON schema dell'API GET api/myself	20
4.3	JSON schema dell'API GET api/goals	21
4.4	JSON schema dell'API GET api/agendas	22
5.1	Widget CheckLogin	28
5.2	Classe DelayedWidget	29
5.3	Costruzione delle card con individuazione a runtime del tipo	37

Capitolo 1

Introduzione

1.1 L'azienda



Figura 1.1: Logo Datasoil S.r.l.

Datasoil S.r.l. è una startup innovativa che investe nel capitale umano al fine di formare sviluppatori che riescano a prendere decisioni e responsabilità di progetto avanzate. La loro mission è quella di impiegare tecnologie sempre più innovative e portare novità ai loro clienti.

Per questi motivi i nuovi sviluppatori junior saranno sempre inseriti su progetti/prodotti core dell'azienda e seguiti da sviluppatori senior full time sulle attività di formazione, progettazione e chiarimenti nel day to day.

Datasoil S.r.l. sviluppa piattaforme per l'industry 4.0 che permettono di fondere le informazioni e gli eventi provenienti da diversi livelli aziendali per creare insight proattivi integrati in tempo reale. Ogni evento, che sia un malfunzionamento o un'eccezione segnalata da altri operatori, viene notificato alla persona giusta al momento giusto anticipando la gestione delle criticità interfunzionali.

Inoltre, in quest'ultimi anni di pandemia, Datasoil S.r.l. ha deciso di sfruttare la competenza dei loro dipendenti per aiutare le aziende a garantire la sicurezza dei lavoratori. Per fare ciò sono stati utilizzati degli smartband, i quali permettono il tracciamento completamente anonimo dei contatti a rischio e garantiscono il rispetto del distanziamento sociale. Nel pieno rispetto della privacy e GDPR gli individui a rischio vengono avvisati da una vibrazione del braccialetto nel caso di contagio.

1.2 Il progetto e lo stage

1.2.1 Descrizione

Il progetto proposto prevede l'implementazione di due applicazioni mobile multipiattaforma: **LifestyleSync** e **LSCoach**.

LifestyleSync, legata al mondo wellness/fitness, serve come supporto della piattaforma di analisi dati realizzata per un prodotto spinoff di Datasoil.

L'idea alla base del progetto è quella di fornire uno strumento per migliorare il benessere dei propri dipendenti alle aziende.

Grazie a questo prodotto ogni dipendente, o cliente, sarà seguito da un coach che lo aiuterà a mantenersi in forma, sia tramite obiettivi di movimento ed allenamento, sia tramite obiettivi alimentari, inoltre i parametri di salute del cliente saranno rilevati e trasmessi al proprio coach in modo da predisporre allenamenti personalizzati.

Lo scopo dell'applicazione è quello di fornire un'interfaccia mobile ai clienti, interfaccia tramite la quale potranno interagire con il proprio coach, visualizzare gli obiettivi a loro assegnati, gli appuntamenti e il loro percorso a livello di fitness.

Per garantire al coach la possibilità di prestare i propri servizi ai clienti in modo unificato, si è resa necessaria la progettazione di una seconda applicazione specifica: **LSCoach**. Il coach potrà vedere i suoi appuntamenti con i clienti, lo stato di forma e gli obiettivi assegnati ad ogni cliente.

1.2.2 Obiettivi

- Studio e comprensione del linguaggio di programmazione *Dart* e del framework *Flutter*;
- Progettazione, completo sviluppo e verifica delle funzionalità di onboarding e primo login dell'app cliente: breve introduzione all'uso dell'applicazione, sistema di autenticazione e questionario conoscitivo;
- Progettazione, completo sviluppo e verifica delle funzionalità di connessione ai device dell'utente dell'app cliente: connessione ai profili fitness Google Fit, Fitbit e Garmin per il rilevamento di dati;
- Progettazione, sviluppo e verifica delle funzionalità individuate per la dashboard utente dell'app cliente: obiettivi di movimenti, lista degli obiettivi raggiunti e falliti, agenda con gli appuntamenti, pagina del profilo con calcolo del **BMI**;
- Progettazione, inizio dello sviluppo e verifica delle funzionalità principali per l'applicazione dei coach: visualizzazione dei clienti con i relativi obiettivi in corso e agenda con gli appuntamenti.

1.2.3 Pianificazione del lavoro

- **Settimana 1:** configurazione dell'ambiente di lavoro e studio di Flutter;
- **Settimana 2:** progettazione e sviluppo delle prime funzionalità onboarding: introduzione all'applicazione;
- **Settimana 3:** implementazione del login tramite auth0, progettazione e sviluppo del sondaggio iniziale;

- **Settimana 4:** progettazione e sviluppo delle prime schermate di dashboard utente: Homepage, Device Connection e Profilo;
- **Settimana 5:** progettazione e sviluppo di ulteriori schermate: Obiettivi con dettaglio, appuntamenti con dettaglio;
- **Settimana 6:** progettazione e sviluppo della agenda con i relativi impegni settimanali e mensili;
- **Settimana 7:** testing di diversi sdk per l'implementazione di una chat, progettazione e sviluppo app coach;
- **Settimana 8:** progettazione e sviluppo di un sistema di notifiche push con l'utilizzo di Google Firebase;
- **Settimana 9:** raffinamento e completamento delle funzionalità precedentemente sviluppate.

1.3 Prodotto ottenuto

Il prodotto ottenuto si compone di due diverse applicazioni: applicazione cliente e applicazione coach.

Entrambe le applicazioni sono state sviluppate utilizzando *Flutter* come framework in modo da valutarne anche le potenzialità, infatti l'azienda ha sempre utilizzato React Native per lo sviluppo mobile. Mentre per lo stile dei componenti dell'applicazione si è deciso di utilizzare il *Material Design* di Google già ampiamente utilizzato da Datasoil S.r.l.

L'applicazione cliente allo stato attuale permette la visione dei propri obiettivi di movimento e degli appuntamenti, oltre che una schermata di profilo personale con la possibilità di associazione ai diversi profili fitness al fine di rilevare dati.

L'app cliente è quindi a circa un terzo dello sviluppo, l'azienda ora potrà continuare con lo sviluppo delle restanti sezioni relative alla dieta e alla visualizzazione dello stato di salute del cliente.

L'applicazione coach allo stato attuale permette la visualizzazione della lista dei propri clienti con i relativi obiettivi di movimento e la visualizzazione di un'agenda con gli appuntamenti. L'app coach è in buona parte sviluppata, l'idea aziendale è quella che dall'app il coach possa solo vedere ciò che ha già assegnato dall'interfaccia web (prodotto già sviluppato da Datasoil S.r.l.).

L'azienda quindi dovrà solo aggiungere la visualizzazione della dieta e dei parametri di salute del cliente, omettendo la possibilità di assegnare ulteriori obiettivi dall'app mobile. Per entrambe le applicazioni è stata predisposta una versione minimale di chat in modo che il cliente possa comunicare con il proprio coach. Questa chat sarà in futuro ampiamente rivista in modo da permettere lo scambio di file multimediali e di video chiamate, oltre che limitare lo scambio di messaggi tramite una sorta di abbonamento.

1.4 Difficoltà incontrate

La principale difficoltà incontrata durante lo svolgimento del progetto è stata l'apprendimento del linguaggio di programmazione *Dart* e soprattutto del framework **Flutter**. Flutter permette di scrivere in modo semplice applicazioni mobile multipiattaforma.

Flutter è un framework più performante di React Native in termini di *rendering* UI, ma allo stesso tempo, vista la sua recente esplosione è ancora poco conosciuto e la documentazione online non è troppo vasta.

Per superare questa difficoltà è servito un breve periodo di studio del framework seguendo le guide presenti nel suo sito e la codifica di diverse applicazioni demo.

Un'altra difficoltà è stato l'apprendimento e la progettazione delle questioni *core* dell'applicazione, come la gestione delle richieste asincrone al backend, la gestione degli errori e la gestione dello stato dell'applicazione.

La soluzione a questa difficoltà è stata la comunicazione frequente con il mio tutor, il quale ha chiarito i miei dubbi principali e mi ha consigliato delle applicazioni demo da sviluppare per assodare i concetti.

Un altro problema è sorto durante lo sviluppo, alcune librerie che si intendeva utilizzare avevano bisogno di alcune migliorie e correzione di bug. È stato quindi necessario fare un [fork](#) alle librerie in questione ed effettuare le diverse modifiche necessarie.

Un'altra difficoltà è sorta nelle ultime settimane: si è deciso di sviluppare una seconda applicazione da utilizzare a lato coach riutilizzando buona parte dei componenti già sviluppati per i clienti.

La difficoltà è stata quella di rendere questi componenti parametrizzabili e di rivedere interamente la struttura della *repository*, oltre che impostare le corrette configurazioni in modo da poter compilare due diverse applicazioni utilizzando dei file in comune.

1.5 Struttura della relazione

Il secondo capitolo descrive gli strumenti e le tecnologie utilizzate per svolgere il progetto, particolare attenzione è posta sulla presentazione del framework Flutter, in quanto è alla base del prodotto ottenuto;

Il terzo capitolo espone un'analisi tecnica dei requisiti individuati che il prodotto finale deve soddisfare al termine del periodo di stage;

Il quarto capitolo descrive le scelte progettuali adottate per ottenere una struttura solida manutenibile;

Il quinto capitolo espone il percorso di realizzazione del prodotto e le soluzioni alle difficoltà incontrate;

Il sesto capitolo descrive l'esperienza di stage e i risultati ottenuti.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Strumenti utilizzati

2.1 Strumenti di sviluppo

2.1.1 Visual Studio Code



Figura 2.1: Logo di Visual Studio Code

Visual Studio Code è un editor di codice sorgente sviluppato da Microsoft. Include il supporto per il *debugging*, un controllo per Git integrato, controllo della sintassi e *refactoring* del codice. Inoltre, grazie alle molte estensioni sviluppate per l'editor è possibile utilizzare *shortcut*, auto completamento del codice, ed altre funzioni utili. Per Flutter in particolare esiste una sua estensione che rende disponibile, oltre che all'auto completamento e al *refactoring* del codice, anche degli strumenti sviluppatore utili per monitorare il traffico dell'app, l'impianto grafico e la gestione della memoria.

2.1.2 Android Studio



Figura 2.2: Logo di Android Studio

Android Studio è un ambiente di sviluppo integrato per lo sviluppo per la piattaforma Android. È possibile sviluppare con Flutter anche su Android Studio, ma personalmente ho preferito usare Visual Studio Code vista l'eccessiva pesantezza di questo IDE.

2.1.3 Docker



Figura 2.3: Logo di Docker

Docker è un progetto open-source che automatizza il processo di *deployment* di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux. Ho dovuto usare docker per virtualizzare nel mio computer la parte di backend durante la fase di sviluppo.

2.1.4 Dart



Figura 2.4: Logo di Dart

Dart è un linguaggio di programmazione sviluppato da Google con l'obiettivo di sostituire *javascript* per lo sviluppo web. Il compilatore Dart permette di scrivere programmi sia per il web sia per desktop e server attraverso l'uso di due diverse piattaforme:

- **Dart Native:** per dispositivi, include sia un VM per la compilazione Just-in-time, sia un compilatore per la produzione di codice eseguibile;
- **Dart Web:** per il web, include un compilatore per lo sviluppo e uno per la produzione.

2.1.5 Flutter



Figura 2.5: Logo di Flutter

Flutter è un framework open source sviluppato da Google per la costruzione di interfacce grafiche.

