Politecnico di Torino

POLITECNICO DI TORINO

Corso di Laurea Magistrale in Computer Engineering

Tesi di Laurea Magistrale

Health Wellbeing: un'applicazione per il benessere

Relatore
Prof. Maurizio Morisio

Candidato
Cosimo Emanuele Desantis

Dicembre 2025

Sommario

L'uso di tecnologia indossabile come gli smartwatch ha recentemente reso possibile al grande pubblico di monitorare autonomamente e in maniera dettagliata il proprio benessere psicofisico.
Questo progetto mira a evolvere e migliorare l'applicazione "Health Wellbeing", che si propone come un metodo per tracciare la qualitá della vita dell'utente e per ricevere consigli personalizzati, analizzando i suoi dati suddivisi in quattro ambiti fondamentali: attività fisica, sonno, alimentazione e umore.
L'intervento ha incluso una riprogettazione completa dal punto di vista delle interfacce grafiche e dell'esperienza utente (UI/UX), ma anche dal punto di vista del codice, adoperando un'architettura MVVM.
In conclusione, il nuovo client risulta notevolmente migliorato, in particolare sotto l'aspetto UI/UX, dove l'intervento è stato piú incisivo. Inoltre, il valore percepito dell'applicazione è stato significativamente aumentato grazie all'integrazione di un sistema di consigli (Recommender) sviluppato esternamente, che ha migliorato di gran lunga la qualità delle valutazioni sullo stile di vita dell'utente.
Si trascina dal passato, invece, alcune delle precedenti limitazioni, prima fra tutte la dipendenza da servizi esterni per l'acquisizione dei dati biometrici.

Indice

Elenco delle figure

Introduzione

Il mercato globale del benessere è cresciuto del 12,8% nel 2017/2018, trasformandolo in un'industria da 4,2 trilioni di dollari [22]. Nel 2023, si stima invece che abbia raggiunto 6,3 trilioni di dollari [21], dimostrando che tuttora è un settore in forte crescita.
Un trend report del 2019 di globalwebindex afferma che nell'era digitale i consumatori cercano di sfruttare al massimo le informazioni sulla loro salute e benessere per migliorarli [22].
Per farlo tendono ad acquisire strumenti utili ad auto analizzare la loro situazione psicofisica e poter così gestire in maniera autonoma la loro salute.
Questi strumenti, nella maggior parte dei casi si rivelano essere tecnologia indossabile come lo smart watch. Lo smart watch è attualmente la tecnologia primaria in questo ambito in quanto si prevede che più della metà degli utenti adulti di tecnologia indossabile ne utilizzerà uno [22].
A tal proposito, uno studio proveniente dal College of Nursing (Yonsei University) ha dimostrato che i risultati, in termini di salute, per utenti non malati sono migliori per chi utilizza applicazioni mobile (e di conseguenza anche smartwatch) rispetto a chi non li utilizza [24].
Questo progetto, quindi, ha l'obiettivo di sviluppare un'applicazione mobile (disponibile per iOS e Android) per il benessere, che mira a migliorare lo stato psicofisico dell'utente attraverso il tracciamento e il monitoraggio di dati. Tra i dati analizzati vi sono dati biometrici, forniti da uno smartwatch, e dati inseriti manualmente dall'utente, come cibo consumato, umore, peso ecc.
L'applicazione fornirà all'utente gli strumenti per valutare autonomamente la propria salute psicofisica, presentando uno storico e delle valutazioni dei dati personali.
Per la realizzazione del prototipo dell'applicazione, ho optato per una combinazione di tecnologie: il framework Flutter, che permette di realizzare interfacce grafiche fluide multi piattaforma, e Firebase, una piattaforma di Google che offre ´out of the box´ tutti i servizi essenziali, dall'autenticazione al database, accelerando così il processo di sviluppo [14, 8].

Obiettivi

L'obiettivo specifico di questo progetto consiste nel contribuire all'evoluzione e al miglioramento della precedente versione dell'applicazione, concentrandosi principalmente su tre aspetti distinti:

- **Front-end:** Riprogettazione dell'interfaccia utente (UI) per ottenere una user experience (UX) superiore, rendendo l'applicazione più moderna, accattivante, accessibile e intuitiva.
- **Back-end:** Riprogettazione dello schema del database per ottimizzare la sincronizzazione dei dati dell'utente. Questa modifica è stata necessaria per supportare la funzionalità dei consigli, ossia l'integrazione di un applicativo esterno, il cui sviluppo non è oggetto di questa tesi, in grado di fornire consigli personalizzati sulla base dei dati dell'utente.
- **Architettura generale:** Riscrittura del codice per garantire scalabilità e longevità, rendendolo più modulare e manutenibile.

Logica Applicazione

La logica dell'applicazione consiste nell'ottenere dati riguardanti l'utente e utilizzarli come input per fare delle valutazioni sullo stato di salute. La lista completa dei dati significativi dell'utente è:

- **Sonno**
  - Durata del sonno (specificamente numero di minuti per fase di sonno)
  - Orario di sonno (l'ora esatta in cui l'utente va a dormire)
- **Attività Fisica**
  - Numero di passi
  - Calorie bruciate e distanza percorsa nelle varie attivitá fisiche eseguite
  - Frequenza respiratoria
  - Forza della presa
  - Forza fisica (push-up, squat, abs)
  - Frequenza cardiaca
  - Saturazione dell'ossigeno (SpO2)
  - Frequenza cardiaca a riposo

  - Variabilità della Frequenza Cardiaca (HRV)

- **Alimentazione e Idratazione**
  - Log di cibi e bevande (es. acqua, caffè, alcool)
- **Benessere Psicofisico**
  - Risultati dei quiz sull'umore
- **Dati Generali**
  - Peso
  - Altezza
  - Età
  - Lunghezza del girovita
  - Sesso
  - Equilibrio del corpo (secondi in equilibrio sulla gamba destra, gamba sinistra o nella posizione tandem)

I dati possono essere classificati in:

- **dato di input:** dipendente in modo diretto dalle scelte di vita dell'utente (es. numero di passi, cibo ecc..)
- **dato di output:** non manipolabile direttamente dall'utente (es. frequenza cardiaca, calorie bruciate, frequenza respiratoria ecc..)

I dati possono essere caratterizzati ulteriormente in:

- **manuali:** forniti dall'utente inserendo manualmente i dati da schermate predisposte dall'applicazione
- **misurati:** misurazioni biometriche fornite da fonti esterne come gli smartwatch

Le valutazioni possono essere di due tipi:

- **interne all'applicazione:** generate utilizzando formule matematiche ben precise direttamente all'interno del client.
- **esterne all'applicazione:** vengono generati consigli personalizzati da un server esterno sfruttando i dati sincronizzati sul database.

Capitolo 1

Progettazione della UI/UX

Una delle priorità di questa tesi consiste nella riprogettazione della UI/UX della precedente versione per cercare di renderla paragonabile a quella delle soluzioni presenti sul mercato, tra cui Fitbit, Mi Fitness, Garmin Connect ecc. Per tale scopo mi sono affidato alle **dieci euristiche di usabilità di Jakob Nielsen**, che, nonostante risalgano al 1994, sono tutt'oggi tra quelle più utilizzate nel campo del design [28]. Seguendo contemporaneamente tutte le euristiche di Nielsen ho avuto modo di verificare l'usabilità dell'interfaccia che stavo progettando.
Gli aspetti migliorabili che ho riscontrato nella vecchia versione sono diversi:

Usabilità

- **Gerarchia:** l'utilizzo di un componente mobile nativo come la 'bottom navigation bar' è corretto, tuttavia le due sezioni "Home" e "Health Measures" contengono informazioni appartenenti alla stessa sfera, possono quindi essere raggruppati in un'unica sezione. Stessa cosa accade per "Profile" e "Personal Informations": in entrambi i casi si tratta di informazioni riguardanti l'utente. Questo viola l'euristica di usabilità di Nielsen **"Consistency and Standards"**.

- **Spostamento della selezione del periodo di visualizzazione dei dati** dalla schermata principale alle singole schermate delle metriche: per evitare di sovraccaricare le schermate principali delle metriche ho deciso, prendendo spunto dalle altre applicazioni sul mercato, di spostare la selezione della data/settimana/mese direttamente dentro la schermata della singola metrica, lasciando così, nelle schermate delle metriche, solo i dati relativi alla giornata odierna. Inoltre nella schermata "Home" non ci si aspetterebbe di cambiare

il periodo di visualizzazione dei dati dell'applicazione, violando anche qui l'euristica di Nielsen **"Consistency and Standards"**.

Accessibilità

- **Modificare la palette di colori** per renderla più accessibile creando una light mode e una dark mode. La versione precedente si basa su una scala di blu per le superfici, cosa abbastanza atipica per un'applicazione mobile, particolarmente se non viene proposta un'alternativa. Negare all'utente la scelta tra dark e light mode va contro i principi moderni di accessibilità.

- **Migliorare il contrasto** tra alcuni elementi e lo sfondo: ad esempio i pulsanti grigi (come le frecce per selezionare la data) oltre a non avere contrasto sufficiente con lo sfondo, non sembrano essere cliccabili, ma disabilitati.

A partire da queste valutazioni in mente mi sono approcciato al redesign dell'applicazione, partendo dall'elemento più distintivo, il logo.
Il logo è l'elemento più distintivo dell'identità del brand, è ciò che ti rende riconoscibile agli occhi del consumatore, fornendoti unicità nel mercato [4]. Ho scelto di creare una **versione stilizzata del quadrifoglio**, un elemento che si presta bene nell'ambito del benessere. Inoltre presenta quattro foglie, tante quanto sono i quattro ambiti chiave di salute e benessere integrati nel progetto: attività fisica, cibo, sonno e umore. La stilizzazione del logo è stata necessaria per renderlo coerente con lo stile *flat* adottato in seguito per il design dell'interfaccia grafica.

Per rappresentare i quattro ambiti del benessere (attività fisica, cibo, sonno e umore), sono state inserite icone specifiche che riflettono lo stile *flat* del logo, garantendo coerenza visiva in tutta l'applicazione. Queste icone sono visibili nella Figura 1.2.
Il **flat design** si basa su un approccio minimalista costituito da elementi bidimensionali, colori luminosi e accesi in stile pastello e l'abolizione (quasi totale) di tutti gli effetti, bagliori, ombre e sfumature che sono tipici di altri stili come il Neumorphism [18]. Per quanto riguarda la palette, ho utilizzato come **colore primario il verde** del quadrifoglio e come **colore secondario un blu violaceo**. La loro relazione cromatica si colloca tra il complementare e l'analogo, scelta che mi ha permesso di evitare impatti visivi eccessivi. Per le superfici, nella **Light Mode** mi sono affidato a un semplice grigio molto chiaro, con le relative card bianche. Nella **Dark Mode**, invece, ho optato per un nero profondo (per favorire

Figura 1.1. Logo stilizzato dell'applicazione, che riprende il concetto del quadrifoglio.

Figura 1.2. Logo con icone per rappresentare i quattro ambiti chiave di salute e benessere.

gli schermi AMOLED) con card grigie. Le palette cromatiche per entrambe le modalità sono illustrate nella Figura 1.3.

1.1 Software Utilizzato

Per la progettazione del prototipo ho utilizzato il software *Figma*. Figma è un ottimo supporto per creare prototipi di interfacce UI perché mette a disposizione diversi strumenti utili, tra cui i *Componenti*, che offrono la possibilità di riutilizzare gli stessi elementi di design mantenendo un'armonia visiva costante

Figura 1.3. Palette cromatiche per la Dark Mode e la Light Mode.

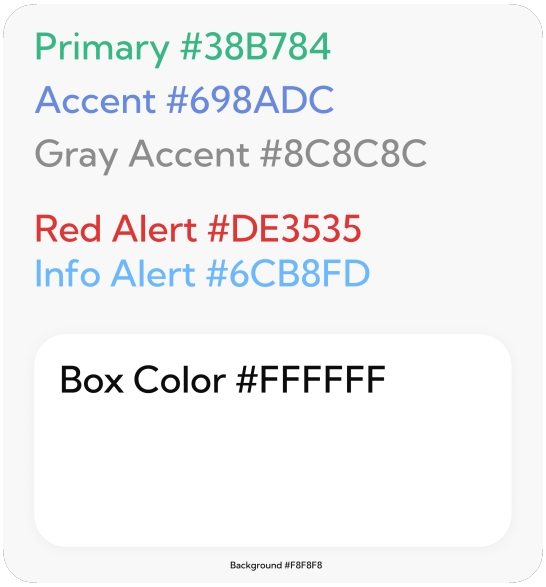


Figura 1.4. Light Mode Palette

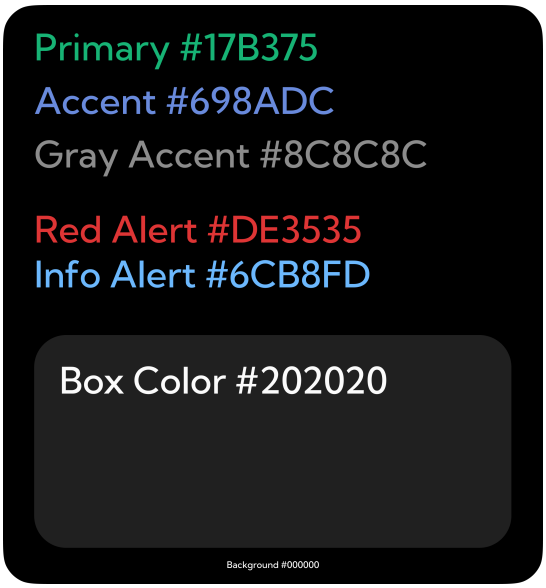


Figura 1.5. Dark Mode Palette

e velocizzando l'apporto di modifiche globali [7]. Inoltre, permette di definire il *Flow* dell'applicazione rendendo al contempo i pulsanti cliccabili e permettendo di ottenere fin da questa fase preliminare un prototipo semi-funzionante per testare l'usabilità dell'applicazione.

In definitiva, prototipare una UI/UX può richiedere diversi cicli di modifiche; utilizzare uno strumento come Figma permette di risparmiare tempo e risorse che, altrimenti, sarebbero andati persi prototipando e validando direttamente in fase di scrittura del codice.

Inoltre, Figma aiuta lo sviluppatore a tradurre il design in codice attraverso la possibilità di ispezionare gli elementi, estraendo tutte le informazioni necessarie. Lo stesso utilizzo dei *Componenti* in Figma suggerisce il bisogno di creare un componente, che nello specifico caso di Flutter coincide con un widget, anche nel codice.

1.2 Onboarding

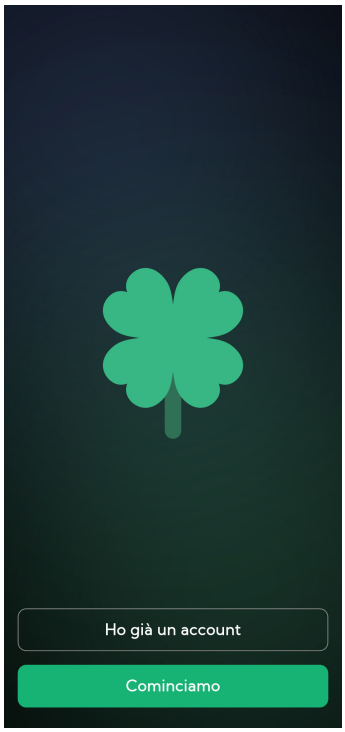
Le prime schermate che ho progettato sono state quelle relative alla fase di onboarding dell'utente. L'onboarding coincide con il primo punto di contatto

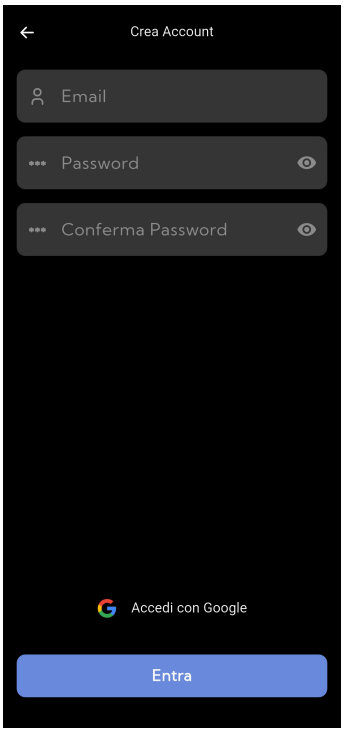
tra l'utente e il prodotto e, essendo noto che la maggior parte delle applicazioni vengono eliminate dopo il loro primo impiego (precisamente oltre l'80%), diventa una fase delicata da gestire. Essa ha il compito di introdurre l'utente all'applicazione e spiegare nel modo più chiaro possibile quello che fa. È quindi importante far percepire il valore dell'applicazione proprio in questa fase [29]. Il processo di onboarding è stato suddiviso in più schermate per rendere il flusso più semplice da seguire. Il flusso inizia con una pagina di benvenuto che consente all'utente di navigare verso le pagine di login o registrazione. Dopo la registrazione, l'utente viene reindirizzato a una pagina animata che spiega brevemente in cosa consistono i quattro ambiti chiave della salute e del benessere. Segue una serie di schermate per l'inserimento di informazioni riguardanti l'utente:

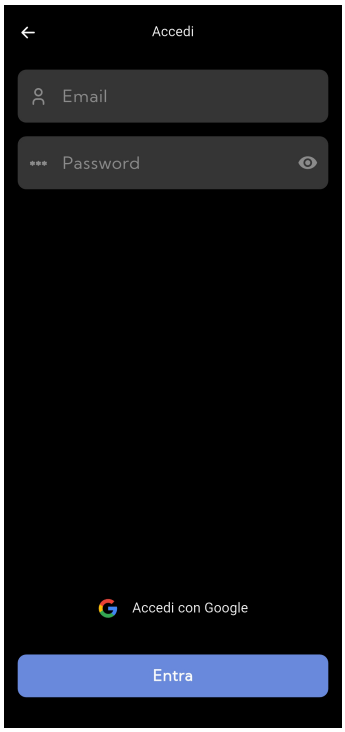
- Nickname
- Data di nascita
- Altezza
- Obiettivi giornalieri (passi, calorie bruciate, ore di sonno, orario di sonno)

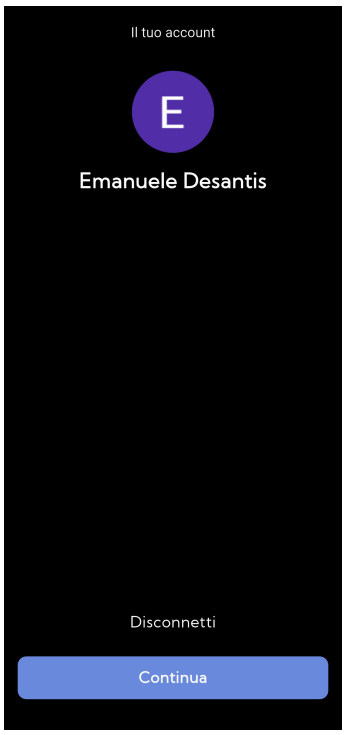
Dopo aver inserito queste informazioni, viene offerta la possibilità di fare il login a servizi esterni di benessere come Fitbit. Come capiremo meglio in futuro, questo passaggio ci permetterà di ottenere alcuni dati biometrici che altrimenti sarebbero mancanti, informando l'utente fin da subito della sua importanza. Una volta completato l'onboarding, l'utente ha definito le informazioni cruciali e può accedere all'applicazione vera e propria. Attraverso una pagina finale che indica il completamento dell'onboarding, viene reindirizzato alla schermata principale, denominata Home. La gerarchia delle schermate è gestita da una barra di navigazione inferiore che include le seguenti voci:

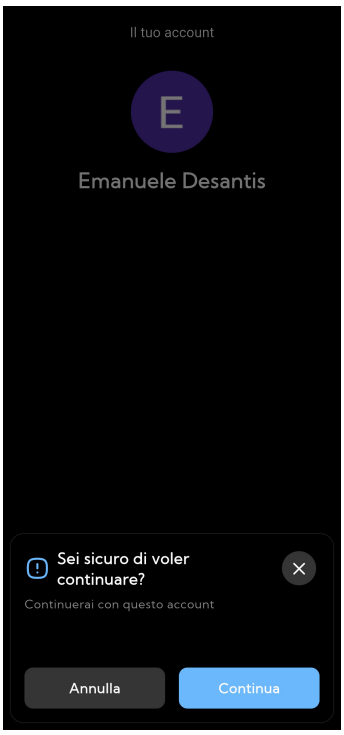
- Home
- Tips
- Lessons


Ho già un account
Cominciamo

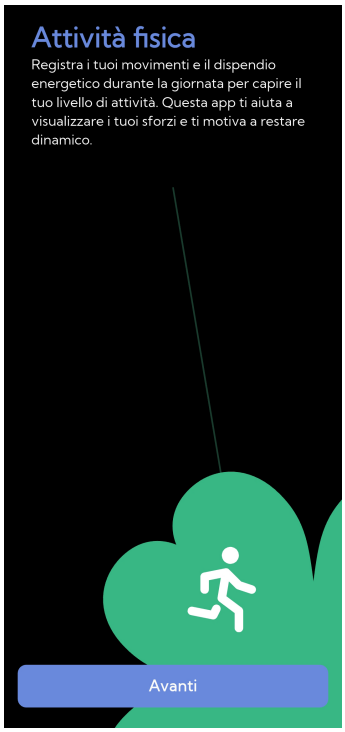

Crea Account
Email
Password
Conferma Password
Accedi con Google
Entra

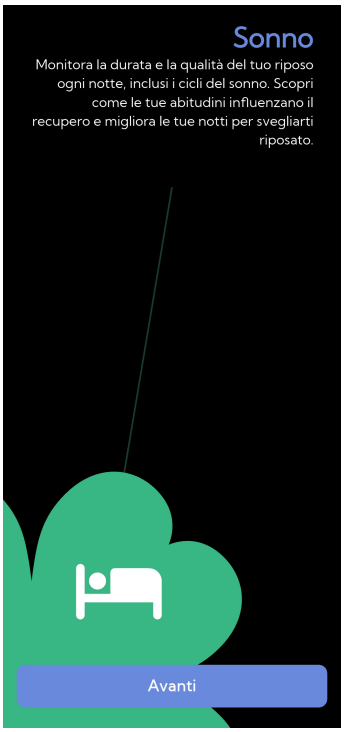

Accedi
Email
Password
Accedi con Google
Entra

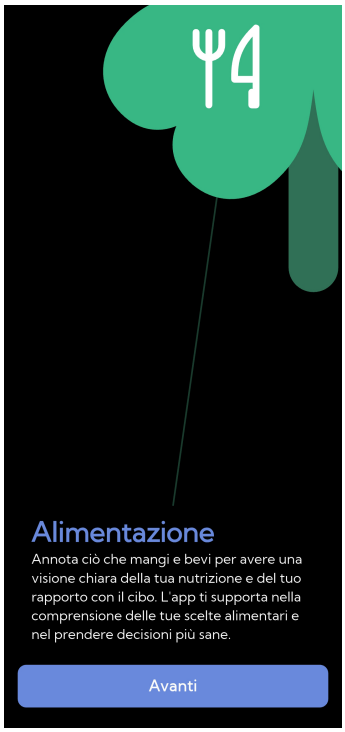

Il tuo account
E
Emanuele Desantis
Disconnetti
Continua

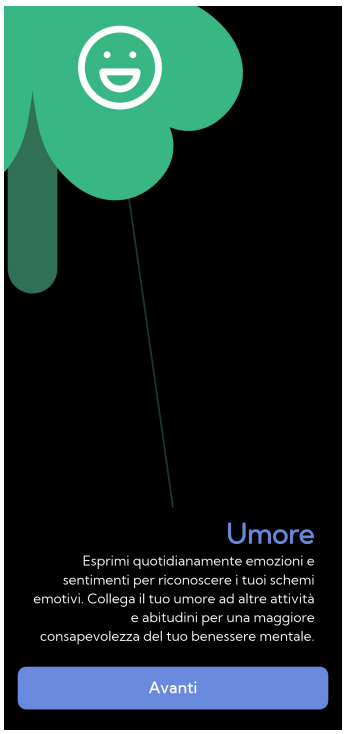

Il tuo account
E
Emanuele Desantis
Sei sicuro di voler continuare?
Continuerai con questo account
Annulla
Continua

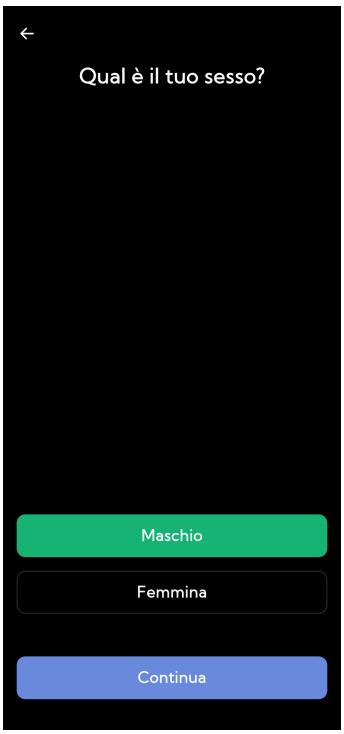

Inizia

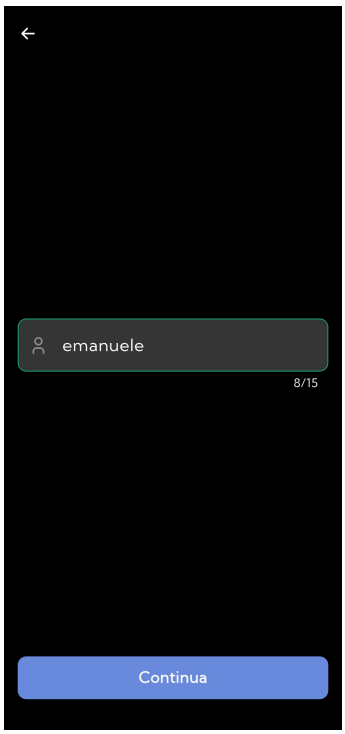

Attività fisica
Registra i tuoi movimenti e il dispendio energetico durante la giornata per capire il tuo livello di attività. Questa app ti aiuta a visualizzare i tuoi sforzi e ti motiva a restare dinamico.
Avanti

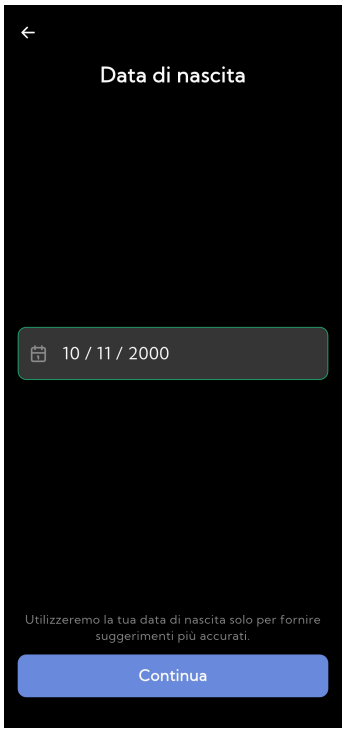

Sonno
Monitora la durata e la qualità del tuo riposo ogni notte, inclusi i cicli del sonno. Scopri come le tue abitudini influenzano il recupero e migliora le tue notti per svegliarti riposato.
Avanti

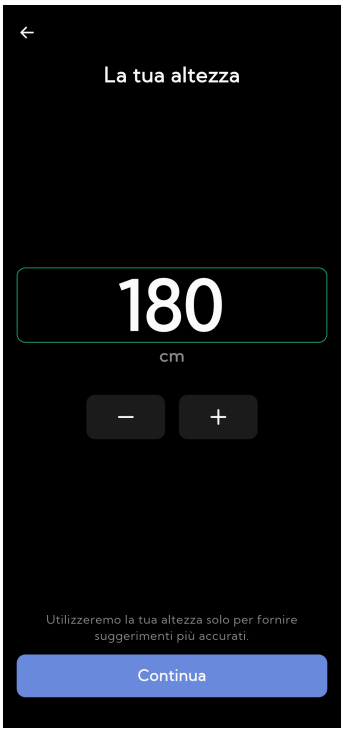

Alimentazione
Annota ciò che mangi e bevi per avere una visione chiara della tua nutrizione e del tuo rapporto con il cibo. L'app ti supporta nella comprensione delle tue scelte alimentari e nel prendere decisioni più sane.
Avanti

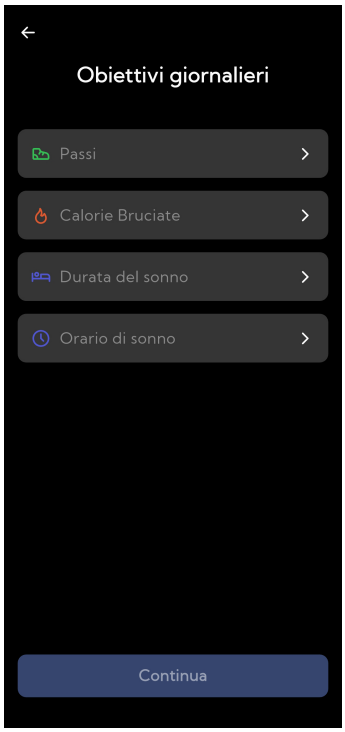

Umore
Esprimi quotidianamente emozioni e sentimenti per riconoscere i tuoi schemi emotivi. Collega il tuo umore ad altre attività e abitudini per una maggiore consapevolezza del tuo benessere mentale.
Avanti

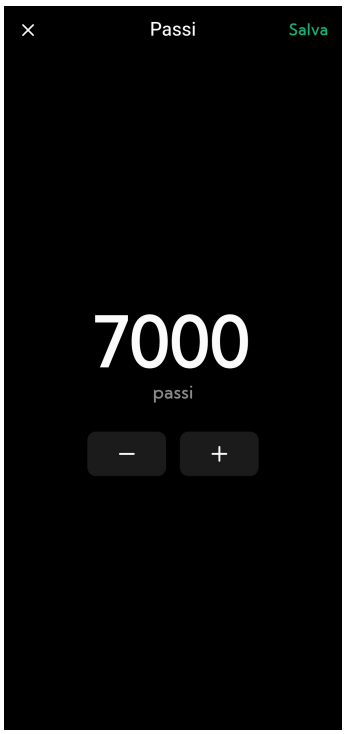

Qual è il tuo sesso?
Maschio
Femmina
Continua

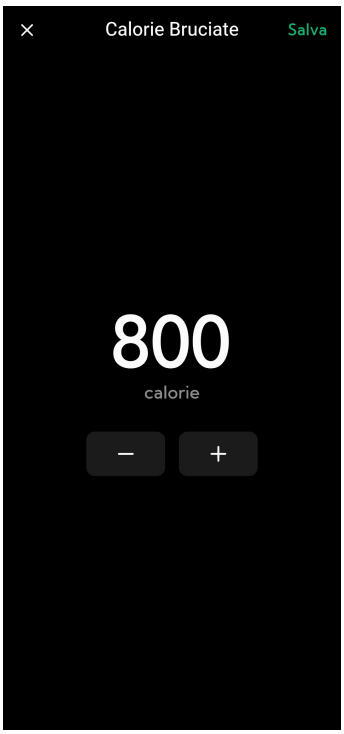

emanuele
8/15
Continua

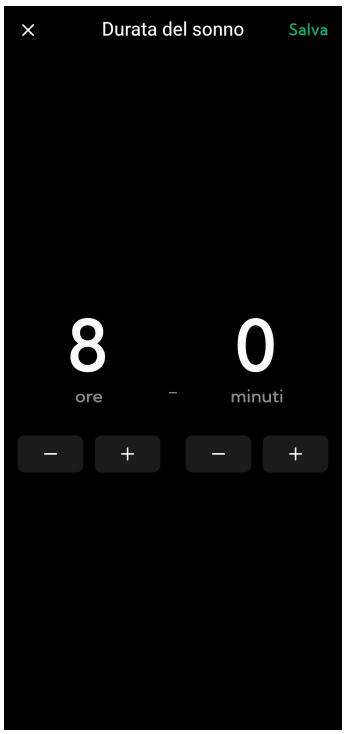

Data di nascita
10 / 11 / 2000
Utilizzeremo la tua data di nascita solo per fornire suggerimenti più accurati.
Continua

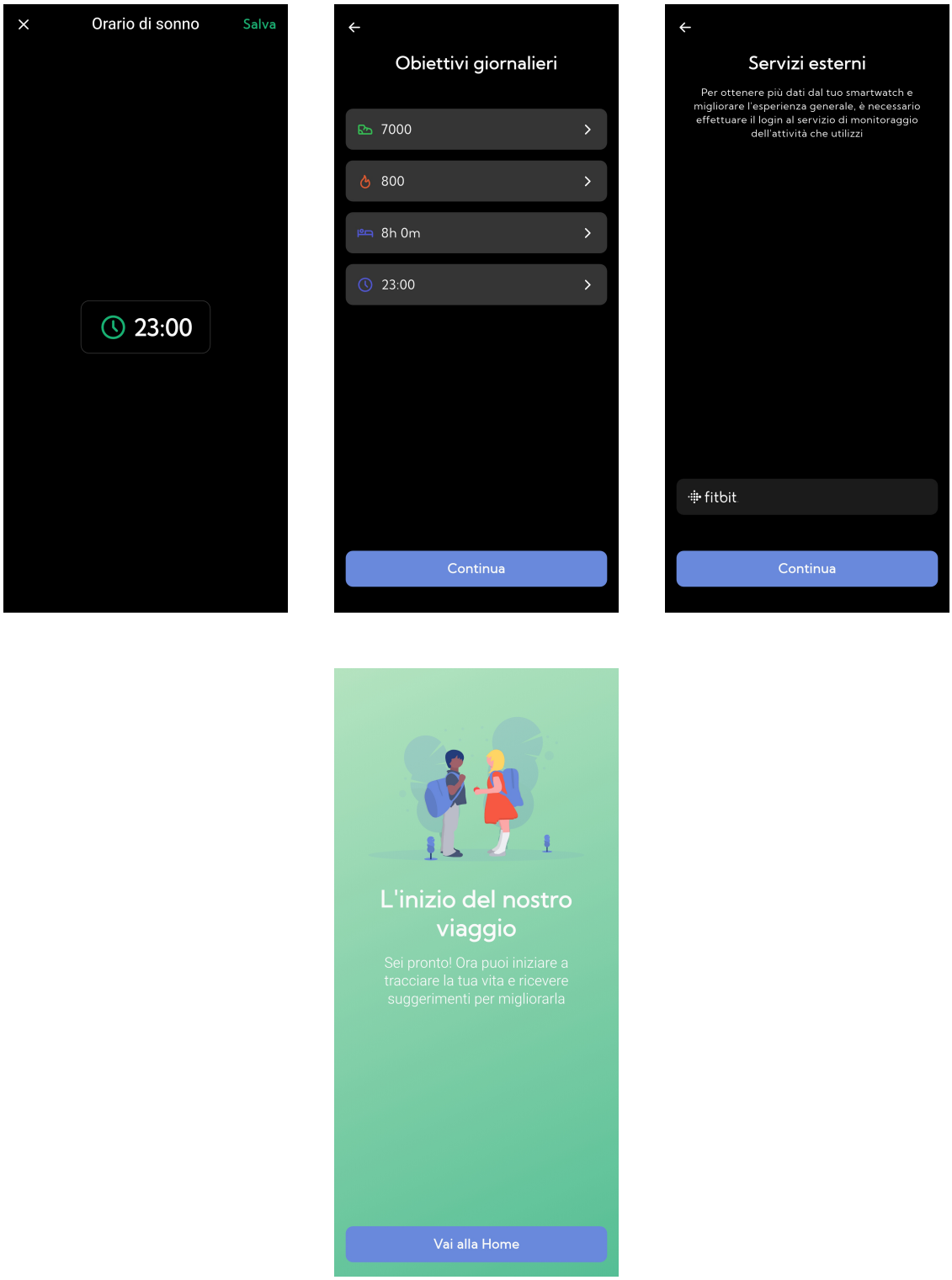

La tua altezza
180
cm
Utilizzeremo la tua altezza solo per fornire suggerimenti più accurati.
Continua


Obiettivi giornalieri
Passi
Calorie Bruciate
Durata del sonno
Orario di sonno
Continua


Passi
Salva
7000
passi


Calorie Bruciate
Salva
800
calorie


Durata del sonno
Salva
8
ore
0
minuti



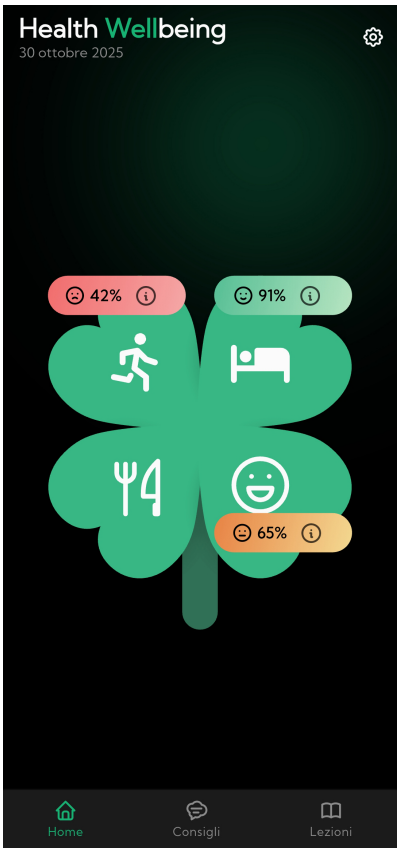
Screenshot 1.2.1: Visualizzazione completa della sequenza di Onboarding.

1.3 Home

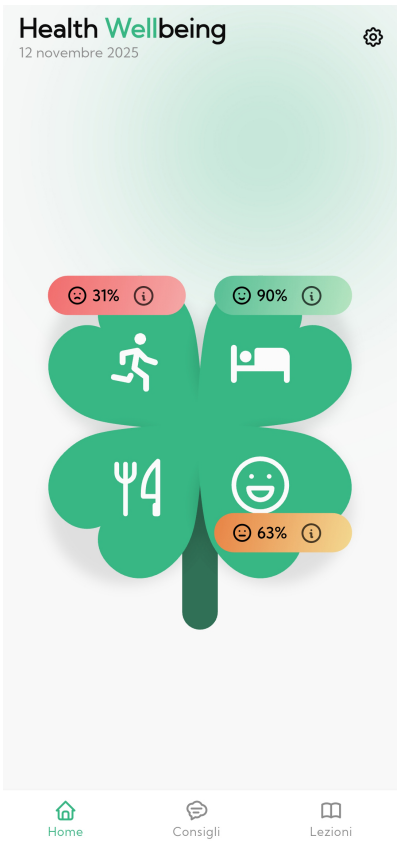
La schermata Home fornisce l'accesso alle funzionalità più importanti e include:

- **AppBar**: Sulla parte sinistra viene mostrato il nome dell'applicazione e la data odierna, mentre sulla destra un pulsante per accedere alle impostazioni.
- **Assessment Widget**: Un elemento grafico con le tipiche quattro foglie del logo cliccabili, ognuna delle quali mostra un'icona che rappresenta uno dei quattro ambiti del benessere. Cliccando su una specifica foglia è possibile accedere alla schermata delle metriche relative a quell'ambito.

