Flutter e Dart - Le basi

Flutter - Gestione dello stato

Luigi Durso

luigi.durso@si2001.it

SI2001

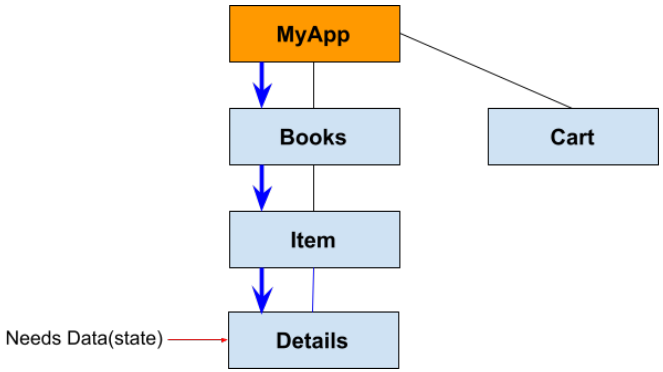
20 ottobre 2022

Sommario

Un po' di codice

Analizziamo l'elaborato precedente!

Stato condiviso



Qual'è il problema?

- Dati condivisi tra più livelli dell'albero dei widgets
- Richiesta di passaggio di parametri anche in widget che non ne hanno bisogno
- Re-build eseguita su tutto il percorso di dipendenze

Idee di soluzione

Un possibile approccio di soluzione

- Pensare allo stato disaccoppiato dai widgets
- Interazione tra widget e stato attraverso eventi e funzioni
- Ogni widget si sottoscrive ai cambiamenti della parte di stato a cui è interessato

Soluzioni

Package	Likes	Pub Points	Popularity	Flutter Favorite
	2022.02.04			
provider	5,972	130	100%	YES
bloc	4,937* (3,468 + 1,469)	130	100%	YES
get	7,861	120	100%	-
riverpod	1,353	130	97%	-
get_it	1,873	130	100%	-
redux	613** (338 + 275)	120	97%	YES
mobx	1,242*** (452 + 790)	130	98%	YES

BLoC

Bloc

(**B**usiness **Lo**gic **C**omponent)

Perché usare BLoC

Bloc segue tre principi fondamentali:

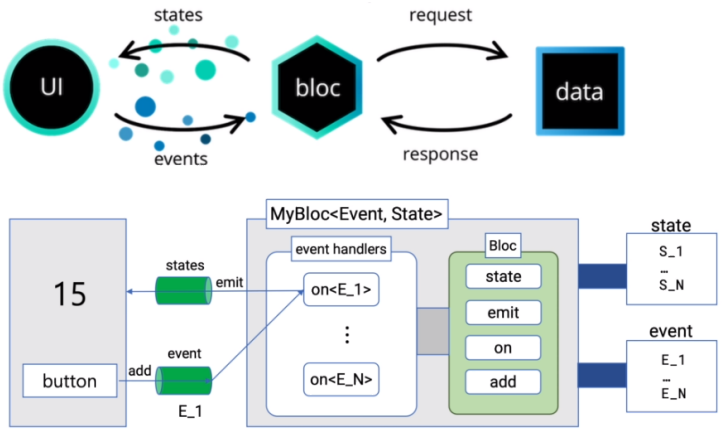
- **Semplicità**: Facile da imparare, chiunque può approcciarsi facilmente a questa tecnologia.
- **Potenza**: Attraverso la creazione di piccoli componenti si può creare un'applicativo di grande complessità.
- **Testabilità**: Ogni aspetto dell'applicativo è facilmente testabile.

Approcci diversi

Possiamo usare approcci diversi in base alle nostre esigenze:

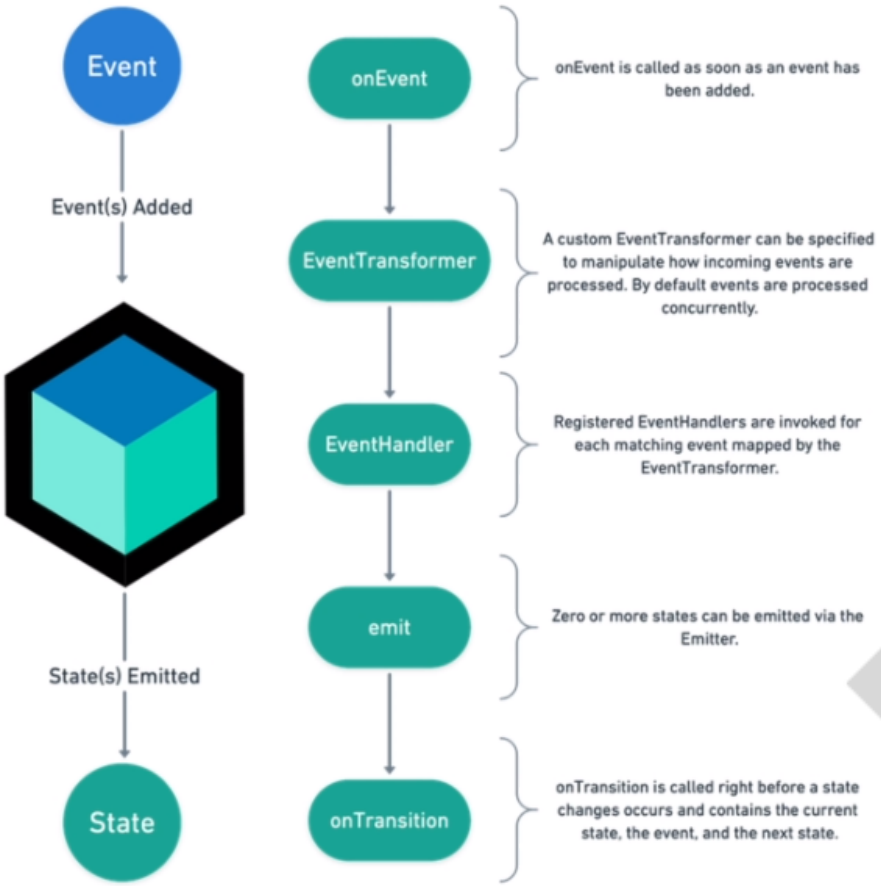
- Approccio event-driven con **BLoC**
- Approccio basato su funzioni con **Cubit**

Struttura di BLoC



- Lettura degli eventi scatenati dalla UI
- Ricezione dell'evento e gestione mediante il giusto event handler
- Elaborazione dell'evento (Es. Interazione con Repository)
- Rendering del nuovo stato nella UI

Cosa possiamo osservare



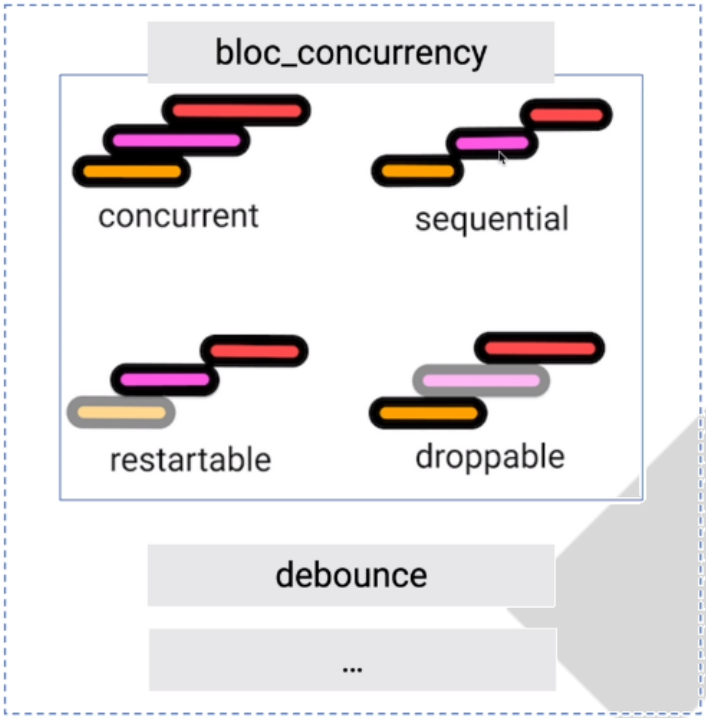

```
print('$event');

IncrementCounterEvent
```