Flutter usa come linguaggio di programmazione Dart e si compone di quattro diversi componenti principali:

- **Dart platform:** Piattaforma per l'utilizzo del linguaggio Dart, comprende anche la Dart Virtual Machine sulla quale girano le applicazioni Flutter;
- **Flutter engine:** fornisce il supporto per il rendering a basso livello utilizzando la libreria grafica di Google, Skia Graphics. La caratteristica di spicco del Flutter Engine è quella di poter effettuare degli *hot-reload*, ossia visualizzare le modifiche effettuate al codice senza riavviare completamente l'app, ma iniettando il nuovo codice durante lo stato di esecuzione dell'app;
- **Foundation library:** fornisce le classi e le funzioni di base utilizzate per costruire applicazione che utilizzano Flutter ed è scritta in Dart. La creazione di interfacce con Flutter è effettuata per composizione di widget. Ogni widget è renderizzato dal metodo build() che viene chiamato ricorsivamente per ogni figlio, generando così un albero di widget;
- **Design-specific widget:** contiene due set di widget conformi a specifici linguaggi di programmazione. I widget in stile Material Design implementano il design di Google, mentre i widget di Cupertino imitano il design iOS di Apple;

Una libreria fondamentale per semplificare l'utilizzo di Flutter è **Flutter Hooks**. Questa cerca di replicare gli *Hooks* di *React*, ed è utile in quanto si evita di usare *Statefull widget* e *Stateless widget* per la costruzione dei widget. Con Flutter Hooks ogni widget eredita da *Hook widget*, e si può dare uno stato al widget usando l'hook *useState*. Inoltre viene anche più semplice la gestione dei side-effect e del ciclo di vita del widget grazie all'hook *useEffect*.

2.2 Strumenti organizzativi

2.2.1 Slack



Figura 2.6: Logo di Slack

Slack è un software che rientra nella categoria degli strumenti di collaborazione aziendale utilizzato per inviare messaggi in modo istantaneo ai diversi membri del team.

Slack è utilizzabile da browser, applicazione desktop e applicazione o mobile. Oltre ai messaggi diretti tra i membri è anche possibile creare diversi canali in modo da separare le conversazioni per argomento, per esempio nel nostro caso utilizzavamo due canali: #frontend, #backend. Slack inoltre permette di integrare nel sistema di messaggistica anche altri servizi, come per esempio Github o Jira.

2.2.2 Jira

Jira è un software utilizzato per gestire il lavoro collaborativo, segue il principio agile e permette di creare ticket e gestire degli sprint settimanali.



Figura 2.7: Logo di Jira

Nel nostro caso chiunque poteva aprire un ticket nella sezione *TODO* e assegnarlo a chi di dovere, specificandone i dettagli. Quando il ticket veniva preso in carico lo si spostava nell' apposita sezione *IN PROGRESS* e una volta terminato veniva chiuso nella sezione *DONE*. In questo modo ognuno sa in quale stato è il prodotto e a cosa sta lavorando ogni membro del team.

2.2.3 Github



Figura 2.8: Logo di GitHub

GitHub è un servizio di hosting per progetti software ed è una implementazione del sistema di versionamento distribuito Git.

Il sito è principalmente utilizzato dagli sviluppatori, che caricano il codice sorgente dei loro programmi e lo possono rendere disponibile al resto degli utenti. Questi ultimi possono interagire con lo sviluppatore tramite un sistema di *issue tracking*, *pull request* e commenti che permette di migliorare il codice del *repository* risolvendo bug o aggiungendo funzionalità.

Capitolo 3

Analisi dei requisiti

3.1 Requisiti individuati

I requisiti individuati a partire dal problema proposto e dalle discussioni con il tutor aziendale sono stati catalogati secondo il codice:

R[Importanza][Tipo][Codice]

dove:

- Importanza: specifica l'importanza del requisito e può assumere i seguenti valori:
 - **1:** Requisito obbligatorio;
 - **2:** Requisito desiderabile ma non essenziale per il funzionamento;
 - **3:** Requisito opzionale.
- Tipologia: specifica la tipologia del requisito e può assumere i seguenti valori:
 - **F:** *funzionale*, cioè determina una funzionalità dell'applicazione;
 - **V:** *vincolo*, che riguarda un vincolo che il prodotto deve rispettare.
- Codice: identificativo univoco del requisito espresso in forma gerarchica padre/fi-glio.

3.1.1 Requisiti app cliente

Requisito	Descrizione
R1F1	L'utente può utilizzare l'applicazione solo se autenticato
R1F2	Al primo accesso deve essere presentata una breve descrizione dell'applicazione.
R1F3	Al primo accesso deve essere presentato all'utente il suo coach.
R1F4	Al primo accesso l'utente deve compilare un questionario.

Requisito	Descrizione
R1F5	L'utente deve poter effettuare il logout dall'applicazione.
R1F6	L'utente deve poter vedere il totale dei suoi minuti attivi settimanali.
R1F6.1	L'utente deve poter vedere il totale dei suoi minuti attivi di attività settimanali.
R1F6.2	L'utente deve poter vedere il totale dei suoi minuti attivi non programmati settimanali.
R1F6.3	L'utente deve poter analizzare un confronto tra i suoi minuti attivi e il target stabilito dall'OMS
R1F7	L'utente deve poter vedere la lista dei suoi obiettivi attivi in quel determinato momento.
R1F7.1	Per ogni voce di un obiettivo (calorie, passi, minuti di attività, minuti non programmati) deve essere mostrato il progresso.
R1F7.2	L'utente deve poter sapere in quanti giorni scade un dato obiettivo.
R1F7.3	Per ogni obiettivo l'utente deve poter vedere le informazioni in dettaglio.
R1F8	L'utente deve poter vedere gli obiettivi passati o non attivi.
R1F8.1	L'utente deve poter scegliere per quale periodo vedere gli obiettivi passati.
R1F8.2	L'utente deve poter scegliere per un dato periodo se vedere gli obiettivi passati raggiunti o falliti.
R1F9	L'utente deve poter vedere un riepilogo con il totale degli obiettivi completati, falliti e in corso.
R1F10	L'utente deve poter vedere una lista con i prossimi appuntamenti del mese.
R1F10.1	Per ogni appuntamento l'utente deve poter vedere data, ora, titolo e coach.
R1F10.2	Per ogni appuntamento l'utente deve poter vedere le informazioni in dettaglio.
R1F11	L'utente deve poter vedere un calendario in cui saranno presenti tutti i suoi obiettivi attivi e i suoi appuntamenti passati e prossimi.
R1F11.1	Per ogni giorno del calendario l'utente deve poter vedere una lista con gli obiettivi del giorno e gli appuntamenti del giorno.
R1F12	L'utente deve poter associare il suo account dell'app a diversi servizi di fitness.

Requisito	Descrizione
R1F12.1	L'utente deve poter associare il suo account dell'app all'account di Google Fit
R1F12.2	L'utente deve poter associare il suo account dell'app all'account di Fitbit
R1F12.3	L'utente deve poter associare il suo account dell'app all'account di Garmin
R1F13	L'utente deve poter vedere una schermata con le sue informazioni.
R1F13.1	L'utente deve poter vedere la sua altezza, il suo peso, la sua età e il suo BMI .
R1F13.2	Deve essere presente una tabella di spiegazione per il BMI .
R1F14	L'utente deve poter vedere chi è il suo coach anche in seguito al primo accesso.
R2F15	L'utente deve essere avvisato al raggiungimento di determinati obiettivi tramite popup.
R2F15.1	Quando l'utente sblocca un achievement deve ricevere una notifica e vedere un popup.
R2F15.2	Quando l'utente completa un obiettivo deve ricevere una notifica e vedere un popup.
R2F15.3	Quando l'utente raggiunge i minuti attivi consigliati dall'OMS deve vedere un popup.

Tabella 3.1: Tabella del tracciamento dei requisiti funzionali

Requisito	Descrizione
R1V1	L'applicazione deve essere sviluppata usando Flutter.
R1V2	L'applicazione deve funzionare completamente su dispositivi Android.
R2V3	L'applicazione deve funzionare completamente su dispositivi iOS.
R1V4	L'interfaccia deve essere facilmente fruibile.
R2V5	L'applicazione deve presentare elementi di gamification .

Tabella 3.2: Tabella del tracciamento dei requisiti vincolo

3.1.2 Requisiti app coach

Requisito	Descrizione
R1F1	Il coach può utilizzare l'applicazione solo se autenticato
R1F2	Il coach deve poter effettuare il logout dall'applicazione.
R1F3	Il coach deve poter vedere una lista con i prossimi appuntamenti del mese.
R1F3.1	Per ogni appuntamento il coach deve poter vedere data, ora, titolo e cliente.
R1F3.2	Per ogni appuntamento il coach deve poter vedere le informazioni in dettaglio.
R1F4	Il coach deve poter vedere un calendario in cui saranno presenti tutti i suoi appuntamenti passati e prossimi.
R1F4.1	Per ogni giorno del calendario il coach deve poter vedere una lista con gli appuntamenti del giorno.
R1F5	Il coach deve poter scegliere il tenant di cui vedere i clienti.
R1F6	Il coach deve poter vedere i dettagli di ogni cliente
R1F6.1	Per ogni cliente il coach deve poter vedere l'altezza, il peso, l'età e il suo BMI
R1F6.2	Per ogni cliente il coach deve poter vedere gli obiettivi in corso a lui assegnati
R1F6.3	Per ogni cliente il coach deve poter vedere gli obiettivi passati a lui assegnati

Tabella 3.3: Tabella del tracciamento dei requisiti funzionali

Requisito	Descrizione
R1V1	L'applicazione deve essere sviluppata usando Flutter.
R1V2	L'applicazione deve funzionare completamente su dispositivi Android.
R2V3	L'applicazione deve funzionare completamente su dispositivi iOS.
R1V4	L'interfaccia deve essere facilmente fruibile.
R2V5	L'applicazione deve presentare elementi di gamification .

Tabella 3.4: Tabella del tracciamento dei requisiti vincolo

3.2 Riepilogo requisiti

3.2.1 Riepilogo requisiti app cliente

In totale sono stati individuati 39 requisiti, ripartiti tra le varie tipologie secondo quanto riportato nelle seguenti tabelle.

Importanza	#
Obbligatori	33
Desiderabili	6

Tabella 3.5: Numero di requisiti per importanza app cliente

Tipologia	#
Funzionali	34
Vincolo	5

Tabella 3.6: Numero di requisiti per tipologia app cliente

3.2.2 Riepilogo requisiti app coach

In totale sono stati individuati 17 requisiti, ripartiti tra le varie tipologie secondo quanto riportato nelle seguenti tabelle.

Importanza	#
Obbligatori	15
Desiderabili	2

Tabella 3.7: Numero di requisiti per importanza app coach

Tipologia	#
Funzionali	12
Vincolo	5

Tabella 3.8: Numero di requisiti per tipologia app coach

Capitolo 4

Progettazione

4.1 Organizzazione repository e sistema di ticketing

A livello di organizzazione del repository si è deciso di adottare lo stile generalmente utilizzato dall'azienda.

Partendo da una base stabile dell'applicazione si procede in modo incrementale per aggiunta di funzionalità seguendo un approccio [agile](#).

Ogni nuova funzionalità o *issue* viene segnalata attraverso Jira da un membro del team, il quale si occupa anche di fornire un titolo, una breve descrizione e di assegnarla ad uno sviluppatore frontend o backend, in base al tipo di funzionalità. Jira automaticamente crea un ticket con un codice identificativo e la descrizione del problema.

