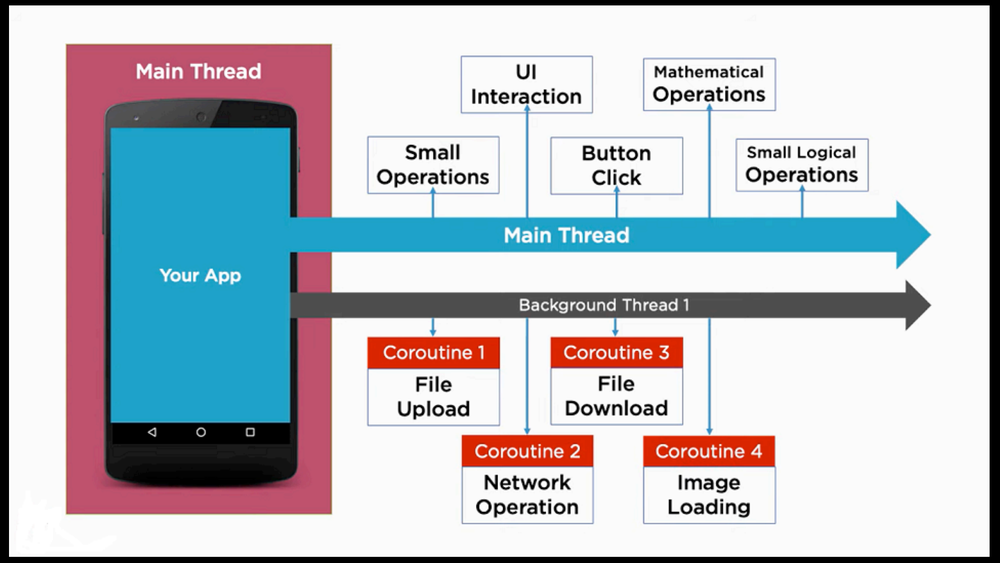
Future/Stream 톺아보기

Flutter 비동기 프로그래밍

suojae 24.06.17

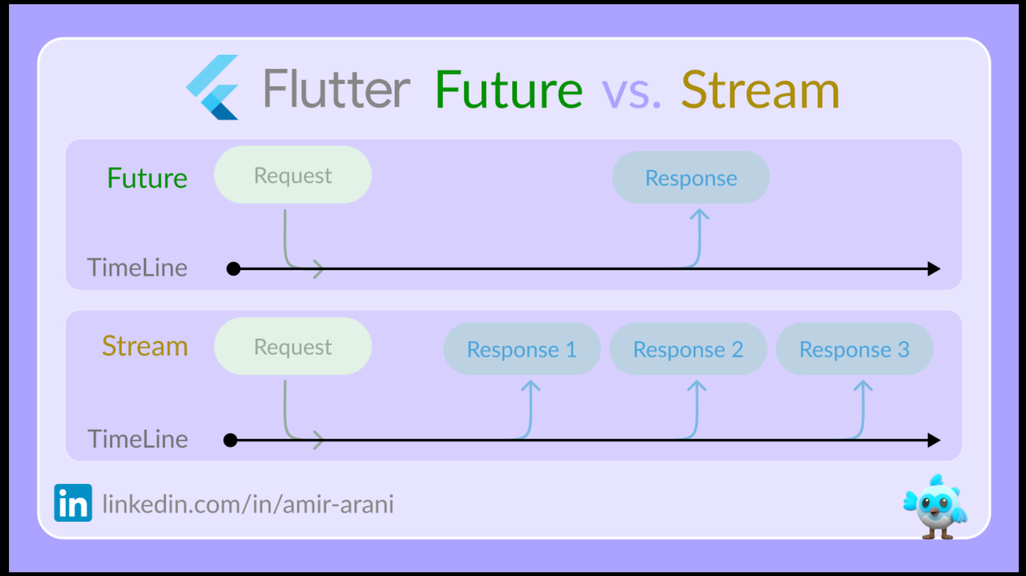
UI는 멈추지 말아야 한다

- UI 작업은 Main Thread 담당
- Main Thread가 멈추면 UI도 멈추게 된다
- 시간이 오래걸리는 작업은 하위 Thread로 보냄으로써 사용자에게는 멈추지 않는 UI 제공