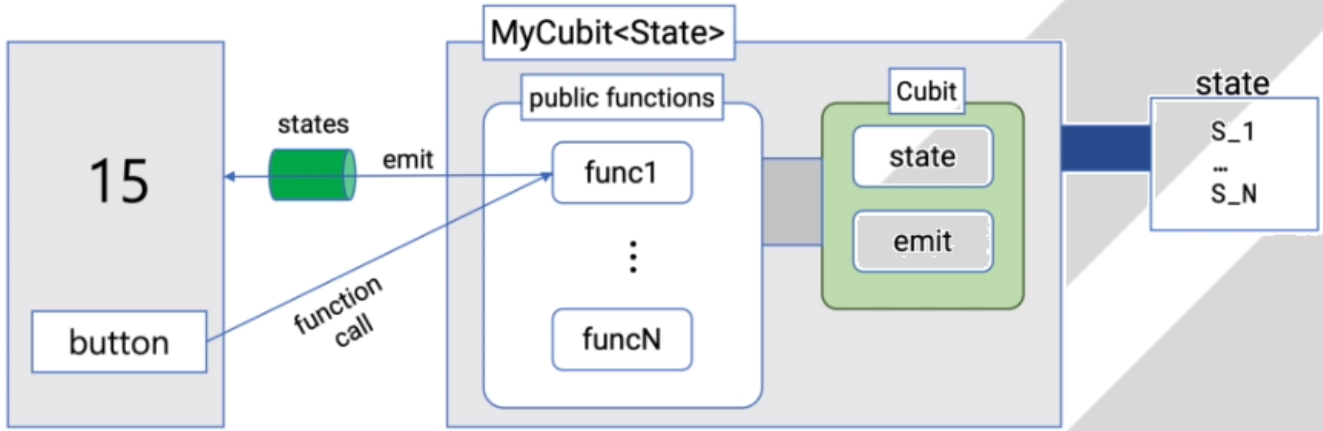
```
print('$transition');

Transitioin {
  currentState: currentVal,
  event: IncrementCounterEvent
  nextState: nextVal
}
```

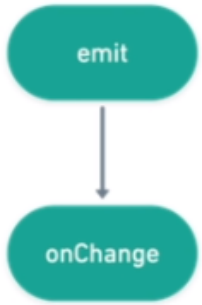
Event transformation



Struttura di Cubit



Cosa possiamo osservare



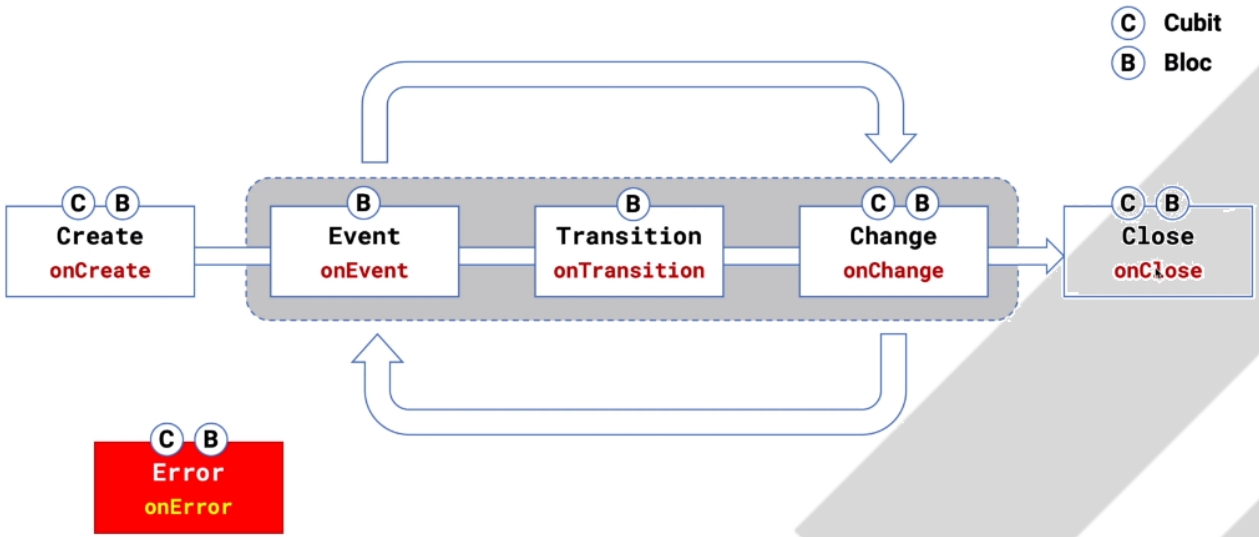
emit new states in response to function calls.

