Recipes: Bloc Access

> In this recipe, we're going to take a look at how to use `BlocProvider` to make a bloc accessible throughout the widget tree. We're going to explore three scenarios: Local Access, Route Access, and Global Access.

Local Access

> In this example, we're going to use `BlocProvider` to make a bloc available to a local sub-tree. In this context, local means within a context where there are no routes being pushed/popped.

Bloc

For the sake of simplicity we're going to use a `Counter` as our example application.

Our `CounterBloc` implementation will look like:

```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 currentState - 1;
        break;
      case CounterEvent.increment:
        yield currentState + 1;
```

```
          break;
      }
    }
  }
```

UI

We're going to have 3 parts to our UI:

- App: the root application widget
- CounterPage: the container widget which will manage the `CounterBloc` and exposes `FloatingActionButtons` to `increment` and `decrement` the counter.
- CounterText: a text widget which is responsible for displaying the current `count`.

App

Copy to clipboard

```
import 'package:flutter/material.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

void main() => runApp(App());

class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: BlocProvider(
        create: (BuildContext context) => CounterBloc(),
        child: CounterPage(),
      ),
    );
  }
}
```

Our `App` widget is a `StatelessWidget` that uses a `MaterialApp` and sets our `CounterPage` as the home widget. The `App` widget is responsible for creating and

closing the `CounterBloc` as well as making it available to the `CounterPage` using a `BlocProvider`.

> **Note:** When we wrap a widget with `BlocProvider` we can then provide a bloc to all widgets within that subtree. In this case, we can access the `CounterBloc` from within the `CounterPage` widget and any children of the `CounterPage` widget using `BlocProvider.of<CounterBloc>(context)`.

CounterPage

```dart
class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterBloc = BlocProvider.of<CounterBloc>(context);
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: Center(
        child: CounterText(),
      ),
      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);
              },
            ),
```

```
            ),
          ],
        ),
      );
    }
  }
```

The `CounterPage` widget is a `StatelessWidget` which is accesses the `CounterBloc` via the `BuildContext`.

CounterText

```dart
class CounterText extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<CounterBloc, int>(
      builder: (context, count) {
        return Text('$count');
      },
    );
  }
}
```

Our `CounterText` widget is using a `BlocBuilder` to rebuild itself whenever the `CounterBloc` state changes. We use `BlocProvider.of<CounterBloc>(context)` in order to access the provided `CounterBloc` and return a `Text` widget with the current count.

That wraps up the local bloc access portion of this recipe and the full source code can be found here.

Next, we'll take a look at how to provide a bloc across multiple pages/routes.

Anonymous Route Access

> In this example, we're going to use `BlocProvider` to access a bloc across routes. When a new route is pushed, it will have a different `BuildContext`

> which no longer has a reference to the previously provided blocs. As a result, we have to wrap the new route in a separate `BlocProvider`.

Bloc

Again, we're going to use the `CounterBloc` for simplicity.

```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 currentState - 1;
        break;
      case CounterEvent.increment:
        yield currentState + 1;
        break;
    }
  }
}
```

UI

Again, we're going to have three parts to our application's UI:

- App: the root application widget
- HomePage: the container widget which will manage the `CounterBloc` and exposes `FloatingActionButtons` to `increment` and `decrement` the counter.
- CounterPage: a widget which is responsible for displaying the current `count` as a separate route.

App

```dart
import 'package:flutter/material.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

void main() => runApp(App());

class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: BlocProvider(
        create: (BuildContext context) => CounterBloc(),
        child: HomePage(),
      ),
    );
  }
}
```

Again, our `App` widget is the same as before.

HomePage

```dart
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterBloc = BlocProvider.of<CounterBloc>(context);
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.of(context).push(
              MaterialPageRoute<CounterPage>(
                builder: (context) {
                  return BlocProvider.value(
                    value: counterBloc,
                    child: CounterPage(),
                  );
```

```
                },
              ),
            );
          },
          child: Text('Counter'),
        ),
      ),
      floatingActionButton: Column(
        crossAxisAlignment: CrossAxisAlignment.end,
        mainAxisAlignment: MainAxisAlignment.end,
        children: <Widget>[
          Padding(
            padding: EdgeInsets.symmetric(vertical: 5.0),
            child: FloatingActionButton(
              heroTag: 0,
              child: Icon(Icons.add),
              onPressed: () {
                counterBloc.add(CounterEvent.increment);
              },
            ),
          ),
          Padding(
            padding: EdgeInsets.symmetric(vertical: 5.0),
            child: FloatingActionButton(
              heroTag: 1,
              child: Icon(Icons.remove),
              onPressed: () {
                counterBloc.add(CounterEvent.decrement);
              },
            ),
          ),
        ],
      ),
    );
  }
}
```

