BLoC Study

> referenced by https://bloclibrary.dev/#/

Introduction

개요

> Bloc은 몇몇의 pub packages로 구성된다

- bloc - bloc의 코어 library
- flutter_bloc - bloc을 사용하기 위해 구축된 powerful flutter widget: (블럭이 미리 구현되어 있는 패키지)

설치

> pubspec.yaml에 추가

```
dependencies:
  bloc: ^2.0.0
  flutter_bloc: ^2.0.0
```

Import

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

Why Bloc?

> Bloc makes it easy to separate presentation(UI) from business logic, making your code *fast*, *easy to test*, and *reusable*.

- 왜 Bloc패턴을 사용해야 하는 것인가?

- 그전에 Bloc은 무엇인가?
  BLoC은 **Business Logic Component**의 줄임말로 **Business logic**과 **UI**의 분리가 특징인 **Flutter**의 **state**를 관리하는 디자인 패턴 중 하나이다.

- **state** 관리! **state**란 사용자의 행동에 따라 **변하는 데이터**를 말한다. **state**관리는 왜 필요한 것인가?
  - 앱을 개발 할 때 간단한 state 변화는 setState를 통해 해결할 수 있다. 앱을 개발할 때 한 페이지 안에 state가 나눠져있는데 setState()를 호출하면 페이지 모든 data가 rebuild된다.

  - 복잡한 계층구조를 가지고 있는 앱을 만들었을 때 멀리 떨어진 위젯끼리의 통신이 잦아지면 코드가 점점 복잡해질 가능성이 크다.

> BLoC은 Business logic과 UI의 분리가 가능하게 한다.(협업 면에서도 좋음) 정말 간단하게 말하면 UI별로 StreamBuilder를 적용한 것과 비슷하다고 생각하면 된다. 그래서 위와 같은상황이 올 때 setState 없이, 깔끔하게 UI별로 state관리를 할 수 있다. 그 한 곳에서 따로 해주기 때문에 코드가 꼬일 가능성이 적어지게 된다.
> 결론적으로 더 쉽게 빠르고 효율적으로 반응하는 앱을 만들 수 있게 된다.

Core Concepts

> 그럼 이제 Bloc 사용 방법을 이해하는데 중요한 몇 가지 핵심 개념을 알아보도록 하자.

Bloc Core Concepts

1. Events

- Bloc의 input. 일반적으로 사용자의 버튼을 누르거나 페이지를 로드하거나 하는 상호작용의 응답으로 추가된다.

> 앱을 만들때는 사용자와 상호작용하는 방법을 정의해야 한다. 예를들어 counter app을 만든다고 하면, count를 올리거나 내리는 버튼을 사용자가 눌렀을 때 다음과 같이 증가와 감소 정보를 알려주는 이벤트를 정의해야 한다.

```
enum CounterEvent { increment, decrement }
```

2. States

- Bloc의 output이며 앱 state의 일부분을 나타낸다. 위에서 말했듯이 사용자의 상호작용에 따라 변하는 data들이다.

> counter app에서 state는 카운터의 현재 값을 대표하는 정수이다.

3. Transitions

- state의 변화를 뜻한다. transition은 current state, event, next state로 구성되어 있다.

> 사용자가 count 증가 버튼을 누르면 다음과 같은 transition을 볼 수 있다.

```
{
  "currentState": 0,
  "event": "CounterEvent.increment",
  "nextState": 1
}
```

4. Stream

- 비동기 data의 연속. 비동기 data가 물이라면 stream은 파이프다. 비동기인 data가 흘러올 때 마다 처리해 줄 수 있다.

