Sviluppo mobile con Flutter: introduzione a Dart e architettura delle App

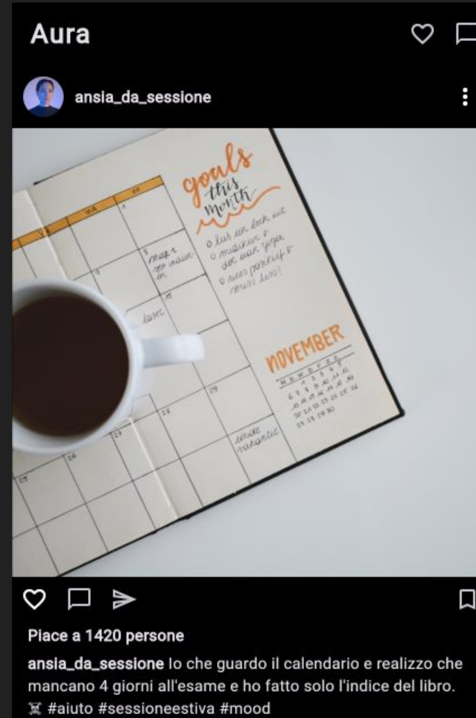
Marco Abbadini

26/05/2026

Cosa facciamo oggi?

Vedremo come sviluppare una nostra app: ✨ Aura ✨

L'app sarà un clone di Instagram, vedremo come utilizzare gli strumenti che Flutter ci mette a disposizione per creare pagine della nostra app, gestire la logica e i dati 😼

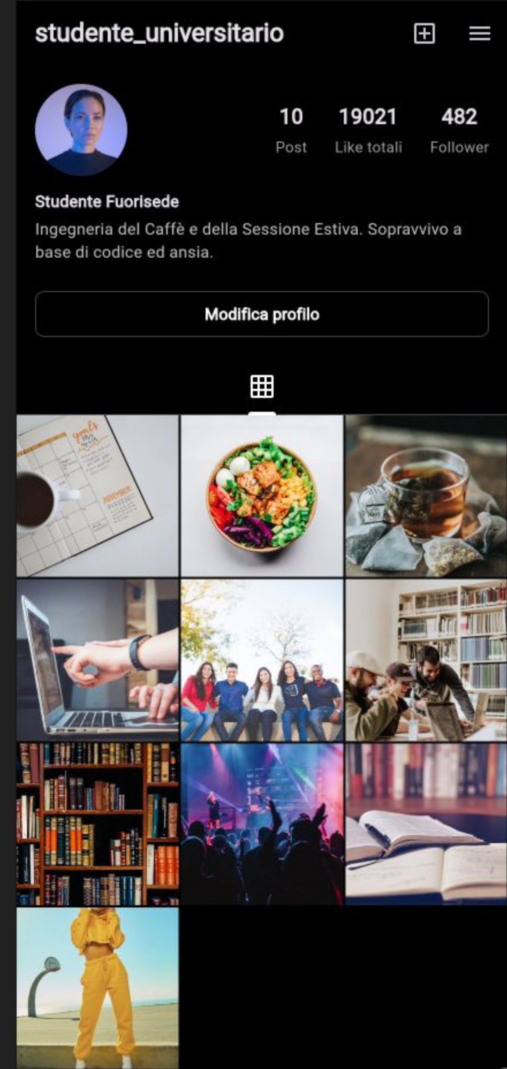


Aura

Ci concentreremo sulla realizzazione della pagine di Feed e di Profilo.

Il Feed è costituito da una lista di Post

Il Profilo mostrerà tutti i Post di un utente



Flutter

Flutter è un framework sviluppato da Google che consente di sviluppare applicazioni mobile che siano compatibili con tutte le principali piattaforme: Android/iOs e cloud

Per semplicità la lezione mostra lo sviluppo dell'app nella versione cloud, il codice per le versioni mobile rimane comunque lo stesso.

