Flutter'da Durum Yönetiminde Ustalaşmak: BLoC Modeli

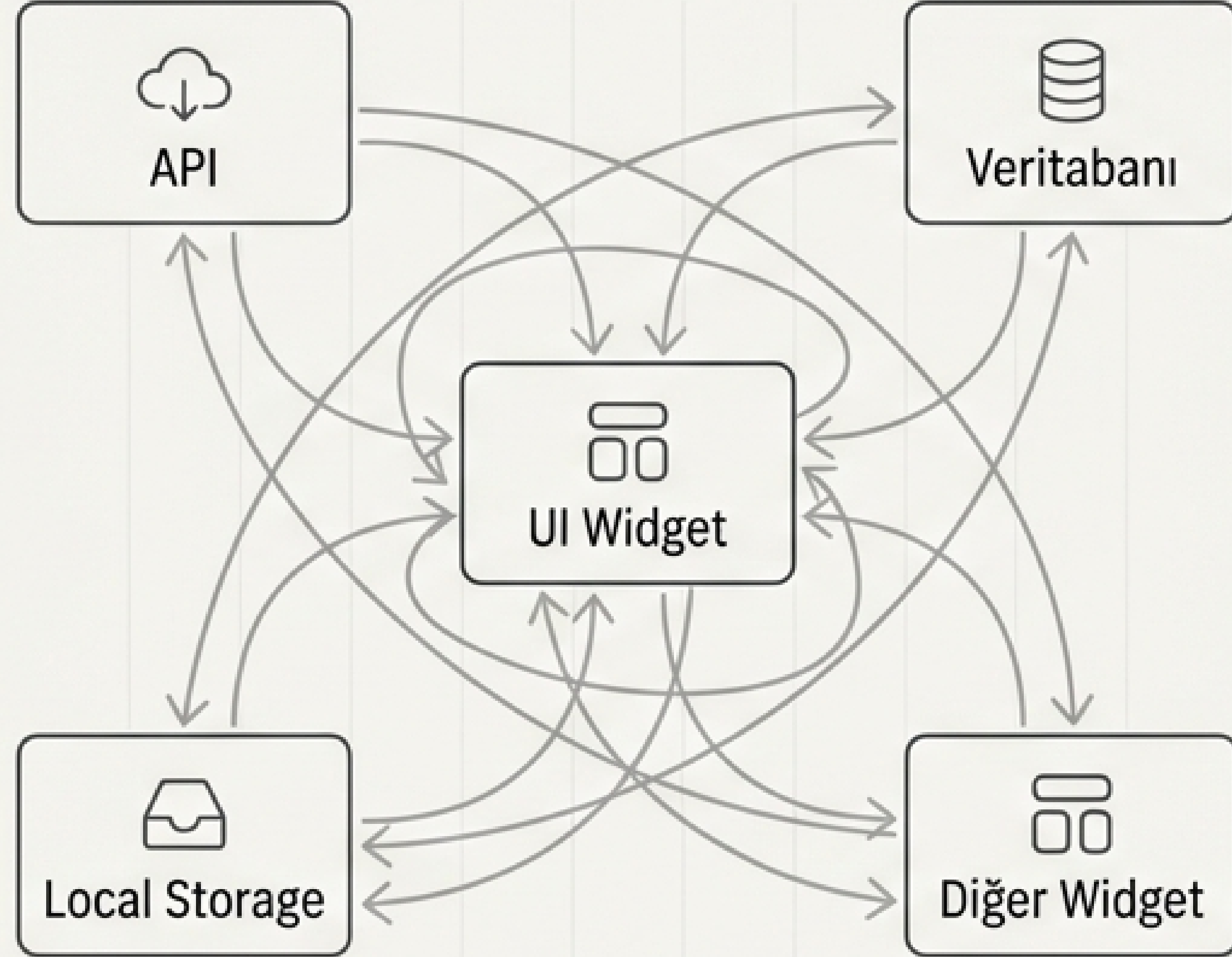
Olaylardan Durumlara: Ölçeklenebilir ve Test Edilebilir Uygulamalar İçin Kapsamlı Bir Rehber

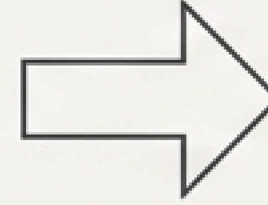
NotebookLM

setState'in Ötesinde: Neden BLoC?

