Flutter Mobile Programming:
Advance
(Integrative-Generative AI Edition)

Student Price Book Center

คำนำ

ในยุคของการพัฒนาแอปพลิเคชันมือถือที่เติบโตอย่างรวดเร็ว การสร้างแอปที่มีประสิทธิภาพ ครบฟีเจอร์ และตอบสนองผู้ใช้อย่างราบรื่นกลายเป็นความท้าทายสำหรับนักพัฒนาทุกระดับ หนังสือ **Flutter Mobile Programming: Advance** เล่มนี้ถูกออกแบบมาเพื่อเป็นคู่มือเชิงลึกสำหรับผู้ที่มีพื้นฐาน Flutter อยู่แล้วและต้องการยกระดับความสามารถไปสู่การพัฒนาแอปขั้นสูง ครอบคลุมทั้งการจัดการ State, การทำงานกับฐานข้อมูลและ API, การเข้าถึงฟีเจอร์ Native, การใช้งาน Firebase, การจัดการแผนที่และ Location, ตลอดจนการสร้างแอนิเมชันที่น่าสนใจ

ผู้พัฒนาจะได้เรียนรู้ **การจัดการ State ขั้นสูง** โดยเปรียบเทียบเฟรมเวิร์กยอดนิยมอย่าง Provider, Riverpod, Bloc, Redux และ GetX พร้อมตัวอย่างการสร้างแอปด้วย Bloc และการใช้ Streams กับ RxDart เพื่อจัดการข้อมูลแบบ Reactive ทำให้สามารถออกแบบโครงสร้างแอปที่ยืดหยุ่นและรองรับการขยายตัวในโปรเจกต์ขนาดใหญ่

นอกจากนี้ หนังสือยังเน้นการ **ทำงานกับฐานข้อมูลและ API ขั้นสูง** ครอบคลุมการทำ Pagination, การทำ Cache ข้อมูล API, การใช้ GraphQL Client และการสร้าง Offline Mode ด้วยฐานข้อมูล Local เช่น Hive และ Isar ช่วยให้ผู้พัฒนาสามารถสร้างแอปที่ตอบสนองเร็ว แม้ในสภาพแวดล้อมที่มีการเชื่อมต่อจำกัด

การเข้าถึง **ฟีเจอร์ Native** ของอุปกรณ์ก็เป็นอีกหัวข้อสำคัญ หนังสือเล่มนี้สอนการขอ Permission สำหรับ Camera, Storage และ Location การใช้ camera package เพื่อถ่ายภาพและเลือกไฟล์ การเล่นวิดีโอและเสียงด้วย video_player และ just_audio รวมถึงการส่ง Notification ด้วย Firebase Cloud Messaging (FCM) เพื่อให้แอปมีฟังก์ชันครบถ้วนและใช้งานได้เต็มประสิทธิภาพ

การเชื่อมต่อกับ **Firebase** ถูกอธิบายอย่างละเอียด ตั้งแต่การเชื่อมต่อ Firebase Project, การใช้ Firebase Authentication (Email, Google, Facebook), การจัดการ Firestore Database สำหรับ CRUD ข้อมูล, การอัปเดตแบบเรียลไทม์ด้วย Firestore Snapshot และการใช้งาน Firebase Storage สำหรับอัปโหลดและดาวน์โหลดไฟล์ ทำให้ผู้พัฒนาสามารถสร้างแอปที่ครบวงจรและรองรับการทำงานจริงได้อย่างมั่นใจ

สำหรับแอปที่ต้องใช้ข้อมูลตำแหน่งและแผนที่ บทนี้ครอบคลุมการ **ใช้งาน Map และ Location** ด้วย geolocator และ Google Map การวาง Marker และติดตามตำแหน่งแบบ Real-time ทำให้สามารถสร้างแอปนำทาง แอปติดตาม หรือแอปโซเชียลที่ตอบสนองต่อพิกัดผู้ใช้ได้อย่างแม่นยำ

สุดท้าย การสร้าง **แอนิเมชัน** เป็นอีกหนึ่งหัวใจสำคัญของ UX บทนี้สอนการใช้ AnimatedContainer, AnimatedOpacity, การสร้าง Custom Animation ด้วย AnimationController, การใช้ Hero Animation และการทำ Page Transition ระหว่างหน้า ช่วยให้นักพัฒนาสร้างแอปที่มีชีวิตชีวา น่าสนใจ และตอบสนองต่อผู้ใช้อย่างราบรื่น

ด้วยเนื้อหาและตัวอย่างเชิงลึกจากบทต่าง ๆ ผู้อ่านจะสามารถ **พัฒนาแอป Flutter ระดับ Advance** ได้อย่างครบวงจร มีทักษะและความเข้าใจทั้งเชิงทฤษฎีและเชิงปฏิบัติ พร้อมต่อยอดไปสู่การสร้างแอปพลิเคชันที่มีคุณภาพและรองรับการใช้งานจริงได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 12
การจัดการ State ขั้นสูง
(Advanced Stat Management)

เนื้อหา

- การจัดการ State ขั้นสูง
- การจัดการ State ขั้นสูง (Advanced State Management)
- ความแตกต่างระหว่าง Provider, Riverpod, Bloc, Redux, GetX
- การสร้างแอปตัวอย่างด้วย Bloc — รายละเอียดเชิงลึก
- การใช้ Streams ในการจัดการ State — อธิบายละเอียด
- การใช้ RxDart สำหรับ Reactive Programming — อธิบายโดยละเอียด
- ตัวอย่างบูรณาการ

บทนำบทที่ 12: การจัดการ State ขั้นสูง

การจัดการ State เป็นหัวใจสำคัญของการพัฒนาแอปพลิเคชันที่ซับซ้อนบนแพลตฟอร์มสมัยใหม่ โดยเฉพาะอย่างยิ่งในสภาพแวดล้อมการพัฒนาแบบ Reactive อย่าง Flutter การควบคุมและติดตามการเปลี่ยนแปลงของ State ให้ถูกต้อง มีความแม่นยำ และสอดคล้องกับการแสดงผลบนหน้าจอ ถือเป็นปัจจัยสำคัญที่ส่งผลต่อความเสถียร ความลื่นไหล และประสบการณ์ผู้ใช้ (User Experience)

ปัจจุบันมีเฟรมเวิร์กและไลบรารีมากมายที่ช่วยในการจัดการ State แต่ละตัวมีจุดแข็งและจุดอ่อนที่แตกต่างกัน เช่น **Provider** ซึ่งเป็นโซลูชันมาตรฐานและเข้าใจง่าย **Riverpod** ที่แก้ไขข้อจำกัดของ Provider และทำงานแบบปลอดภัยจาก Context **Bloc** ที่เน้นการแยกหน้าที่ชัดเจนและทดสอบง่าย **Redux** ที่ใช้แนวคิด Single Source of Truth และเหมาะกับแอปขนาดใหญ่ และ **GetX** ที่เน้นความกระชับและประสิทธิภาพสูง

บทนี้จะเริ่มต้นด้วยการอธิบายความแตกต่างเชิงสถาปัตยกรรมและหลักการทำงานของ Provider, Riverpod, Bloc, Redux และ GetX พร้อมทั้งวิเคราะห์กรณีการใช้งาน (Use Cases) ที่เหมาะสม เพื่อให้ผู้อ่านสามารถเลือกเครื่องมือจัดการ State ได้อย่างเหมาะสมกับโครงสร้างและความต้องการของโครงการ

หนึ่งในหัวข้อสำคัญของบทนี้คือการสร้างแอปพลิเคชันตัวอย่างโดยใช้ **Bloc** ซึ่งจะช่วยให้เข้าใจแนวคิด Event-Driven Architecture และการจัดการ State แบบชัดเจน ด้วยการแยกการรับเหตุการณ์ (Event) และการอัปเดตสถานะ (State) ออกจากกันอย่างเป็นระบบ ส่งผลให้โค้ดมีความเป็นโมดูลสูงและบำรุงรักษาได้ง่าย

นอกจากนี้ บทนี้ยังครอบคลุมการใช้ **Streams** เพื่อจัดการข้อมูลที่เปลี่ยนแปลงอย่างต่อเนื่อง ซึ่งเป็นแนวคิดสำคัญในโลกของการพัฒนาแอปเชิง Reactive Streams ช่วยให้การอัปเดต UI แบบ Real-time มีความราบรื่น และสามารถประมวลผลข้อมูลจากหลายแหล่งได้อย่างมีประสิทธิภาพ

สำหรับผู้ที่ต้องการยกระดับการพัฒนาไปอีกขั้น บทนี้จะนำเสนอการใช้ **RxDart** ซึ่งเป็นการต่อยอดความสามารถของ Streams ให้รองรับรูปแบบการเขียนโปรแกรมเชิง Reactive Programming อย่างเต็มรูปแบบ ด้วยชุด Operator ที่หลากหลาย ทำให้นักพัฒนาสามารถจัดการข้อมูลที่ซับซ้อนได้อย่างยืดหยุ่น

