

Sport Mate

Relazione progetto LAM



Giovanni Spadaccini

giovanni.spadaccini3@studio.unibo.it

Indice

1	Introduzione	2
2	Scelta delle tecnologie	2
3	Features dell'applicazione	3
4	Struttura del codice sorgente	4
5	Scelte implementative	5
5.1	Dati	5
5.2	Provider	6
5.3	Pagine	7
5.3.1	Home	7
5.4	Registrazione e Accesso	8
5.5	Search	8
5.6	Aggiungi attività	10
5.7	Partecipa ad un'attività	11
5.8	Storia	11
5.9	Impostazioni	12

1 Introduzione

Sport Mate è un'applicazione progettata per organizzare e partecipare ad eventi sportivi nella propria zona.

L'obiettivo principale è quello di facilitare la connessione tra appassionati di sport, permettendo loro di creare, trovare e partecipare a varie attività sportive.

2 Scelta delle tecnologie

Dart è un buon linguaggio per diverse ragioni. È stato progettato per essere semplice e produttivo, con una sintassi chiara e coerente che facilita la scrittura e la manutenzione del codice. Inoltre, Dart offre prestazioni elevate grazie alla sua compilazione ahead-of-time (AOT) che consente di produrre codice nativo efficiente.

Flutter è preferibile a React Native per alcuni motivi chiave. Prima di tutto, Flutter offre una maggiore coerenza e controllo sull'aspetto e il comportamento delle applicazioni, grazie al suo set completo di widget personalizzabili. Inoltre, Flutter utilizza il motore grafico, che garantisce prestazioni elevate. Infine, Flutter ha un supporto eccellente per lo sviluppo multi-piattaforma, permettendo di scrivere una sola volta il codice e distribuirlo su iOS, Android, web e desktop.

Per quanto riguarda il backend è stato scritto in python con fastApi, e un database postgres.

3 Features dell'applicazione

L'applicazione comprende diverse pagine chiave:

- **Login:** Pagina di accesso per autenticare gli utenti.
- **Signup:** Pagina di accesso creare un profilo utente.
- **Search:** Motore di ricerca per trovare eventi o partner sportivi nella tua zona, con possibilità d'impostare dei filtri. Supporta due modalità di visualizzazione:
 - **Mappa:** visualizza gli eventi attraverso la loro localizzazione sulla mappa
 - **Lista:** lista che permette di visualizzare con maggiori dettagli gli eventi nella zona selezionata
- **Creazione Attività:** Interfaccia per creare nuovi eventi sportivi.
- **Storico:** Raccolta degli eventi passati a cui si è partecipato con la possibilità di aggiungere un feedback chiamato ricordo.
- **Ricordo:** Sezione per lasciare un messaggio di ricordo dell'attività, e aggiungerci un punteggio che rappresenta il gradimento dell'evento.
- **Visualizzazione di un'Attività e Partecipazione:** Pagina dettagliata di ogni evento, che include opzioni per iscriversi all'evento, ottenere informazioni aggiuntive e aggiungere una notifica per ricordarsi dell'evento.

4 Struttura del codice sorgente

La struttura del codice sorgente è divisa nelle seguenti cartelle:

- **config:** Contiene le configurazioni globali dell'applicazione che non cambiano durante l'esecuzione, o che se cambiano non hanno un'influenza sulla UI
- **data:** Contiene le strutture dati delle activity e del feedback.
- **provider:** Contiene i dati che, quando cambiano, hanno un riscontro nella UI e quindi dev'essere aggiornata anche la UI.
- **pages:** Sono definiti tutti i widget delle varie pagine e le pagine stesse.
- **utils.dart:** Contiene utility functions utilizzate in varie parti dell'applicazione.

I dati vengono caricati da un backend facendo richieste progressive (che si aggiornano rispetto all'ultima richiesta fatta) e vengono salvati in locale, così da averli offline.

```
-- config
|-- config.dart
|-- theme.dart
-- data
|-- activity_data.dart
|-- feedback_data.dart
-- main.dart
-- pages
|-- feedback
|-- feedback.dart
|-- feedback_list.dart
|-- history.dart
|-- rating_stars.dart
|-- general_purpose
|-- activity_card.dart
|-- activity_details.dart
|-- activity_page.dart
|-- loader.dart
|-- map_markers.dart
|-- maps_search.dart
|-- splash_screen.dart
-- home.dart
-- login
|-- action_state_login.dart
|-- login_signup.dart
|-- login_signup_widget.dart
-- new_activity
|-- layout_widget.dart
|-- new_activity.dart
|-- pos_selector.dart
-- root.dart
-- search
|-- Search.dart
|-- choose_position.dart
|-- filter_chips.dart
|-- filter_data.dart
|-- filter_dialog.dart
|-- map_search.dart
-- settings
|-- settings.dart
-- upcoming_activity
|-- upcoming_activity.dart
-- provider
|-- auth_provider.dart
|-- data_provider.dart
-- utils.dart
```