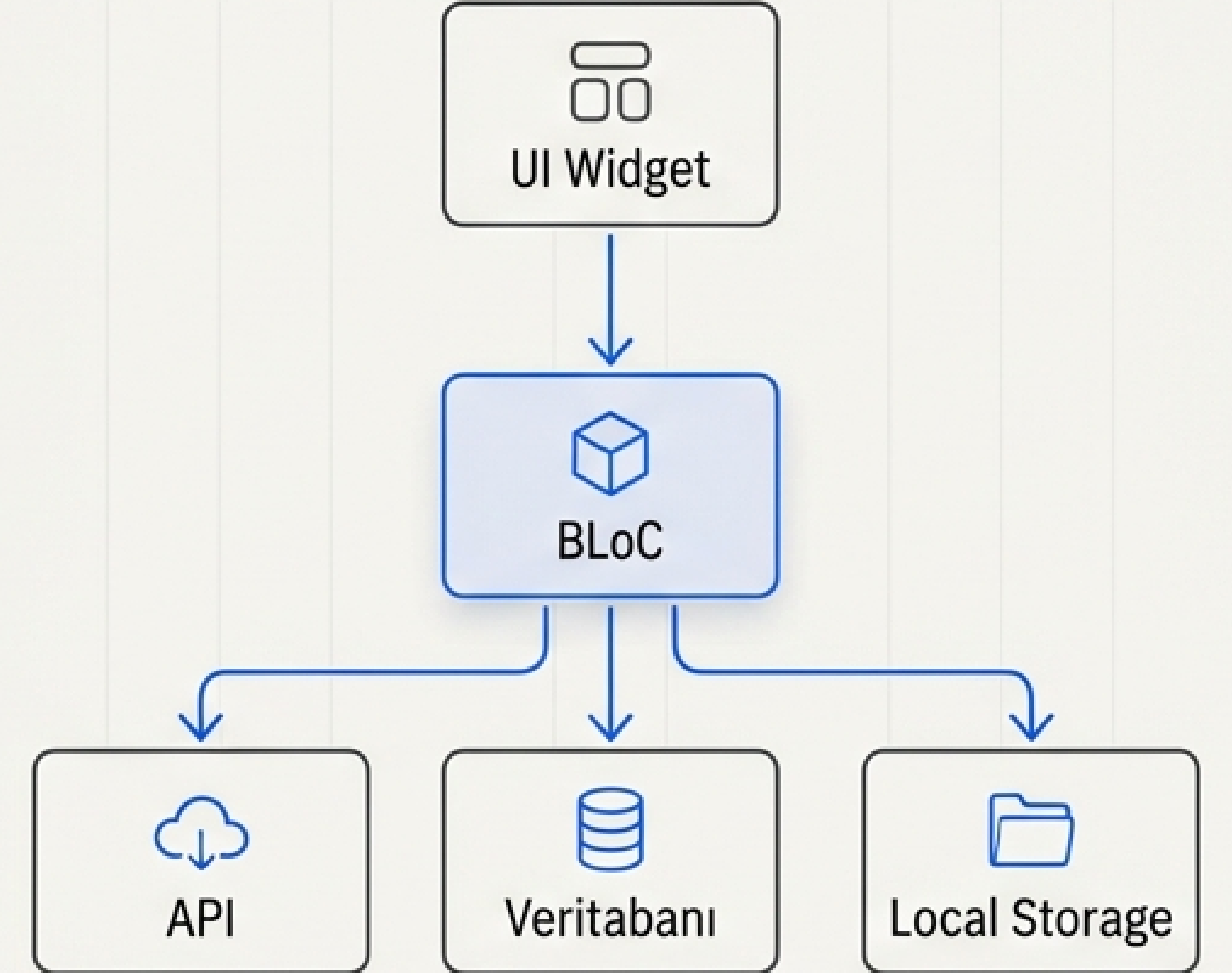
Uygulama büyüdükçe, arayüz (UI) mantığı ile iş (business) mantığını birbirinden ayırmak kritik hale gelir. BLoC, bu ayrımı zorunlu kılarak daha temiz, test edilebilir ve sürdürülebilir bir kod yapısı sunar. Bu, Tek Sorumluluk Prensibi (Single Responsibility Principle) lehine atılmış önemli bir adımdır.