The `HomePage` is similar to the `CounterPage` in the above example; however, instead of rendering a `CounterText` widget, it renders a `ElevatedButton` in the center which allows the user to navigate to a new screen which displays the current count.

When the user taps the `ElevatedButton`, we push a new `MaterialPageRoute` and return the `CounterPage`; however, we are wrapping the `CounterPage` in a `BlocProvider` in order to make the current `CounterBloc` instance available on the next page.

> It is critical that we are using `BlocProvider's` value constructor in this case because we are providing an existing instance of `CounterBloc`. The value constructor of `BlocProvider` should be used only in cases where we want to provide an existing bloc to a new subtree. In addition, using the value constructor will not close the bloc automatically which, in this case, is what we want (since we still need the `CounterBloc` to function in the ancestor widgets). Instead, we simply pass the existing `CounterBloc` to the new page as an existing value as opposed to in a builder. This ensures that the only top level `BlocProvider` handles closing the `CounterBloc` when it is no longer needed.

CounterPage

```dart
class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter'),
      ),
      body: BlocBuilder<CounterBloc, int>(
        builder: (context, count) {
          return Center(
            child: Text('$count'),
          );
        },
      ),
    );
  }
}
```

`CounterPage` is a super simple `StatelessWidget` which uses `BlocBuilder` to re-render a `Text` widget with the current count. Just like before, we are able to use

`BlocProvider.of<CounterBloc>(context)` in order to access the `CounterBloc`.

That's all there is to this example and the full source can be found here.

Next, we'll look at how to scope a bloc to just one or more named routes.

Named Route Access

> In this example, we're going to use `BlocProvider` to access a bloc across multiple named routes. When a new named route is pushed, it will have a different `BuildContext` (just like before) which no longer has a reference to the previously provided blocs. In this case, we're going to manage the blocs which we want to scope in the parent widget and selectively provide them to the routes that should have access.

Bloc

Again, we're going to use the `CounterBloc` for simplicity.

```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 currentState - 1;
        break;
      case CounterEvent.increment:
        yield currentState + 1;
        break;
    }
  }
}
```

UI

Again, we're going to have three parts to our application's UI:

- App: the root application widget which manages the `CounterBloc` and provides it to the appropriate named routes.
- HomePage: the container widget which accesses the `CounterBloc` and exposes `FloatingActionButtons` to `increment` and `decrement` the counter.
- CounterPage: a widget which is responsible for displaying the current `count` as a separate route.

App

```dart
import 'package:flutter/material.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

void main() => runApp(App());

class App extends StatefulWidget {
  @override
  _AppState createState() => _AppState();
}

class _AppState extends State<App> {
  final _counterBloc = CounterBloc();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      routes: {
        '/': (context) => BlocProvider.value(
              value: _counterBloc,
              child: HomePage(),
            ),
        '/counter': (context) => BlocProvider.value(
              value: _counterBloc,
              child: CounterPage(),
            ),
      },
```

```
      );
    }

    @override
    void dispose() {
      _counterBloc.close();
      super.dispose();
    }
  }
```

Our `App` widget is responsible for managing the instance of the `CounterBloc` which we'll be providing to the root (`/`) and counter (`/counter`) routes.

> ! It's critical to understand that since the `_AppState` is creating the `CounterBloc` instance it should also be closing it in the `dispose` override.

> ! We're using `BlocProvider.value` when providing the `CounterBloc` instance to the routes because we don't want the `BlocProvider` to handle disposing the bloc (since `_AppState` is responsible for that).