Inoltre su ogni foglia vi è una "pillola" che mostra un punteggio in centesimi. Questo punteggio, come vedremo, è calcolato tramite formule matematiche.



Schermata Home in modalità scura (Dark Mode).



Schermata Home in modalità chiara (Light Mode).

Screenshot 1.3.1: Schermate Home in modalità scura e chiara.

1.4 Schermata specifica di un ambito

Schermata accessibile cliccando su una delle foglie della "Home". Composta da:

- **Appbar**: composta dal nome dello specifico ambito e da un pulsante per tornare alla schermata precedente.

- **Griglie di card**: Due griglie, distinte, una per gli input e una per gli output relativi all'ambito, contenenti una card per ciascuna misurazione o log appartenenti a quella categoria. Le card variano in base al tipo di dato:

  - **Misurazione biometrica multivalore**: La card mostra un grafico giornaliero suddiviso in quattro fasce orarie, con una barra verticale che rappresenta il valore del dato per ogni fascia.

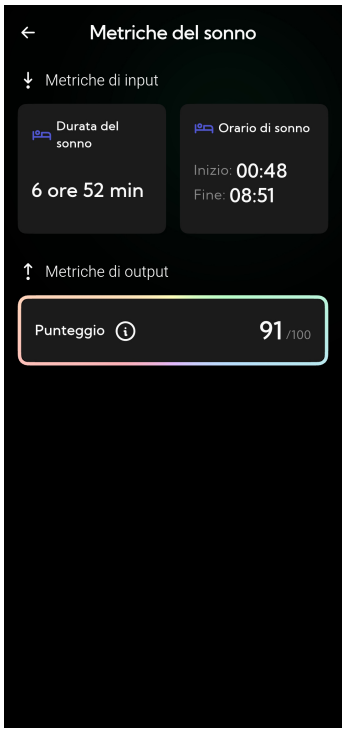
  - **Misurazione manuale**: Un testo che indica se l'utente ha inserito o meno il log per la giornata odierna.

  - **Misurazione biometrica singola**: La card mostra il valore odierno con la relativa unità di misura.

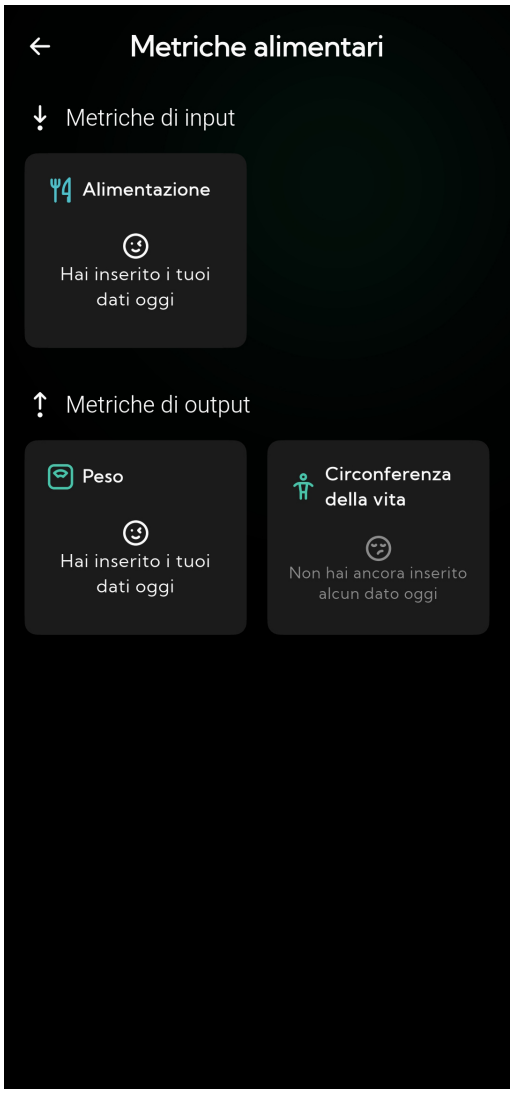
Per la progettazione di questa schermata mi sono ispirato alle applicazioni ´Fitbit´ e ´Mi Fitness´, che presentano entrambe un cruscotto nella parte alta della pagina principale e una lista di cards, per le misurazioni biometriche, subito sotto. A differenza delle due applicazioni prese come esempio, per evitare uno scroll verticale eccessivo e per strutturare in modo gerarchico i vari dati, ho optato per questa soluzione a due livelli (il passaggio dalla home a questa schermata) che sfrutta il logo per guidare graficamente l'utente.

Metriche attività fisica.

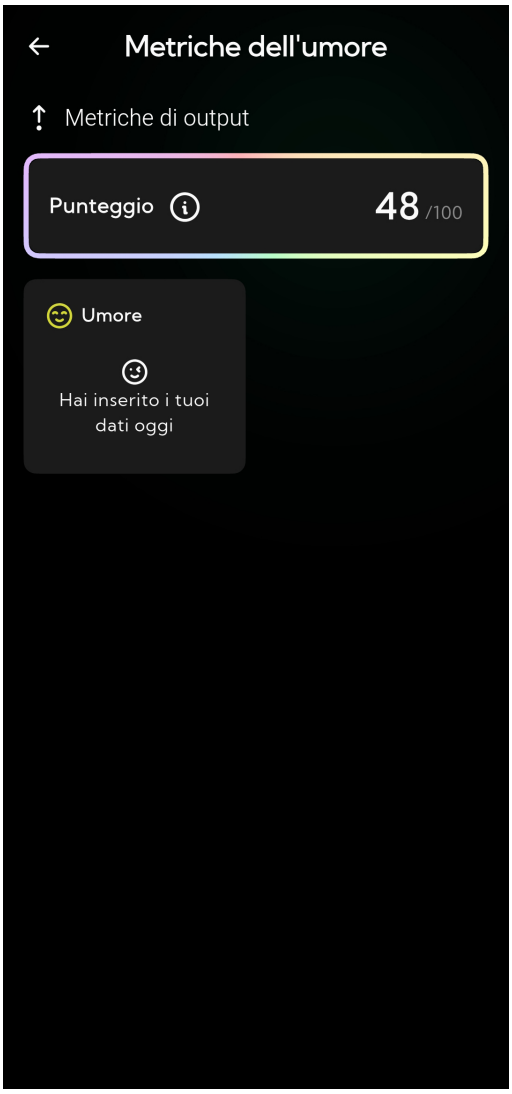
Continuazione metriche attività fisica.



Metriche del sonno.



Metriche alimentari.



Metriche dell'umore.

Screenshot 1.4.1: Metriche dei quattro ambiti

1.5 Schermata Metrica

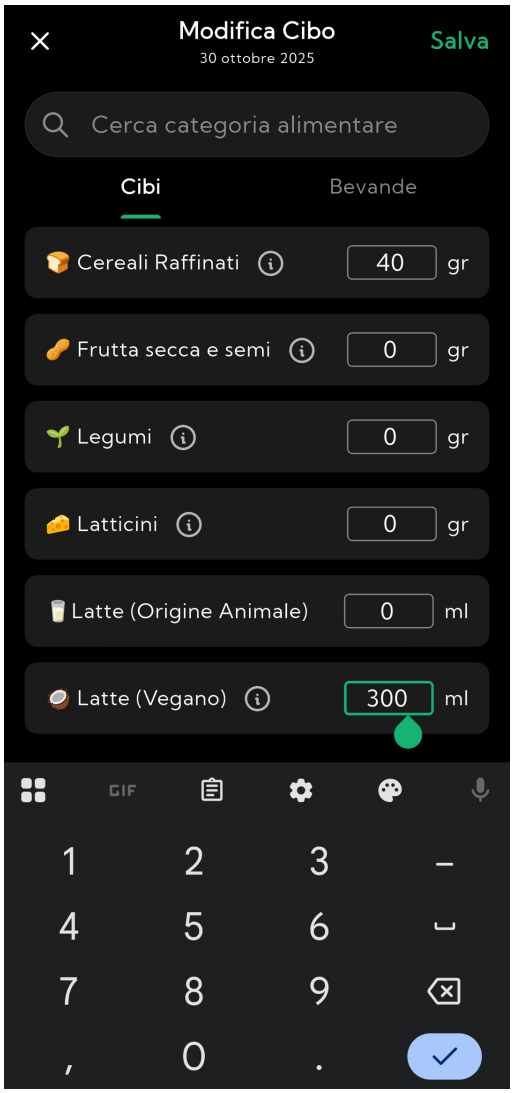
Cliccando su una card in particolare, si accede alla schermata specifica della metrica. Ogni pagina di questo tipo ha un pulsante nella barra superiore per tornare alla schermata precedente soddisfando così la terza euristica di Nielsen [28] (User control and freedom). Sotto la barra superiore si trova un cruscotto che consente di selezionare la granularità del periodo da visualizzare: giorno, settimana e mese. Per tutti e tre i periodi è possibile scorrere avanti e indietro nel tempo usando due frecce. Nel caso della granularità "giorno", è inoltre possibile cliccare sull'etichetta tra le due frecce per selezionare una data direttamente da un calendario a comparsa. Infine, abbiamo la dashboard che può presentare i dati in due diverse modalità:

- **Grafico a barre o lineare** (talvolta a scorrimento orizzontale): Utilizzato per misure multivalore.

- **Misurazione singola**: Visualizzata solo per alcune misure a valore singolo con granularità "giorno".

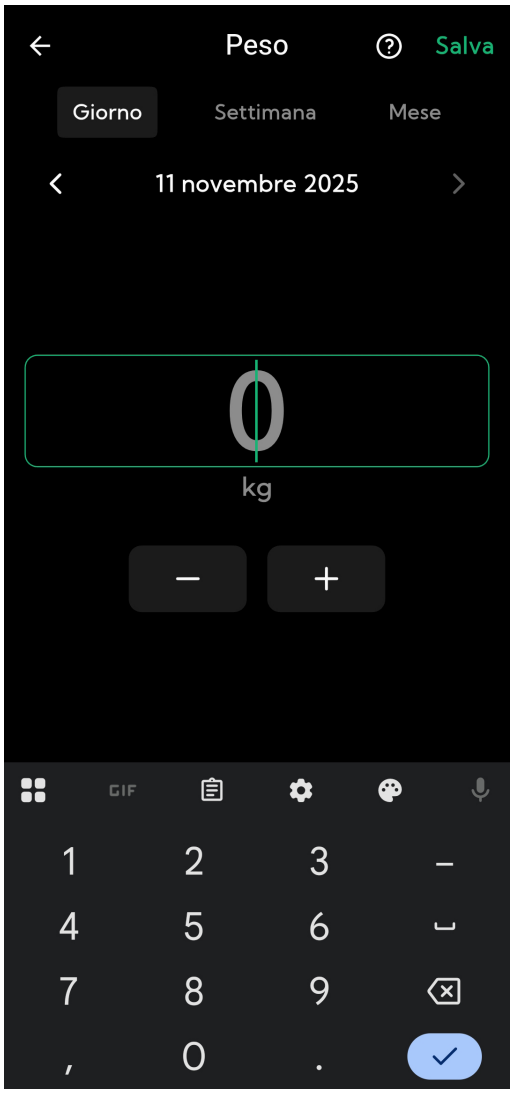
Il grafico a barre o lineare visualizza un valore per giorno quando si seleziona "settimana" e "mese", e un valore ogni tre ore per la granularità "giorno". Inoltre, se i valori della metrica possono essere aggregati, appena sopra la dashboard viene indicato il totale o la media dei valori per il periodo selezionato.

Nel caso di metriche manuali, è possibile inserire i valori tramite degli input field e salvare cliccando sul tasto "Salva" posto in alto a destra.

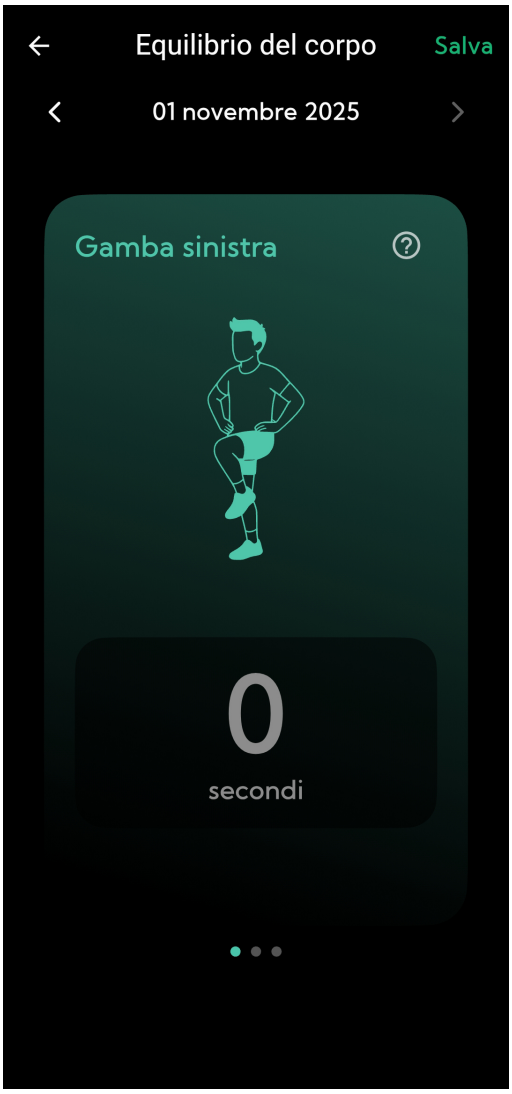


Registrazione del cibo e quantità.

Valutazione dell'umore su scala 1-10.



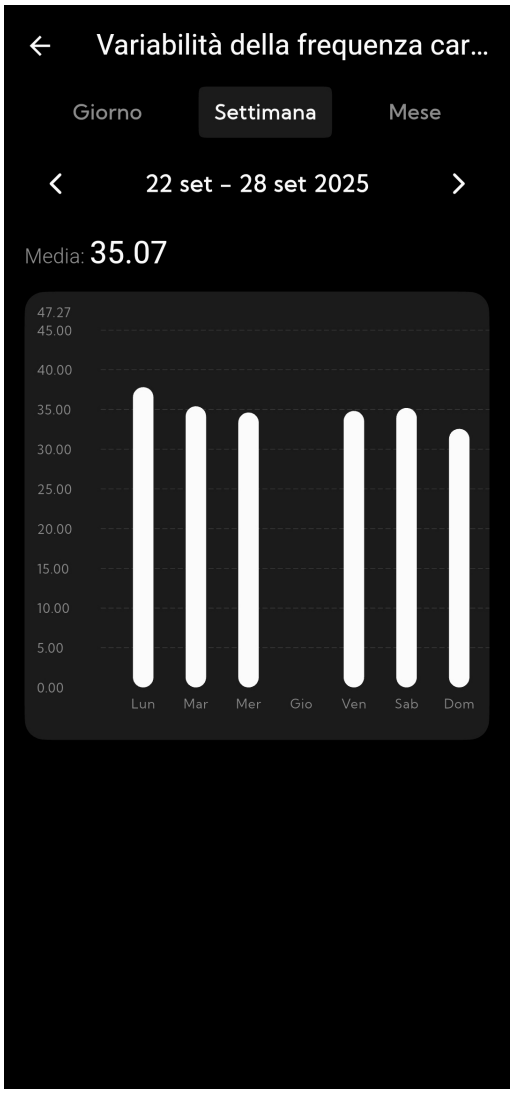
Inserimento del peso corporeo (kg) tramite tastierino numerico.



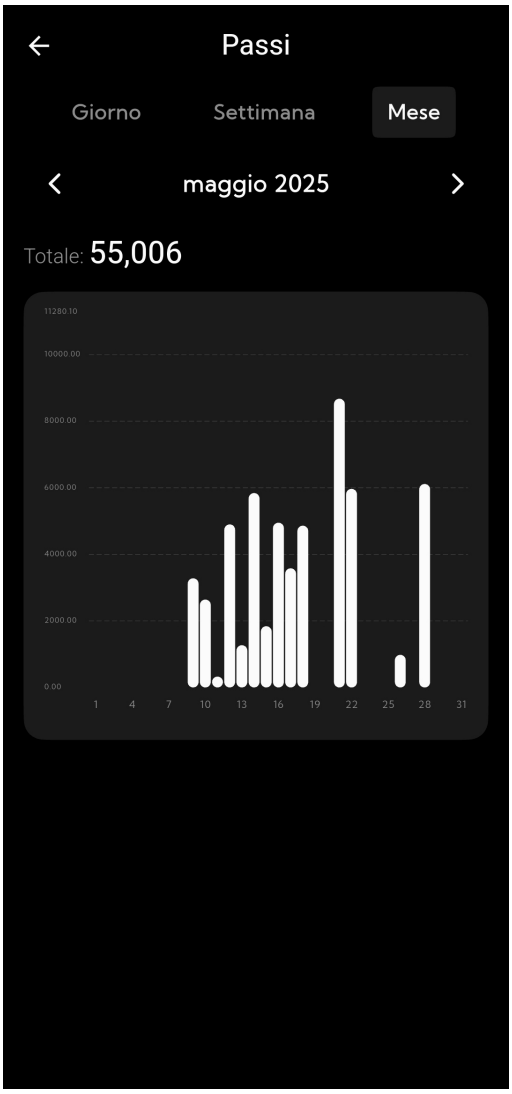
Schermata per l'inserimento dei dati relativi all'equilibrio corporeo.

Inserimento del numero di ripetizioni per uno degli esercizi di forza (Push up).

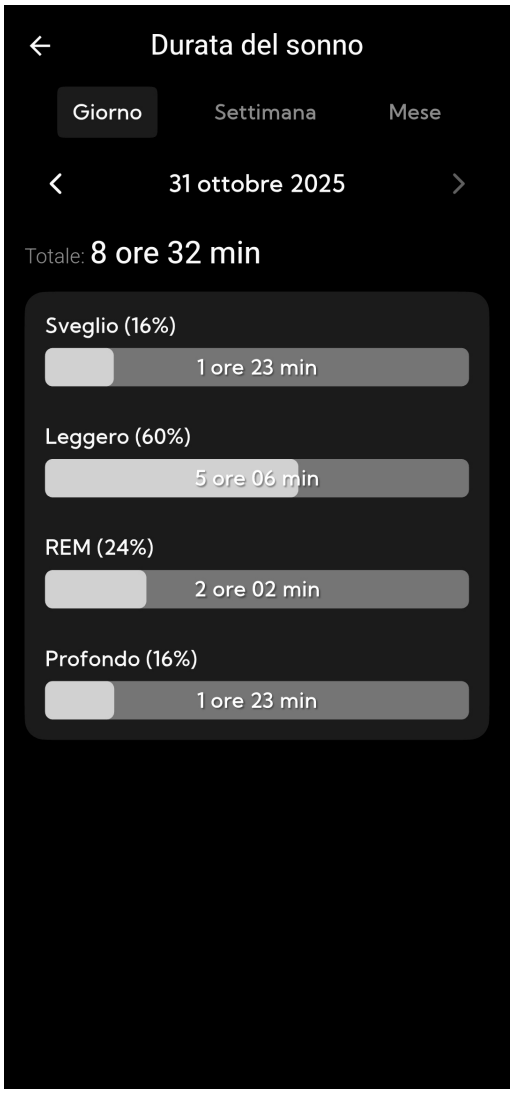
Screenshot 1.5.1: Esempi di schermate per l'input dei dati di metriche manuali



Variabilità della Frequenza Cardiaca (HRV) media settimanale.



