Bloc Documentation

Contents

I) Giới thiệu

1) Tổng quan:

- Bloc: thư viện core của bloc
- Flutter_bloc: là flutter widget mạnh mẽ được xây dựng để làm việc với bloc với nhiệm vụ xây dựng ứng dụng mobile nhanh và phản hồi nhanh
- Angular_bloc: là flutter widget mạnh mẽ được xây dựng để làm việc với bloc với nhiệm vụ xây dựng ứng dụng web nhanh và phản hồi nhanh
- Hydrated_bloc: một extension thư viện quản lý state của bloc, tự động hoá các trạng thái tồn tại và phục hồi của state
- Replay_bloc: extension thư viện quản lý state của bloc, them hỗ trợ cho việc undo và redo.

2) Cài đặt:

- Thêm bloc vào file pubspec.yaml (Dart):

```
- dependencies:
-   bloc: ^8.0.0
```

- Thêm flutter_bloc vào file pubspec.yaml (Flutter):

```
- dependencies:
-   bloc: ^8.0.0
```

- Thêm angular_bloc vào file pubspec.yaml (AngularDart):

```
- dependencies:
-   bloc: ^8.0.0
```

- Tiếp theo, cài đặt bloc
- Dart/Angular: chạy lệnh **pub get**
- Flutter: chạy lệnh **flutter packages get**

3) Import:

- Dart:

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

- Flutter:

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

- AngularDart:

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

4) Ứng dụng của BloC:

- Dễ dàng để tách bussiness logic và presentation, khiến cho code chạy nhanh hơn, dễ dàng để test và có khả năng tái sử dụng. Khi các ứng dụng thành product thì khả năng quản lý state trở nên quan trọng hơn.

- BloC được tạo nên để đáp ứng các nhu cầu:

- Biết úng dụng đang ở trạng thái nào vào một thời điểm nào đó
- Dễ dàng cho tất cả các test case để đảm bảo ứng dụng chạy chính xác
- Thu tập từng tương tác của người dùng trong ứng dụng để có thể quyết định dựa trên dữ liệu
- Hoạt động hiệu quả, các component tái sử dụng lẫn trong và ngoài các ứng dụng khác.
- Có nhiều lập trình viên làm việc trên một code base và phải theo một khuôn mẫu và quy ước chung
- Phát triển ứng dụng nhanh và phản hồi nhanh

- BloC được thiết kế với ba tiêu chí:
- Đơn giản: Dễ hiểu và dê sử dụng với mọi lập trình viên ở các level khác nhau
- Mạnh mẽ: Hỗ trợ tạo ra các ứng dụng phức tạp bằng cách chia chúng thành các component nhỏ hơn
- Dễ để kiểm thử: Dễ để kiểm thử một tính năng nào đó của ứng dụng

II) Package bloc (Dart):

(updating)

III) Package flutter_bloc (Flutter):

1) BlocBuilder

- BlocBuilder là một Flutter Widget yêu cầu một Bloc và một Builder. BlocBuilder sẽ có nhiệm vụ xử lý build các widget khi respone với một state mới. BlocBuilder tương tự như StreamBuilder nhưng là một API đơn giản hơn để giảm thiểu code mẫu cần có. Builder sẽ được gọi nhiều lần và nên là một pure function trả về môt widget để response cho state.
- Dùng BlocListener để response khi thay đổi state như hiển thị một dialog, navigation, …
- Nếu tham số bloc bị bỏ qua, BlocBuilder sẽ tự động tra cứu sử dụng BlocProvider và BuildContext hiện tại

```
- BlocBuilder<BlocA, BlocAState>(
-   builder: (context, state) {
-     // return widget here based on BlocA's state
-   }
- )
-
```

