http://dream-implementation.com

info@dream-implementation.com

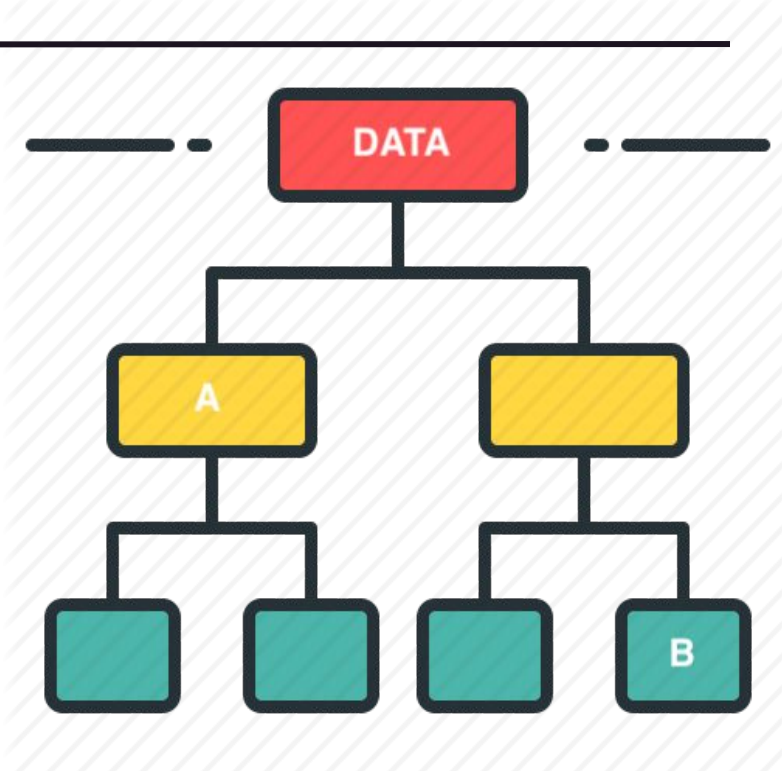
Provider - what is it?

- https://pub.dev/packages/provider
- Unofficial flutter plugin suggested by Google
- *“A mixture between dependency injection (DI) and state management, built with widgets for widgets”*
- With emergence and rising popularity of the BLoC architecture, provider focuses more on DI and leaves state management to the BLoC
- *“Provider is InheritedWidgets for humans”*

Provider - why use it?

Problem of accessing shared data (objects) at various points in widget tree

1. Passing parameters to widget constructors
2. Global variables & singletons
3. InheritedWidgets



Passing parameters - manual way

PROS

- Customizable
- Testable
- Reusable
- Safe

CONS

- Very verbose
- Not very efficient when rebuilding

```
class MainScreen extends StatelessWidget {
  const MainScreen({this.bloc});

  final Bloc bloc;

  @override
  Widget build(BuildContext context) {
    return Column(
        children: <Widget>[
          WidgetOne(bloc: bloc),
          WidgetTwo(bloc: bloc),
          WidgetThree(bloc: bloc),
        ]
    );
  }
}
```

Global variables & singletons - easy way

PROS

- Short
- Simple

```
class Bloc {
  const Bloc._internal();

  static final Bloc _instance = Bloc._internal();

  factory Bloc() => _instance;
}
```

CONS

- Hard to test
- Hard to reuse
- Requires a lot of discipline
- No built-in rebuild mechanism

InheritedWidgets - Flutter way

- InheritedWidget takes data in as parameter and stores it in it's widget **state**.
- Data is than **accessed** by searching the **BuildContext** for nearest InheritedWidget ancestor with *InheritedWidget.of(context)* syntax
- Frequently used in Flutter - *Navigator.of(), Theme.of(), Localization.of()*...

```
void main() {
  runApp(
      MyInheritedWidget(
        bloc: Bloc(),
        child: MyApp(),
      )
  );
}
```

```
Bloc _bloc;

@override
Widget build(BuildContext context) {
  _bloc ??= MyInheritedWidget.of(context).bloc;

  return Scaffold(
```