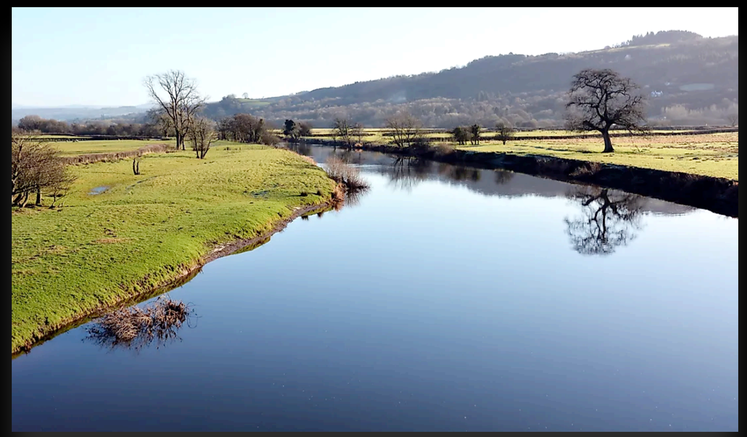


비동기 작업 결과를 보내는 방법

- 하위 Thread 결과물로 무언가 다른 작업을 수행
- 따라서 비동기 프로그래밍의 핵심은 **“끝”**을 파악하는 것
- **“끝”**지점을 받는 방법 call-back / stream



Future & Stream



고정된 결과를 한 번 제공하는 Future

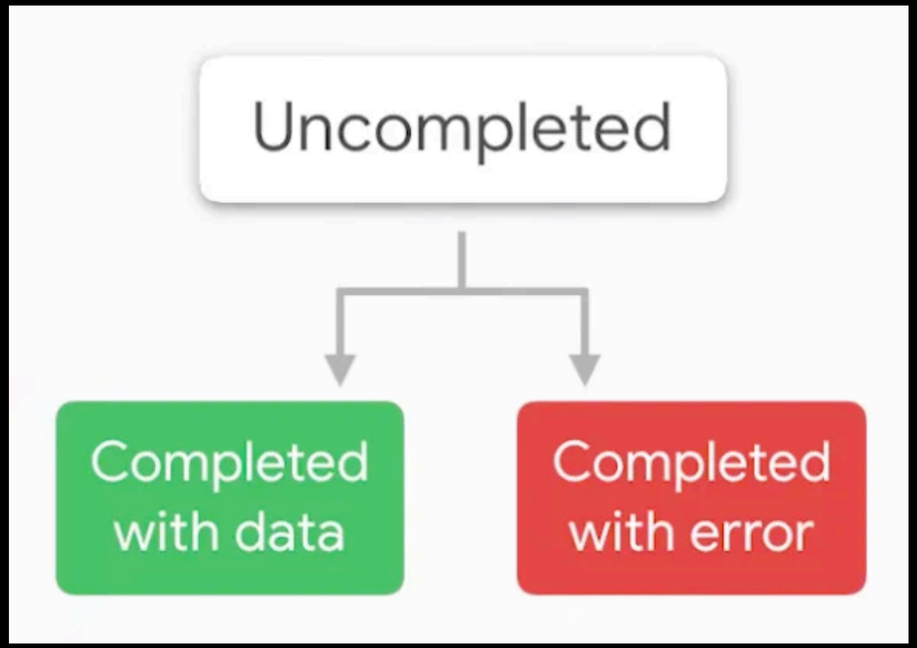
흐르는 데이터에서 값을 복사해오는 Stream

Future

Future 의 3가지 상태

```dart
// 비동기 작업을 모방하는 Future 함수
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2)); // 2초 지연
  return "Data fetched successfully!";
}
```

```dart
body: Center(
  child: FutureBuilder<String>(
    future: fetchData(), // Future를 제공
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return CircularProgressIndicator(); // 데이터 로드 중일 때
      } else if (snapshot.hasError) {
        return Text('Error: ${snapshot.error}'); // 에러가 발생했을 때
      } else {
        return Text('Result: ${snapshot.data}'); // 데이터 로드가 완료되었을 때
      }
    },
  ),
),
```



