

UNIVERSITÀ POLITECNICA DELLE MARCHE
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Sviluppo di Inkspire
Applicazione per generare sfide artistiche



Corso di
PROGRAMMAZIONE MOBILE
Anno accademico 2024-2025

Studente:
Alessia Capancioni

Professore:
Emanuele Storti



Dipartimento di Ingegneria dell'Informazione

INDICE

1. Introduzione	4
1.1. Descrizione in linguaggio naturale	4
1.2. Obiettivo del progetto	5
2. Progettazione e Sviluppo Android	6
2.1. Sintesi dell'approccio	6
2.2. Glossario dei termini	7
2.3. Requisiti	8
2.3.1. Requisiti funzionali	8
2.3.2. Requisiti non funzionali	9
2.3.3. Casi d'uso	10
2.4. Mockup UI	16
2.4.1. Accesso e uscita dall'app	17
2.4.2. Gestione Challenge	18
2.4.3. Profilazione	21
2.4.4. Community	22
2.5. Database e memorizzazione dati	23
2.5.1. Modello relazionale	23
2.5.2. Struttura delle tabelle in Supabase	24
2.5.3. Viste per aggregazione dati	27
2.5.4. Autenticazione con Supabase Auth	29
2.5.5. Policies di sicurezza	29
2.5.6. Supabase Storage	30
2.6. Architettura	31
2.6.1. MVVM Pattern	31
2.6.2. Repository Pattern	32
2.7. Sviluppo	33
2.7.1. Panoramica dello sviluppo	33
2.7.2. Struttura del progetto	34
2.7.3. Layout e UI	37
2.7.4. Autenticazione, Storage e accesso al DB	40
2.7.5. Model	42
2.7.6. Repository	43
2.7.7. ViewModel	45

2.7.8. Adapter	46
2.7.9. Activity e Fragment	47
2.7.10. Navigazione	50
2.8. Testing	51
2.8.1. Local Unit Tests	51
2.8.2. Instrumented Tests	55
3. Progettazione e Sviluppo Flutter	60
3.1. Sviluppo Flutter e differenze con quello Android	60
3.2. Requisiti	61
3.3. Mockup UI	62
3.4. Database e memorizzazione dati	66
3.4.1. Modello relazionale	66
3.4.2. Pacchetto supabase_flutter	66
3.4.3. Policies di sicurezza	66
3.5. Sviluppo	67
3.5.1. Architettura e panoramica	67
3.5.2. Classi Widget in Flutter	69
3.5.3. Struttura del progetto	71
3.5.4. Layout e UI	73
3.5.5. Autenticazione, Storage e accesso al DB	75
3.5.6. Model	77
3.5.7. Repository	78
3.5.8. Provider	79
3.5.9. Routing e Navigazione	80
3.5.10. Gestione cache e refresh	82
3.5.11. Schermate	84
4. Problematiche e considerazioni finali	85
Bibliografia	87

1. Introduzione

1.1. Descrizione in linguaggio naturale

Nel contesto dell'arte digitale e tradizionale, la motivazione e l'ispirazione giocano un ruolo fondamentale per ogni artista. Per quanto grande possa essere la passione, può diventare difficile trovare nuove idee che spingano a disegnare con costanza.

Inkspire nasce proprio come risposta a questa esigenza: una app che aiuta gli utenti a generare sfide artistiche creative e a condividerle con una community di artisti.

L'applicazione permette di creare una sfida (challenge) definendo un titolo, un tema (concept) e un vincolo artistico (art constraint). Questi ultimi due attributi possono essere scelti liberamente o generati casualmente. In particolare, il vincolo artistico è una caratteristica speciale della sfida che impone una limitazione (ad esempio l'uso di soli tre colori o un limite temporale), aggiungendo un livello di difficoltà che spinge l'utente a trovare soluzioni innovative. Ogni sfida può essere arricchita con una descrizione e un'immagine del lavoro artistico realizzato, sia durante la creazione che in un secondo momento. Oltre a creare nuove sfide da zero, Inkspire permette anche di replicarne una già esistente: si può generare una propria variante personale partendo da titolo, tema e vincolo artistico della sfida di un altro utente, mantenendo un riferimento ad esso. L'app comprende inoltre un profilo utente che raccoglie tutte le sfide create e completate, offrendo un modo pratico per conservare i propri lavori e osservare i progressi e i cambiamenti artistici nel tempo.

Inkspire è quindi pensata per offrire uno spazio creativo dove gli utenti possano ispirarsi e migliorare le proprie competenze attraverso l'utilizzo di concetti e vincoli originali e la condivisione dei propri lavori.

1.2. Obiettivo del progetto

L'obiettivo di questo progetto è la progettazione e lo sviluppo di Inkspire, sia in ambiente Android che con Flutter, per assicurarne la flessibilità e l'adattamento a più piattaforme.

L'applicazione deve consentire agli utenti di:

- generare sfide artistiche personalizzate o casuali;
- replicare una sfida già esistente mantenendo il collegamento con l'autore originale;
- completare le sfide caricando un'immagine e una descrizione facoltativa;
- consultare le sfide della community;
- gestire il proprio profilo e visualizzare le proprie attività artistiche.

Tutto questo cercando di garantire un'interfaccia intuitiva per un utilizzo semplice e immediato di tutte le funzionalità.

2. Progettazione e Sviluppo Android

2.1. Sintesi dell'approccio

La progettazione di Inkspire si è articolata in due fasi principali: una prima fase concettuale e una successiva fase implementativa.

Durante la fase concettuale sono stati identificati:

- gli obiettivi funzionali;
- i principali casi d'uso;
- la struttura dei dati e il modello relazionale;
- i flussi di navigazione tra le schermate principali.

Questa fase ha permesso di definire una base solida e chiara, utile a guidare lo sviluppo successivo. La fase implementativa ha invece previsto:

- l'adozione di un'architettura MVVM per separare la logica di business dall'interfaccia grafica;
- l'utilizzo di Supabase come backend per la gestione di autenticazione, database PostgreSQL e storage delle immagini;
- l'integrazione di Ktor per la comunicazione REST con il backend;
- la progettazione di tutti i layout xml e dei temi personalizzati;
- l'organizzazione del progetto in package tematici (model, repository, viewmodel, fragments, adapter);
- l'implementazione della navigazione e del passaggio di parametri tra i fragment con Navigation Component e Safe Args.

L'approccio è stato incrementale e iterativo, alternando momenti di progettazione teorica a cicli di sviluppo pratico e verifica continua del corretto funzionamento delle funzionalità.

2.2. Glossario dei termini

Termine	Descrizione	Sinonimo
Challenge	Sfida artistica creata da un utente, composta da titolo, tema, vincolo artistico e contenuti opzionali (descrizione e immagine). Una challenge è completa solo se ha associata un'immagine.	Sfida artistica
Concept	Tema ispiratore di una sfida, può essere generato casualmente o definito dall'utente (es. viaggio nel tempo, metamorfosi).	Tema, Concetto della sfida
Art Constraint	Vincolo artistico che introduce una limitazione creativa, può essere generato casualmente o definito dall'utente (es. solo tre colori, divieto di linee curve).	Vincolo artistico, Restrizione
User Profile	Profilo di un utente registrato, contenente username, foto profilo, breve descrizione (bio), statistiche ed elenco di tutte le challenge create.	Profilo utente
Community	Insieme di tutti gli utenti registrati che utilizzano l'app.	Insieme degli Utenti/Artisti.

2.3. Requisiti

2.3.1. Requisiti funzionali

- **RF1. Registrazione e Login:** l'utente deve poter registrarsi immettendo username, email e password e successivamente autenticarsi tramite email e password.
- **RF2. Visualizzazione elenco challenge:** l'utente deve poter consultare l'elenco delle challenge della community e dei singoli profili.
- **RF3. Creazione di una nuova challenge:** l'utente deve poter creare una nuova sfida inserendo gli attributi obbligatori titolo, tema e vincolo artistico.
- **RF3.1. Generazione casuale di tema e vincolo artistico:** l'utente deve poter generare casualmente il tema e il vincolo artistico della sfida tramite dei pulsanti dedicati.
- **RF3.2 Inserimento descrizione e immagine:** l'utente deve poter aggiungere una descrizione facoltativa e caricare un'immagine del lavoro artistico realizzato, al fine di rendere la challenge completa.
- **RF3.3 Fork di una challenge:** l'utente deve poter replicare una challenge creata da un altro utente, con titolo, tema e vincolo già compilati, e mantenere un riferimento all'originale.
- **RF4. Visualizzazione dettagli challenge:** l'utente deve poter consultare la schermata di dettaglio di una challenge con tutte le informazioni relative ad essa e, se non appartiene a lui, all'utente che l'ha creata.
- **RF5. Modifica challenge:** l'utente deve poter modificare una challenge creata in precedenza aggiornando titolo, tema, vincolo, descrizione e immagine.
- **RF6. Eliminazione challenge:** l'utente deve poter eliminare una challenge creata in precedenza.
- **RF7. Visualizzazione profilo utente:** l'utente deve poter visualizzare il proprio profilo, comprendente username, bio, statistiche ed elenco delle challenge create e completate.
- **RF8. Modifica profilo utente:** l'utente deve poter modificare la propria biografia e l'immagine del profilo.
- **RF9. Visualizzazione profili degli altri utenti:** l'utente deve poter consultare il profilo di un altro utente della community.
- **RF10. Logout:** l'utente deve poter disconnettersi dal proprio account.

2.3.2. Requisiti non funzionali

- **RNF1. UI:** l'interfaccia deve essere semplice e intuitiva, per garantire una buona esperienza utente.
- **RNF2. Sviluppo:** il linguaggio di sviluppo è Kotlin.
- **RNF3. Architettura:** l'architettura dell'app utilizza il pattern MVVM (Model-View-ViewModel).
- **RNF4. Persistenza dati online:** tutte le informazioni (utenti, sfide, immagini) sono salvate e sincronizzate online tramite Supabase (PostgreSQL + Storage + Auth).
- **RNF5. Visualizzazione immagini:** il caricamento e la visualizzazione delle immagini sono gestiti tramite la libreria Glide.
- **RNF6. Compatibilità:** l'app è compatibile con Android 10 (API 29) e versioni successive.
- **RNF7. Sicurezza:** le operazioni di autenticazione devono utilizzare protocolli sicuri (HTTPS) e token JWT gestiti tramite Supabase Auth.

2.3.3. Casi d'uso

Per descrivere le funzionalità principali che il sistema deve offrire e le sue risposte in base alle azioni compiute dall'utente, viene riportato di seguito l'elenco dei casi d'uso, il quale costituisce una base per la progettazione dell'architettura, la realizzazione dell'interfaccia e la scrittura dei test di verifica dell'app.

I casi d'uso sono divisi in due macro aree: Gestione Utente e Gestione Challenge:

Gestione Utente: questi casi d'uso riguardano la gestione dell'autenticazione e del profilo personale dell'utente.

- **UC1. Registrazione utente**

Si verifica quando l'utente vuole creare un nuovo account.

Pre-condizioni: non deve esistere un altro account utente con lo stesso indirizzo email o lo stesso username e la password deve essere valida (almeno 6 caratteri).

Post-condizioni: l'utente è registrato nel sistema e può effettuare il login.

Sequenza di eventi principali:

1. l'utente avvia la registrazione dall'apposita schermata;
2. il sistema permette di inserire email, username e password;
3. l'utente conferma;
4. il sistema crea il nuovo account e reindirizza l'utente alla schermata di login.

- **UC2. Login utente**

Si verifica quando l'utente vuole accedere all'account.

Pre-condizioni: l'utente è già registrato e inserisce le credenziali corrette.

Post-condizioni: l'utente accede all'app.

Sequenza di eventi principali:

1. l'utente accede alla schermata di login;
2. il sistema permette di inserire le credenziali (email e password);
3. l'utente conferma;
4. il sistema verifica i dati ed effettua l'accesso.

- **UC3. Logout utente**

Si verifica quando l'utente decide di uscire dal proprio account.

Pre-condizioni: l'utente è autenticato e si trova nella schermata del proprio Profilo.

Post-condizioni: l'utente viene disconnesso.

Sequenza di eventi principali:

1. l'utente seleziona il pulsante di logout dal menu del suo profilo;

2. il sistema chiede conferma e in caso affermativo esegue la disconnessione e riporta alla schermata di login.

- **UC4. Visualizzazione profilo personale**

Si verifica quando l'utente vuole visualizzare le informazioni del proprio profilo.

Pre-condizioni: l'utente è autenticato.

Post-condizioni: nessuna.

Sequenza di eventi principali:

1. l'utente accede alla sezione “Profile” del proprio profilo;
2. il sistema mostra la schermata del profilo personale con username, bio, immagine profilo, statistiche ed elenco delle challenge create dall'utente.

- **UC5. Modifica profilo**

Si verifica quando l'utente vuole modificare le informazioni del profilo.

Pre-condizioni: l'utente è autenticato e si trova nella schermata del proprio Profilo.

Post-condizioni: le informazioni vengono aggiornate.

Sequenza di eventi principali:

1. l'utente accede alla schermata di modifica del proprio profilo;
2. il sistema permette di modificare bio e immagine profilo;
3. l'utente salva;
4. il sistema aggiorna i dati e riporta alla schermata del profilo.

- **UC6. Visualizzazione profilo di un altro utente**

Si verifica quando l'utente vuole consultare il profilo pubblico di un altro utente.

Pre-condizioni: l'utente è autenticato e si trova o nella schermata di visualizzazione di una challenge di un altro utente, o nella sezione “Artists” contenente l'elenco di tutti gli utenti registrati.

Post-condizioni: nessuna.

Sequenza di eventi principali:

1. l'utente seleziona il nome di un altro utente;
2. il sistema visualizza in lettura il profilo di tale utente selezionato.

- **UC7. Visualizzazione elenco utenti**

Si verifica quando l'utente si trova nella schermata “Artists”.

Pre-condizioni: l'utente è autenticato.

Post-condizioni: nessuna.

Sequenza di eventi principali:

1. l'utente accede alla sezione "Artists";
2. il sistema mostra l'elenco di tutti gli utenti registrati per ordine di numero di challenge completate.

Gestione Challenge: questi casi d'uso riguardano la creazione, consultazione e gestione delle challenge

- **UC8. Visualizzazione elenco challenge**

Si verifica quando l'utente si trova nella schermata "Home" o in una schermata di Profilo, che sia il suo o di un altro utente.

Pre-condizioni: l'utente è autenticato.

Post-condizioni: nessuna.

Sequenza di eventi principali:

3. l'utente accede alla sezione "Home";
4. il sistema mostra l'elenco di tutte le challenge create dalla community.

- **UC9. Ricerca challenge**

Si verifica quando l'utente vuole filtrare la lista di tutte le challenge.

Pre-condizioni: l'utente si trova nella schermata "Home".

Post-condizioni: la lista viene filtrata.

Sequenza di eventi principali:

1. l'utente inserisce un termine di ricerca nella barra di ricerca del menu;
2. il sistema filtra le challenge in base al testo inserito.

- **UC10. Visualizzazione dettaglio challenge**

Si verifica quando l'utente seleziona una challenge dalla lista presente nella Home o in una schermata di Profilo per vederne i dettagli.

Pre-condizioni: la challenge deve essere presente nel sistema.

Post-condizioni: nessuna.

Sequenza di eventi principali:

1. l'utente seleziona una challenge dall'elenco;
2. il sistema mostra tutte le informazioni riguardanti quella challenge, quali: titolo, tema, vincolo, descrizione, immagine e autore.

- **UC11. Creazione challenge**

Si verifica quando l'utente vuole creare una nuova sfida.

Pre-condizioni: l'utente è autenticato e si trova in una delle schermate principali.

Post-condizioni: la challenge viene inserita nel sistema.

Sequenza di eventi principali:

1. l'utente preme il pulsante “+” del Fab;
2. il sistema mostra la form per la creazione di una nuova challenge;
3. l'utente inserisce titolo, tema, vincolo ed eventuali descrizione e immagine;
4. l'utente conferma;
5. il sistema salva la challenge.

- **UC12. Generazione concept/vincolo casuali**

Si verifica quando l'utente vuole generare un tema/vincolo casuale.

Pre-condizioni: l'utente si trova nel modulo di creazione o modifica di una challenge.

Post-condizioni: il campo “concept/art constraint” viene compilato automaticamente.

Sequenza di eventi principali:

1. l'utente preme l'icona di generazione casuale;
2. il sistema propone un tema/vincolo casuale.

- **UC13 Fork Challenge**

Si verifica quando l'utente vuole replicare una sfida della community.

Pre-condizioni: la challenge selezionata appartiene a un altro utente.

Post-condizioni: viene creata una nuova challenge che eredita titolo, tema e vincolo, ma senza descrizione né immagine, e con un riferimento che la collega all'originale.

Sequenza di eventi principali:

1. l'utente accede al dettaglio di una challenge non propria;
2. il sistema mostra l'opzione di fork della challenge;
3. l'utente preme il pulsante “Copia”;
4. il sistema apre la form di creazione già precompilata;
5. l'utente conferma o modifica i dati;
6. il sistema salva la nuova challenge come derivata dall'originale.

- **UC14 Modifica challenge**

Si verifica quando l'utente vuole modificare una challenge creata in precedenza.

Pre-condizioni: la challenge è stata creata dall'utente.

Post-condizioni: la challenge viene aggiornata.

Sequenza di eventi principali:

1. l'utente seleziona la propria challenge dal profilo o dall'elenco nella schermata Home;
2. il sistema mostra la form di modifica della challenge;

- 3. l'utente aggiorna i dati;
- 4. l'utente conferma;
- 5. il sistema salva le modifiche.

- **UC15. Eliminazione Challenge**

Si verifica quando l'utente vuole eliminare una challenge creata.

Pre-condizioni: la challenge è stata creata dall'utente.

Post-condizioni: la challenge viene rimossa.

Sequenza di eventi principali:

- 1. l'utente seleziona la propria challenge;
- 2. l'utente preme l'icona di eliminazione;
- 3. il sistema chiede conferma;
- 4. l'utente conferma;
- 5. il sistema elimina la challenge.

- **UC16. Completamento challenge**

Si verifica quando l'utente vuole aggiungere l'immagine del risultato alla challenge.

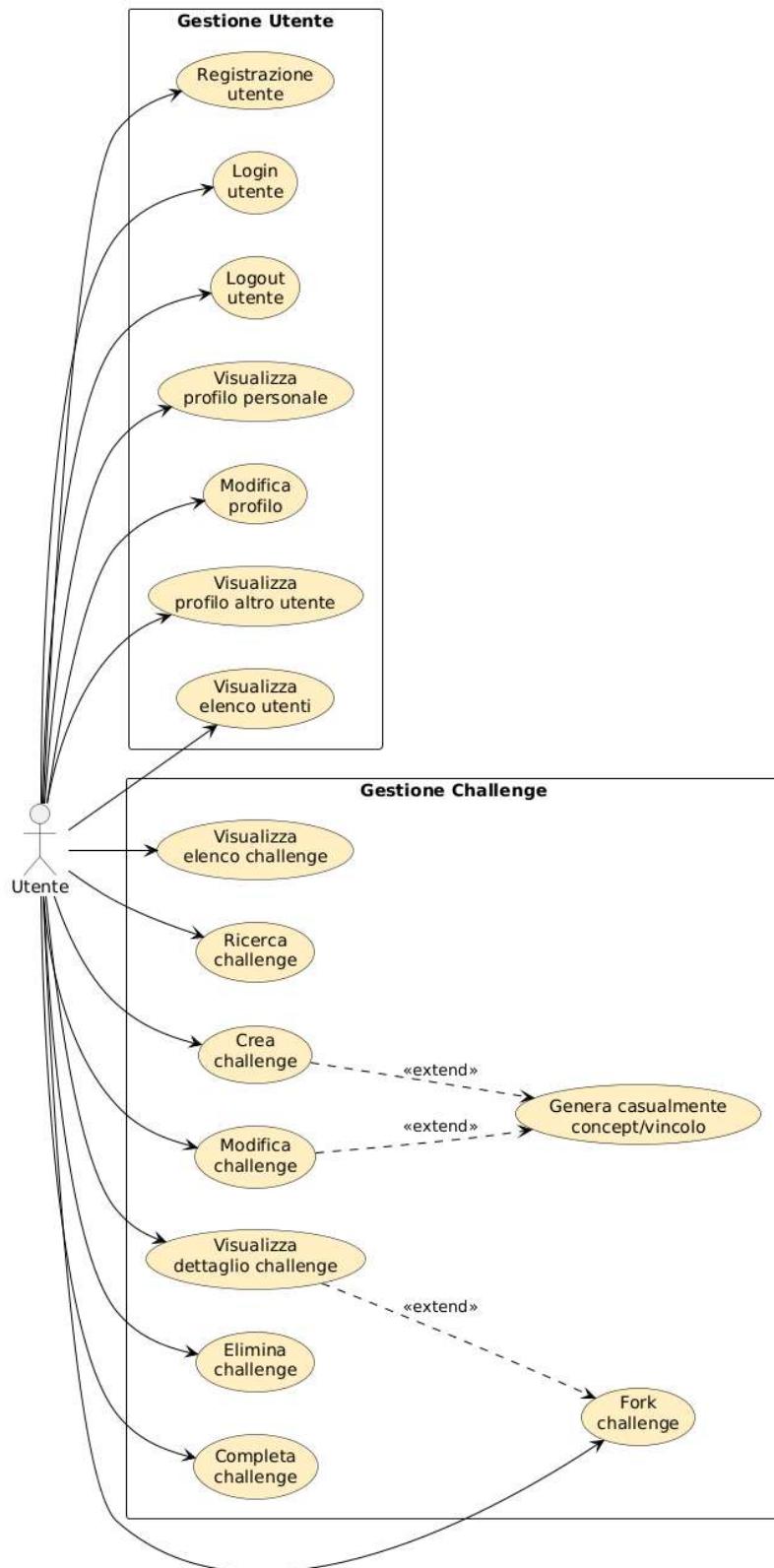
Pre-condizioni: la challenge esiste o, se in fase di creazione, presenta già tutti gli attributi obbligatori compilati.