HomePage

dart

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterBloc = BlocProvider.of<CounterBloc>(context);
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: Center(
        child: ElevatedButton(
          onPressed: () => Navigator.of(context).pushNamed('/counter'),
          child: Text('Counter'),
        ),
      ),
      floatingActionButton: Column(
        crossAxisAlignment: CrossAxisAlignment.end,
        mainAxisAlignment: MainAxisAlignment.end,
```

```
          children: <Widget>[
            Padding(
              padding: EdgeInsets.symmetric(vertical: 5.0),
              child: FloatingActionButton(
                heroTag: 0,
                child: Icon(Icons.add),
                onPressed: () {
                  counterBloc.add(CounterEvent.increment);
                },
              ),
            ),
            Padding(
              padding: EdgeInsets.symmetric(vertical: 5.0),
              child: FloatingActionButton(
                heroTag: 1,
                child: Icon(Icons.remove),
                onPressed: () {
                  counterBloc.add(CounterEvent.decrement);
                },
              ),
            ),
          ],
        ),
      );
    }
  }
```

The `HomePage` is similar above example; however, when the user taps the `ElevatedButton`, we push a new named route to navigate to the `/counter` route we defined above.

CounterPage

```dart
class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter'),
      ),
```

```
        body: BlocBuilder<CounterBloc, int>(
          builder: (context, count) {
            return Center(
              child: Text('$count'),
            );
          },
        ),
      );
    }
  }
```

`CounterPage` is a super simple `StatelessWidget` which uses `BlocBuilder` to re-render a `Text` widget with the current count. Just like before, we are able to use `BlocProvider.of<CounterBloc>(context)` in order to access the `CounterBloc`.

That's all there is to this example and the full source can be found here.

Next, we'll look at how to create a `Router` to manage and scope a bloc to just one or more generated routes.

Generated Route Access

> In this example, we're going to create a `Router` and use `BlocProvider` to access a bloc across multiple generated routes. We're going to manage the blocs which we want to scope in the `Router` and selectively provide them to the routes that should have access.

Bloc

Again, we're going to use the `CounterBloc` for simplicity.

```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 currentState - 1;
        break;
      case CounterEvent.increment:
        yield currentState + 1;
        break;
    }
  }
}
```

UI

Again, we're going to have three parts to our application's UI but we're also going to add an `AppRouter`:

- App: the root application widget which manages the `AppRouter`.
- AppRouter: class which will manage and provide the `CounterBloc` to the appropriate generated routes.
- HomePage: the container widget which accesses the `CounterBloc` and exposes `FloatingActionButtons` to `increment` and `decrement` the counter.
- CounterPage: a widget which is responsible for displaying the current `count` as a separate route.

App

dart

```
import 'package:flutter/material.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

void main() => runApp(App());

class App extends StatefulWidget {
  @override
  _AppState createState() => _AppState();
}

class _AppState extends State<App> {
  final _router = AppRouter();
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      onGenerateRoute: _router.onGenerateRoute,
    );
  }

  @override
  void dispose() {
    _router.dispose();
    super.dispose();
  }
}
```

Our `App` widget is responsible for managing the instance of the `AppRouter` and uses the router's `onGenerateRoute` to determine the current route.

> We need to dispose the `_router` when the `App` widget is disposed in order to close all blocs in the `AppRouter`.

App Router

dart

```
class AppRouter {
  final _counterBloc = CounterBloc();

  Route onGenerateRoute(RouteSettings settings) {
    switch (settings.name) {
      case '/':
        return MaterialPageRoute(
          builder: (_) => BlocProvider.value(
            value: _counterBloc,
            child: HomePage(),
          ),
        );
      case '/counter':
        return MaterialPageRoute(
          builder: (_) => BlocProvider.value(
```

```
            value: _counterBloc,
            child: CounterPage(),
          ),
        );
      default:
        return null;
    }
  }

  void dispose() {
    _counterBloc.close();
  }
}
```

Our `AppRouter` is responsible for managing the instance of the `CounterBloc` and provides `onGenerateRoute` which returns the correct route based on the provided `RouteSettings`.

> Since the `AppRouter` creates the `CounterBloc` instance it must also expose a `dispose` which `closes` the `CounterBloc` instance. `dispose` is called from the `_AppState` widget's `dispose` override.

> We're using `BlocProvider.value` when providing the `CounterBloc` instance to the routes because we don't want the `BlocProvider` to handle disposing the bloc (since `AppRouter` is responsible for that).

HomePage

