Self teaching myself Flutter, part 3: State management

In the previous section, I discussed fundamental dart and flutter concepts, as well as the importance of flutter's extensive tooling. This section will cover state management using the BLoC library.

State management libraries

As an application grows in complexity, simply using Stateful widget for all of the application's state management can become an issue, multiple problems arise such as an increased risk of code repetition, as well as an unclear separation of concerns.

Many state management libraries attempt to solve this problem, each in their own unique way:

- A developer experienced with ReactJS development may choose to use Redux, which also exists for dart ;
- Someone looking for quick iteration may start with simpler, more lightweight libraries such as Riverpod or GetX.

The library we will be looking into in this article is the BLoC library.

BLoC

BLoC stands for Business Logic Component, its main aim is to provide a state management solution that completely separates Business logic from Presentation logic. The classic three-tiered model is obtained once we integrate providers in BLoC components for external data fetching.

Figuring out BLoC, even using the official documentation, turned out to be a difficult task that spanned a few days, as I needed to grasp the fundamentals of both this library, Dart, and the Flutter framework in general.

Bloc VS Cubit

Blocs and Cubits are the two state management entities provided by the library.

A Bloc is more verbose and communicates with the presentation layer via events, implementing a Bloc is more verbose as every event has to be manually implemented.

A Cubit is simpler to understand and set up, it directly communicates with the presentation layer by exposing its methods.

A Cubit is better suited to rapid development cycles, while a Block is preferable if a high level of observability is required (as triggered events have more informations on what triggered them and how).

In this article (and in my application), I will mainly focus on Cubits.

Fundamental workings of the BLoC library

Here's how a Bloc/Cubit is implemented in a flutter application:

1. We choose the flutter Widget(s) for which we want to create the new Cubit;
2. We create the Cubit's state. Since data classes don't exist in Dart, we need to manually implement:
    1. Partial state updates using a `CopyWith` method;
    2. Deep object comparisons by extending `Equatable`;
    3. Without these two implementations, the component will not be able to detect or apply its state changes;
3. We create the Cubit, it contains the Flutter component's state, as well as the functions the component can call to alter this state;
    1. To send/retrieve external data, a `repository` is injected into the Cubit. The `repository` has one or more `providers`, whose role is to make requests to a data source;
4. In the presentation layer :
    1. We Create the repository using a `RepositoryProvider`;
    2. In the child properties of the `RepositoryProvider`, we create the Cubit and inject the repository with a `BlocProvider`;
    3. All child components of the `BlocProvider` can now access the Cubit, its state, and its functions through the `context`.

About Flutter's `context`

The `context` variable that gets passed around the widget tree is a crucial element of Flutter, essential to the operation of the Blocs/Cubits.

In flutter, each widget has a context that tells it where it is in the widget tree. If a parent widget uses a `Provider` to provide a resource, then all child widgets can use their context to retrieve it. This helps to avoid code duplication, as each widget does not need to have its own bloc/provider/service instance.

In complex screens or component, it is possible that multiple nested widgets use a `builder` function, causing many "context" arguments to be defined. In which case, one must be particularly careful of always providing the correct context variable so as to avoid tedious debugging (for example, we may want to provide a `Provider` and then a `Bloc` in our widget tree, but then accidentally use the `context` of our `Provider` widget, thus not being able to access our provided `Bloc`), it is recommended to rename the "context" argument when this occurs.

BLoC widgets with the bloc-flutter library

The BLoC library gives us multiple wrapper widgets to work with in the presentation layer.

1. `BlocBuilder` retrieves the complete state of the block and rebuilds all children as soon as the state changes;
    1. Ex. Wrapping an entire form
2. `BlocSelector` retrieves a targeted property of the block state to avoid unnecessary updates;
3. `BlocListener` performs logic that has no impact on the user interface when a state change occurs.