Post-condizioni: la challenge presenta un'immagine e viene considerata completata.

Sequenza di eventi principali:

- 1. l'utente si trova nella forma di creazione o modifica di una sua challenge;
- 2. l'utente carica l'immagine del lavoro artistico;
- 3. l'utente conferma;
- 4. il sistema salva/aggiorna i dati.

Di seguito è riportato il diagramma UML dei casi d'uso, realizzato con l'editor online PlantUML:



2.4. Mockup UI

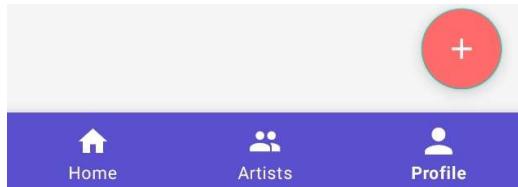
Di seguito sono riportate le immagini del mockup dell’interfaccia di Inkspire con la rappresentazione grafica dei casi d’uso.

Dopo aver effettuato la registrazione e successivamente il login, l’utente accede direttamente alla Home, la schermata principale dell’app, che mostra l’elenco di tutte le challenge create dagli utenti. Da qui può:

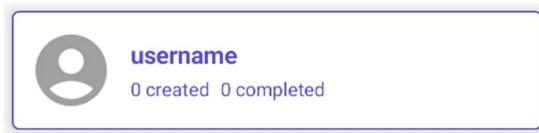
- visualizzare e filtrare tramite la barra di ricerca le challenge della community elencate nella sezione “Home”: cliccando sopra una card si entra nella schermata di visualizzazione dettaglio di quella challenge (View Challenge). Nel caso in cui la challenge sia propria, nella schermata di visualizzazione sarà presente anche un’icona con una matita per passare alla schermata di modifica (Edit Challenge);



- creare una nuova challenge cliccando sul pulsante “+” (Floating Action Button);



- visualizzare e filtrare tramite la barra di ricerca tutti gli utenti della community elencati nella sezione “Artists”: cliccando sopra una card si entra nella schermata di visualizzazione del profilo di quell’utente (Other User Profile);

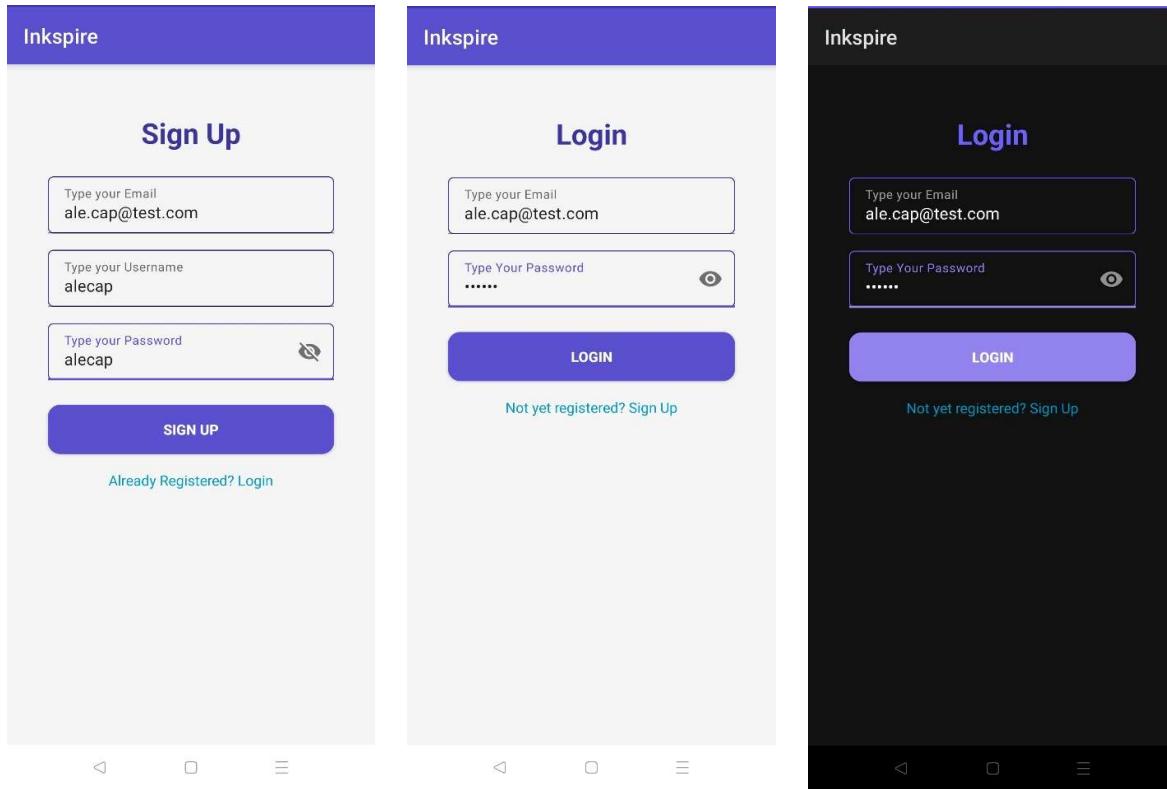


- visualizzare il proprio profilo personale navigando alla sezione “Profile”, dove sono mostrati username, foto profilo, bio, statistiche ed elenco delle proprie challenge e dove è possibile entrare nella schermata di modifica (Edit User Profile) per aggiornare foto profilo e bio.

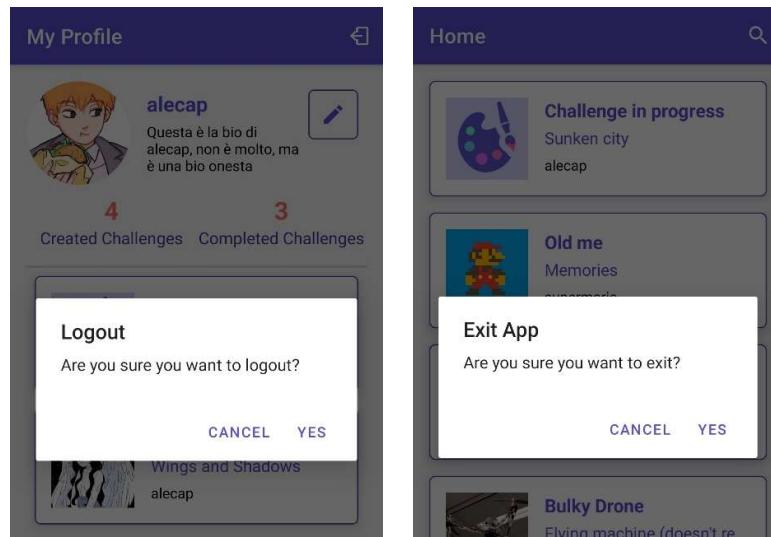
La navigazione avviene tramite la barra inferiore che consente di passare rapidamente alle sezioni principali, seguendo un flusso logico e intuitivo.

2.4.1. Accesso e uscita dall'app

Schermate di SignUp/Login



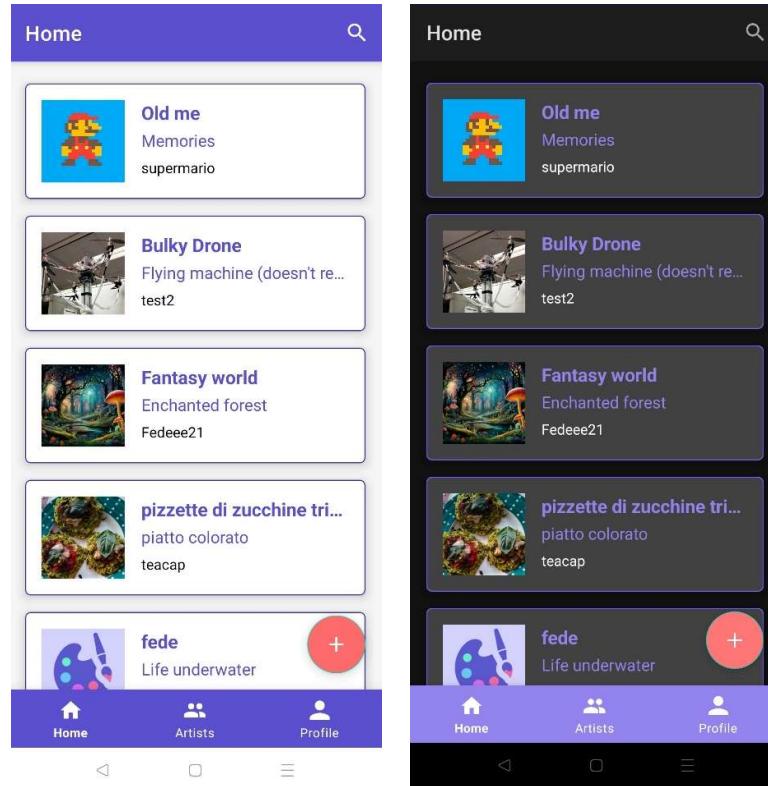
Finestre di dialogo



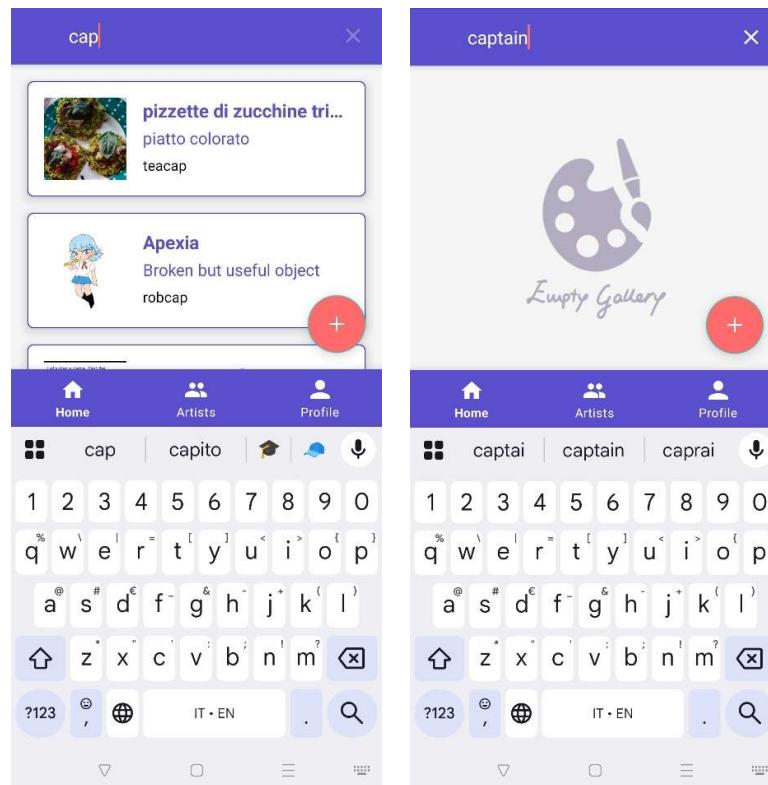
Il dialog di Logout compare appena l'utente clicca sul pulsante di logout nel menu del proprio profilo. Confermando, esso viene disconnesso e reindirizzato alla schermata di Login. Il dialog di Exit compare appena l'utente clicca il tasto “back” dalla schermata Home. Confermando esso esce dall'app senza disconnessione.

2.4.2. Gestione Challenge

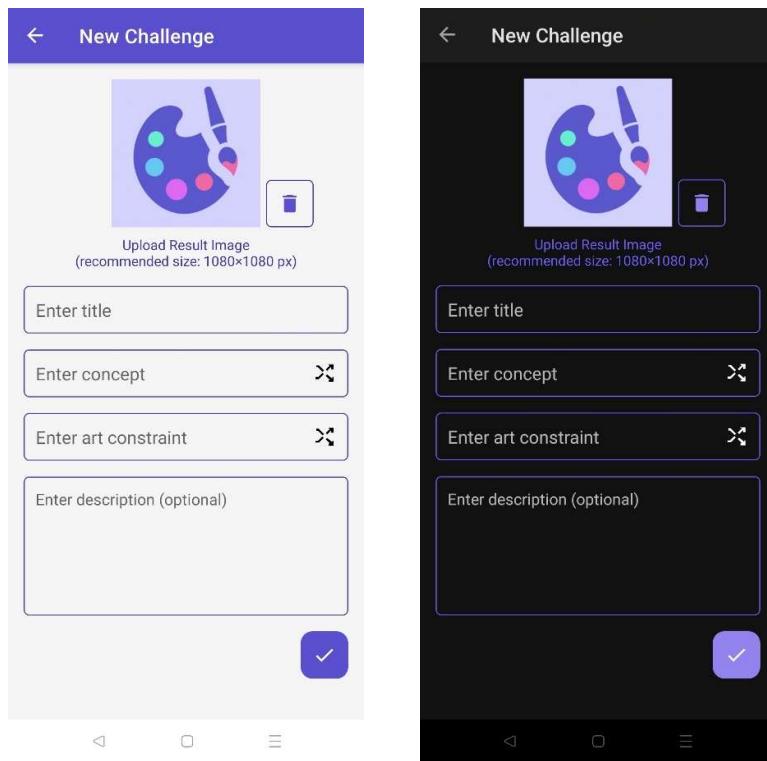
- Schermata Home



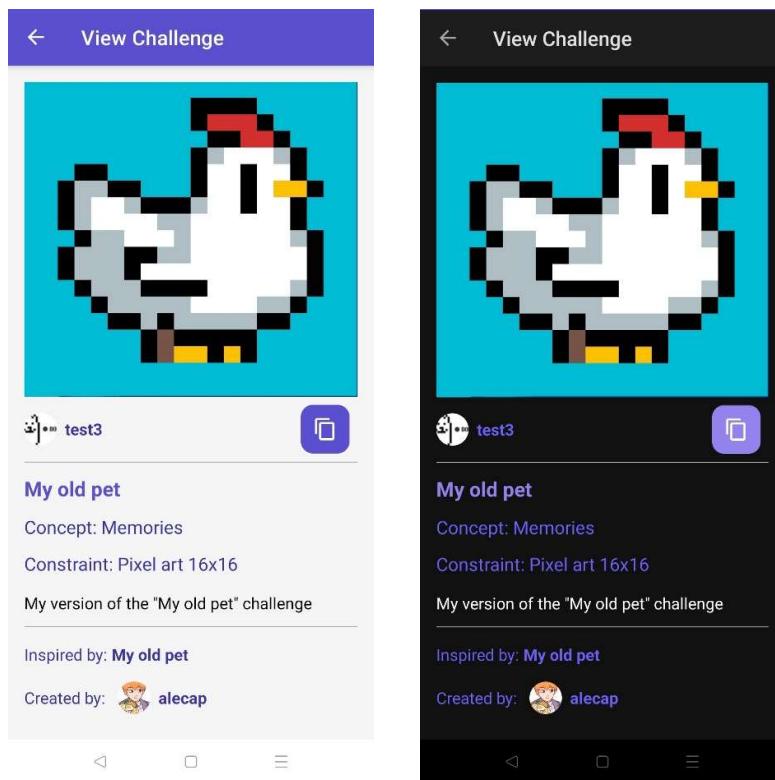
- Ricerca Challenge



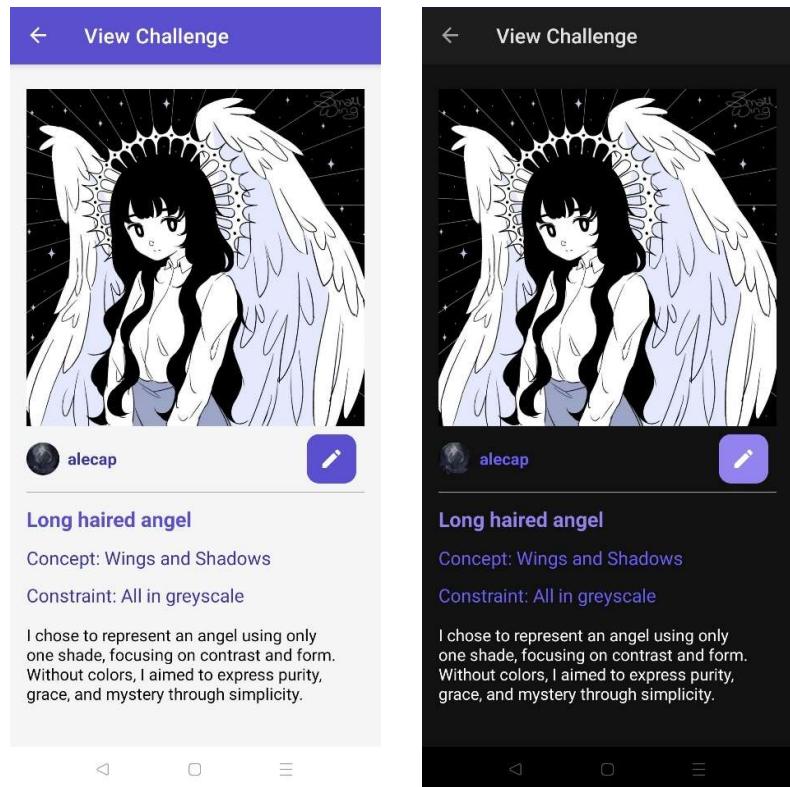
- **Schermata New Challenge**



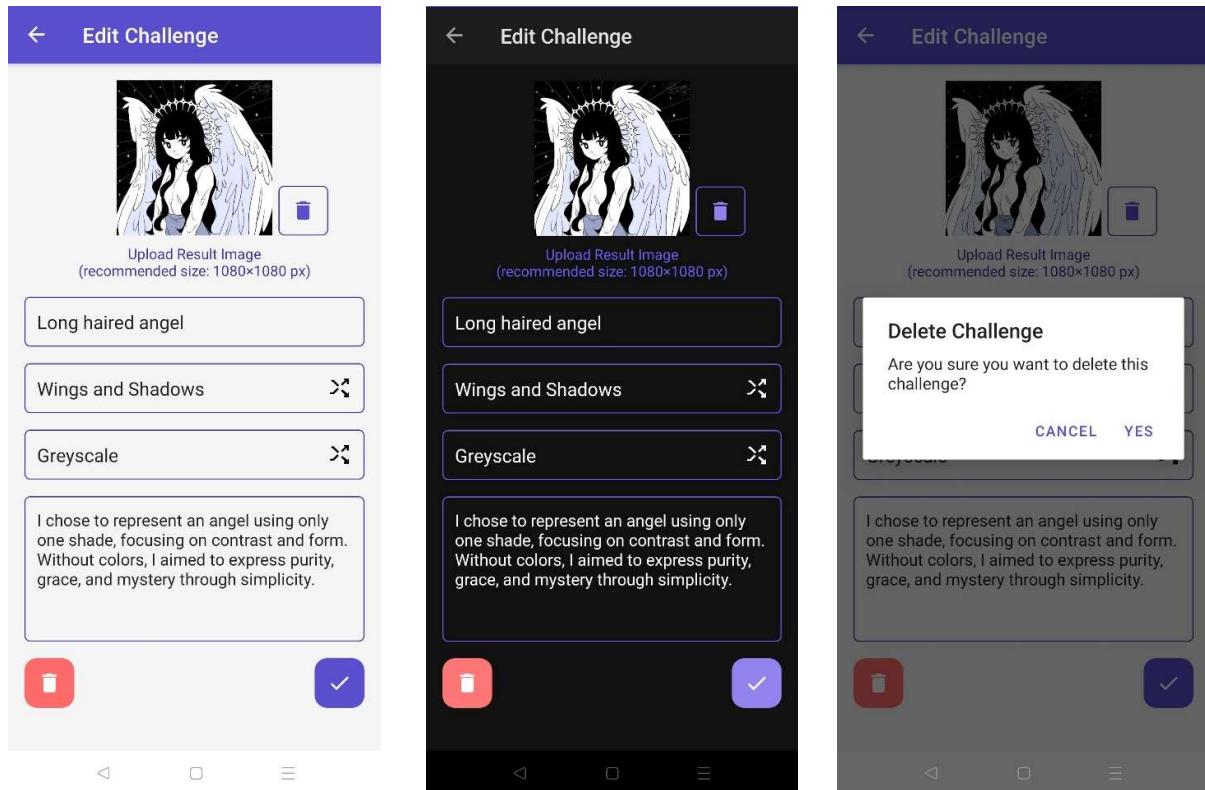
- **Schermata View Challenge** (con pulsante “Copia” per la fork, riferimento al profilo dell’autore, ed eventuali riferimenti della challenge copiata)



- **Schermata View Challenge** (propria challenge)

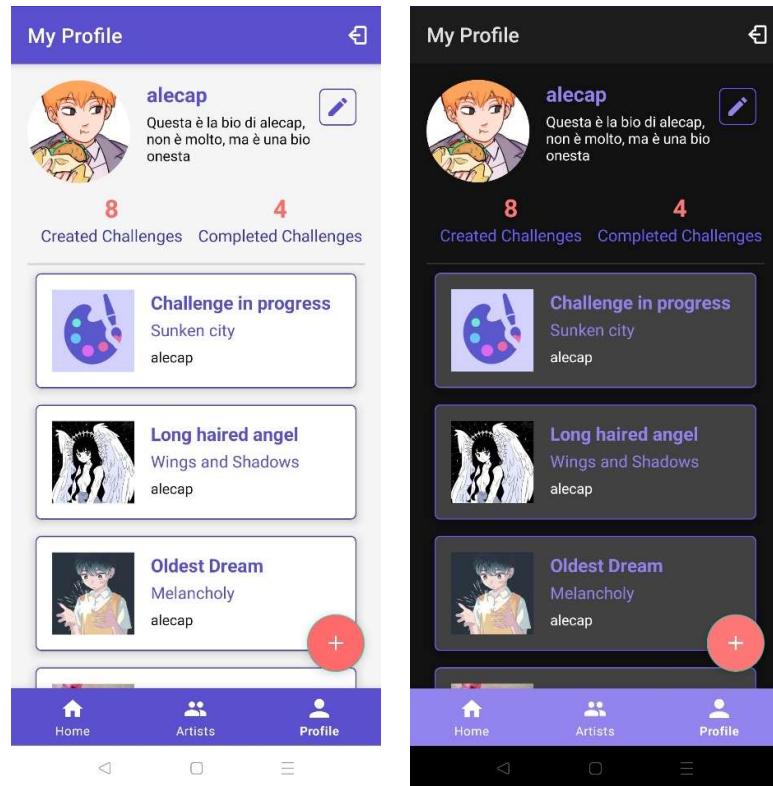


- **Schermata Edit Challenge** (raggiungibile cliccando sull'icona con la matita)