> async* 함수를 이용해 Stream을 만들 수 있다

```
Stream<int> countStream(int max) async* {
    for (int i = 0; i < max; i++) {
        yield i;
    }
}
```

> yield : async* 함수에서 사용할수 있다. 열려있는 return이라고 생각하면 된다. 데이터가 있을때마다 계속 리턴할 수 있다.

```
Future<int> sumStream(Stream<int> stream) async {
    int sum = 0;
    await for (int value in stream) {
        sum += value;
    }
    return sum;
}
```

> await : async함수에서 사용할수 있다. stream에서의 각 값을 기다리고 값이 더 이상 없다면 리턴해준다.

```
void main() async {
    /// Initialize a stream of integers 0-9
    Stream<int> stream = countStream(10);
    /// Compute the sum of the stream of integers
    int sum = await sumStream(stream);
    /// Print the sum
    print(sum); // 45
}
```

> 두 함수를 동시에 사용하면 위 코드처럼 사용할 수 있다.

5. Blocs

- Bloc은 들어오는 Stream을 나가는 Stream으로 바꿔주는 구성요소이다.

```
import 'package:bloc/bloc.dart';

class CounterBloc extends Bloc<CounterEvent, int> {

}
```

> 모든 Bloc은 core bloc package를 extend해 주어야 한다.
>
> CounterBloc은 CounterEvent를 int(state)로 변환하는 역할을 한다.

```
@override
int get initialState => 0;
```

> 모든 Bloc은 event를 받기 전 초기화를 해 주어야 한다.

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

> mapEventToState 함수가 꼭 있어야 한다.
>
> => 들어오는 event를 새로운 state의 Stream으로 반환한다.

완전한 Counter Bloc

```dart
import 'package:bloc/bloc.dart';

enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  @override
  int get initialState => 0;

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

```dart
void main() {
    CounterBloc bloc = CounterBloc();

    for (int i = 0; i < 3; i++) {
        bloc.add(CounterEvent.increment);
    }
}
```

> Add 함수가 꼭 있어야 한다 => event가 일어날 때 mapEventToState 함수를 호출하기 위함이다.
>
> FIFO(먼저들어간것 먼저처리)형식으로 데이터를 처리한다.

```
@override
void onTransition(Transition<CounterEvent, int> transition) {
    print(transition);
}
```

> Bloc state가 업데이트 되기 전에 호출된다.(bloc의 로깅과 분석)

```
@override
void onError(Object error, StackTrace stackTrace) {
  print('$error, $stackTrace');
}
```

> error handling(Bloc의 기능은 영향받지 않는다)

6. BlocDelegate

- 모든 state변화(transition)와 error, 그리고 event 처리를 한 곳에서 처리할 수 있다. 개발자 로그를 남기거나 분석을 할 때 매우 유용하다.

```
class SimpleBlocDelegate extends BlocDelegate {
  @override
  void onEvent(Bloc bloc, Object event) {
    super.onEvent(bloc, event);
    print(event);
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print(transition);
  }

  @override
  void onError(Bloc bloc, Object error, StackTrace stacktrace) {
    super.onError(bloc, error, stacktrace);
    print('$error, $stacktrace');
  }
}
```

> BlocDelegate를 extend하고 onTransition을 override 하기만 하면 된다.

```
void main() {
  BlocSupervisor.delegate = SimpleBlocDelegate();
  CounterBloc bloc = CounterBloc();

  for (int i = 0; i < 3; i++) {
    bloc.add(CounterEvent.increment);
  }
}
```

> Bloc에서 사용하기 위해서 main함수에 추가해준다.
>
> BlocSupervisor는 BlocDelegate에 대한 모든 책임을 감독하는 싱글톤이다.

Flutter Bloc Core Concepts

> bloc을 사용하기 위해 구축된 powerful flutter widget

1. BlocBuilder

- Bloc과 builder의 기능을 제공하는 위젯이다.

> 새로운 state에 반응하여 build를 처리해준다.
>
> Streambuilder와 매우 유사하다.

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

> [bloc: blocA] : parent BlocProvider와 curret BuildContext에 접근할 수 없는 single widget으로 사용하고 싶을 경우 지정해준다.
>
> Bloc이 여러개 있는 경우 어떤 bloc을 쓸 것인지 지정하는 할을 한다.