5 Scelte implementative

5.1 Dati

Le due strutture di dati principali dell'applicazione sono definite sotto **data** e sono:

```
class Activity {
    final String description;
    final DateTime time;
    final LatLng position;
    final Attributes attributes;
    final int numberOfPeople;
    final int id;
    final List<String> participants;
    final String creator;
}

class Attributes {
    final String level;
    final int price;
    final String sport;
}
```

```
class FeedbackActivity {
    String username;
    int activityId;
    int rating;
    String comment;
}
```

- **Activity:** contiene i dati relativi ad un evento sportivo, come il nome, la descrizione, la posizione, la data e l'ora, il numero di partecipanti, ecc.
- **Feedback:** contiene i dati relativi al feedback lasciato da un utente relativamente ad un'attività, come il punteggio, il messaggio e l'attività associata.

5.2 Provider

Un provider in Flutter è una classe che gestisce e fornisce dati e stato all'applicazione, facilitando la separazione tra logica di business e interfaccia utente. Questo approccio migliora la leggibilità e la manutenzione del codice. Il provider viene inserito nel widget tree e consente a ogni widget di accedere e reagire agli aggiornamenti dello stato. Utilizzando il provider, si può garantire una gestione efficiente e reattiva dei dati, poiché i widget si aggiornano automaticamente quando il provider notifica un cambiamento di stato.

Nell'applicazione, come visto nella sezione di struttura del codice sorgente, ne abbiamo due:

- **Activity Data:** specifica i metodi e i dati per caricare, modificare ed eliminare i dati relativi alle Activity
- **Auth:** specifica i metodi e i dati per gestire l'autenticazione e l'accesso dell'utente.

Activity Data in particolare è un provider che si occupa di caricare i dati delle activity, e di salvarli in locale, così da averli offline. Implementa la politica Single Source of Truth, ovvero i dati vengono caricati da un backend facendo richieste progressive (che si aggiornano rispetto all'ultima richiesta fatta) e vengono salvati in locale, così da averli offline.

Auth questo provider permette di aggiornare tutta la UI quando l'utente si autentica o si disconnette.

```
class DataProvider with ChangeNotifier {
  final storage = const FlutterSecureStorage();

  bool loading = false;
  bool isConnected = false;
  List<Activity> activities = [];
  List<FeedbackActivity> feedbacks = [];
  DateTime? lastUpdate = null;
  LatLng lastPos = LatLng(44.498955, 11.327591);
  bool loadingPos = false;

  Future<void> deleteAllStoredData() async {...}
  Future<void> loadFromStorage() async {...}
  Future<void> saveToStorage(DateTime lastUpdate) async {...}

  Future<void> load(token) async {
    await loadFromStorage();
    try {
      var lastUpdate = DateTime.now().subtract(lastUpdate.timeZoneOffset);
      this.feedbacks = await loadFeedback(token);
      update_activity(await _loadActivities(), await _deletedActivities(token));
      this.lastUpdate = DateTime.now();
      await saveToStorage(lastUpdate);
      loading = false; isConnected = true;
    } catch (e) {
      loading = false; isConnected = false;
    }
  }

  Future<List<FeedbackActivity>> loadFeedback(token) async {
    final req = await http.get(Uri.https(Config().host, '/feedback'),
      headers: {'Authorization': 'Bearer ${token}'});
    if (req.statusCode != 200) throw Exception('Impossibile caricare i feedback');
    return json.decode(req.body).cast<FeedbackActivity>();
  }

  Future<List<Activity>> _loadActivities(List<int> ids) async {
    Map<String, dynamic> params = {};
    if (lastUpdate != null) params['last_update'] = lastUpdate!.toIso8601String();
    return await http.get(Uri.https(Config().host, '/activities/search', params))
      .cast<Activity>();
  }

  Future<List<int>> _deletedActivities(token) async {...}
  void addFeedback(FeedbackActivity feedback) {...}
  void joinActivity(int id, String user) {...}
  void leaveActivity(int id, String user) {...}
  void deleteActivity(int id) {...}
}
```

5.3 Pagine

5.3.1 Home

All'esecuzione carica il file home, che controlla se è stato salvato un authentication token valido, in caso affermativo carica la pagina Search 5.5, in caso contrario carica la pagina di Registrazione e Login 5.4. Il cambio è fatto attraverso provider che, in base al cambiamento del token sceglie quale pagine caricare.