Una volta preso in carico un ticket, a livello di repository, si procede creando un nuovo [branch](#) con nome uguale al codice identificativo della *issue*. Al termine dello sviluppo per consolidare il lavoro svolto nel [branch](#) viene creata una [pull request](#) verso il [branch](#) principale, ossia il *main*. La richiesta, una volta controllata e priva di conflitti, viene confermata e tutto il codice prodotto viene unito nel [branch](#) principale, così da avere sempre un prodotto stabile e aggiornato all'ultima funzionalità aggiunta.

The screenshot shows a Jira ticket interface. At the top, there are buttons for 'Aggiungi epic' (Add epic), 'Aggiungi LS-94 CODICE IDENTIFICATIVO' (Add LS-94 IDENTIFYING CODE), and other standard Jira buttons like 'Allega' (Attach), 'Aggiungi un ticket figlio' (Add a child ticket), and 'Collega ticket' (Link ticket). The ticket title is '[LS-APP] OMS guidelines on Homepage'. Below the title, there are buttons for 'Completa' (Complete) and 'Completato' (Completed). The ticket body contains a list of requirements for the OMS duration metric:

- have a clear indicator for the progress
- see the current value for the duration
- see the OMS value
- see the comparison between the two values
- see an animated modal/splash when the user reaches OMS target (same as goal achievement)

Both for _a and _n

At the bottom, there's a comment section with a placeholder 'Aggiungi un commento...' (Add a comment...) and a note: 'Suggerimento: premere M per aggiungere un commento' (Tip: press M to add a comment).

Figura 4.1: Esempio di ticket su Jira

Si è deciso inoltre di mantenere entrambe le applicazioni sullo stesso *repository*, per questo motivo si sono dovute organizzare le cartelle in modo da avere un ambiente di lavoro ordinato. La scelta è stata quella di mantenere nella *root* del progetto delle cartelle contenenti i file condivisi tra le due applicazioni, e di creare altre due cartelle: *client* e *coach* aventi come figli delle cartelle con lo stesso nome di quelle presenti nella *root*. In queste ultime cartelle vengono raccolti i file utilizzati solamente da una delle due applicazioni. La struttura è risultata quindi la seguente:

- auth, contiene il Provider per l'autenticazione di entrambe le applicazioni
- models, contiene le classi del modello di entrambe le applicazioni
- pages, contiene i widget *fullscreen* utilizzati da entrambe le applicazioni
- widget, contiene i widget utilizzati da entrambe le applicazioni
- client, cartella contenente i file per l'applicazione client
 - providers, contiene i provider dell'app client
 - models, contiene le classi del modello dell'app client
 - pages, contiene i widget *fullscreen* dell'app client
 - widget, contiene i widget utilizzati dell'app client
- coach, cartella contenente i file per l'applicazione coach
 - providers, contiene i provider dell'app coach
 - models, contiene le classi del modello dell'app coach
 - pages, contiene i widget *fullscreen* dell'app coach
 - widget, contiene i widget utilizzati dell'app coach

4.2 Scelta framework Flutter

Inizialmente l'azienda non aveva ancora deciso se l'app sarebbe stata sviluppata con *Flutter* o con *React Native*, framework che loro utilizzano quotidianamente per altri progetti.

Sono state quindi analizzate le principali differenze tra i due framework fino a giungere alla conclusione di utilizzare *Flutter*: sia perché ritenuto su carta il più performante, sia per testarne i limiti.

Essendo infatti questo un progetto di dimensioni ridotte si è deciso di provare *Flutter* per testarne le potenzialità e capire se possa essere utilizzato dall'azienda anche per applicazioni di dimensioni maggiori.

4.3 Onboarding e primo login

Come prima parte dell'applicazione ho concluso la progettazione delle funzionalità onboarding e di primo login dell'utente, che in parte era già stata effettuata dal mio tutor.

4.3.1 Autenticazione tramite auth0

Per l'autenticazione si è deciso di appoggiarsi al servizio Auth0, già ampiamente utilizzato dall'azienda.

Grazie ad Auth0 ci è possibile delegare ad un servizio terzo la gestione dell'autenticazione attraverso diversi servizi come Google, Facebook o semplicemente con nome utente e password.

Auth0 fornisce una pagina web di autenticazione unica per qualsiasi provider indicato. A login avvenuto, l'applicazione riceve dall'API di Auth0 un ID token ed un Refresh Token.

Mediante l'ID token sarà possibile accedere alle API LifestyleSync tramite autenticazione **JWT**, includendolo in ogni header delle chiamate HTTP. In questo modo il backend può verificare che il token sia presente nel registro di Auth0 e che l'utente abbia i permessi per accedere ai dati. Tuttavia, prima di effettuare qualsiasi richiesta al backend, è buona pratica controllare la scadenza temporale indicata in ID token: se questo è scaduto è possibile utilizzare Refresh Token per chiedere ad Auth0 un nuovo ID token. Rinnovando i token prima della scadenza è quindi possibile mantenere l'accesso all'applicazione inserendo le credenziali soltanto al primo login.

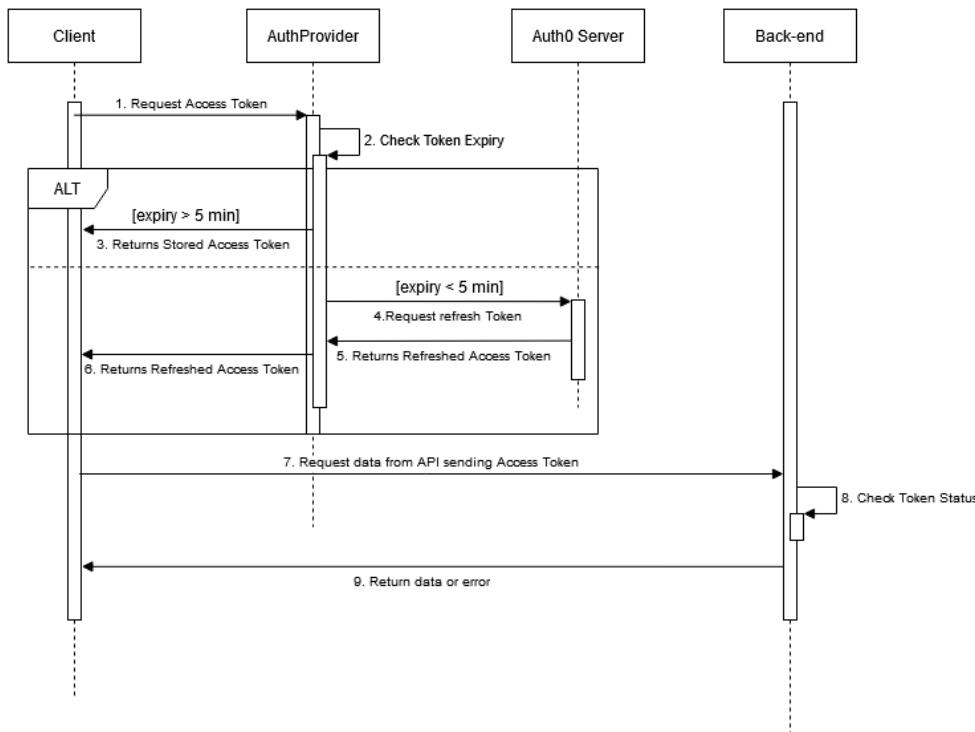


Figura 4.2: Diagramma di sequenza dell'autenticazione usando auth0

4.3.2 Chiamate asincrone

Un punto chiave per lo sviluppo dell'app è quello delle chiamate asincrone.

Quando una funzione richiede molto tempo per essere completata, come ad esempio una

funzione che richiede dati ad un server, è opportuno chiamarla in modo asincrono, così da non fermare tutta l'applicazione in attesa del completamento di quella funzione. Utilizzando le chiamate asincrone infatti il flusso delle operazioni che non necessitano il termine della chiamata può continuare indisturbato.

Dart mette a disposizione due modi per la gestione di questo tipo di chiamate, noi in accordo comune abbiamo deciso di usare il metodo *async* e *await*: le funzioni che prevedono la possibilità di fare chiamate asincrone sono contrassegnate con la parola chiave *async*. All'interno delle funzioni asincrone è possibile quindi usare la parola chiave *await* che serve per dire al programma di attendere quell'istruzione prima di proseguire con la successiva. Se non viene utilizzato il comando *await* allora tale funzione asincrona verrà eseguita parallelamente al proseguo del programma fino al suo completamento.

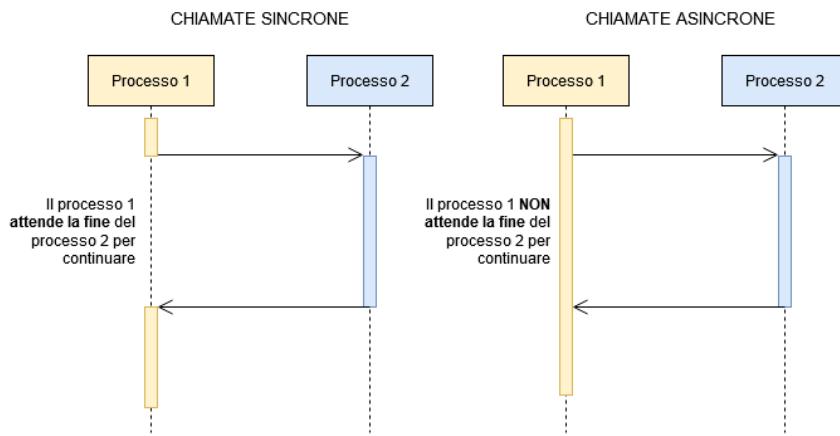


Figura 4.3: Chiamate sincrone VS asincrone

4.3.3 Richieste API

Essendo l'applicazione un supporto ad una piattaforma di analisi dati la presenza di chiamate asincrone al backend è molto frequente, è stato quindi fondamentale progettare un pattern solido da riutilizzare per ogni chiamata. Il *pattern* da noi adottato prevede che una qualsiasi chiamata API CRUD possa trovarsi in tre diversi stati:

- **Loading:** la chiamata è iniziata e non è ancora stata risolta;
- **Error:** la chiamata è terminata con un errore;
- **Response:** la chiamata è terminata e possiede i dati di mio interesse.

Appena prima di inviare la richiesta HTTP l'applicazione viene messa in uno stato di *loading*, nel caso di errori al backend o nessuna risposta ci si troverà in uno stato di *error*, altrimenti, nel caso in cui il backend risponda con i dati corretti ci si troverà in uno stato di *response*.

L'interfaccia dunque cambierà in base allo stato della chiamata, mostrando un indicatore di progresso mentre ci si trova in uno stato di *loading*, mostrando un errore nel caso ci si trovasse in uno stato di *error*, o mostrando i dati elaborati nel caso di *response*.