- Chỉ khai báo bloc nếu muốn cấp một bloc giưới hạn với một widget nhất định và không thể truy cập qua widget cha BlocProvider và BuildContext hiện tại

```
- BlocBuilder<BlocA, BlocAState>(
-   bloc: blocA, // provide the local bloc instance
-   builder: (context, state) {
-     // return widget here based on BlocA's state
-   }
- )
-
```

- Để kiểm soát chi tiết hơn khi builder được gọi, một hàm buildWhen sẽ được cấp. buildWhen lấy trạng thái của bloc trước và trạng thái của bloc hiện tại và trả về một giá trị Boolean. Nếu

buildWhen trả về true, builder sẽ được gọi với state và widget sẽ được build lại. Nếu buildWhen trả về false, builder không thể gọi state và widget sẽ không được build lại.

```
- BlocBuilder<BlocA, BlocAState>(
-   buildWhen: (previousState, state) {
-     // return true/false to determine whether or not
-     // to rebuild the widget with state
-   },
-   builder: (context, state) {
-     // return widget here based on BlocA's state
-   }
- )
-
```

2) BlocSelector

- BlocSelector và một Flutter Widget tương tự như BlocBuilder nhưng cho phép lập trình viên lọc các cập nhật bằng cách chọn ra một giá trị mới dự trên trạng thái bloc hiện tại. Giá trị được chọn phải là một giá trị bất biến để BlocSelector có thể quyết định chính xác khi nào builder sẽ được gọi lần tiếp theo.
- Trong hầu hết các trường hợp, BlocProvider nên được sử dụng để tạo ra các bloc mới để sẽ có thể khả dụng cho các wdiget con. Trong trường hợp này, BlocProvider có nhiệm vụ tạo bloc, đồng thời cũng tự động xử lý việc đóng bloc.

```
- BlocProvider(
-   create: (BuildContext context) => BlocA(),
-   child: ChildA(),
- );
-
```

- Mặc định BlocProvider sẽ khởi tạo bloc, create sẽ được thực thi khi bloc được tìm kiếm qua BlocProvider.of<BlocA>(context)
- Để khiến create thựuc thi ngay lập tức, lazy có thể set thành false

```
- BlocProvider(
-   lazy: false,
-   create: (BuildContext context) => BlocA(),
-   child: ChildA(),
- );
-
```

3) BlocProvider

BlocProvider là một Flutter widget cung cấp một bloc cho các phần tử con của nó thông qua BlocProvider.of<T>(context). Sử dụng như một widget tiêm phụ thuộc (DI) để một phiên bản duy nhất của một bloc có thể được cung cấp cho nhiều tiện ích trong một cây con. Trong hầu hết các trường hợp, BlocProvider nên được sử dụng để tạo các bloc mới sẽ được cung cấp cho phần còn lại của cây con. Trong trường hợp này, do BlocProvider chịu trách nhiệm tạo bloc nên nó sẽ tự động xử lý việc đóng bloc.

```
BlocProvider(
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

Theo mặc định, BlocProvider sẽ tạo bloc một cách lười biếng, nghĩa là quá trình tạo sẽ được thực thi khi bloc được tra cứu qua BlocProvider.of<BlocA>(context). Để ghi đè hành vi này và buộc tạo phải chạy ngay lập tức, lazy có thể được đặt thành false.

```
BlocProvider(
  lazy: false,
  create: (BuildContext context) => BlocA(),
  child: ChildA(),
);
```

Trong một số trường hợp, BlocProvider có thể được sử dụng để cung cấp một bloc hiện có cho một phần mới của cây widget. Điều này sẽ được sử dụng phổ biến nhất khi một bloc hiện có cần được cung cấp cho một route mới. Trong trường hợp này, BlocProvider sẽ không tự động đóng bloc vì nó không tạo ra nó.