Quindi il codice della homepage è qualcosa come:

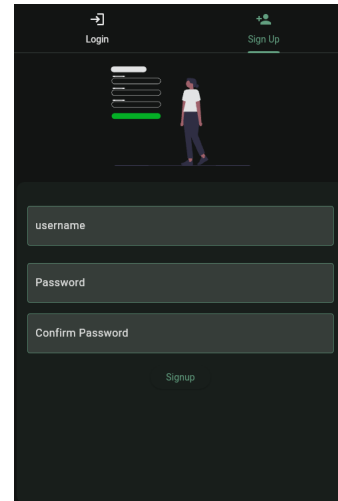
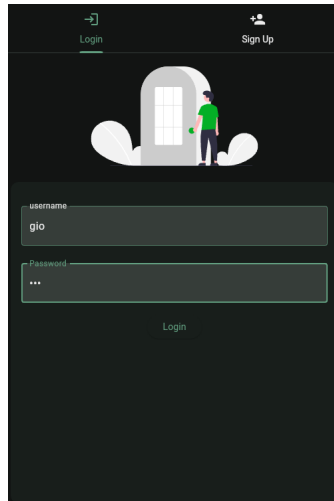


```
class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (ctx) => AuthProvider()..tryAutoLogin()),
        ChangeNotifierProvider(create: (ctx) => DataProvider())
      ],
      child: MaterialApp(
        home: Consumer<AuthProvider>(builder: (ctx, auth, _) {
          return auth.loading
            ? CircularProgressIndicator()
            : auth.isAuthenticated
              ? SearchPage()
              : LoginSignupPage();
        })),
    );
  }
}
```

Inoltre nella home andiamo a definire anche il DataProvider che utilizzeremo successivamente.

5.4 Registrazione e Accesso

La prima pagina visualizzata è quella di accesso e registrazione, questa pagina si connette alle API e impostano l'authentication token attraverso l'auth provider, questo aggiorna la home page cambiando la pagina home a SearchPage 5.5.



5.5 Search

Search è la pagina più complessa dell'applicazione, permette di cercare eventi sportivi nella propria zona, con possibilità d'impostare dei filtri. Supporta due modalità di visualizzazione. Il codice ad alto livello, è composto da due widget stateful, il primo **SearchPage** che è un wrapper, carica i dati da DataProvider e aggiorna **_SearchPage** che contiene l'interfaccia della search. **SearchPage** è il widget root della pagine autenticate, questo significa che quando si modificano dei dati significativi in Data Provider, questo aggiorna tutti i componenti sottostanti.

La **_SearchPage** contiene altri dati come i filtri, la posizione attuale, e il raggio di ricerca.

```

class SearchPage extends StatefulWidget {
  @override
  State<SearchPage> createState() => _SearchPageState();
}

class _SearchPageState extends State<SearchPage> {
  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addPostFrameCallback((_) {
      final authProvider = Provider.of<AuthProvider>(context, listen: false);
      Provider.of<DataProvider>(context, listen: false).load(authProvider.token!);
    });
  }

  @override
  Widget build(BuildContext context) {
    return Consumer<DataProvider>(builder: (context, dataProvider, child) {
      return SearchPage(key: UniqueKey(), data: dataProvider.toApplicationData());
    });
  }
}

