

# Programmazione Mobile

**Nicola Noviello**

[nicola.noviello@unimol.it](mailto:nicola.noviello@unimol.it)

Corso di Laurea in Informatica  
Dipartimento di Bioscienze e Territorio  
Università degli Studi del Molise  
Anno 2023/2024

# Lezione: Utilizzo delle feature native dei device mobile

- Cenni alla persistenza degli stati
- Utilizzo della camera
- Location e Google Maps



# Recap sugli stati

# Un framework dichiarativo

Quando si arriva a Flutter da un framework imperativo (come Android SDK o iOS UIKit), bisogna guardare lo sviluppo delle App da una nuova prospettiva.

La cosa più evidente notata nelle lezioni passate è che con Flutter è accettabile ricostruire parti della UI da zero invece di modificarle. Flutter è abbastanza veloce nel farlo senza che l'esperienza utente venga compromessa.

Quando lo stato dell'App cambia (ad esempio, l'utente attiva uno switch nella schermata delle impostazioni), viene cambiato lo stato, e ciò attiva un ridisegno dell'interfaccia utente.

Non c'è un cambiamento imperativo dell'UI stessa (come `widget.setText`) - cambi lo stato e l'UI si ricostruisce da zero.




# Stato effimero e stato dell'App

Nel senso più ampio possibile, lo **stato** di un'App è tutto ciò che esiste in memoria quando l'App è in esecuzione. Questo include le risorse dell'app, tutte le variabili che il framework Flutter mantiene sull'UI, lo stato delle animazioni, le texture grafiche, i font e così via. Anche se questa definizione più ampia di stato è valida, non è molto utile per progettare un'App, in quanto le casistiche reali richiedono degli approcci più precisi.

Peraltro, gli stati di alcuni elementi, come le texture grafiche, non sono gestiti nemmeno direttamente dall'utente, ma sono delegati al framework. Quindi una definizione più precisa di **stato** è *“qualsiasi dato di cui hai bisogno per ricostruire la tua UI in qualsiasi momento”*.

In secondo luogo, lo stato gestito da uno sviluppatore può essere separato in due tipi concettuali: **stato effimero** e **stato dell'App**.



# Lo stato effimero (o stato locale)

Lo stato effimero è lo stato contenuto in un singolo widget. Questa definizione è di fatto vaga, quindi questi sono alcuni esempi:

- pagina corrente in una `PageView`
- progresso corrente di un'animazione complessa
- scheda selezionata corrente in una `BottomNavigationBar`

In sostanza, parliamo di informazioni che non richiedono l'accesso da parte di altri widget.



# Lo stato effimero (o stato locale) - Esempio

```
class MyHomepage extends StatefulWidget {  
  const MyHomepage({super.key});  
  
  @override  
  State<MyHomepage> createState() => _MyHomepageState();  
}  
  
class _MyHomepageState extends State<MyHomepage> {  
  int _index = 0;  
  
  @override  
  Widget build(BuildContext context) {  
    return BottomNavigationBar(  
      currentIndex: _index,  
      onTap: (newIndex) {  
        setState(() {  
          _index = newIndex;  
        });  
      },  
      // ... items ...  
    );  
  }  
}
```

dart

# App State / Stato Condiviso


Lo stato che per definizione non è effimero, e che deve essere condiviso tra altre parti dell'App è quello che viene chiamato App State o Shared State.

Esempi pratici:

- Preferenze dell'utente
- Informazioni di accesso
- Notifiche in un'App di social networking
- Il carrello della spesa in un'App di e-commerce
- Stato di lettura/non lettura degli articoli in un'app di notizie

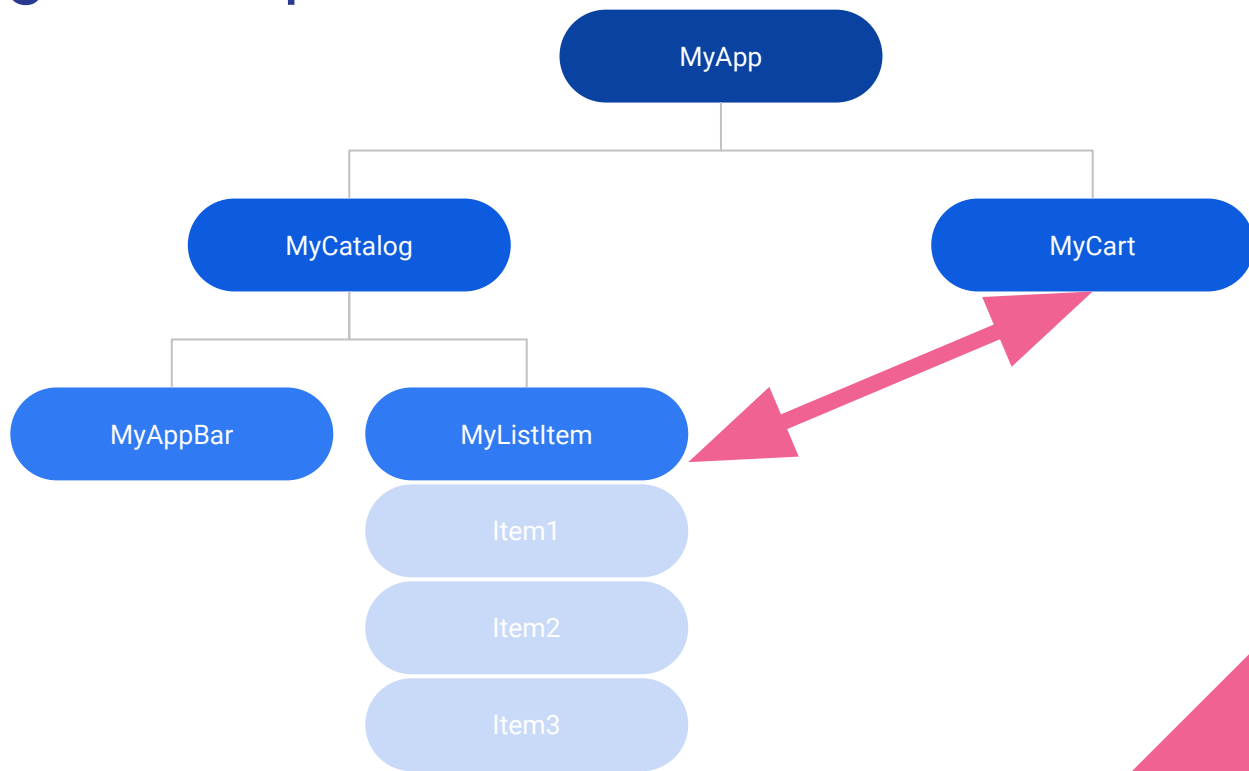
Dalla complessità e natura dell'App, dall'esperienza del team di sviluppo e da molti altri aspetti si effettua la scelta per la gestione dello stato dell'App.

Non esiste una regola universale definita per distinguere se una variabile particolare dovrà essere di stato effimero o di stato condiviso. A volte sarà necessario un refactoring e quindi passare da uno stato chiaramente effimero, ma man mano che l'app cresce in funzionalità, potrebbe essere necessario spostarlo allo stato condiviso.