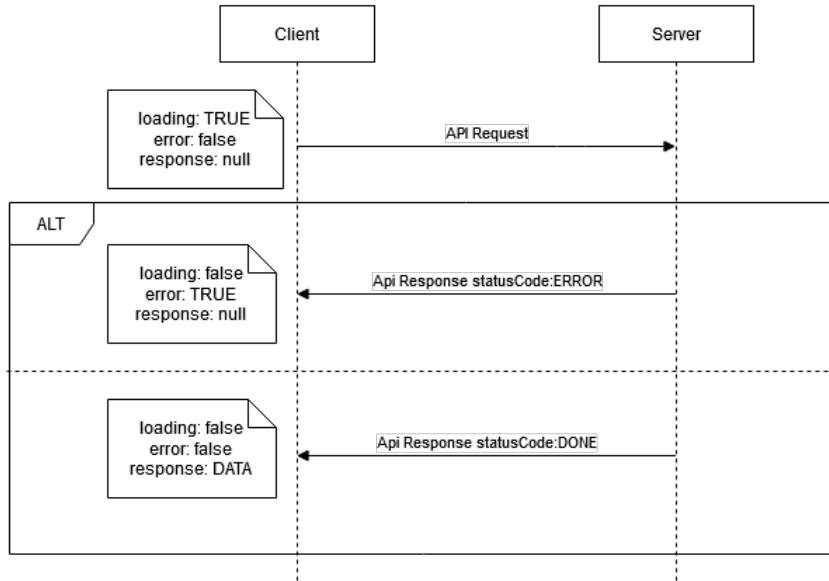


Figura 4.4: Diagramma di sequenza di una richiesta API

```
1 final userChangeNotifier =
2     ChangeNotifierProvider<UserRepo>((ref) => UserRepo(ref.read));
3 ///Fornisce lo stato di errore di [UserRepo].
4 final userError = Provider((ref) => ref.watch(userChangeNotifier).error);
5 ;
6 ///Fornisce il messaggio di errore di [UserRepo].
7 final userErrorMsg = Provider((ref) => ref.watch(userChangeNotifier).errorMsg);
8 ///Fornisce lo stato di loading di [UserRepo].
9 final userIsLoading =
10 Provider((ref) => ref.watch(userChangeNotifier).isLoading);
11 ///Fornisce l'istanza di [User].
12 final userProvider = Provider((ref) => ref.watch(userChangeNotifier).user);
13
14 class UserRepo extends ChangeNotifier {
15     bool error = false;
16     bool isLoading = false;
17     String errorMsg = "";
18     final String apiUrl = dotenv.env['API_URL'] ?? "";
19     User? user;
20     final Reader read;
21     UserRepo(this.read);
22     Future<void> fetchUser() async {
23         var token = await read(authProvider).getAccessToken();
24         this.isLoading=true;
25         notifyListeners();
26         try {
27             final response = await Dio().get(
28                 apiUrl + "/api/myself",
29                 options: Options(
30                     headers:
31                         <String, String>{'Authorization': 'Bearer $token'},);
32         if (response.statusCode == 200) {
```

```

32     this.user = User.fromJson(response.data));
33     this.isLoading = false;
34   } else {
35     this.isLoading = false;
36     this.error = value;
37     this.errorMsg = "richiesta fallita";
38   }
39 } catch (e) {
40   this.isLoading = false;
41   this.error = value;
42   this.errorMsg = "richiesta fallita";
43 } finally {
44   notifyListeners();
45 }
46 }
47 }
```

Listing 4.1: User provider

4.3.4 API di LifestyleSync

Di seguito sono riportate alcune delle API utilizzate durante lo sviluppo delle applicazioni.

Esse si rifanno ad un modello REST-like e non REST puro in quanto tutte le richieste HTTP vengono firmate mediante la tecnica [JWT](#) presentata in precedenza.

Le API verranno riportate nella forma [metodo_HTTP][url/api]

1. GET api/myself

Restituisce un file JSON che descrive l'utente, con le sue info e i dettagli del suo coach, con il seguente schema:

```

1  {
2    "user_id": String,
3    "given_name": String,
4    "family_name": String,
5    "email": String,
6    "picture": String,
7    "info": [
8      {
9        survey_done: bool,
10       coach: Map<String, dynamic>,
11       oms_thresh: Map<String, int>,
12       fit_profiles: Map<String, dynamic>
13     }
14   "user_metadata": Map<String, dynamic>
15 }
```

Listing 4.2: JSON schema dell'API GET api/myself

dove:

- user_id: è l'identificativo dell'utente nel database;
- given_name: è il nome dell'utente;
- family_name: è il cognome dell'utente;
- email: è la e-mail dell'utente;
- picture: è il link dell'immagine del profilo dell'utente;

- user_metadata: sono i meta-dati dell'utente;
- info:
 - survey_done: è true se l'utente ha completato il questionario;
 - coach: sono i dati del coach dell'utente;
 - oms_thresh: sono le soglie da raggiungere secondo l'OMS;
 - fit_profiles: sono i profili fit dell'utente;

2. GET api/goals

Questa API accetta diversi parametri obbligatori:

- end: data di fine dei goals che voglio in formato *IsoTime*;
- active: se voglio i goals attivi o meno in formato *Bool*;
- interval: intervallo di tempo antecedente a *end*, può essere: week, month, 3month o 6month in formato *String*.

Restituisce un array di oggetti JSON che descrivono i goals dell'utente, con il seguente schema:

```

1  {
2    "id": String,
3    "u_id": String,
4    "goals":
5      {
6        "calories":
7          {
8            "m": String,
9            "thresh": int,
10           }
11       "duration_activities":
12         {
13           "m": String,
14           "thresh": int,
15         }
16       "duration_normal":
17         {
18           "m": String,
19           "thresh": int,
20         }
21       "steps":
22         {
23           "m": String,
24           "thresh": int,
25         }
26     }
27   "status":
28     {
29       "calories": int,
30       "duration_activities": int,
31       "duration_normal": int,
32       "steps": int,
33     }
34   "start": IsoTime,
35   "end": IsoTime,
36   "achieved": bool,
37   "active": bool,
38 }
```

Listing 4.3: JSON schema dell'API GET api/goals

dove:

- **id**: è l'identificativo dell'obiettivo nel database;
- **u_id**: è l'identificativo dell'utente nel database;
- **goals**:
 - **calories**: sono le calorie da bruciare, m è il tipo di metrica e $thresh$ è la soglia;
 - **duration_activities**: sono i minuti di attività da raggiungere, m è il tipo di metrica e $thresh$ è la soglia;
 - **duration_normal**: sono i minuti attivi normali da raggiungere, m è il tipo di metrica e $thresh$ è la soglia;
 - **steps**: sono i passi da fare, m è il tipo di metrica e $thresh$ è la soglia;
- **status**:
 - **calories**: sono le calorie bruciate;
 - **duration_activities**: sono i minuti di attività raggiunti;
 - **duration_normal**: sono i minuti attivi normali raggiunti;
 - **steps**: sono i passi fatti;
- **start**: è la data di inizio dell'obiettivo;
- **end**: è la data di fine dell'obiettivo;
- **achieved**: indica se l'obiettivo è stato raggiunto;
- **active**: indica se l'obiettivo è attivo.

3. GET api/agendas

Questa API accetta diversi parametri obbligatori:

- **end**: data di fine degli appuntamenti che voglio in formato *IsoTime*;
- **interval**: intervallo di tempo antecedente a *end*, può essere: week, month, 3month o 6month in formato *String*.

Restituisce un array di oggetti JSON che descrivono gli appuntamenti, con il seguente schema:

```

1   {
2     "id": String,
3     "start": IsoTime,
4     "end": IsoTime,
5     "subject": String,
6     "tid": String,
7     "notes": String,
8     "coach": Map<String, dynamic>,
9     "client": Map<String, dynamic>,
10    }

```

Listing 4.4: JSON schema dell'API GET api/agendas

dove:

- **id**: è l'identificativo dell'appuntamento nel database;
- **start**: è la data e ora di inizio dell'appuntamento;
- **end**: è la data e ora di fine dell'appuntamento;

- subject: è il titolo dell'appuntamento;
- notes: sono le note dell'appuntamento;
- tid: è il tenant a cui appartiene l'appuntamento;
- coach: sono i dettagli del coach;
- client: sono i dettagli del cliente.

4.3.5 Stato globale e locale

Una delle problematiche principali quando si deve sviluppare un applicazione mobile o web è il come salvare i dati e gestire lo stato dell'applicazione.

Flutter prevede l'esistenza di due tipi di stato (simili ad uno *store*): lo stato globale dell'applicazione, che ad ogni update causa un update dell'intero albero di *rendering* dell'applicazione e lo stato locale, proprio del singolo widget e il cui update causa l'aggiornamento unicamente del widget stesso e dei suoi eventuali figli.

A seguito dell'analisi dei requisiti e del comportamento dell'applicazione si è deciso di utilizzare lo stato globale per i dati utilizzati da più widget con o senza legami di parentela. Mentre, seguendo il pattern [SRP](#), lo stato locale viene utilizzato dai widget che non condividono dati con altri componenti (ad esempio widget di pura presentazione o che rappresentano dati provenienti da un'API non sfruttata dall'intero applicativo).

Per la gestione dello stato globale abbiamo usato il provider pattern. Questo prevede l'utilizzo di un oggetto contenente i dati dell'applicazione che notifica ogni cambio di stato, il quale viene fornito all'applicazione utilizzando un *provider*. In questo modo l'interfaccia grafica sarà sempre aggiornata con il valore corrente dello stato.

Per la gestione dello store globale si è deciso di utilizzare la libreria *Riverpod*, questa fornisce una vasta scelta di *provider* diversi, e rende più facile l'accesso ai dati grazie ad un hook da loro predisposto: *useProvider*. Per lo store locale invece, in seguito alla scelta di usare Flutter Hooks, abbiamo sfruttato l'hook *useState*, questo costruisce una variabile che viene osservata dal componente, ad ogni cambiamento di questa variabile il componente si ricostruisce, aggiornandosi con il valore corrente dello stato.

4.3.6 Routing iniziale

Per questa prima parte del progetto è stato fondamentale progettare un buon routing per gestire correttamente i diversi casi di accesso.

Si è deciso per provare automaticamente il login all'avvio dell'applicazione utilizzando un token salvato nel *secure storage* dello smartphone. Nel caso di accesso avvenuto con successo si prosegue, altrimenti si rimanda l'utente ad una schermata di login.

A questo punto è necessario controllare se è la prima volta che il cliente accede all'applicazione o meno, per fare questo controllo vengono richieste al **backend** le informazioni dell'utente e nel mentre viene mostrata una schermata di caricamento, a questo punto le possibili alternative di routing sono:

- **Primo accesso:** viene mostrata all'utente una breve introduzione all'applicazione, gli viene presentato il suo coach, gli viene fatto rispondere ad un questionario rispetto al suo stile di vita e infine viene mandato all'homepage. Nel caso in cui l'esecuzione dell'app venisse interrotta prima del termine del questionario, l'accesso successivo sarà considerato nuovamente un primo accesso.

- **Altro accesso:** recupero i dati del cliente a cui viene mostrata l'homepage. In questo modo l'utente non ha più la possibilità di vedere l'introduzione all'app e di rispondere al questionario.

4.4 Connessione ai device

Altra tematica principale durante la fase di progettazione è stata quella della connessione della nostra app ai profili dei provider di fitness band e altri dispositivi in grado di rilevare passi, frequenza cardiaca e attività fisiche.

L'applicazione non prevede nessuna connessione diretta ai dispositivi, bensì ai profili dell'utente con i quali accede alle funzionalità di raccolta dati offerte dai provider. Ad esempio per rilevare i dati di durata di attività fisica e distanza percorsa da un utente che possiede una smartband Garmin, l'applicazione offrirà la possibilità di connettere il profilo LifestyleSync a quello Garmin.

Questa tematica era già stata trattata dall'azienda prima del mio arrivo e la soluzione scelta è stata quella di connettere la nostra app non direttamente ai device, ma all'account nelle rispettive app dei diversi device. Ossia, per rilevare i dati di una smartband di Garmin andiamo a collegare l'account del client in LifestyleSync al suo account nell'app Garmin.

