Core Concepts (package:flutter_bloc)

Please make sure to carefully read the following sections before working with package:flutter_bloc.

Note: All widgets exported by the `flutter_bloc` package integrate with both `Cubit` and `Bloc` instances.

Bloc Widgets

BlocBuilder

BlocBuilder is a Flutter widget which requires a `Bloc` and a `builder` function. `BlocBuilder` handles building the widget in response to new states. `BlocBuilder` is very similar to `StreamBuilder` but has a more simple API to reduce the amount of boilerplate code needed. The `builder` function will potentially be called many times and should be a pure function that returns a widget in response to the state.

See `BlocListener` if you want to "do" anything in response to state changes such as navigation, showing a dialog, etc...

If the cubit parameter is omitted, `BlocBuilder` will automatically perform a lookup using `BlocProvider` and the current `BuildContext`.

```dart
BlocBuilder<BlocA, BlocAState>(
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Only specify the bloc if you wish to provide a bloc that will be scoped to a single widget and isn't accessible via a parent `BlocProvider` and the current `BuildContext`.

```dart
BlocBuilder<BlocA, BlocAState>(
  cubit: blocA, // provide the local cubit instance
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

For fine-grained control over when the `builder` function is called an optional `buildWhen` can be provided. `buildWhen` takes the previous bloc state and current bloc state and returns a boolean. If `buildWhen` returns true, `builder` will be called with `state` and the widget will rebuild. If `buildWhen` returns false, `builder` will not be called with `state` and no rebuild will occur.

```dart
BlocBuilder<BlocA, BlocAState>(
  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
  }
)
```

BlocProvider

BlocProvider is a Flutter widget which provides a bloc to its children via `BlocProvider.of<T>(context)`. It is used as a dependency injection (DI) widget so that a single instance of a bloc can be provided to multiple widgets within a subtree.

In most cases, `BlocProvider` should be used to create new blocs which will be made available to the rest of the subtree. In this case, since `BlocProvider` is responsible for creating the bloc, it will automatically handle closing the bloc.

```dart
BlocProvider(
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

By default, `BlocProvider` will create the bloc lazily, meaning `create` will get executed when the bloc is looked up via `BlocProvider.of<BlocA>(context)`.

To override this behavior and force `create` to be run immediately, `lazy` can be set to `false`.

```dart
BlocProvider(
  lazy: false,
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

In some cases, `BlocProvider` can be used to provide an existing bloc to a new portion of the widget tree. This will be most commonly used when an existing bloc needs to be made available to a new route. In this case, `BlocProvider` will not automatically close the bloc since it did not create it.

```dart
BlocProvider.value(
  value: BlocProvider.of<BlocA>(context),
  child: ScreenA(),
);
```

then from either `ChildA`, or `ScreenA` we can retrieve `BlocA` with:

```dart
// with extensions
context.read<BlocA>();

// without extensions
BlocProvider.of<BlocA>(context)
```

MultiBlocProvider

MultiBlocProvider is a Flutter widget that merges multiple `BlocProvider` widgets into one. `MultiBlocProvider` improves the readability and eliminates the need to nest multiple `BlocProviders`. By using `MultiBlocProvider` we can go from:

```dart
BlocProvider<BlocA>(
  create: (BuildContext context) => BlocA(),
  child: BlocProvider<BlocB>(
    create: (BuildContext context) => BlocB(),
    child: BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
      child: ChildA(),
    )
  )
)
```

to:

```dart
MultiBlocProvider(
  providers: [
    BlocProvider<BlocA>(
      create: (BuildContext context) => BlocA(),
    ),
    BlocProvider<BlocB>(
      create: (BuildContext context) => BlocB(),
    ),
    BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
    ),
  ],
  child: ChildA(),
)
```

BlocListener

BlocListener is a Flutter widget which takes a `BlocWidgetListener` and an optional `Bloc` and invokes the `listener` in response to state changes in the bloc. It should be used for functionality that needs to occur once per state change such as navigation, showing a `SnackBar` , showing a `Dialog` , etc...

`listener` is only called once for each state change (**NOT** including the initial state) unlike `builder` in `BlocBuilder` and is a `void` function.

If the cubit parameter is omitted, `BlocListener` will automatically perform a lookup using `BlocProvider` and the current `BuildContext` .

```dart
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

Only specify the bloc if you wish to provide a bloc that is otherwise not accessible via `BlocProvider` and the current `BuildContext` .

```dart
BlocListener<BlocA, BlocAState>(
  cubit: blocA,
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container()
)
```

For fine-grained control over when the `listener` function is called an optional `listenWhen` can be provided. `listenWhen` takes the previous bloc state and current bloc state and returns a boolean. If `listenWhen` returns true, `listener` will be called with `state` . If `listenWhen` returns false, `listener` will not be called with `state` .