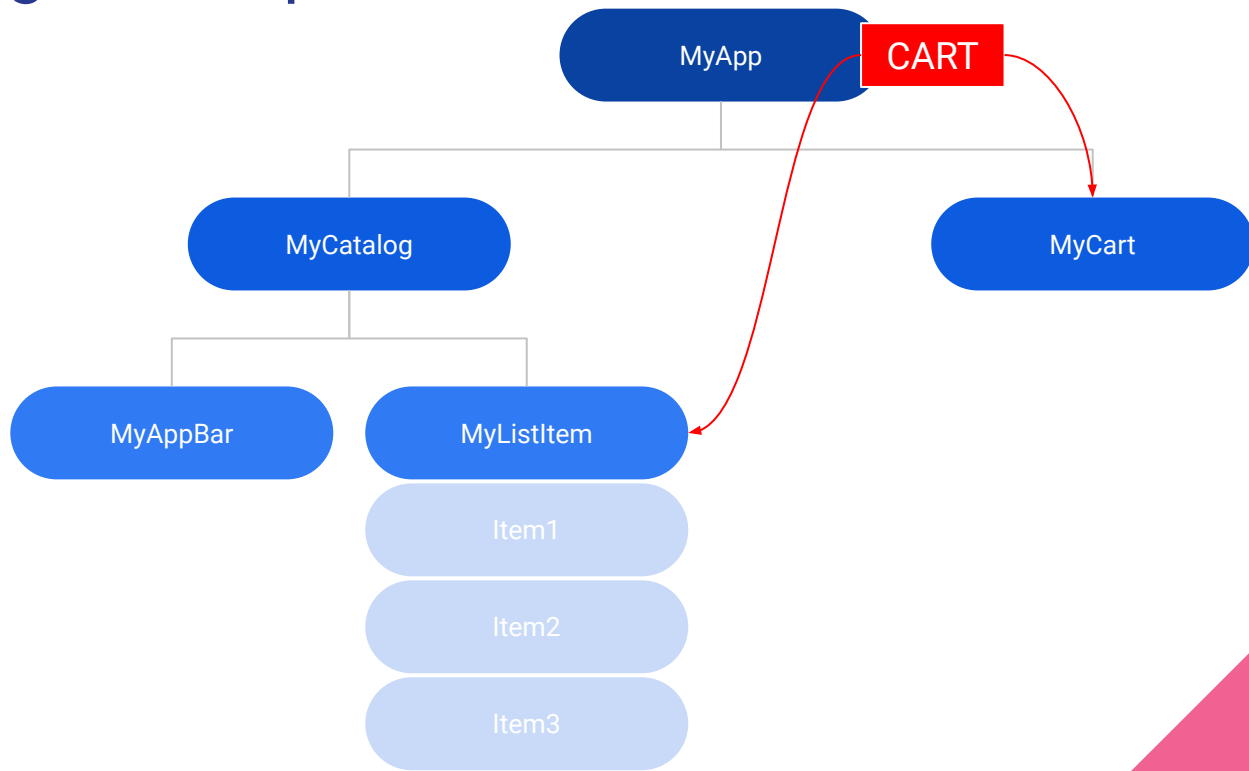




# Lifting state up



# Lifting state up



# Gestione degli Stati in Flutter: Oltre lo Stato Effimero e il Lifting State Up

- Stato Effimero
  - Gestito direttamente dai widget.
  - Utilizzato per stati temporanei che non influenzano altri widget (es. animazioni, contatori).
- Lifting State Up
  - Quando lo stato è condiviso tra più widget, si solleva lo stato al widget più vicino che include tutti i widget che lo usano.
  - Promuove la riusabilità e la gestione centralizzata dello stato.



# Gestione degli Stati con Componenti di Terze Parti

## Per stati complessi e condivisi

Utilizzo di pacchetti e librerie esterni per una gestione dello stato più robusta e scalabile.

- **Provider:**
  - Uno dei pacchetti più popolari per la gestione dello stato;
  - Facilita la condivisione e il consumo dello stato attraverso il widget tree;
- **Bloc/Cubit:**
  - Basato sull'architettura BLoC (Business Logic Component);
  - Separa la logica di business dalla presentazione, rendendo il codice più testabile e gestibile;
- **Riverpod:**
  - Evoluzione del pacchetto Provider;
  - Fornisce una sintassi più sicura e flessibile per la gestione dello stato.




# Vantaggi dell'utilizzo di componenti per la gestione avanzata degli stati

**Scalabilità:** Facilita la gestione di stati complessi in grandi applicazioni;

**Testabilità:** Rende la logica di business facilmente testabile;

**Manutenibilità:** Promuove una separazione chiara tra logica e UI.





Progetto di partenza  
(facciamo qualche step e  
poi vi fornisco il codice)

# Progetto di partenza

```
lib > main.dart > ...
1 import 'package:flutter/material.dart';
2
3 import 'package:google_fonts/google_fonts.dart';
4
5 final colorScheme = ColorScheme.fromSeed(
6   brightness: Brightness.dark,
7   seedColor: const Color.fromARGB(255, 102, 6, 247),
8   surface: const Color.fromARGB(255, 56, 49, 66),
9 ); // ColorScheme.fromSeed
10
11 final theme = ThemeData().copyWith(
12   scaffoldBackgroundColor: colorScheme.surface,
13   colorScheme: colorScheme,
14   textTheme: GoogleFonts.ubuntuCondensedTextTheme().copyWith(
15     titleSmall: GoogleFonts.ubuntuCondensed(
16       fontWeight: FontWeight.bold,
17     ),
18     titleMedium: GoogleFonts.ubuntuCondensed(
19       fontWeight: FontWeight.bold,
20     ),
21     titleLarge: GoogleFonts.ubuntuCondensed(
22       fontWeight: FontWeight.bold,
23     ),
24   ),
25 );
26
27 void main() {
28   runApp(
29     const MyApp(),
30   );
31 }
32
33 class MyApp extends StatelessWidget {
34   const MyApp({Key? key}) : super(key: key);
35
36   @override
37   Widget build(BuildContext context) {
38     return MaterialApp(
39       title: 'Great Places',
40       theme: theme,
41       home: ...,
42     );
43   }
44 }
```

# Progetto di partenza (cosa fa)

- Importa i pacchetti necessari, tra cui `flutter/material.dart` per i widget di base di Flutter e `google_fonts/google_fonts.dart` per l'utilizzo dei font di Google;
- Definisce un `ColorScheme` chiamato `colorScheme` che viene utilizzato per definire i colori dell'App. Questo schema di colori è impostato sulla modalità scura (`Brightness.dark`) e utilizza un colore specifico come colore di base;
- Crea un tema chiamato `theme` che copia il tema di default e lo modifica con il `ColorScheme` definito sopra e un `textTheme` personalizzato che utilizza il font `Ubuntu Condensed`;
- Definisce la funzione `main` che avvia l'App Flutter. Questa funzione chiama `runApp` con un'istanza della classe `MyApp`;
- Definisce una classe `MyApp` che estende `StatelessWidget`. Questa classe rappresenta l'App stessa. Il metodo `build` di questa classe restituisce un widget `MaterialApp` che utilizza il tema definito sopra e ha un titolo 'Great Places'. Il widget che viene mostrato quando l'App viene avviata è definito dalla proprietà `home`, che è attualmente omessa (...).