```
BlocProvider.value(
  value: BlocProvider.of<BlocA>(context),
  child: ScreenA(),
);
```

sau đó từ ChildA hoặc ScreenA, ta có thể truy xuất BlocA bằng:

```
// with extensions
context.read<BlocA>();

// without extensions
BlocProvider.of<BlocA>(context)
```

4) MultiBlocProvider

MultiBlocProvider là một Flutter widget hợp nhất nhiều BlocProvider thành một. MultiBlocProvider cải thiện khả năng đọc và loại bỏ nhu cầu lồng nhiều BlocProvider. Bằng cách sử dụng MultiBlocProvider, chúng ta có thể đi từ:

```
BlocProvider<BlocA>(
  create: (BuildContext context) => BlocA(),
  child: BlocProvider<BlocB>(
    create: (BuildContext context) => BlocB(),
    child: BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
      child: ChildA(),
    )
  )
```

```
)
```

Đến:

```
MultiBlocProvider(
  providers: [
    BlocProvider<BlocA>(
      create: (BuildContext context) => BlocA(),
    ),
    BlocProvider<BlocB>(
      create: (BuildContext context) => BlocB(),
    ),
    BlocProvider<BlocC>(
      create: (BuildContext context) => BlocC(),
    ),
  ],
  child: ChildA(),
)
```

5) BlocListener

BlocListener là một widget Flutter nhận BlocWidgetListener và một Bloc tùy chọn và gọi listener để đáp ứng với các thay đổi state trong bloc. Nó nên được sử dụng cho chức năng cần xảy ra một lần cho mỗi thay đổi trạng thái, chẳng hạn như điều hướng, hiển thị SnackBar, hiển thị Dialog (Hộp thoại), v.v... listener chỉ được gọi một lần cho mỗi thay đổi state (trạng thái) (KHÔNG bao gồm state ban đầu) không giống như builder trong BlocBuilder là hàm void. Nếu tham số khối bị bỏ qua, BlocListener sẽ tự động thực hiện tra cứu bằng BlocProvider và BuildContext hiện tại.

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

Chỉ chỉ định khối nếu bạn muốn cung cấp một khối không thể truy cập được thông qua BlocProvider và BuildContext hiện tại.

```
BlocListener<BlocA, BlocAState>(
  bloc: blocA,
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container()
)
```

Để kiểm soát chi tiết khi listener được gọi là tùy chọn listenWhen có thể được cung cấp. listenWhen lấy trạng thái khối trước đó và trạng thái khối hiện tại và trả về một giá trị boolean. Nếu listenWhen trả về true, listener sẽ được gọi bằng state. Nếu listenWjem trả về false, listener sẽ không được gọi với state.

```
BlocListener<BlocA, BlocAState>(
  listenWhen: (previousState, state) {
    // return true/false to determine whether or not
    // to call listener with state
  },
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  child: Container(),
)
```

6) MultiBlocListener

MultiBlocListener là một Flutter widget hợp nhất nhiều widget BlocListener thành một. MultiBlocListener cải thiện khả năng đọc và loại bỏ nhu cầu lồng nhiều BlocListener. Bằng cách sử dụng MultiBlocListener, chúng ta có thể đi từ:

```
BlocListener<BlocA, BlocAState>(
  listener: (context, state) {},
  child: BlocListener<BlocB, BlocBState>(
    listener: (context, state) {},
    child: BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
      child: ChildA(),
    ),
  ),
)
```

Thành

```
MultiBlocListener(
  listeners: [
    BlocListener<BlocA, BlocAState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocB, BlocBState>(
      listener: (context, state) {},
    ),
    BlocListener<BlocC, BlocCState>(
      listener: (context, state) {},
    ),
  ],
  child: ChildA(),
)
```

7) BlocConsumer

BlocConsumer hiển thị builer và listener để phản ứng với các state mới. BlocConsumer tương tự như BlocListener và BlocBuilder lồng nhau nhưng giảm số lượng bản soạn sẵn cần thiết. BlocConsumer chỉ nên được sử dụng khi cần xây dựng lại giao diện người dùng và thực hiện các phản ứng khác đối với các thay đổi trạng thái trong bloc. BlocConsumer nhận BlocWidgetBuilder và BlocWidgetListener bắt buộc và một bloc tùy chọn, BlocBuilderCondition và BlocListenerCondition. Nếu tham số bloc bị bỏ qua, BlocConsumer sẽ tự động thực hiện tra cứu bằng BlocProvider và BuildContext hiện tại.