Spagetti Kod





Sorumlulukların Ayrılması



NotebookLM

BLoC’un Kalbi: Olaydan Duruma Dönüşüm

BLoC’un tek bir görevi vardır: Gelen olay akışını işleyip yeni durumlara dönüştürmek. Arayüz, bu durumları dinler ve kendini günceller.

NotebookLM

İlk BLoC'umuz: Sayaç Uygulaması - Mantık Katmanı

İş mantığı, UI'dan tamamen izole edilmiş bir sınıfta yaşar. Bu, kodun test edilmesini ve yönetilmesini kolaylaştırır.

1. Olaylar (Girdiler): BLoC'a ne yapması gerektiğini söyleyen basit, listelenmiş komutlar.

2. BLoC Sınıfı: `CounterEvent` türünde olaylar alıp `int` türünde durumlar üreten bileşenimiz.

3. Başlangıç Durumu: BLoC'un ilk oluşturulduğunda sahip olacağı başlangıç değeri. Bu değer zorunludur.

4. Çekirdek Mantık: İş mantığının yaşadığı yer. Gelen olayın yeni bir duruma dönüştürüldüğü metot.

Not: `state` değişkeni, BLoC'un mevcut durumunu temsil eden bir getter'dır.

counter_bloc.dart

```dart
enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.increment:
        yield ++state;
        break;
      case CounterEvent.decrement:
        yield --state;
        break;
    }
  }
}
```

NotebookLM

Arayüzü BLoC’a Bağlamak: `BlocProvider` ve `BlocBuilder`

BLoC’u widget ağacına tanıtmak ve durum değişikliklerine göre UI’ı verimli bir şekilde güncellemek için flutter_bloc paketinin sunduğu widget’ları kullanırız.

```
// main.dart
return BlocProvider(
  create: (context) => CounterBloc(),
  child: const DemoPage(),
);
```

`BlocProvider`, `CounterBloc` örneğini oluşturur ve altındaki tüm widget’ların erişimine sunar. Bu, dahili olarak `provider` paketini kullanır.

```
// DemoPage.dart
onPressed: () => context.read<CounterBloc>()
  .add(CounterEvent.increment),

BlocBuilder<CounterBloc, int>(
  builder: (context, count) => Text('$count'),
),
```

Butonlar, BLoC’a olay göndererek iş mantığını tetikler.

Yeni durumları dinler ve yalnızca `builder` içindeki widget’ı yeniden oluşturarak performansı optimize eder.

NotebookLM

Daha Güçlü Olaylar ve Durumlar: `enum` Yerine `class`

Olaylar veya durumlar veri taşımaya başladığında, `enum`'lar yetersiz kalır. Sınıflar (`class`), daha karmaşık senaryolar için **esneklik, yapı ve veri taşıma kabiliyeti** sağlar. **Çoğu çevrimiçi örnek ve dokümantasyon** bu yapıyı kullanır.

Basit

```
enum CounterEvent {
  increment, decrement
}
```

Gelişmiş

```
abstract class CounterEvent extends Equatable {
  const CounterEvent();
  @override
  List<Object> get props => [];
}

class Increment extends CounterEvent {}
class Decrement extends CounterEvent {}
```

