Todo 앱
- StateManagement

7 장단점 총 정리

좋은 앱의

- 아키텍처는?
- 상태 관리 방법은?

Why?

1. 쉬운 개발
   - 구조 파악 쉬움
   - 작성하는 코드의 양
   - 좋은 가독성
2. 안정성
   - Test Code 작성 가능
3. ~~성능 (500만 연산 미만은 무시)~~
4. 확장성
   - 변경 또는 기능 추가
5. 제공되는 추가 기능들

아키텍처 & 상태관리는

맞고 틀리고의 정답은 없습니다.
차이점만 있을 뿐

어떤 것이 좋은지 알아보고, 살펴보는데
너무 많은 시간을 쏟는것도 낭비입니다.

직접 손으로 경험 해보시길
추천드립니다.

패키지 차이점 비교

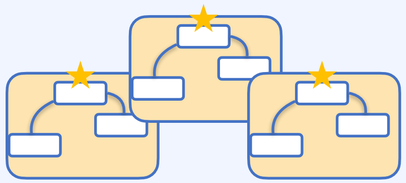
GetX

Bloc

Riverpod

State Management 구조의 종류

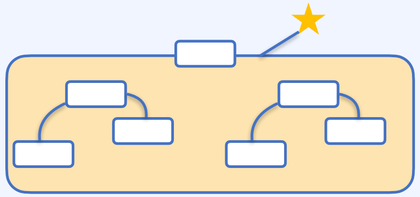
Scoped Model



Bloc (Provider기반 context.read)

Riverpod (ref.read)

Static Model



GetX (Get.find())

1. 접근성 면: GetX가 조금 코드 작성 측면에서 편하나, 구조적으로는 모두 접근에 문제 없음
2. Widget tree에 기반한 설계의 경우 GetX는 직접 구조 구현을 해야함

State 객체 보관 방법

GetX

```
class TodoDataHolder extends GetxController {
  final RxList<Todo> todoList = <Todo>[].obs;
```

1. GetxController 안에 직접 필드로 들고 있음

State 객체 보관 방법

Bloc

```
class TodoCubit extends Cubit<TodoBlocState> {
```

```
@freezed
class TodoBlocState with _$TodoBlocState{
  const factory TodoBlocState(
      BlocStatus status,
      List<Todo> todoList,
      ) = _TodoBlocState;

}
```

1. Cubit, Bloc마다 1 종류의 State객체만 관리

   => 때문에 보통 기능/화면 별 wrapping state data 가 필요하다.

2. copyWith 함수가 늘 필요함

   => 때문에 직접 구현 또는 이를 만들어주는 freezed 패키지가 필요.

State 객체 보관 방법

Riverpod

```
class TodoDataHolder extends StateNotifier<List<Todo>>
```

```
StateNotifierProvider<TodoDataHolder, List<Todo>>
```

1. Cubit, Bloc처럼 StateNotifier안에 State 객체를 한 종류만 넣을 수 있음.

2. 가이드에서는 새 state객체 사용을 권함
   copyWith 함수 사용을 권함
   state = newState (state.copyWith(바뀐필드))

   => 그러나 꼭 필수는 아님

부가 기능 - 뷰 없이 다른 State끼리 연결하기

GetX

- 별도로 구현해줘야함 하지만 그렇게 어려운 편은 아님

```
class FirstController extends GetxController {
  int id = 1;

  int method(String param){
    return 30;
  }
}
```

```
class SecondController extends GetxController {
  final firstController = Get.find<FirstController>();

  @override
  void onInit() {
    super.onInit();

    debugPrint('${firstController.id}'); // prints 1
    debugPrint('${firstController.method('Test Param')}'); // prints 1
  }
}
```

부가 기능 - 뷰 없이 다른 State끼리 연결하기

Bloc

- 구조적으로 가능하지만 공식 가이드에서 이렇게 사용하면 안좋다고 가이드.
  (tight 커플링)

```dart
class BadBloc extends Bloc {
  final OtherBloc otherBloc;
  late final StreamSubscription otherBlocSubscription;

  BadBloc(this.otherBloc) {
    // No matter how much you are tempted to do this, you should not do this
    // Keep reading for better alternatives!
    otherBlocSubscription = otherBloc.stream.listen((state) {
      add(MyEvent());
    });
  }

  @override
  Future<void> close() {
    otherBlocSubscription.cancel();
    return super.close();
  }
```

부가 기능 - 뷰 없이 다른 State끼리 연결하기

Bloc

- 가이드: 이렇게 각 Bloc의 state가 변할때 이벤트를 발생시켜서 전달하길 권함

```
Widget build(BuildContext context) {
  return BlocListener<FirstBloc, FirstState>(
    listener: (context, state) {
      BlocProvider.of<SecondBloc>(context).add(SecondEvent());
    },
    child: TextButton(
      child: const Text('Hello'),
      onPressed: () {
        BlocProvider.of<FirstBloc>(context).add(FirstEvent());
      },
    ),  // TextButton
  );  // BlocListener
}
```

이 구조가 불편한점:

1. build는 위젯 UI 그리는 공간인데 로직이 들어감

2. 위젯이 사라지면, 리스닝 로직도 사라짐 (최상위에 선언해야함)

3. 상태 변화 및 함수 호출을 위한 이벤트를 늘 새로 생성해서 만들어야함

부가 기능 - 뷰 없이 다른 State끼리 연결하기

Riverpod

- 가장 직관적인 방법으로 서로 참조 및 관찰까지 가능.

```
final weatherProvider = FutureProvider((ref) async {
  final city = ref.watch(cityProvider);
  // We can then use the result to do something based on the value of `cityProvider`.
  return fetchWeather(city: city);
});
```

쉽다 → 어렵다

기능이 별로 없다.
(상태 관리로서)

기능이 많다.
제약이 비교적 많다.

GetX

RiverPod

+ State간에 참조가 직관적
+ 별도의 watch를 달아서
각각의 state 관찰/통제 가능

BloC

+ 이벤트 중앙 관찰/통제/변환 가능