InheritedWidgets

PROS

- Very efficient
- Less code overall
- Idiomatic to Flutter

CONS

- Lots of boilerplate
- Very nested code
- Hard to understand

```
class MyInheritedWidget extends InheritedWidget {
  const MyInheritedWidget({
    Key key,
    @required this.bloc,
    @required Widget child,
  }) : super(key: key, child: child);

  final Bloc bloc;

  static MyInheritedWidget of(BuildContext context) {
    return context.dependOnInheritedWidgetOfExactType(aspect: Bloc);
  }

  @override
  bool updateShouldNotify(MyInheritedWidget old) {
    return bloc != old.bloc;
  }
}
```

Provider

- **wrapper around InheritedWidget** - comes with all benefits and no major drawbacks of IW (easy to understand, implement and reuse without all the boilerplate)
- **compatible with BLoC architecture** (provider handles dependency injection and BLoC handles state managment)
- **many different types** of provider for various use cases (Provider, ListenableProvider, ChangeNotifierProvider, StreamProvider, MultiProvider...)

Provider - exposing a value

- creates an object in **create()** function and exposes it to the widget tree
- disposes of the object in **dispose()** function if needed
- all objects are **compatible** with provider
- provided object is created **lazily** - the first time it is accessed

```
void main() {
  runApp(
      Provider<Bloc>(
          create: (BuildContext context) => Bloc(),
          dispose: (BuildContext context, Bloc bloc) => bloc?.dispose(),
          child: MyApp()
      )
  );
}
```

Provider - exposing an existing value

- named constructor **Provider<T>.value**
- takes an existing object and exposes it's reference to the widget tree

```
final Bloc _bloc = Bloc();

void main() {
  runApp(
      Provider<Bloc>.value(
          value: _bloc,
          child: MyApp()
      )
  );
}
```

Provider - reading a value

- access provided objects from any widget in the widget tree using short and simple syntax
- **Provider.of<T>(context)**

```
Bloc _bloc;

@override
Widget build(BuildContext context) {
  _bloc ??= Provider.of<Bloc>(context);

  return Scaffold(
```

Provide multiple objects?

Nested providers? Just no.

```
@override
Widget build(BuildContext context) {
  return Provider<MainBloc>(
    create: (BuildContext context) => MainBloc(),
    child: Provider<LoginBloc>(
      create: (BuildContext context) => LoginBloc(),
      child: Provider<FeedBloc>(
        create: (BuildContext context) => FeedBloc(),
        child: MaterialApp(
```

MultiProvider

Convenience widget - provides more than one object without nesting

```
@override
Widget build(BuildContext context) {
  return MultiProvider(
    providers: [
      Provider<MainBloc>(create: (BuildContext context) => MainBloc()),
      Provider<LoginBloc>(create: (BuildContext context) => LoginBloc()),
      Provider<FeedBloc>(create: (BuildContext context) => FeedBloc()),
    ],
    child: MaterialApp(
```

Async Providers

- For exposing data that is obtained asyncronously
- **FutureProvider** - takes an async function and provides the result
- **StreamProvider** - takes a stream and provides the result
- Similar to FutureBuilder and StreamBuilder

```
FutureProvider(
    initialData: loadInitialData(),
    create: (context) => fetchSomethingAsync(),
    child: MyApp()
)
```

```
StreamProvider(
    initialData: loadInitialData(),
    create: (context) => _myStreamController.stream,
    child: MyApp()
)
```

ChangeNotifierProvider and Consumer

- unlike the basic Provider widget, **ChangeNotifierProvider** listens for changes in the provided object
- if there are changes, it rebuilds all widgets under the **Consumer** widget

```
ChangeNotifierProvider<User>(
    create: (BuildContext context) => User('John', 'Doe'),
    child: Consumer<User>(
        builder: (BuildContext context, User user, Widget child) {
            return Text('${user.firstName} ${user.lastName}');
        },
    )
)
```