Visualizzazione dei passi totali tracciati mensilmente.



Vista giornaliera della durata del sonno e la suddivisione nelle varie fasi.

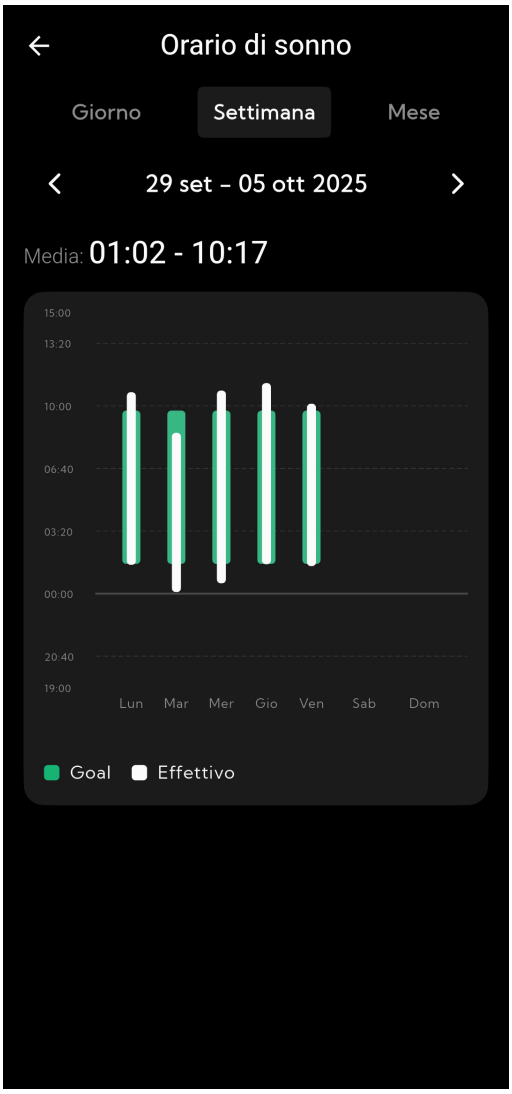
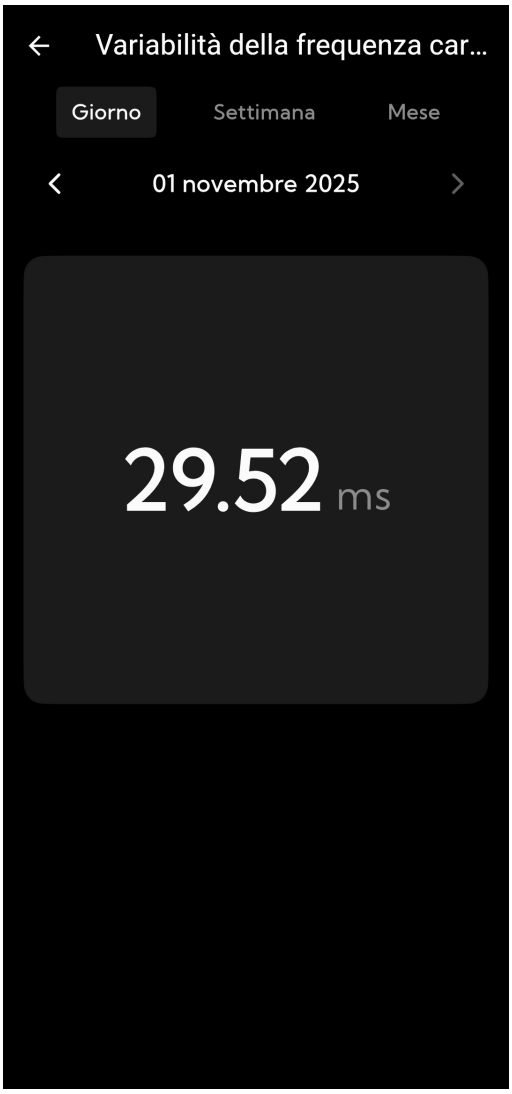


Grafico settimanale che confronta obiettivo e orario effettivo del sonno.



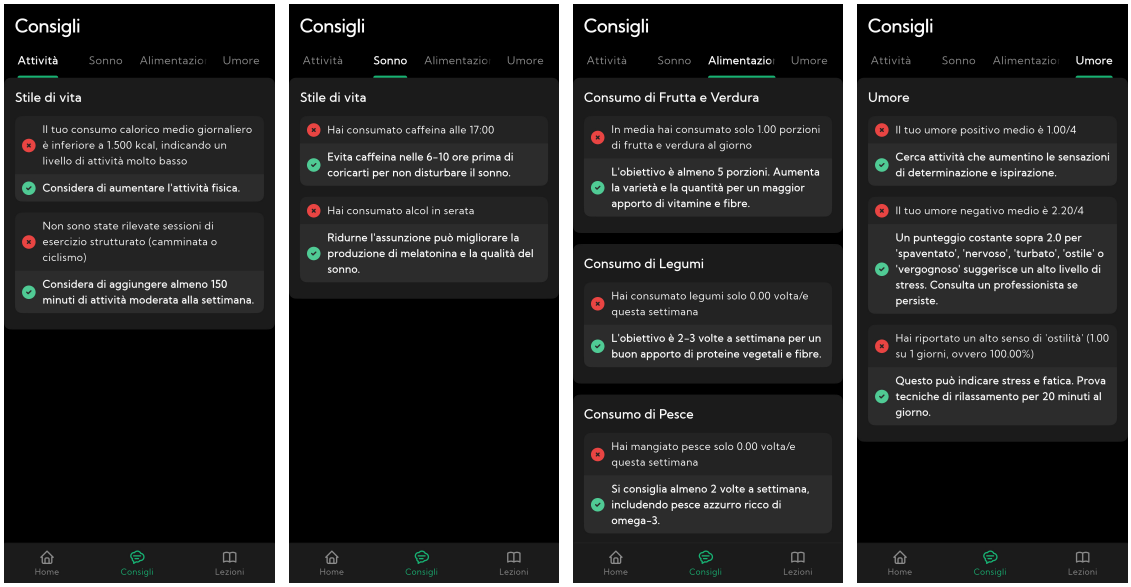
Vista giornaliera della Variabilità della Frequenza Cardiaca (HRV) in ms.

Screenshot 1.5.2: Esempi di visualizzione dei dati di singole metriche

1.6 Tips

Questa schermata mostra semplicemente i consigli personalizzati forniti dal server del Recommender. Vengono suddivisi in due sezioni, Sonno e Cibo, che sono commutabili grazie a una scheda di navigazione (tab) posta in alto. Ciascun consiglio è presentato in un blocco visivamente distinto e strutturato in due componenti principali: la prima evidenzia l'azione (o abitudine) dell'utente

considerata errata, mentre la seconda fornisce la raccomandazione specifica per evitarla in futuro.



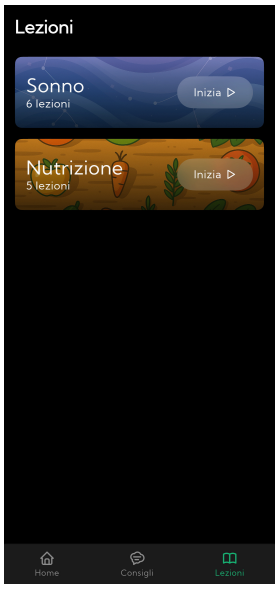
Consigli sull'attività fisica. | Consigli sul sonno. | Consigli sull'alimentazione. | Consigli sull'umore.

Screenshot 1.6.1: Schermate relative ai consigli nei quattro ambiti

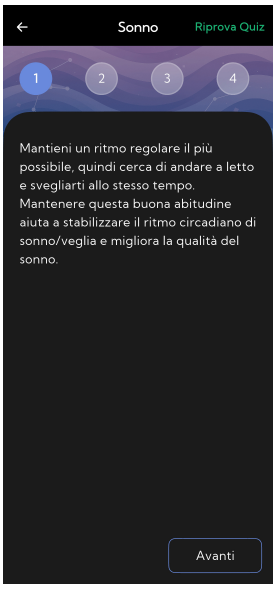
1.7 Lessons

La schermata Lessons é stata ereditata dalla vecchia versione, pur restando soggetta ad un completo redesign. Il suo compito è mostrare, categorizzate per argomento, una serie di pillole (piccoli contenuti informativi) e permettere di mettersi alla prova con dei quiz. Da questa schermata è possibile visualizzare la lista di tutti gli argomenti disponibili nella piattaforma attraverso delle cards che indicano il nome della lezione, il numero di pillole per quella lezione e un tasto start per iniziare. Cliccando sul tasto "Start" viene visualizzata una nuova schermata, da cui è possibile tornare indietro con un tasto presente nella topbar, che mostra il contenuto della pillola nella parte centrale e due tasti di navigazione ´Next´ e ´Back´ posti in fondo alla pagina. Una timeline numerata indica invece la pillola corrente. Arrivati all'ultima pillola in sostituzione al tasto "Back" compare il tasto "Finish" che, una volta cliccato, fa apparire un popup che comunica il completamento della lezione e da cui è anche possibile decidere se tentare o meno il quiz. Inoltre nell'appbar, nel caso in cui l'utente abbia in

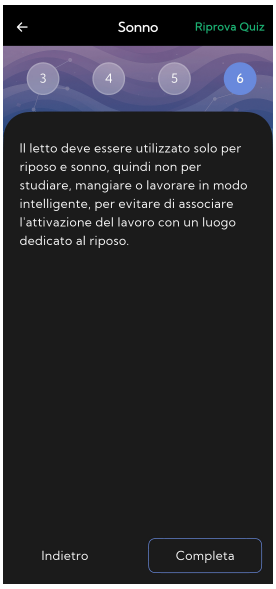
passato completato il quiz relativo alla lezione, vi è un tasto ´Retry Quiz´ per ritentarlo velocemente, evitando di ripercorrere tutte le pillole.



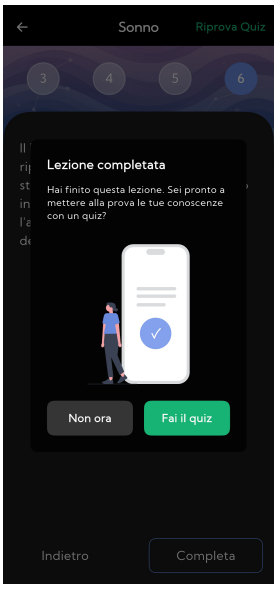
Schermata principale.



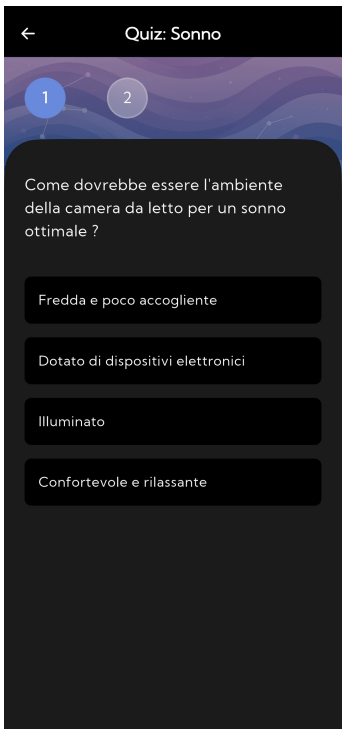
Pagina della lezione sul sonno.



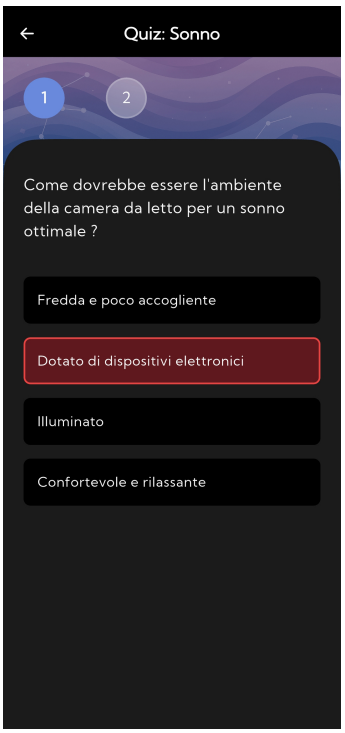
Pagina dell'ultima pillola informativa.



Completamento della lezione.



Schermata del quiz.



Esempio di risposta sbagliata.

Completamento del quiz.

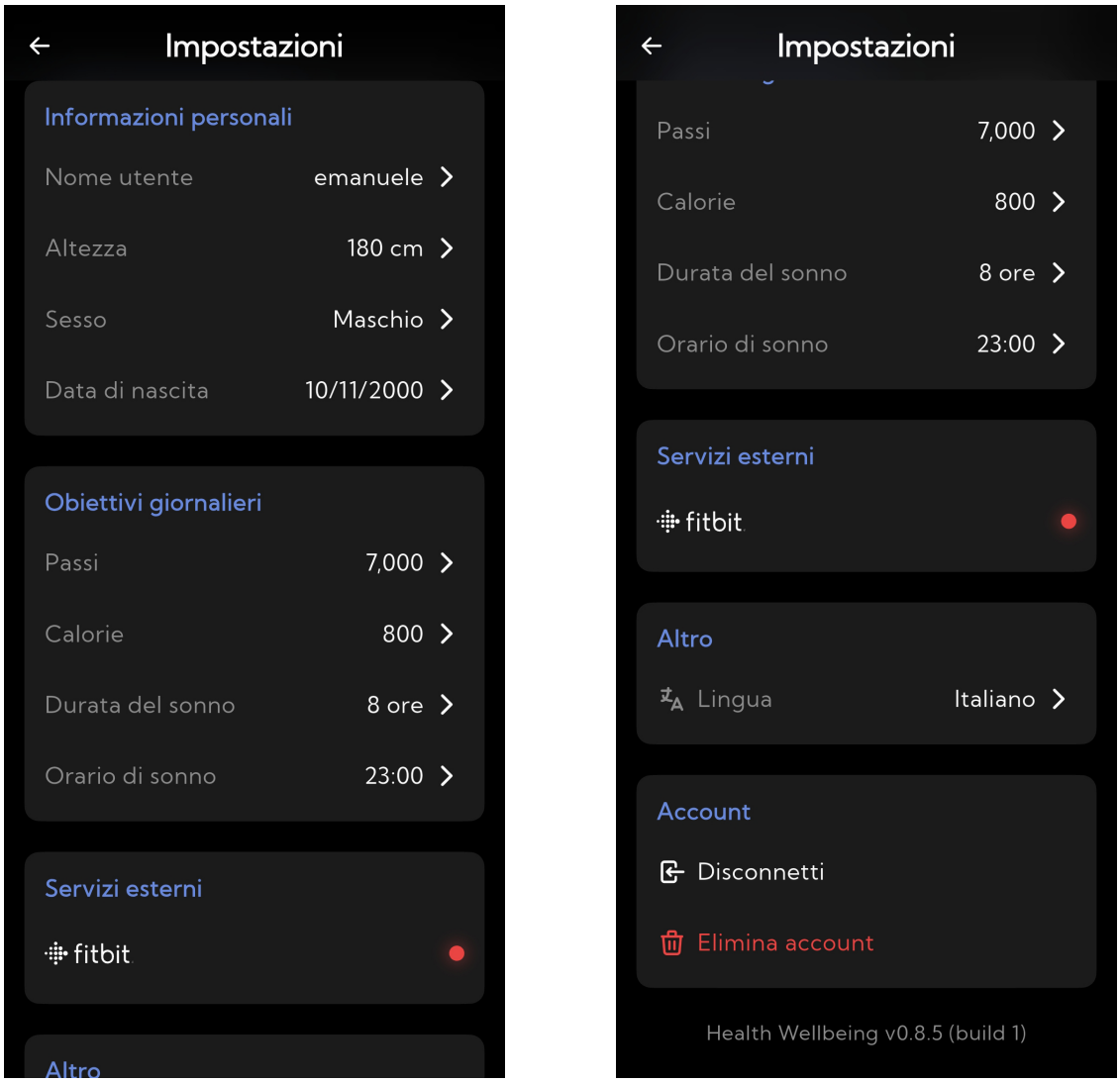
Screenshot 1.7.1: Schermate relative alla lezione sul sonno

1.8 Settings

La schermata ´Settings´, accessibile da un "icon button" presente nell'app bar della "home", contiene tutte le informazioni riguardanti l'utente e gli strumenti per gestirle. Questa schermata presenta diverse voci cliccabili divise per categoria:

- **Personal Information**:
  - Username
  - Height
  - Gender
  - Birthday
- **Daily Goals**:
  - Steps
  - Calories
  - Sleep Duration
  - Sleep Time
- **External services**:
  - fitbit: attualmente l'unica voce presente, se cliccato, permette di fare il login in fitbit.
- **Other**:
  - Language: permette di cambiare la lingua dell'applicazione
- **Account**:
  - Logout: permette di fare il logout
  - Delete Account: permette di eliminare l'account

Tutte le voci della sezione ´Personal Information´ e ´Daily Goals´ se cliccate permettono di modificare il valore da una schermata costruita ad-hoc per quel tipo di informazione. Ho evitato di inserire una schermata transitoria "Profile", tipicamente presente nella maggior parte delle applicazioni, perché avrebbe mostrato informazioni ridondanti accessibili direttamente da "Settings".



Screenshot 1.8.1: Schermata delle impostazioni

Capitolo 2

Scelte Implementative

2.1 Scelta delle Tecnologie

La necessitá di questo progetto è sicuramente quella di creare un'applicazione nativa per offrire buone performance e, soprattutto, facilitare la comunicazione tra quest'ultima e alcuni servizi eseguiti localmente sul dispositivo (come vedremo sará di vitale importanza la comunicazione con Health Connect di Google). Quindi ho da subito scartato l'idea di creare una Web Application che, al contrario, non permetterebbe di accedere facilmente alle risorse software del dispositivo. Allo stesso tempo vi è la necessità di ottenere un prodotto che sia supportato sia dal sistema iOS che Android.

2.1.1 Flutter

Flutter: è un framework open-source supportato da Google che, con una singola codebase, permette di costruire applicazioni multi-piattaforma compilate nativamente [14]. Così come la vecchia versione, ho sviluppato l'applicazione utilizzando questo framework, che si presta bene per lo sviluppo contemporaneo della versione Android e iOS, permette un'elevata personalizzazione dei widget grafici e beneficia di un continuo supporto della community. La decisione è stata inoltre influenzata dalle elevate performance rispetto a quelle offerte da altre soluzioni, dovute alla presenza di un motore di rendering grafico 2D, chiamato Skia, che comunica direttamente con la GPU del dispositivo per disegnare i pixel sullo schermo. Per questo motivo, ho evitato l'uso di soluzioni simili a React Native in cui vi è un 'ponte' che permette al codice non nativo di comunicare con le API e i componenti nativi della piattaforma utilizzata [27].