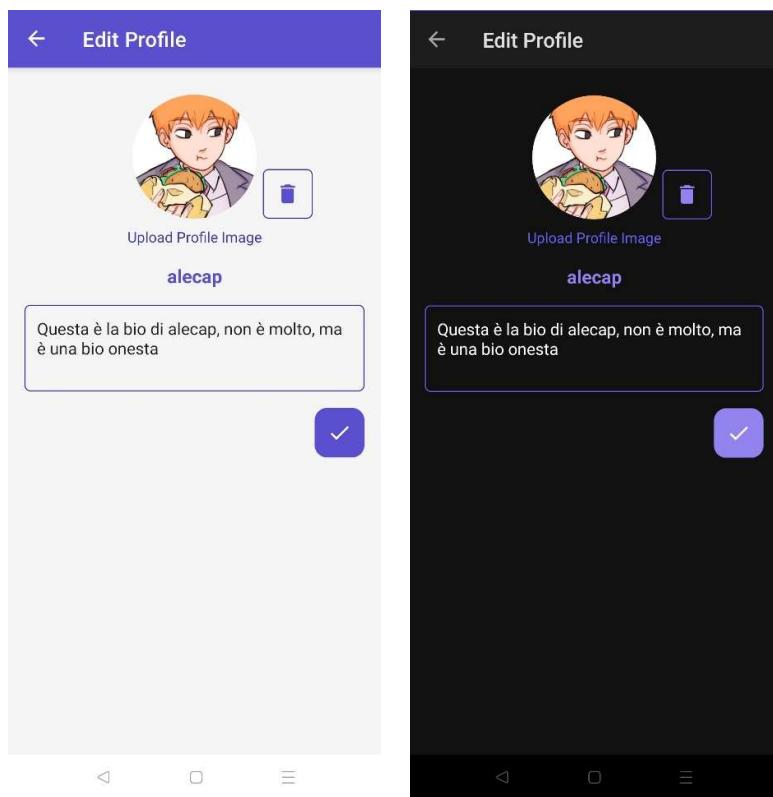


2.4.3. Profilazione

- Schermata User Profile (proprio profilo)

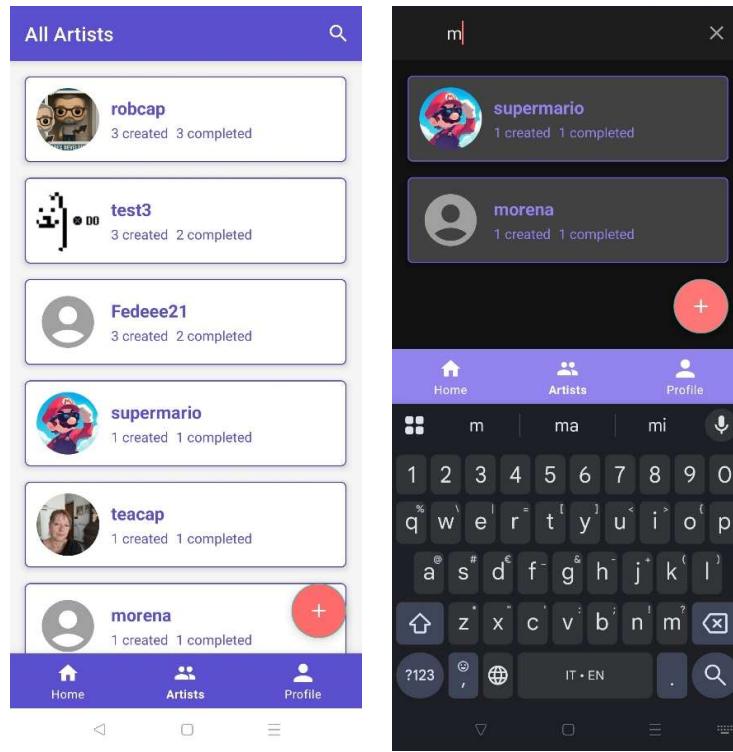


- Schermata Edit User Profile

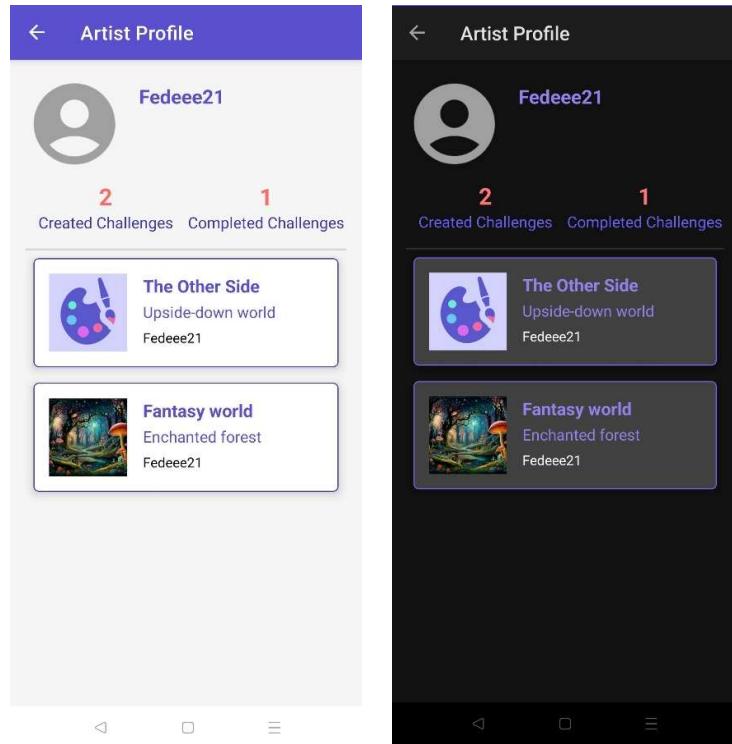


2.4.4. Community

- Schermata Artists con Ricerca utenti



- Schermata Other User Profile (profilo di un altro utente, raggiungibile o dalla sezione Artists o da una View Challenge di tale utente)



2.5. Database e memorizzazione dati

Inkspire utilizza **Supabase** come backend per la gestione del database relazionale PostgreSQL, dell'autenticazione utente e dello storage per le immagini.

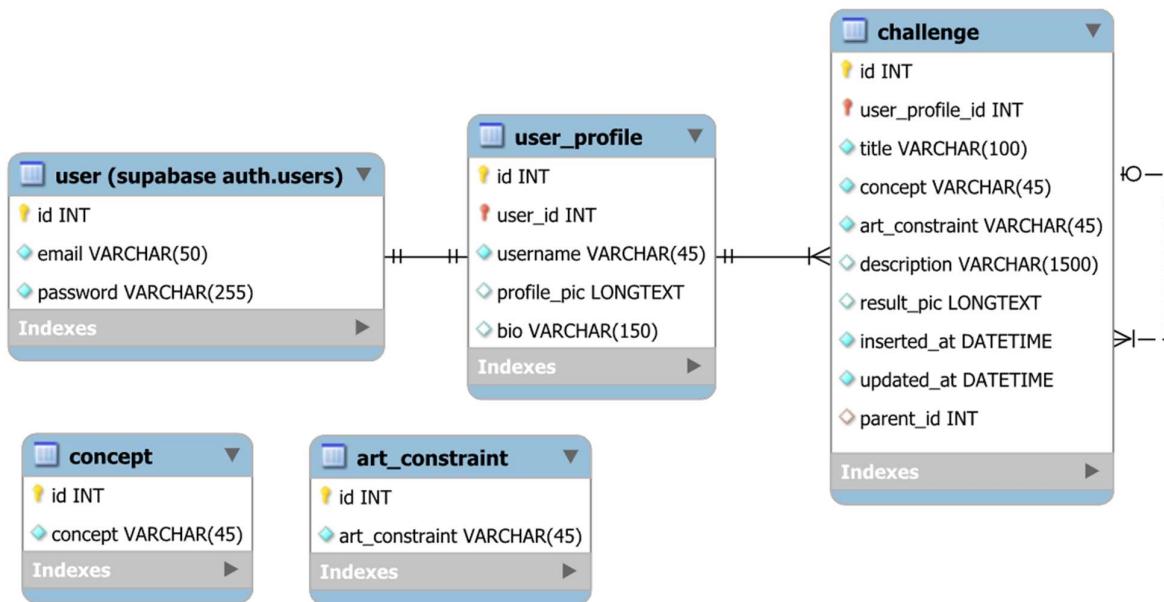
La progettazione del database è partita da una modellazione concettuale tramite diagramma E-R in MySQL Workbench, tradotta poi in una struttura fisica implementata in Supabase attraverso script SQL.

L'obiettivo principale è garantire semplicità, integrità referenziale e sicurezza, pur mantenendo un accesso efficiente alle informazioni, anche aggregate.

2.5.1. Modello relazionale

Il modello relazionale è stato progettato per rappresentare in maniera chiara la relazione tra utenti e le challenge da essi create. Le principali scelte progettuali includono:

- una relazione 1:N (uno a molti) tra le tabelle user_profile e challenge;
- una relazione ricorsiva 1:N tra challenge.id e challenge.parent_id della tabella challenge, per collegare le sfide derivate a quelle originali;
- tabelle autonome per temi (concept) e vincoli (art_constraint) usati nelle generazioni casuali;
- uso di viste (challenge_vw e user_profile_vw) per aggregare informazioni complesse per la UI, come negli elenchi delle challenge e degli utenti.



2.5.2. Struttura delle tabelle in Supabase

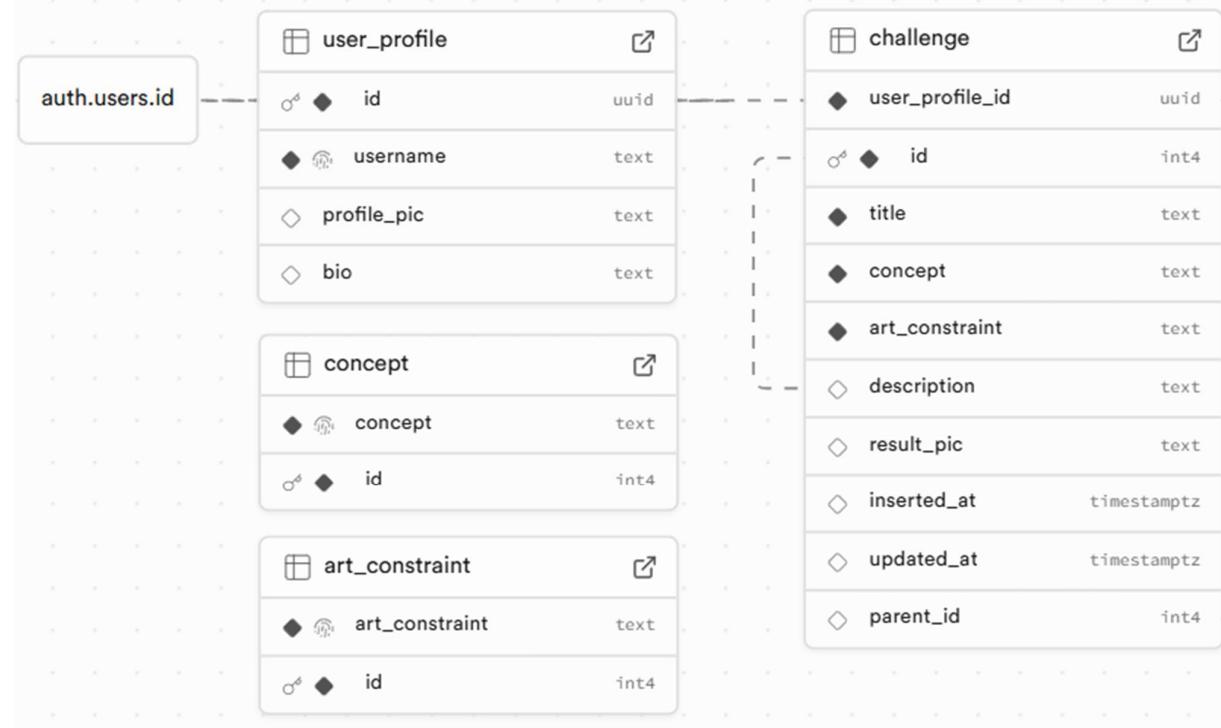


Tabella user_profile

```
create table public.user_profile (
    id uuid not null,
    username text not null,
    profile_pic text null,
    bio text null,
    constraint user_profile_pkey primary key (id),
    constraint user_profile_username_key unique (username),
    constraint user_profile_id_fkey foreign KEY (id)
        references auth.users (id) on delete CASCADE
) TABLESPACE pg_default;
```

Collega ogni utente autenticato, dalla tabella `auth.users` di supabase, al suo profilo personalizzato. Include:

- `username` univoco
- immagine del profilo e biografia (opzionali).

Tabella challenge

```
create table public.challenge (
    id serial not null,
    user_profile_id uuid not null,
    title text not null,
    concept text not null,
    art_constraint text not null,
    description text null,
    result_pic text null,
    inserted_at timestamp with time zone null default now(),
    updated_at timestamp with time zone null default now(),
    parent_id integer null,
    constraint challenge_pkey primary key (id),
    constraint challenge_parent_fk foreign KEY (parent_id)
        references challenge (id) on delete set null,
    constraint challenge_user_profile_id_fkey foreign KEY (user_profile_id)
        references user_profile (id) on delete CASCADE,
    constraint challenge_parent_not_self check (
        (
            (parent_id is null)
            or (parent_id <> id)
        )
    )
) TABLESPACE pg_default;

create index IF not exists idx_challenge_parent_id
    on public.challenge
    using btree (parent_id) TABLESPACE pg_default;

create trigger set_updated_at_challenge
before update on challenge for EACH row
execute FUNCTION update_updated_at_column ();
```

Contiene le sfide create dagli utenti, con i seguenti campi:

- title, concept, art_constraint (obbligatori);
- description, result_pic (facoltativi);
- user_profile_id: chiave esterna verso il profilo;
- parent_id (facoltativo): chiave esterna verso la stessa tabella challenge, per collegare le sfide derivate a quelle originali.

Tabelle concept e art_constraint

```
create table public.concept (
    id serial not null,
    concept text not null,
    constraint subject_pkey primary key (id),
    constraint subject_subject_key unique (concept)
) TABLESPACE pg_default;

create table public.art_constraint (
    id serial not null,
    art_constraint text not null,
    constraint challenge_constraint_pkey primary key (id),
    constraint challenge_constraint_challenge_constraint_key unique (art_constraint)
) TABLESPACE pg_default;
```

Tabelle scollegate, usate per selezionare casualmente temi e vincoli artistici durante la creazione e la modifica delle challenge.

2.5.3. Viste per aggregazione dati

Vista challenge_vw

```
create or replace view public.challenge_vw as
select c.id,
       u.id as user_id,
       c.title,
       c.concept,
       c.art_constraint,
       c.description,
       c.result_pic,
       c.inserted_at,
       c.updated_at,
       u.username,
       u.profile_pic,
       u.bio,
       -- campi parent
       c.parent_id,
       up_parent.id      as parent_user_id,
       up_parent.username as parent_username,
       up_parent.profile_pic as parent_profile_pic,
       c_parent.title    as parent_title
  from challenge c
  join user_profile u
    on u.id = c.user_profile_id
 left join challenge c_parent
    on c_parent.id = c.parent_id
 left join user_profile up_parent
    on up_parent.id = c_parent.user_profile_id;
```

Questa vista unisce i dati della challenge con quelli del suo autore e, in caso di fork, include anche le informazioni relative alla challenge originale (titolo, autore e immagine profilo) tramite join ricorsiva sulla tabella challenge.

Serve a popolare:

1. le cards degli elenchi delle challenge presenti nella Home e nei profili degli utenti;
2. le pagine di dettaglio delle challenge, includendo, in caso di fork, le informazioni di quelle originali.

Vista user_profile_vw

```
create view user_profile_vw as
select
    u.id,
    u.username,
    u.profile_pic,
    u.bio,
    count(c.id) as created_count,
    count(c.result_pic) filter (where c.result_pic is not null) as completed_count
from user_profile u
left join challenge c
    on c.user_profile_id = u.id
group by
    u.id,
    u.username,
    u.profile_pic,
    u.bio;
```

Unisce i dati dell'utente (username, bio, ecc.) ai valori calcolati del numero totale di challenge create e del numero di challenge completate (con result_pic presente). Serve principalmente a popolare le cards dell'elenco di utenti presenti nella sezione Artists.

Queste due viste ottimizzano le query nella UI evitando di scrivere e ripetere direttamente le join nel codice.

2.5.4. Autenticazione con Supabase Auth

Inkspire usa il provider email/password di Supabase. Gli utenti vengono registrati in auth.users, ma i dati visibili pubblicamente sono gestiti nella tabella user_profile, collegata tramite id. Questo consente di separare i dati sensibili da quelli pubblici e applicare policies diverse a seconda del contesto.

2.5.5. Policies di sicurezza

Per garantire la sicurezza a livello riga (Row Level Security), sono state attivate su tutte le tabelle le **RLS policies** con le seguenti regole principali:

- **Lettura pubblica:** per concept, art_constraint, user_profile e challenge (solo lettura)
- **Scrittura protetta:** solo l'autore può creare, modificare o eliminare le proprie challenge e modificare lo user_profile. Tutte le policies usano la condizione auth.uid() = user_profile_id per garantire che ogni utente possa agire solo sui propri dati.

Esempio: policy di SELECT e DELETE challenge

```
alter policy "Authenticated can read all challenges"
on "public"."challenge"
to authenticated
using (true);

alter policy "Users can delete their own challenge"
on "public"."challenge"
to authenticated
using (user_profile_id = auth.uid());

alter policy "Users can insert their own challenge"
on "public"."challenge"
to authenticated
with check (user_profile_id = auth.uid());

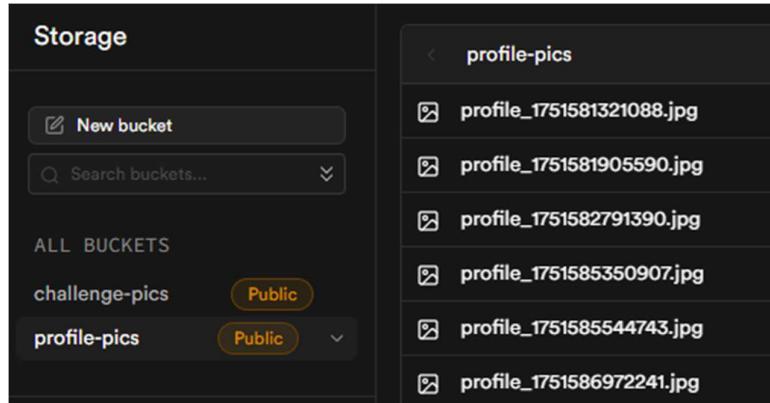
alter policy "Users can update their own challenge"
on "public"."challenge"
to authenticated
using (user_profile_id = auth.uid())
with check (user_pr);
```

challenge	
SELECT	Authenticated can read all challenges Applied to: authenticated role
DELETE	Users can delete their own challenge Applied to: authenticated role
INSERT	Users can insert their own challenge Applied to: authenticated role
UPDATE	Users can update their own challenge Applied to: authenticated role

2.5.6. Supabase Storage

Sono stati creati due bucket:

- **profile-pics**: immagini profilo utente
- **challenge-pics**: immagini delle challenge completate

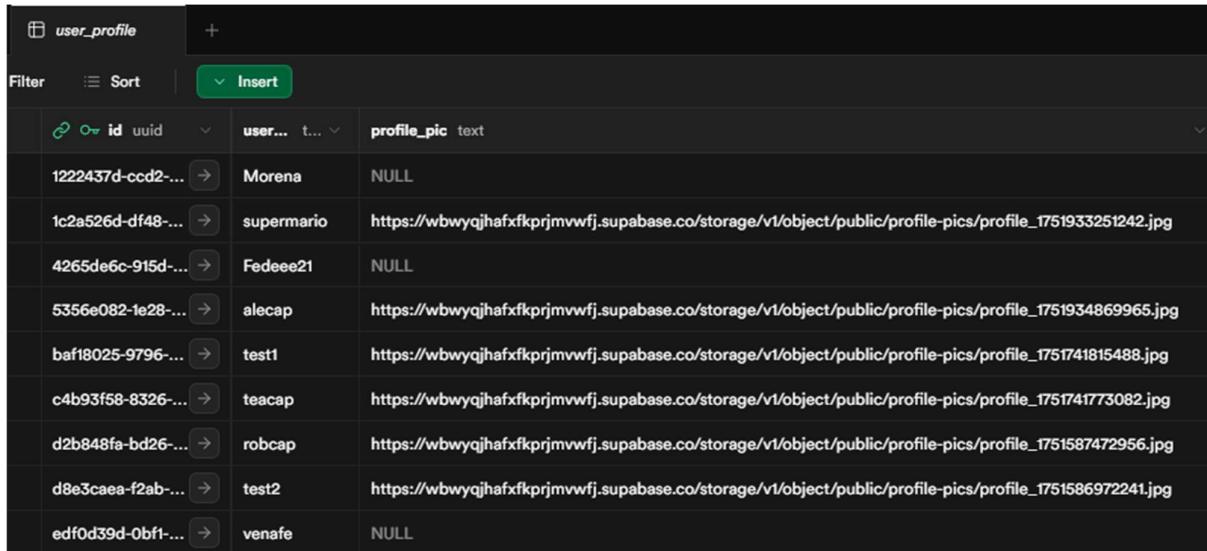


The screenshot shows the Supabase Storage interface. On the left, there's a sidebar with a 'New bucket' button and a search bar. Below it, under 'ALL BUCKETS', are two buckets: 'challenge-pics' and 'profile-pics', both set to 'Public'. The 'profile-pics' bucket is expanded, showing a list of six files, each with a thumbnail icon and a download link.