เมื่ออ่านบทนี้จบ ผู้อ่านจะมีความเข้าใจทั้งในเชิงทฤษฎีและปฏิบัติ เกี่ยวกับเครื่องมือจัดการ State ขั้นสูง สามารถเลือกใช้เทคโนโลยีให้เหมาะสมกับสถานการณ์ ออกแบบโครงสร้างโค้ดที่รองรับการเติบโตของแอป และพัฒนาระบบที่มีประสิทธิภาพทั้งในด้านการประมวลผลและการแสดงผลต่อผู้ใช้ได้อย่างมั่นใจ

การจัดการ State ขั้นสูง

- ความแตกต่างระหว่าง Provider, Riverpod, Bloc, Redux, GetX
- สร้างแอปตัวอย่างด้วย Bloc
- การใช้ Streams ในการจัดการ State
- การใช้ RxDart สำหรับ Reactive Programming

---

1. ความแตกต่างระหว่าง Provider, Riverpod, Bloc, Redux, GetX

เทคนิค/แพ็กเกจ	แนวคิดหลัก	ข้อดี	ข้อควรระวัง/ข้อจำกัด
Provider	ใช้งานง่าย บนพื้นฐาน InheritedWidget และ ChangeNotifier	เรียนรู้เร็ว, integration กับ Flutter ดี, ใช้กันแพร่หลาย	ไม่เหมาะกับแอปซับซ้อนมากๆ, ไม่มีโครงสร้างจัดการ event
Riverpod	Provider แบบปรับปรุงใหม่, เขียนแบบปลอดภัย, test ง่าย	ปลอดภัยกว่า Provider, รองรับ Async, test ง่าย, ไม่ผูกกับ Widget tree	ต้องเรียนรู้ใหม่ มี concept เพิ่มขึ้น
Bloc (Business Logic Component)	Pattern แยก Business Logic ออกจาก UI โดยใช้ Event/State, Streams	โครงสร้างชัดเจน, รองรับ event-driven, ดีสำหรับแอปใหญ่	โค้ด verbose, เรียนรู้ค่อนข้างยาก
Redux	State container ที่จัดการ State เป็น Store เดียว ใช้ reducer และ action	ควบคุม State ได้ดี, predictable, debug ง่าย	boilerplate เยอะ, เหมาะกับแอปขนาดใหญ่

เทคนิค/แพ็กเกจ	แนวคิดหลัก	ข้อดี	ข้อควรระวัง/ข้อจำกัด
GetX	Framework ครบวงจร มี State Management, Routing, Dependency Injection	เรียนรู้ง่าย, สั้นกระชับ, ไม่ต้อง boilerplate เยอะ	บางคนมองว่าใช้แล้วไม่เป็น Flutter แบบดั้งเดิม, มี API จำนวนมาก

---

2. สร้างแอปตัวอย่างด้วย Bloc (flutter_bloc)

หลักการสำคัญ

- **Event:** สิ่งที่เกิดขึ้น เช่น กดปุ่ม, โหลดข้อมูล
- **State:** สถานะของ UI ที่เปลี่ยนไปตาม Event
- **Bloc:** ตัวกลางรับ Event และแปลงเป็น State ผ่าน Streams

ตัวอย่าง Counter App ด้วย Bloc

2.1. เพิ่ม dependencies ใน pubspec.yaml

```
dependencies:
  flutter_bloc: ^8.1.1
  bloc: ^8.1.0
```

2.2. สร้างไฟล์ counter_bloc.dart

```
import 'package:bloc/bloc.dart';

// Event
abstract class CounterEvent {}

class Increment extends CounterEvent {}

class Decrement extends CounterEvent {}

// State
class CounterState {
  final int count;
  CounterState(this.count);
}
```

```
// Bloc
class CounterBloc extends Bloc<CounterEvent, CounterState> {
  CounterBloc() : super(CounterState(0)) {
    on<Increment>((event, emit) => emit(CounterState(state.count + 1)));
    on<Decrement>((event, emit) => emit(CounterState(state.count - 1)));
  }
}
```

2.3. ใช้งาน Bloc ใน UI (main.dart)

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'counter_bloc.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Bloc Counter Example',
      home: BlocProvider(
        create: (_) => CounterBloc(),
        child: const CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  const CounterPage({super.key});
  @override
  Widget build(BuildContext context) {
```

```
    final counterBloc = context.read<CounterBloc>();

    return Scaffold(
      appBar: AppBar(title: const Text('Bloc Counter')),
      body: Center(
        child: BlocBuilder<CounterBloc, CounterState>(
          builder: (_, state) {
            return Text('Count: ${state.count}', style: const TextStyle(fontSize: 40));
          },
        ),
      ),
      floatingActionButton: Column(
        mainAxisSize: MainAxisSize.min,
        children: [
          FloatingActionButton(
            child: const Icon(Icons.add),
            onPressed: () => counterBloc.add(Increment()),
          ),
          const SizedBox(height: 8),
          FloatingActionButton(
            child: const Icon(Icons.remove),
            onPressed: () => counterBloc.add(Decrement()),
          ),
        ],
      ),
    );
  }
}
```

---

3. การใช้ Streams ในการจัดการ State

- **Stream:** เป็น sequence ของข้อมูลที่ไหล (asynchronous data flow)
- Flutter ใช้ StreamBuilder เพื่อเชื่อม UI กับ Stream
- Bloc ใช้ Streams อยู่เบื้องหลัง เพื่อ broadcast State changes

ตัวอย่างง่าย ๆ

```
Stream<int> counterStream() async* {
  for (int i = 0; i <= 10; i++) {
    await Future.delayed(const Duration(seconds: 1));
    yield i;
  }
}
```

ใช้ใน UI ด้วย

```
StreamBuilder<int>(
  stream: counterStream(),
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text('Count: ${snapshot.data}');
    }
    return const CircularProgressIndicator();
  },
)
```

---

4. การใช้ RxDart สำหรับ Reactive Programming

- **RxDart** คือการเสริมฟีเจอร์ Reactive Extensions (Rx) บน Dart Streams
- ช่วยจัดการ Stream ได้ดีขึ้น เช่น combineLatest, debounce, throttle
- เหมาะกับการจัดการ event ซับซ้อน หรือ UI ที่ตอบสนองแบบ reactive

ตัวอย่าง RxDart กับ BehaviorSubject

```
import 'package:rxdart/rxdart.dart';

final BehaviorSubject<int> _counterSubject = BehaviorSubject<int>.seeded(0);

void increment() {
  _counterSubject.add(_counterSubject.value + 1);
}

Stream<int> get counterStream => _counterSubject.stream;
```

UI

```
StreamBuilder<int>(
  stream: counterStream,
```

```
  builder: (context, snapshot) {
    final count = snapshot.data ?? 0;
    return Text('Count: $count');
  },
);
```

สรุป

หัวข้อ	คำอธิบายสั้น
Provider	เรียบง่าย เหมาะสำหรับแอปขนาดเล็ก-กลาง
Riverpod	Provider รุ่นใหม่ ปลอดภัยขึ้น ทดสอบง่าย
Bloc	เน้น event/state ใช้ Streams, เหมาะแอปใหญ่
Redux	State container แบบเดียว, boilerplate มาก
GetX	Framework ครบครัน เรียนรู้ง่าย ใช้งานเร็ว
Streams	ข้อมูลไหลแบบ async เชื่อม UI ผ่าน StreamBuilder
RxDart	เพิ่มความสามารถจัดการ Stream แบบ reactive

การจัดการ State ขั้นสูง (Advanced State Management)

---

1. ความแตกต่างและพื้นฐานเชิงลึกของแต่ละเทคนิค

1.1 Provider

- **พื้นฐาน:**

  Provider คือ wrapper ของ InheritedWidget เพื่อช่วยให้ส่งข้อมูล (state/data) ผ่าน Widget tree ได้ง่ายและมีประสิทธิภาพ

- **หลักการ:**
  - ใช้กับ ChangeNotifier หรือ ValueNotifier เพื่อแจ้ง UI ว่าข้อมูลเปลี่ยนแปลง
  - มีการฟังข้อมูลผ่าน Consumer หรือ context.watch()
- **ข้อดี:**
  - เรียนรู้ง่าย, syntax เข้าใจง่าย
  - ไม่ต้องเขียน boilerplate เยอะ
- **ข้อจำกัด:**
  - ไม่เหมาะกับ logic ที่ซับซ้อนหรือ event เยอะ เพราะไม่มี concept ของ event/state separation

- **ตัวอย่าง:**