onChange is called right before a state change occurs and contains the current and next state.

```
print('$change');

Change { currentState: currentVal, nextState: nextVal }
```

Cosa possiamo osservare - Confronto



Cubit vs BLoC

	Cubit	Bloc
Simplicity	• states • functions	• states • events • event handlers
Traceability		• Transition (onTransition)
Advanced Event Transformations		• EventTransformer

- Una struttura ad eventi necessità di complessità aggiuntiva.
- Non sempre si ha bisogno di una gestione complessa.
- Si può sempre optare per soluzioni ibride, **BLoC più Cubit**

Rendere disponibile BLoC

- Instance of cubit/bloc

```
BlocProvider<CounterCubit>(
  create: (context) => CounterCubit(),
  child: …,
)
```

```
BlocProvider<CounterBloc>(
  create: (context) => CounterBloc(),
  child: …,
)
```

  - lazy

- Access cubit/bloc instance
  - of static method of BlocProvider

```
BlocProvider.of<CounterCubit>(context)
```

```
BlocProvider.of<CounterBloc>(context)
```

- Attraverso il provider rendiamo iniettabile nel sottoalbero il BLoC.
- Dal context recuperiamo il BLoC per l'utilizzo.

BlocBuilder

```
BlocBuilder<BlocA, BlocAState>(
  bloc: blocA,  // provide the local bloc instance
  buildWhen: (previousState, state) {
    // return true/false to determine whether or not
    // to rebuild the widget with state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

BlocListener

```
BlocListener<BlocA, BlocAState>( a
  bloc: blocA, // provide the local bloc instance
  listenWhen: (previousState, state) {
    // return true/false to determine whether or not
    //to call listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

BlocConsumer

```
BlocConsumer<BlocA, BlocAState>( a
  bloc: blocA, // provide the local bloc instance
  listenWhen: (previous, current) {
    // return true/false to determine whether or not
    // to call listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  buildWhen: (previous, current) {
    // return true/false to determine whether or not
    // to rebuild the widget with state
  },
  builder: (BuildContext context, BlocAState state) {
    // return widget here based on BlocA's state
  },
)
```

Extension Methods

- context.watch<T>(), which makes the widget listen to changes on T
- context.read<T>(), which returns T without listening to it
- context.select<T, R>(R cb(T value)), which allows a widget to listen to only a small part of T.

```
Widget build(BuildContext context) {
  final person = context.watch<Person>();
  return Text(person.name);
}
```

```
Widget build(BuildContext context) {
  final name = context.select((Person p) => p.name);
  return Text(name);
}
```

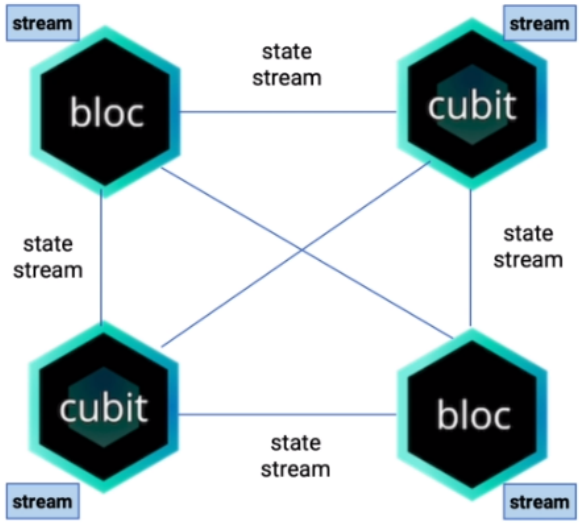
Watch vs BlocBuilder

```dart
@override
Widget build(BuildContext context) {
  final state = context.watch<MyBloc>().state;
  return Scaffold(
    body: Text('$state'),
  );
}
```

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: BlocBuilder<MyBloc, MyState>(
      builder: (context, state) => Text('$state'),
    ),
  );
}
```

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Builder(
      builder: (context) {
        final state = context.watch<MyBloc>().state;
        return Text('$state');
      },
    ),
  );
}
```

Cubits/BLoCs possono comunicare tra loro




```
myCubit.stream.listen((MyCubitState state) {
  // ...
})


myBloc.stream.listen((MyBlocState state) {
  // ...
})
```

Migliorare la precedente esercitazione

Introdurre una gestione di memoria con Bloc/Cubit

Grazie per l'attenzione!