# Aggiungo un model: place.dart

## DEVICENATIVE

> .dart\_tool

> .idea

> android

> ios

✓ lib

✓ models

place.dart

main.dart

1

lib > models > place.dart > ...

```
1  import 'package:uuid/uuid.dart';
2
3  const uuid = Uuid();
4
5  class Place {
6    Place({required this.title}) : id = uuid.v4();
7
8    final String id;
9    final String title;
10 }
```

# Creo un widget per una schermata: PlacesScreen

```
lib > screens > places.dart > ...
1  import 'package:devicenative/widgets/places_list.dart';
2  import 'package:flutter/material.dart';
3
4  class PlacesScreen extends StatelessWidget {
5    const PlacesScreen({super.key});
6
7    @override
8    Widget build(BuildContext context) {
9      return Scaffold(
10        appBar: AppBar(
11          title: const Text('I tuoi luoghi preferiti'),
12          actions: [
13            IconButton(
14              icon: const Icon(Icons.add),
15              onPressed: () {},
16            ), // IconButton
17          ],
18        ), // AppBar
19        body: const PlacesList(
20          places: [],
21        ), // PlacesList
22      ); // Scaffold
23    }
24  }
```

# Creo un widget per una schermata: PlacesScreen

- Definisce una classe PlacesScreen che estende StatelessWidget. Questa classe rappresenta una schermata dell'App che mostra una lista di luoghi;
- Il metodo build di PlacesScreen restituisce un widget Scaffold, che fornisce la struttura di base per la schermata. Questo include una barra delle applicazioni (AppBar) con un titolo e un pulsante, e un corpo (body) che è un'istanza del widget PlacesList;
- La AppBar ha un titolo 'I tuoi luoghi preferiti' e un'azione che è un IconButton. Questo pulsante ha un'icona Icon(Icons.add), ma attualmente non fa nulla quando viene premuto (onPressed: () {});
- Il corpo della schermata è un'istanza del widget PlacesList (che integreremo). Attualmente, viene passata una lista vuota di luoghi (places: []), quindi non verrà mostrato alcun luogo.

# Creo un widget per un elemento: PlacesList

```
lib > widgets > places_list.dart > ...
1  import 'package:flutter/material.dart';
2
3  import 'package:devicenative/models/place.dart';
4
5  class PlacesList extends StatelessWidget {
6    const PlacesList({super.key, required this.places});
7
8    final List<Place> places;
9
10   @override
11   Widget build(BuildContext context) {
12     if (places.isEmpty) {
13       return Center(
14         child: Text(
15           'Non hai ancora aggiunto luoghi preferiti.',
16           style: Theme.of(context).textTheme.titleMedium!.copyWith(
17             color: Theme.of(context).colorScheme.onSurface,
18           ),
19         ), // Text
20       ); // Center
21     }
22
23     return ListView.builder(
24       itemCount: places.length,
25       itemBuilder: (ctx, index) => ListTile(
26         title: Text(
27           places[index].title,
28           style: Theme.of(context).textTheme.titleMedium!.copyWith(
29             color: Theme.of(context).colorScheme.onSurface,
30           ),
31         ), // Text
32       ), // ListTile
33     ); // ListView.builder
34   }
35 }
36
```

# Creo un widget per un elemento: PlacesList

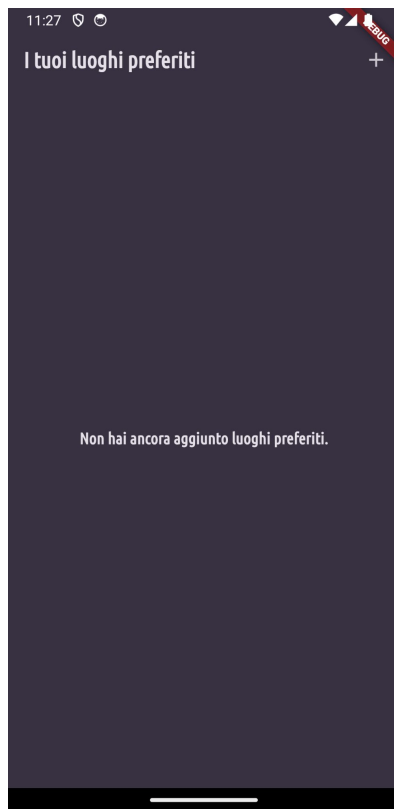
- Il costruttore di PlacesList accetta una lista di oggetti Place come parametro. Questa lista viene passata quando si crea un'istanza di PlacesList;
- Il metodo build di PlacesList controlla se la lista di luoghi è vuota. Se lo è, restituisce un widget Center che contiene un widget Text con il messaggio 'Non hai ancora aggiunto luoghi preferiti.';
- Se la lista di luoghi non è vuota, build restituisce un widget ListView.builder. Questo widget crea una lista di elementi basata sulla lunghezza della lista di luoghi. Per ogni luogo, crea un ListTile con il titolo del luogo come testo;
- Il testo sia nel caso di lista vuota che nel caso di ListTile segue lo stesso stile, che è definito dal tema corrente dell'App. Il colore del testo è impostato per contrastare con il colore di superficie del tema, rendendo il testo leggibile indipendentemente dal colore di sfondo.

# ListTile

**ListTile** è un widget per la gestione delle entry in una lista che segue le linee guida del Material Design ed è un'alternativa alla costruzione manuale di un widget per gestire gli elementi di una tabella, tra i vari vantaggi rispetto ad un'implementazione manuale troviamo:

- **Consistenza:** ListTile segue le linee guida del Material Design, il che significa che avrà un aspetto e un comportamento coerenti con altri elementi dell'interfaccia utente di Material Design;
  - **Risparmio di tempo:** ListTile include molte funzionalità comuni per un elemento di lista, come un titolo, un sottotitolo, icone leading e trailing, e gestori per tap e long press. Se si dovesse costruire un elemento di lista a mano, come fatto nella precedente lezione, tutte queste funzionalità andrebbero implementate manualmente;
  - **Adattabilità:** ListTile si adatta automaticamente a diverse dimensioni di schermo e densità di pixel, il che significa che avrà un aspetto ottimale su una varietà di dispositivi senza doversi preoccupare di nulla;
  - **Accessibilità:** ListTile include alcune funzionalità di accessibilità di base, come il supporto per screen reader.
- 

# Risultato attuale



# Creo un widget per una schermata: AddPlaceScreen

```
lib > screens > add_place.dart > _AddPlaceScreenState > build
1 import 'package:flutter/material.dart';
2
3 class AddPlaceScreen extends StatefulWidget {
4   const AddPlaceScreen({super.key});
5
6   @override
7   State<AddPlaceScreen> createState() {
8     return _AddPlaceScreenState();
9   }
10 }
11
12 class _AddPlaceScreenState extends State<AddPlaceScreen> {
13   final _titleController = TextEditingController();
14
15   @override
16   void dispose() {
17     _titleController.dispose();
18     super.dispose();
19   }
20
21   @override
22   Widget build(BuildContext context) {
23     return Scaffold(
24       appBar: AppBar(
25         title: const Text('Aggiungi un luogo'),
26       ), // AppBar
27       body: SingleChildScrollView(
28         padding: const EdgeInsets.all(12),
29         child: Column(
30           children: [
31             TextField(
32               decoration: const InputDecoration(labelText: 'Titolo'),
33               controller: _titleController,
34               style: TextStyle(
35                 color: Theme.of(context).colorScheme.onSurface,
36               ), // TextStyle
37             ), // TextField
38             const SizedBox(height: 16),
39             ElevatedButton.icon(
40               onPressed: () {},
41               icon: const Icon(Icons.add),
42               label: const Text('Add Place'),
43             ), // ElevatedButton.icon
44           ],
45         ), // Column
46       ), // SingleChildScrollView
47     ); // Scaffold
48   }
49 }
50
51
```



# Creo un widget per una schermata: AddPlaceScreen

- `_AddPlaceScreenState` ha un campo `_titleController`, che è un `TextEditingController`. Questo controller permette di controllare il testo e la selezione in un `TextField`;
- Il metodo `dispose` di `_AddPlaceScreenState` chiama `dispose` su `_titleController` per liberare le risorse quando il widget viene rimosso dall'albero dei widget, e poi chiama `super.dispose()`;
- Il metodo `build` di `_AddPlaceScreenState` restituisce un `Scaffold` che contiene un `AppBar` con un titolo, e un `SingleChildScrollView` come corpo. Il `SingleChildScrollView` contiene una `Column` con un `TextField` per il titolo del luogo, uno spazio, e un `ElevatedButton` per aggiungere il luogo;
- Il `TextField` usa `_titleController` per controllare il suo testo e la sua selezione, e il suo colore di testo è impostato per contrastare con il colore di superficie del tema corrente;
- L'`ElevatedButton` non fa nulla quando viene premuto, perché la sua proprietà `onPressed` è impostata su una funzione vuota.