```

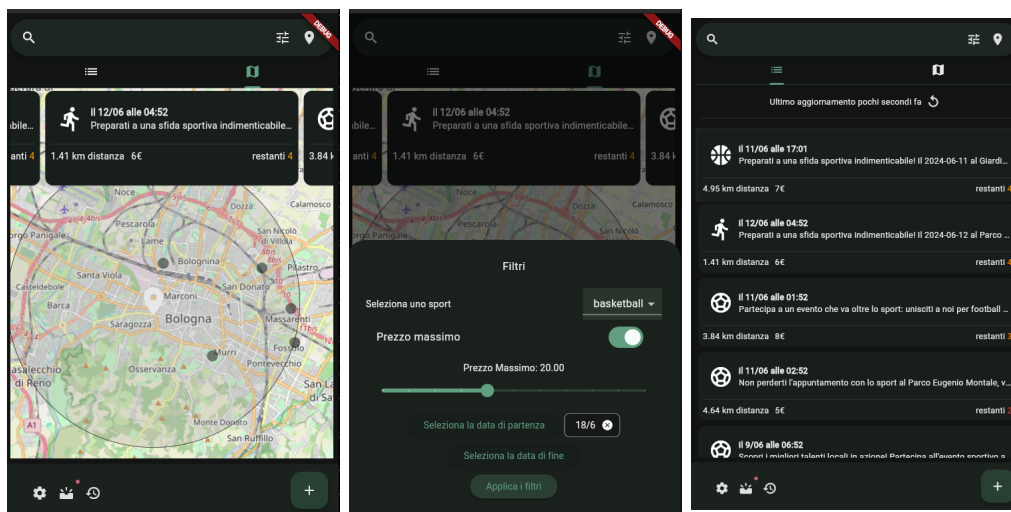
```

class _SearchPage extends StatefulWidget {
  final ApplicationData data;
  _SearchPage(super.key, required this.data);
  @override
  State<_SearchPage> createState() => _SearchPageStateFilter(data);
}

class _SearchPageStateFilter extends State<_SearchPage> {
  LatLng? pos;
  double radius = 5000;
  List<Activity> displayActivities = [];
  final SearchController searchController = SearchController();
  FilterData filterData = FilterData.init();
}

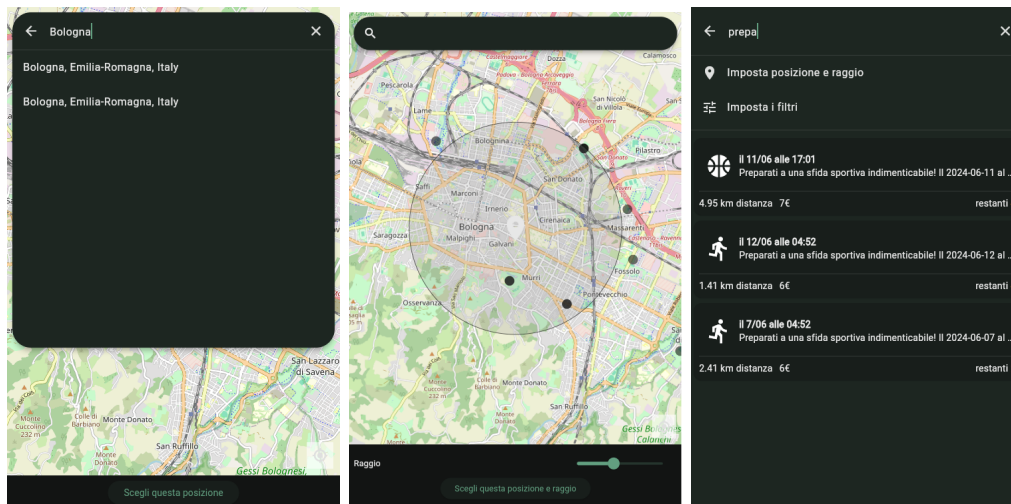
```

In search possiamo compiere varie azioni, tra cui visualizzare le attività sia in modalità lista 5.5, che in modalità mappa 5.5, permette di visualizzare gli eventi sia tramite la posizione. Infatti nella modalità mappa possiamo visualizzare gli eventi tramite il marker. I filtri agiscono invece su entrambe le visualizzazioni degli eventi.



Di default la posizione e raggio sono preimpostati su Bologna o la posizione da cui è avvenuto l'accesso più recente. Per cambiarla si può cliccare sul bottone in alto a destra con il simbolo dell'puntatore, questo porta ad un'altra pagina che permette di modificare la posizione e il raggio di ricerca. Trascinando la mappa si può cambiare la posizione e il raggio di ricerca tramite lo slider. Inoltre è possibile utilizzare anche cercare una posizione tramite la barra di ricerca, o impostare quella corrente con il simbolo della posizione accanto.

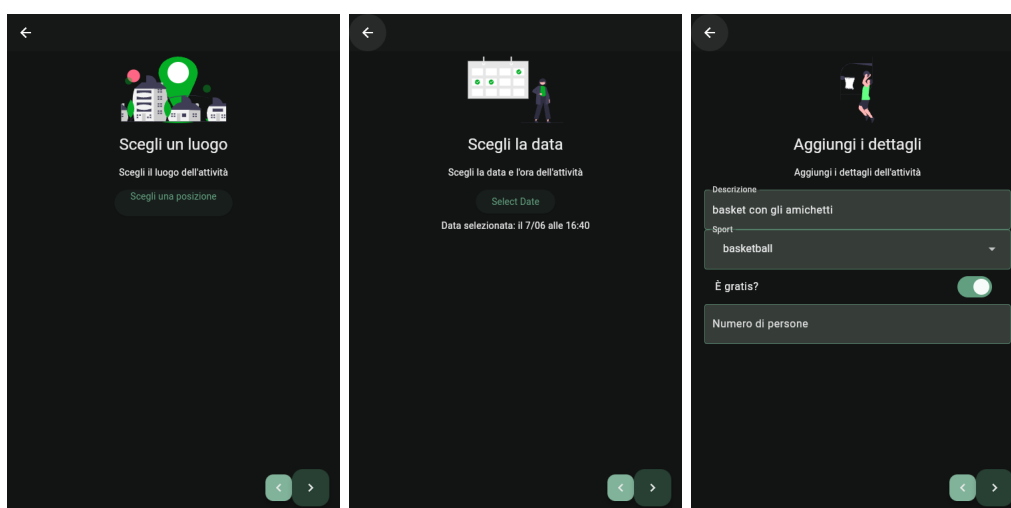
Oltre i filtri si può usare anche la barra di ricerca per cercare un'attività tramite la descrizione.