I servizi che attualmente si prevede di utilizzare sono: Google Fit, Fitbit, e Garmin. L'idea è stata quindi quella di progettare una sezione all'interno dell'app, in cui l'utente possa vedere a quali account è già associato e potrà quindi disconnettersi, e a quali account può collegarsi. Cliccando su uno di questi nel caso di disconnessione verrà fatta una richiesta POST al *backend* che si occuperà della disconnessione, mentre nel caso di connessione l'utente verrà reindirizzato alla pagina di login del servizio scelto, una volta fatto il login verrà riportato alla pagina dell'app LifestyleSync e tramite una richiesta POST al *backend* verrà fatta l'associazione.

4.5 Dashboard utente

L'app dovrà avere diverse sezioni principali: *Homepage, Move, Food, Health*; e altre secondarie: *Agenda, Chat*.

Abbiamo quindi fin da subito pensato di sfruttare una struttura a pagine.

Per fare ciò si è resa quindi necessaria una barra di navigazione e un componente per scorrere tra le diverse pagine.

Da qui si sono progettate ad una ad una tutte le diverse schermate.

4.5.1 Homepage

È la pagina principale, qui l'utente deve poter vedere un breve riassunto del suo percorso e del suo stato di movimento.

Dovrà quindi essere presente il nome del cliente con la sua immagine del profilo e un piccolo badge con un resoconto degli obiettivi in corso, falliti, completati e il badge relativo all'ultimo *achievement* sbloccato.

Dovrà essere presente un indicatore dei minuti attivi settimanali a confronto con i minuti consigliati dall'OMS, la lista degli obiettivi attualmente attivi con la relativa scadenza e il progresso e la lista degli appuntamenti in arrivo.

Questa pagina deve essere facilmente scalabile con l'aggiunta di ulteriori informazioni.

4.5.2 Move

In questa pagina l'utente dovrà poter vedere tutti i suoi obiettivi legati al movimento. Dovranno quindi essere presenti tutti gli obiettivi attivi e passati, per ogni obiettivo attivo dovrà potersi vedere il dettaglio e lo stato attuale, mentre per i passati il cliente dovrà poterli filtrare per data e per completamento, così da capire dove può migliorare e in che periodo è stato più produttivo.

Dovrà inoltre essere presente una schermata in cui il cliente possa vedere tutti gli *achievement* relativi al movimento da lui sbloccati come se fosse una sorta di medagliere, quest'ultima opzione è stata pensata in ottica [gamification](#).

4.5.3 Food

Questa pagina non è oggetto del mio progetto di tirocinio.

Dovranno essere presenti tutti gli obiettivi attivi e passati legati allo stato di alimentazione del cliente.

4.5.4 Health

Questa pagina non è oggetto del mio progetto di tirocinio.

Dovranno essere presenti degli indicatori sullo stato di salute del cliente, come l'andamento della frequenza cardiaca nell'arco della giornata.

4.5.5 Agenda

In questa pagina l'utente dovrà avere un resoconto di tutti i suoi obiettivi e appuntamenti nel calendario. Sarà quindi presente un calendario in cui per ogni giorno l'utente dovrà vedere i suoi appuntamenti e i suoi obiettivi.

4.5.6 Chat

Questa pagina non è oggetto del mio progetto di tirocinio a livello implementativo, mi sono occupato principalmente della progettazione. In questa pagina l'utente può comunicare con il suo coach.

Sono stati analizzati diversi servizi ed [SDK](#) per l'implementazione di un sistema di messaggistica, che in futuro dovrà anche gestire delle video chiamate.

Tra tutti i servizi analizzati si è pensato di utilizzare Google Firebase, in quanto rende possibile anche l'utilizzo delle notifiche push che possono tornare utili per altri scopi. Qui il cliente avrà sempre aperta la chat con il suo coach, al quale potrà inviare messaggi testuali, foto, video o messaggi vocali. Per tutelare cliente e coach non si utilizzerà in alcun modo il numero di cellulare, la chat verrà gestite interamente da Google Firebase e dal nostro backend.

Inoltre per evitare che il coach si trovi troppi messaggi da tutti i suoi clienti, ogni cliente avrà una sorta di piano di abbonamento. Con l'abbonamento base gratuito avrà a disposizione un numero limitato di messaggi e minuti di chiamate al mese, mentre acquistando un abbonamento premium potrà aumentare il numero di messaggi e di minuti di chiamate.

4.6 Applicazione coach

La progettazione dell'applicazione coach segue la progettazione dell'applicazione client. Infatti viste le somiglianze e le funzionalità in comune tra le due applicazioni, si è deciso di utilizzare la stessa [codebase](#), e di sfruttare i flavors per compilare due diverse applicazioni a partire dallo stesso sorgente.

I flavors permettono di definire configurazioni di build separate che, tramite parametri e variabili d'ambiente, ci consentono di costruire l'applicazione distinguendo quali moduli fanno parte dell'applicazione coach e quali client. In questo modo possiamo inoltre selezionare icone diverse, indicare URL diversi per le API coach e client e generare *appid* differenti per poter pubblicare negli store due applicativi differenti.

Capitolo 5

Sviluppo

Il capitolo descrive le attività svolte e le principali difficoltà incontrate durante lo sviluppo delle applicazioni. Per prima verrà trattata l'applicazione cliente, in seguito l'applicazione coach e saranno disponibili gli screenshot dei risultati finali ottenuti.

La codifica è avvenuta per moduli: prima di passare al modulo successivo il risultato è stato visionato dall'azienda e dal tutor.

Saranno riportati solamente gli esempi di codice ritenuti più significativi.

5.1 Onboarding e primo login

La prima cosa su cui si è lavorato è stato il sistema di login e di recupero delle informazioni del cliente.

Una volta ricevuto dal mio tutor il *provider* che si occupa del login tramite *Auth0* sono stati sviluppati diversi widget per gestire il routing dell'applicazione in base al login avvenuto con successo o meno.

Inoltre è stato definito un modello per memorizzare i dati dell'utente ricevuto tramite chiamata API.

5.1.1 Widget di routing iniziale

Per gestire il routing iniziale dell'applicazione si è deciso di sviluppare multipli widget *statefull* in ascolto sullo stato di diverse richieste API, così facendo, al cambiamento di stato della richiesta API il widget viene ricostruito e indirizza l'applicazione ad un widget successivo in ascolto su di un'altra chiamata API. Questa opzione è stata scelta in quanto all'avvio dell'applicazione vengono eseguite diverse chiamate API e in questo modo è possibile osservare i campi *Loading* ed *Error*, come spiegato a sezione 4.3.3, in modo da avere un controllo sul flusso.

Di seguito elenco la lista di widget con i diversi instradamenti.

- **MyApp:** prova l'autenticazione e finché il provider di autenticazione resta nello stato di *Loading* mostra una *SplashScreen*, successivamente indirizza verso il widget **CheckLogin**;
- **CheckLogin:** controlla se l'accesso è avvenuto con successo o meno. Se si, in base al tipo di applicazione (client o coach) indirizza al widget **LoadUser** o **LoadCoach**, se l'accesso è fallito indirizza alla **LoginScreen**;

- **LoginScreen:** widget per riprovare l’accesso inserendo username e password;
- **LoadUser:** richiede al backend le informazioni dell’utente che ha effettuato l’accesso e mostra una schermata di caricamento mentre la chiamata è in stato di *Loading*, successivamente indirizza al widget **CheckBoolUser**;
- **CheckBoolUser:** controlla se è il primo accesso dell’utente o meno, se lo è indirizza ad una schermata con l’introduzione all’applicazione, altrimenti riporta alla schermata di *Homepage*.
- **LoadCoach:** richiede al backend le informazioni del coach che ha effettuato l’accesso e mostra una schermata di caricamento mentre la chiamata è in stato di *Loading*, successivamente indirizza al widget **CheckBoolCoach**;
- **CheckBoolUser:** controlla se ci sono stati errori nel recupero delle informazioni, se ci sono stati restituisce un widget di errore, altrimenti riporta alla schermata di *Homepage*.

```

1 class CheckLogin extends HookWidget {
2   @override
3   Widget build(BuildContext context) {
4     bool isLoggedIn = useProvider(authIsLoggedIn);
5     bool error = useProvider(authError);
6     String errorMsg = useProvider(authErrorMessage);
7     if (error) return GenericErrorPage(errorMsg: errorMsg);
8     if (!isLoggedIn) {
9       return LoginScreen();
10    } else {
11      switch (envType) {
12        case "client":
13          return LoadUser();
14        case "coach":
15          return LoadCoach();
16        default:
17          return GenericErrorPage(errorMsg: "non sei ne coach ne client
18        ");
19      }
20    }
21 }
```

Listing 5.1: Widget CheckLogin

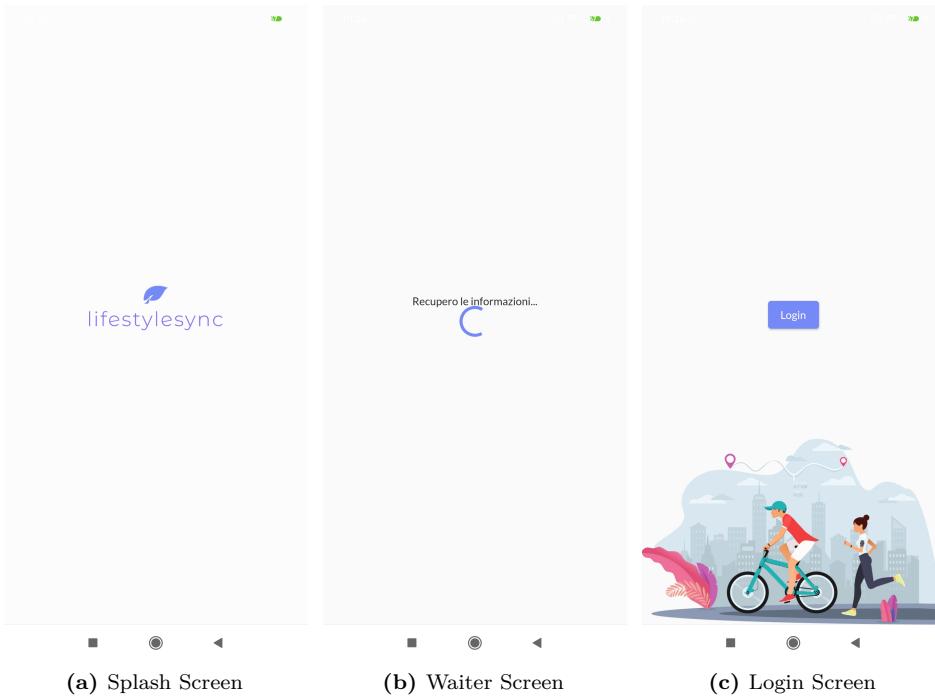


Figura 5.1: Prime schermate dell'applicazione

A supporto di questi widget sono stati sviluppati tre widget fondamentali:

- GenericErrorPage: accetta come parametro un messaggio di errore e presenta all’utente una generica pagina di errore con il messaggio passato come parametro;
- Waiter: accetta come parametro un messaggio e presenta all’utente una generica pagina di caricamento con il messaggio passato come parametro;
- DelayedWidget: accetta come parametro un numero di secondi ed una condizione da soddisfare. Alla costruzione del widget si avvia un timer della durata dei secondi passati come parametro. Mentre il timer è attivo o la condizione non è soddisfatta si presenta un widget di caricamento, solitamente **Waiter**, altrimenti viene mostrato all’utente un secondo widget passato anch’esso come parametro. Questo widget è fondamentale per evitare di avere caricamenti talmente brevi da non far vedere all’utente che l’applicazione sta recuperando dei dati.

```

1 class DelayedWidget extends HookWidget {
2   final Widget child;
3   final int timer;
4   final bool condition;
5   DelayedWidget({
6     Key? key,
7     required this.child,
8     required this.timer,
9     required this.condition,
10    }) : super(key: key);

```

```

11  @override
12  Widget build(BuildContext context) {
13    final timerCondition = useState(false);
14    useEffect(() {
15      Future.delayed(Duration(seconds: timer))
16        .then((value) => timerCondition.value = true);
17    }, []);
18    if (timerCondition.value && condition)
19      return child;
20    else
21      return Waiter(msg: "Caricamento");
22  }
23 }
```

Listing 5.2: Classe DelayedWidget

5.1.2 User model

La seguente classe è stata definita per contenere tutti i dati che l'applicazione necessita di conoscere rispetto all'utente che ha effettuato l'accesso. Nel caso dell'applicazione client, esiste una sola unica istanza di *User* e questa viene fornita a qualsiasi widget la necessiti attraverso un provider dedicato.