Context extensions (dev)

- currently available in the latest version of provider flutter dev channel (provider: 4.1.0-dev+3), limited to flutter dev channel
- simplified syntax for reading provided values
- Implementation of extension functions (feature of dart 2.7.0 - add new functionality to existing flutter classes - in this case BuildContext)

  `User user = context.read<User>()` - obtains the value

  `User user = context.watch<User>()` - obtains and subscribes to it, rebuilding the widget tree on changes

  `String name = context.select((user) => user.name)` - like context.watch, but focuses on a single property

Summary

- **What is Provider?** A wrapper around InheritedWidget which is easy to understand, implement and reuse
- **Should you use it?** Probably, if you like your code clean and maintainable
- **What for?** We find it works best in conjunction with BLoC architecture for dependancy injection, but it is not limited to that purpose

flutter_bloc

- https://pub.dev/packages/flutter_bloc
- Uses Provider to provide bloc instances to UI

BlocProvider

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

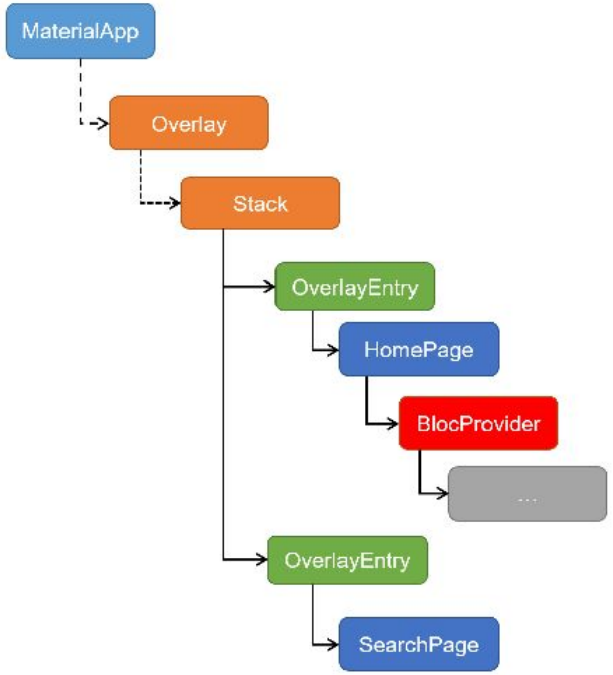
- from ChildA retrieve bloc via:

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

BlocProvider.value

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

- provides existing bloc instance, doesn't close it on dispose
- useful when going to another widget via Navigator



BlocBuilder

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

- optional parameters: bloc (local instance), condition (callback which returns true/false to determine whether or not to rebuild the widget with state)

BlocListener

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

- optional parameters: bloc (local instance), condition (callback which returns true/false to determine whether or not to call listener with state)
- for functionality that needs to occur once per state change, e.g. navigation, showing a SnackBar, showing a Dialog
- MultiBlocListener

BlocConsumer

```
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
  }
)
```

- optional parameters: bloc (local instance), listenWhen (callback which returns true/false to determine whether or not to invoke listener with state), buildWhen (callback which returns true/false to determine whether or not to rebuild the widget with state)

Useful links

- Flutter Europe Provider
  - https://www.youtube.com/watch?v=BuIIREvHBWg
- https://www.didierboelens.com/2019/07/provider-points-of-interest-points-to-care-about/
- https://medium.com/flutter-community/making-sense-all-of-those-flutter-providers-e842e18f45dd
- https://medium.com/fluttervn/simplify-flutter-state-management-with-provider-and-bloc-dcfad49bedf2
- Flutter Europe Bloc
  - https://www.youtube.com/watch?v=knMvKPKBzGE
- https://bloclibrary.dev

Thank you

Ivan Celija: ivan@dream-implementation.com

Goran Kovač: goran@dream-implementation.com