Policies impostate:

- Upload: consentito solo agli utenti autenticati
- Lettura: disponibile agli utenti autenticati

Le immagini vengono salvate tramite Supabase Storage API e referenziate nel database via URL nei campi `profile_pic` o `result_pic`.



	<code>id</code> <small>uuid</small>	<code>user...</code> <small>t...</small>	<code>profile_pic</code> <small>text</small>
	1222437d-ccd2-...	Morena	NULL
	1c2a526d-df48-...	supermario	https://wbwyqjhafxfkprjmwwfj.supabase.co/storage/v1/object/public/profile-pics/profile_1751933251242.jpg
	4265de6c-915d-...	Fedeee21	NULL
	5356e082-1e28-...	alecap	https://wbwyqjhafxfkprjmwwfj.supabase.co/storage/v1/object/public/profile-pics/profile_1751934869965.jpg
	baf18025-9796-...	test1	https://wbwyqjhafxfkprjmwwfj.supabase.co/storage/v1/object/public/profile-pics/profile_1751741815488.jpg
	c4b93f58-8326-...	teacap	https://wbwyqjhafxfkprjmwwfj.supabase.co/storage/v1/object/public/profile-pics/profile_1751741773082.jpg
	d2b848fa-bd26-...	robcap	https://wbwyqjhafxfkprjmwwfj.supabase.co/storage/v1/object/public/profile-pics/profile_1751587472956.jpg
	d8e3caea-f2ab-...	test2	https://wbwyqjhafxfkprjmwwfj.supabase.co/storage/v1/object/public/profile-pics/profile_1751586972241.jpg
	edf0d39d-0bf1-...	venafe	NULL

2.6. Architettura

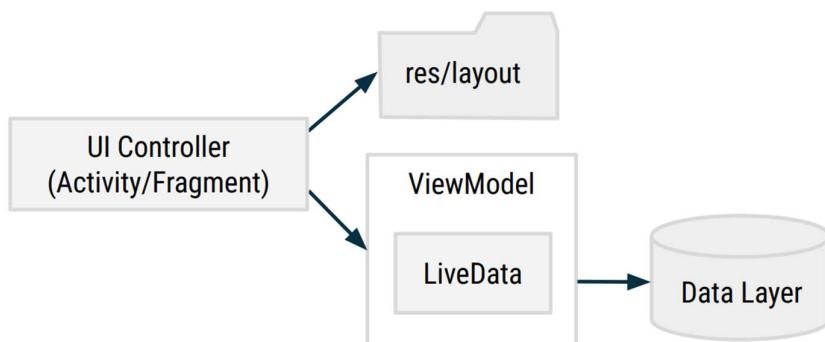
L'applicazione Inkspire è stata progettata secondo il principio della **Separation of Concerns**, ovvero la separazione delle responsabilità tra i vari componenti del software, al fine di garantire modularità, manutenibilità e facilità di test.

È stato adottato il pattern architettonico **MVVM** (Model–View–ViewModel), integrato con il **Repository Pattern** e l'utilizzo di **Supabase** come backend completo per autenticazione, database e storage. La comunicazione REST con il backend è gestita tramite il client **Ktor**.

2.6.1. MVVM Pattern

L'approccio MVVM prevede una chiara separazione tra:

- **Model**: classi dati che rappresentano le entità logiche dell'app. In questo progetto rispecchiano anche la struttura del database relazionale su Supabase;
- **View**: responsabile della visualizzazione dei dati e dell'interazione utente. Include i layout XML, gli Adapter, le Activity e i Fragment. Le View sono collegate ai rispettivi ViewModel tramite Data Binding e osservano lo stato tramite LiveData;
- **ViewModel**: mediatore tra View e Model. Espone dati osservabili (LiveData) e gestisce lo stato e la logica della UI in modo reattivo, indipendentemente dal ciclo di vita della View. I ViewModel invocano i metodi dei Repository ed eseguono operazioni asincrone tramite Kotlin Coroutine.



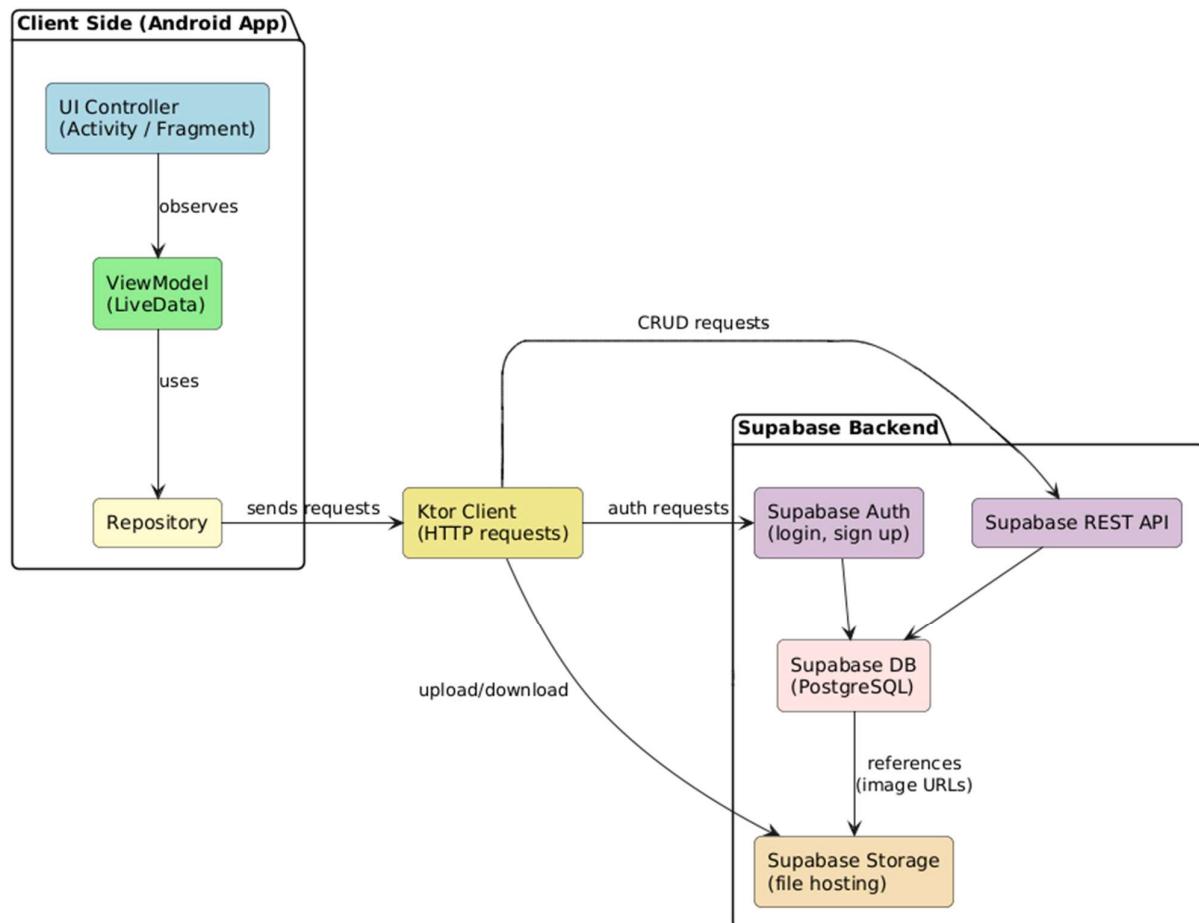
2.6.2. Repository Pattern

Il Repository è l'unico punto di accesso ai dati per il ViewModel. Ha il compito di astrarre l'origine dei dati (che può essere remota o locale) e incapsulare la logica di business necessaria per recuperarli o modificarli.

Nel caso di Inkspire, i Repository comunicano con il backend Supabase attraverso richieste HTTP gestite dalla libreria **Ktor**.

Questo approccio garantisce:

- maggiore testabilità dei singoli componenti;
- disaccoppiamento dalle librerie di rete o backend;
- facilità di estensione (es. aggiunta di cache o sorgenti dati alternative).



2.7. Sviluppo

2.7.1. Panoramica dello sviluppo

Il processo di sviluppo è partito dalla progettazione dei layout xml, che ha permesso di definire in anticipo la struttura visiva dell'app e fornire un riferimento concreto per l'implementazione successiva della logica.

Parallelamente sono stati definiti i Model, in modo da rispecchiare la struttura delle tabelle e delle viste definite sul database Supabase.

Dopo aver identificato le operazioni necessarie per interagire con i dati, sono stati sviluppati i Repository, contenenti le funzioni per la comunicazione col database, e a tal fine è stata utilizzata la libreria open-source Supabase Kotlin, che si appoggia su Ktor e offre un'interfaccia tipizzata per l'accesso ai servizi PostgREST, Auth e Storage di Supabase.

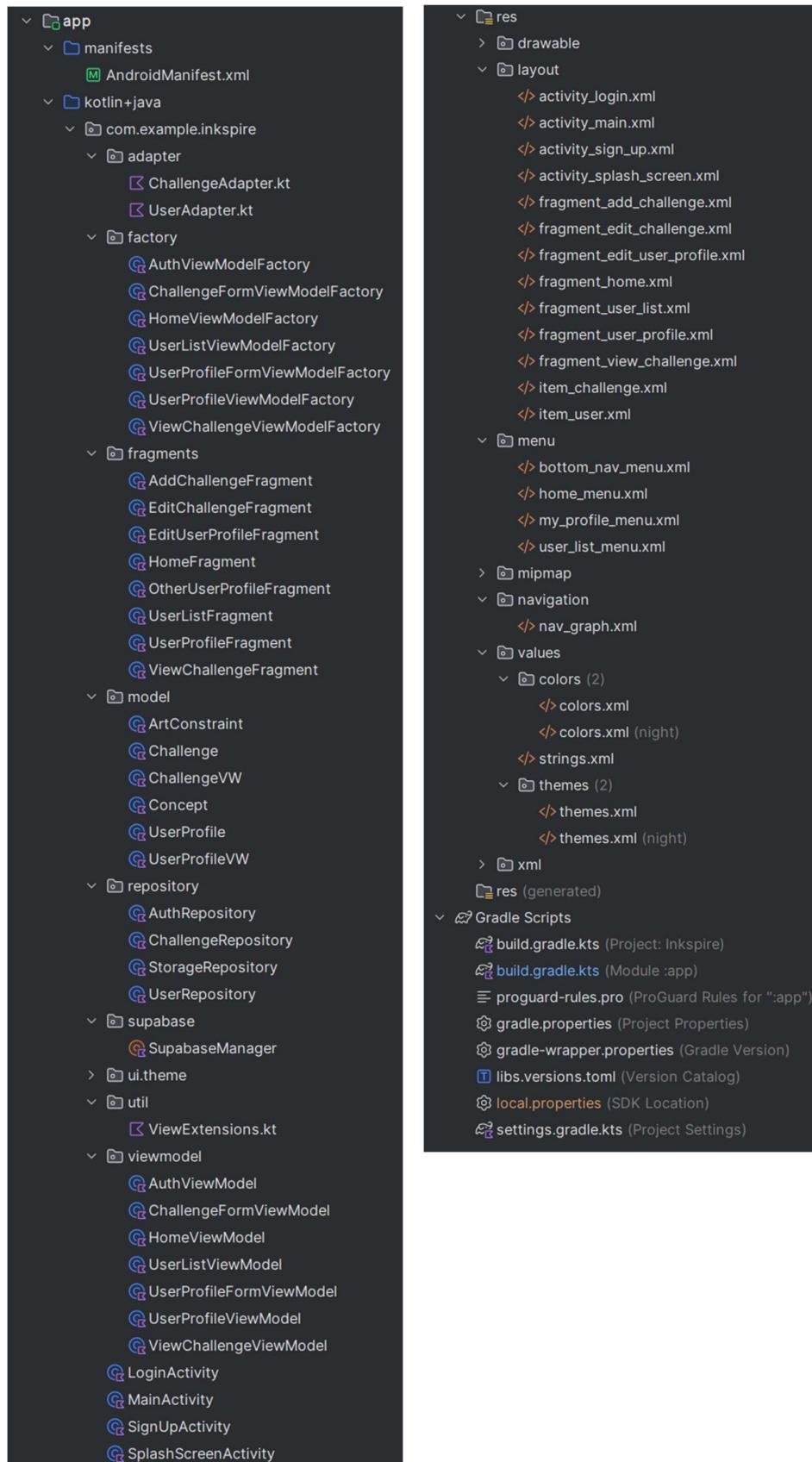
I ViewModel sono stati creati per ciascuna schermata che richiedesse logica o gestione dello stato, agendo da ponte tra repository e UI, per esporre in modo reattivo dati e operazioni verso i fragment.

Successivamente si è passati alla definizione degli Adapter per le liste dinamiche, all'implementazione dei Fragments e delle Activity, e infine all'integrazione del sistema di navigazione con Navigation Component e Safe Args.

L'autenticazione è stata una delle prime funzionalità implementate, e ha richiesto la configurazione iniziale di Supabase, il collegamento sicuro dell'app e la definizione dei relativi repository e.viewmodel per la gestione della registrazione e dell'accesso utente. Il ciclo di sviluppo è poi proseguito con la gestione delle challenge, l'implementazione delle sezioni dedicate al profilo utente e alla visualizzazione della community, e infine la gestione delle immagini attraverso Supabase Storage e Glide.

Proteggono la privacy di un utente Android. Vengono dichiarati con il tag <uses-permission> nel file AndroidManifest.xml

2.7.2. Struttura del progetto



Il codice è stato suddiviso in file e package distinti, ognuno con una responsabilità ben definita:

- **AndroidManifest.xml**: file xml che descrive i componenti principali dell'app al sistema Android prima che venga eseguita. Contiene:
 - permessi Android di connessione e interazione con storage esterno, dichiarati con il tag <uses-permission>;
 - dichiarazioni delle activity, specificando quali sono esportabili (avviabili direttamente dall'esterno) e quali interne, e quale tra queste avvia l'app (launcher);
 - configurazione e impostazioni globali app (nome, icona, tema, targetApi, ecc.)
- **build.gradle**: file di configurazione del progetto, in cui è stata disattivata la libreria Jetpack Compose in favore dei layout xml classici. Contiene:
 - dichiarazioni dei plugin utilizzati (kotlin-parcelize, serialization, navigation.safeargs);
 - caricamento delle chiavi segrete SUPABASE_URL e SUPABASE_KEY dal file .gitignored local.properties, evitando di esporle nel codice pubblico;
 - configurazione del modulo Android (namespace, compileSdk, ecc.) con l'abilitazione di dataBinding e viewBinding;
 - dipendenze utilizzate (ui classica, supabase, ktor, glide, navigation).
- **res**: sezione contenente i package con i file xml relativi ai layout delle schermate (layout), i temi (values/themes.xml) con i colori e gli stili, le immagini (drawable) e i menu di navigazione (menu);
- **navigation**: package contenente il file nav_graph.xml, definito con Android Navigation Component, che rappresenta la mappa di navigazione tra i vari fragment dell'app e gestisce anche il passaggio dei parametri in modo sicuro tramite Safe Args;
- **model**: package contenente le classi dati (data class) che rappresentano le entità dell'app. Ogni model rispecchia fedelmente la struttura delle tabelle e delle viste definite su Supabase e alcuni includono annotazioni utili per la serializzazione con Kotlinc Serialization, in quanto Supabase utilizza JSON come formato di comunicazione;
- **repository**: package contenente le classi che fungono da intermediari tra i viewmodel e il backend Supabase. Le funzioni all'interno dei repository sono

implementate utilizzando le API fornite dalla libreria postgrest-kt di Supabase, e permettono operazioni CRUD sulle challenge, autenticazione, gestione profili e accesso ai dati delle viste;

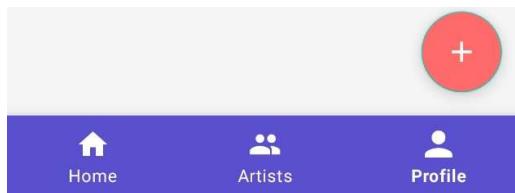
- **viewmodel**: package contenente i ViewModel associati ai diversi fragment. Ogni viewmodel incapsula lo stato e la logica della relativa schermata e interagisce esclusivamente con il repository per recuperare e modificare i dati. Sono utilizzati LiveData per gestire lo stato in modo reattivo;
- **factory**: package contenente le classi ViewModelFactory necessarie per l'inizializzazione dei ViewModel con parametri personalizzati;
- **adapter**: package contenente gli Adapter utilizzati per le RecyclerView, in particolare per visualizzare l'elenco delle challenge e l'elenco degli utenti. In particolare gli adapter gestiscono anche il binding dei dati ai layout delle singole card;
- **manager**: package contenente una singola classe SupabaseManager, che centralizza la configurazione e l'accesso all'istanza di Supabase. Questo file inizializza i moduli auth, postgrest e storage, rendendoli disponibili a tutta l'applicazione tramite singleton;
- **utils**: raccoglie estensioni riutilizzabili per migliorare l'interattività dei componenti UI (EditText, View) e la loro integrazione con i LiveData.

2.7.3. Layout e UI

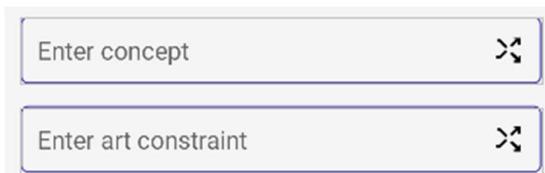
L’interfaccia di Inkspire è stata progettata con l’obiettivo di garantire semplicità, coerenza visiva e un’esperienza utente intuitiva. Tutti i layout sono stati implementati tramite file xml e resi dinamici attraverso il collegamento con i ViewModel e i LiveData, sfruttando il binding dei dati e gli observer per aggiornare la UI in tempo reale.

Layout di Activity e Fragment

- **activity_splash_screen.xml:** layout per lo splash screen iniziale di ingresso all’app, contiene solo una ImageView dell’icona di Inkspire e una ProgressBar;
- **activity_main.xml:** layout principale con NavHostFragment per la gestione della navigazione, una BottomAppBar con BottomNavigationView per la navigazione tra schermate principali (Home, Artists, Profile) e un FloatingActionButton centrale per l’aggiunta di una nuova challenge;



- **fragment_login.xml e fragment_signup.xml:** schermate di autenticazione con i relativi campi.
- **fragment_home.xml:** contiene una RecyclerView per visualizzare le challenge e un’ImageView (empty_bg) da mostrare quando l’elenco è vuoto;
- **fragment_add_challenge.xml e fragment_edit_challenge.xml:** form di creazione e modifica di una challenge, con campi EditText, di cui quelli del concept e art_constraint dotati di drawable per lo shuffle, ImageView per l’immagine e Button per il salvataggio/eliminazione;



- **fragment_view_challenge.xml:** mostra i dettagli di una challenge con titolo, tema, vincolo, descrizione, immagine e link al profilo dell’autore. Se la challenge non appartiene all’utente corrente, il pulsante di “Edit” viene sostituito dal pulsante “Copia” per fare la fork. In caso di challenge derivata, sotto il box

della descrizione vengono mostrati anche il riferimento alla challenge e all'autore originali;

Inspired by: Parent Challenge Title

Created by:  username

- **fragment_user_profile.xml** e **fragment_edit_profile.xml**: rispettivamente per la visualizzazione e la modifica del profilo utente. Il file di layout è lo stesso per il profilo dell'utente corrente e per quello di un altro utente, con la differenza che nel secondo caso non è presente il pulsante per passare alla schermata di modifica del profilo;
- **fragment_user_list.xml**: contiene una RecyclerView per visualizzare tutti gli utenti e un'ImageView (empty_bg) da mostrare quando l'elenco è vuoto.

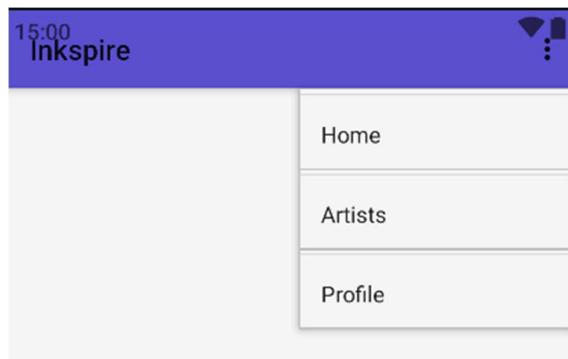
Layout degli items

- **Item_challenge.xml** e **item_user.xml**: layout per le CardView di challenge e utente da visualizzare nella RecyclerView.



Menu

Il menu principale è gestito tramite il file **bottom_nav_menu.xml**, che definisce le voci di navigazione visualizzate nella BottomNavigationView.



Sono presenti anche altri menu differenziati in base al fragment: `home_menu` e `fragment_user_list` con l'icona per la ricerca e `my_profile_menu` con l'icona di logout.

Gestione temi e colori

L'app supporta tema chiaro e tema scuro, definiti nel file `themes.xml` e `themes-night.xml`. I colori principali sono personalizzati e centralizzati nel file `colors.xml`.

Risorse grafiche

Le immagini statiche, le icone personalizzate e gli sfondi sono contenuti nella cartella `res/drawable`. Vengono per lo più utilizzate icone vettoriali per garantire una buona scalabilità e qualità visiva su tutti i dispositivi.

2.7.4. Autenticazione, Storage e accesso al DB

L'intero back-end di Inkspire è ospitato su Supabase, scelto perché combina:

- un **database PostgreSQL** gestito (accessibile via PostgREST);
- un modulo **Auth** pronto all'uso;
- uno **Storage** per file multimediali;
- **SDK Kotlin** (supabase-kt): una libreria specifica che usa internamente Ktor per comunicare con il database, offrendo un'interfaccia più comoda e tipizzata.

Auth: Registrazione e login e-mail/password

Per l'autenticazione viene utilizzata la tabella di sistema auth.users gestita da Supabase. Il flusso di tale processo è il seguente:

1. **Sign-up:** l'AuthRepository controlla che lo username sia libero (user_profile), poi invoca auth.signInWith(Email)
2. **Login/Logout:** tramite auth.signInWith(Email) e auth.signOut()
3. **Sessione:** il client è configurato con autoLoadFromStorage = true e alwaysAutoRefresh = true, per cui il token JWT viene rinnovato in background e l'utente resta loggato tra un avvio e l'altro.