2.1.2 Firebase

Firebase: è un insieme di servizi per lo sviluppo del back-end tra cui Cloud Firestore, un database noSQL reattivo, Authentication, un servizio che gestisce in modo sicuro le autenticazioni degli utenti, e Cloud Storage, un servizio per l'archiviazione di file [8]. Anche in questo caso ho deciso di ereditare una tecnologia utilizzata dalla versione precedente: questi servizi velocizzano il processo di creazione e gestione del backend, permettendo, durante la creazione del prototipo, di concentrarsi sugli aspetti fondamentali del progetto (come la logica dell'applicazione e la user experience) e posticipare compiti ripetitivi.

2.2 Scelte progettuali del Frontend

2.2.1 Architettura

Un principio di design molto importante in ambito di sviluppo di applicazioni mobile è il SoC (Separation-of-concerns). Questo principio afferma che, per rendere il prodotto testabile, manutenibile e scalabile, bisogna dividerlo in piú unitá, ognuna dedicata a svolgere un ruolo specifico [6]. Questo si traduce nell'implementare le funzionalità seguendo un preciso pattern architetturale che rispetti il principio appena descritto.

Il primo passo è stato quindi scegliere il pattern software architetturale per la creazione del frontend. Considerando l'utilizzo di Flutter ho ridotto le opzioni disponibili a due:

- utilizzare il pattern consigliato da Google (principale supporter del framework) per questo framework, ossia MVVM (Model-View-ViewModel) [33].
- utilizzare BLoC (Business Logic Component), molto famoso nella community di Flutter data la sua capacitá di gestire applicazioni complesse:

Il BLoC è un'architettura in cui lo stato viene gestito tramite stream (sequenza asincrona di dati). La modalitá di funzionamento puó essere descritta in 3 fasi distinte:

1. L'interfaccia invia eventi, come il click su un pulsante da parte dell'utente, al BLoC tramite metodi specifici.
2. Il BLoC basandosi sulla logica di business elabora gli eventi per ottenere un nuovo stato.
3. Infine, il nuovo stato viene inviato alla UI che si aggiorna automaticamente.

Lo stato, come già detto prima, viene gestito sotto forma di stream a cui la UI si sottoscrive. In questo modo l'interfaccia riconosce subito i cambiamenti e può aggiornarsi di conseguenza [25].

Nonostante BLoC sia molto valido e supportato, ho deciso di adoperare il pattern consigliato da Google. L'approvazione di Google garantisce che il design pattern permetta la costruzione di applicazioni robuste, migliorando la manutenzione e velocizzando il flusso di sviluppo [33]. Questo avviene grazie alla separazione della logica dell'applicazione in tre strati:

- **Model:** rappresenta i dati e la logica di business dell'applicazione. Ha il compito di ottenere e processare dati provenienti da un repository o API.

- **View:** Coincide con la UI dell'applicazione. Ha il compito di mostrare l'interfaccia e ascoltare gli aggiornamenti degli stati messi a disposizione dal ViewModel. Quando l'utente compie un'azione, l'evento viene inviato al ViewModel chiamando un metodo definito dallo stesso.

- **ViewModel:** è un ponte tra ´View´ e ´Model´. Offre una serie di metodi per consentire alla View di operare sul Model e allo stesso tempo fornisce proprietà osservabili, a cui la View può essere sottoscritta, che riflettono i dati del Model. Ha anche il compito di trasformare i data model in un formato che faciliti la loro visualizzazione nella UI.

Questa divisione dei ruoli rende l'applicazione maggiormente manutenibile, testabile e scalabile [33].

Per permettere a ´View´ di ascoltare gli aggiornamenti degli stati del ´ViewModel´ si utilizza il ´data binding´, ossia un tecnica generale per sincronizzare i dati tra un producer e un consumer in maniera automatica e reattiva [26]. Nel mio caso per gestire gli stati ho utilizzato il pacchetto ´provider´. Questo pacchetto fornisce un modo per utilizzare in maniera piú semplice e controllata la classe ´InheritedWidget´, una classe nativa di Flutter che permette di propagare le informazioni nell'albero dei widget [5].

I componenti messi a disposizione dal pacchetto che ho utilizzato maggiormente sono:

- la classe `Provider<T>`: permette di rendere disponibile ai figli discendenti un valore T contenuto in esso. Per accedere al valore T, si fa uso di un suo particolare metodo: `Provider.of<T>(BuildContext context, {bool listen = true})`. Per quanto riguarda l'istanziazione degli oggetti, invece, ho fatto uso, per comoditá e per migliorare la leggibilitá del codice, di un MultiProvider che incapsula piú Provider<T> al suo interno [5].

- la classe `ChangeNotifierProvider<T extends ChangeNotifier?>`: è concettualmente simile a Provider, ma in questo caso T è un ChangeNotifier. La classe ChangeNotifier fa parte del core di Flutter. Esso notifica ai suoi "listeners" quando si verificano cambiamenti relativi agli stati al suo interno. Ogni qualvolta nella classe ChangeNotifier viene chiamato il metodo 'notifyListeners()' viene inviata una notifica agli ascoltatori. In definitiva, questo è un modo facile e pronto per implementare il pattern Observer [5].

- la classe `Consumer<T>`: ottiene la classe Provider<T> dai suoi antenati e la passa al builder. All'interno del builder non fa altro che richiamare il metodo Provider.of<T>. Il vantaggio dell'uso di questa classe sta nel fatto che i rebuild non influenzano i widget antenati, ma sono circoscritti al widget costruito dal builder, in questo modo è possibile evitare ricostruzioni inutili. Nel mio caso T si è sempre trattato di un oggetto ChangeNotifier [5].

Sfruttando gli strumenti offerti dal pacchetto, per implementare il pattern MVVM ho potuto:

- Creare i ''ViewModel" attraverso classi che estendono ChangeNotifier: in questo modo possono notificare i ''subscribers", ossia i widgets.

- Istanziare i ''ViewModel" in dei ''ChangeNotifierProvider" posti nel punto dell' albero dei widget in cui diventano necessari per i widget sottostanti.

- All'interno della ''View", ottenere l'istanza del ViewModel di interesse e sottoscriversi ai cambiamenti dei suoi stati creando un ''Consumer".

La 'View' e il 'ViewModel' compongono il "UI Layer", al contrario il 'Model' fa parte del "Data layer", ossia lo strato che si occupa di gestire la logica e i dati di business [33]. Questo layer è formato a sua volta da ''Repositories" e ''Services":

- **Repository:** sono le classi che si occupano di ottenere i dati grezzi dai servizi per poi trasformarli in 'domain model', ossia in dati utili all' applicazione e formattati in modo che possano essere consumati dai ViewModel [33].

- **Service:** sono le classi che compongono il livello piú basso della nostra architettura. Hanno il compito di incapsulare la logica di caricamento dei dati da fonti diverse (ex. REST Endpoint o API native del dispositivo) [33].

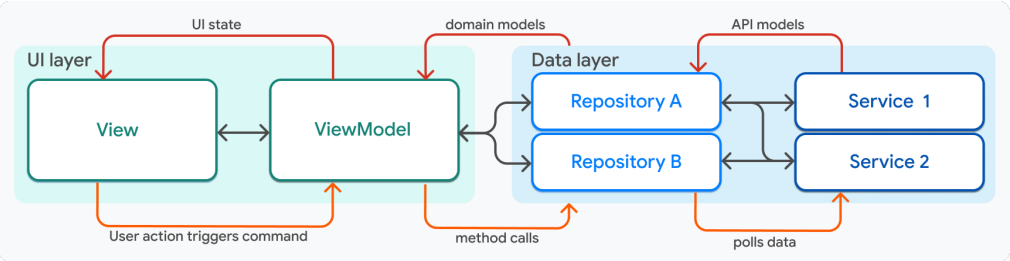


Figura 2.1. Architettura MVVM semplificata in Flutter [33]. Immagine tratta dalla documentazione ufficiale di Flutter (`https://docs.flutter.dev`). © Google LLC, distribuita con licenza Creative Commons Attribution 4.0 International (CC BY 4.0).

2.2.2 Dependency Injection

Talvolta gli oggetti dipendono da altri oggetti che ne forniscono le funzionalità necessarie per raggiungere i propri scopi. Piuttosto che creare gli oggetti necessari all'interno della classe stessa, questo pattern suggerisce di iniettarli dall'esterno.

Per ridurre l'accoppiamento tra le classi, possiamo semplicemente fare riferimento alle dipendenze tramite interfacce, così facendo, in futuro, potremo fornire implementazioni differenti in base alle necessità. A questo punto, per fornire le implementazioni (ossia le istanze delle classi necessarie) di queste interfacce, ci viene in aiuto un framework di dependency injection.

Applicando questi principi otteniamo diversi vantaggi:

- **Accoppiamento debole tra le classi**: aumentando, di conseguenza, la flessibilità del codice.
- **Aumento della testabilità**: grazie alla possibilità di fornire implementazioni diverse delle classi.
- **Riduzione della riscrittura del codice**: grazie al fatto che gli oggetti sono più generici.
- **Migliore gestione delle risorse**: utilizzando un injector è possibile tenere traccia più facilmente di tutti gli oggetti creati, gestendoli tutti nella stessa posizione all'interno del codice.

Nel mio caso, per la gestione delle dipendenze, ho utilizzato il pacchetto `get_it`. `get_it` non è esattamente un dependency injector, ma un *Service Locator* [15]. Nel Service Locator Pattern vi è un registro centrale a cui gli oggetti possono richiedere le dipendenze di cui hanno bisogno; non vengono quindi iniettate direttamente dall'esterno come nel caso precedente.

Entrambi, Service Locator Pattern e Dependency Injection Pattern, implementano il principio IoC (*Inversion of Control*) in cui il controllo non è gestito dagli oggetti stessi, ma da un'entità esterna.

Ho ritenuto l'utilizzo di un Service Locator sufficiente e necessario per la gestione dei vari oggetti della mia applicazione. Ho, di conseguenza, evitato l'utilizzo di librerie di dependency injection perché questo pattern si basa sulla 'riflessione', ossia la capacità di un programma di interrogarsi a runtime per scoprire, ad esempio, quali classi e metodi contiene. Flutter non supporta out-of-the-box la riflessione, in modo da preservare le performance e per via delle restrizioni sulla dimensione dell'applicazione [32].

2.2.3 Struttura del progetto

Dopo aver deciso di adottare il pattern MVVM e aver selezionato gli strumenti necessari per implementarlo, ho focalizzato l'attenzione sulla struttura generale del progetto. Per mantenere il codice organizzato ho creato delle directory apposite avendo come obiettivo una chiara separazione delle responsabilità.

Ho quindi organizzato la struttura del progetto come segue:

- **dtos/**: contiene i *Data Transfer Object*. I DTO sono oggetti il cui scopo è trasferire dati tra il nostro sistema e quelli esterni, ma anche tra servizi e repositories. I servizi, una volta ricevuti i dati grezzi in formato JSON dalle fonti esterne, li convertono in DTO, per ripulire le informazioni non necessarie e per renderle più appetibili dai repositories, e viceversa.

- **exceptions/**: contiene le classi di eccezioni personalizzate. L'uso di eccezioni specifiche per l'applicazione ha migliorato la fase di debug del codice, grazie alle informazioni contenute intrinsecamente in esse.

- **mappers/**: questa cartella contiene le classi che si occupano della mappatura dei dati. Sono di cruciale importanza per poter convertire in maniera dichiarativa e veloce i dati dal formato DTO a Model e viceversa.

- **models/**: in questo caso con "models" mi riferisco ai domain models, ossia degli oggetti che rappresentano i concetti del nostro dominio di business. Essi contengono dati e comportamenti utili ai livelli più alti della nostra applicazione.

- **repositories/**: contiene i repositories, ossia classi il cui scopo è quello di fornire un'astrazione per l'accesso ai dati ai view models. Il loro compito principale è quello di gestire i dati, occupandosi in particolare di trasformare i dati grezzi in domain models e viceversa.

- **services/**: contiene i servizi. Il loro compito è semplicemente quello di incapsulare la logica di accesso ai dati appartenenti a fonti esterne.
- **theme/**: è la cartella che si occupa di contenere il codice dedicato alla gestione della parte estetica dell'applicazione. Qui sono definiti tutti gli elementi (stili, colori, font ecc...) che compongono il design system del progetto.
- **util/**: contiene logica generica che può essere utilizzata in più punti del progetto. Esempi di queste logiche sono i formattatori di date, validatori ecc.
- **viewmodels/**: dove sono salvati tutti i viewmodels dell'applicazione.
- **screens/**: questa cartella contiene tutti i widget che definiscono le pagine complete dell'applicazione (es. `HomeScreen.dart`).
- **widgets/**: contiene widget grafici utilizzati dagli screens; spesso sono modulari e quindi riutilizzati più volte all'interno di una schermata o in più schermate.

Capitolo 3

Backend

Come affermato in precedenza, trattandosi di un prototipo, ho deciso di velocizzare la costruzione del backend affidandomi ad un servizio esterno in cloud che offre la maggior parte delle funzionalità necessarie nel mio caso, ossia **Firebase**. In particolare, questo prodotto mi ha permesso di avere a disposizione, fin da subito, un sistema di autenticazione, un database e uno storage di file, tutti funzionanti e già accessibili in rete [8]. In questo modo ho evitato di dover gestire autonomamente un server e ho potuto concentrarmi sull'implementazione dell'applicazione mobile.

Un lato negativo di questa scelta è l'obbligo di spostare la complessità sul client, dato che il servizio di Firebase per scrivere codice arbitrario, e quindi logica di business, chiamato *Cloud Functions*, è a pagamento. Tuttavia, non avendo per ora logica particolarmente complessa e trattandosi, come già affermato in precedenza, di un prototipo, ho deciso di accettare comunque questo limite.

3.1 Schema del Database

Tralasciando le tecnologie fornite da Firebase, la cui implementazione e funzionamento esulano dagli scopi di questa tesi, mi concentrerò sulla progettazione dello schema del database e sul suo legame con il client.

Rifacendomi alla precedente applicazione e considerando le funzionalità e le informazioni necessarie per animare il mio prototipo UI/UX, ho potuto cogliere le entità fondamentali e definire i loro attributi.

Il database fornito da Firebase, **Firestore Database**, è di tipo **NoSQL** e si basa su documenti, record caratterizzati da un id univoco che memorizzano i dati in coppie chiave-valore. Questa architettura, da un lato, facilita lo sviluppo perché offre flessibilità, grazie alla sua natura *schema-less*, ma dall'altro limita,

come vedremo successivamente, le performance nelle operazioni di ricerca dei dati [10].

Le entità fondamentali, divise per categoria, che ho individuato sono:

- **Entità principale**
  - **User**: composto da tutte le informazioni utili riguardanti un utente.
- **Gestione dei contenuti statici**
  - **Lesson**: composto da un titolo e da tutte le pillole informative relative a quell'argomento.
  - **Quiz**: insieme di domande, ognuna accompagnata da quattro possibili risposte, di cui una corretta.
  - **Food**: nome del cibo con relativa unità di misura.
- **Log e misurazioni**
  - **Measure**: una misurazione biometrica di un certo tipo, relativa a uno specifico utente in un dato giorno.
  - **Quiz Score**: punteggio migliore per uno specifico quiz svolto da uno user.
  - **Food Log**: log di un cibo specifico nella relativa unità di misura effettuato da uno specifico user in un dato giorno.
  - **Mood**: log contenente le risposte al quiz del mood, date da uno specifico user in un dato giorno.
  - **Body Balance**, **Body Strength**, **Grip Strength**, **Weight**, **Waist Circumference**: log relativi alle misurazioni fisiche dell'utente.

Come si può notare, la categoria "Log e misurazioni" (in particolare l'entità *Measure*) contiene dati che necessitano di essere aggregati per utente, tipo e giorno; questo mi ha portato a manipolare lo schema del database per migliorare le performance e adattarlo alle esigenze dell'applicazione mobile.

Complessivamente, la struttura del database è rimasta invariata rispetto alla versione precedente. La differenza sostanziale risiede nel salvataggio delle misurazioni (*Measure*): in precedenza venivano archiviate in file JSON nello storage di Firebase. Per migliorare l'interoperabilità con applicazioni esterne e la velocità di ricerca, ho creato una collezione dedicata nel database, sfruttando le funzionalità di query di Firestore.

Un'altra differenza è la presenza di **Food**, che consente di aggiungere dinamicamente nuovi cibi all'interno dell'applicazione. Ciò evita la duplicazione di informazioni (es. categoria e unità di misura del cibo) e riduce lo spazio di archiviazione.

Attributi specifici delle entità

Entità principale

- **User**:
  - `birth_date (timestamp)`: data di nascita
  - `gender (boolean)`: vero = maschio, falso = femmina
  - `height (number)`: altezza in centimetri
  - `is_onboarding_complete (boolean)`: indica se la fase di onboarding è completata
  - `nickname (string)`
  - `steps_goal (number)`: obiettivo giornaliero di passi
  - `time_sleeping_goal (number)`: minuti di sonno giornalieri
  - `calories_goal (number)`: calorie bruciate giornaliere
  - `bedtime_goal (map<string, number>)`: orario di coricarsi (ora e minuti)

Gestione dei contenuti statici

- **Lesson**:
  - `banner_url (string)`: URL dell'immagine banner (file nello storage di Firebase)
  - `pills (array<string>)`: elenco delle pillole informative
  - `quizId (number)`: ID del quiz associato
  - `title (string)`: titolo della lezione
- **Quiz**:
  - `questions`: array di oggetti con i campi:
    - `questionText (string)`: testo della domanda
    - `possibleAnswers (array<string>)`: risposte possibili
    - `correctAnswer (string)`: risposta corretta
- **Food**:
  - `measure_type (string)`: unità di misura predefinita
  - `name (string)`: nome del cibo

Log e misurazioni

- **Measure** [1]:
  - `measurement_date (timestamp)`: data di misurazione
  - `type (string)`: tipo di misura (es. SpO2, TOTAL_CALORIES_BURNED)
  - `user_id (string)`: ID dell'utente
  - `value (number)`: valore della misurazione
- **Quiz Score**:
  - `score (number)`: punteggio espresso in frazione
- **Food Log**:
  - `date (timestamp)`: data del log
  - `logs (array)`: oggetti con:
    - `food_id (string)`
    - `quantity (number)`

Traducendo in termini di modello E-R, tralasciando le entità relative alla gestione dei contenuti statici e astraendo anche le entità di tipo log otteniamo: 3.1

3.2 Organizzazione del modello nel database NoSQL

Per l'implementazione è stato necessario adattare il modello E–R alle caratteristiche dei database NoSQL a documenti. A differenza dei database relazionali, quelli non relazionali non supportano operazioni di *join*, rendendo indispensabile una progettazione che privilegi l'efficienza delle letture.