# Aggiungo la navigazione

Da PlacesScreen → AddPlacesScreen

```
icon: const icon(Icons.add),  
onPressed: () {  
  Navigator.of(context).push(  
    MaterialPageRoute(  
      builder: (ctx) => const AddPlaceScreen(),  
    ), // MaterialPageRoute  
  );  
}
```

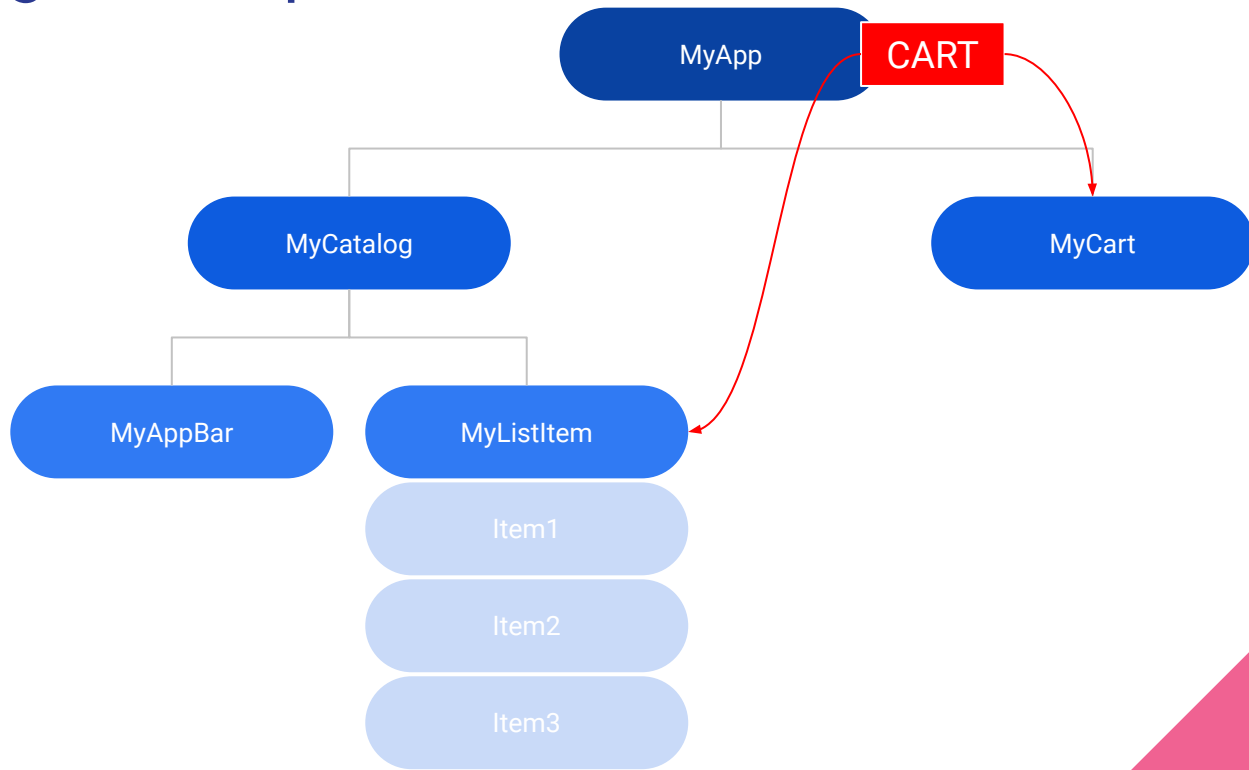
# Integrazione di riverpod

# Base di partenza

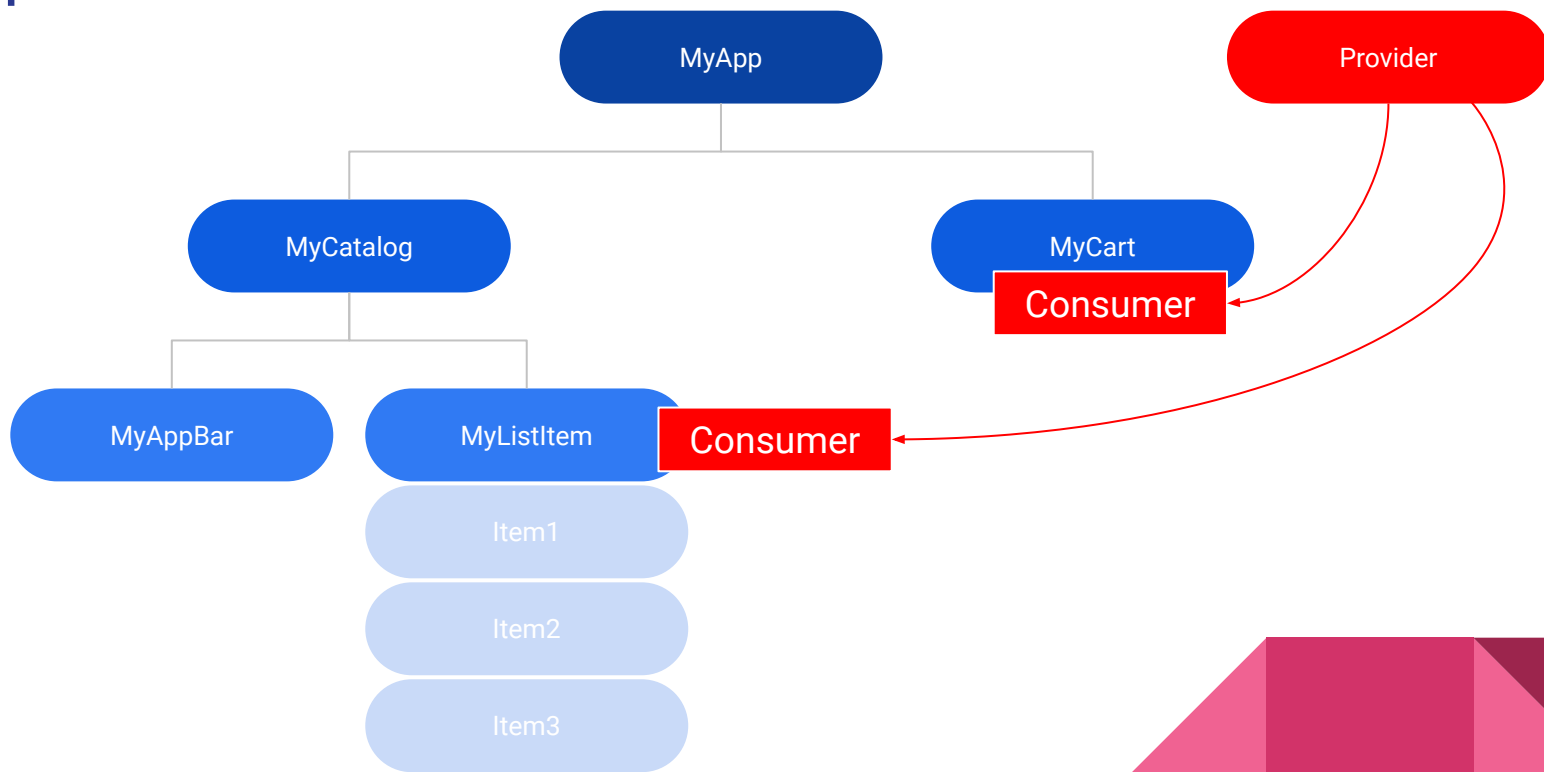
La base di partenza è pronta, integrate le cose già viste a lezione possiamo passare a trattare un nuovo argomento, potete scaricare il progetto sviluppato sin d'ora: `progetto_di_partenza.zip`