5.6 Aggiungi attività

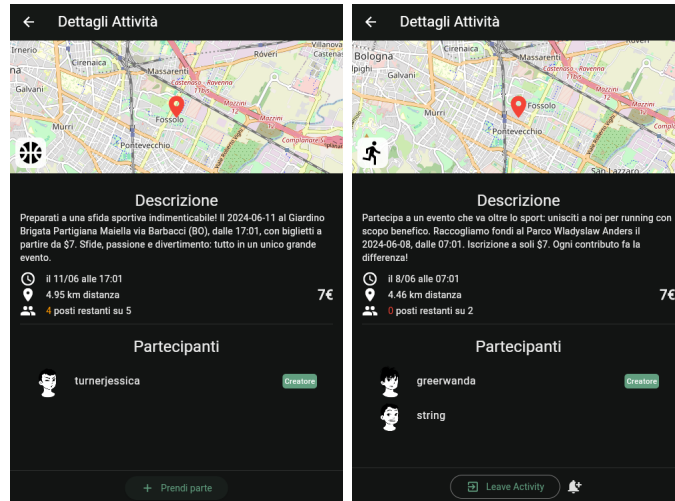
La creazione nel creare un attività sportiva richiede la compilazione di diverse pagine:

- **Posizione:** permette di selezionare la posizione dell'evento attraverso una mappa.
- **Data:** permette di selezionare la data e l'ora dell'evento.
- **Dettagli:** permette di inserire una descrizione dell'evento, la disciplina sportiva, il numero di partecipanti e il prezzo.



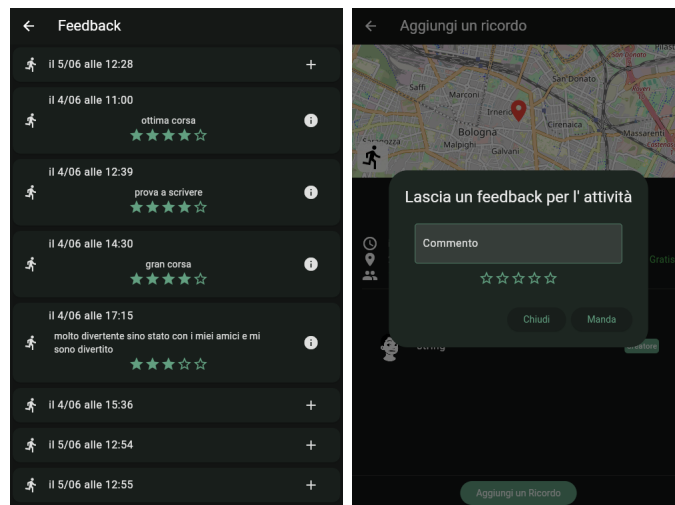
5.7 Partecipa ad un'attività

Per partecipare ad un'attività basta cliccare sul bottone partecipa, e si verrà aggiunti alla lista dei partecipanti. Inoltre si può aggiungere un promemoria per l'evento. Questo promemoria consiste in una notifica che verrà inviata n minuti prima dell'evento, dove n sono i minuti definiti nelle impostazioni.



5.8 Storia

Permette di visualizzare gli eventi passati a cui si è partecipato con il relativo Ricordo, o di aggiungerlo in caso non lo si abbia ancora fatto. Quando aggiunge un feedback utilizza il DataProvider per aggiornare i dati sia in locale che nel backend.



5.9 Impostazioni

Le impostazioni, mostrano informazioni sull'utente, e permettono di modificare quanto prima dell'evento ricevere una notifica. Inoltre permette di disconnettersi dall'account e quindi eliminare anche i dati in locale. Le configurazioni a differenza di altre impostazioni non devono essere inserite dentro un Provider in quanto non influenzano la UI.