Ek Not

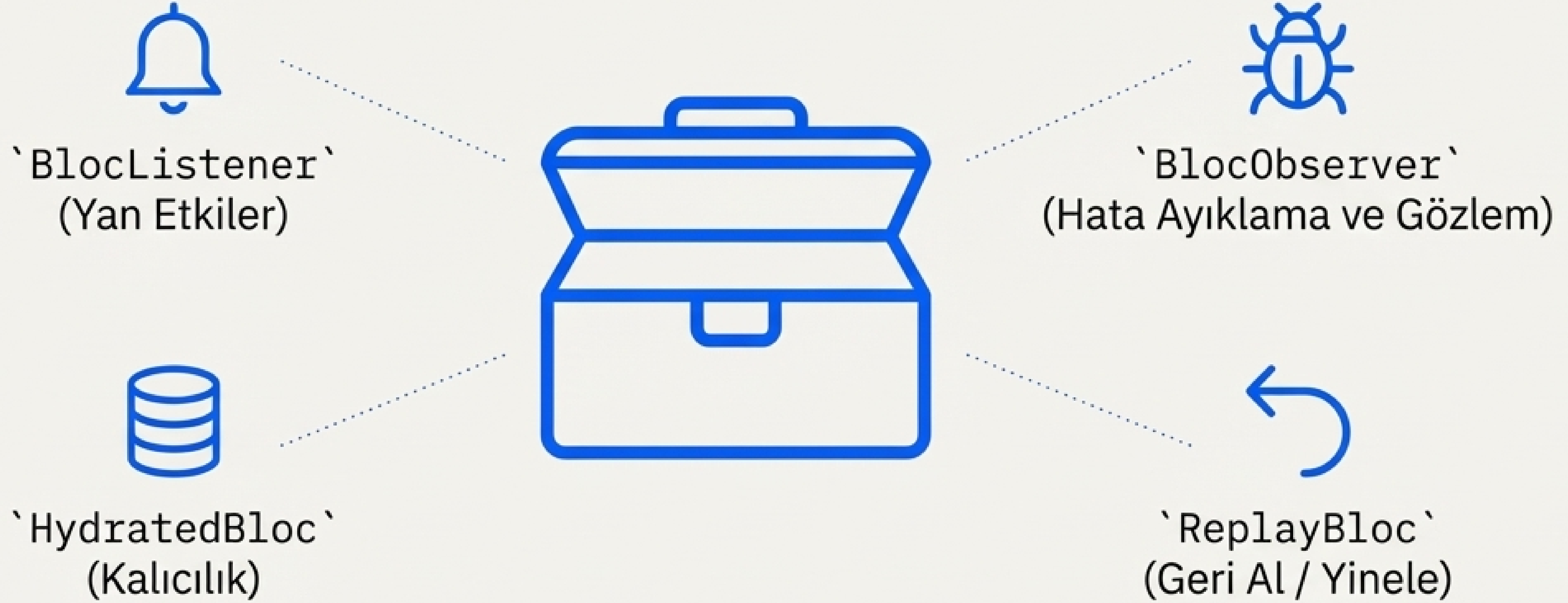
Sınıfların değişmez (immutable) olması önemlidir. `Equatable` paketi, nesne karşılaştırmaları için gereken standart kod miktarını azaltarak kolaylık sağlar.

```
// mapEventToState içinde:
if (event is Increment) { ... }
```

NotebookLM

BLoC Ekosistemi: Özel Görevler İçin Uzmanlaşmış Araçlar

BLoC, sadece bir durum yönetimi deseni değildir; aynı zamanda state değişikliklerine tepki verme, tüm akışı gözlemleme ve durumu kalıcı hale getirme gibi görevler için güçlü araçlar içeren bir kütüphanedir.



NotebookLM

Yan Etkileri Yönetmek: `BlocListener`

`BlocBuilder` arayüzü yeniden oluşturur. `BlocListener` ise arayüzü etkilemeden, durum değişikliklerine yanıt olarak bir kerelik eylemler gerçekleştirmek için kullanılır. Örneğin: `SnackBar` göstermek, yeni bir sayfaya yönlendirmek veya bir diyalog kutusu açmak.

```
BlocListener<CounterBloc, int>(
  listenWhen: (previous, current) {
    // Yalnızca sayaç 5'ten büyükse dinleyiciyi tetikle
    return (previous != current) && (current > 5);
  },
  listener: (context, state) {
    // SnackBar göster
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Sayaç 5\'i geçti!')),
    );
  },
  child: YourPageContent(),
);
```

İsteğe bağlı `listenWhen` parametresi, `listener`'ın ne zaman çağrılacağını hassas bir şekilde kontrol etmenizi sağlar.

Bu fonksiyon UI'ı yeniden çizmez, sadece bir eylem gerçekleştirir.

NotebookLM

Hassas Hata Ayıklama: `BlocObserver`

Uygulamanızdaki tüm BLoC'larda gerçekleşen her olayı, geçişi ve hatayı merkezi bir yerden gözlemleyin.
Bu, özellikle çok sayıda BLoC'un olduğu karmaşık uygulamalarda hata ayıklama için paha biçilmezdir.

