# Flutter Bloc Dokümantasyonu

## 1. Kullanılan Paketler ve Açıklamaları

### 1.1 flutter\_bloc

* Flutter uygulamalarında state management için kullanılan temel paket
* BLoC (Business Logic Component) pattern’ini uygulamamıza olanak sağlar
* State, Event ve Bloc sınıflarını içerir
* Uygulama içindeki veri akışını yönetir
* UI ve iş mantığı arasında temiz bir ayrım sağlar
* Reactive programlama yaklaşımını destekler
* Stream tabanlı state yönetimi sunar

### 1.2 bloc\_concurrency

* Bloc event’lerinin nasıl işleneceğini kontrol etmemizi sağlayan paket
* Farklı transformer’lar sunar
* Event’lerin işlenme sırasını ve şeklini yönetir
* Performans optimizasyonu sağlar
* Race condition’ları önler
* Event’lerin önceliklendirilmesini sağlar
* Event işleme stratejilerini belirler

### 1.3 equatable

* State ve Event sınıflarının karşılaştırılmasını kolaylaştırır
* Gereksiz build işlemlerini önler
* Performansı artırır
* Value equality sağlar
* Memory kullanımını optimize eder
* State değişikliklerinin doğru tespitini sağlar

## 2. Kart Listeleme Yapısı

### 2.1 CardsBloc Sınıfı

* Kart işlemlerini yöneten ana sınıf
* CardRepository ile iletişim kurar
* Üç temel event’i yönetir:
  + LoadCards: Tüm kartları yükler
  + LoadCardBalance: Her kartın bakiyesini yükler
  + FilterCards: Kartları türlerine göre filtreler
* State yönetimini sağlar
* Hata yönetimini gerçekleştirir
* Kart işlemlerinin senkronizasyonunu sağlar
* Veri tutarlılığını korur

#### 2.1.1 Event Sınıfları

// Kart yükleme event'i  
class LoadCards extends Equatable {  
 @override  
 List<Object> get props => [];  
}  
  
// Kart bakiyesi yükleme event'i  
class LoadCardBalance extends Equatable {  
 final String cardId;  
   
 LoadCardBalance(this.cardId);  
   
 @override  
 List<Object> get props => [cardId];  
}  
  
// Kart filtreleme event'i  
class FilterCards extends Equatable {  
 final String filterType;  
   
 FilterCards(this.filterType);  
   
 @override  
 List<Object> get props => [filterType];  
}

#### 2.1.2 State Sınıfları

// Başlangıç durumu  
class CardsInitial extends CardsState {}  
  
// Yükleme durumu  
class CardsLoading extends CardsState {}  
  
// Yükleme başarılı durumu  
class CardsLoaded extends CardsState {  
 final List<Card> cards;  
   
 CardsLoaded(this.cards);  
   
 @override  
 List<Object> get props => [cards];  
}  
  
// Hata durumu  
class CardsError extends CardsState {  
 final String message;  
   
 CardsError(this.message);  
   
 @override  
 List<Object> get props => [message];  
}

### 2.2 State Yönetimi

* CardsInitial: Başlangıç durumu
* CardsLoading: Yükleme durumu
* CardsLoaded: Kartların yüklendiği durum
* CardsError: Hata durumu
* Her state için özel UI gösterimi
* State geçişlerinin yönetimi
* State değişikliklerinin izlenmesi
* State senkronizasyonu

#### 2.2.1 State Geçişleri

// State geçiş örneği  
on<LoadCards>((event, emit) async {  
 emit(CardsLoading());  
 try {  
 final cards = await \_cardRepository.getCards();  
 emit(CardsLoaded(cards));  
 } catch (e) {  
 emit(CardsError(e.toString()));  
 }  
});

## 3. Bloc Concurrency ve Transformer Özellikleri

### 3.1 sequential()

* Event’leri sırayla işler
* Bir event işlenirken diğer event’ler bekler
* Kullanım Senaryosu:
* on<LoadCardBalance>(\_onLoadCardBalance, transformer: sequential());
* Örnek Senaryo: Kart bakiyelerinin sıralı yüklenmesi
* Avantajları:
  + Sıralı işlem garantisi
  + Veri tutarlılığı
  + Kaynak kullanımının kontrolü
  + İşlem önceliklendirmesi
  + Hata yönetimi kolaylığı

#### 3.1.1 Olay Akışı

1. LoadCardBalance event’i tetiklenir
2. Önceki event’ler tamamlanana kadar bekler
3. Sıra kendisine geldiğinde işleme başlar
4. İşlem tamamlanana kadar diğer event’ler bekler
5. İşlem tamamlandığında sıradaki event’e geçilir