[1]Tra le misure vi sono anche documenti relativi alle **attività fisiche svolte**. Questi adottano una struttura diversa che rispecchia, come vedremo in futuro, una classe definita da un pacchetto dart, chiamato "health", che ho impiegato lato front-end:

- `workoutActivityType`: tipo di allenamento
- `totalEnergyBurned`: energia totale bruciata durante l'allenamento
- `totalEnergyBurnedUnit`: unità di misura delle calorie bruciate
- `totalDistance`: distanza totale dell'allenamento
- `totalDistanceUnit`: unità di misura della distanza totale

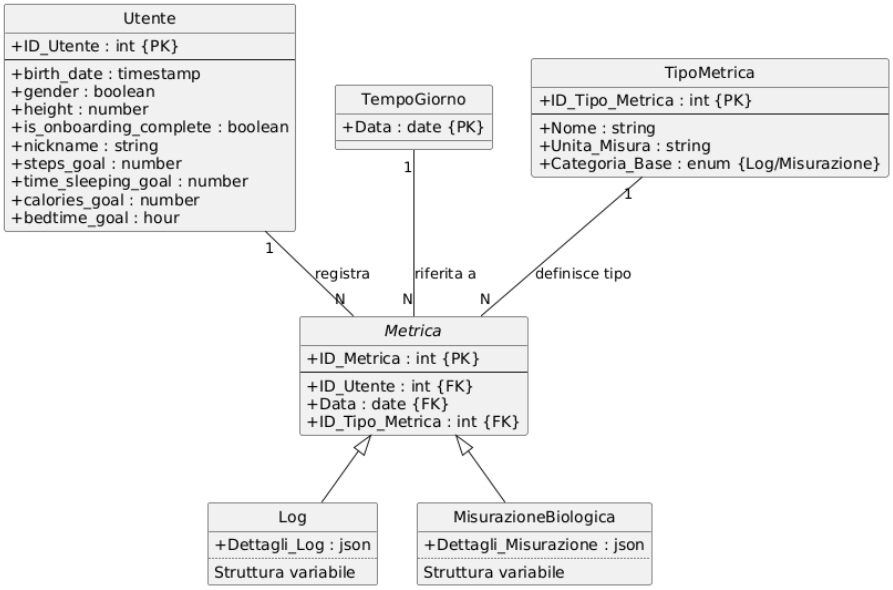


Figura 3.1. Diagramma Entità-Relazione (ER)

Le relazioni uno-a-molti e molti-a-molti sono gestite tramite riferimenti o dati ridondanti, per ottimizzare le query.

Gestione della sincronizzazione dei dati

Essendo che le misurazioni provengono da due fonti diverse nel client e che i log hanno strutture molto differenti, i dati dell'utente vengono organizzati come segue:

```
sync_data (Collezione)
|- {userId} (Documento)
   |- body_balance_data (Collezione)
   |- body_strength_data (Collezione)
   |- fitbit_data (Collezione)
   |- food_data (Collezione)
   |- grip_strength_data (Collezione)
   |- health_package_data (Collezione)
   |- mood_data (Collezione)
   |- quiz_scores (Collezione)
   |- waist_circumference_data (Collezione)
   |- weight_data (Collezione)
   |- fitbitLastSyncDate: String (es. "2025-10-10T16:32:26...")
```

```
\- healthLastSyncDate: String (es. "2025-10-16T09:38:34...")
```

In questo modo, tutti i log sono salvati in sottocollezioni specifiche per ogni utente, mentre le misurazioni si trovano nelle sottocollezioni delle fonti (*Health Connect/Apple HealthKit* o *Fitbit*). I due timestamp finali tracciano l'istante dell'ultima sincronizzazione, centralizzando l'accesso a un solo documento per utente.

3.3 Problema delle query dipendenti da più campi

Per la gestione delle query in Firestore vengono creati automaticamente indici su singoli campi, consentendo ricerche basate su un solo attributo.

Il problema emerge quando le query coinvolgono più campi. In questi casi è necessario usare **indici compositi**, creati su combinazioni specifiche di campi [11].

Nel mio caso questa possibilità è stata essenziale per effettuare query sulla collezione *Measure* che filtrassero per periodo di tempo, utente e tipo di misurazione simultaneamente.

Poiché la creazione di indici composti non è automatica, ogni nuova collezione che ne richiede l'uso necessita di configurazione manuale nella console di Firebase. Tuttavia, nel mio caso — in cui viene creata una collezione per utente alla registrazione — questa soluzione non è praticabile.

Per risolvere il problema, ho creato una collezione dedicata per *Measure*, e per mantenere l'organizzazione logica originale ho impostato gli indici compositi con ambito **"Gruppo di raccolte"** invece che **"Raccolta"**. Questo approccio consente di creare indici validi per tutte le collezioni con lo stesso ID, indipendentemente dal percorso, unificando logicamente le raccolte e permettendo ricerche globali pur preservando la gerarchia originale.

Capitolo 4

Frontend

Dopo aver definito lo schema del database, possiamo passare alla struttura del front-end a partire dai servizi. Una cosa importante da sottolineare è che, nonostante sia partito a discutere del back-end, la sua progettazione è il risultato di un processo che si è svolto in contemporanea con l'applicazione client. Questo perché è stato necessario apportare delle modifiche all'uno e all'altro in corso d'opera, per soddisfare bisogni reciproci come l'aggiunta a posteriori di nuovi campi in delle collezioni specifiche per supportare nuove funzionalità.

Come detto in precedenza, i servizi incapsulano la logica di accesso ai dati forniti da fonti esterne. A questo punto è naturale chiedersi quali siano queste fonti.

Sicuramente una di queste è la nostra applicazione **Firebase**, che abbiamo costruito ad-hoc per archiviare i dati strutturati e non strutturati che il client consuma e manipola. Ma ripensando allo scopo originale dell'applicazione, ci si accorge che buona parte dei dati essenziali sono costituiti dalle misurazioni biometriche fornite dallo smartwatch.

Serviva quindi un modo per accedere in modo unificato a queste misurazioni, considerando che i produttori di smartwatch sono vari, e ognuno di essi offre applicazioni diverse e quindi sistemi diversi per gestirle.

La soluzione perfetta a questo problema sembrava essere il pacchetto per Flutter chiamato **`health`**. Questa libreria funge da supporto per scrivere e leggere dati sanitari e di fitness da e verso le due principali piattaforme attuali, ossia **Apple HealthKit** e **Health Connect**. Sostanzialmente, il pacchetto fa da tramite tra la nostra applicazione Flutter e le API native delle due piattaforme appena citate, facilitando di gran lunga l'accesso ai dati di nostro interesse [1].

Entrando nello specifico, Apple HealthKit (presente in iOS) e Health Connect (presente in Android) sono due "centralizzatori" di dati sanitari e di fitness, permettendo alle varie applicazioni di condividere questi tipi di dati in maniera

sicura.

Il rovescio della medaglia sta nel fatto che, i produttori di terze parti di smartwatch sono svariati, così come svariate sono le relative applicazioni di gestione. Di conseguenza, non possiamo avere la certezza che tutti i produttori abbiano sviluppato un'applicazione che si integri con Apple HealthKit e Health Connect, e che fornisca loro tutte le informazioni per noi importanti. Per questo motivo è stato necessario accogliere l'idea di affidarsi direttamente, completamente o in parte, alle API dei produttori, presupponendo che la maggior parte di essi le mettano a disposizione pubblicamente.

Considerando il gran numero di produttori presenti nel mercato, è difficile pensare di poter creare un supporto per ognuno di essi con il tempo avendo a disposizione. Perciò, ho scelto di integrare solo uno di questi servizi esterni, ossia **Fitbit** di Google, nella speranza che in futuro tutti i produttori supportino i due aggregatori al massimo delle loro capacità.

La stessa Google non scrive tutti i dati ricevuti da Fitbit nel suo aggregatore Health Connect. In particolare, non mette a disposizione quattro misurazioni per noi di vitale importanza: **SpO_2** (saturazione dell'ossigeno), **HRV** (variabilità della frequenza cardiaca), **Resting HR** (frequenza cardiaca a riposo) e **Breathing Rate** (frequenza respiratoria).

Considerato quest'ultimo problema, complessivamente avremo **tre fonti di dati diverse**: la nostra applicazione **Firebase**, **Apple HealthKit/Health Connect** (pacchetto `health`) e **Fitbit API**.

4.1 Servizi

L'architettura del sistema, a questo livello, è costituita da un numero di servizi pari al numero di concetti appartenenti al nostro dominio.

Caratteristiche comuni a tutti i servizi è la loro funzione di **conversione di dati grezzi**, provenienti dall'esterno, in **DTO** e viceversa, e l'impiego di blocchi `try-catch` per la gestione dell'errore. In particolare in caso di fallimento viene lanciata, ai livelli più alti, un'eccezione personalizzata o non, che ne descrive, in ogni caso, la motivazione.

4.1.1 1 - AuthService

Questo servizio si occupa principalmente di interagire con il **sistema di autenticazione di Firebase**. Fornisce metodi per accedere e registrarsi (con email e password oppure utilizzando Google), e per eseguire il logout. Inoltre ha anche il compito di inizializzare nella collezione `users_data` il documento relativo all'utente, appena prima dell'inizio della fase di onboarding.

Le dipendenze sono un'istanza di `FirebaseAuth`, `FirebaseFirestore` (per inizializzare i dati dell'utente in fase di registrazione) e `GoogleSignIn` (per gestire il login con Google).

4.1.2 2 - UserService

Dedicato a creare, ottenere e modificare i dati riguardanti l'utente loggato in quel momento. Cosa importante da notare è che il metodo che si occupa di fornire le informazioni dell'utente restituisce uno **stream**, permettendo così di ottenere **aggiornamenti in tempo reale** nella nostra applicazione.

L'unica dipendenza che troviamo in questo caso è un'istanza di `FirebaseFirestore`.

4.1.3 3 - HealthService

Questo servizio è dedicato all'interazione con le **API di salute native dei dispositivi** (Health Connect e Apple HealthKit). Ci permette di acquisire facilmente i cosiddetti `HealthDataPoint`, ossia una classe fornita dalla libreria, le cui istanze corrispondono all'unità di dato di HealthKit o Health Connect. Questi dati, sono gli stessi che verranno restituiti ai livelli superiori, avendo ritenuto non necessario creare una classe personalizzata.

La libreria offre anche funzioni di aggregazione, ma queste si limitano ad aggregare i dati utilizzando unicamente l'operatore di somma. Questo mi ha portato, come vedremo nella sezione dedicata ai repository, a implementarle autonomamente.

Le sue dipendenze sono un'istanza di `Health`, una classe messa a disposizione dalla libreria `health`, di cui abbiamo già discusso, e un'istanza di `ServiceStatusProvider`, una classe da me creata che fornisce informazioni sul sistema operativo attualmente in uso e un booleano per capire se l'utente è loggato in Fitbit o meno. In questo caso, ci interessa conoscere l'OS del dispositivo perché la libreria `health` fornisce due interfacce diverse per i dati sulla variabilità cardiaca, `HealthDataType.HEART_RATE_VARIABILITY_RMSSD` per Android e `HealthDataType.HEART_RATE_VARIABILITY_SDNN` per iOS [1].

4.1.4 4 - FitbitService

Si occupa di gestire le chiamate alle **API di Fitbit**. Per supportare questa funzionalità mi sono affidato ad un'altra libreria, **Fitbitter** che gestisce automaticamente il flusso di **autenticazione OAuth2** e fornisce dei metodi semplici per richiedere i dati [2].

Questo servizio offre quindi metodi per autorizzare Fitbit, rimuovere l'autorizzazione, gestire le credenziali e per richiedere dati relativi alla variabilità cardiaca, SpO_2, frequenza respiratoria e battito cardiaco a riposo, ossia i dati di cui abbiamo bisogno e che attualmente non vengono forniti dall'applicazione di Fitbit ai due aggregatori di dati di salute.

4.1.5 5 - SynchronizationService

Servizio dedicato alla **gestione della sincronizzazione** delle misurazioni biometriche di Fitbit e Health Connect/Apple HealthKit con il database di Firebase. Fornisce due metodi per accedere al timestamp dell'ultima volta in cui è stata effettuata la sincronizzazione (uno per Fitbit e uno per i due aggregatori) e ulteriori due metodi per salvare effettivamente i nuovi dati nelle due sottocollezioni distinte, `health_package_data` e `fitbit_data`. Per il salvataggio di questi dati ho creato un'istanza di **WriteBatch** offerta dalla libreria di Firebase, un oggetto che permette di eseguire **scritture multiple in maniera atomica** incapsulando tutte le modifiche temporanee prima di eseguire il commit; in questo modo evitiamo scritture incomplete o non coerenti [9]. In queste operazioni di scrittura rientra anche quella per aggiornare il timestamp dell'ultima sincronizzazione effettuata.

4.1.6 6 - FoodService

Dedicato alla gestione del cibo. Offre un metodo per ottenere la lista di tutti i cibi disponibili e, corrispettivamente, un metodo per salvare i log dei cibi nella giornata odierna e un altro per ottenere quelli relativi a una data specifica.

Dipende unicamente dall'istanza di `FirebaseFirestore`.

4.1.7 7 - MoodService

Dedicato alla gestione dei log sull'umore.

Dipende solo dall'istanza di `FirebaseFirestore` e fornisce un metodo per ottenere il log di uno specifico giorno e un altro per salvare un log nella giornata odierna.

4.1.8 8 - LessonService

Questo servizio gestisce l'accesso a **lezioni e quiz**. Anche in questo caso, dipende unicamente da un'istanza di `FirebaseFirestore`.

In particolare, offre un metodo per ottenere una lista di tutte le lezioni disponibili nel database di Firestore, con associati i quiz. Offre inoltre altri

due metodi: uno per salvare il punteggio migliore raggiunto dall'utente per uno specifico quiz e uno per ottenerlo.

4.1.9 9 - RecommenderService

Un servizio che comunica con il server del **"Recommender"**, il quale fornisce i consigli da visualizzare nella schermata "Tips".

Offre due metodi per ottenere i consigli dell'utente autenticato, relativi al sonno e all'alimentazione. Una sua dipendenza importante è un'istanza del **client HTTP** fornito dal pacchetto **"Dio"**. Ho scelto questa soluzione perché permette di definire degli **interceptor**, ossia dei middleware che permettono di intercettare e modificare le richieste (in entrata) e le risposte (in uscita) [16].

Questo strumento ha semplificato notevolmente il processo di inclusione in ogni richiesta dell'**header di autenticazione** (contenente il bearer Token di Firebase) richiesto dal destinatario.

4.1.10 Servizi Restanti

I servizi restanti, `BodyBalanceService`, `BodyStrengthService`, `GripStrengthService`, `WaistCircumferenceService` e `WeightService` offrono tutti le stesse funzionalità ma per strutture dati diverse. Ognuno di essi implementa due metodi, uno per salvare i log e l'altro per ottenerli in base a un periodo di tempo specifico. Per evitare duplicazione del codice, ho optato per la creazione di una **classe astratta**, `ManualInputMeasureService` che, con la condizione di ricevere una struttura dati serializzabile in JSON, implementa le due funzioni citate sopra. Successivamente ho creato le classi concrete facendo in modo che estendessero quella astratta.

4.2 Repository

Passando al livello superiore, abbiamo i **repository**. I repository sono quelle classi che hanno il compito di fornire ai `ViewModel` l'accesso ai **domain model**. Nel mio caso avendo un backend privo di logica di business, ho sfruttato questo livello architetturale per **gestire ulteriormente i dati**, in particolare per **aggregarli**, ma anche per implementare funzioni come la **sincronizzazione**.

Questo sposta la complessità nel client, che deve gestire tutta la logica internamente, causando notevoli rallentamenti dell'applicazione, soprattutto in fase di startup per via della sincronizzazione. Come discusso in precedenza, è un rischio che ho accettato ripensando ai fini del progetto.

Inoltre, considerando che il nostro sistema è relativamente semplice, limitandosi spesso a eseguire operazioni CRUD, e che l'architettura dello schema del database è suddiviso rispettando i concetti di dominio di cui abbiamo bisogno nella nostra applicazione client, il livello repository si rivela spesso superfluo, riducendosi a convertire DTO in domain model e viceversa, e a chiamare i metodi dei servizi. Nonostante ciò, ho deciso di lasciare intatta questa struttura da un lato per inserire la logica aggiuntiva di cui ho bisogno, dall'altro per facilitare sviluppi futuri.

Tutti i repository dipendono da **uno o più servizi**, per l'accesso ai dati, e da **istanze di classi `Mapper`**, per convertire DTO in Domain Model e viceversa.

4.2.1 HealthRepository

Il repository che tratterò in maniera più approfondita è quello relativo alle **misurazioni biometriche e ai dati sanitari**, perché è l'unico, in questo livello architetturale, che contiene **logica complessa**. Tutti gli altri repository, come affermato poco fa, si limitano a fare da strato intermedio tra `ViewModel` e servizi, non avendo l'esigenza di processare particolarmente i dati.

L'`HealthRepository` ha il compito di **unificare l'accesso ai dati sanitari**, che provengono sia dal dispositivo stesso (Health Connect/Apple HealthKit), attraverso la libreria `health`, sia dal sistema esterno di Fitbit. Più precisamente, vedremo che a questi due si aggiungerà come terza fonte Firebase, per via di alcuni limiti imposti da Google che riguardano le API di Fitbit.

Questa classe complessivamente offre:

- Metodi per richiedere tutte le misurazioni biometriche utili all'applicazione in un certo periodo di tempo e, qualora il tipo di dato lo supporti (es. passi, battito cardiaco), **aggregati in intervalli** suddivisi in base a un numero arbitrario di ore.

- Due funzioni per avviare, rispettivamente, la **sincronizzazione** dei dati provenienti da Google Health/Apple HealthKit e quelli provenienti dal sistema di Fitbit.

1 - Metodi per ottenere i dati sanitari

Questi metodi richiedono come parametri:

- `startDate`: inizio dell'intervallo richiesto

- `endDate`: fine dell'intervallo richiesto

- `aggregationIntervalHours`: numero di ore da cui sono costituiti gli intervalli richiesti

Possiamo suddividere il loro funzionamento in tre fasi:

1. **Acquisizione dei dati dai servizi**: nel caso in cui il dato è ottenibile da `FitbitService` e l'utente ha effettuato il login in Fitbit, viene **preferito come fonte** dei dati. L'output di questa fase consiste in istanze di `HealthDataPoint` oppure istanze di `FirebaseMeasurementDTO`.

2. **Aggregazione dei dati**: questa fase viene gestita tramite i metodi offerti da una classe di utilità creata appositamente, `HealthDataAggregator`.

   Questa classe offre metodi per aggregare diversi tipi di dati tra cui, `HealthDataPoint`, classe definita dalla libreria `health`, `FirebaseMeasurementDTO`, classe creata ad-hoc e tipi specifici di `HealthDataPoint`, ossia quelli riguardanti le varie fasi del sonno e `WORKOUT` che, avendo una struttura più complessa da aggregare, vengono gestiti singolarmente.

   Il cuore della logica di queste funzioni è la stessa e può essere suddivisa in varie fasi:

   - **Inizializzazione**: se il vettore dei dati in ingresso non è vuoto viene inizializzata la lista, chiamata `aggregatedResults`, che conterrà i dati aggregati. In caso contrario la funzione restituisce il valore `null`.
   - **Iterazione sugli intervalli di tempo**: viene avviato un ciclo `while` con indice inizializzato a `startTime` che si conclude una volta raggiunto `endTime`. Ad ogni iterazione l'indice viene incrementato della durata dell'intervallo ricevuto come parametro.
   - **Filtraggio dei dati di input**: la lista ricevuta come parametro viene filtrata eliminando tutti i punti non appartenenti all'intervallo relativo all'iterazione corrente.
   - **Aggregazione**: i punti vengono aggregati in base al valore del parametro `AggregateType`, un `enum` che può assumere due valori:
     - `AggregateType.total`: Se il tipo di aggregazione è `total`, la funzione somma i valori numerici di tutti i punti presenti nell'intervallo, utilizzando il metodo `fold` per una somma efficiente.
     - `AggregateType.avg`: Se il tipo è `avg`, la funzione calcola la media dei valori.

     Se non ci sono punti nell'intervallo, viene assegnato il valore del parametro opzionale `defaultValueForEmptyIntervals`, che se non fornito è 0.0.