Il codice chiave risiede nel SupabaseManager, dove il modulo Auth viene installato contestualmente alla creazione del client:

```
object SupabaseManager {

    val client: SupabaseClient by lazy {
        createSupabaseClient(
            supabaseUrl = BuildConfig.SUPABASE_URL,
            supabaseKey = BuildConfig.SUPABASE_ANON_KEY
        ) {
            install(Auth) {
                autoLoadFromStorage = true
                alwaysAutoRefresh = true
            }
            install(Postgres)
            install(Storage)
        }
    }
}
```

SupabaseManager.kt: classe singleton (object) per logiche di servizio. A differenza dei ViewModel che sono legati all'UI e al suo ciclo di vita (LiveData), i Manager sono classi usate solo per incapsulare una certa logica (funzioni di basso livello, accesso API,

ecc.) che vengono istanziate una sola volta e sono accessibili globalmente. SupabaseManager contiene la funzione per inizializzare il client Supabase, e viene chiamato all'avvio dell'app:

- lazy: crea pigramente un'istanza del client, cioè un'istanza che si inizializza solo al primo accesso e non subito all'avvio dell'app se non serve immediatamente;
- install(Auth): installa il modulo di auth supabase e tenta di recuperare token/sessione salvata in locale;
- install(Postgrest): installa il modulo per interagire col database Supabase (PostgreSQL) tramite il client REST PostgREST.

Storage: Upload delle immagini

Per le immagini dei risultati delle challenge e delle foto profilo degli utenti vengono creati i due buckets pubblici challenge-pics e profile-pics con le relative policy: gli utenti autenticati possono caricare (upload) e leggere (download) i file nel proprio bucket.

L'utilizzo dell'SDK è riportato nella funzione uploadImage() dello StorageRepository:

```
return try {
    SupabaseManager.storage.from(bucket)
        .upload(
            path = filePath,
            data = byteArray
        )
    ) {
        upsert = true
        this.contentType = contentType
    }
}

val publicUrl = "https://${SupabaseManager.client.supabaseUrl}/storage/v1/object/public/$bucket/$filePath"
```

Nella quale terminato l'upload viene costruito manualmente l'URL pubblico che viene poi salvato nel database e caricato nell'app con Glide.

Accesso al database con la libreria Supabase-kt

Le operazioni sul database avvengono tramite chiamate REST generate dall'SDK, che serializza/deserializza JSON in oggetti Kotlin:

```
supabase.from("challenge")
    .update(updateData) { filter { eq("id", challenge.id) } }
    | select()
}
| .decodeSingle<Challenge>()
```

2.7.5. Model

Tutti i Model dell'app sono stati definiti utilizzando classi dati di Kotlin annotate con **@Serializable** per consentire la serializzazione automatica da e verso JSON, necessaria per la comunicazione con il backend Supabase tramite la libreria supabase-kt. L'annotazione **@Parcelize** è stata aggiunta ai model relativi alle challenge e agli utenti per permettere il passaggio sicuro e diretto degli oggetti tra fragment tramite SafeArgs.

L'approccio seguito è stato quello di mantenere una **corrispondenza 1:1** tra i model e le tabelle e viste del database Supabase, così da semplificare il mapping automatico con le risposte JSON ottenute via PostgREST.

I model realizzati sono:

- **Challenge**: rappresenta una sfida creata da un utente, associata tramite foreign key al suo profilo ed eventualmente alla challenge “parent” se è stata replicata.
- **UserProfile**: contiene le informazioni pubbliche del profilo utente (username, bio, immagine).
- **ChallengeVW**: model mappato sulla view challenge_vw creata in Supabase per ottenere i dati aggregati tra challenge e autore. In caso di fork include anche le informazioni relative alla challenge originale. Viene utilizzato per popolare le recycler view e per semplificare l'accesso ai dettagli completi delle challenge.
- **UserProfileVW**: model mappato sulla view user_profile_vw, utile per ottenere statistiche sull'attività dell'utente (challenge create e completate) da mostrare nel suo profilo e nella recycler view degli utenti.
- **Concept** e **ArtConstraint**: modelli per la generazione casuale delle challenge, corrispondenti a due tabelle distinte e indipendenti.

2.7.6. Repository

I repository rappresentano il livello di accesso ai dati dell'applicazione, centralizzando le operazioni di comunicazione tra l'interfaccia utente e il backend Supabase. Tutti i repository sono scritti come classi Kotlin che sfruttano le **coroutine** e la libreria supabase-kt per eseguire richieste asincrone in formato JSON tramite PostgREST.

Ogni repository si occupa di un insieme logico di operazioni. Oltre all'AuthRepository e StorageRepository visti in precedenza, i due repository principale dell'app sono:

- **ChallengeRepository.kt**: gestisce tutte le operazioni CRUD sulle challenge: inserimento, aggiornamento, cancellazione e ricerca. Utilizza metodi come decodeSingle<T>() o decodeList<T>() per convertire automaticamente le risposte JSON in oggetti Kotlin, e restituisce Boolean o liste di oggetti a seconda del contesto. Come esempio si ritportano le funzioni:

```
suspend fun insertChallenge(challenge: Challenge): Boolean {
    return try {
        supabase.from("challenge")
            .insert(challenge) {
                select()
            }
            .decodeSingle<Challenge>()
        true
    } catch (e: Exception) {
        false
    }
}
```

insertChallenge() inserisce un oggetto Challenge nel database e ritorna true in caso di successo.

```
suspend fun getAllChallenges(): List<ChallengeVW> {
    return supabase.from("challenge_vw")
        .select {
            order("updated_at", Order.DESCENDING)
        }
        .decodeList<ChallengeVW>()
}
```

getAllChallenges() restituisce l'elenco completo delle challenge aggregate (tramite la view challenge_vw), ordinate per data di aggiornamento.

- **UserRepository.kt**: gestisce il recupero e l'aggiornamento dei profili utente, l'accesso ai dati aggregati view (come user_profile_vw) e il calcolo delle

statistiche individuali (challenge totali e completate). Anche in questo caso è stato adottato un approccio robusto con try/catch per la gestione delle eccezioni e l'utilizzo di mappe (Map<String, Any?>) per aggiornare solo i campi modificabili (come bio e immagine):

```
suspend fun updateUserProfile(userProfile: UserProfile): Boolean {
    return try {
        val updateData = mapOf(
            "bio" to userProfile.bio,
            "profile_pic" to userProfile.profile_pic
        )
        client.from("user_profile")
            .update(updateData) {
                filter { eq("id", userProfile.id) }
                select()
            }
            .decodeSingleOrNull<UserProfile>() != null
    } catch (e: Exception) {
        e.printStackTrace()
        false
    }
}
```

Tutte le query sono eseguite utilizzando una sintassi fluida, definita tramite lambda per impostare i filtri o l'ordinamento.

2.7.7. ViewModel

Per ciascuna schermata con logica associata è stato definito un ViewModel dedicato. I ViewModel fungono da intermediari tra la UI e i repository, mantenendo separata la logica di business dalla logica di presentazione. La comunicazione tra ViewModel e Fragment avviene in modo unidirezionale, principalmente tramite LiveData, osservata all'interno della UI.

Ogni ViewModel espone:

- Dati di dominio (es. elenco challenge, profili utente, dettagli)
- Eventuali messaggi o segnali di stato (es. esito delle operazioni, caricamento in corso)
- Funzioni per attivare operazioni asincrone (es. login, inserimento, aggiornamento, eliminazione)

I ViewModel definiti sono i seguenti:

- **HomeViewModel**: gestisce il recupero e la ricerca delle challenge visibili nella Home;
- **ChallengeFormViewModel**: fornisce tutte le operazioni CRUD su una challenge, gestisce l'upload dell'immagine su Supabase Storage e suggerisce parametri random per tema e vincolo artistico;
- **UserProfileViewModel** e **UserProfileFormViewModel**: si occupano del profilo utente, rispettivamente per la visualizzazione e la modifica dei dati.
- **ViewChallengeViewModel**: carica i dati di una challenge verificando anche se l'utente corrente è l'autore della challenge selezionata, in modo da far adattare l'interfaccia di conseguenza;
- **AuthViewModel**: gestisce la logica di autenticazione e la propagazione di messaggi di feedback;
- **UserListViewModel**: mostra gli utenti della community e filtra in base a ricerche testuali.

In presenza di dipendenze da repository, i ViewModel sono stati istanziati tramite **Factory** personalizzate, al fine di supportare parametri nel costruttore. Tutte le operazioni asincrone sono gestite con viewModelScope.launch per evitare memory leak e rispettare il ciclo di vita dell'interfaccia.

2.7.8. Adapter

Per la visualizzazione delle liste dinamiche in RecyclerView, sono stati definiti due adapter principali che estendono **ListAdapter**, sfruttando DiffUtil.ItemCallback per ottimizzare il rendering e ridurre i ricaricamenti superflui:

- **ChallengeAdapter**: utilizzato per mostrare l'elenco delle challenge pubblicate, sia nella home che nei profili utente. Ogni elemento della lista è rappresentato dal layout item_challenge.xml, che mostra il titolo, il concept e l'autore della challenge, insieme all'immagine del risultato (se disponibile). Le immagini vengono caricate asincronamente tramite Glide, con supporto a placeholder e progress bar.
- **UserAdapter**: utilizzato per visualizzare la lista degli utenti. Ogni elemento (item_user.xml) mostra il nome utente, le statistiche delle challenge create/completate e l'immagine del profilo, anch'essa caricata tramite Glide con effetto circleCrop().

Entrambi gli adapter espongono una **lambda onClick** per gestire gli eventi di selezione dell'elemento, facilitando la navigazione verso i dettagli o i profili utente.

L'organizzazione degli adapter segue il **pattern ViewHolder** interno e utilizza il binding automatico delle view tramite classi generate (ItemChallengeBinding, ItemUserBinding), favorendo la leggibilità e riduzione di codice ridondante.

```
class ChallengeAdapter(
    private val onChallengeClick: (ChallengeVW) -> Unit
) : ListAdapter<ChallengeVW, ChallengeAdapter.ChallengeViewHolder>(ChallengeDiffCallback()) {

    inner class ChallengeViewHolder(private val binding: ItemChallengeBinding) :
        RecyclerView.ViewHolder(binding.root) {

        fun bind(challengeUser: ChallengeVW) {...}

    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ChallengeViewHolder {...}
    override fun onBindViewHolder(holder: ChallengeViewHolder, position: Int) {...}
}

class ChallengeDiffCallback : DiffUtil.ItemCallback<ChallengeVW>() {
    override fun areItemsTheSame(oldItem: ChallengeVW, newItem: ChallengeVW) =
        oldItem.id == newItem.id

    override fun areContentsTheSame(oldItem: ChallengeVW, newItem: ChallengeVW) =
        oldItem == newItem
}
```

2.7.9. Activity e Fragment

L'architettura dell'applicazione Inkspire è basata su una singola **MainActivity** che ospita tutti i fragment dell'applicazione tramite un **NavHostFragment**. A questa struttura si affiancano due activity distinte dedicate all'autenticazione (**LoginActivity** e **SignUpActivity**) e una **SplashScreenActivity** che viene visualizzata all'avvio per verificare lo stato di login dell'utente.

La **MainActivity** funge da **contenitore** per tutti i fragment dell'interfaccia principale, integrando una **BottomNavigationView** e un **FloatingActionButton**, la cui visibilità viene gestita dinamicamente in base alla destinazione attiva. I principali fragment dell'app si suddividono in due macro-categorie:

Area challenge e community

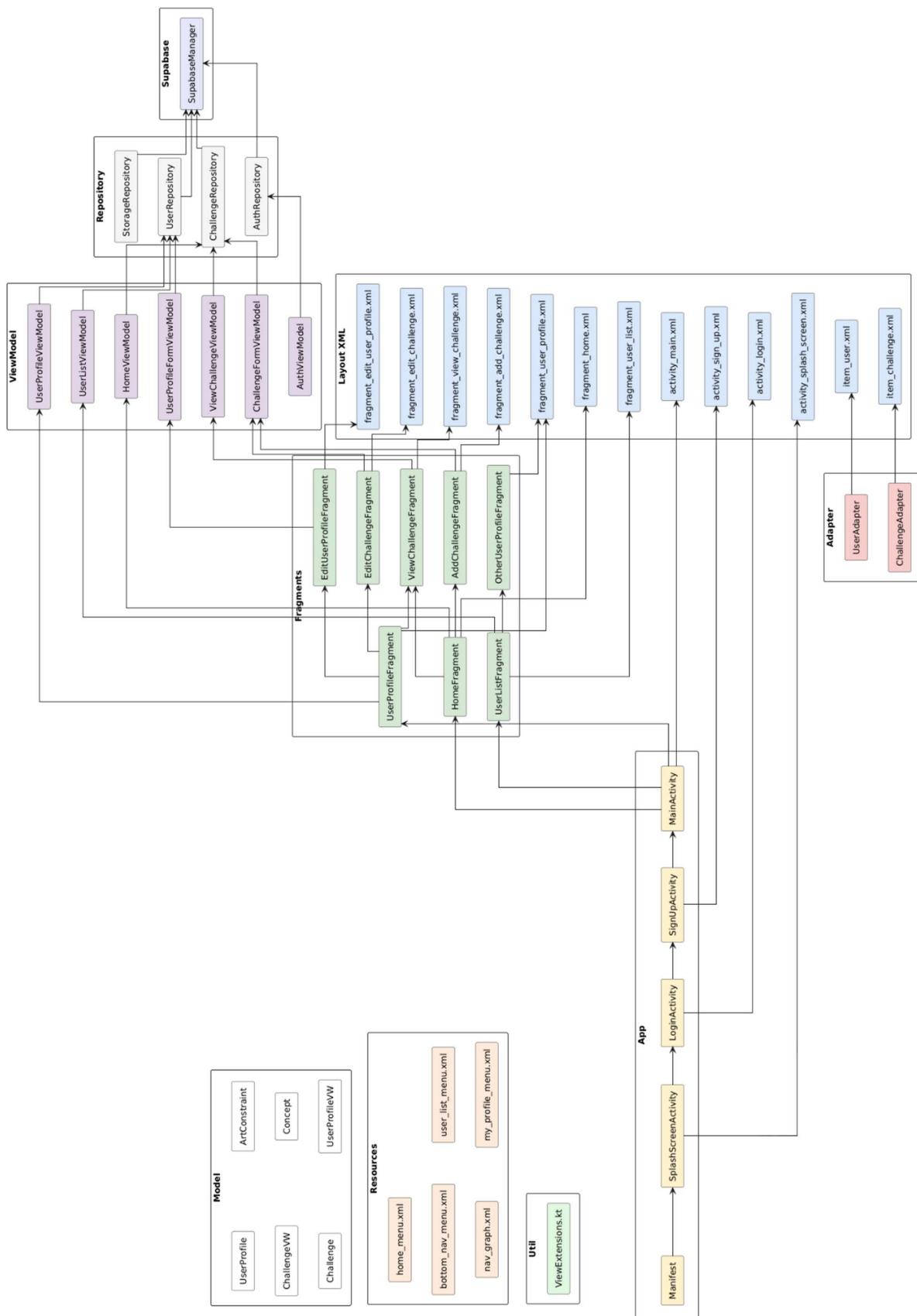
- **HomeFragment**: mostra l'elenco delle challenge pubblicate, tramite una RecyclerView alimentata da **HomeViewModel**. Supporta la ricerca tramite campo testuale e osserva lo stato di loading per aggiornare l'interfaccia;
- **AddChallengeFragment**: consente la creazione di una nuova challenge, con caricamento opzionale di un'immagine e suggerimenti random per il concept e l'art constraint;
- **EditChallengeFragment**: consente la modifica o eliminazione di una challenge esistente. I dati della challenge vengono caricati tramite ID e gestiti dal **ChallengeFormViewModel**;
- **ViewChallengeFragment**: visualizza i dettagli di una singola challenge. Se l'utente corrente è l'autore, viene mostrato il pulsante “Edit” per modificare il contenuto, altrimenti è presente il pulsante “Copia” per la fork. In caso la challenge sia derivata, vengono mostrati anche i riferimenti alla challenge originale e al suo autore;
- **UserListFragment**: elenca tutti gli utenti registrati, ordinati per numero di challenge completate, con possibilità di ricerca. Le informazioni sono fornite dallo **UserListViewModel**.

Area profilo utente

- **UserProfileFragment**: mostra i dati del profilo corrente. È possibile accedere alla modifica del profilo o effettuare il logout tramite i pulsanti dedicati;

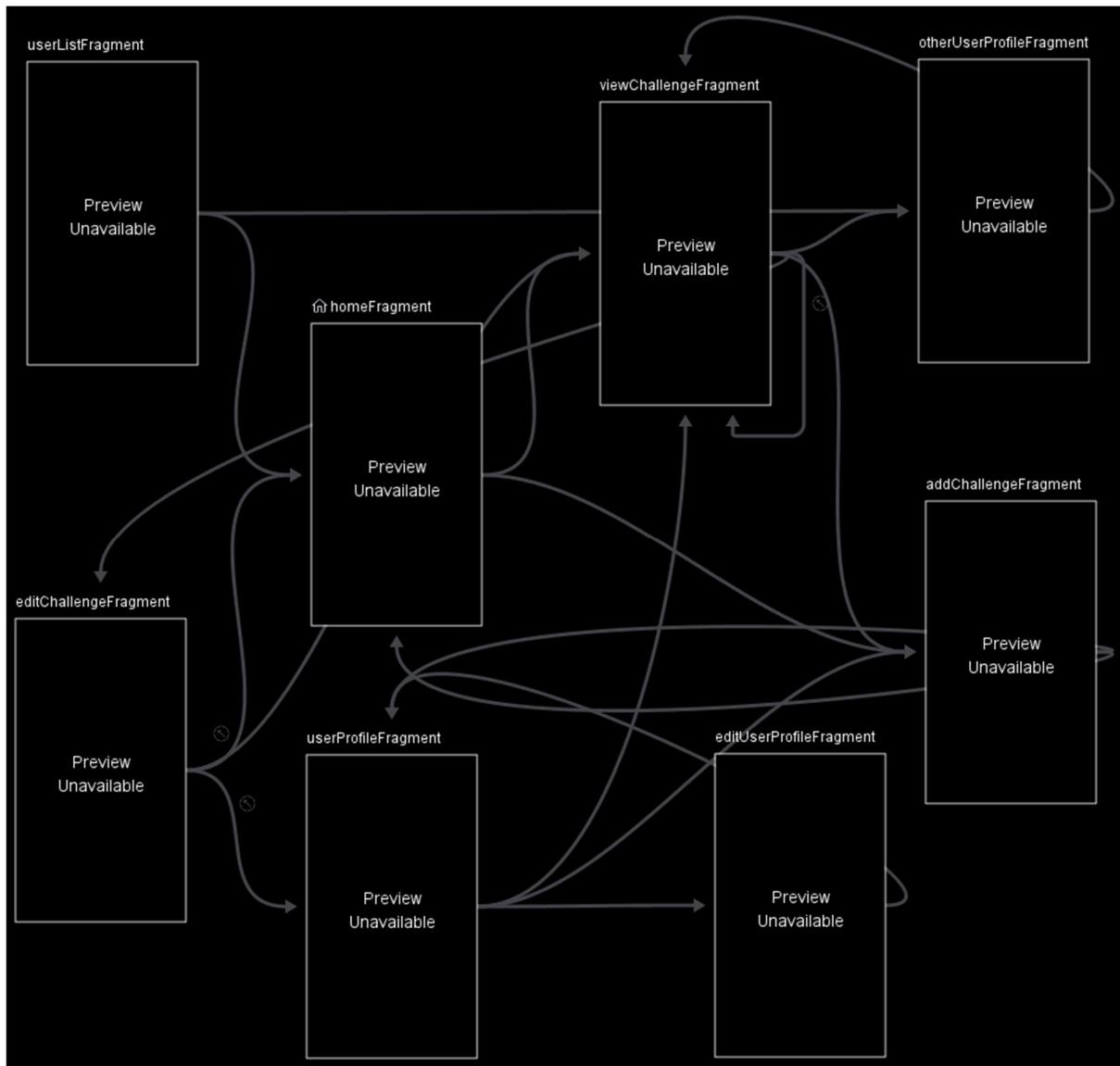
- **EditUserProfileFragment**: permette all’utente di aggiornare bio e immagine del profilo. I dati sono gestiti dallo UserProfileFormViewModel;
- **OtherUserProfileFragment**: mostra il profilo di un altro utente in sola lettura, raggiungibile dai dettagli di una challenge.

Di seguito lo schema dei collegamenti tra tutti i file del progetto:



2.7.10. Navigazione

La navigazione tra i fragment è gestita dal **Navigation Component**, con grafo di navigazione definito nel file nav_graph.xml. I parametri vengono passati in modo type-safe tramite **SafeArgs**, assicurando coerenza tra fragment. La gestione del back stack e della navigation bar è centralizzata in MainActivity, in base alla destinazione corrente.



2.8. Testing

Questa sezione della relazione è dedicata alle attività di testing svolte sul progetto. In particolare, sono state prese in esame due tipologie di test complementari:

1. **Local Unit Tests**: test unitari svolti su singoli componenti per volta, mirati a verificare la correttezza della logica applicativa senza dipendere dal framework Android;
2. **Instrumented test**: test eseguiti su emulatore o dispositivo reale, per validare il corretto funzionamento delle componenti legate all’interfaccia utente e all’interazione con il sistema operativo.

2.8.1. Local Unit Tests

Questi test sono contemporaneamente:

- **test delle unità**: test di piccole dimensioni che verificano solo una sezione ristretta dell'app, come un metodo o una classe;
- **test locali**: test “lato host” eseguiti sulla macchina di sviluppo.

I Local Unit Tests sono eseguiti sulla JVM del PC, quindi non richiedono emulatori o dispositivi, e servono per testare la logica indipendente dall’interfaccia utente.