### 3.2 concurrent()

* Event’leri eş zamanlı olarak işler
* Tüm event’ler paralel olarak çalışır
* Kullanım Senaryosu:
* on<LoadCards>(\_onLoadCards, transformer: concurrent());
* Örnek Senaryo: Çoklu kart yükleme işlemleri
* Avantajları:
  + Hızlı işlem
  + Kaynakların etkin kullanımı
  + Performans optimizasyonu
  + Paralel işlem desteği
  + Yüksek throughput

#### 3.2.1 Olay Akışı

1. Birden fazla LoadCards event’i tetiklenir
2. Her event bağımsız olarak işlenir
3. Event’ler birbirini beklemek zorunda değildir
4. İşlemler paralel olarak yürütülür
5. Her event kendi sonucunu döndürür

### 3.3 droppable()

* Yeni bir event geldiğinde, önceki event’i iptal eder
* Kullanım Senaryosu:
* on<FilterCards>(\_onFilterCards, transformer: droppable());
* Örnek Senaryo: Hızlı filtreleme işlemleri
* Avantajları:
  + Gereksiz işlemlerin önlenmesi
  + UI yanıt süresinin iyileştirilmesi
  + Kaynak tasarrufu
  + Kullanıcı deneyiminin iyileştirilmesi
  + Performans optimizasyonu

#### 3.3.1 Olay Akışı

1. FilterCards event’i tetiklenir
2. Önceki filtreleme işlemi iptal edilir
3. Yeni filtreleme işlemi başlatılır
4. Sadece en son event’in sonucu kullanılır
5. Gereksiz işlemler önlenir

### 3.4 restartable()

* Yeni bir event geldiğinde, önceki event’i iptal eder ve yeni event’i baştan başlatır
* Kullanım Senaryosu:
* on<RefreshEvent>(\_onRefresh, transformer: restartable());
* Örnek Senaryo: Sayfa yenileme işlemleri
* Avantajları:
  + Temiz başlangıç
  + Veri tutarlılığı
  + Kullanıcı deneyiminin iyileştirilmesi
  + İşlem kontrolü
  + Hata durumlarının yönetimi

#### 3.4.1 Olay Akışı

1. RefreshEvent tetiklenir
2. Önceki yenileme işlemi iptal edilir
3. Yeni yenileme işlemi başlatılır
4. İşlem baştan başlar
5. En güncel veriler yüklenir

### 3.5 throttleTime()

* Belirli bir süre içinde gelen event’leri sınırlar
* Kullanım Senaryosu:
* on<ScrollEvent>(\_onScroll, transformer: throttleTime(Duration(milliseconds: 300)));
* Örnek Senaryo: Scroll olaylarının kontrolü
* Avantajları:
  + Performans optimizasyonu
  + Gereksiz işlemlerin önlenmesi
  + Kaynak kullanımının kontrolü
  + UI akıcılığının sağlanması
  + Kullanıcı deneyiminin iyileştirilmesi

#### 3.5.1 Olay Akışı

1. ScrollEvent tetiklenir
2. Son 300ms içinde başka event varsa işlenmez
3. 300ms geçtikten sonra yeni event işlenir
4. Event’ler belirli aralıklarla işlenir
5. Performans optimize edilir

## 4. Örnek Senaryolar

### 4.1 Kart Yükleme Senaryosu

// Kartların paralel yüklenmesi  
on<LoadCards>(\_onLoadCards, transformer: concurrent());  
  
// Bakiyelerin sıralı yüklenmesi  
on<LoadCardBalance>(\_onLoadCardBalance, transformer: sequential());

#### 4.1.1 Olay Akışı

1. Uygulama başlatılır
2. LoadCards event’i tetiklenir
3. Kartlar paralel olarak yüklenir
4. Her kart için LoadCardBalance event’i tetiklenir
5. Bakiyeler sıralı olarak yüklenir

### 4.2 Filtreleme Senaryosu

// Hızlı filtreleme işlemleri  
on<FilterCards>(\_onFilterCards, transformer: droppable());

#### 4.2.1 Olay Akışı

1. Kullanıcı filtreleme yapar
2. FilterCards event’i tetiklenir
3. Önceki filtreleme iptal edilir
4. Yeni filtreleme başlatılır
5. Sonuçlar gösterilir

### 4.3 Yenileme Senaryosu

// Sayfa yenileme işlemi  
on<RefreshEvent>(\_onRefresh, transformer: restartable());