Adım 1: Gözlemciyi Tanımla

```
class MyObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    print('onEvent: $event');
  }
  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print('onTransition: $transition');
  }
}
```

Adım 2: Gözlemciyi Kaydet

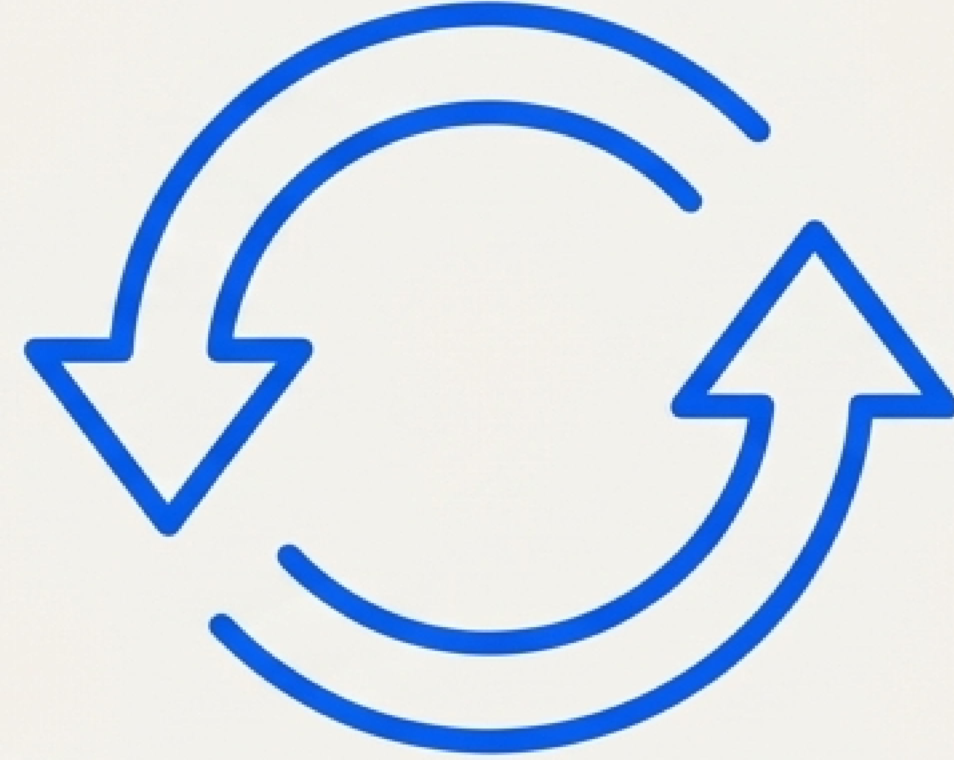
```
void main() {
  Bloc.observer = MyObserver();
  runApp(const MyApp());
}
```

NotebookLM

Gelişmiş Yetenekler: Kalıcılık ve Geçmiş Yönetimi

BLoC ekosistemi, temel durum yönetiminin ötesine geçer. Harici paketler sayesinde, uygulama kapansa bile durumu hatırlama veya kullanıcı eylemlerini geri alma gibi karmaşık özellikleri kolayca entegre edebilirsiniz.

Durumu Kaydetme: `HydratedBloc`



Geri Al / Yinele: `ReplayBloc`

NotebookLM

Durumu Hatırlamak: `HydratedBloc` ile Kalıcılık

Uygulama kapansa bile durumu otomatik olarak cihaz hafızasına kaydeder ve yeniden açıldığında geri yükler. Tema tercihleri, dil ayarları gibi kullanıcıya özel ayarlar için idealdir.

```
class ThemeBloc extends HydratedBloc<ThemeEvent, ThemeData> {
  ThemeBloc() : super(ThemeData.light());

  @override
  Stream<ThemeData> mapEventToState(ThemeEvent event) async* { /* ... */ }

  @override
  ThemeData? fromJson(Map<String, dynamic> json) {
    // Kaydedilmiş JSON'dan durumu geri yükle
    try {
      if (json['isDarkMode'] as bool) return ThemeData.dark();
      return ThemeData.light();
    } catch (_) { return null; }
  }

  @override
  Map<String, dynamic>? toJson(ThemeData state) {
    // Mevcut durumu JSON'a çevir ve kaydet
    try {
      return {'isDarkMode': state == ThemeData.dark()};
    } catch (_) { return null; }
  }
}
```

Başlatma için `main()` fonksiyonunda `HydratedBloc.storage = await HydratedStorage.build();` çağrısının yapılması gerektiğini unutmayın.

NotebookLM

Geri Al / Yinele: `ReplayBloc` ile Zahmetsizce

Kullanıcı eylemlerinin geçmişini tutarak geri alma (`undo`) ve yineleme (`redo`) işlevselliğini neredeyse hiç çaba harcamadan ekleyin.

Olay Sınıfı

```
class CounterEvent extends ReplayEvent {
  ...
  ...
}
```

Olay sınıflarınızın `Equatable` yerine `ReplayEvent`'ten türemesi yeterlidir.