```
class CounterModel with ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}
```

1.2 Riverpod

- **พื้นฐาน:**

  Riverpod พัฒนาต่อยอดจาก Provider โดยเพิ่มความปลอดภัยและ testability
- **ข้อดีเด่น:**
  - ไม่ผูกกับ Widget tree (สามารถสร้างและทดสอบนอก context ได้)
  - ป้องกันปัญหา context หายหรือเรียกผิดเวลา
  - รองรับ AsyncValue สำหรับ Async State เช่น Future/Stream
- **แนวคิด:**
  - ใช้ Provider ที่สร้างขึ้นแบบ global โดยไม่ต้องผูก widget
- **ตัวอย่าง:**

```
final counterProvider = StateProvider<int>((ref) => 0);

Widget build(BuildContext context, WidgetRef ref) {
  final count = ref.watch(counterProvider);
  return Text('$count');
}
```

1.3 Bloc (Business Logic Component)

- **พื้นฐาน:**
  - Bloc ใช้แนวคิด Event ⟶ Bloc ⟶ State โดยแยกความรับผิดชอบของ UI และ Business Logic ออกจากกัน
  - Bloc ใช้ Streams เพื่อส่ง Event เข้าและ Broadcast State ออก
- **เหตุผล:**

    - เหมาะกับแอปที่มี flow ซับซ้อน, event เยอะ
    - เน้นแยกเลเยอร์และโครงสร้างให้จัดการง่าย
- **ส่วนประกอบสำคัญ:**
    - Event: อินพุตที่ UI ส่งเข้ามา เช่น กดปุ่ม, โหลดข้อมูล
    - State: สถานะของ UI ที่เปลี่ยนตาม Event
    - Bloc: ตัวกลางรับ Event และแปลงเป็น State ผ่าน Stream
- **ข้อดี:**
    - สร้างแอปที่ test ได้ง่าย
    - โค้ดสวยและจัดระเบียบดี
- **ข้อเสีย:**
    - Boilerplate เยอะ เรียนรู้ยากสำหรับมือใหม่
- **ไดอะแกรม:**

[UI] --> Event --> [Bloc] --> State --> [UI]

1.4 Redux

- **พื้นฐาน:**
    - Redux คือการจัดเก็บ State ทั้งหมดใน Store เดียว (single source of truth)
    - ใช้ Reducer (pure functions) เพื่อจัดการ State ตาม Action
- **ข้อดี:**
    - State predictable
    - Debug ง่ายด้วย DevTools
- **ข้อเสีย:**
    - Boilerplate เยอะมาก
    - ไม่ค่อยเหมาะกับแอปขนาดเล็กหรือกลาง
- **Redux flow:**

Action --> Reducer(State, Action) --> New State --> UI

1.5 GetX

- **พื้นฐาน:**
    - เป็น Framework ครบวงจรที่รวบรวม State Management, Routing และ Dependency Injection
- **จุดเด่น:**
    - ใช้งานง่าย, โค้ดสั้น
    - ไม่ต้องใช้ context ในการเรียกดู State

- **ข้อควรระวัง:**
  - API เยอะและบางส่วนแตกต่างจากแนวทาง Flutter ปกติ
  - บางคนมองว่าทำให้ codebase ซับซ้อนในระยะยาว

---

2. การสร้างแอปตัวอย่างด้วย Bloc (เจาะลึก)

2.1 Bloc State Management โดยใช้ Streams

- Bloc สร้างจากพื้นฐาน Streams: รับ Event ผ่าน Sink แล้วแปลงเป็น State ผ่าน Stream
- ใช้ flutter_bloc package ที่เป็น wrapper ง่ายขึ้น
- สร้าง Bloc โดยสืบทอด Bloc<Event, State> และ override method on<Event> เพื่อจับ Event ต่าง ๆ

2.2 ตัวอย่าง CounterBloc เชิงลึก

```
import 'package:bloc/bloc.dart';

// Event
abstract class CounterEvent {}

class Increment extends CounterEvent {}

class Decrement extends CounterEvent {}

// State
class CounterState {
  final int count;
  const CounterState(this.count);
}

// Bloc
class CounterBloc extends Bloc<CounterEvent, CounterState> {
  CounterBloc() : super(const CounterState(0)) {
    on<Increment>((event, emit) {
      emit(CounterState(state.count + 1));
    });

    on<Decrement>((event, emit) {
```

```
      emit(CounterState(state.count - 1));
    });
  }
}
```

- on<Event> ฟัง Event และใช้ callback เพื่อ emit State ใหม่
- State ต้องเป็น immutable (ประกาศ const) เพื่อป้องกันการเปลี่ยนแปลงโดยไม่ตั้งใจ

2.3 UI ใช้ BlocBuilder และ BlocProvider

```
BlocProvider(
  create: (context) => CounterBloc(),
  child: BlocBuilder<CounterBloc, CounterState>(
    builder: (context, state) {
      return Text('Count: ${state.count}');
    },
  ),
);
```

---

3. การใช้ Streams ในการจัดการ State

3.1 พื้นฐาน Streams

- **Stream** คือ sequence ของข้อมูลที่อาจเกิดขึ้นในอนาคต (asynchronous)
- ใช้สำหรับ event-driven หรือ data ที่เปลี่ยนแปลงตลอดเวลา เช่น ข้อมูล API, Sensor, User Input

3.2 StreamController

- ใช้ควบคุม Stream เช่น เพิ่มข้อมูลเข้า Stream หรือปิด Stream

```
final controller = StreamController<int>();

controller.stream.listen((data) {
  print('Data: $data');
});

controller.sink.add(1);
controller.sink.add(2);

controller.close();
```

3.3 Flutter ใช้ StreamBuilder เชื่อม UI กับ Stream

```
StreamBuilder<int>(
  stream: controller.stream,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text('Value: ${snapshot.data}');
    }
    return CircularProgressIndicator();
  },
);
```

---

4. การใช้ RxDart สำหรับ Reactive Programming (เชิงลึก)

4.1 RxDart คืออะไร?

- RxDart เป็น Dart extension บน Streams ที่เพิ่ม operator แบบ reactive เช่น
  - debounceTime(), throttle(), combineLatest(), switchMap(), distinct(), เป็นต้น
- ช่วยให้จัดการ event ซับซ้อนและ async flow ง่ายขึ้น

4.2 BehaviorSubject

- เป็น StreamController ที่เก็บค่า last value และ broadcast ให้ Listener ใหม่ทันที

```
final BehaviorSubject<int> _counter = BehaviorSubject.seeded(0);

Stream<int> get counterStream => _counter.stream;