Tale classe si compone di diversi attributi primitivi e di due attributi che sono istanze di altre classi:

- **Info:** contiene le informazioni dell'utente tra cui anche un campo *coach*, istanza di una classe **Coach** la quale contiene i dati del proprio coach;
- **Metadata:** contiene i meta-dati dell'utente.

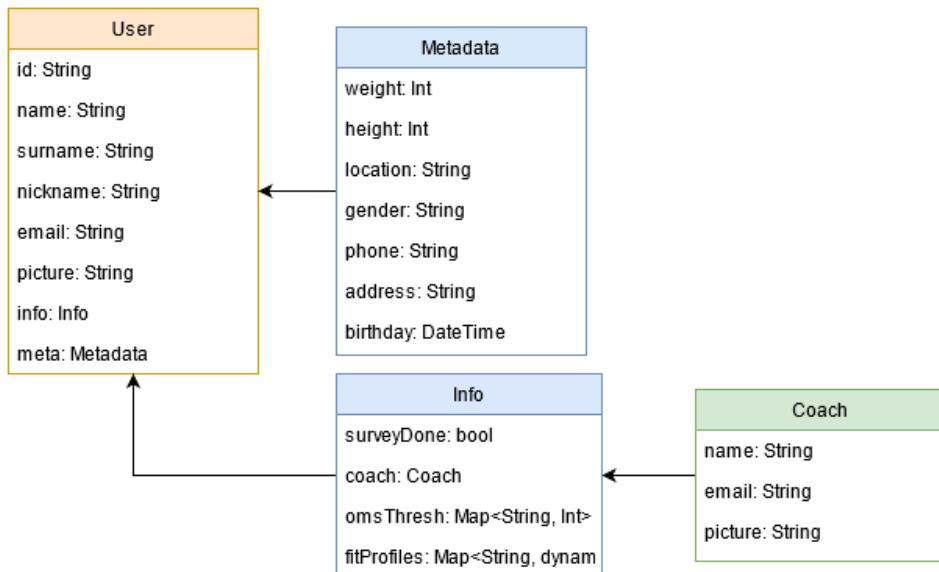


Figura 5.2: Modello di User

5.1.3 Introduzione all'app e questionario

Nel caso in cui sia il primo accesso di un utente, si è deciso di introdurlo all'utilizzo di LifestyleSync attraverso alcune schermate riassuntive delle principali funzionalità dell'applicazione.

Come ultima schermata inoltre viene presentato all'utente il coach che gli è stato assegnato, e successivamente gli viene presentato un questionario che è obbligato a compilare per continuare con l'utilizzo dell'applicazione.

Al termine del questionario le risposte vengono inviate al backend che si occupa anche di aggiornare le informazioni dell'utente segnando che ha completato il questionario, di conseguenza al prossimo login non dovrà ripeterlo e potrà utilizzare l'applicazione normalmente.

Per la realizzazione dei widget del questionario si è deciso di utilizzare una libreria anziché costruire tutte le diverse *form* manualmente. Tra tutte le librerie trovate e provate la migliore si è rivelata *SurveyKit*.

SurveyKit rende disponibile un buon numero di widget altamente personalizzabili per diversi tipi di domande: risposta singola, multipla, radio button, checkbox, e via dicendo. Inizialmente non si sono presentati problemi nell'utilizzo di tale libreria, ma una volta terminato lo sviluppo si è presentato un bug, il quale porta al *crash* dell'applicazione nel caso in cui l'utente annulli la compilazione del questionario.

Lo sviluppatore della libreria ha deciso di assegnare ad un tasto "Annulla" un metodo che va a rimuovere l'ultima pagina caricata nello stack del *Navigator* di Flutter (Widget standard per la gestione del routing in Flutter) senza però controllare che una volta rimossa l'ultima pagina ce ne siano delle altre da presentare.

Nel nostro caso il questionario non viene caricato nello stack e di conseguenza, alla rimozione dell'ultima pagina caricata viene rimossa l'unica pagina caricata nello stack e di conseguenza l'applicazione va in errore e termina.

Per risolvere tale problema si è reso necessario il [fork](#) della libreria e la rimozione del tasto "Annulla" dai diversi widget. Questo non risulta un problema in quanto l'utente è obbligato a compilare il questionario per proseguire nell'utilizzo dell'applicazione.

Generalmente viene sconsigliato il [fork](#) di una libreria in quanto si perde la possibilità di mantenerla aggiornata con le future *release*. Nel nostro caso però, avendo effettuato una minima modifica, ci rimane comunque possibile aggiornare la libreria alle future versioni ed eventualmente andare a ritoccare qualche riga di codice.

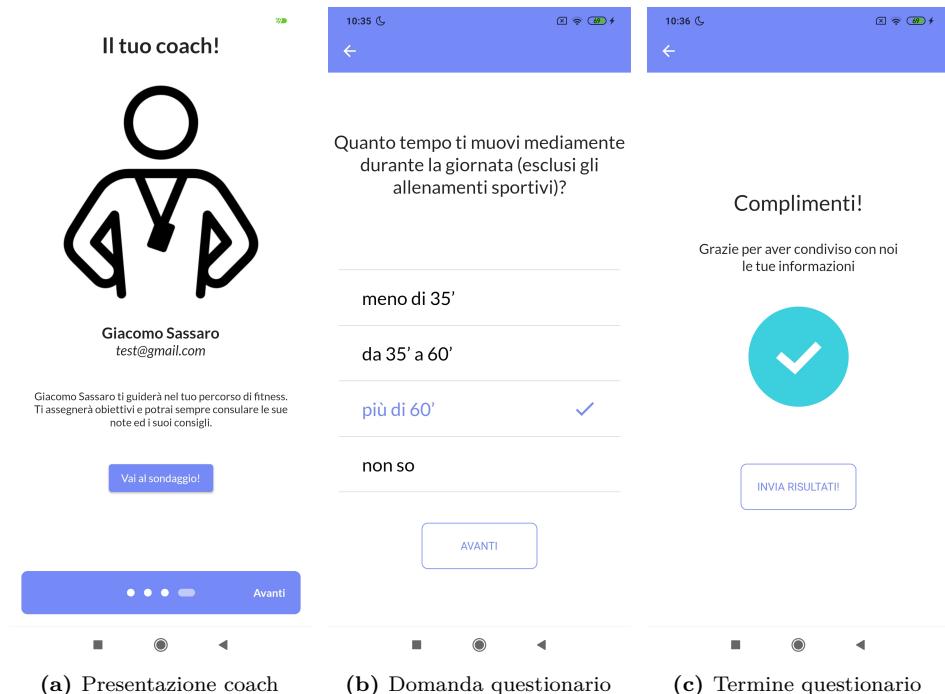


Figura 5.3: Alcune schermate onboarding

5.2 Connessione ai device

La connessione ai device è stata gestita per la maggior parte dal mio tutor in quanto aveva già utilizzato l'API per l'associazione del profilo LifestyleSync ai profili fitness nel sito web. Il mio compito è stato solamente quello di sviluppare l'interfaccia utente utilizzando i dati che mi venivano restituiti dal provider sviluppato dal mio tutor.

In fase di progettazione si pensava di connettere l'applicazione direttamente alle smartband dei diversi produttori (Google Fit, Fitbit, Garmin), in fase di sviluppo però ci si è resi conto che la connessione diretta al dispositivo fisico risultava eccessivamente complessa e si è deciso quindi di associare gli account delle diverse piattaforme di fitness all'account di LifestyleSync, così da prelevare le misurazioni delle smartband dai dati presenti negli account online e non direttamente dal dispositivo fisico.

È stata quindi sviluppata una sezione, nella schermata del profilo, attraverso la quale l'utente possa vedere a quali servizi è già connesso e può quindi disconnettersi o viceversa.

Nel caso l'utente provi la connessione a un nuovo servizio, premendo nell'apposito pulsante verrà aperta una pagina nel browser dello smartphone in cui è possibile effettuare l'accesso al servizio in questione e autorizzarne il prelievo dei dati. Nel caso in cui l'associazione vada a buon fine viene visualizzato un messaggio di successo e i dati iniziano ad essere salvati nel nostro backend, altrimenti viene visualizzato un messaggio di errore ed è necessario riprovare l'associazione.

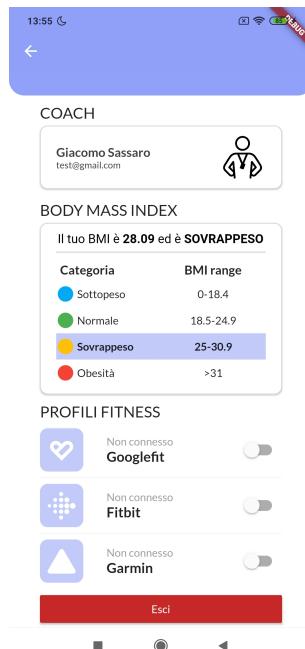


Figura 5.4: Schermata del profilo cliente con sezione relativa ai profili fitness

5.3 Dashboard utente

Una volta completato lo sviluppo delle funzionalità Onboarding si è progredito con lo sviluppo della dashboard utente, ossia delle schermate principali con cui l’utente interagisce durante l’uso quotidiano dell’applicazione.

5.3.1 Models

Nelle varie schermate di dashboard utente sono visualizzati i diversi dati relativi a Obiettivi, Appuntamenti, Risultati e Attività. Tutti questi dati vengono richiesti al backend al caricamento dell’applicazione o nelle schermate che ne fanno uso, questo in base alla scelta progettuale di dove salvare i dati ricevuti: nello store globale o nello store locale di un widget.

Per memorizzare tali dati si sono dunque rese necessarie diverse classi: *Goals*, *Appointment*, *Achievement*, *Duration* Siccome sia *Appointment* che *Goals* possiedono una data di inizio e una di fine, si è deciso di sviluppare una classe astratta *Event* da cui farle ereditare. Questa scelta è stata fatta anche per permettere una scalabilità all’applicazione in quanto in futuro saranno sviluppate altre classi che avranno data di inizio e fine e potranno quindi anche loro ereditare da *Event*.

Event Classe astratta rappresentante un evento generico.