BLoC Sınıfı

```
class CounterBloc extends ReplayBloc<...>
  ...
  ...
}
```

BLoC'unuz `Bloc` yerine `ReplayBloc`'tan türemelidir.

UI Kodları

```
// onPressed metotları
onPressed: () => counterBloc.add(...),
onPressed: () => counterBloc.undo(),
onPressed: () => counterBloc.redo(),
```

Artık bu iki metodu doğrudan çağırabilirsiniz.

NotebookLM

Kaputun Altında: BLoC'un Motoru `Cubit`

`Cubit`, BLoC'un temelini oluşturan daha basit bir sınıftır. Olay (Event) katmanı yoktur; bunun yerine, durumları doğrudan yayan (`emit`) fonksiyonları vardır. Basit durum yönetimi senaryoları için harika bir alternatiftir.

Özellik	BLoC	Cubit
Karmaşıklık	Daha fazla standart kod, karmaşık iş mantığı için ideal.	Daha az kod, basit durumlar için ideal.
Çalışma Prensibi	Olaylar girer -> `mapEventToState` -> Durumlar çıkar.	Fonksiyon çağrılır -> `emit(newState)` -> Durumlar çıkar.
İzlenebilirlik	Mükemmel. Olaylar, durum değişikliklerinin nedenini açıkça belirtir.	İyi. Ancak olaylar kadar açık bir geçmiş sunmaz.

Cubit (Solda)

```
class CounterCubit extends Cubit<int> {
  void increment() => emit(state + 1);
}
```

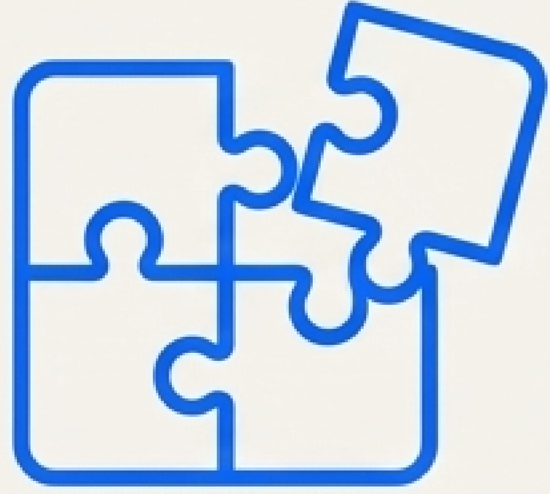
BLoC (Sağda)

```
class CounterBloc extends Bloc<...> {
  mapEventToState(...) {
    yield state + 1;
  }
}
```

NotebookLM

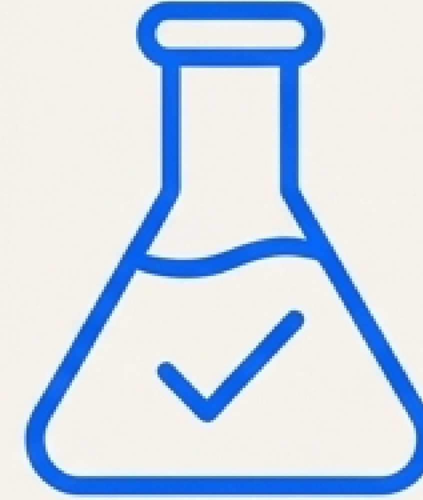
Neden BLoC Kullanmalısınız? Temel Avantajlar

BLoC, Flutter uygulamalarınız için yapılandırılmış, ölçeklenebilir ve test edilebilir bir durum yönetimi çözümü sunar.



Ayrım (Separation):

İş ve arayüz mantığını kusursuzca ayırır.



Test Edilebilirlik (Testability):

İş mantığı UI'dan bağımsız olduğu için kolayca birim testleri yazılabilir.

Ölçeklenebilirlik (Scalability):

Basit sayaçlardan karmaşık kurumsal uygulamalara kadar her projede ölçeklenir.

Güçlü Ekosistem:

Kalıcılık, geri alma/yineleme ve gelişmiş hata ayıklama için kullanıma hazır çözümler sunar.

NotebookLM

Keşfetmeye Devam Edin

BLoC kütüphanesinin resmi doküm**antasyo**nu, daha fazla örnek, ayrıntılı eğitimler ve derinlemesine bilgi için en iyi kaynaktır.

bloclibrary.dev

NotebookLM