void increment() {
  _counter.add(_counter.value + 1);
}

void dispose() {
  _counter.close();
}
```

4.3 Operator ตัวอย่าง

- **debounceTime:** รอจนกว่าจะไม่มี event ใหม่ในช่วงเวลาหนึ่ง ก่อนส่ง event ออกไป
- **combineLatest:** รวมหลาย stream ให้ได้ค่าล่าสุดจากแต่ละ stream

---

5. สรุปเชิงลึก

เทคโนโลยี	เหมาะกับ	ข้อดีเชิงลึก	ข้อจำกัดเชิงลึก

เทคโนโลยี	เหมาะกับ	ข้อดีเชิงลึก	ข้อจำกัดเชิงลึก
Provider	แอปขนาดเล็ก-กลาง, เรียบง่าย	ปรับแต่งง่าย, integration กับ Flutter	ไม่มีการจัดการ event/state separation
Riverpod	แอปขนาดเล็ก-กลาง, test-driven	ปลอดภัย, ไม่ผูกกับ Widget tree	ต้องเรียนรู้ syntax ใหม่
Bloc	แอปขนาดกลาง-ใหญ่, event-driven	โครงสร้างชัด, testable, ใช้ Streams	Boilerplate เยอะ, learning curve สูง
Redux	แอปใหญ่ที่ต้องการ predictable state	State มี single source, debug ง่าย	Boilerplate มาก, verbose
GetX	แอปทุกขนาด, developer ต้องการรวดเร็ว	เรียนรู้ง่าย, ไม่ต้องใช้ context	อาจสับสน API เยอะ, โค้ดที่ไม่ได้มาตรฐาน
Streams	Data flow แบบ asynchronous	เชื่อม UI กับ data real-time	จัดการ error/close stream ยาก
RxDart	Stream ซับซ้อน, reactive UI	Operator ครบถ้วน, ทำงานกับหลาย stream	ต้องเข้าใจ reactive programming

ความแตกต่างระหว่าง Provider, Riverpod, Bloc, Redux, GetX

ความแตกต่างระหว่าง Provider, Riverpod, Bloc, Redux, GetX

เทคโนโลยี	แนวคิดหลัก	การจัดการ State	การเรียนรู้ & ใช้งาน	โครงสร้างและความซับซ้อน	จุดเด่น	ข้อจำกัด
Provider	Wrapper บน InheritedWidget ใช้งานง่าย	ใช้ ChangeNotifier, ValueNotifier แจ้ง UI	เรียนรู้ง่าย, เหมาะมือใหม่	โครงสร้างเรียบง่าย	ผูกกับ Widget tree, integration ดี	ไม่เหมาะกับแอปที่มี logic ซับซ้อนมาก
Riverpod	Provider รุ่นใหม่ ปลอดภัยและ testable	State management ที่ไม่ผูกกับ Widget tree	Syntax ใหม่ ต้องเรียนรู้	โครงสร้างคล้าย Provider แต่แยกออกจาก widget	ปลอดภัยกว่า Provider, test ง่าย, Async	มี concept ใหม่ที่ต้องเรียนรู้

เทคโนโลยี	แนวคิดหลัก	การจัดการ State	การเรียนรู้ & ใช้งาน	โครงสร้างและความซับซ้อน	จุดเด่น	ข้อจำกัด
					support	
Bloc	Event ⟶ State pattern ใช้ Streams	แยก Business Logic ออกจาก UI โดยชัดเจน	เรียนรู้ยากกว่า Provider, ต้องเข้าใจ Streams	โครงสร้างชัดเจน เน้นแยกเลเยอร์	เหมาะกับแอปขนาดใหญ่, test ง่าย	Boilerplate เยอะ, ซับซ้อนสำหรับมือใหม่
Redux	Single Store + Reducer + Action	State เดียวใน Store แก้ไขผ่าน Reducer	มี boilerplate สูง ต้องเข้าใจ flux pattern	โครงสร้างเข้มงวด, เหมาะกับแอปใหญ่	Debug ง่าย, predictable state	ใช้งานซับซ้อน, verbose, ไม่เหมาะกับแอปเล็ก
GetX	Framework ครบวงจรทั้ง State, Routing, DI	ใช้ Reactive variables (Rx) และ Controller	เรียนรู้เร็ว, ใช้งานง่าย	โค้ดกระชับ, ออกแบบง่าย	ไม่ต้องใช้ context, API ครบ	บางคนมองว่า “too magic”, ใช้ผิดได้ง่าย

รายละเอียดเพิ่มเติม

Provider

- ใช้งานง่ายและมี community ใหญ่
- ต้องผูกอยู่กับ Widget tree ทำให้บางครั้งเกิดปัญหา context ใช้งานผิดพลาด
- เหมาะสำหรับแอปที่มี state ไม่ซับซ้อน หรือเริ่มต้นเรียนรู้ Flutter State Management

Riverpod

- ปรับปรุงจุดอ่อนของ Provider โดยไม่ต้องผูกกับ Widget tree
- สร้าง provider แบบ global ได้, test ได้ง่ายกว่า
- มีเครื่องมือช่วยจัดการ Async เช่น FutureProvider และ StreamProvider
- มี syntax ที่ใหม่และต้องทำความเข้าใจ

Bloc

- ใช้แนวคิด event-driven แบบ reactive programming
- แยกความรับผิดชอบอย่างชัดเจนระหว่าง UI และ business logic
- เหมาะกับแอปขนาดใหญ่ที่มี flow ซับซ้อนและต้องการ test ง่าย

- มี boilerplate มาก ต้องใช้ความเข้าใจ Streams และ asynchronous programming

Redux

- มาจาก JavaScript ecosystem
- State ของแอปถูกเก็บไว้ในที่เดียว (single source of truth)
- State จะถูกแก้ไขผ่าน reducer function ที่เป็น pure function
- ต้องเขียน action, reducer และ store เยอะ
- เหมาะกับแอปขนาดใหญ่ที่ต้องการ predictability สูง

GetX

- ครอบคลุมทั้ง State management, Routing และ Dependency Injection
- ใช้ reactive variables ที่มี syntax สั้นและใช้งานง่ายมาก
- ไม่ต้องใช้ BuildContext ในการอัพเดต UI
- มีความยืดหยุ่นสูง แต่บางคนมองว่ามี magic เยอะเกินไป
- เหมาะกับโปรเจกต์ที่ต้องการ rapid development

สรุปการเลือกใช้

เหมาะกับ	แนะนำเทคนิค
เริ่มต้นเรียนรู้ Flutter, แอปเล็ก	Provider
ต้องการ test ง่าย, แอปขนาดกลาง	Riverpod
แอปขนาดใหญ่, ต้องแยกเลเยอร์ชัดเจน	Bloc
แอปที่ต้องการ predictable state สูง, มีทีมใหญ่	Redux
ต้องการพัฒนาเร็ว, ใช้งานง่าย, โปรเจกต์เล็กถึงกลาง	GetX

ชุดตัวอย่างพื้นฐาน (3 ตัว)

เน้นแสดงแนวทางใช้และความแตกต่างง่าย ๆ ของ Provider, Riverpod, Bloc

---

1. Provider — Counter App

โครงสร้างไฟล์

lib/

  main.dart

main.dart

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
```

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => CounterModel(),
      child: const MyApp(),
    ),
  );
}

class CounterModel with ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('Provider Counter')),
        body: Center(
          child: Consumer<CounterModel>(
            builder: (_, counter, __) => Text('Count: ${counter.count}', style: const
TextStyle(fontSize: 32)),
          ),
        ),
        floatingActionButton: FloatingActionButton(
```

```
          onPressed: () => context.read<CounterModel>().increment(),
          child: const Icon(Icons.add),
        ),
      ),
    );
  }
}
```

คำอธิบาย

- ใช้ Provider แบบง่ายกับ ChangeNotifier
- UI ฟังค่าผ่าน Consumer
- เรียกเพิ่มค่าโดยใช้ context.read

---

2. Riverpod — Counter App

โครงสร้างไฟล์

```
lib/
  main.dart
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_riverpod/flutter_riverpod.dart';

final counterProvider = StateProvider<int>((ref) => 0);

void main() {
  runApp(const ProviderScope(child: MyApp()));
}

class MyApp extends ConsumerWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final count = ref.watch(counterProvider);
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: const Text('Riverpod Counter')),
```

```
      body: Center(child: Text('Count: $count', style: const TextStyle(fontSize: 32))),
      floatingActionButton: FloatingActionButton(
        onPressed: () => ref.read(counterProvider.notifier).state++,
        child: const Icon(Icons.add),
      ),
    ),
  );
 }
}
```

คำอธิบาย

- ใช้ Riverpod StateProvider
- ใช้ ProviderScope เป็น root widget
- ConsumerWidget รู้จัก ref สำหรับอ่าน/ฟัง state

---

3. Bloc — Counter App

โครงสร้างไฟล์

```
lib/
  main.dart
  counter_bloc.dart
```

counter_bloc.dart

```
import 'package:bloc/bloc.dart';

// Event
abstract class CounterEvent {}
class Increment extends CounterEvent {}

// State
class CounterState {
  final int count;
  const CounterState(this.count);
}

// Bloc
class CounterBloc extends Bloc<CounterEvent, CounterState> {
```

```
  CounterBloc() : super(const CounterState(0)) {
    on<Increment>((event, emit) => emit(CounterState(state.count + 1)));
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'counter_bloc.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => CounterBloc(),
        child: const CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  const CounterPage({super.key});
  @override
  Widget build(BuildContext context) {
    final bloc = context.read<CounterBloc>();
    return Scaffold(
      appBar: AppBar(title: const Text('Bloc Counter')),
      body: Center(
```

```
        child: BlocBuilder<CounterBloc, CounterState>(
          builder: (_, state) => Text('Count: ${state.count}', style: const TextStyle(fontSize: 32)),
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => bloc.add(Increment()),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

คำอธิบาย

- Bloc แยก Event และ State
- UI ฟัง State ผ่าน BlocBuilder
- ส่ง Event เข้า Bloc เพื่อเปลี่ยน State

ชุดตัวอย่างแนวประยุกต์ (3 ตัว)
เน้นแอปที่มี UI และฟีเจอร์มากขึ้น โดยใช้ Provider, Redux, GetX