```
BlocConsumer<BlocA, BlocAState>(
  listener: (context, state) {
    // do stuff here based on BlocA's state
  },
  builder: (context, state) {
    // return widget here based on BlocA's state
  }
)
```

Có thể triển khai tùy chọn listenWhen và buildWhen để kiểm soát chi tiết hơn khi listener và builder được gọi. listenWhen và buildWhen sẽ được gọi trên mỗi thay đổi state bloc. Mỗi cái lấy state trước đó và state hiện tại và phải trả về một bool xác định xem builder và listener có được gọi hay không. State trước đó sẽ được khởi tạo thành state của bloc khi BlocConsumer được khởi tạo. listenWhen và buildWhen là tùy chọn và nếu chúng không được triển khai, mặc định là true.

```
            BlocConsumer<BlocA, BlocAState>(
           listenWhen: (previous, current) {
                // return true/false to determine whether or not
                // to invoke listener with state
           },
              listener: (context, state) {
                // do stuff here based on BlocA's state
              },
              buildWhen: (previous, current) {
                // return true/false to determine whether or not
                // to rebuild the widget with state
              },
              builder: (context, state) {
                // return widget here based on BlocA's state
              }
            )
```

8) RepositoryProvider

RepositoryProvider là một Flutter widget cung cấp repository cho các phần tử con của nó thông qua RepositoryProvider.of<T>(context). Nó được sử dụng như một widget nội xạ phụ thuộc (dependency injection - DI) để một phiên bản duy nhất của repository có thể được cung cấp cho nhiều widget trong một cây con. BlocProvider nên được sử dụng để cung cấp các bloc trong khi RepositoryProvider chỉ nên được sử dụng cho các repository

```
RepositoryProvider(
  create: (context) => RepositoryA(),
  child: ChildA(),
);
```

9) MultiRepositoryProvider

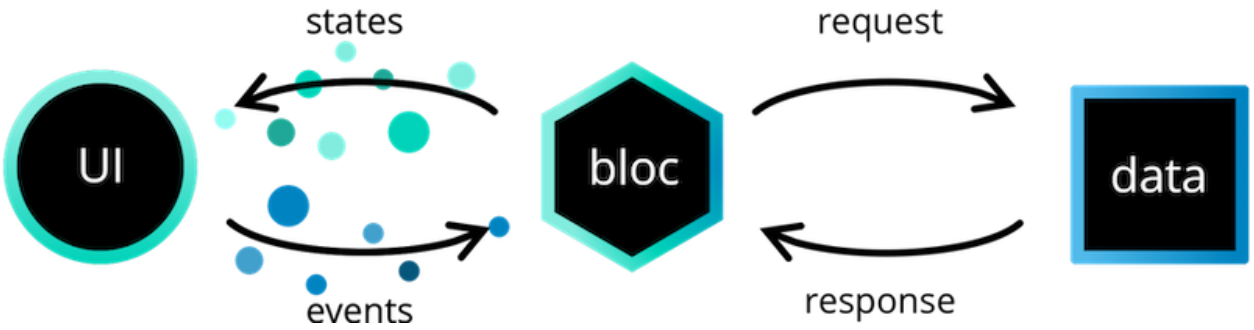
MultiRepositoryProvider là một Flutter widget hợp nhất nhiều wdiget RepositoryProvider thành một. MultiRepositoryProvider cải thiện khả năng đọc và loại bỏ nhu cầu lồng nhiều RepositoryProvider. Bằng cách sử dụng MultiRepositoryProvider, chúng ta có thể đi từ:

```
RepositoryProvider<RepositoryA>(
  create: (context) => RepositoryA(),
  child: RepositoryProvider<RepositoryB>(
    create: (context) => RepositoryB(),
    child: RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
      child: ChildA(),
    )
  )
)
```

Thành

```
MultiRepositoryProvider(
  providers: [
    RepositoryProvider<RepositoryA>(
      create: (context) => RepositoryA(),
    ),
    RepositoryProvider<RepositoryB>(
      create: (context) => RepositoryB(),
    ),
    RepositoryProvider<RepositoryC>(
      create: (context) => RepositoryC(),
    ),
  ],
  child: ChildA(),
)
```

IV) Kiến trúc



Việc sử dụng thư viện khối cho phép chúng ta tách ứng dụng của mình thành ba lớp:

- Presentation
- Business Logic
- Data:

  + Repository

  + Data Provider

Chúng ta sẽ bắt đầu ở lớp cấp thấp nhất (xa giao diện người dùng nhất) và tiến tới lớp trình bày (presentation layer)

1) Data Layer

Data Layer có nhiệm vụ truy xuất và thao tác các dữ liệu từ một hay nhiều nguồn

Data Layer có thể chia ra làm hai phần:

- Repository
- Data Provider
- Đây là tầng thấp nhất của ứng dụng và tương tác với database, các yêu cầu của network, và các nguồn dữ liệu bất đồng bộ

2) Data provider

Nhiệm vụ của Data Provider là cung cấp dữ liệu thô. Data Provider nên chung chung và đa dụng

Nhà cung cấp dữ liệu thường sẽ hiển thị các API đơn giản để thực hiện các thao tác CRUD. Chúng ta có thể có phương thức createData, readData, updateData và deleteData như một phần của Data Layer.

```
class DataProvider {
    Future<RawData> readData() async {
        // Read from DB or make network request etc...
    }
}
```

3) Repository

Nhiệm vụ của Repository là lớp bao bọc xung quanh một hoặc nhiều data provider mà Bloc Layer giao tiếp.

```
class Repository {
    final DataProviderA dataProviderA;
    final DataProviderB dataProviderB;
```

```
    Future<Data> getAllDataThatMeetsRequirements() async {
        final RawDataA dataSetA = await
dataProviderA.readData();
        final RawDataB dataSetB = await
dataProviderB.readData();

        final Data filteredData = _filterData(dataSetA,
dataSetB);
        return filteredData;
    }
}
```

Lớp Repository Layer có thể tương tác với nhiều data provider và thực hiện chuyển hoá dữ liệu trước khi gửi kết quả đến business logic layer

4) Business logic layer

Nhiệm vụ của business logic layer là trả lời input từ presentation layer với state mới. Layer này phụ thuộc vào một hay nhiều repository để lấy data cần thiết giúp xây dựng state của ứng dụng.

Business Logic layer như một cầu nối giữa lớp presentation layer (interface) và data layer. Business logic layer được thông báo các sự kiện, hành động từ presentation layer và tương tác với repository để build một state mới cho lớp presentation

```
class BusinessLogicComponent extends Bloc<MyEvent, MyState> {
    BusinessLogicComponent(this.repository) {
        on<AppStarted>((event, emit) {
            try {
                final data = await
repository.getAllDataThatMeetsRequirements();
                emit(Success(data));
            } catch (error) {
                emit(Failure(error));
            }
        });
    }