```
BlocBuilder<BlocA, BlocAState>(
  condition: (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
  }
)
```

> condition을 이용하면 더 세부적인 제어를 할 수 있다.
>
> 이전 bloc state와 현재 bloc state를 사용해 boolean을 리턴해서 true면 rebuild해준다.

2. BlocProvider

- child에 bloc을 제공해주는 위젯이다.

> bloc의 단일 인스턴스가 서브 트리 내 여러 위젯에 제공될 수 있도록 Dependency Injection 위젯으로 사용된다.

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

> 대부분의 경우에는 subtree에서 사용할 수 있는 새로운 bloc을 만들어야 한다. 그래서 이 경우에 creating bloc에 반응하여 자동으로 bloc을 closing 처리한다.

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

> 경우에 따라 widget tree의 새로운 부분에 기존 bloc을 제공할 수도 있다. 이는 기존 bloc을 새로운 route로 이용할 필요가 있을 때 가장 일반적으로 사용된다. 이 경우에 BlocProvider에서는 bloc을 만들지 않았기 때문에 closing 처리하지 않는다.

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

> 위의 ChildA()와 ScreenA()에서 blocA를 위와같이 검색할 수 있다.

3. MultiBlocProvider

- 여러 blocProvider를 하나로 결합한 위젯이다.

> 가독성을 향상시키고 blocProvider를 중첩해서 선언할 필요성을 제거한다.

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

=>

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

---

4. BlocListener

- 해당 Bloc의 state가 변경되었을 때 호출되는 위젯이다.

> navigation, snackbar 표시, dialog 표시 등 state 변경당 한 번 발생해야 하는 기능에 사용해야 한다.
>
> (build하는 것이 아닌 다른 것들)

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

```
BlocListener<BlocA, BlocAState>(
  condition: (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(),
)
```

> BlocListener도 마찬가지로 condition에 따른 세부 조정이 가능하다.

5. MuiltiBlocListener

- 여러 BlocListener 위젯을 하나로 결합한 위젯이다.

> 가독성을 향상시키고 BlocListener를 중첩해서 선언할 필요성을 제거한다.

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

=>

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

---

6. RepositoryProvider

- child에게 repository를 제공하는 위젯이다.

> BlocProvider는 bloc을 제공하는데 사용되어야 하고, RepositoryProvider는 repository에만 사용해야 한다.

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