- **id**: identificativo dell’evento;
- **start**: data di inizio;
- **end**: data di fine.

Goals Insieme di obiettivi, questo insieme può essere attivo o passato e può essere stato raggiunto o meno. Attualmente un *Goals* si compone di quattro *Goal*: passi, minuti di attività, minuti non programmati, calorie.

- **goals**: lista di *Goal* indicizzata (steps, durationA, durationN, calories);
- **achieved**: obiettivo completato o meno;
- **active**: obiettivo attivo o meno;

Goal Un obiettivo da raggiungere che può essere: passi, minuti di attività, minuti non programmati, calorie.

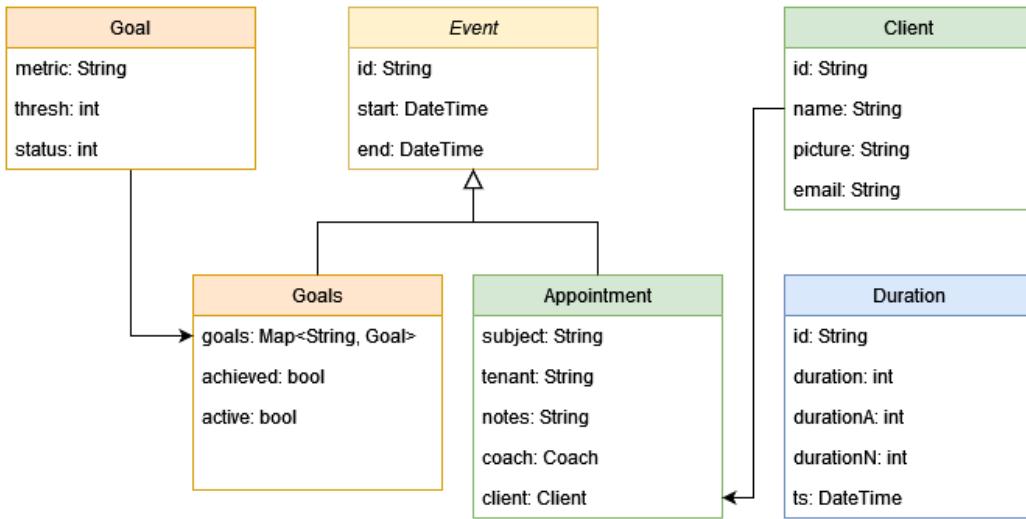
- **metric**: metrica di rilevazione;
- **thresh**: valore da raggiungere;
- **status**: valore attualmente raggiunto;

Appointment Rappresenta un appuntamento tra un *Coach* e un *Client*.

- **subject**: titolo dell'appuntamento;
- **tenant**: gruppo di clienti al quale appartiene l'appuntamento;
- **notes**: appunti lasciati dal coach;
- **coach**: coach che partecipa all'appuntamento;
- **client**: cliente che partecipa all'appuntamento.

Duration Minuti di movimento di una giornata.

- **id**: identificativo;
- **duration**: somma di durationA e durationN;
- **durationA**: minuti di attività giornalieri;
- **durationN**: minuti non programmati giornalieri;
- **ts**: data della rilevazione.

**Figura 5.5:** Modelli dei dati

5.3.2 Homepage

Una volta predisposti i diversi modelli si sono sviluppati i widget per la rappresentazione di tali dati nell'homepage. Si è deciso di dare un resoconto di quasi tutti i dati nell'homepage, quindi mostrare tramite una card i minuti di movimento (durations), una lista di card con gli obiettivi (goals) e una lista di card con gli appuntamenti (appointment).

Da questa schermata l'utente può approfondire tutti i diversi ambiti cliccando su degli appropriati widget. Cliccando sul proprio nome l'utente può vedere i dettagli del suo profilo, cliccando sul pulsante "Vedi tutti" di Obiettivi può visualizzare tutto il suo storico di obiettivi, ossia la sezione *Move*, inoltre cliccando su una qualsiasi card può vedere i dettagli dell'evento premuto sia nel caso di un obiettivo sia nel caso di un appuntamento.

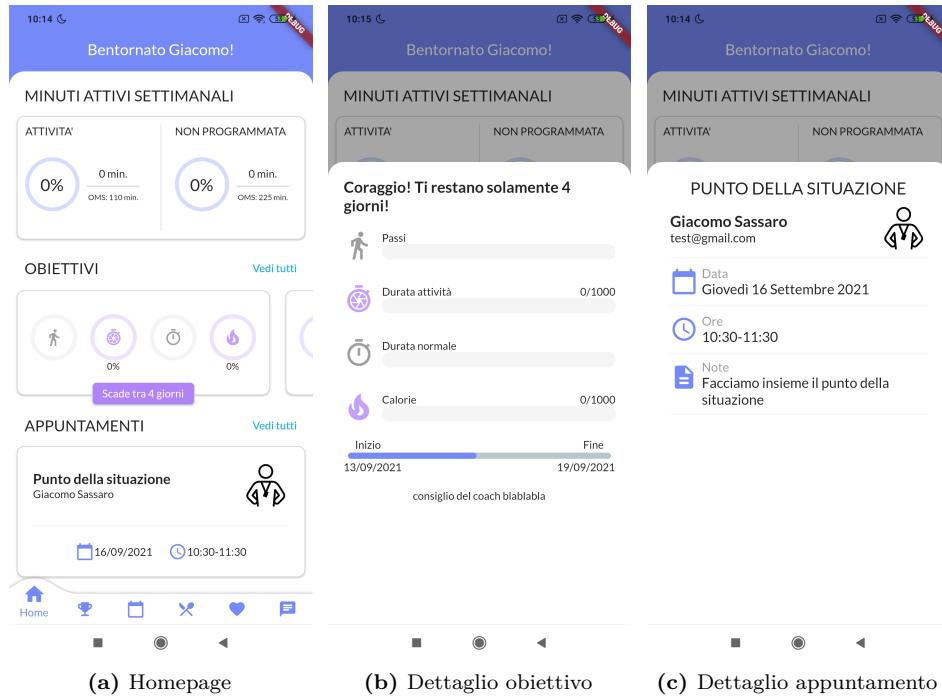
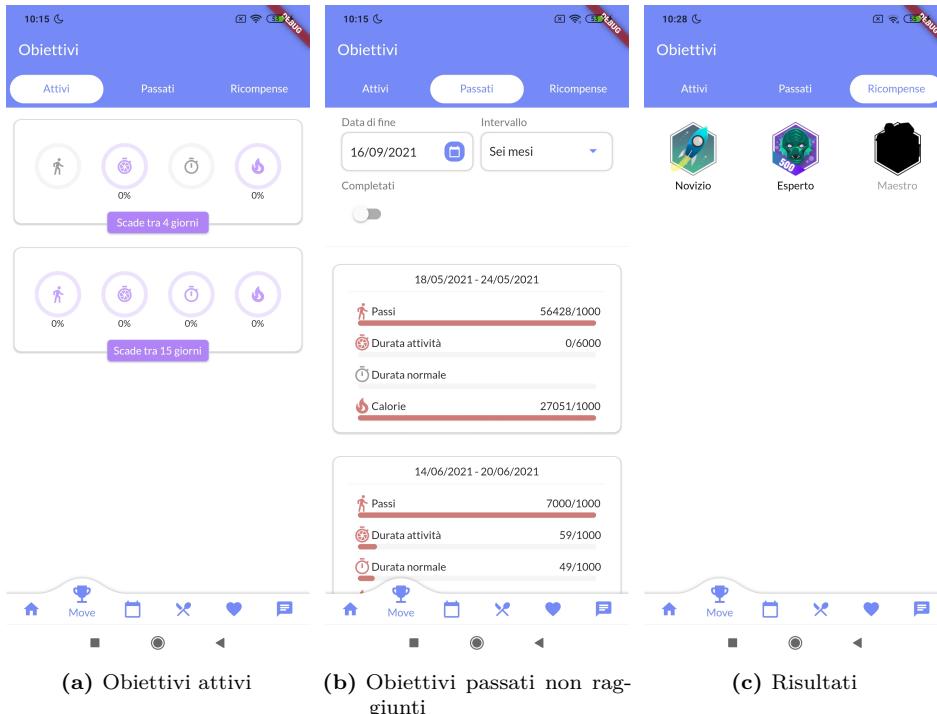


Figura 5.6: Homepage e dettagli eventi

5.3.3 Move

Questa schermata è raggiungibile dalla barra di navigazione o cliccando sul pulsante "Vedi tutti" degli obiettivi nella *Homepage*. Qui si è deciso di presentare all'utente la lista di tutti gli obiettivi attivi, la lista degli obiettivi passati e una sezione con i risultati raggiunti e da raggiungere. Quest'ultima sezione è stata aggiunta in ottica *gamification* in modo da spronare l'utente ad interagire con l'applicazione.

Per la sezione di obiettivi in corso si è potuto riutilizzare la card già sviluppata per l'*Homepage*, mentre per la sezione di obiettivi passati si è deciso di sviluppare una nuova card e di aggiungere una form per la ricerca tra gli obiettivi passati. L'utente può decidere la data di fine, l'intervallo e il completamento per filtrare gli obiettivi passati.

**Figura 5.7:** Schermata Move

5.3.4 Agenda

Questa schermata è raggiungibile dalla barra di navigazione. Tale schermata è leggermente diversa nelle due applicazioni: i clienti possono vedere nel calendario sia gli appuntamenti sia gli obiettivi, mentre i coach vedono solamente gli appuntamenti in quanto non possiedono obiettivi.

Cliccando su un giorno del calendario vengono mostrate all'utente le card con obiettivi e appuntamenti di quel giorno, le card sono le stesse mostrate nell'*Homepage*, si è potuto quindi riutilizzare buona parte del codice.

L'agenda è stata sviluppata in modo da essere altamente scalabile, è possibile infatti visualizzare gli eventi di tutto il mese o di settimana in settimana, inoltre nel caso in cui in futuro saranno aggiunti altri tipi di eventi sarà facile aggiungerli tra quelli visualizzati nel calendario.