# Lifting state up



# Riverpod




# Introduzione a Riverpod

## Cos'è Riverpod?

- Riverpod è un pacchetto di terze parti per la gestione dello stato nelle applicazioni Flutter;
- Evoluzione del pacchetto Provider, creato dagli stessi autori;
- Progettato per semplificare la gestione dello stato tra widget.

## Perché Usare Riverpod?

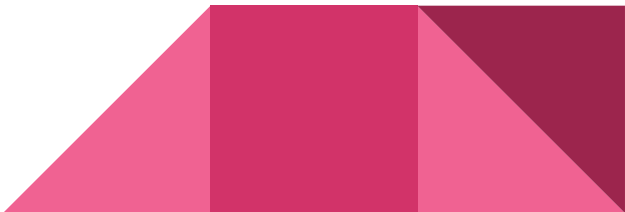
- **Modernità:** Versione più recente e migliorata rispetto a Provider;
  - **Type Safety:** Riduce gli errori di runtime;
  - **Flessibilità:** Permette la gestione dello stato sia per widget singoli che tra più widget;
  - **Supporto Avanzato per il Test:** Facilita la scrittura e l'esecuzione dei test.
- 

# Funzionamento di Riverpod

## Concetti Chiave

- **Provider:** Oggetto che gestisce e fornisce un valore dinamico
  - Può fornire metodi per cambiare tale valore.
- **Consumer:** Widget che si connette automaticamente al provider
  - Ascolta i cambiamenti del valore e può triggerare aggiornamenti.

## Vantaggi Operativi

- **Gestione Centralizzata dello Stato:** Non è necessario passare lo stato tra i widget;
  - **Connessione Diretta:** Ogni widget può connettersi direttamente al provider;
  - **Scalabilità:** Adatto per gestire stati semplici e complessi.
- 



# L'idea

Nel nostro caso concreto, vogliamo quindi accedere e modificare una lista di oggetti Place (`models/place.dart`) da qualsiasi widget della nostra App, senza strani passaggi di informazioni.



# Creo un Provider: UserPlacesNotifier

▼ DEVICENATIVE

> .dart\_tool

> .idea

> android

> ios

▼ lib

▼ models

place.dart

▼ providers

user\_places.dart

▼ screens

add\_place.dart

places.dart

▼ widgets

places\_list.dart

main.dart

> linux

> macos

lib > providers > user\_places.dart > ...

1 import 'package:flutter\_riverpod/flutter\_riverpod.dart';

2

3 import 'package:devicenative/models/place.dart';

4

5 class UserPlacesNotifier extends StateNotifier<List<Place>> {

6 UserPlacesNotifier() : super(const []);

7

8 void addPlace(String title) {

9 // questo potete scriverlo come volete, è a vostra discrezione

10 // potete passare anche un oggetto Place direttamente

11 final newPlace = Place(title: title);

12 // aggiorna lo stato con il nuovo luogo e il resto della lista "spalmata"

13 state = [newPlace, ...state];

14 }

15 }

16

17 final userPlacesProvider =

18 StateNotifierProvider<UserPlacesNotifier, List<Place>>(  
19 (ref) => UserPlacesNotifier(),  
20 ); // StateNotifierProvider

21

# Creo un Provider: UserPlacesNotifier

- **UserPlacesNotifier** è una classe che estende `StateNotifier`. `StateNotifier` è una classe fornita dal pacchetto `flutter_riverpod` che ti permette di gestire lo stato di un'App Flutter. In questo caso, lo stato è una lista di oggetti `Place`;
- Il costruttore di **UserPlacesNotifier** chiama il costruttore della superclasse con una lista vuota di oggetti `Place`. Questo significa che lo stato iniziale è una lista vuota;
- `UserPlacesNotifier` ha un metodo **addPlace** che accetta un titolo come stringa. Questo metodo crea un nuovo oggetto `Place` con il titolo fornito, e poi aggiorna lo stato per includere il nuovo luogo all'inizio della lista;
- **userPlacesProvider** è un `StateNotifierProvider` che fornisce un'istanza di `UserPlacesNotifier`. `StateNotifierProvider` è un tipo di provider Riverpod che fornisce un `StateNotifier`. Il costruttore di `StateNotifierProvider` accetta una funzione che crea un'istanza di `UserPlacesNotifier`. In questo caso, la funzione è `(ref) => UserPlacesNotifier()`, che crea un nuovo `UserPlacesNotifier` senza argomenti.



# Creo un Provider: UserPlacesNotifier

Quello che in soldoni ci interessa sapere è che in questo modo possiamo accedere all'attuale lista di luoghi usando `context.read(userPlacesProvider)`, e possiamo aggiungere un nuovo luogo usando `context.read(userPlacesProvider.notifier).addPlace(title)`.



## Adeguiamo il resto del codice: Aggiungiamo Riverpod nel main!

lib > main.dart > main

```
1 import 'package:flutter/material.dart';
2 import 'package:flutter_riverpod/flutter_riverpod.dart';
3 import 'package:google_fonts/google_fonts.dart';
4
5 import 'package:devicenative/screens/places.dart';
6
```

Run | Debug | Profile

```
void main() {
  runApp(
    const ProviderScope(child: MyApp()),
  );
}
```

# Adeguamento di AddPlaceScreen

```
lib > screens > add_place.dart > _AddPlaceScreenState
1  import 'package:flutter/material.dart';
2  import 'package:flutter_riverpod/flutter_riverpod.dart';
3
4  import 'package:devicenative/providers/user_places.dart';
5
6  class AddPlaceScreen extends ConsumerStatefulWidget {
7    const AddPlaceScreen({super.key});
8
9    @override
10    ConsumerState<AddPlaceScreen> createState() {
11      return _AddPlaceScreenState();
12    }
13  }
14
15  class _AddPlaceScreenState extends ConsumerState<AddPlaceScreen> {
16    final _titleController = TextEditingController();
17
18    void savePlace() {
19      final enteredTitle = _titleController.text;
20
21      if (enteredTitle.isEmpty) {
22        return;
23      }
24
25      ref.read(userPlacesProvider.notifier).addPlace(enteredTitle);
26
27      Navigator.of(context).pop();
28    }
29  }
```

# Adeguamento di AddPlaceScreen

- AddPlaceScreen è stato modificato in un ConsumerStatefulWidget, che è una versione di StatefulWidget che può accedere ai provider Riverpod. Questo significa che può leggere e manipolare lo stato gestito da Riverpod;
- \_AddPlaceScreenState ha un metodo \_savePlace che viene chiamato quando l'utente vuole salvare un nuovo luogo. Questo metodo legge il testo dal \_titleController, e se il testo non è vuoto, chiama il metodo addPlace sul notifier dell' userPlacesProvider;
- ref.read(userPlacesProvider.notifier).addPlace(enteredTitle); è la linea di codice che interagisce con il provider Riverpod. ref.read(userPlacesProvider.notifier) ottiene il notifier del userPlacesProvider, che è un'istanza di UserPlacesNotifier. Poi chiama il metodo addPlace su quel notifier, passando il titolo inserito dall'utente. Questo aggiorna lo stato del userPlacesProvider per includere il nuovo luogo;
- Navigator.of(context).pop(); rimuove la schermata AddPlaceScreen dallo stack di navigazione, ritornando alla schermata precedente. Questo avviene dopo che il nuovo luogo è stato aggiunto, quindi l'utente non vedrà la schermata AddPlaceScreen dopo aver aggiunto un luogo.