Il framework di riferimento per il testing in Java e Kotlin è JUnit, mentre la libreria di asserzioni utilizzata per una migliore lettura dei test è **junit.Assert**. In questi test si è utilizzato il framework **JUnit 4**, integrato in Android Studio.

Sono stati sviluppati due unit test d’esempio per:

- verificare il corretto funzionamento della logica di gestione delle challenge: operazioni CRUD e funzioni di ricerca nel ChallengeRepository;
- validare la generazione casuale di concept e vincoli artistici.

Unit Test 1: ChallengeRepository

Poiché il repository originale ChallengeRepository si appoggia a Supabase e quindi dipende da una connessione esterna, è stato creato un **FakeChallengeRepository**, ossia una versione semplificata in memoria che replica le principali operazioni del repository reale. Questo approccio ha permesso di eseguire i test in modo isolato, rapido e ripetibile, senza dipendere dal database remoto.

I test sviluppati hanno avuto come obiettivo la verifica delle operazioni CRUD e delle funzioni di ricerca:

- **FakeChallengeRepository.kt**: versione fittizia del ChallengeRepository, realizzata in memoria, che non si collega a Supabase, ma mantiene le challenge in una lista locale mutabile. Implementa gli stessi metodi del repository reale: insert/update/delete challenge, getChallengeById, getChallengesByUser e searchChallenges.

Di seguito alcuni esempi dei metodi replicati:

```
class FakeChallengeRepository {

    private val challenges = mutableListOf<Challenge>()
    private var autoIncrementId = 1

    fun insertChallenge(challenge: Challenge): Boolean {
        val newChallenge = challenge.copy(id = autoIncrementId++)
        challenges.add(newChallenge)
        return true
    }

    fun updateChallenge(challenge: Challenge): Boolean {
        val index = challenges.indexOfFirst { it.id == challenge.id }
        return if (index != -1) {
            challenges[index] = challenge
            true
        } else {
            false
        }
    }

    fun deleteChallenge(challengeId: Int): Boolean {
        return challenges.removeIf { it.id == challengeId }
    }

    suspend fun getAllChallenges(): List<Challenge> {
        delay(10) // Delay Coroutines che simula un ritardo come se fosse una query
        return challenges.toList()
    }
}
```

- **ChallengeRepositoryTest.kt**: Contiene le classi di test JUnit per validare il comportamento del repository:
 - in `@Before` viene creato un **ambiente di test** inizializzando il `FakeChallengeRepository` con alcuni dati di esempio;
 - i metodi `@Test` verificano le principali operazioni;
 - ogni test confronta i valori ottenuti con quelli attesi tramite `assertEquals` e altre asserzioni.

Per la gestione delle coroutine è stato utilizzato il costrutto `runBlocking`, il quale avvia una coroutine ma blocca il thread finché il codice asincrono non ha finito. Tale costrutto permette di chiamare direttamente funzioni suspend (es. `getAllChallenges()`) dentro i test JUnit, che altrimenti non potrebbero gestirle.

```
class ChallengeRepositoryTest {

    private lateinit var repository: FakeChallengeRepository

    @Before
    fun setUp() {
        repository = FakeChallengeRepository()
    }

    @Test
    fun insertChallenge_shouldAddChallenge() = runBlocking {
        val challenge = Challenge(
            id = 0,
            user_profile_id = "user1",
            title = "Test Title",
            concept = "Concept",
            art_constraint = "Constraint"
        )

        val result = repository.insertChallenge(challenge)
        val all = repository.getAllChallenges()

        assertTrue(result)
        assertEquals(1, all.size)
        assertEquals("Test Title", all[0].title)
    }
}
```

Unit Test 2: Generazione Casuale

Oltre ai test sulle operazioni CRUD del ChallengeRepository, è stato sviluppato un test dedicato alla funzione di generazione casuale dei concept. Il test verifica che il valore restituito dalla funzione getRandomConcept() appartenga sempre all'insieme dei valori validi, garantendo la correttezza logica della generazione casuale.

È stata aggiunta la seguente funzione al FakeChallengeRepository:

```
private val concepts = listOf("Lonely tree", "Sunset desert", "Wildflowers", "Enchanted forest")

fun getRandomConcept(): String? {
    return concepts.randomOrNull()
}
```

la quale è stata testata nel file di test **RandomConceptTest.kt**:

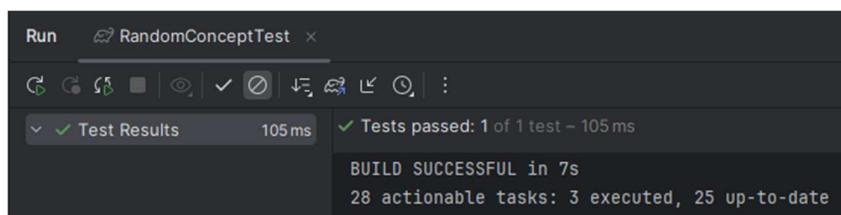
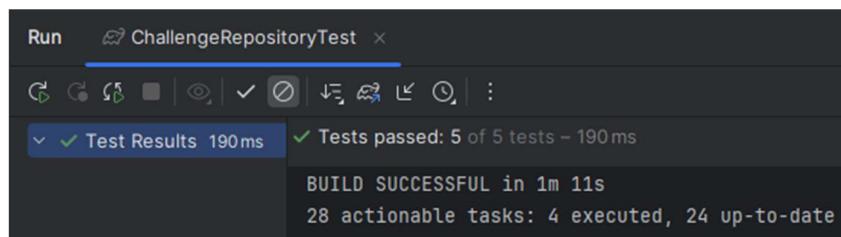
```
class RandomConceptTest {

    private lateinit var repo: FakeChallengeRepository

    @Before
    fun setUp() {
        repo = FakeChallengeRepository()
    }

    @Test
    fun testRandomConceptIsValid() = runBlocking {
        val concept = repo.getRandomConcept()
        val validConcepts = listOf("Lonely tree", "Sunset desert", "Wildflowers", "Enchanted forest")
        assertTrue(validConcepts.contains(concept))
    }
}
```

L'esecuzione degli Unit Test ha confermato il corretto comportamento della logica testata:



2.8.2. Instrumented Tests

I Test Strumentati vengono eseguiti direttamente sul dispositivo fisico o emulato e consentono di verificare il corretto funzionamento dell'app in un contesto reale, con accesso al framework Android e alle componenti di sistema. A differenza degli Unit Test, che si focalizzano sulla logica delle singole classi, i test strumentali hanno come obiettivo l'interazione con l'interfaccia utente e la validazione dei principali flussi applicativi.

Per la loro implementazione è stato utilizzato **Espresso**, il framework ufficiale di Google per i **test UI su Android**, che permette di simulare interazioni dell'utente come click, inserimento di testo e scorrimenti, e di verificare lo stato dei widget a schermo tramite asserzioni. Le annotazioni principali impiegate sono:

- `@RunWith(AndroidJUnit4::class)` per specificare il runner dei test strumentali;
- `@LargeTest` per marcare i test di UI di più ampia portata;
- `@Rule ActivityScenarioRule` per avviare automaticamente l'Activity interessata durante il test.

Sono stati implementati quattro casi principali:

- Login con credenziali corrette;
- Creazione di una challenge e verifica della sua visualizzazione nella lista;
- Apertura della schermata di dettaglio challenge;
- Modifica del profilo utente e aggiornamento della bio.

InstTest 1: Login e apertura Home

Questo test verifica il corretto funzionamento della procedura di Login con credenziali valide e della successiva apertura della schermata Home.

The screenshot shows the Android Studio interface. In the top half, the code editor displays `LoginInstrumentedTest.kt` with the following content:

```
13  @RunWith(AndroidJUnit4::class)
14  @LargeTest
15  class LoginInstrumentedTest {
16
17      @Test
18      fun loginWithValidCredentials_opensHome() {
19
20          // Avvia LoginActivity
21          ActivityScenario.launch(LoginActivity::class.java)
22
23          // Inserisci email e password
24          onView(withId(R.id.loginEmail))
25              .perform(typeText("ale.cap@test.com"), closeSoftKeyboard())
26
27          onView(withId(R.id.loginPassword))
28              .perform(typeText("alecap"), closeSoftKeyboard())
29
30          // Clicca il bottone di login
31          onView(withId(R.id.loginButton)).perform(click())
32
33          // Pausa manuale per dare tempo al cambio Activity + caricamento dati
34          Thread.sleep(5000)
35
36          // Verifica che la RecyclerView della Home sia visibile
37          onView(withId(R.id.challengeRecyclerView))
38              .check(matches(isDisplayed()))
39      }
40  }
```

In the bottom right corner, the "Test Results" window shows the following details:

Status	1 passed	1 tests, 43s 607ms	
Filter tests:	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>		
Tests		Duration	OPPO CPH2565
✓ Test Results		9s	1/1
✓ LoginInstrumentedTest		9s	1/1
✓ loginWithValidCredentials_opensHome		9s	✓

Test Results:
Starting 1 tests on CPH2565 - 14
CPH2565 - 14 Tests 1/1 completed. (0 skipped) (0 failed)
Finished 1 tests on CPH2565 - 14
BUILD SUCCESSFUL in 43s
71 actionable tasks: 5 executed, 66 up-to-date

Il codice in **LoginInstrumentedTest.kt** è strutturato come segue:

- viene lanciata la `LoginActivity`;
- si inseriscono credenziali già esistenti nel database (`ale.cap@test.com` / `alecap`);
- dopo il click sul pulsante di login, si attende il cambio Activity;
- si verifica la presenza della `RecyclerView` nella Home, a conferma che l'autenticazione è andata a buon fine.

InstTest 2: Creazione e visualizzazione Challenge nella Home

Questo test è stato sviluppato per la creazione di una challenge, che compila i campi obbligatori, la salva e verifica che la nuova sfida compaia nella lista della Home.

The screenshot shows the Android Studio interface with the code editor and the run tab. The code editor contains the `AddChallengeInstrumentedTest.kt` file, which includes a `@Before` block for logging in and a `@Test` block for creating a challenge and verifying its display. The run tab shows the test results: 1 passed, 1 test, duration 42s 497ms, and a detailed breakdown of the test execution.

```
17 class AddChallengeInstrumentedTest {
18
19     @Before
20     fun ensureLoggedIn() {
21         runBlocking {
22             val repo = AuthRepository()
23             repo.login("ale.cap@test.com", "alecap")
24         }
25     }
26
27     @Test
28     fun createChallenge_displaysInRecyclerView() {
29         onView(withId(R.id.addChallengeFab)).perform(click())
30
31         onView(withId(R.id.addChallengeTitle))
32             .perform(typeText("Espresso Test Challenge"), closeSoftKeyboard())
33
34         onView(withId(R.id.addChallengeConcept))
35             .perform(typeText("Testing Concept"), closeSoftKeyboard())
36
37         onView(withId(R.id.addChallengeConstraint))
38             .perform(typeText("Use 3 colors"), closeSoftKeyboard())
39
40         onView(withId(R.id.addChallengeButton)).perform(click())
41
42         Thread.sleep(3000)
43
44         onView(withId(R.id.challengeRecyclerView))
45             .check(matches(hasDescendant(withText("Espresso Test Challenge"))))
46     }
47 }
```

Status			Test Results	
			CPH2565 - 14 Tests 0/1 completed. (0 skipped) (0 failed)	
			Finished 1 tests on CPH2565 - 14	
Filter tests:	✓	✗	Duration	OPPO CPH2565
Tests	10s	1/1	BUILD SUCCESSFUL in 42s	
✓ Test Results	10s	1/1	71 actionable tasks: 5 executed, 66 up-to-date	
✓ AddChallengeInstrumentedTest	10s	1/1		
✓ createChallenge_displaysInRecyclerView	10s	✓		

Il codice in `AddChallengeInstrumentedTest.kt` è strutturato come segue:

- prima dell'avvio, il metodo `@Before ensureLoggedIn()` effettua un login tramite `AuthRepository`;
- si apre la schermata di inserimento con il FAB;
- si compilano i campi obbligatori: titolo, concept e constraint;
- si salva la challenge;
- si verifica che la challenge appena creata compaia nella lista della Home.

InstTest 3: Visualizzazione dettaglio Challenge

Questo test ha lo scopo di verificare la corretta apertura del dettaglio di una challenge.

The screenshot shows the Android Studio interface. At the top, there is a code editor with the following Java code:

```
16 > class ViewChallengeInstrumentedTest {
17
18     @get:Rule
19     val activityRule = ActivityScenarioRule(MainActivity::class.java)
20
21     @Test
22     fun openChallengeView_displaysContent() {
23
24         Thread.sleep(2000)
25
26         // Clicca sul primo elemento della RecyclerView
27         onView(withId(R.id.challengeRecyclerView))
28             .perform(
29                 RecyclerViewActions.actionOnItemAtPosition<RecyclerView.ViewHolder>(
30                     0, click()
31                 )
32             )
33
34         Thread.sleep(2000)
35
36         // Verifica che il titolo della challenge sia visibile
37         onView(withId(R.id.viewChallengeTitle))
38             .check(matches(isDisplayed()))
39
40         // Verifica anche che il concept sia mostrato
41         onView(withId(R.id.viewChallengeConcept))
42             .check(matches(isDisplayed()))
43     }
44 }
```

Below the code editor is a "Run" tab with the text "ViewChallengeInstrumentedTest". The main window displays the test results:

Status	1 passed	1 tests, 45 s 175 ms							
Filter tests:	✓	↻	◊	✗	↑	↓	⌚	⟳	⟳
Tests	Duration	OPPO CPH2565							
✓ Test Results	7 s	1/1							
✓ ViewChallengeInstrumentedTest	7 s	1/1							
✓ openChallengeView_displaysContent	7 s	✓							

On the right side, there is a "Test Results" panel with the following output:

```
✓ Test Results
Connected to process 15684 on device 'oppo-cph2565-e4f6ef01'.
Starting 1 tests on CPH2565 - 14
Finished 1 tests on CPH2565 - 14
BUILD SUCCESSFUL in 44s
71 actionable tasks: 5 executed, 66 up-to-date
```

Il codice in **ViewChallengeInstrumentedTest.kt** è strutturato come segue:

- viene avviata la MainActivity;
- si clicca sul primo elemento della RecyclerView della Home;
- si attende il caricamento del fragment di dettaglio;
- si controlla che titolo e concept della challenge siano effettivamente visibili a schermo;

InstTest 4: Modifica Profilo

Questo test mira a verificare la procedura di modifica del proprio profilo utente, in particolare l'aggiornamento della bio.

The screenshot shows the Android Studio interface. At the top is the code for `EditUserProfileInstrumentedTest.kt`. Below the code is the test run results window. The results show 1 test passed in 45s, 818ms. The test name is `editUserProfile_updatesBio`. The results panel also displays build information: "BUILD SUCCESSFUL in 45s" and "71 actionable tasks: 5 executed, 66 up-to-date".

```
17 class EditUserProfileInstrumentedTest {
18     @Before
19     fun ensureLoggedIn() {
20         runBlocking {
21             val repo = AuthRepository()
22             repo.login("ale.cap@test.com", "alecap")
23         }
24     }
25
26     @Test
27     fun editUserProfile_updatesBio() {
28         // Vai al profilo utente tramite bottom navigation
29         onView(withId(R.id.userProfileFragment)).perform(click())
30
31         Thread.sleep(2000)
32
33         onView(withId(R.id.editProfileButton)).perform(click())
34
35         Thread.sleep(2000)
36
37         val newBio = "Espresso updated bio"
38         onView(withId(R.id.editProfileBio))
39             .perform(clearText(), typeText(newBio), closeSoftKeyboard())
40
41         onView(withId(R.id.saveProfileButton)).perform(click())
42
43         Thread.sleep(2000)
44
45         // Verifica che la nuova bio sia visibile nel fragment del profilo
46         onView(withId(R.id.profileBio)).check(matches(withText(newBio)))
47     }
48 }
49
50 
```

Run		EditUserProfileInstrumentedTest	
Run	Stop	...	
Status	1 passed	1 tests, 45s 818 ms	✓ Test Results
Filter tests:	✓	0	CPH2565 - 14 Tests 0/1 completed. (0 skipped) (0 failed)
Tests	Duration	OPPO CPH2565	Finished 1 tests on CPH2565 - 14
✓ Test Results	12s	1/1	BUILD SUCCESSFUL in 45s
✓ EditUserProfileInstrumentedTest	12s	1/1	71 actionable tasks: 5 executed, 66 up-to-date
✓ editUserProfile_updatesBio	12s	✓	

Il codice in `EditUserProfileInstrumentedTest.kt` è strutturato come segue:

- tramite `@Before` viene eseguito il login automatico;
- si accede al profilo utente dalla bottom navigation;
- si apre la schermata di modifica profilo;
- si aggiorna il campo bio con un nuovo testo;
- si salva la modifica;
- si verifica che la nuova bio sia visibile nel fragment del profilo.

3. Progettazione e Sviluppo Flutter

Flutter è un framework open-source di Google che permette di sviluppare app per Android, iOS, Web e Desktop partendo da un solo codice sorgente scritto in Dart.

Il punto di forza di Flutter è il suo **motore di rendering**: non si appoggia ai componenti nativi del sistema, ma disegna direttamente i propri widget. Questo garantisce aspetto e comportamento coerenti su tutte le piattaforme.

L’interfaccia nasce dalla composizione di **widget**, componenti grafici riutilizzabili che possono avere o meno uno stato interno (stateful/stateless).

Durante lo sviluppo si possono sfruttare le funzionalità **hot reload** e **hot restart** di Flutter, le quali risultano molto più veloci di una ricompilazione completa. L’hot reload inietta il nuovo codice nell’app senza riavviarla, ricostruisce l’interfaccia e mantiene lo stato corrente, rendendo immediata la verifica di modifiche a layout e logica di UI. L’hot restart riavvia rapidamente l’app e reimposta lo stato, utile quando cambiano inizializzazioni o configurazioni globali.

3.1. Sviluppo Flutter e differenze con quello Android

La versione Flutter di Inkspire adotta un’architettura analoga alla MVVM usata nello sviluppo Android, ma con strumenti diversi:

- **UI**: in Android i layout sono dei file xml gestiti in modo **imperativo** da Activity/Fragment, dove i componenti grafici vengono trovati tramite id e viene scritto del codice per variarli esplicitamente. In Flutter, invece, la UI è scritta interamente in **Dart** come composizione di **widget dichiarativi**: ogni widget descrive come deve apparire l’interfaccia in base allo stato corrente, e Flutter ricostruisce automaticamente ciò che serve quando lo stato cambia.
- **Stato e logica**: in Android ci sono i ViewModel che espongono LiveData, mentre in Flutter lo stato è gestito con **Riverpod**, un framework di state management che fornisce dei Provider per salvare, leggere e aggiornare lo stato dell’app.
- **Backend**: in Android si usa il client Supabase per Kotlin, mentre in Flutter l’equivalente è **supabase_flutter**, che copre autenticazione, database e storage.

- **Navigazione:** in Android c'è Navigation Component, mentre in Flutter c'è `go_router` che gestisce le rotte con redirect protetti che instradano automaticamente da Splash a Login o area autenticata.

In sintesi, Flutter mantiene la stessa struttura concettuale, ma con strumenti più adatti a una UI multiplattforma, riducendo differenze tra ambienti.

3.2. Requisiti

I **requisiti funzionali**, quelli **non funzionali** e i **casi d'uso** sono gli stessi dell'app Android, esclusi la fork di una challenge e l'aggiunta di una challenge altrui tra le proprie:

- **RF1.** Sign Up e Login via Supabase Auth.
- **RF2.** Lista challenge community e profili.
- **RF3–RF6.** CRUD challenge con titolo, concept, art constraint (con generazioni casuali) e descrizione e immagine opzionali. Visualizzazione dettaglio challenge.
- **RF7–RF9.** Visualizzazione e modifica del profilo proprio e visualizzazione profili altrui.
- **RF10.** Logout con invalidazione dati.

I requisiti non funzionali sono:

- **RNF1.** UI semplice e coerente con Material 3.
- **RNF2.** Linguaggio: Dart/Flutter.
- **RNF3.** Architettura MVVM-like con Riverpod.
- **RNF4.** Persistenza dati online su Supabase (PostgreSQL + Storage + Auth).
- **RNF5.** Visualizzazione immagini con `cached_network_image` e cache buster sugli URL.
- **RNF6.** Compatibilità Android API 33+.
- **RNF7.** Sicurezza via HTTPS/JWT con RLS Supabase lato server.

I casi d'uso sono elencati sinteticamente sono:

- **UC1–UC3:** Sign Up, Login e Logout.
- **UC4–UC7:** Profilo proprio, altrui e lista utenti.
- **UC8–UC10:** Lista con Search e Dettaglio challenge.
- **UC11–UC15:** CRUD challenge, generazione casuale, completamento con immagine.

3.3. Mockup UI

Sign Up

Email
test3@gmail.com

Username
test3

Password
password3|

Exception: Username already taken

Sign Up

Already registered? Login

Login

Email
ale.cap@test.com

Password
alecap|

Min 6 characters

Login

Don't have an account? Sign up

Home

The Other Side
Upside-down World • Use three-point perspective

Challenge in progress
Sunken city • All in greyscale

Long haired angel
Ethereal • All in greyscale

Home
 Artists
 Profile

Home

No challenge found

Try adjusting your search or create a new challenge.

mystery

Home
 Artists
 Profile

Home

The Other Side
Upside-down World • Use three-point perspective

fede | Fedee | fedee

Home
 Artists
 Profile

Long haired angel



 alecap 

Concept
Ethereal

Art constraint
All in greyscale

Description
I chose to represent an angel using only one shade, focusing on contrast and form. Without colors, I aimed to express purity, grace, and mystery through simplicity.

Edit Challenge



 Change pic  Remove
Recommended size: 1080x1080 (1:1)

Title
Long haired angel

Concept
Ethereal 

Art Constraint
All in greyscale 

Description (optional)
I chose to represent an angel using only one shade, focusing on contrast and form. Without colors, I aimed to express purity, grace, and mystery through simplicity.