Flutter si basa sul linguaggio **Dart** per lo sviluppo delle applicazioni

Sviluppo

Vediamo di seguito una veloce e breve introduzione a Dart: solo quello che è necessario sapere per lo sviluppo della nostra applicazione

Successivamente vedremo come definire l’architettura della nostra applicazione

Potete seguire lo sviluppo clonando il repo github a questo link: https://github.com/marcoabbadini/unibg-flutter-seminar

Seguire lo sviluppo

Non abbiamo il tempo per scrivere manualmente tutte le parti dell'applicazione: per seguire la lezione il repo Github è organizzato in branch:

1. `01-setup`: contiene il setup iniziale dell'app (il progetto vuoto, da cui partire)
2. `02-feed-ui`: introduce la grafica del feed
3. `03-refactor`: rende il progetto modulare
4. `04-likes`: introduce la logica per la gestione dei like
5. `05-state`: introduce il concetto di stato e la sua gestione
6. `06-profile`: definisce la pagine del profilo
7. `07-final`: rifinisce il progetto (versione più aggiornata)

Seguire lo sviluppo (2)

Come si utilizza?

All’inizio potete clonare il progetto nel branch 01:

```
git clone -b 01-setup
https://github.com/marcoabbadini/unibg-flutter-seminar.git

cd unibg-flutter-seminar

flutter pub get
```

Potrete passare alla fase successiva facendo il checkout al branch successivo

Seguire lo sviluppo (3)

```
# 1. Annulla le modifiche locali sovrascrivendole (evita conflitti di merge)
git reset --hard
# 2. Scarica le informazioni sui nuovi branch dal server
git fetch origin
# 3. Salta al codice della fase successiva
git checkout 02-feed-ui
# 4. Sincronizza i pacchetti di Flutter per sicurezza
flutter pub get
```

Dart (1)

```
// Inferenza del tipo
var nome = 'Aura App';
// Tipizzazione esplicita (fortemente consigliata per le proprietà delle classi)
String ateneo = 'Università';
int codiceEsame = 1042;
double mediaVoti = 28.5;
bool sessioneAttiva = true;
// Immutabilità a runtime: il valore è protetto dopo l'assegnazione
final DateTime dataAccesso = DateTime.now();
// Immutabilità a tempo di compilazione: costante assoluta
const int colonneGriglia = 3;
```

Dart (2)

```
class AuraPost {
  final String username;
  final String postImage;
  int likes;
  bool isLiked;

 // Costruttore con parametri nominali e obbligatori (required)
  AuraPost({
    required this.username,
    required this.postImage,
    required this.likes,
    required this.isLiked,
  });
}
```

Dart (3)

void main() {

```
// Istanza della classe: l'ordine dei parametri non conta grazie ai nomi espliciti var
nuovoPost = AuraPost( username: 'studente_fuorisede',
            postImage: 'https://link-immagine.com/foto.jpg',
            likes: 42,
            isLiked: false, );
}
```

Dart (4)

```
// La funzione restituisce un Future vuoto (void) al termine dell'operazione
Future<void> caricaDatiDallServer() async {
  print('Avvio della richiesta di rete...');

  // Ferma l'esecuzione di questa funzione finché il timer non è scaduto,
  // senza bloccare l'intera applicazione nel frattempo
  await Future.delayed(const Duration(seconds: 2));
  print('Dati ricevuti con successo.');
}
```

Dart (5)

```
Future<void> fetchPosts() async {

final Uri url = Uri.parse('https://api.myurasocial.com/v1/posts');

final response = await http.get(url); // 1. Richiesta di rete asincrona

if (response.statusCode == 200) { // 2. Decodifica del testo JSON grezzo

final List<dynamic> dataGrezza = json.decode(response.body);

// 3. Conversione nei nostri modelli dati

posts = dataGrezza.map((json) => AuraPost( username: json['author_name'],
postImage: json['image_url'], likes: json['likes_count'], isLiked:
json['user_has_liked'] ?? false, )).toList(); }

}
```

Dart (6)

```
void calcoliLista() {
  List<int> voti = [24, 28, 30, 18];
 // Filtrare una lista (es. prende solo i voti superiori o uguali a 25)
  var votiAlti = voti.where((voto) => voto >= 25);// Risultato: (28, 30)
 // Accumulare i valori (es. calcolare la somma totale dei voti)
 // Il primo parametro (0) è il valore iniziale dell'accumulatore
  int sommaVoti = voti.fold(0, (accumulatore, voto) => accumulatore + voto);
}
```

`main.dart`

- `main()`: è l’entry point da cui parte l’intera esecuzione del codice Dart
- `runApp()`: avvia a tutto schermo il Widget principale dell’applicazione
- `MyApp`: definisce le configurazioni globali dell’app
  - tema scuro
  - colori principali
  - schermata iniziale

main.dart (2)

```
void main() {
  runApp(const MyApp()); // Avvia l'applicazione
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Aura',
      theme: ThemeData.dark(), // Configurazione del tema scuro globale
      home: const MainScreen(), // La prima pagina che l'utente vede
    );
  }
}
```