```dart
BlocListener<BlocA, BlocAState>(
  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(),
  )
```

MultiBlocListener

MultiBlocListener is a Flutter widget that merges multiple `BlocListener` widgets into one. `MultiBlocListener` improves the readability and eliminates the need to nest multiple `BlocListeners`. By using `MultiBlocListener` we can go from:

```dart
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {},
  child: BlocListener<BlocB, BlocBState>(
    listener: (context, state) {},
    child: BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
      child: ChildA(),
    ),
  ),
)
```

to:

```dart
MultiBlocListener(
  listeners: [
    BlocListener<BlocA, BlocAState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocB, BlocBState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
    ),
  ],
  child: ChildA(),
)
```

BlocConsumer

BlocConsumer exposes a `builder` and `listener` in order to react to new states. `BlocConsumer` is analogous to a nested `BlocListener` and `BlocBuilder` but reduces the amount of boilerplate needed. `BlocConsumer` should only be used when it is necessary to both rebuild UI and execute other reactions to state changes in the `cubit` . `BlocConsumer` takes a required `BlocWidgetBuilder` and `BlocWidgetListener` and an optional `cubit` , `BlocBuilderCondition` , and `BlocListenerCondition` .

If the `cubit` parameter is omitted, `BlocConsumer` will automatically perform a lookup using `BlocProvider` and the current `BuildContext` .

```dart
BlocConsumer<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

An optional `listenWhen` and `buildWhen` can be implemented for more granular control over when `listener` and `builder` are called. The `listenWhen` and `buildWhen` will be invoked on each `cubit` `state` change. They each take the previous `state` and current `state` and must return a `bool` which determines whether or not the `builder` and/or `listener` function will be invoked. The previous `state` will be initialized to the `state` of the `cubit` when the `BlocConsumer` is initialized. `listenWhen` and `buildWhen` are optional and if they aren't implemented, they will default to `true` .

```dart
BlocConsumer<BlocA, BlocAState>(
  listenWhen: (previous, current) {
    // return true/false to determine whether or not
    // to invoke 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: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

RepositoryProvider

RepositoryProvider is a Flutter widget which provides a repository to its children via `RepositoryProvider.of<T>(context)`. It is used as a dependency injection (DI) widget so that a single instance of a repository can be provided to multiple widgets within a subtree. `BlocProvider` should be used to provide blocs whereas `RepositoryProvider` should only be used for repositories.

```dart
RepositoryProvider(
  create: (context) => RepositoryA(),
  child: ChildA(),
);
```

then from `ChildA` we can retrieve the `Repository` instance with:

```dart
// with extensions
context.read<RepositoryA>();
```

```
// without extensions
RepositoryProvider.of<RepositoryA>(context)
```

MultiRepositoryProvider

MultiRepositoryProvider is a Flutter widget that merges multiple `RepositoryProvider` widgets into one. `MultiRepositoryProvider` improves the readability and eliminates the need to nest multiple `RepositoryProvider`. By using `MultiRepositoryProvider` we can go from:

```dart
RepositoryProvider<RepositoryA>(
  create: (context) => RepositoryA(),
  child: RepositoryProvider<RepositoryB>(
    create: (context) => RepositoryB(),
    child: RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
      child: ChildA(),
    )
  )
)
```

to:

```dart
MultiRepositoryProvider(
  providers: [
    RepositoryProvider<RepositoryA>(
      create: (context) => RepositoryA(),
    ),
    RepositoryProvider<RepositoryB>(
      create: (context) => RepositoryB(),
    ),
    RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
    ),
  ],
  child: ChildA(),
)
```

Usage

Lets take a look at how to use `BlocBuilder` to hook up a `CounterPage` widget to a `CounterBloc`.

counter_bloc.dart

```dart
enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.decrement:
        yield state - 1;
        break;
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }
}
```

counter_page.dart

```dart
class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final CounterBloc counterBloc = BlocProvider.of<CounterBloc>(context)

    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: BlocBuilder<CounterBloc, int>(
        builder: (context, count) {
          return Center(
                child: Text(
                  '$count',
                  style: TextStyle(fontSize: 24.0),
                ),
              );
            },
          ),
          floatingActionButton: Column(
            crossAxisAlignment: CrossAxisAlignment.end,
            mainAxisAlignment: MainAxisAlignment.end,
            children: <Widget>[
              Padding(
                padding: EdgeInsets.symmetric(vertical: 5.0),
                child: FloatingActionButton(
                  child: Icon(Icons.add),
                  onPressed: () {
                    counterBloc.add(CounterEvent.increment);
                  },
                ),
              ),
              Padding(
                padding: EdgeInsets.symmetric(vertical: 5.0),
                child: FloatingActionButton(
                  child: Icon(Icons.remove),
                  onPressed: () {
                    counterBloc.add(CounterEvent.decrement);
                  },
                ),
              ),
            ],
          ),
        );
      }
    }
```

At this point we have successfully separated our presentational layer from our business logic layer. Notice that the `CounterPage` widget knows nothing about what happens when a user taps the buttons. The widget simply tells the `CounterBloc` that the user has pressed either the increment or decrement button.

‹ PREVIOUS
package:bloc