Flutter (01.18-19)

1. Firebase를 Flutter에 적용하는 방법

Firebase는 Flutter 애플리케이션에서 인증, 데이터베이스, 클라우드 스토리지, 호스팅 등 다양한 서비스를 쉽게 구현할 수 있도록 도와줍니다.

1.1. Firebase CLI 설치

Firebase 프로젝트를 초기화하려면 Firebase CLI가 필요합니다.

1. Node.js 설치 (Node.js 공식 사이트):

```
bash
복사편집
node -v
npm -v
```

2. Firebase CLI 설치:

```
bash
복사편집
npm install -g firebase-tools
```

3. Firebase CLI 로그인:

```
bash
복사편집
firebase login
```

1.2. Firebase 프로젝트 생성 및 초기화

1. **Firebase 콘솔에서 프로젝트 생성**:
   - Firebase 콘솔에 접속하여 프로젝트를 생성합니다.
2. **Flutter 프로젝트에서 Firebase 초기화**:

- Flutter 프로젝트 디렉토리에서 CLI를 실행:

```
bash
복사편집
firebase init
```

- Firebase 프로젝트를 선택하거나 새 프로젝트를 생성합니다.

1.3. FlutterFire 설정

1. **FlutterFire CLI 설치**:

```
bash
복사편집
dart pub global activate flutterfire_cli
```

2. **Firebase 설정 파일 생성**:

```
bash
복사편집
flutterfire configure
```

- Android: `google-services.json` 생성.
- iOS: `GoogleService-Info.plist` 생성.
- `firebase_options.dart` 파일이 자동 생성됩니다.

3. **Firebase 관련 패키지 추가**:

`pubspec.yaml`에 필요한 Firebase 패키지를 추가:

```
yaml
복사편집
dependencies:
  firebase_core: ^2.0.0
  firebase_auth: ^4.0.0
```

```
  cloud_firestore: ^4.0.0
```

4. **Firebase 초기화 코드 작성**:

   `main.dart`에서 Firebase를 초기화:

```
dart
복사편집
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );
  runApp(MyApp());
}
```

---

2. BLoC이란 무엇인가?

2.1. BLoC의 정의

- **BLoC**(Business Logic Component): Flutter에서 **비즈니스 로직과 UI를 분리**하기 위한 상태 관리 패턴입니다.
- Google이 추천하는 **reactive programming** 방식으로 동작하며, 스트림(Stream)을 사용하여 데이터 흐름을 관리합니다.

2.2. BLoC의 주요 구성 요소

1. **Event**: 사용자의 액션(클릭, 입력 등)을 나타냄.
2. **State**: UI가 표시해야 할 데이터 상태.
3. **Bloc**: Event를 받아서 State를 생성하는 핵심 비즈니스 로직.

2.3. BLoC의 동작 흐름

1. **사용자 이벤트 발생**: 버튼 클릭, 텍스트 입력 등.
2. **Bloc에서 이벤트 처리**: 이벤트를 받아서 새로운 상태를 생성.
3. **UI 업데이트**: 상태 변화에 따라 위젯 트리를 재구성.

2.4. BLoC 사용 이유

- **비즈니스 로직 분리**: UI와 로직이 독립적이므로 코드의 가독성과 유지보수성이 향상됨.
- **테스트 용이**: 비즈니스 로직을 쉽게 단위 테스트 가능.
- **일관성 있는 상태 관리**: 앱 상태를 체계적으로 관리.

---

3. 각 Flutter 위젯의 기능과 사용 이유

3.1. 주요 위젯

1. **Scaffold**:
   - 화면 구조를 정의.
   - 앱바, 하단바, 플로팅 버튼 등을 포함할 수 있음.
   - **사용 이유**: Flutter 앱의 기본 레이아웃을 설정하기 위해 사용.
2. **AppBar**:
   - 상단의 앱바를 생성.
   - 타이틀, 액션 버튼 등을 포함.
   - **사용 이유**: 앱의 제목과 기본 네비게이션 제공.
3. **Text**:
   - 텍스트 표시 위젯.
   - **사용 이유**: 간단한 텍스트 렌더링.
4. **Column**:
   - 수직 방향으로 위젯을 정렬.
   - **사용 이유**: 세로 레이아웃을 구현.
5. **Row**:
   - 수평 방향으로 위젯을 정렬.
   - **사용 이유**: 가로 레이아웃을 구현.

6. **Container**:
    - 박스를 생성하여 크기, 색상, 마진 등을 설정.
    - **사용 이유**: 간단한 스타일링과 레이아웃 구성.
7. **ListView**:
    - 스크롤 가능한 목록을 생성.
    - **사용 이유**: 스크롤 가능한 아이템 목록을 표시할 때 사용.
8. **Stack**:
    - 위젯을 겹쳐서 배치.
    - **사용 이유**: 레이어를 쌓는 구조를 구현할 때 사용.

3.2. 위젯 사용 이유

- Flutter는 UI를 위젯 기반으로 구성하며, 각 위젯은 특정 기능을 담당.
- 위젯의 조합을 통해 유연한 UI를 구현 가능.

---

4. Flutter와 Firebase 및 BLoC 결합의 장점

4.1. Firebase와 Flutter

- **빠른 개발**: 백엔드 코드를 거의 작성하지 않고 데이터베이스, 인증 등 복잡한 기능을 구현 가능.
- **실시간 동기화**: Firestore, Realtime Database를 사용하면 실시간으로 데이터 변경 사항을 반영 가능.