---

1. Provider — Todo List

โครงสร้าง

```
lib/
  main.dart
  todo_model.dart
```

todo_model.dart

```
import 'package:flutter/foundation.dart';

class Todo {
  final String title;
  bool done;
  Todo(this.title, {this.done = false});
}

class TodoListModel with ChangeNotifier {
```

```
  final List<Todo> _items = [];
  List<Todo> get items => List.unmodifiable(_items);

  void add(String title) {
    _items.add(Todo(title));
    notifyListeners();
  }

  void toggleDone(int index) {
    _items[index].done = !_items[index].done;
    notifyListeners();
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'todo_model.dart';

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => TodoListModel(),
      child: const MyApp(),
    ),
  );
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(home: const TodoPage());
  }
}
```

```
class TodoPage extends StatelessWidget {
  const TodoPage({super.key});

  @override
  Widget build(BuildContext context) {
    final todoList = context.watch<TodoListModel>();
    final controller = TextEditingController();

    return Scaffold(
      appBar: AppBar(title: const Text('Provider Todo List')),
      body: Column(
        children: [
          Padding(
            padding: const EdgeInsets.all(8),
            child: TextField(
              controller: controller,
              onSubmitted: (value) {
                if (value.isNotEmpty) {
                  context.read<TodoListModel>().add(value);
                  controller.clear();
                }
              },
              decoration: const InputDecoration(labelText: 'Add Todo'),
            ),
          ),
          Expanded(
            child: ListView.builder(
              itemCount: todoList.items.length,
              itemBuilder: (_, i) {
                final todo = todoList.items[i];
                return ListTile(
                  title: Text(todo.title,
                      style: TextStyle(
```

```
                    decoration:
                        todo.done ? TextDecoration.lineThrough : null)),
                trailing: Checkbox(
                  value: todo.done,
                  onChanged: (_) => context.read<TodoListModel>().toggleDone(i),
                ),
              );
            },
          ),
        )
      ],
    ),
  );
 }
}
```

---

2. Redux — Simple Counter

โครงสร้าง

```
lib/
  main.dart
  redux_counter.dart
```

redux_counter.dart

```
import 'package:redux/redux.dart';

// Actions
class IncrementAction {}

// Reducer
int counterReducer(int state, dynamic action) {
  if (action is IncrementAction) {
    return state + 1;
  }
  return state;
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_redux/flutter_redux.dart';
import 'package:redux/redux.dart';
import 'redux_counter.dart';

void main() {
  final store = Store<int>(counterReducer, initialState: 0);
  runApp(MyApp(store: store));
}

class MyApp extends StatelessWidget {
  final Store<int> store;
  const MyApp({required this.store, super.key});

  @override
  Widget build(BuildContext context) {
    return StoreProvider(
      store: store,
      child: MaterialApp(home: const CounterPage()),
    );
  }
}

class CounterPage extends StatelessWidget {
  const CounterPage({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Redux Counter')),
      body: Center(
        child: StoreConnector<int, int>(
          converter: (store) => store.state,
          builder: (_, count) => Text('Count: $count', style: const TextStyle(fontSize: 32)),
```

```
        ),
      ),
      floatingActionButton: StoreConnector<int, VoidCallback>(
        converter: (store) {
          return () => store.dispatch(IncrementAction());
        },
        builder: (_, callback) => FloatingActionButton(
          onPressed: callback,
          child: const Icon(Icons.add),
        ),
      ),
    );
  }
}
```

---

3. GetX — Simple Counter with Reactive Variable

โครงสร้าง

```
lib/
  main.dart
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';

class CounterController extends GetxController {
  var count = 0.obs;

  void increment() {
    count++;
  }
}

void main() {
  runApp(const MyApp());
}
```

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      home: CounterPage(),
    );
  }
}

class CounterPage extends StatelessWidget {
  final CounterController controller = Get.put(CounterController());

  CounterPage({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('GetX Counter')),
      body: Center(
        child: Obx(() => Text('Count: ${controller.count}', style: const TextStyle(fontSize: 32))),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: controller.increment,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

ผลการรัน

- ทั้ง 6 ตัวอย่างนี้จะเปิดแอปที่มีปุ่ม + ให้กดเพิ่มค่า count หรือเพิ่มรายการ todo และแสดงผลลัพธ์บนหน้าจอทันที

- ความแตกต่างหลัก ๆ อยู่ที่วิธีการจัดการ state, การแจ้ง UI, และโครงสร้างของโค้ดที่แต่ละเทคนิคใช้

การสร้างแอปตัวอย่างด้วย Bloc — รายละเอียดเชิงลึก

---

1. แนวคิดหลักของ Bloc

- **Bloc = Business Logic Component**
  เป็น pattern ที่แยกหน้าที่ของการจัดการข้อมูล (business logic) ออกจาก UI
- **Event ⟶ Bloc ⟶ State**
  UI ส่ง **Event** เข้าไปใน Bloc ⟶ Bloc ประมวลผลและเปลี่ยนแปลงข้อมูล ⟶ ส่ง **State** ใหม่กลับมา ⟶ UI รับ State มาปรับการแสดงผล
- **State คือข้อมูลที่ UI แสดง**
- **Event คือสิ่งที่เกิดขึ้น (เช่น user กดปุ่ม)**
- **Bloc ใช้ Streams ในการรับ Event และส่ง State** ทำให้เป็นระบบ reactive

---

2. โครงสร้างหลักของ Bloc

- **Event class:**
  นิยาม event ทั้งหมดที่ Bloc จะรับ เช่น Increment, Decrement
- **State class:**
  นิยามสถานะทั้งหมดที่ UI สามารถแสดงได้ (ส่วนใหญ่เป็น immutable)
- **Bloc class:**
  รับ Event และแปลงเป็น State ผ่านฟังก์ชันที่กำหนดไว้ เช่น on<Event>((event, emit) => ...)

---

3. ตัวอย่างแอป Counter ด้วย Bloc

3.1 สร้างไฟล์ counter_bloc.dart

```
import 'package:bloc/bloc.dart';

// 1. Event
abstract class CounterEvent {}

class IncrementEvent extends CounterEvent {}

class DecrementEvent extends CounterEvent {}
```

```
// 2. State
class CounterState {
  final int count;
  const CounterState(this.count);
}

// 3. Bloc
class CounterBloc extends Bloc<CounterEvent, CounterState> {
  CounterBloc() : super(const CounterState(0)) {
    on<IncrementEvent>((event, emit) {
      emit(CounterState(state.count + 1));
    });

    on<DecrementEvent>((event, emit) {
      emit(CounterState(state.count - 1));
    });
  }
}
```

อธิบาย:

- สร้าง abstract class CounterEvent เพื่อเป็นฐานของ event
- สร้าง event 2 ตัวคือ IncrementEvent กับ DecrementEvent
- สร้าง CounterState เพื่อเก็บจำนวน count ที่ UI ต้องการแสดง
- สร้าง CounterBloc ที่เริ่มต้น count = 0
- ใน constructor ใช้ on<Event> เพื่อตั้ง handler ว่าถ้าเจอ IncrementEvent ให้เพิ่ม count และส่ง state ใหม่ออกมา

3.2 สร้างไฟล์ main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'counter_bloc.dart';

void main() {
  runApp(const MyApp());
}
```

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Bloc Counter Demo',
      home: BlocProvider(
        create: (_) => CounterBloc(),
        child: const CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  const CounterPage({super.key});
  @override
  Widget build(BuildContext context) {
    final bloc = context.read<CounterBloc>();

    return Scaffold(
      appBar: AppBar(title: const Text('Bloc Counter')),
      body: Center(
        child: BlocBuilder<CounterBloc, CounterState>(
          builder: (context, state) {
            return Text('Count: ${state.count}', style: const TextStyle(fontSize: 48));
          },
        ),
      ),
      floatingActionButton: Row(
        mainAxisAlignment: MainAxisAlignment.end,
        children: [
          FloatingActionButton(
```

```
            heroTag: 'dec',
            onPressed: () => bloc.add(DecrementEvent()),
            child: const Icon(Icons.remove),
          ),
          const SizedBox(width: 20),
          FloatingActionButton(
            heroTag: 'inc',
            onPressed: () => bloc.add(IncrementEvent()),
            child: const Icon(Icons.add),
          ),
        ],
      ),
    );
  }
}
```

อธิบาย:

- BlocProvider เป็น widget ที่สร้างและแจกจ่าย Bloc ให้ Widget ลูก ๆ ได้ใช้งาน
- ใช้ BlocBuilder ฟัง State ที่ Bloc ส่งออกมา แล้วสร้าง UI ใหม่ทุกครั้งเมื่อ State เปลี่ยน
- เมื่อผู้ใช้กดปุ่ม + หรือ - จะส่ง Event (IncrementEvent, DecrementEvent) เข้า Bloc เพื่อเปลี่ยน State
- UI จะแสดงค่าปัจจุบันของ count ตาม State ที่ได้รับ

---

4. ผลการรัน

- เมื่อเปิดแอปจะเห็นตัวเลข 0 ตรงกลางจอ
- กดปุ่ม + ตัวเลขจะเพิ่มขึ้นทีละ 1
- กดปุ่ม - ตัวเลขจะลดลงทีละ 1
- UI จะอัพเดตแบบ realtime ตาม State ของ Bloc

---

5. ข้อดีของการใช้ Bloc ในกรณีนี้

- **แยก Business Logic ออกจาก UI:**
  โค้ดเพิ่ม/ลดค่าอยู่ใน Bloc ไม่ผสมกับ UI widget ทำให้โค้ดสะอาดและแก้ไขง่าย
- **Test ได้ง่าย:**
  เราสามารถเขียน unit test ตรวจสอบได้ว่า เมื่อส่ง Event เข้า Bloc แล้วจะได้ State ถูกต้อง