    final Repository repository;
}
```

5) Bloc to Bloc Communication

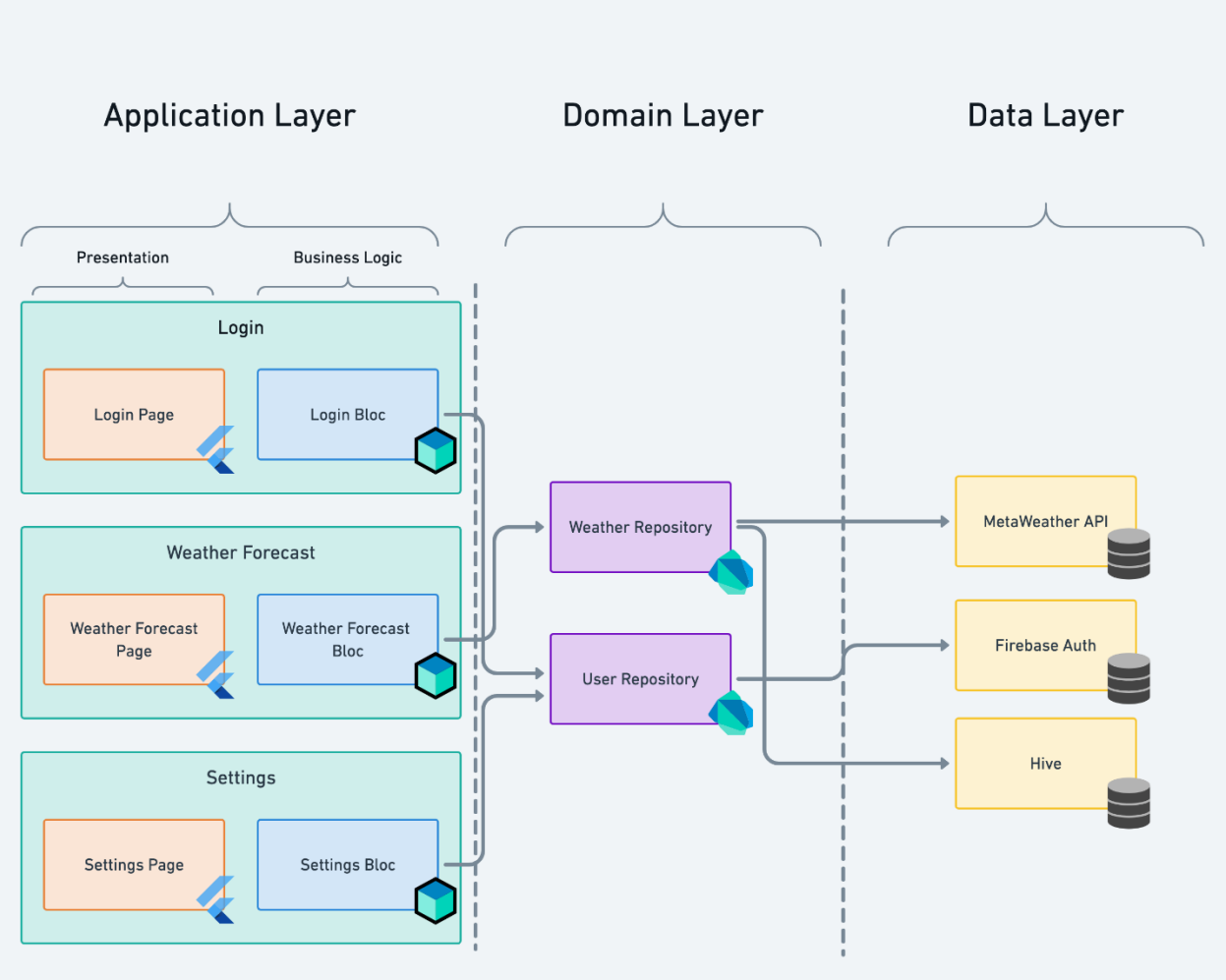
Bởi vì các bloc hiển thị các stream (luồng), vậy không nên việc tạo một bloc lắng nghe một bloc khác. Có những lựa chọn thay thế tốt hơn.