#### 4.3.1 Olay Akışı

1. Kullanıcı sayfayı yeniler
2. RefreshEvent tetiklenir
3. Önceki yenileme iptal edilir
4. Yeni yenileme başlatılır
5. Veriler güncellenir

### 4.4 Scroll Senaryosu

// Scroll olaylarının kontrolü  
on<ScrollEvent>(\_onScroll, transformer: throttleTime(Duration(milliseconds: 300)));

#### 4.4.1 Olay Akışı

1. Kullanıcı scroll yapar
2. ScrollEvent tetiklenir
3. Son 300ms içinde event varsa işlenmez
4. 300ms geçtikten sonra işlenir
5. UI akıcı çalışır

## 5. Best Practices

### 5.1 Transformer Seçimi

* Sıralı işlemler için: sequential()
* Paralel işlemler için: concurrent()
* UI etkileşimleri için: droppable()
* Yenileme işlemleri için: restartable()
* Hızlı olaylar için: throttleTime()
* Her senaryoya uygun transformer seçimi
* Performans ve kullanıcı deneyimi dengesi

### 5.2 Performans İyileştirmeleri

* Gereksiz state güncellemelerinden kaçının
* Uygun transformer’ları seçin
* Büyük veri setleri için pagination kullanın
* Hata yönetimini doğru şekilde yapın
* Memory leak’leri önleyin
* Dispose işlemlerini düzgün yapın
* Cache mekanizması kullanın
* Lazy loading uygulayın

### 5.3 Kod Organizasyonu

* Event’leri mantıksal gruplara ayırın
* State’leri basit ve anlaşılır tutun
* Repository pattern’i kullanın
* Dependency injection uygulayın
* Clean Architecture prensiplerini uygulayın
* SOLID prensiplerine uyun
* Modüler yapı oluşturun
* Kod tekrarından kaçının

### 5.4 Test Edilebilirlik

* Unit testler yazın
* Widget testleri ekleyin
* Integration testleri yapın
* Mock kullanımını öğrenin
* Test coverage’ı artırın
* Test senaryolarını belgeleyin
* CI/CD pipeline’ına testleri ekleyin
* Test otomasyonu sağlayın

### 5.5 Hata Yönetimi

* Try-catch bloklarını doğru kullanın
* Hata mesajlarını kullanıcı dostu yapın
* Loglama mekanizması kurun
* Crash reporting ekleyin
* Offline durumları yönetin
* Retry mekanizması ekleyin
* Error boundary’ler kullanın
* Hata izleme ve analiz yapın

### 5.6 Güvenlik

* API anahtarlarını güvenli şekilde saklayın
* Kullanıcı verilerini şifreleyin
* SSL/TLS kullanın
* Input validasyonu yapın
* XSS ve CSRF koruması ekleyin
* Güvenlik testleri yapın
* Güvenlik güncellemelerini takip edin
* Güvenlik politikaları belirleyin

### 5.7 Monitoring ve Logging

* Performans metriklerini izleyin
* Hata loglarını tutun
* Kullanıcı davranışlarını analiz edin
* Sistem sağlığını kontrol edin
* Alert mekanizması kurun
* Log rotasyonu yapın
* Log seviyelerini belirleyin
* Monitoring dashboard’ları oluşturun

## 6. Profil Özelliği Geliştirmeleri

### 6.1 API Response Yapısı

@freezed  
class ApiResponse<T> with \_$ApiResponse<T> {  
 const factory ApiResponse.success(T data) = ApiResponseSuccess<T>;  
 const factory ApiResponse.error(String message, {String? type}) = ApiResponseError<T>;  
 const factory ApiResponse.noContent() = ApiResponseNoContent<T>;  
}

* Freezed kullanarak immutable API yanıt yapısı oluşturduk
* Üç farklı durum tanımladık: success, error ve noContent
* Generic yapı sayesinde farklı veri tipleriyle kullanılabilir
* Pattern matching ile kolay kullanım sağlar
* Type-safe yapı sayesinde hata riskini azaltır

### 6.2 Base State Yapısı

abstract class BaseState<T> extends Equatable {  
 final T? \_data;  
 final String? \_message;  
 final bool \_isLoading;  
  
 const BaseState({  
 T? data,  
 String? message,  
 bool isLoading = false,  
 });  
}

* Tüm state’ler için temel yapı sağlar
* Equatable ile gereksiz build’leri önler
* Generic yapı ile farklı veri tipleri destekler
* Loading, error ve data durumlarını yönetir
* Immutable yapı ile state değişikliklerini kontrol eder

### 6.3 Repository Mixin

mixin BaseRepositoryMixin {  
 T? parseResponse<T>(ApiResponse<T> response) {  
 return response.maybeWhen(  
 success: (data) => data,  
 orElse: () => null,  
 );  
 }  
  