# Adeguamento di PlacesScreen

```
places.dart x places_list.dart add_place.dart user_places.dart main.dart
lib > screens > places.dart > ...
1 import 'package:flutter/material.dart';
2 import 'package:flutter_riverpod/flutter_riverpod.dart';
3
4 import 'package:devicenative/providers/user_places.dart';
5 import 'package:devicenative/widgets/places_list.dart';
6 import 'package:devicenative/screens/add_place.dart';
7
8 class PlacesScreen extends ConsumerWidget {
9   const PlacesScreen({super.key});
10
11   @override
12   Widget build(BuildContext context, WidgetRef ref) {
13     final userPlaces = ref.watch(userPlacesProvider);
14     return Scaffold(
15       appBar: AppBar(
16         title: const Text('I tuoi luoghi preferiti'),
17         actions: [
18           IconButton(
19             icon: const Icon(Icons.add),
20             onPressed: () {
21               Navigator.of(context).push(
22                 MaterialPageRoute(
23                   builder: (ctx) => const AddPlaceScreen(),
24                 ), // MaterialPageRoute
25               );
26             },
27           ), // IconButton
28         ], // AppBar
29       ), // Scaffold
30       body: PlacesList(
31         places: userPlaces,
32       ), // PlacesList
33     ); // Scaffold
34   }
35 }
36
```

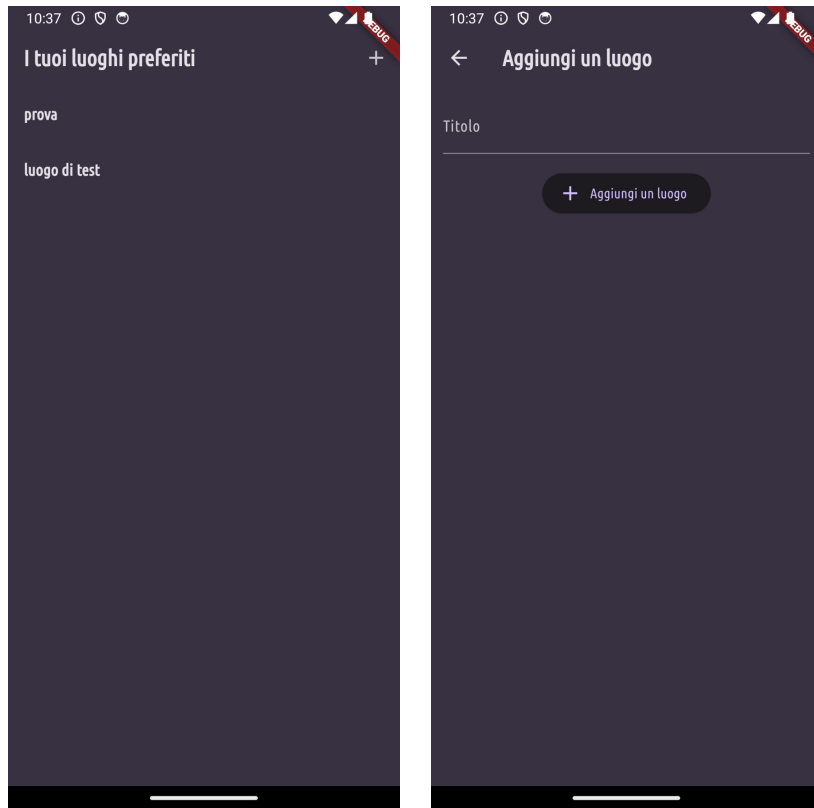


# Adeguamento di PlacesScreen

- PlacesScreen è stato modificato in un ConsumerWidget, che è una versione di Widget che può accedere ai provider Riverpod. Questo significa che può leggere e manipolare lo stato gestito da Riverpod;
- Il metodo build di PlacesScreen accetta due argomenti: un BuildContext e un WidgetRef. BuildContext è un oggetto che contiene informazioni sul luogo del widget nell'albero dei widget. WidgetRef è un oggetto fornito da Riverpod che permette di accedere ai provider;
- `final userPlaces = ref.watch(userPlacesProvider);` è la linea di codice che interagisce con il provider Riverpod. `ref.watch(userPlacesProvider)` “osserva” il `userPlacesProvider`, che significa che ogni volta che lo stato del `userPlacesProvider` cambia, il widget PlacesScreen viene ricostruito con il nuovo stato. `userPlaces` è una variabile che contiene l'attuale stato del `userPlacesProvider`.