Fundamental workings of the BLoC library: a practical example

I shall be using the Login Screen of my application to demonstrate how a Cubit is actually implemented in the application's code.

```
part of 'login_form_cubit.dart';

sealed class LoginFormState extends Equatable {
  final AutovalidateMode autovalidateMode;
  final String email;
  final String password;
  final bool obscureText;

  const LoginFormState({
    this.autovalidateMode = AutovalidateMode.disabled,
    this.email = '',
    this.password = '',
    this.obscureText = true,
  });

  LoginFormState copyWith({
    AutovalidateMode? autovalidateMode,
    String? email,
    String? password,
    bool? obscureText,
  });

  @override
  List<Object?> get props => [
        autovalidateMode,
        email,
        password,
        obscureText,
      ];
}

class LoginFormUpdate extends LoginFormState {
  const LoginFormUpdate({
    super.autovalidateMode,
    super.email,
    super.password,
    super.obscureText,
  });

  @override
  LoginFormUpdate copyWith({
    AutovalidateMode? autovalidateMode,
    String? email,
    String? password,
    bool? obscureText,
  }) {
    return LoginFormUpdate(
      autovalidateMode: autovalidateMode ?? this.autovalidateMode,
      email: email ?? this.email,
      password: password ?? this.password,
      obscureText: obscureText ?? this.obscureText,
    );
  }
}
```

The implementation of a Cubit's state is a fairly tedious and repetitive task when using this library. It goes as follows:

- We first create an abstract class to be used for typing our `Cubit` later on, this class must implement all state properties and provide constructor interfaces;
  - `copyWith` is an agreed-upon name for implementing a function that allows partial updates of the state;
  - The `Equatable` class provides the `props` getter, which is essential for doing deep object comparisons. Such comparisons are crucial as the library needs to know what constitutes a "state change" in order to update the Presentation layer;
- We then implement an "update" class, this is the class that will be instantiated whenever we need to apply state updates;
  - The `copyWith` implementation uses Dart's null safety to implement its partial updates;
- State classes are immutable, even a partial update actually means "creating a new, partially updated object.

Here is the Cubit implementation:

```
import 'package:bloc/bloc.dart';
import 'package:daily_message_app/auth/data/models/login_model.dart';
import
'package:daily_message_app/auth/data/repositories/auth_repository.dart';
import 'package:equatable/equatable.dart';
import 'package:flutter/material.dart';

part 'login_form_state.dart';

class LoginFormCubit extends Cubit<LoginFormState> {
  LoginFormCubit(this._authRepository) : super(const LoginFormUpdate());

  final AuthRepository _authRepository;

  void initForm({
    String email = '',
    String password = '',
  }) {
    emit(state.copyWith(
      email: email,
      password: password,
    ));
  }

  void resetForm() {
    emit(LoginFormUpdate());
  }

  void updateAutoValidateMode(AutovalidateMode? autovalidateMode) {
    emit(state.copyWith(autovalidateMode: autovalidateMode));
  }

  void updateEmail(String? username) {
    emit(state.copyWith(email: username));
  }

  void updatePassword(String? password) {
    emit(state.copyWith(password: password));
  }

  void toggleObscureText() {
    emit(state.copyWith(obscureText: !state.obscureText));
  }

  Future<LoginModel> submit() async {
    return _authRepository.login(state.email, state.password);
  }
}
```

The cubit uses the `emit` method whenever the state needs to be updated. In the cubit, we implement every method that will be used by the Presentation layer. Since this is a login-form, we need to update the following state properties:

- email address;
- password;
- password obscuration.

The AutoValidateMode is a helper for form validation, its value is used to decide whenever validation should be done at submit time, or at every form change.

The authRepository uses the state to call an external api for user login.