```
RepositoryProvider.of<RepositoryA>(context)
```

> 위의 방식으로 repository instance를 검색할 수 있다.

7. MultiRepositoryProvider

- 여러 RepositoryProvider위젯을 하나로 결합한 위젯이다.

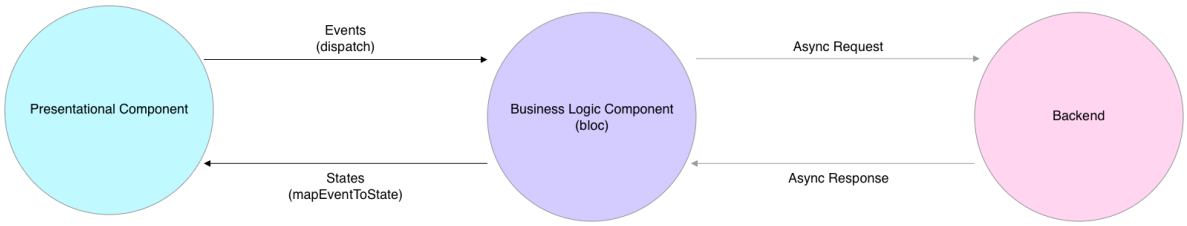
> 가독성을 향상시키고 RepositoryProvider를 중첩해서 선언할 필요성을 제거한다.

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

=>

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

Architecture- bloc의 작동방식



Bloc을 사용함으로써 애플리케이션을 세 가지 layer로 분리시킬 수 있다.

- Backend
  - 일반적으로 알고있는 서버
  - 클라이언트에서 요청하면 결과를 전송해주는 역할을 수행
- Business Logic Component(bloc)
  - UI와 Backend 사이의 중간다리 역할을 수행
- Presentational Component
  - 사용자에게 보여지는 부분(UI)

- Flutter widget들로 구성

작동방식

1. UI 부분에서 event가 발생 -> bloc으로 dispatch
2. event를 전달받은 bloc이 Backend에 data 요청
3. Backend에서 요청한 데이터를 bloc으로 전송
4. bloc에서 전달받은 데이터를 필요한 형태로 가공하여 state에 반영
5. update된 state를 화면에 보여짐

- 일반적으로는 위와 같은 과정을 따라 상호작용하지만, 경우에 따라서는 bloc과 Backend의 상호작용 없이 state를 update 하는 경우도 있다.

Naming Conventions(권장사항)

Event Conventions

- 이벤트는 bloc의 관점에서 이미 발생한 것이므로 과거시제로 이름을 지정한다

> BlocSubject + Noun(optional) + Verb (event)
>
> Initial load event에서는 BlocSubject + Started

- Ex)

✅ **Good**

`CounterStarted` `CounterIncremented` `CounterDecremented` `CounterIncrementRetried`

❌ **Bad**

`Initial` `CounterInitialized` `Increment` `DoIncrement` `IncrementCounter`

State Conventions

- State는 특정 시점의 snapshot이므로 명사로 이름을 지정한다.

> BlocSubject + Verb (action) + State

- Ex)

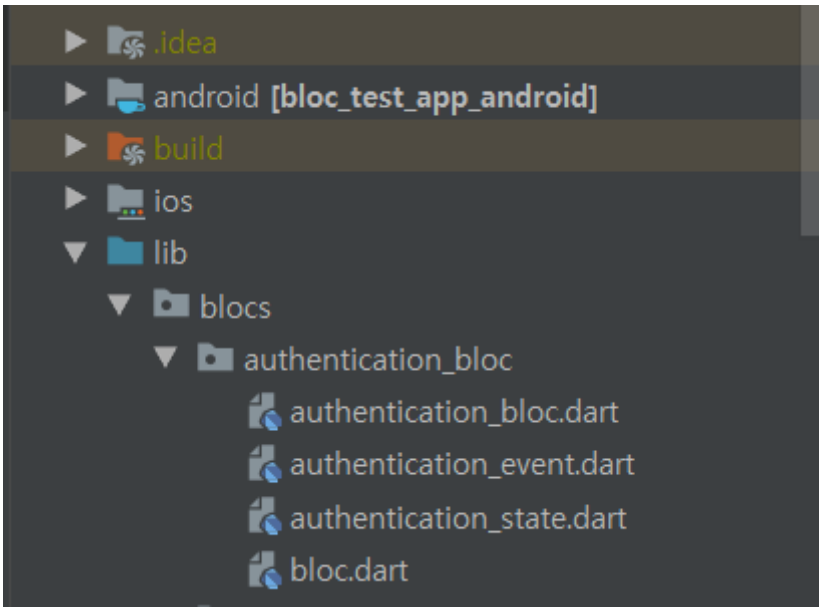
✅ Good

`CounterInitial` `CounterLoadInProgress` `CounterLoadSuccess` `CounterLoadFailure`

❌ Bad

`Initial` `Loading` `Success` `Succeeded` `Loaded` `Failure` `Failed`

프로젝트 내 Bloc의 구성



> 기본적으로 기존 프로젝트 구조에서 바뀌는 것은 없다.
>
> 각 bloc들을 저장할 blocs 폴더를 만들고, 그 안에 bloc별로 폴더를 만들어 저장한다.
>
> 각 bloc들의 폴더 안에는 event를 정의하는 event 파일, state를 정의하는 state 파일, event에 따라 데이터를 처리하고, state를 업데이트하는 bloc파일과 이 세 파일을 서 export하는 역할을 하는 bloc.dart파일이 있다.