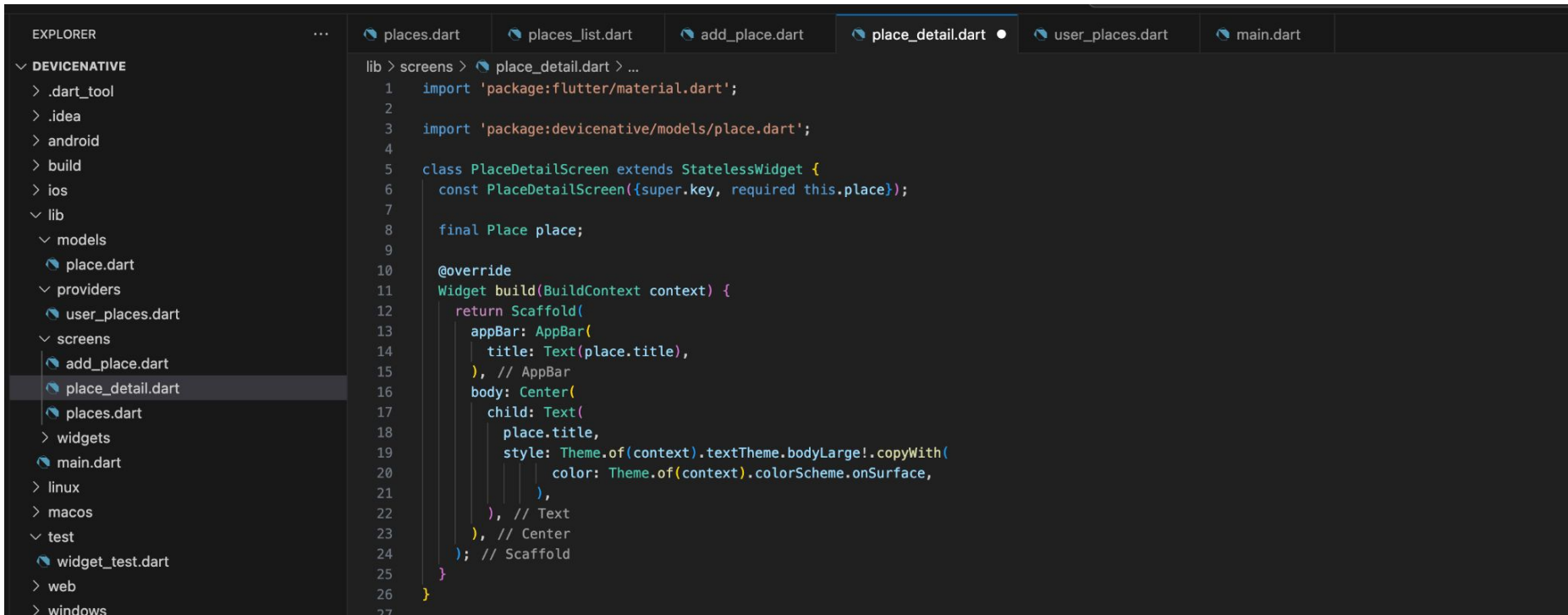


# Il risultato?



Identico a prima, ma stiamo usando un provider esterno, condiviso tra i vari widget, in grado di centralizzare i dati.

# Aggiunta di uno screen di dettaglio: PlaceDetailScreen



The screenshot shows an IDE with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like .dart\_tool, .idea, android, build, ios, lib, models, providers, screens, widgets, and test. The 'screens' folder is expanded, showing 'add\_place.dart', 'place\_detail.dart' (selected), and 'places.dart'. The code editor shows the implementation of 'PlaceDetailScreen' in 'place\_detail.dart'. The code includes imports for 'package:flutter/material.dart' and 'package:devicenative/models/place.dart'. It defines a 'PlaceDetailScreen' class that extends 'StatelessWidget'. The class has a constructor 'const PlaceDetailScreen({super.key, required this.place})' and a 'final Place place' property. The 'build' method returns a 'Scaffold' with an 'AppBar' containing a 'Text' widget for the place title, and a 'body' containing a 'Center' widget with a 'Text' widget for the place title, styled with the theme's bodyLarge text and onSurface color.

```
lib > screens > place_detail.dart > ...
1  import 'package:flutter/material.dart';
2
3  import 'package:devicenative/models/place.dart';
4
5  class PlaceDetailScreen extends StatelessWidget {
6    const PlaceDetailScreen({super.key, required this.place});
7
8    final Place place;
9
10   @override
11   Widget build(BuildContext context) {
12     return Scaffold(
13       appBar: AppBar(
14         title: Text(place.title),
15       ), // AppBar
16       body: Center(
17         child: Text(
18           place.title,
19           style: Theme.of(context).textTheme.bodyLarge!.copyWith(
20             color: Theme.of(context).colorScheme.onSurface,
21           ),
22         ), // Text
23       ), // Center
24     ); // Scaffold
25   }
26 }
```

# Aggiunta di uno screen di dettaglio: PlaceDetailScreen

- Il metodo build di PlaceDetailScreen costruisce l'interfaccia utente del widget. Utilizza un Scaffold per creare la struttura di base dell'interfaccia utente, che include una AppBar e un body;
- La AppBar ha un titolo che è il titolo del luogo;
- Il corpo del Scaffold è un Center che contiene un Text widget. Questo Text widget visualizza il titolo del luogo. Il testo è stilizzato utilizzando il tema corrente del context.



# Aggiorniamo il widget PlacesList

```
return ListView.builder(  
  itemCount: places.length,  
  itemBuilder: (ctx, index) => ListTile(  
    title: Text(  
      places[index].title,  
      style: Theme.of(context).textTheme.titleMedium!.copyWith(  
        color: Theme.of(context).colorScheme.onSurface,  
      ),  
    ), // Text  
    onTap: () {  
      Navigator.of(context).push(  
        MaterialPageRoute(  
          builder: (ctx) => PlaceDetailScreen(place: places[index]),  
        ), // MaterialPageRoute  
      );  
    },  
  ), // ListTile  
); // ListView.builder  
}
```

# Aggiorniamo il widget PlacesList

È stato semplicemente integrato un gestore di eventi onTap per un widget, quando l'utente tocca il widget, viene eseguito il codice per mandare tutto alla schermata di dettaglio appena creata, PlaceDetailScreen.




Abbiamo integrato il  
nostro base project che  
supporta anche un  
provider esterno per gli  
stati, e ora?

# Pausa (di riflessione)

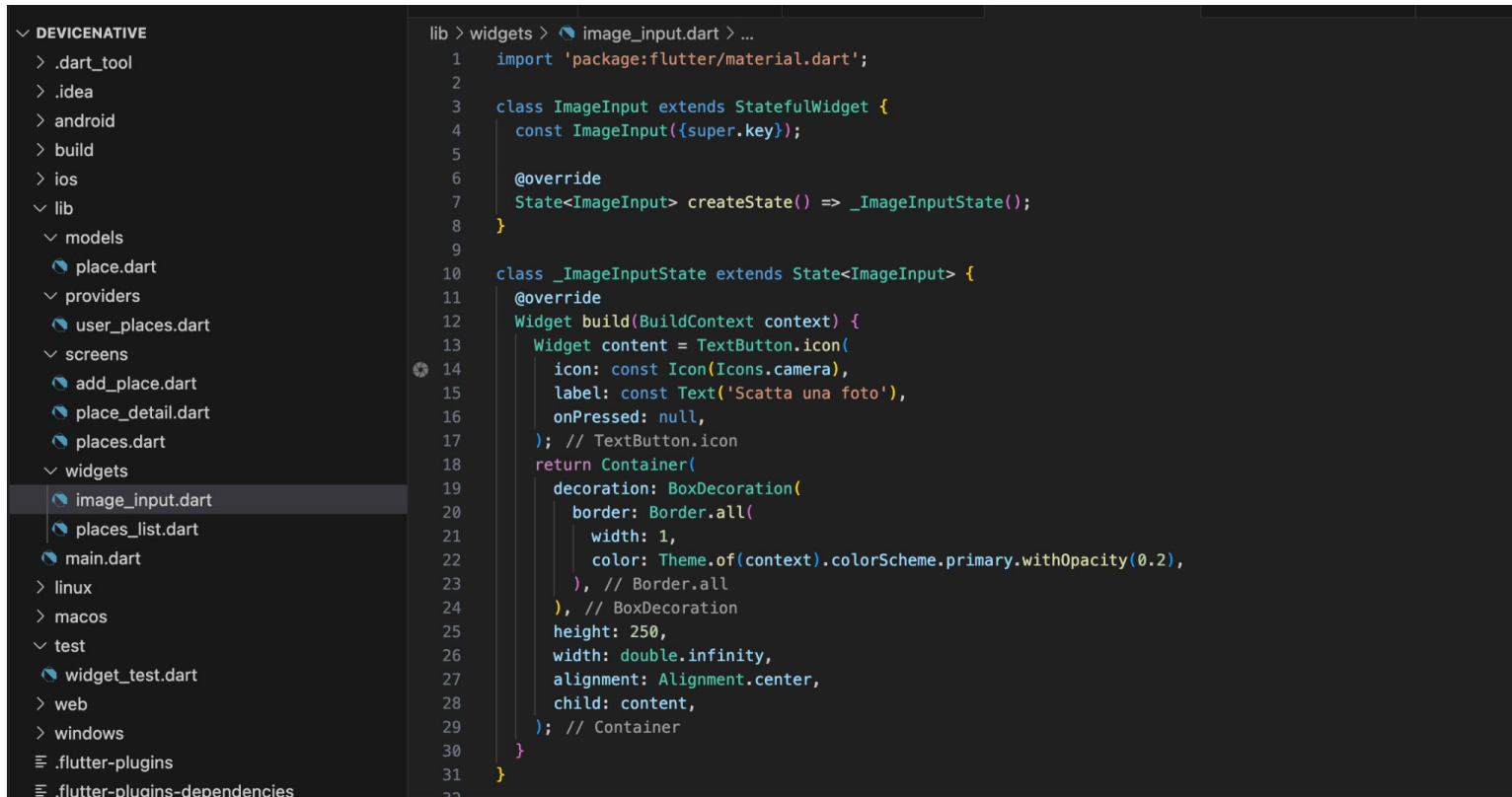
Usate 10 minuti per scaricare il file riverpod.zip e guardare quanto mostrato fino ad ora a lezione. Pensate a come adattare quanto visto al vostro progetto.