La scalabilità è stata raggiunta utilizzando come oggetti da visualizzare nel calendario una lista di *Event* e facendo un cast a runtime al tipo corretto di oggetto per costruire la card corretta.

```

1 List<Widget> buildCards(List<Event> selectedElements, BuildContext
2   context) {
3   List<Widget> children = [];
4   children.addAll(buildGoalsCards(
5     selectedElements
6       .where((element) => element is Goals)
6       .map((e) => e as Goals))

```

```

7     .toList(),
8     context));
9     children.addAll(buildAppointmentCards(
10       selectedElements
11       .where((element) => element is Appointment)
12       .map((e) => e as Appointment)
13       .toList(),
14       context));
15   return children;
16 }

```

Listing 5.3: Costruzione delle card con individuazione a runtime del tipo

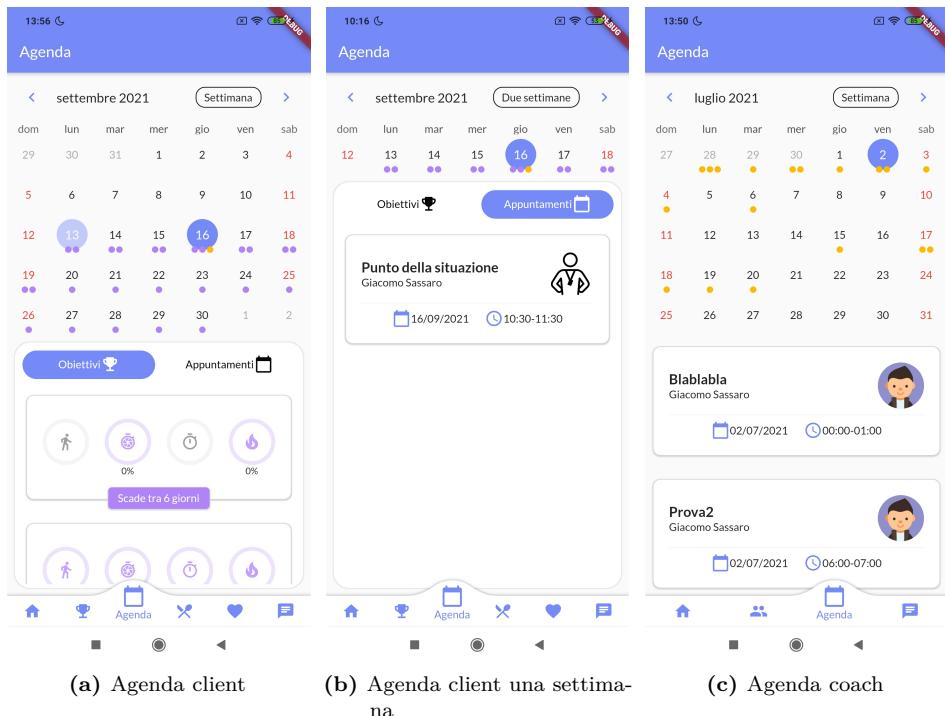


Figura 5.8: Homepage e dettagli eventi

5.3.5 Chat

Quest'ultima schermata è raggiungibile in entrambe le applicazioni dalla barra di navigazione.

Per i clienti, arrivati in questa schermata, è subito visibile la chat con il proprio coach ed è possibile inviare messaggi o media. Nel caso invece dell'applicazione coach, arrivati nella schermata della chat, è visibile una lista con i contatti di tutti i clienti e solo cliccando sopra una di questa voce si aprirà la chat con il cliente.

Purtroppo, viste le poche ore rimaste a disposizione per lo sviluppo di tale sezione, al momento è solamente possibile inviare messaggi di testo, foto e video grazie ad una

libreria già sviluppata per Flutter che sfrutta *Google Firebase*. Inoltre non essendo ancora integrato Google Firebase con il nostro backend, gli utenti della chat sono simulati e non sono in alcun modo collegati agli account di LifestyleSync.

La libreria utilizzata per la costruzione della chat, nonostante inizialmente si fosse rivelata valida, manca di molte funzionalità. Infatti nella libreria è implementato solo il messaggio di tipo testuale e immagine, ho dovuto quindi fare un [fork](#) per implementare il messaggio video e in futuro sarà sviluppato anche il messaggio vocale.

5.4 Applicazione coach

L'applicazione coach è stata sviluppata quando lo sviluppo dell'applicazione client era già iniziato da alcune settimane, buona parte delle componenti e delle schermate è stato quindi possibile riutilizzarla apportando poche modifiche al codice. Si è deciso infatti di mantenere la stessa [codebase](#) per entrambe le applicazioni e sfruttare i flavors. La principale differenza di quest'app da quella dei clienti è che qui non sono presenti le schermate *Move*, *Food* ed *Health* in quanto un coach non possiede obiettivi propri. Nell'app coach è però presente una schermata in cui il coach può vedere tutti i suoi clienti con i relativi obiettivi a loro assegnati.

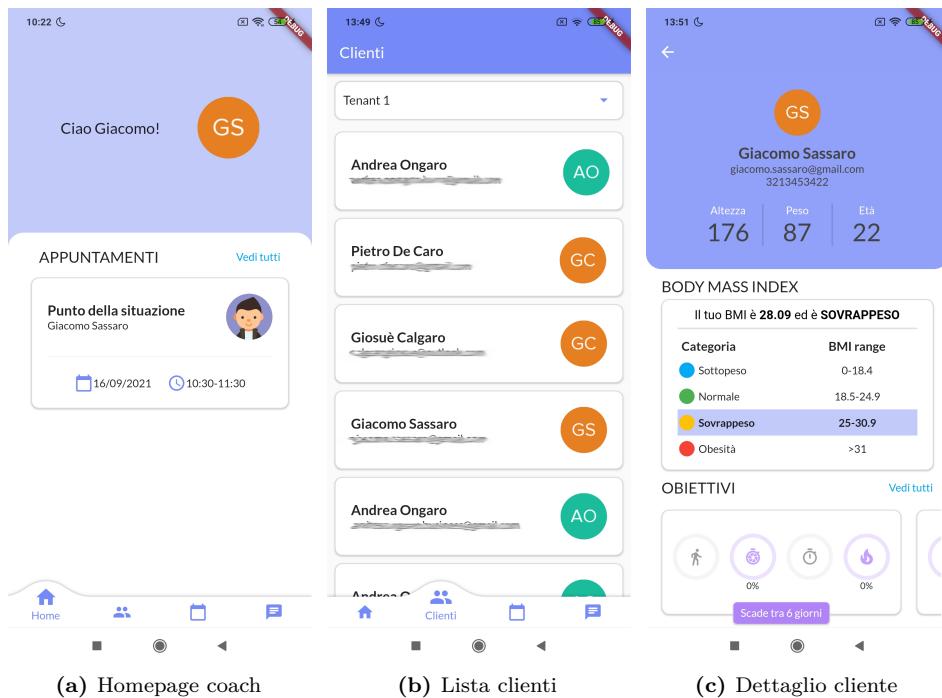


Figura 5.9: Schermate app coach

Capitolo 6

Conclusioni

In questo capitolo vengono analizzati i risultati ottenuti durante l'attività di stage e vengono esposte le conclusioni tratte.

6.1 Valutazione dei risultati ottenuti

Il prodotto finale ottenuto rispetta le aspettative dell'azienda. L'interfaccia grafica risulta sobria e *user-friendly*, inoltre grazie alla buona gestione dello stato dell'applicazione i rendering dei widget sono ben gestiti e non sono presenti rallentamenti nell'utilizzo dell'app.

L'app client e l'app coach rispettano tutti i requisiti funzionali obbligatori e desiderevoli delineati nell' Analisi dei Requisiti. Per quanto riguarda invece i requisiti di vincolo, per entrambe le app non è stato rispettato il requisito desiderabile **R2V3**: "L'applicazione deve funzionare completamente su dispositivi iOS". Questo è dovuto al poco tempo disponibile dedicato allo sviluppo per iOS, si è preferito infatti completare lo sviluppo e il testing per Android.

6.2 Sviluppi futuri

L'applicazione client una volta conclusa dovrà comporsi di tre diverse sezioni principali: Move, Food e Health.

Il mio stage ha coperto principalmente la sezione Move, predisponendo solamente delle schermate vuote per le altre due rimanenti.

I prossimi sviluppi saranno dunque la sezione di Food: sezione in cui il cliente avrà degli obiettivi alimentari da raggiungere e potrà vedere la sua cronistoria alimentare; e la sezione Health in cui il cliente potrà monitorare il suo stato di salute attraverso diversi grafici che andranno ad illustrare l'andamento del suo battito cardiaco nell'arco delle giornate e delle attività motorie.

Altra miglioria futura dovrà essere il *restyling* grafico di alcuni elementi dell'applicazione che per mancanza di tempo si è deciso di lasciare solamente abbozzati.

Allo stesso modo anche per l'applicazione coach dovranno essere sviluppati degli elementi per raffigurare gli obiettivi di alimentazione e lo stato di salute di un cliente, per questa applicazione però non si dovranno produrre delle intere sezioni ma solamente alcuni elementi riassuntivi contenenti tutti i dati alimentari e di salute del cliente.

Invece per entrambe le applicazioni una funzionalità da sviluppare che andrà a dare

grande valore al progetto è il sistema di messaggistica attualmente abbozzato. L'idea è quella di sviluppare un **bot** che risponda automaticamente alle domande che più frequentemente vengono fatte dai clienti, mentre la chat diretta tra coach e cliente viene aperta solamente se il **bot** non sa dare risposta ai dubbi del cliente.

Chiaramente il cliente non potrà abusare di questo sistema e avrà quindi un numero limitato di messaggi da poter inviare gratuitamente, al raggiungimento della soglia massima il cliente potrà sottoscrivere un abbonamento per aumentare il numero di messaggi inviabili.

Ultimo punto di estensione è la possibilità di effettuare delle video chiamate tra cliente e coach. Per quest'ultimo punto non è stata effettuata nessun tipo di progettazione ma l'idea è piaciuta all'azienda e in futuro verrà implementata.

6.3 Conoscenze acquisite

Durante il percorso di stage ho potuto approfondire le mie conoscenze di sviluppo mobile e di conoscere il framework Flutter. Prima dello stage avevo già avuto l'occasione di sviluppare per dispositivi mobile e di utilizzare il framework React per la costruzione di interfacce grafiche web. Sicuramente la mia esperienza mi ha aiutato nell'iniziare il progetto e il percorso di stage mi ha permesso di migliorare nell'ambito dello sviluppo frontend per dispositivi mobile.

In particolare partecipando attivamente alla fase di progettazione e di realizzazione del prodotto ho imparato come si può strutturare un progetto complesso e che durante la realizzazione è importante fare attenzione a come le proprie scelte incidano sulla user experience del prodotto finale.

Dal punto di vista dell'integrazione all'interno dell'azienda, ho potuto acquisire nozioni sull'organizzazione e sulle fasi del ciclo di vita di un software, oltre alla possibilità di capire come relazionarsi all'interno di un team in ambito lavorativo.

6.4 Utilizzo di Flutter

L'esperienza con Flutter è stata positiva, infatti in seguito a qualche settimana di apprendimento mi è stato possibile costruire un'applicazione complessa in maniera semplice ed intuitiva. Inizialmente ho dedicato molto tempo al capire come funzionasse la gestione dello stato e delle chiamate asincrone, questa è stata la parte sicuramente più complessa nello studio del framework.

Mentre per la parte di gestione dei widget e di costruzione dell'interfaccia utente Flutter è molto semplice, grazie anche alla mia esperienza con React sono riuscito fin da subito a costruire schermate in modo intuitivo.

6.5 Valutazione personale

Questo stage è stato una buona esperienza formativa, ho avuto la possibilità di crescere a livello lavorativo sia acquisendo nuove competenze ma anche imparando a lavorare in un team nel quale è sempre stato possibile dialogare e scambiare idee.

Durante questa esperienza mi sono trovato spesso davanti a compiti che inizialmente non sapevo come affrontare ma che alla fine ho sempre portato a termine, penso

dunque che le competenze fornite dalla laurea triennale siano una buona base dalla quale partire per studiare la maggior parte delle tecnologie usate in ambito lavorativo.

Sitografia

Siti web consultati

Auth0. URL: <https://auth0.com/docs/>.

Dart. URL: <https://dart.dev/>.

DataSoil S.r.l. URL: <https://datasoil.it/>.

Docker. URL: <https://www.docker.com/>.

Documentazione Flutter. URL: <https://flutter.dev/docs>.

Git. URL: <https://git-scm.com/>.

Github. URL: <https://github.com/>.

Jira. URL: <https://www.atlassian.com/it/software/jira>.

Manifesto Agile. URL: <http://agilemanifesto.org/iso/it/>.

Material Design. URL: <https://material.io/design/introduction>.

SurveyKit. URL: https://pub.dev/packages/survey_kit.

Visual Studio Code. URL: <https://code.visualstudio.com/>.