- **จัดการ State แบบ reactive:**
  Bloc ใช้ Streams ส่ง State ใหม่ ทำให้ UI update อัตโนมัติ
- **ปรับขนาดได้:**
  แม้แอปจะซับซ้อนขึ้น ก็ยังสามารถจัดการ event/state ได้ง่ายและเป็นระบบ

ชุดตัวอย่างพื้นฐาน (3 โปรแกรม)
แสดงการใช้ Bloc กับแอปง่าย ๆ ที่เน้นแนวคิดหลัก

---

1. Counter App with Increment/Decrement (Bloc)

โครงสร้างไฟล์

```
lib/
├── main.dart
└── counter_bloc.dart
```

counter_bloc.dart

```
import 'package:bloc/bloc.dart';

abstract class CounterEvent {}
class IncrementEvent extends CounterEvent {}
class DecrementEvent extends CounterEvent {}

class CounterState {
  final int count;
  const CounterState(this.count);
}

class CounterBloc extends Bloc<CounterEvent, CounterState> {
  CounterBloc() : super(const CounterState(0)) {
    on<IncrementEvent>((event, emit) {
      emit(CounterState(state.count + 1));
    });
    on<DecrementEvent>((event, emit) {
      emit(CounterState(state.count - 1));
    });
  }
```

```
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'counter_bloc.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => CounterBloc(),
        child: const CounterPage(),
      ),
    );
  }
}

class CounterPage extends StatelessWidget {
  const CounterPage({super.key});
  @override
  Widget build(BuildContext context) {
    final bloc = context.read<CounterBloc>();
    return Scaffold(
      appBar: AppBar(title: const Text('Bloc Counter')),
      body: Center(
        child: BlocBuilder<CounterBloc, CounterState>(
          builder: (_, state) {
            return Text('Count: ${state.count}', style: const TextStyle(fontSize: 48));
```

```
          },
        ),
      ),
      floatingActionButton: Row(
        mainAxisAlignment: MainAxisAlignment.end,
        children: [
          FloatingActionButton(
            heroTag: 'dec',
            onPressed: () => bloc.add(DecrementEvent()),
            child: const Icon(Icons.remove),
          ),
          const SizedBox(width: 20),
          FloatingActionButton(
            heroTag: 'inc',
            onPressed: () => bloc.add(IncrementEvent()),
            child: const Icon(Icons.add),
          ),
        ],
      ),
    );
  }
}
```

---

2. Simple Toggle Switch with Bloc

โครงสร้างไฟล์

```
lib/
 ├── main.dart
 └── toggle_bloc.dart
```

toggle_bloc.dart

```
import 'package:bloc/bloc.dart';

abstract class ToggleEvent {}
class TogglePressed extends ToggleEvent {}
```

```
class ToggleState {
  final bool isOn;
  const ToggleState(this.isOn);
}

class ToggleBloc extends Bloc<ToggleEvent, ToggleState> {
  ToggleBloc() : super(const ToggleState(false)) {
    on<TogglePressed>((event, emit) {
      emit(ToggleState(!state.isOn));
    });
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'toggle_bloc.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => ToggleBloc(),
        child: const TogglePage(),
      ),
    );
  }
}
```

```
class TogglePage extends StatelessWidget {
  const TogglePage({super.key});
  @override
  Widget build(BuildContext context) {
    final bloc = context.read<ToggleBloc>();
    return Scaffold(
      appBar: AppBar(title: const Text('Bloc Toggle')),
      body: Center(
        child: BlocBuilder<ToggleBloc, ToggleState>(
          builder: (_, state) {
            return Text(
              state.isOn ? 'ON' : 'OFF',
              style: const TextStyle(fontSize: 48, color: Colors.blue),
            );
          },
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => bloc.add(TogglePressed()),
        child: const Icon(Icons.power_settings_new),
      ),
    );
  }
}
```

---

3. Simple Counter with Delay (Simulating Async) using Bloc

โครงสร้างไฟล์

```
lib/
  ├── main.dart
  └── async_counter_bloc.dart
```

async_counter_bloc.dart

```
import 'package:bloc/bloc.dart';

abstract class AsyncCounterEvent {}
```

```
class IncrementAsyncEvent extends AsyncCounterEvent {}

class AsyncCounterState {
  final int count;
  final bool isLoading;
  const AsyncCounterState(this.count, this.isLoading);
}

class AsyncCounterBloc extends Bloc<AsyncCounterEvent, AsyncCounterState> {
  AsyncCounterBloc() : super(const AsyncCounterState(0, false)) {
    on<IncrementAsyncEvent>((event, emit) async {
      emit(AsyncCounterState(state.count, true));
      await Future.delayed(const Duration(seconds: 1));
      emit(AsyncCounterState(state.count + 1, false));
    });
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'async_counter_bloc.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => AsyncCounterBloc(),
        child: const AsyncCounterPage(),
```

```
      ),
    );
  }
}

class AsyncCounterPage extends StatelessWidget {
  const AsyncCounterPage({super.key});
  @override
  Widget build(BuildContext context) {
    final bloc = context.read<AsyncCounterBloc>();
    return Scaffold(
      appBar: AppBar(title: const Text('Async Bloc Counter')),
      body: Center(
        child: BlocBuilder<AsyncCounterBloc, AsyncCounterState>(
          builder: (_, state) {
            if (state.isLoading) {
              return const CircularProgressIndicator();
            }
            return Text('Count: ${state.count}', style: const TextStyle(fontSize: 48));
          },
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => bloc.add(IncrementAsyncEvent()),
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

ชุดตัวอย่างแนวประยุกต์ (3 โปรแกรม)

เน้นแอปที่มี UI ซับซ้อนและการจัดการ State ที่ใช้ Bloc

---

1. Todo List with Bloc

โครงสร้างไฟล์

```
lib/
  ├── main.dart
  ├── todo_bloc.dart
  ├── todo_event.dart
  └── todo_state.dart
```

todo_event.dart

```
abstract class TodoEvent {}

class AddTodo extends TodoEvent {
  final String task;
  AddTodo(this.task);
}

class ToggleTodo extends TodoEvent {
  final int index;
  ToggleTodo(this.index);
}
```

todo_state.dart

```
class TodoState {
  final List<TodoItem> todos;
  TodoState(this.todos);
}

class TodoItem {
  final String task;
  bool completed;
  TodoItem(this.task, {this.completed = false});
}
```

todo_bloc.dart

```
import 'package:bloc/bloc.dart';
import 'todo_event.dart';
import 'todo_state.dart';
```

```
class TodoBloc extends Bloc<TodoEvent, TodoState> {
  TodoBloc() : super(TodoState([])) {
    on<AddTodo>((event, emit) {
      final newTodos = List<TodoItem>.from(state.todos)
        ..add(TodoItem(event.task));
      emit(TodoState(newTodos));
    });

    on<ToggleTodo>((event, emit) {
      final newTodos = List<TodoItem>.from(state.todos);
      final todo = newTodos[event.index];
      todo.completed = !todo.completed;
      emit(TodoState(newTodos));
    });
  }
}
```

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'todo_bloc.dart';
import 'todo_event.dart';
import 'todo_state.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: BlocProvider(
        create: (_) => TodoBloc(),
```

```
        child: const TodoPage(),
      ),
    );
  }
}

class TodoPage extends StatefulWidget {
  const TodoPage({super.key});
  @override
  State<TodoPage> createState() => _TodoPageState();
}

class _TodoPageState extends State<TodoPage> {
  final TextEditingController _controller = TextEditingController();

  @override
  Widget build(BuildContext context) {
    final bloc = context.read<TodoBloc>();
    return Scaffold(
      appBar: AppBar(title: const Text('Todo List with Bloc')),
      body: Column(
        children: [
          Padding(
            padding: const EdgeInsets.all(8),
            child: TextField(
              controller: _controller,
              decoration: const InputDecoration(labelText: 'New Task'),
              onSubmitted: (value) {
                if (value.isNotEmpty) {
                  bloc.add(AddTodo(value));
                  _controller.clear();
                }
              },
            ),
```

```
          ),
          Expanded(
            child: BlocBuilder<TodoBloc, TodoState>(
              builder: (_, state) {
                return ListView.builder(
                  itemCount: state.todos.length,
                  itemBuilder: (_, index) {
                    final todo = state.todos[index];
                    return ListTile(
                      title: Text(
                        todo.task,
                        style: TextStyle(
                          decoration: todo.completed
                              ? TextDecoration.lineThrough
                              : null,
                        ),
                      ),
                      trailing: Checkbox(
                        value: todo.completed,
                        onChanged: (_) {
                          bloc.add(ToggleTodo(index));
                        },
                      ),
                    );
                  },
                );
              },
            ),
          ),
        ],
      ),
    );
  }
}
```