- **Inserimento del risultato**: una volta calcolato il valore aggregato, viene creato un nuovo oggetto `HealthAggregatedData`, una classe di supporto, che contiene il nome del tipo di dato, l'ora di inizio dell'intervallo e il valore calcolato. Questo oggetto viene poi aggiunto alla lista `aggregatedResults`. Fatto ciò, l'indice viene aggiornato e il ciclo viene ripetuto se la condizione non è ancora soddisfatta.

3. **Conversione finale dei dati**: gli `HealthAggregatedData` vengono convertiti, a seconda del tipo di dato, in una semplice lista di `double` oppure in una lista di `HealthData` (classe formata solo da due attributi, `value` e `dateOfMonitoring`) o `SleepData` (classe formata da quattro `double` contenente il numero di minuti, uno per ogni fase del sonno). In particolare, `HealthData` e `SleepData` si riferiscono a metriche giornaliere.

Ottimizzazione delle chiamate In merito alla prima fase, viene da chiedersi cosa sia la classe `FirebaseMeasurementDTO` e perché non vengano semplicemente richiamati i metodi di `FitbitService`, che, diversamente, restituiscono istanze di classi che estendono `FitbitData`.

La classe `FirebaseMeasurementDTO` rappresenta la struttura grezza delle misurazioni presenti nel database di Firebase grazie alla sincronizzazione. La scelta dietro l'utilizzo di `HealthService`, per ottenere indirettamente i dati di Fitbit, è giustificata dalla necessità di un'**ottimizzazione dovuta ai limiti imposti dalle API di Fitbit**. Nello specifico, la limitazione riguarda il numero massimo di richieste effettuabili da un utente in un'ora, che è fissato a 150 [23]. Inoltre, di queste richieste fanno parte anche quelle dedicate alla conferma della validità del token, che sono utilizzate frequentemente dall'applicazione, sia direttamente, che indirettamente attraverso la libreria `fitbitter`. E' stato quindi necessario adottare una serie di contromisure per aggirare questo problema; una tra queste consiste proprio nell'acquisire i dati provenienti originariamente da Fitbit sfruttando la sincronizzazione su Firebase.

2 - Metodi di sincronizzazione

In merito alla sincronizzazione, vengono messi a disposizione due metodi per gestire i dati di salute da due fonti diverse, `FitbitService` e `HealthService`.

Possiamo suddividere la logica di `syncFitbitData()` in cinque fasi:

1. **Verifica dell'autenticazione**: se non vi è nessun login a Fitbit, la funzione ritorna senza fare altre operazioni. Se invece vi è un login a Fitbit ma nessun utente è loggato nella nostra applicazione, attraverso Firebase, viene lanciata un'eccezione.

2. **Controllo della frequenza di sincronizzazione**: questa fase fa parte di quelle soluzioni che ho adottato per aggirare il problema delle 150 chiamate massime orarie alle API di Fitbit. Sostanzialmente viene richiesto a `SynchronizationService` il timestamp dell'ultima volta in cui è stata effettuata la sincronizzazione dei dati da Fitbit: se **non è passata almeno un'ora** da quest'ultima la funzione si ferma.

3. **Calcolo dell'intervallo di tempo**: viene creata una data di inizio, `startDate`, e una data di fine, `endDate`. Quella d'inizio è impostata all'ultima data di sincronizzazione, a patto che **non siano trascorsi più di 29 giorni**. Diversamente viene impostata a 29 giorni prima della data odierna, compreso il caso in cui l'utente si è appena registrato.

   Questo limite è causato ancora una volta dalle API di Fitbit, che per un vasto numero di metriche, impone come limite massimo di intervallo **30 giorni** [13]. Per quanto riguarda `endDate`, quest'ultimo viene sempre impostato alla data odierna.

4. **Recupero dei dati da `FitbitService`**: utilizzando le chiamate specifiche, messe a disposizione dal servizio `FitbitService`, vengono recuperate le quattro metriche di nostro interesse:

   - Variabilità della frequenza cardiaca (HRV)
   - SpO_2 (saturazione di ossigeno nel sangue)
   - Frequenza respiratoria
   - Frequenza cardiaca a riposo

5. **Salvataggio dei dati su Firebase**: Dopo aver recuperato tutti i dati, la funzione li passa al metodo di `SynchronizationService`, che si occupa del salvataggio sul database di Firestore per l'utente corrente.

Invece, per quanto riguarda il metodo `syncHealthData()`, il funzionamento è simile a quello appena visto, se non per il fatto che la fonte dei dati è `HealthService` e che, **non avendo il limite delle API di Fitbit**, `startDate` viene impostato sempre al timestamp dell'ultima sincronizzazione (se disponibile).

4.3 ViewModel

I ViewModel, come anticipato, si occupano di gestire la logica dell'interfaccia utente, esponendo alla View degli stati da osservare e dei metodi.

Interessante, in questo caso, è il modo in cui ho gestito lo stato attuale del ViewModel nei vari metodi che definisce. Questo stato suggerisce alla View cosa visualizzare in quel preciso momento, variando spesso fra: una schermata di caricamento, di errore o con i dati disponibili.
Per farlo ho creato degli enum appositi: un esempio é **FitbitViewState** formato da cinque stati: initial, loading, authorized, unauthorized, error. Inizialmente lo stato é impostato su initial, ogni qualvolta vengano richiesti nuovi dati ai repository, lo stato viene aggiornato a loading. Nel caso in cui si riceva un errore dal repository si passa allo stato error, nel quale è opzionalmente possibile fornire piú informazioni creando nel ViewModel una variabile `String` *_errorMessage*. Sfruttando un enum evitiamo tutte quelle problematiche legate a stati impossibili, che invece sarebbero presenti se, ad esempio, dedicassimo una variabile distinta per stato. Inoltre, questo approccio rende il codice più leggibile e quindi maggiormente manutenibile.

4.3.1 Calcolo dei punteggi per i vari ambiti

Un ViewModel di grande interesse è quello dedicato al calcolo dei punteggi per ogni ambito del benessere psicofisico.
Per quanto riguarda l'ambito "Cibo", non ho ritenuto possibile calcolarne uno, a causa della mancanza di informazioni sulla quantità di calorie o macronutrienti per le singole categorie di cibo.
Una possibile soluzione era quella di considerare nel punteggio delle medie di quei valori per categoria. Tuttavia, a posteriori ho concluso che non sarebbero significative, per via dell'alta variabilità nutrizionale tra cibi diversi appartenenti alla stessa categoria. Di conseguenza, è stato necessario escludere l'ambito "**Cibo**" dal calcolo del punteggio.

Algoritmi per il Calcolo dei Punteggi di Benessere

Il ViewModel `HealthScoresViewModel` è responsabile di calcolare i punteggi per l'attività fisica, l'umore e il sonno.

Punteggio dell'Attività Fisica (`Activity Score`)

Il punteggio dell'attività fisica (Activity Score) è un punteggio composito, che consiste in tre componenti principali: passi, frequenza cardiaca a riposo (RHR) ed allenamenti (Workouts). Ogni metrica ha un peso specifico da prendere in considerazione per il punteggio complessivo.

Formula Composita Il punteggio finale è calcolato come media ponderata dei tre sotto-punteggi:

$$\text{Activity Score} = (\text{Score}_{\text{Passi}} \cdot 0.35) + (\text{Score}_{\text{RHR}} \cdot 0.30) + (\text{Score}_{\text{Workout}} \cdot 0.35) \quad (4.1)$$

Tabella 4.1. Ponderazione delle metriche nel Punteggio di Attività

Metrica	Peso
$\text{Score}_{\text{Passi}}$	35%
$\text{Score}_{\text{RHR}}$	30%
$\text{Score}_{\text{Workout}}$	35%

Sotto-Punteggi

1. **Punteggio Passi ($\text{Score}_{\text{Passi}}$):** Misura il raggiungimento dell'obiettivo giornaliero di passi ($\text{Target}_{\text{Passi}}$). Normalizzato e troncato a 100 se l'obiettivo è superato.
$$\text{Score}_{\text{Passi}} = \min\left(1, \frac{\text{Passi}_{\text{Totali}}}{\text{Target}_{\text{Passi}}}\right) \cdot 100 \quad (4.2)$$

2. **Punteggio Frequenza Cardiaca a Riposo ($\text{Score}_{\text{RHR}}$):** E' stato utilizzato un modello di distribuzione di **Cauchy-Lorentz (a campana)** per assegnare un punteggio ottimale di 100 ad un RHR ideale (fissato a 60 bpm) e penalizzare per le deviazioni.
$$\text{Score}_{\text{RHR}} = 100 \cdot \left(\frac{1}{1 + \left(\frac{|\text{RHR}_{\text{Media}} - 60|}{10}\right)^2}\right) \quad (4.3)$$
Il punteggio decade in maniera proporzionale rispetto ad un fattore 10 nel denominatore man mano che ci si allontana dal valore ideale.

3. **Punteggio Allenamento ($\text{Score}_{\text{Workout}}$):** Combina il raggiungimento degli obiettivi di distanza percorsa e calorie bruciate durante gli allenamenti.
$$\text{Score}_{\text{Workout}} = \min\left(100, \frac{\text{Score}_{\text{Distanza}} + \text{Score}_{\text{Energia}}}{2}\right) \quad (4.4)$$
Dove:
$$\text{Score}_{\text{Distanza}} = \min\left(1, \frac{\text{Distanza}_{\text{Totale}}}{\text{Target}_{\text{Distanza}}}\right) \cdot 100$$

$$\text{Score}_{\text{Energia}} = \min\left(1, \frac{\text{Calorie}_{\text{Bruciate}}}{\text{Target}_{\text{Calorie}}}\right) \cdot 100$$

Punteggio Umore (`Mood Score`)

Il punteggio dell'umore (Mood Score) è calcolato a partire dalle risposte fornite dall'utente a un questionario di 10 domande, ciascuna con risposta su una scala da 0 a 4. Il punteggio massimo possibile è 40.

Algoritmo di Normalizzazione L'algoritmo inverte il punteggio per le domande in cui un valore inferiore indica un benessere maggiore, e successivamente normalizza il punteggio complessivo su una scala 0-100.

1. **Domande** 1 **a** 5 **(Punteggio Diretto):** Vengono sommate direttamente.

$$\text{Somma}_{\text{Pos}} = \sum_{i=1}^{5} \text{Risposta}_i$$

2. **Domande** 6 **a** 10 **(Punteggio Invertito):** Il punteggio viene invertito sottraendo il valore della risposta al massimo possibile (Massimo = 4).

$$\text{Somma}_{\text{Inv}} = \sum_{i=6}^{10} (4 - \text{Risposta}_i)$$

3. **Punteggio Giornaliero:** Il punteggio finale dell'umore viene normalizzato.

$$\text{Score}_{\text{Giornaliero}} = \text{round}\left(\frac{\text{Somma}_{\text{Pos}} + \text{Somma}_{\text{Inv}}}{40} \cdot 100\right)$$

Punteggio Sonno (`Sleep Score`)

Il punteggio del sonno (Sleep Score) è una media ponderata tra la **durata** del sonno e la sua **qualità**, misurata in base alla percentuale di fasi REM e Profondo.

Formula Composita

$$\text{Score}_{\text{Sonno}} = (\text{Score}_{\text{Durata}} \cdot 0.5) + (\text{Score}_{\text{Qualità}} \cdot 0.5) \tag{4.5}$$

Sotto-Punteggi

1. **Punteggio Durata ($\text{Score}_{\text{Durata}}$):** Questo punteggio viene calcolato in base al raggiungimento dell'obiettivo di sonno in minuti, che l'utente ha definito.
$$\text{Score}_{\text{Durata}} = \min\left(1, \frac{\text{Minuti}_{\text{Totali}}}{\text{Target}_{\text{Minuti}}}\right) \cdot 100$$

2. **Punteggio Qualità ($\text{Score}_{\text{Qualità}}$):** Questo punteggio è la media dei punteggi ottenuti per le fasi REM e Profonda.
$$\text{Score}_{\text{Qualità}} = \frac{\text{Score}_{\text{Profondo}} + \text{Score}_{\text{REM}}}{2}$$

Metodologia per le Fasi di Sonno ($\text{Score}_{\text{Fase}}$) Per ogni fase di sonno (Profondo e REM), il punteggio viene assegnato in base alla percentuale di tempo trascorso in quella fase rispetto al totale del sonno, applicando un meccanismo di penalità:

- Il punteggio è 100 se la percentuale della fase rientra nell'intervallo ideale. Secondo studi medici è ormai chiaro che la qualità del sonno in un adulto è fortemente correlata alle ore passate durante le fasi ristorative. In particolare si raccomandano percentuali sul totale del sonno di: Sonno REM: 20%-25% Sonno Profondo: 13%-23% [3].
- Se la percentuale si discosta dall'intervallo ideale, viene applicata una penalità proporzionale alla deviazione.

$$\text{Score}_{\text{Fase}} = \max\left(0, 100 - \left(\frac{|\text{Valore}_{\text{attuale}} - \text{Valore}_{\text{ideale}}|}{\text{Valore}_{\text{ideale}}} \cdot 100\right)\right)$$

Dove $\text{Valore}_{\text{ideale}}$ è l'estremo dell'intervallo ideale più vicino al $\text{Valore}_{\text{attuale}}$. Il risultato finale è la media dei punteggi giornalieri.

4.4 View

Le viste, in Flutter, sono rappresentate dai **widget**, i quali, nell'architettura MVVM, dovrebbero visualizzare i dati di business solo se forniti dai rispettivi `ViewModel`.

Per poter osservare i dati che ci interessano dal ViewModel all'interno del widget, possiamo sfruttare le classi del pacchetto "provider" (es. `Consumer`)

oppure alcune classi disponibili nativamente in Flutter (`StreamBuilder`, `FutureBuilder`).

Dovendo fare richiesta dei dati a un database remoto, ci si ritrova spesso a dover gestire oggetti di tipo:

- **Future**: Rappresenta un'azione asincrona il cui risultato non è immediatamente disponibile.
- **Stream**: Rappresenta una sequenza di dati asincrona, consentendo di ottenere aggiornamenti in tempo reale del dato T.

Nel mio approccio, dal punto di vista dell'interfaccia, ho gestito gli `Stream` tramite il widget `StreamBuilder`, che definisce l'aspetto che deve assumere l'interfaccia grafica per ogni stato del flusso asincrono (riconducibili a `WAITING`, `ERROR`, `HAS_DATA`) e la ricostruisce in modo reattivo ogni qualvolta vi è un aggiornamento.

Per i `Future` invece, ho alternato l'utilizzo di due metodi:

- L'utilizzo del widget `FutureBuilder`, che ha un comportamento concettualmente simile a `StreamBuilder`.
- La chiamata al metodo che restituisce il `Future` direttamente nell'`initState` del widget:

```
@override
void initState() {
  super.initState();
  //richiedo il riferimento all'istanza del ViewModel
  //che mi interessa
  viewModel = Provider.of(
    context,
    listen: false);
  WidgetsBinding.instance.addPostFrameCallback((_) {
    //aggiorno una variabile di stato esposta dal ViewModel
    //per ottenere il dato di interesse
    onboardingViewModel.obtainMyFutureMethod();
  });
}
```

In questo semplice esempio, il metodo `obtainMyFutureMethod` definito nel `genericViewModel`, è incaricato di gestire manualmente alcune variabili definite nel `ViewModel` stesso. Tra queste, la variabile che ospiterà il risultato del `Future` (inizialmente `null`) ed eventuali variabili di stato per gestire le

fasi di caricamento dell'interfaccia (es. una variabile booleana `isLoading`). Queste variabili possono essere "osservate" nei widget utilizzando `Consumer`, con l'accortezza di fornire il ViewModel T attraverso il metodo `Provider.of` impostando il parametro `listen` a `true`, per attivare la ricostruzione reattiva.

4.4.1 Notifica giornaliera

Una delle azioni fondamentali per sfruttare appieno le funzionalità dell'applicazione, è l'inserimento giornaliero dei log da parte dell'utente. Questa necessità dovrebbe invogliare l'utente a compiere abitudinalmente tutte le azioni necessarie per registrare questi dati, compreso lo svolgimento di alcuni esercizi.

Uno studio dimostra che è attraverso la trasformazione dei comportamenti salutari in abitudini che si può ottenere uno stile di vita sano [19].

Per ricordare all'utente l'inserimento dei log giornalieri (circonferenza della vita, peso, cibo ecc.) ho configurato un processo in background che invia una notifica push locale ogni giorno intorno alle ore 14:00. Questa notifica mostra all'utente un messaggio che gli ricorda per quali metriche non ha ancora inviato il suo log nella giornata corrente.

Per implementare questa funzionalità, ho utilizzato il pacchetto `workmanager`, che funge da wrapper Flutter attorno alle API native di background processing di Android (`WorkManager`) e iOS/macOS (`Background Tasks`) [17].

Il funzionamento è molto semplice e si distingue in due passaggi:

1. Si istanzia un oggetto di tipo `Workmanager` e lo si inizializza passandogli come parametro una funzione di *entry point* chiamata `callbackDispatcher`, che è definita al di fuori del `main` dell'applicazione. Questa funzione si occupa di eseguire la logica del task, che nel nostro caso consiste nel recuperare lo stato delle metriche mancanti e, se necessario, inviare la notifica locale all'utente.

2. Sull'oggetto `Workmanager` viene chiamata una funzione chiamata `registerPeriodTask` che si occupa di schedulare il task con una cadenza a nostro piacere. In particolare ho impostato i seguenti parametri:

   - `frequency: const Duration(hours: 24)`, //il task verrà eseguito ogni 24 ore
   - `initialDelay: next14oclockDelay()`, //il task appena creato verrà eseguito con un delay iniziale pari al tempo che intercorre tra l'istante attuale e le 14:00

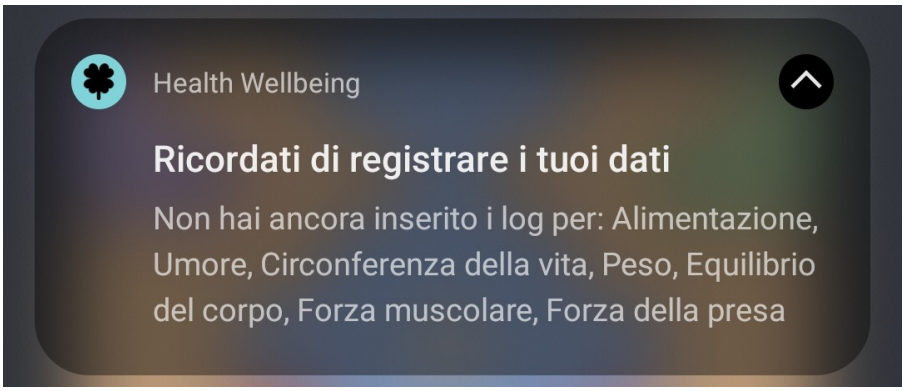


Figura 4.1. Esempio di notifica giornaliera.

Capitolo 5

Sicurezza

La sicurezza è un fattore importantissimo in qualsiasi applicazione.

Così come avviene nelle applicazioni web, costituite nella loro forma più semplice da un client e un server, sappiamo per certo che non basta implementare funzioni di validazione lato client per impedire all'utente di inserire valori non consentiti prima di inviarli al server. Questo perché, la richiesta, potrebbe essere manipolata aggirando la nostra applicazione client. Per questo motivo, la validazione dei dati, viene affidata in maniera particolare al back-end, prima che essi vengano processati e salvati. A supporto dell'approccio proposto vi è uno dei principi di sicurezza dell'OWASP (Requisito V3.1) nell'ASVS [30], che afferma la non affidabilità di ogni tipo di dato generato dal client.