Acquisizione immagini  
dalla camera del device

# Creazione di un nuovo widget: ImageInput



# Creazione di un nuovo widget: ImageInput

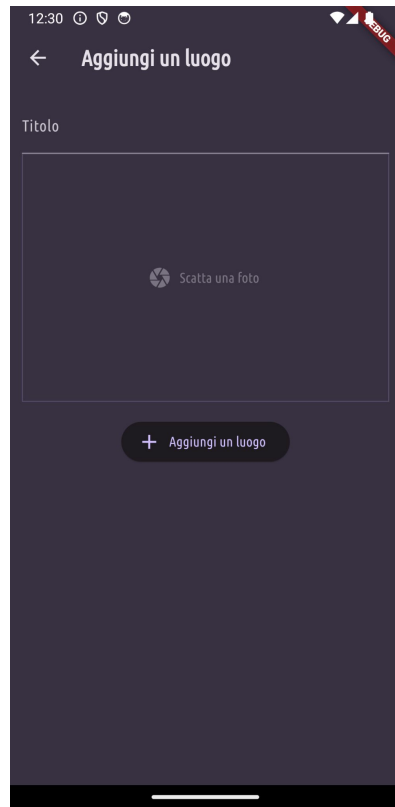
Al momento è un semplice widget che mostra un'icona ed un testo...



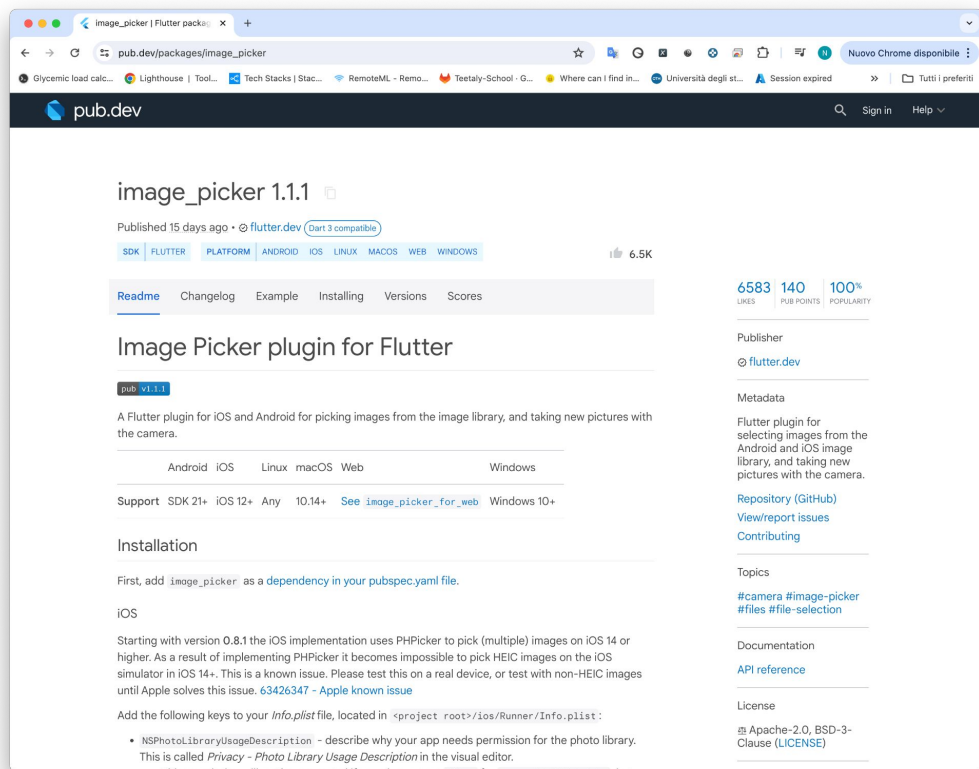
# Lo richiamiamo all'interno di AddPlaceScreen

```
style: TextStyle(  
  color: Theme.of(context).colorScheme.onSurface,  
), // TextStyle  
) , // TextField  
const ImageInput(),  
const SizedBox(height: 16),  
ElevatedButton.icon(  

```



# image\_picker - [https://pub.dev/packages/image\\_picker](https://pub.dev/packages/image_picker)



The screenshot shows the official pub.dev page for the `image_picker` package. The page is titled "image\_picker 1.1.1" and indicates it was published 15 days ago. It is compatible with Flutter and Dart 3. The package has 6,583 likes, 140 pub points, and 100% popularity. The publisher is flutter.dev. The package is described as a Flutter plugin for iOS and Android for picking images from the image library and taking new pictures with the camera. It supports Android, iOS, Linux, macOS, Web, and Windows. The installation instructions are provided for both Android and iOS. The Android section shows the dependency in the `pubspec.yaml` file. The iOS section shows the dependency in the `Info.plist` file. The license is Apache-2.0, BSD-3-Clause (LICENSE).

image\_picker 1.1.1

Published 15 days ago • flutter.dev (Dart 3 compatible)

SDK FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS 6.5K

Readme Changelog Example Installing Versions Scores

## Image Picker plugin for Flutter

pub 1.1.1

A Flutter plugin for iOS and Android for picking images from the image library, and taking new pictures with the camera.

Platform	Support
Android	SDK 21+
iOS	iOS 12+
Linux	Any
macOS	10.14+
Web	See <a href="#">image_picker_for_web</a>
Windows	Windows 10+

### Installation

First, add `image_picker` as a dependency in your `pubspec.yaml` file.

#### iOS

Starting with version 0.8.1 the iOS implementation uses PHPicker to pick (multiple) images on iOS 14 or higher. As a result of implementing PHPicker it becomes impossible to pick HEIC images on the iOS simulator in iOS 14+. This is a known issue. Please test this on a real device, or test with non-HEIC images until Apple solves this issue. [63426347 - Apple known issue](#)

Add the following keys to your `Info.plist` file, located in `<project root>/ios/Runner/Info.plist`:

- `NSPhotoLibraryUsageDescription` - describe why your app needs permission for the photo library. This is called *Privacy - Photo Library Usage Description* in the visual editor.

6583 140 100%

LIKES PUB POINTS POPULARITY

Publisher

flutter.dev

Metadata

Flutter plugin for selecting images from the Android and iOS image library, and taking new pictures with the camera.

Repository (GitHub)  
View/report issues  
Contributing

Topics

#camera #image-picker #files #file-selection

Documentation

API reference

License

Apache-2.0, BSD-3-Clause (LICENSE)

# Esercizio

Provate a vedere la documentazione di `image_picker`. Durante la prossima lezione mostreremo una possibile implementazione.