Future

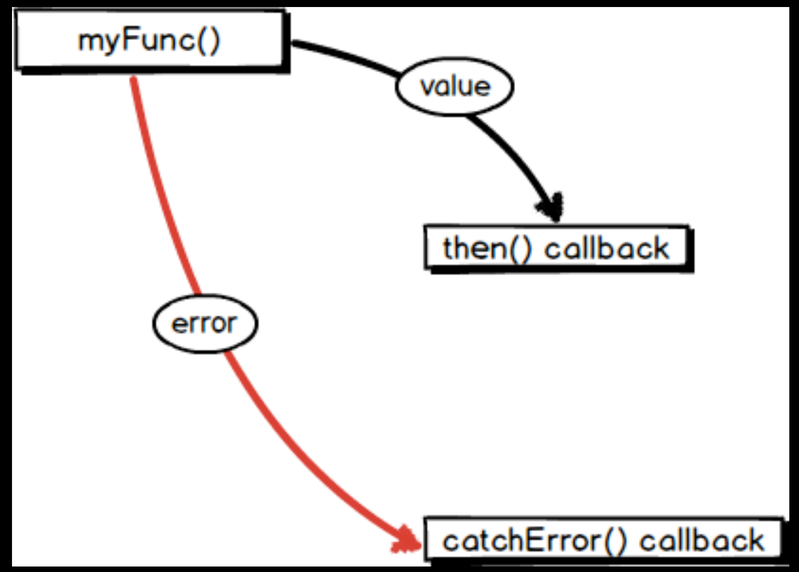
Future 간결하게 사용하기 - then / catch error

```dart
String _result = "Press the button to start";

Future<String> _fetchData() async {
  // 비동기 작업을 모방하기 위해 2초 지연
  await Future.delayed(Duration(seconds: 2));
  // 작업 완료 후 결과 반환
  return "Data fetched successfully!";
}

void _startFetching() {
  setState(() {
    _result = "Fetching data...";
  });

  // Future 작업 시작
  _fetchData()
    .then((data) {
      setState(() {
        _result = data;
      });
    })
    .catchError((error) {
      setState(() {
        _result = "Error: $error";
      });
     });
}
```



Future

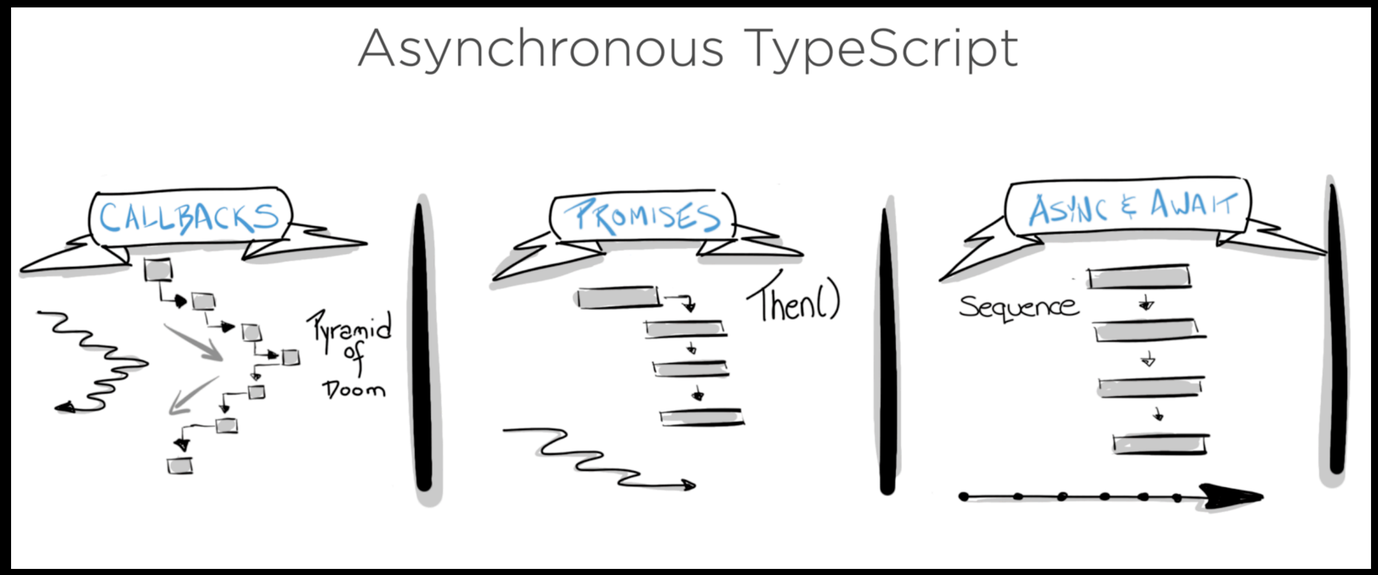
Future 간결하게 사용하기 - async / await

```dart
String _result = "Press the button to start";

Future<String> _fetchData() async {
  // 비동기 작업을 모방하기 위해 2초 지연
  await Future.delayed(Duration(seconds: 2));
  // 작업 완료 후 결과 반환
  return "Data fetched successfully!";
}

void _startFetching() async {
  setState(() {
    _result = "Fetching data...";
  });

  try {
    String data = await _fetchData();
    setState(() {
      _result = data;
    });
  } catch (error) {
    setState(() {
      _result = "Error: $error";
    });
  }
}
```