 Save

Edit Challenge



 Change pic  Remove
Recommended size: 1080x1080 (1:1)

Title
Long haired angel

Concept
Ethereal

Art Constraint
All in greyscale

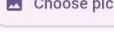
Delete Challenge

Are you sure you want to delete this challenge?

 Cancel 

 Save

New Challenge

 Choose pic
Recommended size: 1080x1080 (1:1)

Title
Solitude 

Concept
Sunset desert 

Art Constraint
Only 3 colors 

Description (optional)

 Create

New Challenge



 Change pic 
Recommended size: 1080x1080 (1:1)

Title
Solitude

Concept
Sunset desert 

Art Constraint
Only 3 colors 

Profile

alecap
Questa è la bio di alecap, non è molto, ma è una bio onesta

created: 5 completed: 3

Solitude
Sunset desert • Only 3 colors

Challenge in progress
Sunken city • All in greyscale

Profile

alecap
Questa è la bio di alecap, non è molto, ma è una bio onesta

created: 5 completed: 3

Edit Profile

alecap
Questa è la bio di alecap, non è molto, ma è una bio onesta

Logout

Are you sure you want to logout?

Cancel Logout

Artists

Search artists

- alecap**
created: 5 • completed: 3
- robcap**
created: 3 • completed: 3
- test3**
created: 2 • completed: 1
- test1**
created: 2 • completed: 0
- Fedeee21**
created: 2 • completed: 1
- supermario**
created: 1 • completed: 1
- teacap**
created: 1 • completed: 1

Artists

Search artists

No artists found
Try adjusting your search.

Artists

Search artists

- Fedeee21**
created: 2 • completed: 1
- venafe**
created: 1 • completed: 0

Profile



test1
Una bio di esempio

created: 0 completed: 0



This artist has no challenge yet
Come back later to see their challenges.

Profile



Fedeee21

created: 3 completed: 2

F Alpi Nature
Lucid dream • Time limit 1hr

F The Other Side
Upside-down World • Use three-point perspective



F Fantasy world
Enchanted forest • Only 3 colors

Fantasy world



F **Fedeee21**

Concept
Enchanted forest

Art constraint
Only 3 colors

Home

Search challenge

Solitude
Sunset desert • Only 3 colors

The Other Side
Upside-down World • Use three-point

Exit

Are you sure you want to exit the app?

Cancel **Exit**



+

Home **Artists** **Profile**

3.4. Database e memorizzazione dati

Anche la versione Flutter di Inkspire utilizza Supabase per la gestione di database PostgreSQL, autenticazione e storage immagini.

3.4.1. Modello relazionale

Il modello relazionale è identico alla versione Android:

- relazione 1:N tra user_profile e challenge;
- tabelle concept e art_constraint per suggerimenti e generazione casuale;
- viste challenge_vw e user_profile_vw per servire UI e liste senza join ripetute.

3.4.2. Pacchetto supabase_flutter

Per database, autenticazione e storage si usa il pacchetto ufficiale supabase_flutter, che integra in un'unica libreria i moduli Auth, Database (PostgREST) e Storage, semplificando l'integrazione nel progetto Flutter.

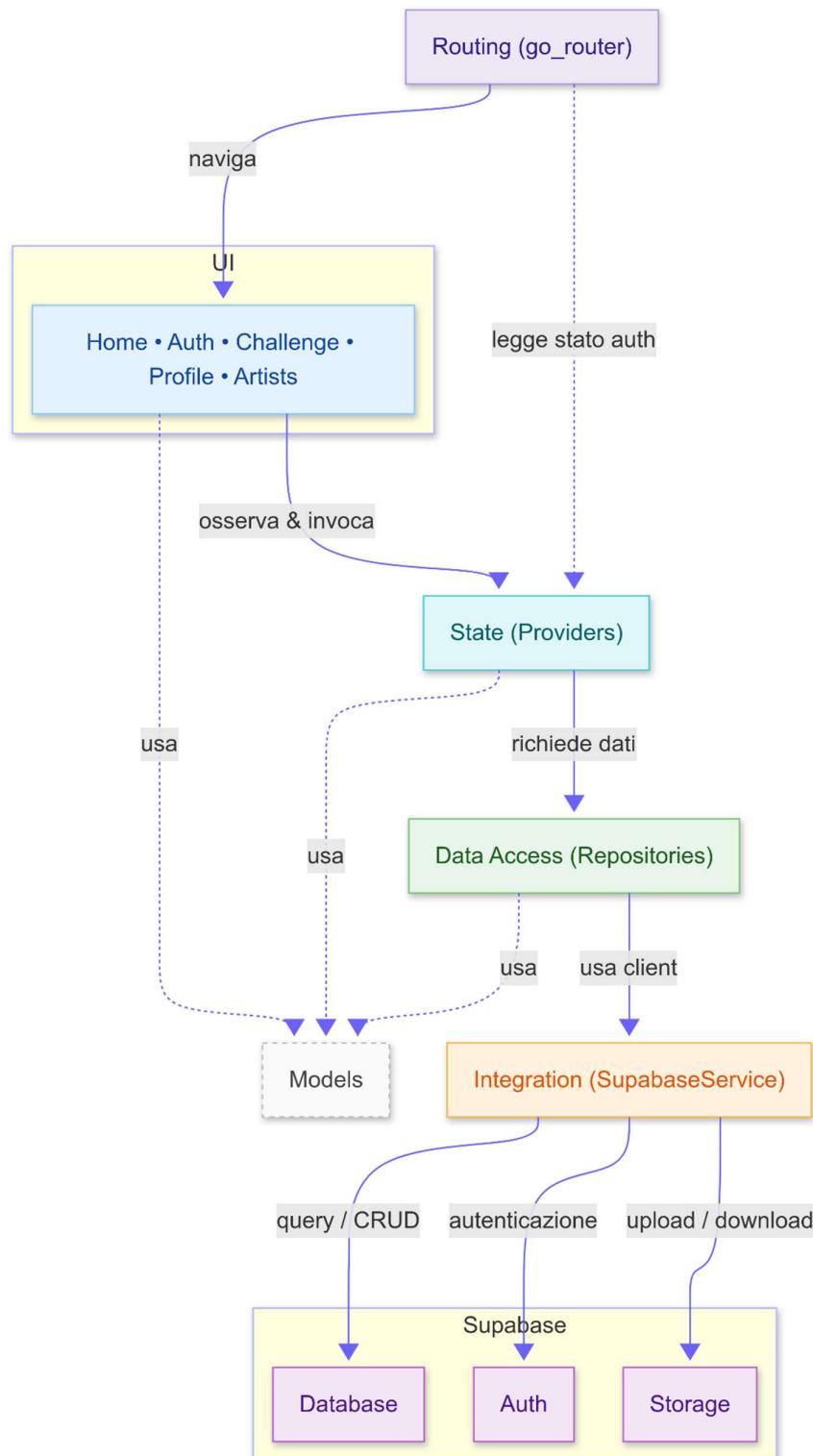
3.4.3. Policies di sicurezza

Le RLS sono le stesse utilizzate nel progetto Android e l'app si limita a esporre o nascondere pulsanti in base allo stato di autenticazione.

3.5. Sviluppo

3.5.1. Architettura e panoramica

Il seguente diagramma riassume i livelli dell'architettura e i collegamenti tra UI, stato nei Providers, dati nei Repositories, servizio d'integrazione e Supabase.



Tale diagramma ha lo scopo di illustrare il flusso principale delle azioni, dall’interfaccia fino a Supabase e ritorno. I livelli architetturali sono i seguenti:

- **UI** (Features & Widgets): le schermate dell’app e i componenti riutilizzabili (challenge_card) mostrano i dati e reagiscono ai tocchi dell’utente. Non fanno query dirette, ma osservano i Provider e invocano azioni esposte da loro.
- **Routing** (go_router): decide dove navigare e applica i redirect protetti: la schermata splash manda a Login o Home leggendo lo stato di autenticazione esposto dai Provider.
- **Providers** (Riverpod): i Provider orchestrano operazioni chiedendo ai Repository i dati necessari, memorizzano lo stato osservabile, invalidano la cache, e forniscono dati reattivi alla UI tramite AsyncValue.
- **Repositories** (accesso dati): contengono le funzioni per accedere e compiere operazioni sul database Supabase. Qui si applicano filtri, ordinamenti e si traducono le risposte in Model.
- **SupabaseService** (integrazione): inizializza il client e fornisce un punto unico di accesso a Auth, Database e Storage. Viene utilizzato dai repository per tutte le chiamate.
- **Models**: mappano 1:1 tabelle e viste del database con parsing null-safe, così la UI non va in errore se qualche campo è mancante.
- **Supabase** (Cloud): Autenticazione, Database e Storage con RLS lato server. La UI mostra o nasconde le azioni in base all’utente loggato, ma il controllo effettivo dei permessi viene sempre applicato dal database tramite le RLS.

Rispetto ad Android non ci sono Activities, Fragments e Adapters, ma in Flutter tutto è un **Widget** con lo stato gestito dai providers Riverpod.

Lo sviluppo è stato incrementale: setup iniziale del progetto e di Supabase, poi autenticazione, Home con ricerca, dettaglio e modifica delle challenge, Profili e sezione Artists.

3.5.2. Classi Widget in Flutter

Prima di passare alla spiegazione dettagliata dei file del progetto, è importante introdurre il concetto di Widget: tutto in Flutter è un widget, dai pulsanti, ai testi, ai layout e persino l'app stessa.

Un widget è una classe Dart che estende una delle due grandi famiglie:

- **StatelessWidget**: widget “statico”, che non cambia lo stato interno.
- **StatefulWidget**: widget con stato che può cambiare.

La struttura di un widget possiede tipicamente:

- un **costruttore** con i parametri necessari;
- un **metodo build(BuildContext context)** obbligatorio, che restituisce un altro widget.

Di seguito due esempi base di widgets, uno stateless e uno stateful.

Lo StatelessWidget ha una sola classe, il widget:

```
class MyButton extends StatelessWidget {
  final String label;
  const MyButton({super.key, required this.label});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: () {},
      child: Text(label),
    ); // ElevatedButton
  }
}
```

Lo StatefulWidget ha invece due classi:

- il **widget** (Counter)
- lo **State** (_CounterState).

Lo stato (_count) vive nello State, e quando varia in setState(), viene ricostruito il build:

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

  @override
  State<Counter> createState() => _CounterState();
}

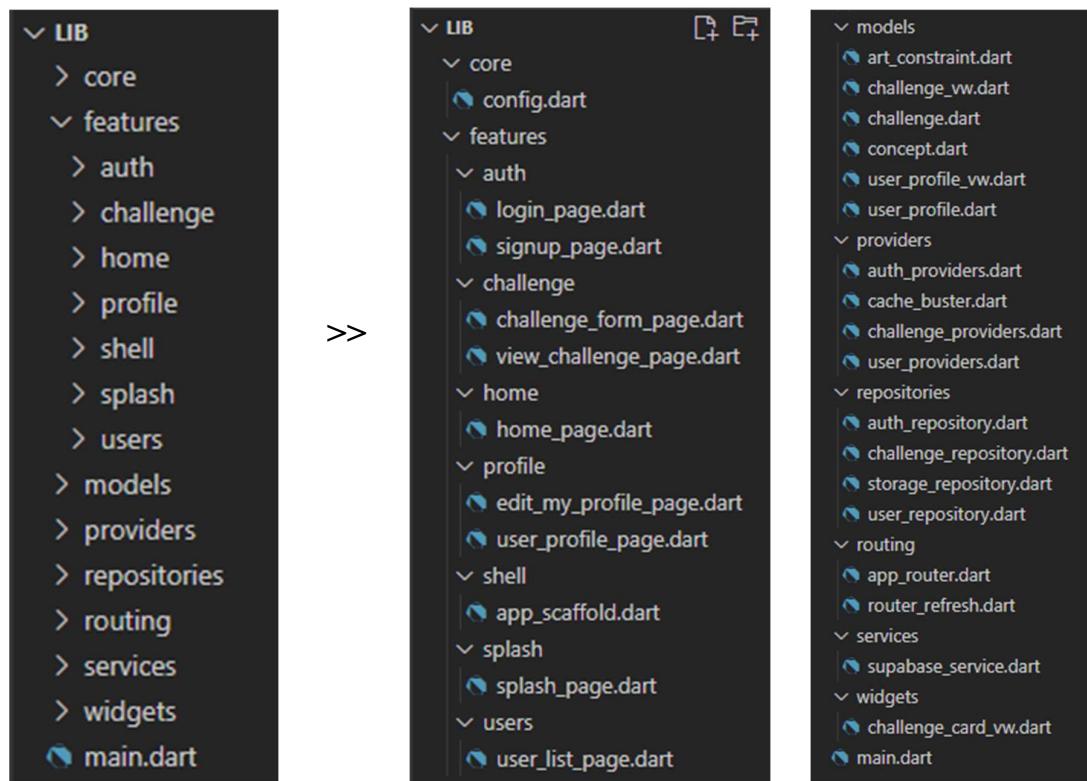
class _CounterState extends State<Counter> {
  int _count = 0;

  void _increment() {
    setState(() {
      _count++; // quando lo stato _count varia, viene ricostruito il build.
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Value: $_count'),
        ElevatedButton(onPressed: _increment, child: const Text('Count')),
      ],
    ); // Column
  }
}
```

3.5.3. Struttura del progetto

Struttura delle cartelle e dei file del progetto Flutter sviluppato in Visual Studio Code:



Il progetto è organizzato in cartelle e file con responsabilità chiare, in modo analogo alla suddivisione Android ma coerente col framework Flutter:

- **pubspec.yaml**: file di configurazione principale di Flutter che definisce dipendenze (es. supabase_flutter, flutter_riverpod, go_router, cached_network_image, image_picker), eventuali asset e il nome dell'app.
- **build.gradle** (android/app): imposta i livelli SDK (\geq API 33 per compatibilità) e le configurazioni Android.
- **AndroidManifest.xml** (android/src/main): metadati dell'app e permessi di connessione internet.

Di seguito i packages presenti nella cartella ‘lib’ che raccoglie tutti i file dart del progetto:

- **core**: package contenente il file di configurazione config.dart con l’URL e la chiave pubblica di accesso a Supabase.

- **services**: package contenente il file supabase_service.dart di inizializzazione di Supabase all'avvio e accesso centralizzato a auth, db e storage.
- **models**: package contenente tutti i file dart dei models, le entità dati mappate 1:1 con le tabelle in Supabase, così da tradurre in modo sicuro (null-safe) i payload JSON in oggetti Dart e viceversa.
- **repositories**: package contenente tutti i file dei repository che incapsulano l'accesso a Supabase (Auth, PostgREST, Storage), comprese le query e le operazioni CRUD, le regole di filtro e ordinamento e, in generale, tutta la logica di accesso ai dati.
- **providers**: package contenente i file dei providers che gestiscono lo stato dell'app e la logica di presentazione tramite Riverpod: espongono alla UI dati pronti (liste, dettagli, profilo) e piccoli stati locali (come la ricerca), gestendo le chiamate ai repository senza conoscere la rappresentazione grafica.
- **routing**: package contenente i file di configurazione della navigazione con go_router. In essi sono definiti le rotte, i redirect protetti in base alla sessione e l'aggancio allo stream di autenticazione per instradare automaticamente dallo splash verso login o area autenticata, oltre a gestire pattern e priorità delle rotte.
- **features**: racchiude i package contenenti tutti i file delle schermate dell'app, organizzate per area funzionale (autenticazione, home, challenge, profili, community, splash). Ogni pagina è un widget “magro”: si limita a raccogliere input, mostrare dati esposti dai provider e reagire alle azioni utente, lasciando ai repositories le operazioni di rete.
- **widgets**: package contenente i file dei componenti grafici riutilizzabili (cards) che garantiscono coerenza visiva e riducono duplicazioni.
- **main.dart**: file di avvio dell'app in cui avviene l'inizializzazione Supabase e il montaggio ProviderScope e MaterialApp.router con tema Material 3 e router configurato.

In sintesi, il flusso delle dipendenze procede dalla UI verso i dati: le features osservano i providers, che a loro volta utilizzano i repositories, i quali si appoggiano ai services inizializzati con i parametri del core. Questa disposizione rende il progetto modulare, leggibile e facile da estendere.

3.5.4. Layout e UI

L’interfaccia è costruita con Material 3, mantiene una palette coerente e una struttura ripetibile: AppBar, NavigationBar in basso e FAB per creare una nuova challenge. Le pagine mostrano sempre lo stato dell’operazione, come caricamento, elenco vuoto o errore.

- **main.dart**: file di avvio dell’app in cui avviene l’inizializzazione Supabase e il montaggio ProviderScope e MaterialApp.router con tema Material 3 e router configurato.

```
void main() async {
    WidgetsFlutterBinding.ensureInitialized();
    await SupabaseService.init();
    runApp(const ProviderScope(child: InkspireApp()));
}

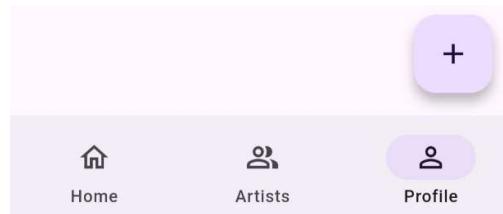
class InkspireApp extends ConsumerWidget {
    const InkspireApp({super.key});