```
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();
  }
}
```

Mặc dù đoạn code trên không có lỗi, nhưng nó có một vấn đề lớn hơn là tạo ra sự phụ thuộc giữa hai khối. Bằng mọi giá nên tránh sự phụ thuộc giữa hai thực thể trong cùng một lớp kiến trúc, vì nó tạo ra sự liên kết chặt chẽ khó duy trì. Vì các bloc nằm trong lớp business logic architectural layer nên không bloc nào được biết về bất kỳ bloc nào khác



Một bloc chỉ nên nhận thông tin thông qua các event và từ các repository được đưa vào (nghĩa là các repository được cung cấp cho bloc trong hàm tạo). Giả dụ đang ở trong tình huống mà một bloc cần phản hồi lại một khối khác, còn các hai lựa chọn khác: Đẩy vấn đề lên một lớp (vào lớp presentation) hoặc xuống một lớp (vào lớp domain).
Kết nối các bloc thông qua Presentation:
Sử dụng BlocListener để nghe một bloc và thêm một event vào một bloc khác bất cứ khi nào bloc đầu tiên thay đổi.

```
class MyWidget extends StatelessWidget {
  const MyWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocListener<FirstBloc, FirstState>(
      listener: (context, state) {
        // When the first bloc's state changes, this will be
called.
        //
        // Now we can add an event to the second bloc without
it having
```

```
        // to know about the first bloc.

BlocProvider.of<SecondBloc>(context).add(SecondEvent());
      },
      child: TextButton(
        child: const Text('Hello'),
        onPressed: () {

BlocProvider.of<FirstBloc>(context).add(FirstEvent());
        },
      ),
    );
  }
}
```

Đoạn code trên ngăn SecondBloc cần biết về FirstBloc, khuyến khích khớp nối lỏng lẻo. Ứng dụng flutter_weather sử dụng kỹ thuật này để thay đổi chủ đề của ứng dụng dựa trên thông tin thời tiết nhận được. Trong một số trường hợp, bạn có thể không muốn ghép hai bloc trong lớp trình bày. Thay vào đó, việc hai bloc chia sẻ cùng một nguồn dữ liệu và cập nhật bất cứ khi nào dữ liệu thay đổi.

Kết nối Blocs qua Domain:

Hai bloc có thể lắng nghe stream từ repository và cập nhật state của chúng độc lập với nhau bất cứ khi nào dữ liệu repository thay đổi. Việc sử dụng các repository phản ứng để giữ cho state được đồng bộ hóa là phổ biến trong các ứng dụng doanh nghiệp quy mô lớn.

Đầu tiên, tạo hoặc sử dụng repository cung cấp Stream. Ví dụ: repository sau hiển thị stream không bao giờ kết thúc của cùng một số ý tưởng ứng dụng:

```
class AppIdeasRepository {
  int _currentAppIdea = 0;
  final List<String> _ideas = [
    "Future prediction app that rewards you if you predict the
next day's news",
    'Dating app for fish that lets your aquarium occupants find
true love',
    'Social media app that pays you when your data is sold',
    'JavaScript framework gambling app that lets you bet on the
next big thing',
    'Solitaire app that freezes before you can win',
  ];

  Stream<String> productIdeas() async* {
    while (true) {
      yield _ideas[_currentAppIdea++ % _ideas.length];
      await Future<void>.delayed(const Duration(minutes: 1));
    }
  }
```

```
}
```

Cùng một repository có thể được đưa vào từng bloc cần phản ứng với các ý tưởng ứng dụng mới. Bên dưới AppIdeaRankingBloc mang lại state cho mỗi ý tưởng ứng dụng đến từ repository ở trên:

```
class AppIdeaRankingBloc
    extends Bloc<AppIdeaRankingEvent, AppIdeaRankingState> {
  AppIdeaRankingBloc({required AppIdeasRepository
appIdeasRepo})
      : _appIdeasRepo = appIdeasRepo,
        super(AppIdeaInitialRankingState()) {
    on<AppIdeaStartRankingEvent>((event, emit) async {
      // When we are told to start ranking app ideas, we will
listen to the
      // stream of app ideas and emit a state for each one.
      await emit.forEach(
        _appIdeasRepo.productIdeas(),
        onData: (String idea) => AppIdeaRankingIdeaState(idea:
idea),
      );
    });
  }

  final AppIdeasRepository _appIdeasRepo;
}
```

6) Presentation Layer

Trách nhiệm của Presentation Layer là tìm ra cách hiển thị chính nó dựa trên một hoặc nhiều trạng thái bloc. Ngoài ra, nó sẽ xử lý các event n vòng đời ứng dụng và input của người dùng.

Hầu hết các luồng ứng dụng sẽ bắt đầu bằng sự kiện AppStart kích hoạt ứng dụng tìm nạp một số dữ liệu để hiển thị cho người dùng. Trong trường hợp này, presentation sẽ thêm một sự kiện AppStart. Ngoài ra, presentation layer sẽ phải tìm ra nội dung sẽ hiển thị trên màn hình dựa trên state từ bloc layer.

```
class PresentationComponent {
    final Bloc bloc;

    PresentationComponent() {
        bloc.add(AppStarted());
    }

    build() {
        // render UI based on bloc state
    }
}
```

V) Kiểm thử:

Bloc được thiết kế để cực kỳ dễ kiểm tra. Viết các bài kiểm tra cho CounterBloc mà chúng đã được tạo trong Core Concepts. Tóm lại, việc triển khai CounterBloc trông như sau:

```
abstract class CounterEvent {}

class CounterIncrementPressed extends CounterEvent {}

class CounterDecrementPressed extends CounterEvent {}

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {
    on<CounterIncrementPressed>((event, emit) => emit(state + 1));
    on<CounterDecrementPressed>((event, emit) => emit(state - 1));
  }
}
```

Thêm test và bloc_test vào pubspec.yaml

```
dev_dependencies:
  test: ^1.16.0
  bloc_test: ^9.0.0
```

Import thư viện vào counter_bloc_test.dart

```
import 'package:test/test.dart';
import 'package:bloc_test/bloc_test.dart';
```

Tạo main

```
void main() {
    group('CounterBloc', () {

    });
}
```

Tạo một phiên bản CounterBloc được sử dụng trong tất cả các test

```
group('CounterBloc', () {
    late CounterBloc counterBloc;

    setUp(() {
        counterBloc = CounterBloc();
    });
});
```

Viết từng test

```dart
group('CounterBloc', () {
    late CounterBloc counterBloc;

    setUp(() {
        counterBloc = CounterBloc();
    });

    test('initial state is 0', () {
        expect(counterBloc.state, 0);
    });
});
```

Viết test phức tạp hơn

```dart
blocTest(
    'emits [1] when CounterIncrementPressed is added',
    build: () => counterBloc,
    act: (bloc) => bloc.add(CounterIncrementPressed()),
    expect: () => [1],
);

blocTest(
    'emits [-1] when CounterDecrementPressed is added',
    build: () => counterBloc,
    act: (bloc) => bloc.add(CounterDecrementPressed()),
    expect: () => [-1],
);
```

VI) Quy ước đặt tên:

Event Convention:

BlocSubject + Noun (optional) + Verb (event)

Ví dụ:

```dart
abstract class CounterEvent {}
class CounterStarted extends CounterEvent {}
class CounterIncrementPressed extends CounterEvent {}
class CounterDecrementPressed extends CounterEvent {}
class CounterIncrementRetried extends CounterEvent {}
```

State Convention:

BlocSubject + Verb (action) + State

Single class:

BlocSubject + State

Subclass

```
abstract class CounterState {}
class CounterInitial extends CounterState {}
class CounterLoadInProgress extends CounterState {}
class CounterLoadSuccess extends CounterState {}
class CounterLoadFailure extends CounterState {}
```

Single class

```
enum CounterStatus { initial, loading, success, failure }
class CounterState {
  const CounterState({this.status = CounterStatus.initial});
  final CounterStatus status;
}
```

VII) Demo