Stream

상태 변화를 추적하고 싶을 때

```dart
// StreamController를 생성하여 Stream을 관리
final StreamController<int> _streamController = StreamController<int>();

@override
void initState() {
  super.initState();
  // 주기적으로 데이터를 방출하는 함수 호출
  _startStreamingData();
}

@override
void dispose() {
  // StreamController를 닫아줌
  _streamController.close();
  super.dispose();
}

void _startStreamingData() {
  // 1초마다 숫자를 증가시키며 데이터를 방출하는 Stream 생성
  Timer.periodic(Duration(seconds: 1), (timer) {
    _streamController.add(timer.tick);
    if (timer.tick >= 10) { // 10까지 카운트한 후 스트리밍 종료
      timer.cancel();
    }
  });
}
```

```dart
body: Center(
  child: StreamBuilder<int>(
    stream: _streamController.stream, // Stream을 제공
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return CircularProgressIndicator(); // 스트림이 대기 중일 때
      } else if (snapshot.hasError) {
        return Text('Error: ${snapshot.error}'); // 스트림에 에러가 있을 때
      } else if (snapshot.hasData) {
        return Text('Counter: ${snapshot.data}'); // 새로운 데이터가 있을 때
      } else {
        return Text('Stream closed'); // 스트림이 닫혔을 때
      }
    },
  ),
),
```

Stream은 메모리 누수의 위험이 있기때문에 필요한 경우에만 사용

Stream Builder

Stream 구독하기

```dart
int _counter = 0;

Stream<int> _counterStream() async* {
  while (true) {
    await Future.delayed(Duration(milliseconds: 2000)); //2초 텀
    yield _counter;
  }
}

void _incrementCounter() {
  setState(() {
    _counter++;
  });
}
```

```dart
body: Center(
  child: StreamBuilder<int>(
    stream: _counterStream(),
    initialData: _counter,
    builder: (context, snapshot) {
      return Text(
        'Counter: ${snapshot.data}',
        style: TextStyle(fontSize: 24),
      );
    },
  ),
),
```

Future와 마찬가지로 snapshot을 통해 데이터를 가져온다

차이점으로 Stream을 방출하는 async에는 *(asterisk) 을 붙인다

Stream Controller

Stream을 동적으로 처리하고 싶을 때

```dart
class StreamExample extends StatelessWidget {
  Stream<int> _counterStream() {
    return Stream<int>.periodic(Duration(seconds: 1), (count) => count + 1);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Stream Example'),
      ),
      body: Center(
        child: StreamBuilder<int>(
          stream: _counterStream(),
          builder: (context, snapshot) {
            if (!snapshot.hasData)  CircularProgressIndicator();
            return Text('Count: ${snapshot.data}', style: TextStyle(fontSize: 24));
          },
        ),
      ),
    );
  }
}
```

```dart
final StreamController<int> _streamController = StreamController<int>();
int _counter = 0;

void _incrementCounter() {
  _counter++;
  _streamController.sink.add(_counter);
}

@override
void dispose() {
  _streamController.close();
  super.dispose();
}
```

```dart
body: Center(
  child: StreamBuilder<int>(
    stream: _streamController.stream,
    builder: (context, snapshot) {
      if (!snapshot.hasData) Text('Press the button to start counting',
                                  style: TextStyle(fontSize: 24));
      return Text('Count: ${snapshot.data}', style: TextStyle(fontSize: 24));
    },
  ),
),
```

Stream은 외부에서 내부 값을 제어할 수 없다

사용자 이벤트에 반응하기 위해서는 StreamController가 필요하다