Avendo a disposizione un piano gratuito di Firebase, vi sono dei limiti nella definizione di questi vincoli, per questo motivo lascio questo importantissimo requisito strutturale da implementare nei piani futuri, prima del rilascio dell'applicazione ufficiale.

5.1 Regole di Sicurezza

Quello che possiamo fare, però, è sfruttare quantomeno le **Firebase Security Rules**, per definire l'autorizzazione di lettura e scrittura dei vari utenti. Questo strumento permette anche di definire forme base di validazione dei dati in entrata, non paragonabili a quelle offerte da un backend completo [12].

Per quanto riguarda l'autenticazione invece, mi sono affidato completamente al solido **Firebase Authentication**, che fornisce un sistema sicuro e gestito da Google. Non mi soffermerò quindi su questo punto.

Le regole di Firebase definite sono: 5.1

Analisi delle Regole Come è possibile notare dalle regole sopra, questo strumento sfrutta l'oggetto `request.auth` per identificare l'utente.

- Ho fatto in modo che i dati relativi a un utente possano essere letti e scritti solo da quell'utente, escludendo l'admin, attraverso la condizione `request.auth != null` (autenticazione necessaria) `&& request.auth.uid == userId` (l'id dell'utente autenticato deve coincidere con `userId`, che è l'id del documento[1]).
- Invece, i dati generici dell'applicazione (`food_items`, `lessons` e `quizzes`) possono essere **letti da chiunque** ma **scritti solo da un admin**.
- Tutti i documenti restanti, invece, possono essere letti e scritti solo dagli admin.

Regola `fitbit_data` Interessante è la regola relativa alla sottocollezione `fitbit_data`: oltre ad essere una sottocollezione è contemporaneamente, come affermato in passato, un **gruppo di raccolta** per supportare query complesse. Non inserendo questa regola, gli utenti non potrebbero eseguire queste query, perché dovrebbero avere accesso anche ai documenti degli altri utenti, possibilità che abbiamo per ovvie ragioni negato. La regola si basa sul verificare che il valore del campo `user_id` dei documenti, che fanno parte di quel gruppo di raccolta, coincida con quello dell'utente autenticato.

5.2 Sicurezza Lato Client (Bearer Token)

Un'altra forma di supporto alla sicurezza, presente questa volta nel client Flutter, è l'inclusione, nelle richieste indirizzate al server dedicato alle Recommendation, del **bearer token Firebase** dell'utente autenticato. In questo modo, il server in questione può fornire una risposta valida solo agli utenti autorizzati.

[1]Durante la progettazione dello schema del database ho fatto sì che, per semplicità e per evitare dati ridondanti, gli id dei documenti appartenenti a un utente coincidessero con l'id dell'utente stesso.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users_data/{userId} {
      // Accesso privato: Solo l'utente con uid corrispondente puo' leggere/scrivere
      allow read, write: if request.auth != null && request.auth.uid == userId;
    }
    match /sync_data/{userId} {
      // Accesso privato per i dati di sincronizzazione
      allow read, write: if request.auth != null && request.auth.uid == userId;
      match /{document=**} {
        allow read, write: if request.auth != null && request.auth.uid == userId;
      }
    }
    match /{path=**}/fitbit_data/{docId} {
      // Regola per i gruppi di raccolta (collection groups)
      allow read: if request.auth != null &&
                resource.data.user_id == request.auth.uid;
      allow write: if request.auth != null &&
                 request.resource.data.user_id == request.auth.uid &&
                 resource.data.user_id == request.auth.uid;
    }
    match /food_items/{itemId} {
      // Dati generici: Lettura a tutti gli utenti, Scrittura solo admin
      allow read: if request.auth != null;
      allow write: if request.auth != null && request.auth.token.admin == true;
    }
    match /lessons/{lessonId} {
      // Dati generici: Lettura a tutti gli utenti, Scrittura solo admin
      allow read: if request.auth != null;
      allow write: if request.auth != null && request.auth.token.admin == true;
    }
    match /quizzes/{quizId} {
      // Dati generici: Lettura a tutti gli utenti, Scrittura solo admin
      allow read: if request.auth != null;
      allow write: if request.auth != null && request.auth.token.admin == true;
    }
    match /{document=**} {
      // Regola di default: Accesso solo admin per tutti gli altri documenti
      allow read, write: if request.auth != null && request.auth.token.admin == true;
    }
  }
}
```

Listing 5.1. Regole di Sicurezza Firebase Firestore

Capitolo 6

Conclusione e Piani Futuri

6.1 Conclusione

La tesi ha conseguito gli obiettivi prefissati, contribuendo all'evoluzione positiva della precedente applicazione per il benessere, offrendo **interfacce più accessibili** e **user flow più intuitivi** a quelle che erano le funzionalità principali dell'applicazione, utilizzando le euristiche di Jakob Nielsen e ispirandosi all'estetica dei prodotti più validi presenti sul mercato.

L'applicazione è stata riscritta da zero adottando il pattern **MVVM** e la **Dependency Injection** per garantire uno sviluppo e un mantenimento futuro più agevole.

Inoltre sono state apportate modifiche alla metodologia di salvataggio dei dati biometrici degli utenti, supportando al meglio la loro elaborazione da parte di applicazioni esterne, come il Recommender.

6.1.1 Criticità e Limitazioni Attuali

Dal punto di vista della **sicurezza** però, nonostante siano state definite delle regole di sicurezza nel back-end, al fine di evitare accessi non autorizzati ai dati sensibili degli utenti, sono necessari ulteriori sviluppi. Queste mancanze dal punto di vista della sicurezza, seppur sorvolate consapevolmente in questa fase di prototipizzazione, sarebbero invece criticità inaccettabili in fase di distribuzione al grande pubblico. Di questo discuterò più approfonditamente nella sezione dedicata ai piani futuri.

Altre limitazioni attuali sono invece dovute alla **dipendenza da dispositivi (es. smartwatch) e servizi (es. Fitbit) esterni** per l'acquisizione dei dati biometrici, la risorsa su cui si basa l'intera logica dell'applicazione. Il non totale controllo di questa risorsa ha portato criticità su tre fronti diversi:

- **Performance e ottimizzazione**: come visto nel caso di Fitbit, il processo di sincronizzazione è poco efficiente perché i dati non vengono recuperati direttamente dal dispositivo, ma sono richiesti ad un server privato, la cui latenza non è sotto il nostro controllo.

- **Complessità del codice e manutenibilità**: il codice diventa enormemente complesso man mano che le diverse implementazioni vengono inserite nel progetto, anche dal punto di vista della sincronizzazione dei dati. Ad esempio la gestione delle limitazioni di frequenza (*rate limits*) imposte dalle API Fitbit, ha reso il codice enormemente più complesso.

- **Dipendenza da aziende terze**: idealmente bisognerebbe sviluppare un'interfaccia per ciascuna azienda, generando alti rischi futuri. Infatti, non è purtroppo garantito che ciascuna di esse fornisca delle API pubbliche, che siano gratuite e che lo faccia anche in futuro. In più, da un momento all'altro, nelle API di terze parti potrebbe esserci un *breaking change* (modifica non retrocompatibile) che causerebbe un costo di manutenzione imprevedibile e non trascurabile a lungo termine, considerando fra l'altro che le aziende produttrici in questione sono molteplici.

Dal mio punto di vista, la soluzione è **rendere l'acquisizione di questi dati indipendente da infrastrutture esterne**. Una possibilità è creare dei supporti software su modelli mirati di smartwatch esistenti in commercio, oppure, in un'ottica di massima indipendenza, producendo uno smartwatch *ad-hoc*. Potremmo evitare soluzioni costose come quelle già citate, se la maggior parte dei produttori di smartwatch supportasse pienamente gli aggregatori `Health Connect/Apple Health`.

6.2 Piani Futuri

Per quanto riguarda il futuro, il progetto necessita di una evoluzione per permettere la distribuzione al grande pubblico.

Tra le necessità vi sono:

1. **La creazione e l'impiego di un server back-end** che si sostituisca alla complessità presente nel client, aumentando contemporaneamente la sicurezza e la consistenza dell'applicazione. Potrebbe essere, inoltre, una soluzione a diversi problemi o limitazioni incontrati in fase di sviluppo:

    - La già citata mancanza di **validazione dei dati lato server**. Risolvibile sfruttando la piena libertà offerta da questa soluzione nella scrittura

della logica di business, rispetto a quella ristretta messa a disposizione dalla versione gratuita di Firebase.

- **Sincronizzazione fragile dei dati biometrici dell'utente**: la possibilità di effettuare la sincronizzazione in questo livello, piuttosto che delegarla al client, permetterebbe di avere totale controllo sulla gestione dei timestamp. Diversamente, affidarsi all'orologio dell'OS del dispositivo utente, potrebbe potenzialmente causare inconsistenza dei dati. Per esempio, potrebbe banalmente capitare nel caso in cui il sistema operativo del client abbia impostato un orario errato o che quest'ultimo sia stato manomesso da un malintenzionato.
- **Autenticazione ai servizi esterni (attualmente solo Fitbit) completamente gestita dal client**. Questa condizione comporta la presenza in chiaro, nei binari dell'applicazione, della chiave segreta di accesso a Fitbit. Si rende necessario quindi un server intermediario che gestisca lo scambio dei token del protocollo OAuth con il servizio in questione. Considero questa modifica essenziale prima che il software entri nella fase di produzione.

2. **Definizione di una solida catena di CI/CD** (*Continuous Integration* e *Continuous Deployment*) per automatizzare, semplificare e accelerare il ciclo di sviluppo del software. A tal proposito, potrebbe essere utile impiegare uno strumento come **Github Actions** [20] che, in seguito alla configurazione di un file `yaml`, permette di eseguire qualsiasi operazione in maniera automatica, come test, build e deploy, sulle varie piattaforme. Importante da citare, nel particolare caso di Flutter, è **Shorebird**, uno strumento che, fra le altre cose, oltre a configurare automaticamente la pipeline analizzando il progetto, permette di effettuare correzioni istantanee direttamente sui dispositivi degli utenti via OTA (*over-the-air*), superando i limiti imposti dalla fase di revisione dei vari store [31].

Bibliografia

[1] Cachet.dk. *health*. Wrapper per HealthKit su iOS e Health Connect su Android. Consente la lettura e scrittura di dati sanitari come passi, peso, pressione sanguigna e altro. Licenza MIT. 2025. URL: https://pub.dev/packages/health.

[2] Giacomo Cappon. *fitbitter*. Pacchetto Flutter per interagire con le API di Fitbit. Supporta piattaforme Android, iOS, Linux, macOS, web e Windows. Licenza BSD-3-Clause. 2025. URL: https://pub.dev/packages/fitbitter.

[3] Mary A. Carskadon e William C. Dement. "Monitoring and Staging Human Sleep". In: *Principles and Practice of Sleep Medicine*. A cura di Meir H. Kryger, Thomas Roth e William C. Dement. 6th. Elsevier, 2017, pp. 15–26.

[4] Casa Walden. *Brand Identity: l'importanza del branding e del logo*. Compito del logo è quello di rappresentare immediatamente e in modo distintivo l'azienda, ispirando fiducia e distinzione rispetto alla concorrenza: in altri termini, il logo è portatore degli elementi di identità – unicità – riconoscibilità del brand. 2023. URL: https://casawalden.com/brand-identity-limportanza-del-branding-e-del-logo/ (visitato il giorno 24/10/2025).

[5] dash-overflow.net. *Provider package*. Pacchetto ufficiale per la gestione dello stato in Flutter. Fornisce Provider, ChangeNotifierProvider, MultiProvider, Consumer e strumenti per implementare facilmente il pattern MVVM e il pattern Observer. 2025. URL: https://pub.dev/packages/provider (visitato il giorno 24/10/2025).

[6] Edsger W. Dijkstra. "On the Role of Scientific Thought". In: *Selected Writings on Computing: A Personal Perspective*. Il principio di Separation of Concerns (SoC) sottolinea la necessità di suddividere un sistema in parti distinte e indipendenti per migliorarne la comprensione, la manutenibilità e la scalabilità. Springer, 1982, pp. 60–66.

[7] Figma. *Guide to components in Figma*. Documentazione ufficiale che descrive i componenti come elementi riutilizzabili per garantire coerenza visiva e facilità di aggiornamento globale. Figma Inc. 2025. URL: https://help.figma.com/hc/en-us/articles/360038662654-Guide-to-components-in-Figma (visitato il giorno 24/10/2025).

[8] Firebase. *Firebase - Build and run successful apps*. 2025. URL: https://firebase.google.com/ (visitato il giorno 23/10/2025).

[9] Google Firebase. *WriteBatch | Firebase SDKs for Android*. Documentazione ufficiale di Firebase per la classe WriteBatch, utilizzata per eseguire operazioni di scrittura atomiche su Firestore. 2025. URL: https://firebase.google.com/docs/reference/android/com/google/firebase/firestore/WriteBatch.

[10] Firebase Docs. *Cloud Firestore Documentation*. Documentazione ufficiale di Cloud Firestore che descrive la struttura NoSQL, le collections, i documenti, i campi, le subcollections e le performance. 2025. URL: https://firebase.google.com/docs/firestore (visitato il giorno 24/10/2025).

[11] Firebase Docs. *Querying Data in Cloud Firestore*. Descrive l'uso degli indici singoli e compositi in Firestore per ottimizzare le query. 2025. URL: https://firebase.google.com/docs/firestore/query-data/queries?hl=it#compound_and_queries (visitato il giorno 24/10/2025).

[12] Firebase Documentation. *Firebase Security Rules*. Definizione delle autorizzazioni di lettura e scrittura per utenti e admin. 2025. URL: https://firebase.google.com/docs/rules (visitato il giorno 24/10/2025).

[13] Fitbit. *Web API Reference*. Documentazione ufficiale delle API Web di Fitbit, con indicazioni sui limiti di intervallo per il recupero dei dati. 2023. URL: https://dev.fitbit.com/build/reference/web-api (visitato il giorno 24/10/2025).

[14] Flutter. *Flutter - Build apps for any screen*. 2025. URL: https://flutter.dev/ (visitato il giorno 23/10/2025).

[15] Flutter Devs. *get_it: Service Locator for Flutter*. Pacchetto Flutter che implementa il Service Locator Pattern per gestire le dipendenze e favorire l'Inversion of Control. 2025. URL: https://pub.dev/packages/get_it (visitato il giorno 24/10/2025).

[16] flutter.cn. *dio | Dart package*. 2025. URL: https://pub.dev/packages/dio (visitato il giorno 24/10/2025).

[17] fluttercommunity.dev. *workmanager | Flutter package*. Ver. 0.9.0+3. 2025. URL: https://pub.dev/packages/workmanager.

[18] Alessandro Frangioni. *Linee Guida per Un Buon Flat Design*. Set. 2014. URL: https://medium.com/provami/linee-guida-per-un-buon-flat-design-5a692c9b65ff (visitato il giorno 23/10/2025).

[19] Benjamin Gardner, P. Lally e J. Wardle. "Making Health Habitual: The Psychology of "Habit-Formation" and General Practice". In: *British Journal of General Practice* 62.605 (dic. 2012), pp. 664–666. DOI: 10.3399/bjgp12x659466.

[20] Inc. GitHub. *GitHub Actions Documentation*. Documentazione ufficiale di GitHub Actions, per la configurazione di workflow automatizzati. 2025. URL: https://docs.github.com/en/actions (visitato il giorno 24/10/2025).

[21] Global Wellness Institute. *2024 Global Wellness Economy Monitor*. Research Report. Global Wellness Institute, 2024. URL: https://globalwellnessinstitorg/industry-research/2024-global-wellness-economy-monitor/.

[22] GWI. *Health & Wellbeing Report: Lifestyles, Fitness & Health Brands*. 2019. URL: https://www.gwi.com/reports/health-and-wellbeing (visitato il giorno 23/10/2025).

[23] JohnFitbit. *API limits for huge data requirements*. Discussione ufficiale nella community di Fitbit riguardante i limiti delle API per grandi volumi di dati. 2024. URL: https://community.fitbit.com/t5/Web-API-Development/API-limits-for-huge-data-requirements/td-p/5598375#:~:text=The%20rate%20limit%20is%20150,per%20hour%2C%20not%20per%20client.

[24] Mikyung Lee et al. "Mobile App-Based Health Promotion Programs: A Systematic Review of the Literature". In: *International Journal of Environmental Research and Public Health* 15.12 (dic. 2018), p. 2838. DOI: 10.3390/ijerph15122838. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6313530/.

[25] LogRocket. *State management in Flutter using the BLoC design pattern*. Descrive il flusso in tre fasi dell'architettura BLoC: invio di eventi dalla UI, elaborazione nel BLoC e aggiornamento dello stato nella UI tramite stream. 2025. URL: https://blog.logrocket.com/state-management-flutter-bloc-pattern/ (visitato il giorno 24/10/2025).

[26] Microsoft. *Data binding e MVVM*. Descrive come il data binding consenta di sincronizzare automaticamente i dati tra il ViewModel e la View, migliorando la separazione delle preoccupazioni e la manutenibilità del codice. 2023. URL: https://learn.microsoft.com/it-it/windows/uwp/data-binding/data-binding-and-mvvm (visitato il giorno 24/10/2025).

[27] Gleb Morgachev. *Why does React Native have performance issues?* Analisi delle problematiche del "bridge" in React Native: il layer di comunicazione tra JS e moduli nativi può introdurre latenza e colli di bottiglia. Product Science AI. 2022. URL: https://medium.com/product-science-ai/why-does-react-native-have-performance-issues-22494d3447ca (visitato il giorno 24/10/2025).

[28] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. Nielsen Norman Group. 1994. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (visitato il giorno 24/10/2025).

[29] Ninja Marketing. *Utili consigli per sviluppare la strategia di onboarding mobile*. Articolo che descrive l'importanza dell'onboarding mobile e segnala che oltre l'80% delle applicazioni viene eliminata dopo il primo impiego. Ninja Marketing. 2017. URL: https://www.ninja.it/utili-consigli-per-sviluppare-strategia-onboarding-mobile/ (visitato il giorno 24/10/2025).

[30] OWASP Foundation. *OWASP Application Security Verification Standard (ASVS) 4.0.3*. V5.1 Input Validation. 2019. URL: https://owasp.org/www-project-application-security-verification-standard/ (visitato il giorno 24/10/2025).

[31] Inc. Shorebird. *Shorebird Documentation*. Documentazione ufficiale di Shorebird, strumento per distribuzione OTA e gestione pipeline Flutter. 2025. URL: https://docs.shorebird.dev/ (visitato il giorno 24/10/2025).

[32] Nidhi Sorathiya e Ashok Sisara. *Exploring the Capabilities of Flutter Reflectable*. Descrive come il pacchetto Flutter Reflectable consenta l'uso della riflessione in tempo di esecuzione attraverso la generazione di codice, superando le limitazioni di performance e dimensione tipiche delle applicazioni mobili. 2025. URL: https://www.dhiwise.com/post/exploring-the-capabilities-of-flutter-reflectable (visitato il giorno 24/10/2025).

[33] Muhammad Sufiyan. *MVVM: Officially Recommended by Google for Flutter Development*. Articolo che discute l'adozione ufficiale dell'architettura MVVM da parte di Google per lo sviluppo di applicazioni Flutter,

evidenziandone i vantaggi in termini di manutenibilità e separazione delle responsabilità. 2024. URL: https://medium.com/@ksufi7350/mvvm-officially-recommended-by-google-for-flutter-development-ba17f899d320 (visitato il giorno 24/10/2025).