api 핸들링은 총 3가지 layer를 거쳐서 합니다.

UI 레이어 ↔ 로직 레이어 ↔ ApiService

또한, `Result` 객체와 Rust의 `Some, Error` 개념을 이용해서 에러를 핸들링합니다.

try{}catch{}를 사용하지 않아도 됩니다. 서버에서 발생하는 에러는 interceptor를 통해 handling되어 `Result` 객체로 전달됩니다.

UI 레이어

UI 레이어는 위젯 내부의 코드를 의미합니다.

BuildContext를 직접적으로 다루는 코드입니다.

절대로 비지니스 로직 레이어에 BuildContext를 넘기면 안됩니다.

> 엥? 그러면 dialog를 어떻게 띄우나요?

로직 레이어로부터 응답 결과를 받아서 해당 결과에 따라 dialog를 띄웁니다.

Routing도 해당 레이어에서만 진행합니다.

예시 코드

```
final res = await AuthService.I.setInquire( // res 는 bool
  title,
  text,
  phoneNum.formatPhoneNumber(),
);
if (res) {
  showCustomSuccessDialog(
    context: context,
    title: "1:1 문의 등록 완료!",
    content: "고객님의 문의내역이\\n 정상적으로 접수되었습니다.",
    buttonText: "마이페이지로 돌아가기기",
    onPressed: () {
```

```
        Navigator.pop(context); // 다이얼로그 닫기
        Navigator.pop(context); // 현재 화면 닫기
      },
    );
  }
```

로직 레이어

로직 레이어는 특정 screen에서 다루는 로직(버튼 클릭이나 service에 해당하지 않는 것 들)에 대한 코드입니다.

대체로 riverpod에 의하여 provider 패턴으로 다루게 됩니다.

절대로 buildcontext를 다루지 않습니다.

ApiService 레이어로부터 온 `Result` 응답에 맞추어 성공적이지 않았을때/성공적일때 데이터를 처리하던지 저장하던지 하는 등의 모든 로직을 작성합니다.

절대로 서버에 직접적으로 요청을 하지 않습니다.

기본적으로 `Result` 응답은 홀로 쓰이지 않기 때문에, 해당 응답을 파싱해서 `fromJson` 을 이용한다던가 하는 역할이 위치하게 됩니다.

예시코드

```
Future<bool> deleteAccount(List<bool> selectedList) async {
  List<String> selectedReasons = deleteReasons
      .asMap()
      .entries
      .where((entry) => selectedList[entry.key])
      .map((entry) => entry.value)
      .toList();
  Map<String, dynamic> requestBody = {"reason":
selectedReasons.join(", ")};

  final result = await api.deleteAccount(requestBody);
  final isSucceed = result.isSuccess && (result.data["isSuccess"]
== true);
  secureStorageService.deleteAll();
```

```
  return isSucceed;
}
```

ApiService 레이어

해당 레이어는 기본적으로 서버와 통신하는 모든 api 코드를 모아놓는 레이어입니다.

고로, api 수정이 일어날 경우 해당 레이어와, 해당 api를 사용하는 로직레이어만 수정 해 주면 됩니다.

로직 레이어가 직접적으로 서버와 통신하지 않기 때문에, 해당 코드와 연관 있는 코드를 손 쉽게 파악할 수 있습니다. 모든 서버로부터의 결과 값은 `Future<Result>` 형태여야 합니다. 여타 formatting이라던지 data를 만드는 과정은 로직 레이어에게 맡겨, 최대한 간결한 코드를 작성할 수 있게 해야 합니다.

예시코드

```
  Future<Result<bool>> deleteAccount(dynamic body) =>
      _authDio.patch('/user/leave', data: body);

  Future<Result<User>> getUserByEmail(String email) =>
      _noAuthDio.get('/user/email/$email', fromJson:
User.fromJson);
```

`Result` 객체

모든 api는 해당 객체로 wrapping 됩니다.

제네릭으로 구현되어 Result<성공시 데이터타입, 에러>로 구현되어있는데,

ApiService 레이어에서 리턴합니다.

해당 레이어에서 리턴 해 줄때 type을 명시해줍니다. json으로 오는 결과의 경우 fromJson으로 매핑해줍니다.

`isSuccess` 필드를 이용해서 성공 여부를 판별해서 success data를 가져오는 방식 도 있겠지만,

`fold` 함수를 이용하는것을 추천합니다.

```
//위의 예시 코드 변경
final result = await api.deleteAccount(requestBody);
final isSucceed = result.isSuccess && (result.data["isSuccess"]
== true);
secureStorageService.deleteAll();
return isSucceed;

-->

final result = await api.deleteAccount(requestBody);
return result.fold(
  onSuccess: (data) {
    secureStorageService.deleteAll();
    return data["isSuccess"];
  },
  onFailure: (e) {
    return false;
  },
);
```

이외에도 `map, asyncMap` 의 함수로 fromJson을 보다 쉽게 쓴다든지 할 수 있습니다.

이 객체로 try~catch 의 반복적인 코드를 줄일 수 있습니다.