4.2. BLoC 사용

- **UI와 로직 분리**: 대규모 프로젝트에서 UI와 비즈니스 로직을 분리하여 코드 품질을 높임.
- **재사용성 증가**: Bloc을 다른 위젯이나 화면에서 재사용 가능.
- **테스트 가능**: 독립적인 비즈니스 로직 테스트가 가능.

정리

1. **Firebase**:
    - Flutter 앱에 Firebase 서비스를 연동하여 인증, 데이터베이스 등 기능을 구현.

2. **BLoC**:
    - 비즈니스 로직과 UI를 분리하여 유지보수성과 테스트 용이성을 높이는 상태 관리 패턴.
3. **Flutter 위젯**:
    - 앱 UI를 구성하는 기본 요소로, 각 위젯은 특정 레이아웃이나 기능을 담당.

프로젝트 상태관리 고민(?)

Riverpod vs BLoC 비교

기준	Riverpod	BLoC
학습 곡선	상대적으로 쉬움. Provider 기반이라 익숙하면 더 빠르게 적응 가능.	다소 복잡. Event-Stream-State 패턴을 학습해야 하며 Boilerplate가 많음.
코드 간결성	코드가 간결하며, 의존성 주입이 매우 직관적.	Event와 State 사이의 Boilerplate 코드가 많아질 수 있음.
상태 공유 및 전파	간단히 상태를 공유할 수 있고, 구독 및 데이터 흐름 관리가 쉽다.	명확한 Event와 State로 상태 관리 가능하지만, Streams 때문에 비동기 작업이 복잡해질 수 있음.
확장성	구조가 간단하고 대규모 프로젝트에서도 유연하게 확장 가능.	대규모 프로젝트에 적합하지만, 코드가 복잡해지고 관리가 어려워질 가능성 있음.
리액티브 프로그래밍	구독 기반으로 리액티브 프로그래밍 가능. 비동기 상태 관리에 유리.	Streams 기반으로 리액티브 프로그래밍을 제공하지만 복잡한 스트림 체인을 관리해야 함.
UI와의 결합	Provider와 유사하게 UI에 쉽게 연결 가능.	상태 변화와 UI의 분리가 명확하지만 그로 인해 코드 간결성이 낮아질 수 있음.

주요 기능 분석

1. 주변 클라이밍장 정보 제공

- **Riverpod 추천**:
    - 클라이밍장 정보를 지도에서 가져와 뿌려주는 작업은 비동기 처리(API 호출)가 빈번합니다.
    - Riverpod의 `FutureProvider` 또는 `AsyncNotifier`를 활용하면 간결하게 처리 가능.

- 상태 공유가 직관적이라 UI에 데이터를 쉽게 연결.

2. 인스타그램 연동 및 커뮤니티 기능

- **Riverpod 추천**:
  - 커뮤니티에서 게시글 필터링(최신순, 인기순) 및 좋아요, 댓글 등은 여러 상태 관리가 필요합니다.
  - Riverpod의 `StateNotifier`를 사용하면 각각의 상태를 쉽게 분리 및 관리 가능.
- BLoC은 Event와 State로 세분화해야 해서 코드가 복잡해질 수 있음.

3. 클라이밍 성장 기록 그래프

- **Riverpod 추천**:
  - 성장 데이터를 가져오고 일자별로 그래프에 연결하는 비동기 데이터 처리가 중요.
  - `StateNotifier`를 사용해 상태 변화를 그래프 컴포넌트에 직관적으로 연결 가능.

4. 실시간 매칭 및 랭킹 시스템

- **BLoC 고려 가능**:
  - 복잡한 매칭 로직, 큐 관리, 패널티 부여 등은 Event-Stream 구조가 유리할 수 있음.
  - 하지만 Riverpod도 StreamProvider를 지원하므로 충분히 구현 가능.
- **Riverpod 추천 이유**:
  - 전체적인 로직의 일관성을 위해 Riverpod으로 매칭 상태를 관리하면 더 간결.

5. 클리어 시간 측정 및 악성 유저 관리

- **Riverpod 추천**:
  - 모션 인식, 시간 측정은 클라이언트에서 빠르게 반응해야 하므로 Riverpod의 상태 관리가 적합.
  - 악성 유저 신고 및 신뢰도 관리도 단순히 상태를 업데이트하면 되므로 구현이 쉬움.

6. 영상 촬영 설정 및 추가 기능

- **Riverpod 추천**:
  - 촬영 옵션이나 볼더 색 관리 같은 UI 상태를 관리하는 데 적합.

정리

Riverpod이 **더 간결하고 유지보수하기 쉬운 코드**를 제공하며, **여러 상태를 독립적이면서도 효율적으로 관리**할 수 있습니다.

다만, 실시간 매칭과 같이 복잡한 Event-Stream 구조가 필요한 부분에서는 BLoC이 약간 유리할 수 있지만, **Riverpod의 StreamProvider**를 활용하면 충분히 대체 가능합니다.

하지만 Riverpod 에 대한 강의자료가 부족함 . Bloc 은 강의 자료가 유튜브에 좀 널려 있어서 학습하기는 좋으나 단기간에 학습이 가능할지가 의문... 신중한 선택이 필요할 것 같다.