    @override
    Widget build(BuildContext context, WidgetRef ref) {
        final router = ref.watch(goRouterProvider);
        return MaterialApp.router(
            title: 'Inkspire',
            theme: ThemeData(useMaterial3: true, colorSchemeSeed: Colors.deepPurple),
            routerConfig: router,
        ); // MaterialApp.router
    }
}
```

Azioni:

- Assicura l’inizializzazione dei **binding** dei Widget Flutter.
- Inizializza il **client Supabase** (*SupabaseService.init()*), collegandolo all’app tramite URL e Anon Key e scaricando i suoi moduli.
- Lancia l’app tramite la funzione *runApp()* che prende come argomento un Widget (*InkspireApp()*), dal quale partire per la gerarchia, e lo mostra sullo schermo. Contemporaneamente crea **ProviderScope**, un contenitore Riverpod che possiede la ‘mappa’ di tutti i suoi provider e il loro stato e li rende disponibili all’intera app.
- Estende InkspireApp come **ConsumerWidget**: variante Riverpod di StatelessWidget che accetta anche WidgetRef in build(), in modo da leggere i provider direttamente dentro a tale metodo.

- Legge dal container Riverpod il router (`ref.watch(goRouterProvider)`), un oggetto di tipo **GoRouter** già configurato con tutte le rotte e le regole di redirect. In questo modo la navigazione dipende dallo stato dell'autenticazione.
- Restituisce nel metodo build di `InkspireApp` un **MaterialApp.router**, il widget principale dell'app in cui vengono impostati tema grafico, titolo e navigazione (`routerConfig`) col router esterno, il quale decide dinamicamente quale schermata mostrare in base allo stato (login o home).
- **app_scaffold.dart** (features/shell): widget di layout che fornisce lo scheletro delle schermate principali dell'app: Home, Artists e Profile.



Azioni:

- Riceve un child e lo incapsula in uno **Scaffold**: contenitore base di una schermata che fornisce la struttura “standard” di un’app (AppBar, body, Bottom NavigationBar e FAB).
- Calcola quale **tab è attiva** (`_indexFromPath`) in base al percorso corrente del router (`GoRouterState.of(context)`).
- Mostra in alto un’**AppBar** il cui titolo varia automaticamente in base alla schermata corrente (`_titleFromPath`).
- Mostra in basso una **NavigationBar** con tre voci: Home, Artists e Profile, che cambiano rotta al tocco (`context.go(...)`).
- Aggiunge un **FAB** che apre la schermata di creazione nuova challenge (`context.push('/app/challenge/add')`).
- **challenge_card_vw.dart** (widgets): widget riutilizzabile che rappresenta la card di una *challenge* nelle liste in Home e Profile. La card mostra titolo, concept, vincolo artistico, autore ed eventuale immagine con caricamento progressivo.



Azioni:

- Riceve un oggetto **ChallengeVW** e un’eventuale callback **onTap**.
- Mostra l’immagine del risultato, se presente, con rapporto 4:3 e caricamento ottimizzato tramite **CachedNetworkImage** e placeholder con **CircularProgressIndicator**.
- Mostra un **ListTile** con titolo, concept, constraint e profile pic circolare. In particolare usa la foto profilo dell’autore, oppure l’iniziale dello username se l’immagine non è disponibile.
- Il widget è racchiuso in una **Card cliccabile** (InkWell) che invoca la callback **onTap**, utile per aprire la schermata di dettaglio della challenge.

3.5.5. Autenticazione, Storage e accesso al DB

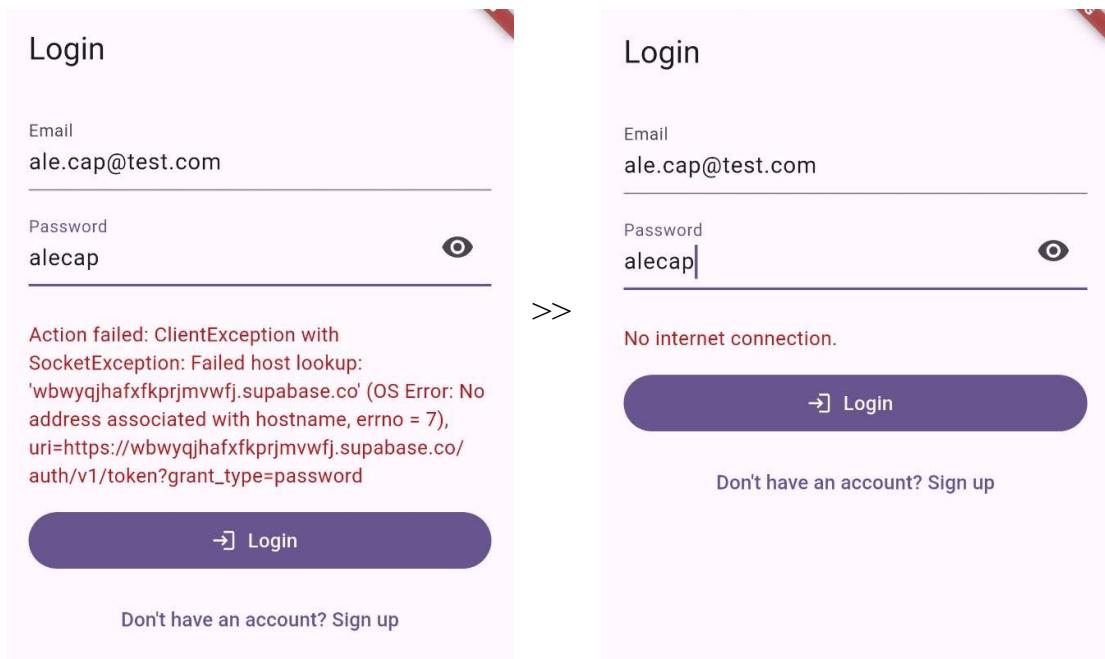
Supabase gestisce l’autenticazione dell’utente, l’accesso al database e lo storage.

- **supabase_service.dart** (services): file di servizio che centralizza la configurazione e l’accesso a Supabase, mettendo a disposizione getter statici che incapsulano i moduli principali:
 - client (generale per query su DB);
 - auth (modulo di autenticazione GoTrue);
 - storage (modulo per caricare e leggere file dai bucket)

Espone un metodo statico `init()` che inizializza Supabase con URL e Anon Key letti da `AppConfig`, impostando anche le opzioni di autenticazione (auto-refresh del token). La dicitura `Future<void>` indica che il metodo è asincrono e che restituisce

un Future, ossia un valore che arriverà in futuro e che potrà essere completato con successo o con errore.

- **auth_repository.dart** (repositories): gestisce autenticazione e profilo utente, con messaggi d'errore puliti per la UI. Azioni:
 - Fornisce currentUser, currentUserID e lo stream onAuthStateChanged.
 - **signIn()** effettua il login con email e password, intercettando eccezioni di Supabase o errori di rete (SocketException, TimeoutException) e convertendoli in messaggi user-friendly tramite l'eccezione personalizzata AppAuthException.
 - **signUp()** registra un nuovo utente: controlla prima l'unicità dello username, crea l'account su Supabase e aggiorna o crea il record corrispondente nella tabella user_profile.
 - Include metodi per **logout**, lettura e aggiornamento del profilo.
 - Il metodo privato **_mapAuthException()** traduce i messaggi tecnici restituiti da Supabase in stringhe leggibili per l'utente finale.
 - Le eccezioni vengono gestite centralmente tramite la classe **AppAuthException**, che incapsula solo il messaggio e implementa l'interfaccia Exception. Questo approccio consente di usare in modo semplice throw/catch nella logica applicativa e mostrare messaggi coerenti nell'interfaccia grafica.



- **storage_repository.dart** (repositories): carica le immagini su Supabase Storage e restituisce l’URL pubblico da salvare.
 - Espone un unico metodo `uploadPublic()`, che carica un file binario (`Uint8List`) in un bucket specificato e restituisce l’URL pubblico corrispondente, se il bucket è configurato come *public*.
 - Supporta l’opzione `upsert` per decidere se sovrascrivere file esistenti e in caso di conflitto e `upsert=true` tenta automaticamente l’aggiornamento tramite `updateBinary()`.

3.5.6. Model

I model rappresentano una copia fedele delle tabelle e delle viste del database. La conversione è tollerante ai campi mancanti e ai tipi opzionali, così la UI non va in errore se un dato non è presente. Tutti i model espongono **factory fromMap** per mappare le righe del DB:

- **challenge.dart**: mappa la tabella delle challenge, distingue tra campi obbligatori e facoltativi (nel costruttore) e prepara i dati per inserimento e modifica. In particolare:
 - la funzione `fromMap()` costruisce un oggetto Challenge a partire da una **mappa** `Map<String, dynamic>` ottenuta dal database (spesso derivata da un JSON), convertendo i valori dinamici in stringa e facendo il parse sicuro delle date.
 - `toInsert()` e `toUpdate()` restituiscono mappe con i campi da passare alle query di Supabase. La prima include solo i campi necessari all’inserimento, senza id e timestamp che sono gestiti dal database, mentre la seconda espone i campi aggiornabili per un’operazione di update.
- **challenge_vw.dart**: mappa la vista che unisce challenge e autore, usata per liste e per la schermata di dettaglio di una challenge.
- **user_profile.dart** e **user_profile_vw.dart**: rappresentano il profilo e la vista con i contatori per la sezione Artists.
- **concept.dart** e **art_constraint.dart**: mappano le tabelle di supporto per i suggerimenti generati casualmente.

3.5.7. Repository

L'accesso al database PostgREST è gestito dai repository, che incapsulano le chiamate a Supabase. Le query lavorano sulle viste per ridurre la duplicazione delle join e applicano i filtri prima dell'ordinamento.

I repository **forniscono metodi ad alto livello** per elenchi, dettagli e operazioni CRUD, nascondendo la logica delle richieste HTTP e restituendo direttamente i model dell'app. Oltre all'auth_repository.dart e allo storage_repository.dart ci sono:

- **challenge_repository.dart**: fornisce l'elenco con ricerca sulla vista aggregata, il dettaglio per id, l'elenco per autore e le operazioni CRUD.
- **user_repository.dart**: espone lettura e aggiornamento del profilo personale e la lista di tutti gli utenti con i relativi contatori.

Esempio di metodo nel ChallengeRepository:

```
Future<List<ChallengeVW>> listAllVW({String? search}) async {
    final s = search?.trim();
    final base = _db.from('challenge_vw').select();

    final filtered = (s != null && s.isNotEmpty)
        ? base.or(
            'title.ilike.%$s%,concept.ilike.%$s%,art_constraint.ilike.%$s%,username.ilike.%$s%',
        )
        : base;

    final res = await filtered.order('updated_at', ascending: false);
    return (res as List)
        .map((e) => ChallengeVW.fromMap(e as Map<String, dynamic>))
        .toList();
}
```

Questa funzione, listAllVW, è asincrona restituisce tramite Future una lista di oggetti ChallengeVW. Quindi:

- il risultato non è immediato, ma arriverà dopo che l'operazione asincrona è completata, in questo caso la query al DB.
- la funzione accetta un parametro opzionale e nullable chiamato *search*, che serve per filtrare i risultati.

3.5.8. Provider

I provider Riverpod sono gli intermediari tra i repository e l’interfaccia: incapsulano la logica e i dati e restituiscono stati e operazioni tipizzate.

Soltamente si utilizzano i Provider per le dipendenze, gli StateProvider per gestire variabili mutabili di stato e gli StreamProvider o i FutureProvider per flussi asincroni, permettendo alla UI di osservare e aggiornare i dati senza interagire direttamente con Supabase.

- **auth_providers.dart**: questi Providers espongono AuthRepository e AuthState, ossia l’utente corrente e lo stream dei suoi cambi di sessione, che il router usa per decidere cosa mostrare in maniera dinamica.
- **challenge_providers.dart**: comprendono il Provider per il ChallengeRepository, uno StateProvider per il testo di ricerca e dei FutureProvider che restituiscono le liste e i dettagli dalla vista challenge_vw.

```
final challengeRepositoryProvider = Provider<ChallengeRepository>((ref) => ChallengeRepository());
final challengeSearchProvider = StateProvider<String>((_) => '');

final challengeListVWProvider = FutureProvider.autoDispose<List<ChallengeVW>>((ref) async {
  final repo = ref.watch(challengeRepositoryProvider);
  final q = ref.watch(challengeSearchProvider);
  return repo.listAllVW(search: q.isEmpty ? null : q);
});
```

Prendendo come esempio la prima parte del codice, si hanno le seguenti azioni:

- Definizione di *challengeRepositoryProvider*, un **Provider** di Riverpod assegnato ad una variabile globale, che **espone un’istanza** del ChallengeRepository: quando lo si utilizza con *ref.watch()* si ottiene un oggetto di tipo ChallengeRepository.
- Definizione di *challengeSearchProvider*, uno **StateProvider** di Riverpod che gestisce una **variabile di stato** di tipo String, la quale viene inizializzata con il valore ‘’ di una stringa vuota. In pratica questo provider è un contenitore reattivo per il testo di ricerca delle challenge.
- Definizione di *challengeListVWProvider*, un **FutureProvider autoDispose** che **fornisce asincronamente la lista** di tutte le challenge List<ChallengeVW>. Al suo interno legge i precedenti due providers del repository e della search (*repo* e *q*) e ogni volta che la query cambia si ricalcola e la passa alla funzione *repo.listAllVW(...)*. Con autoDispose, quando nessun widget lo osserva più, libera risorse e cache.

- **user_providers.dart**: questi Providers espongono lo UserRepository, l'id e il profilo dell'utente corrente e la lista utenti filtrata da uno StateProvider<String> di ricerca.

Relazione tra StateProvider e FutureProvider

In particolare gli StateProvider per la ricerca (challenge e user) permettono di aggiornare lo stato quando l'utente digita un testo: questo triggerà automaticamente i FutureProvider collegati, che ricaricano l'elenco filtrato.

3.5.9. Routing e Navigazione

La navigazione è centralizzata con **GoRouter** e reagisce allo stato di autenticazione.

- **app_router.dart** incapsula tutta la configurazione di GoRouter:

```
final goRouterProvider = Provider<GoRouter>((ref) {
  final authRepo = ref.read(authRepositoryProvider);

  // refresha il router ad ogni variazione di auth
  final refresh = GoRouterRefreshStream(authRepo.onAuthStateChange);
  ref.onDispose(refresh.dispose);

  return GoRouter(
    initialLocation: '/splash',
    refreshListenable: refresh,
    > redirect: (context, state) { ... },
    > routes: [ ... ],
  ); // GoRouter
}); // Provider
```

- L'oggetto **GoRouter** è esposto con un Provider Riverpod (*goRouterProvider*), in modo da essere letto in tutta l'app attraverso l'istruzione *ref.read(goRouterProvider)*.
- La funzione lambda *(ref) { ... }* è la **factory** che crea il router. Al suo interno viene recuperato il repository di autenticazione *authRepo* e viene creato un *GoRouterRefreshStream*, collegato allo **stream di autenticazione** di Supabase (*onAuthStateChange*): ogni volta che cambia lo stato di login/logout, il router viene notificato e riesegue la funzione di redirect.
- L'istanza di GoRouter restituita ha i seguenti parametri chiave:
 1. **initialLocation**: rotta con cui parte l'app al primo avvio, in questo caso la schermata Splash, che funge da smistamento iniziale tra Home e Login.
 2. **refreshListenable**: oggetto che GoRouter osserva per sapere quando deve rivalutare i redirect, in questo caso è il *GoRouterRefreshStream*.

3. **redirect**(*context, state*): funzione centrale che decide, dato il percorso richiesto (*state.uri.path*) e lo stato utente (*isAuthenticated*), in quale schermata reindirizzare.
4. **routes**: elenco di tutte le rotte disponibili nell'app, organizzate con GoRoute per le pagine singole e con ShellRoute per raggruppare più pagine con layout comune dell'AppScaffold (Home, Artists, Profile). Tutte le **rotte** sono **statiche**, con un percorso fisso, ad eccezione di quelle per le schermate di dettaglio e modifica challenge e per quella del profilo di un utente, che sono **dinamiche**, con un parametro variabile nell'URL che dipende dall'id.

```
// Alcune delle quali sono rotte dinamiche con regex (\d+) = '1+ cifre per il valore dell'id'
GoRoute(
  path: '/app/challenge/add',
  builder: (_, __) => const ChallengeFormPage(),
), // GoRoute
GoRoute(
  path: '/app/challenge/:id(\d+)',
  builder: (_, st) => ViewChallengePage(id: int.parse(st.pathParameters['id']!)),
), // GoRoute
```

- **router_refresh.dart** contiene la classe *GoRouterRefreshStream()*, un adattatore che converte uno **Stream** in un **ChangeNotifier** ascoltabile da GoRouter. Questo pattern è necessario perché GoRouter si aggiorna quando un Listenable notifica dei cambiamenti.

```
/// Converte uno Stream in un Listenable per refresh del router.
class GoRouterRefreshStream extends ChangeNotifier {
  GoRouterRefreshStream(Stream<dynamic> stream) {
    _sub = stream.asBroadcastStream().listen((_) => notifyListeners());
  }
  late final StreamSubscription<dynamic> _sub;
  @override
  void dispose() {
    _sub.cancel();
    super.dispose();
  }
}
```

Conversione da Stream a Listenable

Uno **Stream** è un oggetto che rappresenta una sequenza di **eventi asincroni** nel tempo, cioè emette dati man mano che sono disponibili, e che può essere “ascoltato” tramite *await for ()* oppure *stream.listen((value) { ... })*.

In questo caso lo stream è `authRepo.onAuthStateChange`, che emette un nuovo evento ogni volta che cambia la sessione utente.

L'oggetto `GoRouterRefreshStream` si sottoscrive a questo stream e lo espone come **Listenable**, oggetto che notifica dei cambiamenti ai suoi ascoltatori, permettendo così al router di aggiornare automaticamente le rotte in base allo stato di autenticazione.

3.5.10. Gestione cache e refresh

Per mantenere la UI sempre sincronizzata con i dati, l'app invalida e ricarica i provider Riverpod dopo ogni operazione che modifica lo stato, ossia dopo login, logout e vari salvataggi nelle schermate di dettaglio e modifica delle challenge e del profilo.

- **cache_buster.dart**: contiene degli helper per invalidare i providers di utente e challenge al logout o dopo un update.

```
void invalidateAfterAuthChange(WidgetRef ref) {
    ref.invalidate(currentUserProvider);
    ref.invalidate(myProfileProvider);
    ref.invalidate(userListProvider);
    ref.invalidate(challengeListVWProvider);
}

void invalidateProfileCaches(WidgetRef ref, String userId) {
    ref.invalidate(userProfileByIdProvider(userId));
    ref.invalidate(userProfileVWByIdProvider(userId));
    ref.invalidate(challengesByUserProvider(userId));
}

void invalidateChallengeCaches(WidgetRef ref, {int? challengeId}) {
    ref.invalidate(challengeListVWProvider);
    if (challengeId != null) ref.invalidate(challengeByIdProvider(challengeId));
}
```

Applicazioni del cache buster:

- Salvataggi: dopo aver creato o modificato una challenge o il profilo utente, vengono chiamati `ref.invalidate(...)` e `await ref.read(..future)` per forzare il **refresh** dei provider e **ricaricare** subito la UI. In questo modo i widget leggono sempre valori aggiornati e non mostrano dati obsoleti provenienti dalla cache locale.

- Logout: la funzione `invalidateUserScopedCaches(ref)` invalida i provider legati all'utente e alle challenge, garantendo un contesto “pulito” al login successivo.
- Immagini: per le immagini viene invece applicato un sistema di cache busting sugli URL (`?v=timestamp`), così da forzare il caricamento della versione aggiornata.

Esempio di cache busting in challenge_form_page.dart:

```
// Upload immagine se scelta
if (_pickedBytes != null) {
    final storage = StorageRepository();
    final ts = DateTime.now().millisecondsSinceEpoch;
    final path = 'challenge_${challengeId}_${ts}.jpg';
    var url = await storage.uploadPublic(
        bucket: 'challenge-pics',
        path: path,
        bytes: _pickedBytes!,
        contentType: 'image/jpeg',
    );
    if (url != null) {
        url = '$url?v=${ts}'; // cache buster per forzare il refresh della cache
        model = model.copyWith(resultPic: url);
        await repo.update(challengeId, model);
    }
}

// Invalidazione centralizzata delle cache interessate
final authorId = _authorUserId ?? uid!;
invalidateChallengeCaches(ref, challengeId: challengeId);
invalidateProfileCaches(ref, authorId);

// Attesa dei reload per rientrare con dati riaggiornati
await Future.wait([
    ref.read(challengeListVWProvider.future), // Home (lista globale challenges)
    ref.read(challengesByUserProvider(authorId).future), // Lista challenges nel profilo
    ref.read(userProfileVWByIdProvider(authorId).future), // Contatori nel profilo
]);
```

3.5.11. Schermate

Le schermate nel package /features sono organizzate per area funzionale come nel progetto Android: auth, challenge, profile, community.

Ogni pagina:

- legge i dati dai **providers** Riverpod;
- invoca i **repository** per le operazioni;
- gestisce lo **stato asincrono** (loading, error, data) con *AsyncValue.when*;
- in caso di modifica o salvataggio, chiama l'**invalidazione cache** per forzare il refresh.

Tutte le schermate condividono la stessa struttura, solo la pagina di Splash è un'eccezione, poiché funge da passaggio tecnico per il reindirizzamento iniziale.

Esempio di struttura in challenge_form_page.dart:

```
// In EDIT: carica i dati dalla view (ChallengeVW)
final asyncC = ref.watch(challengeByIdProvider(widget.challengeId!));
return asyncC.when(
  data: (c) {
    if (c == null) {
      return const Scaffold(body: Center(child: Text('Challenge not found')));
    }
    // inizializza i controller una sola volta
    if (!_initializedFromData) { ... }
    return _buildScaffold(context, child: _buildForm(isEdit: true));
  },
  loading: () => const Scaffold(body: Center(child: CircularProgressIndicator())),
  error: (e, _) => Scaffold(appBar: AppBar(), body: Center(child: Text('Error: $e'))),
);
}
```

- gestione stato asincrono: *asyncC* è un **AsyncValue<Challenge?>** restituito da *challengeByIdProvider*;
- data: mostra il form precompilato con i dati della challenge quando il provider ha finito di caricare.
- loading: mostra un indicatore di caricamento mentre la challenge viene fetchata;
- error: mostra un messaggio d'errore se il fetch fallisce.

4. Problematiche e considerazioni finali

Durante lo sviluppo dell'applicazione Inkspire sono emerse alcune problematiche tecniche che hanno richiesto interventi puntuali per garantire la stabilità e il corretto funzionamento del sistema.

In fase iniziale dello sviluppo Android si sono riscontrati conflitti tra le librerie di Jetpack Compose e quelle **AppCompat** utilizzate per le activity e i fragment. Per evitare incompatibilità, è stato deciso di rimuovere completamente i riferimenti a Compose dai file Gradle, mantenendo un approccio tradizionale basato su xml per la gestione dei layout.

Un ulteriore ostacolo ha riguardato le **operazioni CRUD** sul database Supabase: in assenza di policy esplicite, le query venivano bloccate per mancanza di permessi. Il problema è stato risolto configurando correttamente le Row Level Security (RLS) e le relative policy per ogni tabella coinvolta.

Per quanto riguarda la **navigazione**, si è optato per un'impostazione che ruota attorno ai fragment principali (Home, Artists, Profile), così da ridurre problematiche legate allo stack di navigazione. Le operazioni come la creazione o modifica di una challenge terminano riportando l'utente alla schermata principale, evitando stacking eccessivo di fragment secondari.

Durante il debugging è stato fatto ampio uso di **Log**, in particolare per verificare il contenuto delle richieste inviate al backend, come ad esempio i payload JSON nella fase di aggiornamento delle challenge.

L'app è stata condivisa con alcuni utenti esterni, che hanno effettuato test reali registrandosi e utilizzando le principali funzionalità. Il feedback ricevuto ha contribuito a identificare ulteriori aree di miglioramento.

Tra i possibili sviluppi futuri si segnalano:

1. Ottimizzazione della navigazione per evitare accumulo di fragment nello stack.
2. Gestione della dimensione delle immagini caricate: attualmente non sono imposti limiti, il che può rallentare il caricamento (soprattutto per immagini pesanti). Potrebbero essere integrate funzioni di **ridimensionamento** o compressione lato client.
3. Possibilità di **rendere private** le proprie challenge, visibili solo all'autore.
4. Aggiunta della funzionalità di **like** per aumentare l'interazione tra utenti e valorizzare i contenuti creati.

5. Refactoring del codice per ridurre la ridondanza, ad esempio introducendo fragment base condivisi tra schermate simili.

Nonostante le criticità iniziali, si conferma che l'applicazione è **funzionante e stabile**. Tutte le funzionalità descritte nell'analisi dei requisiti sono state implementate correttamente, garantendo all'utente la possibilità di registrarsi, creare e completare challenge artistiche, gestire il proprio profilo, visualizzare i contenuti della community e navigare fluidamente tra le sezioni principali dell'app.

Bibliografia

Di seguito viene riportato tutto il materiale utilizzato, con documentazione, risorse e tutorial utili per lo sviluppo:

App structure (CRUD, MVVM, Activity e Fragment, Navigazione)

- The Notes App (MVVM + ROOM Database, CRUD operations)
<https://www.youtube.com/watch?v=zCDB-OqOzfY>
- Types of Login and Signup in Android Studio (Auth + db)
https://www.youtube.com/watch?v=fStYN_AmEc
- Navigation Component
<https://developer.android.com/guide/navigation>

Ktor e Supabase

- Supabase Project creation and implementation on Android Studio
https://www.youtube.com/watch?v=_iXUVJ6HTHU
- Supabase Auth with Email and Google + Login UI + Credentials Manager
<https://www.youtube.com/watch?v=ZgYvexniGDA>
- Supabase plugins and dependencies for Android Studio
<https://supabase.com/docs/reference/kotlin/installing>
- Ktor for API Services
<https://ktor.io/docs/client-engines.html>
- Android Internet Connection (URL + public key)
<https://developer.android.com/develop/connectivity/network-ops/connecting>
- Supabase Auth e GoTrue Auth Service
<https://supabase.com/docs/guides/auth/architecture>

Editor online per i diagrammi

- PlantUML
<https://www.plantuml.com/>

- MermaidChart
<https://www.mermaidchart.com/>

Flutter

- Start building Flutter Android apps on Windows
<https://docs.flutter.dev/get-started/install/windows/mobile>

Testing

- Testing per Android (Concetti fondamentali e Cosa Testare)
<https://developer.android.com/training/testing/fundamentals?hl=it>
<https://developer.android.com/training/testing/fundamentals/what-to-test?hl=it>
- Test Locali
<https://developer.android.com/training/testing/local-tests?hl=it>
- Test strumentati
<https://developer.android.com/training/testing/instrumented-tests?hl=it>
- Espresso
<https://developer.android.com/training/testing/espresso?hl=it>