2. Login Form with Validation and Bloc

แอปตัวอย่างที่มี

- ฟอร์ม Login (email, password)
- ตรวจสอบความถูกต้องข้อมูล (validate) แบบ realtime ด้วย Bloc
- แสดงสถานะการล็อกอิน (loading, success, failure)

โครงสร้างไฟล์

```
lib/
 ├── main.dart
 ├── login_bloc.dart
 ├── login_event.dart
 └── login_state.dart
```

---

login_event.dart

```
abstract class LoginEvent {}

class EmailChanged extends LoginEvent {
  final String email;
  EmailChanged(this.email);
}

class PasswordChanged extends LoginEvent {
  final String password;
  PasswordChanged(this.password);
}

class LoginSubmitted extends LoginEvent {}
```

---

login_state.dart

```
enum FormStatus { initial, submitting, success, failure }

class LoginState {
  final String email;
```

```
  final String password;
  final bool isEmailValid;
  final bool isPasswordValid;
  final FormStatus status;
  final String errorMessage;

  LoginState({
    this.email = '',
    this.password = '',
    this.isEmailValid = true,
    this.isPasswordValid = true,
    this.status = FormStatus.initial,
    this.errorMessage = '',
  });

  LoginState copyWith({
    String? email,
    String? password,
    bool? isEmailValid,
    bool? isPasswordValid,
    FormStatus? status,
    String? errorMessage,
  }) {
    return LoginState(
      email: email ?? this.email,
      password: password ?? this.password,
      isEmailValid: isEmailValid ?? this.isEmailValid,
      isPasswordValid: isPasswordValid ?? this.isPasswordValid,
      status: status ?? this.status,
      errorMessage: errorMessage ?? this.errorMessage,
    );
  }
}
```

login_bloc.dart

```
import 'package:bloc/bloc.dart';
import 'login_event.dart';
import 'login_state.dart';

class LoginBloc extends Bloc<LoginEvent, LoginState> {
  LoginBloc() : super(LoginState()) {
    on<EmailChanged>((event, emit) {
      final isValid = _validateEmail(event.email);
      emit(state.copyWith(email: event.email, isEmailValid: isValid));
    });

    on<PasswordChanged>((event, emit) {
      final isValid = _validatePassword(event.password);
      emit(state.copyWith(password: event.password, isPasswordValid: isValid));
    });

    on<LoginSubmitted>((event, emit) async {
      if (!_validateEmail(state.email) || !_validatePassword(state.password)) {
        emit(state.copyWith(
          isEmailValid: false,
          isPasswordValid: false,
          status: FormStatus.failure,
          errorMessage: 'Invalid email or password',
        ));
        return;
      }

      emit(state.copyWith(status: FormStatus.submitting));

      // Simulate login delay
      await Future.delayed(const Duration(seconds: 2));

      // Simple dummy check
```

```
      if (state.email == 'test@example.com' && state.password == 'password') {
        emit(state.copyWith(status: FormStatus.success));
      } else {
        emit(state.copyWith(
          status: FormStatus.failure,
          errorMessage: 'Login failed. Incorrect credentials.',
        ));
      }
    });
  }

  bool _validateEmail(String email) {
    return RegExp(r'^[^@]+@[^@]+\.[^@]+').hasMatch(email);
  }

  bool _validatePassword(String password) {
    return password.length >= 6;
  }
}
```

---

main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'login_bloc.dart';
import 'login_event.dart';
import 'login_state.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Login Form with Bloc',
      home: BlocProvider(
        create: (_) => LoginBloc(),
        child: const LoginPage(),
      ),
    );
  }
}

class LoginPage extends StatelessWidget {
  const LoginPage({super.key});

  @override
  Widget build(BuildContext context) {
    final bloc = context.read<LoginBloc>();
    return Scaffold(
      appBar: AppBar(title: const Text('Login Form')),
      body: Padding(
        padding: const EdgeInsets.all(16),
        child: BlocConsumer<LoginBloc, LoginState>(
          listener: (context, state) {
            if (state.status == FormStatus.success) {
              ScaffoldMessenger.of(context).showSnackBar(const SnackBar(content: Text('Login
Successful!')));
            }
            if (state.status == FormStatus.failure && state.errorMessage.isNotEmpty) {
              ScaffoldMessenger.of(context).showSnackBar(SnackBar(content:
Text(state.errorMessage)));
            }
          },
          builder: (context, state) {
            return Column(
```

```
            children: [
              TextField(
                onChanged: (email) => bloc.add(EmailChanged(email)),
                decoration: InputDecoration(
                  labelText: 'Email',
                  errorText: state.isEmailValid ? null : 'Invalid email format',
                ),
              ),
              const SizedBox(height: 16),
              TextField(
                obscureText: true,
                onChanged: (password) => bloc.add(PasswordChanged(password)),
                decoration: InputDecoration(
                  labelText: 'Password',
                  errorText: state.isPasswordValid ? null : 'Password too short',
                ),
              ),
              const SizedBox(height: 32),
              state.status == FormStatus.submitting
                  ? const CircularProgressIndicator()
                  : ElevatedButton(
                      onPressed: () => bloc.add(LoginSubmitted()),
                      child: const Text('Login'),
                    ),
            ],
          );
        },
      ),
    ),
  );
 }
}
```

อธิบายโดยสังเขป

- UI ใช้ BlocConsumer เพื่อฟังสถานะของฟอร์มและแสดง error/success snackbar
- ทุกครั้งที่ผู้ใช้เปลี่ยน email หรือ password จะส่ง event เข้า Bloc
- Bloc ตรวจสอบความถูกต้องและอัปเดตสถานะพร้อม error message
- เมื่อกด Login จะตรวจสอบและจำลองการล็อกอินด้วย delay
- UI เปลี่ยนแปลงตามสถานะ (loading, success, failure)

---

3. Weather Fetch App with Bloc and API

แอปดึงข้อมูลสภาพอากาศจาก API, ใช้ Bloc จัดการสถานะโหลดข้อมูล, แสดงผลลัพธ์, และจัดการ error

บอกได้เลยครับว่าจะเริ่มเมื่อไหร่!

ได้เลยครับ! ผมจะสร้างตัวอย่างแอป **Weather Fetch App** โดยใช้ **Bloc** เพื่อ

- ดึงข้อมูลสภาพอากาศจาก API (ในตัวอย่างจะใช้ OpenWeatherMap API แบบจำลอง / mock หรือคุณใส่ API จริงได้)
- จัดการสถานะ Loading, Success, Error ผ่าน Bloc
- แสดงข้อมูลสภาพอากาศใน UI

Weather Fetch App with Bloc and API — ตัวอย่างเต็ม

โครงสร้างไฟล์

```
lib/
├── main.dart
├── weather_bloc.dart
├── weather_event.dart
├── weather_state.dart
├── weather_repository.dart
└── models/
    └── weather.dart
```

---

1. models/weather.dart

```
class Weather {
  final String cityName;
  final double temperature;
  final String description;
```

```
  Weather({
    required this.cityName,
    required this.temperature,
    required this.description,
  });

  factory Weather.fromJson(Map<String, dynamic> json) {
    return Weather(
      cityName: json['name'],
      temperature: (json['main']['temp'] as num).toDouble(),
      description: json['weather'][0]['description'],
    );
  }
}
```

---

2. weather_repository.dart

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'models/weather.dart';

class WeatherRepository {
  final String apiKey = 'YOUR_API_KEY'; // ใส่ API Key ของคุณ

  Future<Weather> fetchWeather(String cityName) async {
    final url = Uri.parse(

'https://api.openweathermap.org/data/2.5/weather?q=$cityName&units=metric&appid=$apiKey');

    final response = await http.get(url);

    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      return Weather.fromJson(data);
    } else {
```

```
      throw Exception('Failed to fetch weather');
    }
  }
}
```

---

3. weather_event.dart

```
abstract class WeatherEvent {}

class FetchWeather extends WeatherEvent {
  final String cityName;
  FetchWeather(this.cityName);
}
```

---

4. weather_state.dart

```
import '../models/weather.dart';

abstract class WeatherState {}

class WeatherInitial extends WeatherState {}

class WeatherLoading extends WeatherState {}

class WeatherLoaded extends WeatherState {
  final Weather weather;
  WeatherLoaded(this.weather);
}

class WeatherError extends WeatherState {
  final String message;
  WeatherError(this.message);
}
```

---

5. weather_bloc.dart

```
import 'package:bloc/bloc.dart';
```

```
import 'weather_event.dart';
import 'weather_state.dart';
import 'weather_repository.dart';

class WeatherBloc extends Bloc<WeatherEvent, WeatherState> {
  final WeatherRepository repository;

  WeatherBloc(this.repository) : super(WeatherInitial()) {
    on<FetchWeather>((event, emit) async {
      emit(WeatherLoading());
      try {
        final weather = await repository.fetchWeather(event.cityName);
        emit(WeatherLoaded(weather));
      } catch (e) {
        emit(WeatherError('Could not fetch weather. Please try again.'));
      }
    });
  }
}
```