 ApiResponse<T> handleResponse<T>({  
 required T? Function() parseData,  
 String? errorMessage,  
 String? errorType,  
 }) {  
 try {  
 final data = parseData();  
 if (data == null || data == {}) {  
 return const ApiResponse.noContent();  
 }  
 return ApiResponse.success(data);  
 } catch (e) {  
 return ApiResponse.error(errorMessage ?? 'Bir hata oluştu', type: errorType);  
 }  
 }  
}

* Repository katmanı için ortak fonksiyonlar sağlar
* API yanıtlarını güvenli şekilde parse eder
* Hata yönetimini merkezi hale getirir
* Kod tekrarını önler
* Tutarlı hata mesajları sağlar

### 6.4 API Call Mixin

mixin HandleApiCallMixin {  
 Future<void> handleApiCall<T>({  
 required Future<ApiResponse<T>> Function() apiCall,  
 required void Function(BaseState<T>) emitState,  
 }) async {  
 emitState(const LoadingState());  
 final response = await apiCall();  
 response.when(  
 success: (data) => emitState(LoadedState(data)),  
 error: (message, type) => emitState(ErrorState(message)),  
 noContent: () => emitState(const NoContentState()),  
 );  
 }  
}

* Bloc’larda API çağrılarını standartlaştırır
* Loading, success, error ve noContent durumlarını yönetir
* State güncellemelerini otomatikleştirir
* Hata yönetimini merkezi hale getirir
* Kod tekrarını önler
* Bakımı kolaylaştırdık

### 6.5 Profil Bloc Yapısı

class ProfileBloc extends Bloc<ProfileEvent, ProfileState> with HandleApiCallMixin {  
 final ProfileRepository \_repository;  
  
 ProfileBloc(this.\_repository) : super(const ProfileState(  
 profileState: InitialState(),  
 detailsState: InitialState(),  
 )) {  
 on<LoadProfileInfo>(\_onLoadProfileInfo);  
 on<LoadProfileDetails>(\_onLoadProfileDetails);  
 }  
}

* Repository pattern ile veri erişimini soyutlar
* HandleApiCallMixin ile API çağrılarını yönetir
* Event’leri ayrı sınıflarda tanımlar
* State’leri immutable yapıda tutar
* Dependency injection ile test edilebilirliği artırır

### 6.6 Profil Sayfası

class ProfilePage extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return BlocBuilder<ProfileBloc, ProfileState>(  
 buildWhen: (previous, current) => previous.profileState != current.profileState,  
 builder: (context, state) {  
 // State'e göre UI güncelleme  
 },  
 );  
 }  
}

* BlocBuilder ile state değişikliklerini izler
* buildWhen ile gereksiz build’leri önler
* State’e göre farklı UI gösterimi sağlar
* Loading, error ve success durumlarını yönetir
* Kullanıcı deneyimini iyileştirir

### 6.7 Neden Bu Yapıyı Kullandık?

#### 6.7.1 API Response

* Type-safe yapı ile hata riskini azalttık
* Pattern matching ile kolay kullanım sağladık
* Immutable yapı ile state değişikliklerini kontrol ettik
* Generic yapı ile kod tekrarını önledik
* Freezed ile boilerplate kodları azalttık

#### 6.7.2 Base State

* Tüm state’ler için ortak yapı sağladık
* Equatable ile performansı artırdık
* Immutable yapı ile state yönetimini kolaylaştırdık
* Generic yapı ile esneklik sağladık
* Kod tekrarını önledik

#### 6.7.3 Repository Mixin

* Merkezi hata yönetimi sağladık
* Kod tekrarını önledik
* Tutarlı API yanıtları sağladık
* Test edilebilirliği artırdık
* Bakımı kolaylaştırdık

#### 6.7.4 API Call Mixin

* Standart API çağrı yapısı oluşturduk
* State yönetimini otomatikleştirdik
* Hata yönetimini merkezi hale getirdik
* Kod tekrarını önledik
* Bakımı kolaylaştırdık

#### 6.7.5 Bloc Yapısı

* Clean Architecture prensiplerini uyguladık
* Test edilebilirliği artırdık
* Kod organizasyonunu iyileştirdik
* State yönetimini kolaylaştırdık
* Dependency injection ile esneklik sağladık

#### 6.7.6 UI Yapısı

* Reactive programlama yaklaşımı kullandık
* Performansı optimize ettik
* Kullanıcı deneyimini iyileştirdik
* State yönetimini kolaylaştırdık
* Kod organizasyonunu iyileştirdik