Finally, we can see how the Cubit is used in Presentation code:

```
/* OMITED: Scaffold widget */
  body: RepositoryProvider(
    create: (context) => AuthRepository(),
    child: BlocProvider(
      create: (context) =>
          LoginFormCubit(context.read<AuthRepository>())..initForm(),
/* OMITED: Form widget */
        TextFormField(
          initialValue:
              context.read<LoginFormCubit>().state.email,
          validator: validateEmail,
          keyboardType: TextInputType.emailAddress,
          onChanged:
              context.read<LoginFormCubit>().updateEmail,
          decoration: InputDecoration(
            labelText: 'email',
            border: OutlineInputBorder(),
          ),
        ),
        BlocSelector<LoginFormCubit, LoginFormState, bool>(
            bloc: context.read<LoginFormCubit>(),
            selector: (state) => state.obscureText,
            builder: (constext, obscureText) {
                return Padding(
                padding: EdgeInsets.symmetric(vertical: 4),
                child: TextFormField(
                    initialValue: context
                        .read<LoginFormCubit>()
                        .state
                        .password,
                    validator: validatePassword,
                    onChanged: context
                        .read<LoginFormCubit>()
                        .updatePassword,
                    decoration: InputDecoration(
                    labelText: 'mot de passe',
                    border: OutlineInputBorder(),
                    suffixIcon: IconButton(
                        onPressed: context
                            .read<LoginFormCubit>()
                            .toggleObscureText,
                        icon: const Icon(Icons.remove_red_eye)),
                    ),
                    obscureText: obscureText,
                    enableSuggestions: false,
                    autocorrect: false,
                ),
                );
            },
        ),
        FilledButtonWithLoader(
            onPressed: () {
            if (_formKey.currentState!.validate()) {
                return context
                    .read<LoginFormCubit>()
                    .submit()
                    .then((value) {
                context
                    .read<GlobalCubit>()
                    .updateLoggedUser(value.user);
                Navigator.pop(context);
                }).catchError((error) {
                handleErrorSnackbar(
                    context,
                    error,
                    unauthorizedText:
                        'Nom d\'utilisateur ou mot de passe invalide',
                );
                });
            } else {
                context
                    .read<LoginFormCubit>()
                    .updateAutoValidateMode(
                        AutovalidateMode.always);
            }
            return Future(() => null);
            },
            child: Text('Connexion'),
        ),
    ),
),
```

We need to provide the repository that the `LoginFormCubit` will use, and then provide the Cubit itself. During creation, the form is initialized with default values. Once the Cubit is available, we can use the context to retrieve it and its state, allowing us to define form fields with update events that are bound to the Cubit code itself.

`context.read<CubitType>()` is heavily used to retrieve a Cubit and its methods. In this situation, state is also retrieved with this method, because the `TextFormField` widget manages visual updates by itself, and thus does not need to be rebuilt to reflect changes. However, in most other scenarios, we usually retrieve state using a `BlocBuilder` or a `BlocSelector`, such is the case for password obscuration in the password field.

When not to use BLoC

BLoC is not a "replacement" to stateful widgets, as shown in the earlier sections, state definition can be a very heavy process as code needs to be added in seven different parts of the class when we need to implement a new property. Because of this, it is still wise to keep using `StatefulWidgets` for features that don't impact the business logic.

```
class _CalendarDetailsScreenState extends State<CalendarDetailsScreen> {
  bool isLoading = false;

  void fetchCalendarDetails(BuildContext context) {
    setState(() {
      isLoading = true;
    });
    context
        .read<CalendarDetailsCubit>()
        .fetchCalendarDetails(widget.calendarCode)
        .then((value) {
      setState(() {
        isLoading = false;
      });
    });
  }
  /* OMITED : Widget code */
}
```

In this widget, the Cubit is used to trigger a data fetch, but the loading state associated to the fetch is a StatefulWidget property.

Conclusion

In this article, I talked about state management libraries, and described the BLoC library in more details. I presented the fundamentals of how Blocs and Cubits work, and then used code examples to detail how exactly I implemented it in my own application.

The next article will be about how to implement routing to navigate between screens and carry data across them.