MainScreen e barra di navigazione (1)

Scaffold: scheletro della pagina principale

NavigationBar: gestisce l'interfaccia grafica dei pulsanti in basso (es. Feed e Profilo)

IndexedStack: Mantiene in memoria lo stato delle pagine quando l'utente passa da una scheda all'altra, evitando di ricaricare i dati da zero

MainScreen e barra di navigazione (2)

```
class MainScreen extends StatefulWidget {
  const MainScreen({super.key});
  @override
  State<MainScreen> createState() => _MainScreenState();
}
class _MainScreenState extends State<MainScreen> {
  int _currentIndex = 0;  // Traccia l'indice della pagina attiva
  @override
  Widget build(BuildContext context) {
    return Scaffold(
   // Cambia dinamicamente la pagina visibile in base all'indice
      body: IndexedStack(
        index: _currentIndex,
        //...
```

MainScreen e barra di navigazione (3)

```
    //...
      children: [ FeedPage(), ProfiloPage() ],
    ),
    bottomNavigationBar: NavigationBar(
      selectedIndex: _currentIndex,
      onDestinationSelected: (index) => setState(() => _currentIndex = index),
      destinations: const [
        NavigationDestination(icon: Icon(Icons.home), label: 'Feed'),
        NavigationDestination(icon: Icon(Icons.person), label: 'Profilo'),
      ],
    ),
  );
 }
}
```

Widgets

Ogni cosa è un Widget

- La UI si crea combinando widget semplici e specializzati
- Gli widget non sono solo i componenti visivi ma anche i componenti di layout (es. Row, Padding), di stile (Theme) e di logica/posizionamento (Center, GestureDetector)
- Esempio: Il Padding è un widget a sè stante che avvolge il widget Text

Stateful e Stateless Widgets

Stateless (immutabile, statico):

- UI fissa: una volta disegnato non cambia da solo
- Nessuna memoria interna: riceve i dati dall'esterno (costruttore)
- Loghi, testi, icone, …

Stateful (dinamico, reattivo):

- UI mutevole: si aggiorna in tempo reale con le azioni
- Possiede l'oggetto State
- Usa `SetState()` per il ridisegno istantaneo dello schermo

Architettura App (1)

Approccio monolitico locale (`setState()`)

- La logica dei dati e la grafica sono nello stesso file
- Il widget controlla se stesso, non considera il resto dell’applicazione
- Ogni widget è isolato
    - I dati restano dentro al widget e non possono essere utilizzati da altri

Architettura App (2)

Separation of Concerns (SoC)

- **UI (Presentazione)**: gestisce solo l'aspetto grafico dell'applicazione
- **Logica (Business Logic)**: si occupa solo di elaborare i dati, fare calcoli, gestire la rete
- Il codice diventa *ordinato*, *testabile* e *manutenibile*

Architettura App (3)

Flusso dei dati (Pattern Observer)

- AppState: contiene le informazioni e la logica per modificarle
- `notifyListeners()`: notifica quando i dati cambiano
- `ListenableBuilder`: consente ai widget grafici di ascoltare segnali di cambiamento e aggiornarsi di conseguenza
- *Esempio:* quando l'utente fa doppio tap su un post, il widget non aggiorna se stesso ma chiama una funzione di AppState. AppState modifica i dati e invia un segnale con `notifyListeners()`. I widget ricevono il segnale e si aggiornano di conseguenza.

Architettura App (4)

Modularità

- `lib/models/`: Strutture dati (AuraPost)
- `lib/state/`: Logica di business (AppState)
- `lib/widgets/`: Widget grafici (PostWidget)
- `lib/screens/`: Pagine dell’app (FeedPage, ProfiloPage)

Architettura App (5)

Aumentando le dimensioni dell’applicazione:

- Interfaccia utente disaccoppiata dai dati
- Da database locale a server reale: AppState
- Nessuna modifica alla grafica

Write Once, Run Everywhere

Un unico codice Dart può diventare un’app per cinque sistemi operativi diversi:

- App Android
- App iOS
- Sito web funzionante
- App per Windows/Mac

LET'S DO THIS!
peacock

DOMANDE?
imgflip.com