```
export 'authentication_bloc.dart';
export 'authentication_event.dart';
export 'authentication_state.dart';
```

> bloc.dart는 export만 하기 때문에 꼭 필요한 파일은 아니지만 세 파일을 한번에 import 할 수 있는 편의성을 위해 작성한다.

*** Recipe와 튜토리얼들은 사이트에 잘 나와있으니 참고하도록 하고 recipe의 중요 포인트만 짚고 넘어가도록 하겠다.**

Firebase Login Tutorial Tip

Plugin을 바로 추가하고 빌드를 하면 AndroidX 문제 때문에 빌드가 되지 않을때가 있다.

그럴때

1.

android/app/build.gradle 파일에 들어가서

defaultConfig 안에 multiDexEnabled true를 넣어준다.

2.

android/gradle.properties 파일에 들어가서

```
android.enableJetifier=true
android.useAndroidX=true
```

를 추가해준다!

Several Recipes

- Show Snackbar
- Navigation
- Bloc Access

Recipes: Show SnackBar with BlocListener

- 이 recipe에서는 BlocListener bloc에서 state 변경이 있을 때 어떻게 BlocListener를 사용해 snackbar를 보여주는지에 대해 알려준다.

> BlocBuiler와 BlocListener의 차이를 명확하게 알 수 있는 recipe이다.
>
> 버튼을 눌러 state 변화를 주어 snackbar를 띄우는 기본적인 예제이다.

> BlocBuilder는 state 변화에 반응하여 “Widget”을 렌더링 하기 위해서 사용한다.
>
> BlocListener는 state 변화에 반응하여 “어떠한 것”을 하기 위해서 사용한다.(위젯말고 다른게 필요할때)

Recipes: Navigation

- 이 recipe에서는 Blocbuilder와 BlocListener를 사용해 어떻게 navagation하는지 알려준다.
- 두 가지 접근 방법이 있다 : Direct Navigation, Route Navigation

Direct Navigation

> BlocBuilder를 사용하는 방법이다.
>
> state에 변경이 있으면 BlocBuilder에서 state에 맞는 페이지를 렌더링한다.

Route Navigation

> BlocListener를 사용하는 방법이다.
>
> state에 변경이 있으면 Navigator.of(context).pushNamed('/pageB'); Navigator로 새 경로를 push한다.

> 페이지를 렌더링하는것과 경로를 push하는 것의 차이:
>
> 페이지가 navigator stack에 쌓이냐 쌓이지 않느냐의 차이이다.

Recipes: Bloc Access

- 이 recipe에서는 위젯 트리 전체에서 access 가능한 bloc을 만들기위해 BlocProvider를 사용하는 방법을 알려준다.
- 세 가지의 시나리오가 있다: Local Access, Route Access, Global Access

Local Access

- 이 예제에서는 BlocProvider를 로컬 하위 트리에서 bloc을 사용할 수 있도록 하는데 사용한다.

  여기서 로컬은 pushed/poped되는 경로가 없는 context 내의 로컬을 의미한다.

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

Route Access

- 이 예제에서는 BlocProvider를 여러 경로에 걸쳐 bloc에 access 하는데 사용한다.
- 새로운 경로가 push되면 이전에 참조되었던 bloc이 더이상 없는 다른 BuildContext를 갖게된다.
- 결과적으로, 새로운 경로를 별도의 BlocProvider로 감싸줄 필요가 있다.

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

=>

```
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: RaisedButton(
          onPressed: () {
            Navigator.of(context).push(
              MaterialPageRoute<CounterPage>(
                builder: (context) {
                  return BlocProvider.value(
                    value: counterBloc,
                    child: CounterPage(),
                  );
                },
              ),
            );
          },
          child: Text('Counter'),
        ),
      ),
```

> 위의 Local Access예제와 비슷하지만, Homepage에서 counterpage로 bloc instance를 넘겨주기 위해 BlocProvider.value 로 감싸주는것을 볼 수 있다.

Global Access

- 이 예제에서는 전체 위젯 트리에서 어떻게 bloc instance를 사용가능하게 하는지를 보여준다.

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

> MaterialApp을 BlocProvider로 감싸줘서 모든 위젯트리에서 bloc에 access 가능하게 한다.

> BlocProvider.of(context);
>
> 이런 방식으로 access해주면 된다.