```dart
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterBloc = BlocProvider.of<CounterBloc>(context);
    return Scaffold(
      appBar: AppBar(title: Text('Counter')),
      body: Center(
        child: ElevatedButton(
          onPressed: () => Navigator.of(context).pushNamed('/counter'),
```

```
            child: Text('Counter'),
          ),
        ),
        floatingActionButton: Column(
          crossAxisAlignment: CrossAxisAlignment.end,
          mainAxisAlignment: MainAxisAlignment.end,
          children: <Widget>[
            Padding(
              padding: EdgeInsets.symmetric(vertical: 5.0),
              child: FloatingActionButton(
                heroTag: 0,
                child: Icon(Icons.add),
                onPressed: () {
                  counterBloc.add(CounterEvent.increment);
                },
              ),
            ),
            Padding(
              padding: EdgeInsets.symmetric(vertical: 5.0),
              child: FloatingActionButton(
                heroTag: 1,
                child: Icon(Icons.remove),
                onPressed: () {
                  counterBloc.add(CounterEvent.decrement);
                },
              ),
            ),
          ],
        ),
      );
    }
  }
```

The `HomePage` is identical to the above example. When the user taps the `ElevatedButton`, we push a new named route to navigate to the `/counter` route we defined above.

CounterPage

```dart
class CounterPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Counter'),
      ),
      body: BlocBuilder<CounterBloc, int>(
        builder: (context, count) {
          return Center(
            child: Text('$count'),
          );
        },
      ),
    );
  }
}
```

`CounterPage` is a super simple `StatelessWidget` which uses `BlocBuilder` to re-render a `Text` widget with the current count. Just like before, we are able to use `BlocProvider.of<CounterBloc>(context)` in order to access the `CounterBloc`.

That's all there is to this example and the full source can be found here.

Last, we'll look at how to make a bloc globally available to the widget tree.

Global Access

> In this last example, we're going to demonstrate how to make a bloc instance available to the entire widget tree. This is useful for specific cases like an `AuthenticationBloc` or `ThemeBloc` because that state applies to all parts of the application.

Bloc

As usual, we're going to use the `CounterBloc` as our example for simplicity.

```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 currentState - 1;
        break;
      case CounterEvent.increment:
        yield currentState + 1;
        break;
    }
  }
}
```

UI

We're going to follow the same application structure as in the "Local Access" example. As a result, we're going to have three parts to our UI:

- App: the root application widget which manages the global instance of our `CounterBloc`.
- CounterPage: the container widget which exposes `FloatingActionButtons` to `increment` and `decrement` the counter.
- CounterText: a text widget which is responsible for displaying the current `count`.

App

```dart
import 'package:flutter/material.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

void main() => runApp(App());
```

```
class App extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (BuildContext context) => CounterBloc(),
      child: MaterialApp(
        title: 'Flutter Demo',
        home: CounterPage(),
      ),
    );
  }
}
```

Much like in the local access example above, the `App` manages creating, closing, and providing the `CounterBloc` to the subtree using `BlocProvider`. The main difference is in this case, `MaterialApp` is a child of `BlocProvider`.

Wrapping the entire `MaterialApp` in a `BlocProvider` is the key to making our `CounterBloc` instance globally accessible. Now we can access our `CounterBloc` from anywhere in our application where we have a `BuildContext` using `BlocProvider.of<CounterBloc>(context);`

> **Note:** This approach still works if you're using a `CupertinoApp` or `WidgetsApp`.

CounterPage

```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: Center(
        child: CounterText(),
      ),
      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);
                },
              ),
            ),
          ],
        ),
      );
    }
  }
```

Our `CounterPage` is a `StatelessWidget` because it doesn't need to manage any of its own state. Just as we mentioned above, it uses `BlocProvider.of<CounterBloc>(context)` to access the global instance of the `CounterBloc`.

CounterText

dart

```
class CounterText extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return BlocBuilder<CounterBloc, int>(
      builder: (context, count) {
        return Text('$count');
      },
    );
  }
}
```

Nothing new here; the `CounterText` widget is the same as in the first example. It's just a `StatelessWidget` which uses a `BlocBuilder` to re-render when the `CounterBloc` state changes and accesses the global `CounterBloc` instance using `BlocProvider.of<CounterBloc>(context)`.

That's all there is to it! The full source can be found here.

< PREVIOUS
Navigation