---

6. main.dart

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'weather_bloc.dart';
import 'weather_event.dart';
import 'weather_state.dart';
import 'weather_repository.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Weather Bloc App',
      home: BlocProvider(
        create: (_) => WeatherBloc(WeatherRepository()),
        child: const WeatherPage(),
      ),
    );
  }
}

class WeatherPage extends StatefulWidget {
  const WeatherPage({super.key});
  @override
  State<WeatherPage> createState() => _WeatherPageState();
}

class _WeatherPageState extends State<WeatherPage> {
  final TextEditingController _cityController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    final weatherBloc = context.read<WeatherBloc>();

    return Scaffold(
      appBar: AppBar(title: const Text('Weather Fetch with Bloc')),
      body: Padding(
        padding: const EdgeInsets.all(16),
        child: Column(
          children: [
            TextField(
              controller: _cityController,
              decoration: const InputDecoration(
```

```
      labelText: 'Enter city name',
      border: OutlineInputBorder(),
    ),
  ),
  const SizedBox(height: 16),
  ElevatedButton(
    onPressed: () {
      final city = _cityController.text.trim();
      if (city.isNotEmpty) {
        weatherBloc.add(FetchWeather(city));
      }
    },
    child: const Text('Get Weather'),
  ),
  const SizedBox(height: 24),
  Expanded(
    child: BlocBuilder<WeatherBloc, WeatherState>(
      builder: (context, state) {
        if (state is WeatherInitial) {
          return const Center(child: Text('Enter a city to get weather'));
        } else if (state is WeatherLoading) {
          return const Center(child: CircularProgressIndicator());
        } else if (state is WeatherLoaded) {
          final weather = state.weather;
          return Center(
            child: Column(
              mainAxisSize: MainAxisSize.min,
              children: [
                Text(
                  weather.cityName,
                  style: const TextStyle(fontSize: 32, fontWeight: FontWeight.bold),
                ),
                const SizedBox(height: 8),
                Text(
```

```
                  '${weather.temperature.toStringAsFixed(1)} °C',
                  style: const TextStyle(fontSize: 24),
                ),
                const SizedBox(height: 8),
                Text(
                  weather.description,
                  style: const TextStyle(fontSize: 20, fontStyle: FontStyle.italic),
                ),
              ],
            ),
          );
        } else if (state is WeatherError) {
          return Center(child: Text(state.message, style: const TextStyle(color:
Colors.red)));
        }
        return const SizedBox.shrink();
      },
    ),
  ),
 ],
 ),
 ),
 );
 }
}
```

วิธีรัน

1. ใส่ API Key จาก OpenWeatherMap ใน weather_repository.dart
2. รัน flutter pub add flutter_bloc http
3. รันแอป แล้วกรอกชื่อเมือง กดปุ่ม Get Weather
4. จะเห็นข้อมูลอุณหภูมิและสถานะอากาศ หรือข้อความ error ถ้าดึงข้อมูลไม่สำเร็จ

การใช้ Streams ในการจัดการ State — อธิบายละเอียด

1. Streams คืออะไร?

- **Stream** คือข้อมูลชุดหนึ่งที่ถูกส่งออกมาเป็นลำดับต่อเนื่อง (sequence) ในช่วงเวลาหนึ่ง
- เปรียบเสมือนท่อส่งข้อมูลที่ส่ง event หรือ data ทีละชิ้น
- สามารถฟัง (listen) stream เพื่อรับข้อมูลใหม่ ๆ ที่เกิดขึ้นแบบ asynchronous ได้

2. Streams vs Future

- **Future** คือข้อมูลที่ส่งกลับมาแค่ครั้งเดียวในอนาคต (เช่น โหลดข้อมูลเสร็จ)
- **Stream** ส่งข้อมูลหลายครั้งในช่วงเวลาหนึ่ง เช่น การรับข้อความแชท, การเปลี่ยนแปลงสถานะของแอป ฯลฯ

3. Streams กับการจัดการ State ใน Flutter

- **Streams ช่วยทำให้ UI "ฟัง" การเปลี่ยนแปลงข้อมูล (State) ได้แบบ realtime**
- เมื่อข้อมูลใน Business Logic เปลี่ยนแปลง จะถูกส่งออกมาใน Stream
- UI ใช้ Widget อย่างเช่น StreamBuilder หรือในกรณีของ Bloc ใช้ BlocBuilder ซึ่งฟัง Stream ของ State และสร้าง UI ใหม่ทุกครั้งที่ State เปลี่ยน

4. ตัวอย่างแนวคิดการใช้ Streams

```
import 'dart:async';

class Counter {
  int _count = 0;

  final _controller = StreamController<int>();

  Stream<int> get countStream => _controller.stream;

  void increment() {
    _count++;
    _controller.sink.add(_count);
  }

  void dispose() {
    _controller.close();
  }
}
```

- ตัวอย่างนี้ สร้างคลาส Counter ที่เก็บตัวแปรนับและมี StreamController ส่งข้อมูล count ใหม่เมื่อมีการเพิ่มค่า
- UI สามารถฟัง countStream และอัพเดตตัวเลขเมื่อมีข้อมูลใหม่เข้ามา

5. Bloc กับ Streams

- Bloc สร้าง **Stream** สำหรับส่งข้อมูล State ออกมา
- รับ **Event** เข้ามา และประมวลผลเพื่อเปลี่ยน State
- ส่ง State ใหม่ผ่าน Stream ไปยัง UI
- UI ฟัง Stream นี้เพื่อรีเฟรชหน้าจออัตโนมัติ

6. สรุปข้อดีของการใช้ Streams

- **ตอบสนองแบบเรียลไทม์**: UI อัพเดตตามข้อมูลที่เปลี่ยนทันที
- **แยกส่วนชัดเจน**: Business Logic ไม่ผสมกับ UI
- **รองรับข้อมูลหลายชุด**: เหมาะกับข้อมูลที่เปลี่ยนแปลงหลายครั้ง
- **รองรับการทำงานแบบ Asynchronous ได้ดี**

ชุดตัวอย่างพื้นฐาน (3 โปรแกรม) — การใช้ Streams ในการจัดการ State

---

1. ตัวอย่าง Counter ใช้ StreamController

โครงสร้างไฟล์

```
lib/
  └── main.dart
```

main.dart

```
import 'dart:async';

import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class CounterBloc {
  int _count = 0;

  final _controller = StreamController<int>();

  Stream<int> get stream => _controller.stream;

  void increment() {
```

```
    _count++;
    _controller.sink.add(_count);
  }

  void dispose() {
    _controller.close();
  }
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Stream Counter',
      home: CounterPage(),
    );
  }
}

class CounterPage extends StatefulWidget {
  @override
  State<CounterPage> createState() => _CounterPageState();
}

class _CounterPageState extends State<CounterPage> {
  late CounterBloc bloc;

  @override
  void initState() {
    super.initState();
    bloc = CounterBloc();
  }
```

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Stream Counter')),
      body: Center(
        child: StreamBuilder<int>(
          stream: bloc.stream,
          initialData: 0,
          builder: (context, snapshot) {
            final count = snapshot.data ?? 0;
            return Text('Count: $count', style: const TextStyle(fontSize: 48));
          },
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: bloc.increment,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

คำอธิบาย

- สร้าง CounterBloc ที่ใช้ StreamController<int> ส่งค่าตัวเลข
- เมื่อกดปุ่ม increment จะเพิ่มค่าแล้วส่งค่าใหม่ผ่าน stream
- StreamBuilder ฟัง stream และอัพเดต UI ทุกครั้งที่ค่ามาใหม่

---

2. Simple Toggle Switch with Stream

main.dart

```
import 'dart:async';

import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class ToggleBloc {
  bool _isOn = false;
  final _controller = StreamController<bool>();

  Stream<bool> get stream => _controller.stream;

  void toggle() {
    _isOn = !_isOn;
    _controller.sink.add(_isOn);
  }

  void dispose() {
    _controller.close();
  }
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: TogglePage(),
    );
  }
}
```

```
class TogglePage extends StatefulWidget {
  @override
  State<TogglePage> createState() => _TogglePageState();
}

class _TogglePageState extends State<TogglePage> {
  late ToggleBloc bloc;

  @override
  void initState() {
    super.initState();
    bloc = ToggleBloc();
  }

  @override
  void dispose() {
    bloc.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Toggle Stream')),
      body: Center(
        child: StreamBuilder<bool>(
          stream: bloc.stream,
          initialData: false,
          builder: (context, snapshot) {
            final isOn = snapshot.data ?? false;
            return Switch(
              value: isOn,
              onChanged: (_) => bloc.toggle(),
            );
```