`Riverpod`와 `Get_it` 을 사용하여 의존성 주입 및 상태 관리를 합니다.

Get_it

기본적으로 get_it으로 의존성이 주입되는 클래스들은 runApp이 실행되기 전 main.dart 파일에 주로 위치합니다.

`AuthService`, `ApiService` 등이 이에 해당합니다.

해당 클래스들은 다음과 같이 접근 가능합니다.

```
final authService = AuthService.I;
authService.logIn();
...
// 클래스에 .I를 붙여 접근 가능합니다.
```

이 중 `AuthService`의 경우, login여부, token 여부 등 상태를 가지기 때문에 `AuthState` 라는 상태 클래스를 따로 두어 riverpod provider 패턴과 함께 사용하고 있습니다.

Riverpod

기본적으로 `NotifierProvider`를 사용합니다.

state가 하나가 아니라 여러개일 경우, `freezed`를 이용해서 state를 정의한 후, 해당 state를 사용하는 Notifier를 같은 파일 내, 위에 위치하게 작성합니다. NotifierProvider또한 같은 파일 내에 작성한 후 최상위에 위치하게 합니다.

코드의 간결함을 위해 Provider 선언 시 Notifier는 생략하고 변수명을 짓습니다.

예시 코드는 다음과 같습니다.

```
// 1. Provider 선언
final influencerStateProvider =
NotifierProvider<InfluencerNotifier, InfluencerState>(
  InfluencerNotifier.new,
)

// 2. Notifier 클래스 정의
```

```dart
class InfluencerNotifier extends Notifier<InfluencerState> {
  @override
  InfluencerState build() {
    return const InfluencerState();
  }

  Future<void> fetchInfluencers() async {
    state = state.copyWith(isLoading: true, errorMessage: null);

    // api를 이용하는 방법에 대한 문서는 추가로 작성 예정입니다.
    final result = await ApiService.I.fetchInfluencers();

    if(result.isSuccess){
      state = state.copyWith(
        influencers:
result.data.map((e)=>InfluencerState.fromJson(e)),
        isLoading: false,
      );
    }else{
      state = state.copyWith(
        isLoading: false,
        errorMessage: '인플루언서 목록을 불러오는데 실패했습니다: $e',
      );
    }
  }

  // 비지니스 로직을 이 위치에 위치시킵니다.
}

// 3. 상태 클래스 정의

@freezed
class InfluencerState with _$InfluencerState {
  const factory InfluencerState({
    @Default([]) List<Influencer> influencers,
    @Default(false) bool isLoading,
    String? errorMessage,
  }) = _InfluencerState;
}
```

UI 적용

적용 예시

```
class InfluencerListScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final influencerState =
ref.watch(influencerNotifierProvider);

    ref.listen(influencerNotifierProvider, (previous, next) {
      if (next.errorMessage != null) {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(content: Text(next.errorMessage!)),
        );
      }
    });

    return Scaffold(
      appBar: AppBar(title: const Text('인플루언서 목록')),
      body: influencerState.isLoading
          ? const Center(child: CircularProgressIndicator())
          : ListView.builder(
              itemCount: influencerState.influencers.length,
              itemBuilder: (context, index) {
                final influencer =
influencerState.influencers[index];
                return ListTile(
                  title: Text(influencer.name),
                  subtitle: Text('팔로워:
${influencer.followerCount}'),
                );
              },
            ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {

 ref.read(influencerNotifierProvider.notifier).fetchInfluencers()
;
        },
```

```
          child: const Icon(Icons.refresh),
        ),
      );
    }
  }
```

- **ref.watch vs ref.read**: `watch`는 값이 변경될 때마다 위젯/Provider를 재빌드합니다. `read`는 일회성 읽기에 사용합니다. 이벤트 핸들러나 메서드 내에서는 `read`를 사용하세요.
- **select 활용**: 큰 객체에서 일부만 관찰하려면 `select`를 사용해 불필요한 재빌드를 방지합니다:

```
// 전체 상태가 아닌 isLoading만 관찰
final isLoading = ref.watch(
  influencerNotifierProvider.select((state) =>
state.isLoading)
);
```

- **family vs autoDispose**: 매개변수가 필요한 Provider는 `family`를, 사용하지 않을 때 메모리를 해제하려면 `autoDispose`를 사용합니다. 두 기능은 함께 사용할 수 있습니다: `Provider.autoDispose.family`.
- **ConsumerWidget vs Consumer**: 위젯 전체가 Riverpod에 의존한다면 `ConsumerWidget`을, 위젯의 일부만 의존한다면 `Consumer`를 사용하세요.